

Артем Груздев



	CUSTOMER LIFETIME VALUE	COVERAGE	EDUCATION	EMPLOYMENT STATUS	GENDER	INCOME	MONTHLY PREMIUM AUTO
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN
2	NaN	NaN	NaN	Employed	F	48767.0	108.0
3	7645.861827	Basic	Bachelor	NaN	NaN	0.0	106.0
4	2813.692575	Basic	Bachelor	NaN	M	43836.0	73.0

# Предварительная подготовка данных в Python Том 1. Инструменты и валидация

В двухтомнике представлены материалы по применению классических методов машинного обучения для различных промышленных задач.

### Прочитав первый том, вы научитесь:

- работать в IPython и Jupyter Notebook;
- применять функции библиотеки NumPy;
- визуализировать результаты анализа с помощью библиотек matplotlib, seaborn и plotly;
- выполнять предварительную подготовку данных в библиотеке pandas;
- работать с классами scikit-learn, строящими модели предварительной подготовки данных и модели машинного обучения;
- применять различные стратегии валидации данных.



**Артем Груздев** – основатель и директор компании «Гевисста», имеет 10-летний опыт прогнозирования кредитных рисков и 18-летний опыт статистического анализа. В последние годы активно занимается практическим построением прогнозных моделей.

Ведет Telegram-канал, посвященный машинному обучению <https://t.me/Gewissta>.

**ИЦ «Гевисста»**



Исследовательский центр «Гевисста» с 2009 г. осуществляет разработку, валидацию, внедрение и мониторинг риск-моделей, моделей оттока, моделей отклика на базе IBM SPSS Statistics, IBM SPSS Modeler, SAS Enterprise Miner, SAS Enterprise Guide, R, Python. Осуществляет подготовку специалистов в сфере прогнозного моделирования и анализа данных.

ISBN 978-5-93700-156-6



9 785937 001566 >

Интернет-магазин:  
[www.dmkpress.com](http://www.dmkpress.com)  
Оптовая продажа:  
КТК «Галактика»  
[books@alians-kniga.ru](mailto:books@alians-kniga.ru)

  
ИЗДАТЕЛЬСТВО  
[www.dmk.pф](http://www.dmk.pф)

А. В. Груздев

# Предварительная подготовка данных в Python

Том 1

Инструменты и валидация



Москва, 2023

УДК 004.04Python

ББК 32.372

Г90

**Груздев А. В.**

**Г90** Предварительная подготовка данных в Python: Том 1. Инструменты и валидация. – М.: ДМК Пресс, 2023. – 816 с.: ил.

**ISBN 978-5-93700-156-6**

В двухтомнике представлены материалы по применению классических методов машинного обучения в различных промышленных задачах. Первый том посвящен инструментам Python – основным библиотекам, классам и функциям, необходимым для предварительной подготовки данных, построения моделей машинного обучения, выполнения различных стратегий валидации. В конце первого тома разбираются задачи с собеседований по SQL, Python, математической статистике и теории вероятностей.

Издание рассчитано на специалистов по анализу данных, а также может быть полезно широкому кругу специалистов, интересующихся машинным обучением.

УДК 004.04Python

ББК 32.372

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

*Моей Матильде, прошедшей  
путь бескорыстной любви  
длиной в 22 года, посвящается*

# Оглавление

<b>Введение .....</b>	<b>10</b>
<b>ЧАСТЬ 1. НЕМНОГО МАТЕМАТИКИ .....</b>	<b>11</b>
1.1. Функция .....	11
1.2. Производная .....	12
1.3. Дифференцирование сложных функций .....	15
1.4. Частная производная .....	16
1.5. Градиент .....	17
1.6. Функция потерь и градиентный спуск .....	18
<b>Часть 2. Инструменты .....</b>	<b>23</b>
<b>1. Введение .....</b>	<b>23</b>
1.1. Структуры данных .....	23
1.1.1. Кортеж (tuple) .....	23
1.1.2. Список (list) .....	24
1.1.3. Словарь (dictionary) .....	27
1.1.4. Множество (set) .....	31
1.2. Функция .....	34
1.3. Полезные встроенные функции .....	35
1.3.1. Функция enumerate() .....	35
1.3.2. Функция sorted() .....	36
1.3.3. Функция zip() .....	36
1.4. Класс .....	38
1.5. Знакомство с Anaconda .....	43
<b>2. IPython и Jupyter Notebook .....</b>	<b>44</b>
<b>3. NumPy .....</b>	<b>50</b>
3.1. Создание массивов NumPy .....	50
3.2. Обращение к элементам массива .....	55
3.3. Получение краткой информации о массиве .....	57
3.4. Изменение формы массива .....	58
3.5. Конкатенация массивов .....	61
3.6. Функции математических операций, знакомство с правилами транслирования .....	65
3.7. Обработка пропусков .....	70
3.8. Функция np.linspace() .....	72
3.9. Функция np.logspace() .....	74

3.10. Функция <code>np.digitize()</code> .....	75
3.11. Функция <code>np.searchsorted()</code> .....	76
3.12. Функция <code>np.bincount()</code> .....	78
3.13. Функция <code>np.apply_along_axis()</code> .....	79
3.14. Функция <code>np.insert()</code> .....	80
3.15. Функция <code>np.repeat()</code> .....	81
3.16. Функция <code>np.unique()</code> .....	82
3.17. Функция <code>np.take_along_axis()</code> .....	84
3.18. Функция <code>np.array_split()</code> .....	86

## 4. Библиотеки Numba, datatable, bottleneck

<b>для ускорения вычислений.....</b>	<b>88</b>
4.1. Numba .....	88
4.2. Datatable .....	94
4.3. Bottleneck.....	98

## 5. SciPy ..... 99

## 6. pandas ..... 111

6.1. Почему pandas? .....	111
6.2. Библиотека pandas построена на NumPy .....	111
6.3. pandas работает с табличными данными .....	111
6.4. Объекты DataFrame и Series .....	111
6.5. Задачи, выполняемые pandas.....	113
6.6. Кратко о типах данных .....	113
6.7. Представление пропусков .....	114
6.8. Какую версию pandas использовать? .....	115
6.9. Подробно знакомимся с типами данных .....	115
6.9.1. Типы данных для работы с числами и логическими значениями .....	115
6.9.2. Типы данных для работы со строками .....	126
6.10. Чтение данных .....	136
6.11. Получение общей информации о датафрейме .....	137
6.12. Изменение настроек вывода с помощью функции <code>get_options()</code> .....	139
6.13. Знакомство с индексаторами <code>[], loc</code> и <code>iloc</code> .....	140
6.14. Фильтрация данных .....	147
6.14.1. Одно условие .....	147
6.14.2. Несколько условий .....	148
6.14.3. Несколько условий в одном столбце.....	148
6.14.4. Использование метода <code>.query()</code> .....	149
6.15. Агрегирование данных .....	151
6.15.1. Группировка и агрегирование с помощью одного столбца .....	151
6.15.2. Группировка и агрегирование с помощью нескольких столбцов .....	153
6.15.3. Группировка с помощью сводных таблиц.....	156
6.16. Анализ частот с помощью таблиц сопряженности .....	166
6.17. Выполнение SQL-запросов в pandas .....	169

<b>7. scikit-learn</b> .....	<b>179</b>
7.1. Основы работы с классами, строящими модели предварительной подготовки данных и модели машинного обучения .....	179
7.2. Строим свой первый конвейер моделей .....	198
7.3. Разбираемся с дилеммой смещения–дисперсии и знакомимся с бутстрепом .....	210
7.4. Обработка пропусков с помощью классов MissingIndicator и SimpleImputer .....	228
7.5. Выполнение дамми-кодирования с помощью класса OneHotEncoder и функции get_dummies(), знакомство с разреженными матрицами.....	235
7.6. Автоматическое построение конвейеров моделей с помощью класса Pipeline.....	246
7.7. Знакомство с классом ColumnTransformer .....	250
7.8. Класс FeatureUnion .....	263
7.9. Выполнение перекрестной проверки с помощью функции cross_val_score(), получение прогнозов перекрестной проверки с помощью функции cross_val_predict(), сохранение моделей перекрестной проверки с помощью функции cross_validate().....	264
7.10. Виды перекрестной проверки для данных формата «один объект – одно наблюдение» (отсутствует ось времени) .....	273
7.10.1. Обычная нестратифицированная $k$ -блочная перекрестная проверка с помощью класса KFold .....	274
7.10.2. Обычная стратифицированная $k$ -блочная перекрестная проверка с помощью класса StratifiedKFold.....	281
7.10.3. Повторная нестратифицированная $k$ -блочная перекрестная проверка с помощью класса RepeatedKFold .....	283
7.10.4. Повторная стратифицированная $k$ -блочная перекрестная проверка с помощью класса RepeatedStratifiedKFold .....	286
7.10.5. $k$ -кратное случайное разбиение на обучающую и тестовую выборки (перекрестная проверка Монте-Карло) .....	288
7.10.6. Перекрестная проверка со случайными перестановками при разбиении с помощью класса ShuffleSplit .....	294
7.10.7. Стратифицированная перекрестная проверка со случайными перестановками при разбиении с помощью класса StratifiedShuffleSplit .....	296
7.10.8. Перекрестная проверка с исключением по одному с помощью класса LeaveOneOut .....	297
7.10.9. Перекрестная проверка с исключением $p$ наблюдений с помощью класса LeavePOut.....	299
7.11. Виды перекрестной проверки для данных формата «один объект – несколько наблюдений» и стратифицированных данных (отсутствует ось времени) .....	301
7.11.1. Перекрестная проверка, учитывающая группы связанных наблюдений, с помощью классов GroupKFold .....	301
7.11.2. Перекрестная проверка, учитывающая группы связанных наблюдений с исключением из обучения одной группы, с помощью класса LeaveOneGroupOut .....	302

7.11.3. Перекрестная проверка, учитывающая группы связанных наблюдений с исключением из обучения $p$ групп, с помощью класса <code>LeavePGroupsOut</code> .....	304
7.11.4. Перекрестная проверка, учитывающая группы связанных наблюдений и распределение классов, с помощью класса <code>StratifiedGroupKFold</code> .....	305
7.11.5. Перекрестная проверка со случайными перестановками при разбиении и учитывающая группы связанных наблюдений с помощью класса <code>GroupShuffleSplit</code> .....	307
7.12. Обычный и случайный поиск наилучших гиперпараметров по сетке с помощью классов <code>GridSearchCV</code> и <code>RandomizedSearchCV</code> .....	309
7.12.1. Обычный поиск оптимальных значений гиперпараметров моделей предварительной подготовки и модели машинного обучения .....	312
7.12.2. Обычный поиск оптимальных значений гиперпараметров моделей предварительной подготовки и модели машинного обучения с добавлением строки прогресса .....	318
7.12.3. Случайный поиск оптимальных значений гиперпараметров моделей предварительной подготовки и модели машинного обучения .....	320
7.12.4. Обычный поиск оптимальных значений гиперпараметров для <code>CatBoost</code> при обработке категориальных признаков «как есть» (заданы индексы категориальных признаков) .....	321
7.12.5. Отбор оптимальной модели предварительной подготовки данных в рамках отдельного трансформера .....	324
7.12.6. Отбор оптимального метода машинного обучения среди разных методов машинного обучения (перебор значений гиперпараметров с отдельной предобработкой данных под каждый метод машинного обучения) .....	329
7.13. Вложенная перекрестная проверка .....	335
7.14. Классы <code>PowerTransformer</code> , <code>KBinsDiscretizer</code> и <code>FunctionTransformer</code> .....	341
7.15. Написание собственных классов предварительной подготовки для применения в конвейере .....	350
7.16. Модификация классов библиотеки <code>scikit-learn</code> для работы с датафреймами .....	375
7.17. Полный цикл построения конвейера моделей в <code>scikit-learn</code> .....	381
7.17.1. Первая задача .....	381
7.17.2. Вторая задача .....	393
7.18. Калибровка модели .....	404
7.18.1. Актуальность калибровки .....	404
7.18.2. Функция <code>calibration_curve()</code> .....	406
7.18.3. Оценка Брайера .....	413
7.18.4. Оценка качества калибровки моделей до применения калибратора .....	415
7.18.5. Класс <code>CalibratedClassifierCV</code> .....	420
7.18.6. Оценка качества калибровки моделей после применения калибратора .....	421

7.18.7. Оценка качества калибровки моделей после применения калибратора с уже обученным классификатором.....	423
7.18.8. Калибровка на основе сплайнов.....	426
7.19. Полезные классы CountVectorizer и TfidfVectorizer для работы с текстом.....	436
7.20. Сравнение моделей, полученных в ходе поиска по сетке, с помощью статистических тестов.....	450
7.20.1. Простое сравнение всех построенных моделей.....	451
7.20.2. Сравнение двух моделей: частотный подход.....	454
7.20.3. Сравнение двух моделей: байесовский подход.....	458
7.20.4. Парное сравнение всех моделей: частотный подход.....	463
7.20.5. Парное сравнение всех моделей: байесовский подход.....	465
7.20.6. Итоговые выводы.....	467
7.21. Разбиение на обучающую, проверочную и тестовую выборки с учетом временной структуры для валидации временных рядов.....	468
7.22. Виды перекрестной проверки для данных формата «один объект – одно наблюдение» (присутствует ось времени).....	521
7.22.1. Перекрестная проверка расширяющимся окном.....	525
7.22.2. Перекрестная проверка скользящим окном.....	542
7.22.3. Перекрестная проверка расширяющимся/скользящим окном с гэпом.....	552
7.23. Перекрестная проверка для данных формата «один объект – несколько наблюдений» (присутствует ось времени).....	563
7.24. Многоклассовая классификация: подходы «один против всех», «один против одного» и «коды, исправляющие ошибки».....	567
7.24.1. Подход «один против остальных» или «один против всех» («one versus rest», «one versus all»).....	568
7.24.2. Подход «один против одного» («one versus one»).....	573
7.24.3. Подход «коды, исправляющие ошибки» («error-correcting output codes»).....	592

## **ЧАСТЬ 3. ДРУГИЕ ПОЛЕЗНЫЕ БИБЛИОТЕКИ..... 602**

### **1. Библиотеки визуализации matplotlib, seaborn и plotly..... 602**

1.1. Matplotlib.....	602
1.2. Seaborn.....	621
1.3. Plotly.....	629

### **2. Библиотека прогнозирования временных рядов ETNA..... 634**

2.1. Общее знакомство.....	634
2.2. Создание объекта TSDataset.....	641
2.3. Визуализация рядов объекта TSDataset.....	645
2.4. Получение сводки характеристик по объекту TSDataset.....	646
2.5. Модель наивного прогноза.....	647
2.6. Модель скользящего среднего.....	654

---

2.7. Модель сезонного скользящего среднего .....	658
2.8. Модель SARIMAX.....	662
2.9. Модель Хольта–Винтерса (модель тройного экспоненциального сглаживания, модель ETS).....	671
2.10. Модель Prophet .....	677
2.11. Модель CatBoost .....	689
2.12. Модель линейной регрессии с регуляризацией «эластичная сеть»....	709
2.13. Объединение процедуры построения модели, оценки качества и визуализации прогнозов в одной функции .....	714
2.14. Перекрестная проверка нескольких моделей .....	717
2.15. Ансамбли .....	722
2.16. Стекинг .....	724
2.17. Создание собственных классов для обучения моделей.....	725
2.18. Импутация пропусков .....	741
2.19. Работа с трендом и сезонностью .....	751
2.20. Обработка выбросов .....	766
2.21. Собираем все вместе .....	772
2.22. Модели нейронных сетей .....	787
2.23. Оптимизация гиперпараметров с помощью Ortuna от разработчиков .....	789
<b>Ответы на вопросы с собеседований.....</b>	<b>794</b>

# Введение

Настоящая книга является коллекцией избранных материалов из первого модуля Подписки – обновляемых в режиме реального времени материалов по применению классических методов машинного обучения в различных промышленных задачах, которые автор делает вместе с коллегами и учениками.

Автор благодарит Игоря Яковлева за предоставленные материалы к первой части, Антона Вахрушева за помощь в подготовке раздела, посвященного NumPy, во второй части книги, Теда Петру за помощь в подготовке раздела, посвященного pandas, во второй части книги.

Первая и вторая части книги содержат несложные вопросы с собеседований по SQL, Python, математической статистике и теории вероятностей. Автором не ставится задача закрыть пробелы соискателей в этих областях, вопросы даны как напоминание, что помимо машинного обучения потребуются знания и в некоторых других сферах. В конце книги вы найдете ответы к вопросам.

В первом томе мы сконцентрируемся на инструментах предварительной обработки данных и рассмотрим различные способы валидации модели.

## Немного математики

### 1.1. Функция

В жизни мы часто встречаемся с зависимостями между различными величинами. Для простоты возьмем зависимость между двумя величинами. Например, мы знаем, что площадь круга зависит от радиуса, площадь квадрата зависит от его стороны. Переменную  $x$ , значения которой выбираются произвольно, называют независимой переменной, а переменную  $y$ , значения которой определяются выбранными значениями  $x$ , называют зависимой переменной. Давайте взглянем на рисунок. На нем изображен график температуры воздуха в течение суток.

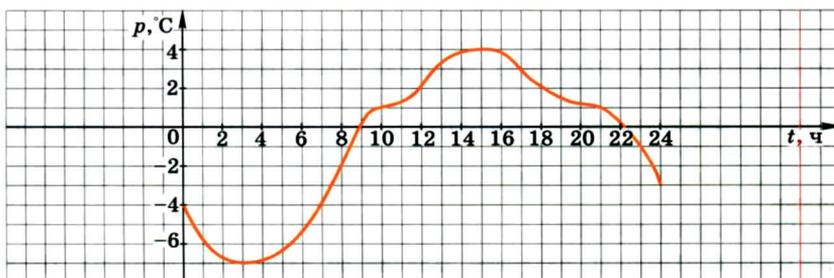


Рис. 1 График температуры воздуха в течение суток

С помощью этого графика для каждого момента времени  $t$  (в часах), где  $0 \leq t \leq 24$ , можно найти соответствующую температуру  $p$  (в градусах Цельсия). Например:

- если  $t = 7$ , то  $p = -4$ ;
- если  $t = 12$ , то  $p = 2$ ;
- если  $t = 17$ , то  $p = 3$ ;
- если  $t = 22$ , то  $p = 0$ .

Здесь  $t$  является независимой переменной, а  $p$  – зависимой переменной. В рассмотренном примере каждому значению независимой переменной соответствует единственное значение зависимой переменной. Такую зависимость одной переменной от другой называют *функциональной зависимостью*, или *функцией*. Например, когда мы пишем  $y = f(x)$  (читается «как  $y$ , равное  $f$  от  $x$ »),

мы как раз и имеем в виду эту идею зависимости: переменная  $y$  зависит от переменной  $x$  по определенному закону (предписанию, правилу). Закон этот обозначен буквой  $f$ .

Независимую переменную иначе называют *аргументом*, а о зависимой переменной говорят, что она является *функцией* от этого аргумента. Путь, пройденный автомобилем с постоянной скоростью, является функцией от времени движения. Например, если автомобиль движется с постоянной скоростью 60 км/ч, зависимость пути от времени можно задать формулой  $s = 60t$ , где  $s$  – пройденный путь (в километрах),  $t$  – время (в часах).

Значения зависимой переменной называют *значениями функции*. Все значения, которые принимает независимая переменная, образуют *область определения функции*. Значения функции в точке максимума (минимума) функции называются, соответственно, *максимумом* и *минимумом* функции. Минимальное или максимальное значения функции на заданном множестве называют экстремумом функции. Минимум и максимум функции может быть *локальным* и *глобальным*.



Рис. 2 Минимум и максимум функции

## 1.2. Производная

Теперь выясним, что такое *производная* функции. Для объяснения производной воспользуемся открытыми материалами Игоря Яковлева<sup>1</sup>.

Представьте, вы едете на автомобиле и спидометр показывает 60 км/ч. Что это значит? Ответ простой: если автомобиль будет ехать так в течение часа, то он проедет 60 км.

Допустим, что автомобиль вовсе не собирается ехать так целый час. Например, водитель разгоняет автомобиль с места, давит на газ, в какой-то момент бросает взгляд на спидометр и видит стрелку на отметке 60 км/ч. В следующий момент стрелка уползет еще выше. Как же понимать, что в данный момент времени скорость равна 60 км/ч?

Давайте выясним это на примере. Предположим, что путь  $s$ , пройденный автомобилем, зависит от времени  $t$  следующим образом:

$$s(t) = t^2,$$

<sup>1</sup> <https://mathus.ru/math/der.pdf>.

где путь измеряется в метрах, а время – в секундах. То есть при  $t = 0$  путь равен нулю, к моменту времени  $t = 1$  пройденный путь равен  $s(1) = 1$ , к моменту времени  $t = 2$  пройденный путь равен  $s(2) = 4$ , к моменту времени  $t = 3$  пройденный путь равен  $s(3) = 9$  и так далее.

Видно, что идет разгон – автомобиль набирает скорость с течением времени. Действительно: за первую секунду пройдено расстояние 1; за вторую секунду пройдено расстояние  $s(2) - s(1) = 4 - 1 = 3$ ; за третью секунду пройдено расстояние  $s(3) - s(2) = 9 - 4 = 5$ , и далее по нарастающей.

А теперь вопрос. Пусть, например, через три секунды после начала движения наш водитель взглянул на спидометр. Что покажет стрелка? Иными словами, какова *мгновенная* скорость автомобиля в момент времени  $t = 3$ ?

Просто поделить путь на время не получится: привычная формула  $v = s/t$  работает только для *равномерного* движения (то есть когда стрелка спидометра застыла в некотором фиксированном положении). Но именно эта формула лежит в основе способа, позволяющего найти мгновенную скорость.

Идея способа такова. Отсчитаем от нашего момента  $t = 3$  небольшой промежуток времени  $\Delta t$ , найдем путь  $\Delta s$ , пройденный автомобилем за этот промежуток, и поделим  $\Delta s$  на  $\Delta t$ . Чем меньше будет  $\Delta t$ , тем точнее мы приблизимся к искомой величине мгновенной скорости.

Давайте посмотрим, как эта идея реализуется. Возьмем для начала  $\Delta t = 1$ . Тогда

$$\Delta s = s(4) - s(3) = 4^2 - 3^2 = 16 - 9 = 7,$$

и для скорости (измеряется в м/с) получаем:

$$\frac{\Delta s}{\Delta t} = \frac{7}{1} = 7. \quad (1)$$

Будем уменьшать промежуток  $\Delta t$ . Берем  $\Delta t = 0,1$ :

$$\begin{aligned} \Delta s &= s(3,1) - s(3) = 3,1^2 - 3^2 = 9,61 - 9 = 0,61, \\ \frac{\Delta s}{\Delta t} &= \frac{0,61}{0,1} = 6,1. \end{aligned} \quad (2)$$

Теперь берем  $\Delta t = 0,01$ :

$$\begin{aligned} \Delta s &= s(3,01) - s(3) = 3,01^2 - 3^2 = 9,0601 - 9 = 0,0601, \\ \frac{\Delta s}{\Delta t} &= \frac{0,0601}{0,01} = 6,01. \end{aligned} \quad (3)$$

Наконец, возьмем  $\Delta t = 0,001$ :

$$\begin{aligned} \Delta s &= s(3,001) - s(3) = 3,001^2 - 3^2 = 9,006001 - 9 = 0,006001, \\ \frac{\Delta s}{\Delta t} &= \frac{0,006001}{0,001} = 6,001. \end{aligned} \quad (4)$$

Глядя на значения (1)–(4), мы понимаем, что величина  $\Delta s/\Delta t$  приближается к числу 6. Это обозначает, что мгновенная скорость автомобиля в момент времени  $t = 3$  составляет 6 м/с.

Таким образом, при безграничном уменьшении  $\Delta t$  путь  $\Delta s$  также стремится к нулю, но отношение  $\Delta s/\Delta t$  стремится к некоторому пределу  $v$ , который и называется мгновенной скоростью в данный момент времени  $t$ :

$$v = \lim_{\Delta t \rightarrow 0} \frac{\Delta s}{\Delta t}. \quad (5)$$

Можно написать и так:

$$v(t) = \lim_{\Delta t \rightarrow 0} \frac{s(t + \Delta t) - s(t)}{\Delta t}. \quad (6)$$

Давайте вернемся к нашему примеру с  $s(t) = t^2$  и проделаем в общем виде те выкладки, которые выше были выполнены с числами. Итак:

$$\Delta s = s(t + \Delta t) - s(t) = (t + \Delta t)^2 - t^2 = t^2 + 2t\Delta t + \Delta t^2 - t^2 = \Delta t(2t + \Delta t),$$

и для мгновенной скорости имеем:

$$v(t) = \lim_{\Delta t \rightarrow 0} \frac{\Delta s}{\Delta t} = \lim_{\Delta t \rightarrow 0} \frac{\Delta t(2t + \Delta t)}{\Delta t} = \lim_{\Delta t \rightarrow 0} (2t + \Delta t) = 2t. \quad (7)$$

В частности, при  $t = 3$  формула (7) дает:  $v(3) = 2 \times 3 = 6$ , как и было получено выше. Скорость бывает не только у автомобиля. Мы можем говорить о скорости изменения чего угодно – например, физической величины или экономического показателя. И производная как раз и служит обобщением понятия мгновенной скорости на случай абстрактных математических функций.

Рассмотрим функцию  $y = f(x)$ . Напомним, что  $x$  называется аргументом данной функции. Отметим на оси  $X$  некоторое значение аргумента  $x$ , а на оси  $Y$  – соответствующее значение функции  $f(x)$ .

Дадим аргументу  $x$  некоторое *приращение*, обозначаемое  $\Delta x$ . Попадём в точку  $x + \Delta x$ . Обозначим ее на рисунке вместе с соответствующим значением функции  $f(x + \Delta x)$ . Величина  $f(x + \Delta x) - f(x)$  называется *приращением функции*, которое отвечает данному приращению аргумента  $\Delta x$ .

Видите сходство с примером, когда мы вычисляли мгновенную скорость автомобиля? Приращение аргумента  $\Delta x$  есть абстрактный аналог промежутка времени  $\Delta t$ , а соответствующее приращение функции  $\Delta f$  – это аналог пути  $\Delta s$ , пройденного за время  $\Delta t$ . Производная – это в точности аналог мгновенной скорости.

Давайте дадим строгое определение производной. Производная  $f'(x)$  функции  $f(x)$  в точке  $x$  – это предел отношения приращения функции к приращению аргумента, когда приращение аргумента стремится к нулю:

$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{\Delta f}{\Delta x} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

Сравните с формулами (5) и (6). По сути, написано одно и то же. Можно сказать, что производная – это мгновенная скорость изменения функции.

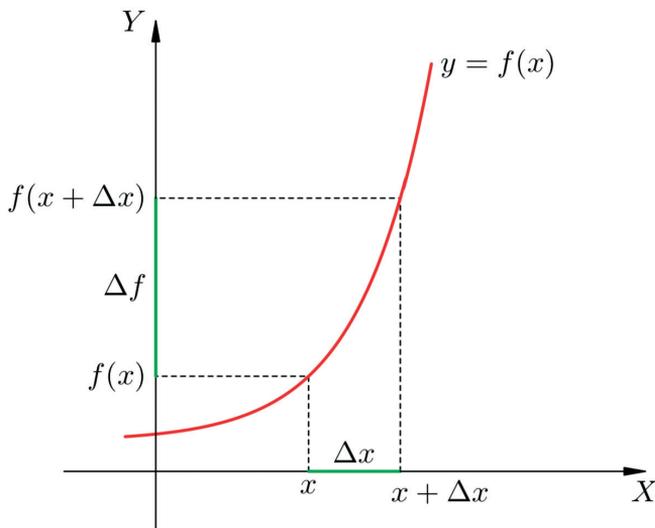


Рис. 3 Приращение аргумента и приращение функции

Для производной используются обозначения:  $f'(x)$  (читается как « $f$  штрих от  $x$ »),  $y'$  (читается как « $y$  штрих»),  $\frac{dy}{dx}$  (читается как « $dy$  по  $dx$ »).

Функцию, имеющую конечную производную (в некоторой точке), называют дифференцируемой (в данной точке).

### 1.3. ДИФФЕРЕНЦИРОВАНИЕ СЛОЖНЫХ ФУНКЦИЙ

Процесс вычисления производной называется *дифференцированием*. Производные простых функций можно легко вычислить с помощью формулы (7).

В машинном обучении вам часто придется дифференцировать степенную функцию – функцию вида  $f(x) = x^a$ . Формула производной степенной функции выглядит так:

$$(x^a)' = ax^{a-1}, a \in \mathbb{R}.$$

Производную степенной функции относят к табличным производным, которые нужно знать наизусть.

Еще чаще в машинном обучении нам придется находить производные сложных функций. Дифференцирование сложной функции происходит следующим образом. Сначала находим производную второй («внешней») функции и затем умножаем ее на производную первой («внутренней») функции. Эту процедуру мы еще называем правилом цепочки. Зная небольшое число табличных производных и располагая правилами дифференцирования, можно вычислять производные огромного количества функций.

Например, нужно вычислить производную функции  $y = (\theta - 5)^2$ .

Функция  $(x - 5)^2$  является композицией  $f(g(x))$  двух функций:  $f(u) = u^2$  и  $u = g(x) = x - 5$ .

Применяем правило цепочки  $(f(g(x)))' = \frac{d}{du} f(u) \cdot \frac{d}{dx} g(x)$ :

$$((x-5)^2)' = \frac{d}{du}(u^2) \frac{d}{dx}(x-5).$$

Применяем правило дифференцирования степенной функции:

$$\begin{aligned} \frac{d}{du}(u^n) &= n \cdot u^{n-1} \text{ с } n=2: \\ \frac{d}{du}(u^2) \frac{d}{dx}(x-5) &= (2u^{2-1}) \frac{d}{dx}(x-5) = 2u \frac{d}{dx}(x-5). \\ 2u \frac{d}{dx}(x-5) &= 2(x-5) \frac{d}{dx}(x-5) \end{aligned}$$

Производная от суммы (разности) равна сумме (разности) производных функций:

$$2(x-5) \frac{d}{dx}(x-5) = 2(x-5) \left( \frac{d}{dx}(x) - \frac{d}{dx}(5) \right) = (2x-10) \left( \frac{d}{dx}(x) - \frac{d}{dx}(5) \right).$$

Применяем правило дифференцирования степенной функции:

$$\begin{aligned} \frac{d}{dx}(x^n) &= n \cdot x^{n-1} \text{ с } n=1, \text{ другими словами, } \frac{d}{dx}(x) = 1: \\ (2x-10) \left( \frac{d}{dx}(x) - \frac{d}{dx}(5) \right) &= (2x-10) \left( (1) - \frac{d}{dx}(5) \right). \end{aligned}$$

Производная константы равна 0:

$$(2x-10) \left( 1 - \frac{d}{dx}(5) \right) = (2x-10)(1-(0)).$$

Таким образом,  $((x-5)^2)' = 2x-10$ .

## 1.4. ЧАСТНАЯ ПРОИЗВОДНАЯ

Встречаются зависимости не только от одной, но и от нескольких переменных. Например, площадь прямоугольника можно записать как  $S = xy$ . Значения  $S$  будут определяться совокупностью значений  $x$  и  $y$ . Это уже будет функция двух переменных. Объем  $V$  прямоугольного параллелепипеда с ребрами  $x$ ,  $y$  и  $z$  выражается формулой  $V = xyz$ , т.е. значения  $V$  зависят от трех переменных. Это уже будет функция трех переменных.

Обобщением понятия производной на случай функции нескольких переменных будет *частная производная*.

Частная производная – это предел отношения приращения функции по выбранной переменной к приращению этой переменной, при стремлении этого приращения к нулю.

Возьмем функцию двух переменных  $f(x, y)$ . Часто бывает важно знать, с какой скоростью меняются значения функции, когда мы перемещаемся по плоскости  $xy$ .

Частная производная по  $x$  от функции  $f(x, y)$  обозначается  $\frac{df}{dx}$ . Таким образом, по определению, частной производной по  $x$  от функции  $f(x, y)$  будет предел отношения частного приращения  $\Delta_x f$  по  $x$  к приращению  $\Delta x$  при стремлении  $\Delta x$  к нулю:

$$\frac{\partial f}{\partial x} = \lim_{\Delta x \rightarrow 0} \frac{\Delta_x f}{\Delta x} = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x, y) - f(x, y)}{\Delta x}.$$

Частная производная по  $y$  от функции обозначается  $\frac{df}{dy}$ . Частной производной по  $y$  от функции  $f(x, y)$  будет предел отношения частного приращения  $\Delta_y f$  по  $y$  к приращению  $\Delta y$  при стремлении  $\Delta y$  к нулю:

$$\frac{\partial f}{\partial y} = \lim_{\Delta y \rightarrow 0} \frac{\Delta_y f}{\Delta y} = \lim_{\Delta y \rightarrow 0} \frac{f(x, y + \Delta y) - f(x, y)}{\Delta y}.$$

С помощью этих частных производных мы можем вычислить скорость изменения функции вдоль координатных осей  $x$  и  $y$ .

При вычислении частной производной  $\frac{df}{dx}$  мы воспринимаем переменную  $y$  как константу. Аналогично, вычисляя  $\frac{df}{dy}$ , мы считаем  $x$  константой. Из этого ясно, что правила вычисления частных производных совпадают с правилами, указанными для функций одной переменной, и только требуется каждый раз помнить, по какой переменной ищется производная.

Теперь перейдем к градиенту.

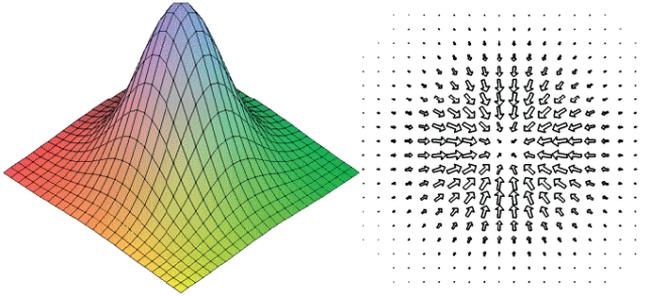
## 1.5. ГРАДИЕНТ

Градиентом функции многих переменных в данной точке называется вектор, координаты которого равны частным производным по соответствующим аргументам, вычисленным в данной точке.

Рассмотрим функцию двух переменных  $f(x, y)$ . Градиентом функции  $f(x, y)$  будет вектор  $\text{grad} f = \nabla f = \left( \frac{df}{dx}, \frac{df}{dy} \right)$ . Производные  $\frac{df}{dx}$  и  $\frac{df}{dy}$  вычисляются в каждой точке  $(x, y)$ . Таким образом, градиент задан в каждой точке и будет меняться от точки к точке.

Чем интересен градиент? Он интересен тем, что в данной точке он будет показывать нам направление наибольшего возрастания функции. Например, если взять высоту поверхности Земли над уровнем моря, то ее градиент в каж-

дой точке поверхности будет показывать направление «в гору». Модуль градиента совпадает с максимальной скоростью возрастания функции в данной точке. Когда у нас есть две переменные, то наш двухкомпонентный градиент может указать направление наибольшего возрастания функции на плоскости. Аналогично с тремя переменными – градиент может указать направление наибольшего возрастания функции в трехмерном пространстве.



**Рис. 4** Визуальная интерпретация градиента. Операция градиента преобразует холм (слева), если смотреть на него сверху, в поле векторов (справа). Видно, что векторы направлены «в гору» и чем длиннее, тем круче наклон. Источник: Википедия

## 1.6. ФУНКЦИЯ ПОТЕРЬ И ГРАДИЕНТНЫЙ СПУСК

Функция, для которой мы будем искать экстремум, в машинном обучении называется **целевой функцией (objective function)**. Задача по нахождению экстремума функции называется задачей оптимизации. Если речь идет о поиске минимума, то употребляют термины **функция стоимости (cost function)**, **функция потерь (loss function)**, **функция ошибок (error function)**.

Если функция дифференцируема, то найти точки, подозрительные на экстремум, можно с помощью необходимого условия экстремума: все частные производные должны равняться нулю, а значит, вектор градиента – нулевому вектору. Но не всегда задачу можно решать аналитически. В таком случае используется численная оптимизация. Наиболее простым в реализации из всех методов численной оптимизации является метод градиентного спуска, тесно связанный с понятием градиента.

Градиентный спуск – итерационный метод. Основная идея градиентного спуска состоит в том, чтобы двигаться к минимуму в направлении наиболее быстрого убывания функции потерь, которое определяется антиградиентом. В ходе градиентного спуска мы итеративно применяем следующее правило обновления:

$$w^t = w^{t-1} - \eta \nabla Q(w^{t-1}),$$

где  $\nabla Q(w^{t-1})$  – это градиент функции потерь, которую мы пытаемся минимизировать, а  $\eta$  – размер шага градиентного спуска, называемый *темпом обучения*, или *скоростью обучения (learning rate)*.

Мы выбираем каким-либо способом начальную точку, вычисляем в ней градиент рассматриваемой функции и делаем небольшой шаг в обратном, антиградиентном направлении. В результате приходим в точку, в которой значение функции будет меньше первоначального. В новой точке повторяем процедуру: снова вычисляем градиент функции и делаем шаг в обратном направлении. Продолжая этот процесс, мы будем двигаться в сторону убывания функции. Можно представить это как движение вниз по холму – сделав шаг вниз, текущая позиция будет ниже, чем предыдущая. Таким образом, на каждом следующем шаге высота будет как минимум не увеличиваться. Поэтому этот метод и называется спуском. Важно, чтобы наша функция была выпуклой и гладкой. Гладкой или непрерывно дифференцируемой функцией называют функцию, имеющую непрерывную производную на всем множестве определения. Выпуклой (или выпуклой вниз) функцией называют функцию, для которой отрезок между любыми двумя точками ее графика в векторном пространстве лежит не ниже соответствующей дуги графика. Выпуклость гарантирует существование лишь одного минимума, а гладкость – существование вектора градиента в каждой точке.

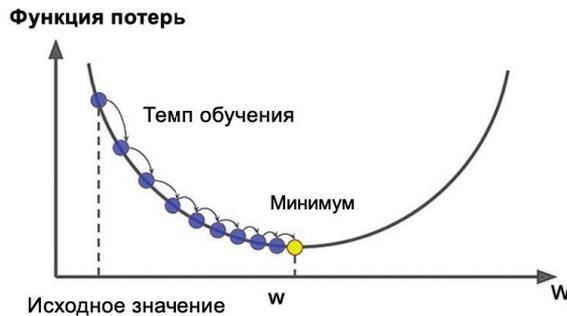


Рис. 5 Градиентный спуск

Допустим, необходимо минимизировать функцию вида  $y = (\theta - 5)^2$ , т.е. нам надо найти, при каком значении  $\theta$  наша функция принимает минимальное значение. Нужно выполнить следующие действия:

- 1) необходима производная по  $\theta$ :  $\frac{dy}{d\theta} = 2(\theta - 5) = 2\theta - 10$ ;
- 2) установим начальное значение  $\theta = 0$ ;
- 3) установим скорость обучения  $\gamma$  равной 0,2;
- 4) 20 раз подряд применим формулу  $\theta^t = \theta^{t-1} - \gamma \frac{dy}{d\theta}$ . У нас только один неизвестный параметр, поэтому будет одна формула.

```
n_iter = 20 # количество итераций
learning_rate = 0.2 # скорость обучения
def func(x):
    return (x - 5) ** 2
def func_derivative(x):
    return 2 * (x - 5)
previous_x, current_x = 0, 0
```

```

for i in range(n_iter):
    current_x = previous_x - learning_rate * func_derivative(previous_x)
    previous_x = current_x
    print("theta", format(current_x, ".6f"),
          "function value=", format(func(current_x), ".6f"),
          "derivative=", format(func_derivative(current_x), ".6f"))

```

theta 2.000000	function value= 9.000000	derivative= -6.000000
theta 3.200000	function value= 3.240000	derivative= -3.600000
theta 3.920000	function value= 1.166400	derivative= -2.160000
theta 4.352000	function value= 0.419904	derivative= -1.296000
theta 4.611200	function value= 0.151165	derivative= -0.777600
theta 4.766720	function value= 0.054420	derivative= -0.466560
theta 4.860032	function value= 0.019591	derivative= -0.279936
theta 4.916019	function value= 0.007053	derivative= -0.167962
theta 4.949612	function value= 0.002539	derivative= -0.100777
theta 4.969767	function value= 0.000914	derivative= -0.060466
theta 4.981860	function value= 0.000329	derivative= -0.036280
theta 4.989116	function value= 0.000118	derivative= -0.021768
theta 4.993470	function value= 0.000043	derivative= -0.013061
theta 4.996082	function value= 0.000015	derivative= -0.007836
theta 4.997649	function value= 0.000006	derivative= -0.004702
theta 4.998589	function value= 0.000002	derivative= -0.002821
theta 4.999154	function value= 0.000001	derivative= -0.001693
theta 4.999492	function value= 0.000000	derivative= -0.001016
theta 4.999695	function value= 0.000000	derivative= -0.000609
theta 4.999817	function value= 0.000000	derivative= -0.000366

В итоге находим значение параметра  $\theta$ , при котором функция  $y = (\theta - 5)^2$  принимает минимальное значение.

Очень важно при использовании метода градиентного спуска правильно подбирать шаг. Каких-либо конкретных правил подбора шага не существует, выбор шага – это искусство, но существует несколько полезных закономерностей. Если длина шага слишком мала, то метод будет не спеша, но верно шагать в сторону минимума. Если же взять размер шага очень большим, появляется риск, что метод будет перепрыгивать через минимум. Более того, есть риск того, что градиентный спуск не сойдется.

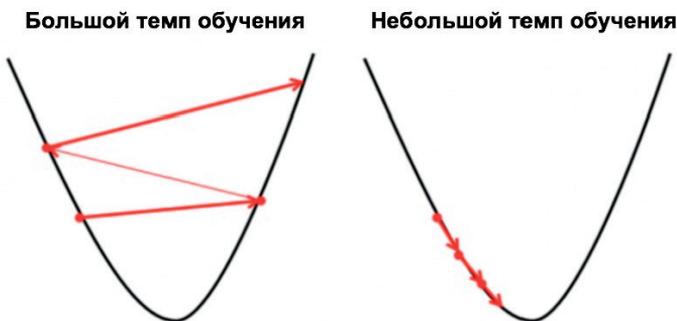


Рис. 6 Большой и маленький темпы обучения

В методе обычного градиентного спуска градиент вычисляется по всем наблюдениям обучающей выборки, формулу обновления для обычного градиентного спуска еще можно записать так:

$$w^t = w^{t-1} - \eta \nabla Q(w^{t-1}, X),$$

где  $X$  – обучающая выборка.

В этом и состоит основной недостаток метода градиентного спуска – в случае большой выборки даже одна итерация метода градиентного спуска будет осуществляться долго.

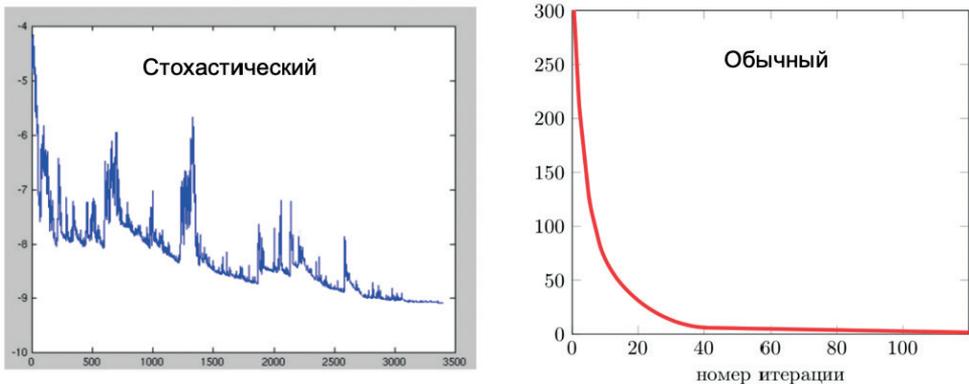
В методе стохастического градиентного спуска градиент функции качества вычисляется только на одном случайно выбранном объекте обучающей выборки:

$$w^t = w^{t-1} - \eta \nabla Q(w^{t-1}, \{x_{ij}\}),$$

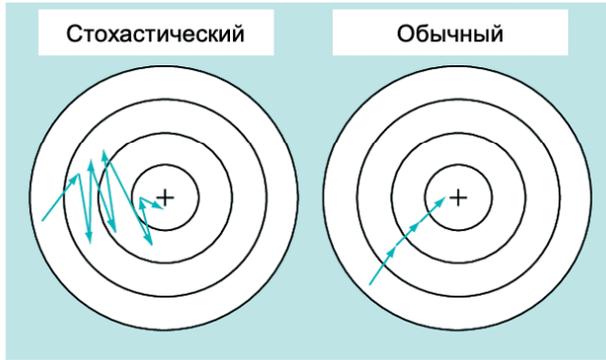
где  $\{x_{ij}\}$  – случайно отобранное наблюдение обучающей выборки.

Это позволяет обойти вышеупомянутый недостаток обычного градиентного спуска.

Показательно посмотреть на графики сходимости градиентного спуска и стохастического градиентного спуска. В обычном градиентном спуске на каждом шаге уменьшается суммарная ошибка на всех элементах обучающей выборки. График в таком случае обычно получается монотонным. В стохастическом градиентном спуске параметры меняются таким образом, чтобы максимально уменьшить ошибку для одного случайно выбранного объекта. Это приводит к тому, что график выглядит пилообразным, то есть на каждой конкретной итерации полная ошибка может как увеличиваться, так и уменьшаться. Но в итоге с ростом номера итерации значение функции уменьшается.



**Рис. 7** Зависимость функции потерь от номера итерации в обычном градиентном спуске и стохастическом градиентном спуске



**Рис. 8** Процесс обучения в обычном градиентном спуске и стохастическом градиентном спуске

### Задача с собеседования (математика)

1. Идут три охотника на охоту. У одного – 3 стакана крупы, у другого – 5 стаканов крупы, у третьего – 8 патронов. Сварили кашу, и все поели поровну. Третий охотник решил отблагодарить двух и отдать им все патроны. Как поделить патроны справедливо?

## Инструменты

### 1. ВВЕДЕНИЕ

Python стал одним из самых популярных языков, применяемых в машинном обучении для выполнения научных и коммерческих проектов. На момент написания книги экосистема Python предлагает широчайшие возможности для предварительной обработки данных и построения моделей машинного обучения.

Давайте вспомним основные структуры данных в Python. Нам понадобятся библиотека NumPy и модуль collections.

```
# импортируем библиотеку NumPy  
# и модуль collections  
import numpy as np  
import collections
```

#### 1.1. СТРУКТУРЫ ДАННЫХ

##### 1.1.1. Кортеж (tuple)

Кортеж – это *неизменяемая* последовательность с упорядоченными элементами. Мы можем создать кортеж, записав последовательность значений через запятую.

```
# создаем кортеж  
tup = 4, 5, 6  
# смотрим кортеж  
tup
```

```
(4, 5, 6)
```

Мы можем создать кортеж кортежей.

```
# создаем кортеж кортежей  
nested_tup = (4, 5, 6), (7, 8)  
# смотрим кортеж кортежей  
nested_tup
```

```
((4, 5, 6), (7, 8))
```

Любую последовательность или итератор можно преобразовать в кортеж с помощью функции `tuple()`.

```
# получаем кортеж
tup = tuple([3, 5, 2])
tup

(3, 5, 2)

# создаем кортеж
tup = tuple('кортеж')
tup

('к', 'о', 'р', 'т', 'е', 'ж')
```

К элементам кортежа, как и к элементам большинства других последовательностей Python, можно обращаться с помощью квадратных скобок []. Нумерация элементов последовательностей в Python начинается с нуля.

```
# обращаемся к первому элементу кортежа
tup[0]

'к'
```

### 1.1.2. Список (list)

Список – это *изменяемая* последовательность с упорядоченными элементами, т.е. в отличие от кортежей содержимое списков можно модифицировать. Список определяется с помощью квадратных скобок [] или функции `list()`.

```
# создаем список
lst = [3, 4, 5]
lst

[3, 4, 5]

# создаем список
lst = list('список')
lst

['с', 'п', 'и', 'с', 'о', 'к']
```

Для добавления элемента в конец списка служит метод `.append()`.

```
# создаем список
lst = ['a', 'b', 'c']
# добавляем элемент в конец списка
lst.append('d')
lst

['a', 'b', 'c', 'd']
```

Метод `.insert()` позволяет вставить элемент в указанную позицию списка. Мы должны указать позицию в списке и собственно сам элемент.

```
# создаем список
lst = [2, 4, 6, 9]
# вставляем 7 третьим элементом (нумерация с 0)
lst.insert(2, 7)
lst

[2, 4, 7, 6, 9]
```

Методом, обратным к `.insert()`, является `.pop()`, он удаляет из списка элемент, находившийся в указанной позиции, и возвращает его.

```
# удаляем третий элемент из списка
lst.pop(2)
7

# смотрим результат
lst

[2, 4, 6, 9]
```

Метод `.remove()` находит и удаляет из списка первый элемент с указанным значением.

```
# создаем список
lst = ['house', 'pop', 'techno', 'pop', 'trance']
# удаляем первый элемент с указанным значением
lst.remove('pop')
lst

['house', 'techno', 'pop', 'trance']
```

Для проверки, содержит ли список некоторое значение, используется оператор `in`.

```
# проверим, входит ли 'hardcore' в список
'hardcore' in lst
False
```

Проверка вхождения значения в случае списка занимает гораздо больше времени, чем в случае словаря или множества, потому что Python должен просматривать список от начала до конца, а это требует линейного времени, тогда как поиск в других структурах занимает постоянное время.

Операция сложения списков конкатенирует списки (то же самое будет с кортежами).

```
# складываем списки
['a', 'b'] + ['c', 'd']

# складываем кортежи
('a', 'b') + ('c', 'd')

['a', 'b', 'c', 'd']
('a', 'b', 'c', 'd')
```

Для добавления в конец имеющегося списка нескольких элементов можно использовать метод `.extend()`.

```
# создаем список
lst = ['house', 'pop', 'techno', 'pop', 'trance']
# добавляем в конец списка несколько элементов
lst.extend(['hardcore', 'speedcore'])
lst

['house', 'pop', 'techno', 'pop', 'trance', 'hardcore', 'speedcore']
```

С помощью метода `.clear()` можно очистить список.

```
# очищаем список
lst.clear()
lst

[]
```

Список можно отсортировать на месте (без создания нового объекта), вызвав его метод `.sort()`.

```
# создаем список
a = [7, 2, 5, 1, 3]
# сортируем
a.sort()
a
[1, 2, 3, 5, 7]
```

Метод `.count()` позволяет вычислить количество элементов с заданным значением.

```
# создаем список
lst = ['house', 'pop', 'techno', 'pop', 'trance']
# смотрим количество элементов с заданным значением
lst.count('pop')

2
```

Часто возникает потребность вычислить абсолютные частоты элементов списка. Это можно сделать с помощью функции библиотеки NumPy `np.unique()`. Сначала с ее помощью получаем массив уникальных значений и массив абсолютных частот, с помощью функции `zip()` «сшиваем» их и с помощью функции `dict()` записываем результат в словарь.

```
# вычисляем абсолютные частоты элементов с помощью np.unique()

# получаем массив уникальных значений и массив
# абсолютных частот с помощью np.unique
unique, counts = np.unique(lst, return_counts=True)
# с помощью zip() "сшиваем" два массива, с помощью
# dict() преобразовываем в словарь
freq = dict(zip(unique, counts))
```

```
freq
```

```
{'house': 1, 'pop': 2, 'techno': 1, 'trance': 1}
```

Кроме того, можно вычислить абсолютные частоты элементов с помощью класса Counter модуля collections.

```
# вычисляем абсолютные частоты элементов с помощью
# класса Counter модуля collections
freq = dict(collections.Counter(lst))
freq
```

```
{'house': 1, 'pop': 2, 'techno': 1, 'trance': 1}
```

От абсолютных частот легко перейти к относительным.

```
# теперь получим относительные частоты
freq = dict(zip(unique, counts * 100 / len(lst)))
freq
```

```
{'house': 20.0, 'pop': 40.0, 'techno': 20.0, 'trance': 20.0}
```

Метод `.reverse()` разворачивает список.

```
# создаем список
lst = [2, 4, 6, 8]
# разворачиваем список
lst.reverse()
lst
```

```
[8, 6, 4, 2]
```

Важно понимать, в каких ситуациях лучше использовать списки, а в каких – кортежи:

- списки содержат однородные данные – они должны быть одного типа, иначе с таким списком будет тяжело работать;
- кортеж, напротив, может содержать разнородные данные, которые в зависимости от позиции могут иметь тот или иной тип;
- список нужно использовать, когда длина заранее неизвестна либо она переменна, например список пользователей;
- кортеж удобно использовать, когда длина данных известна заранее и строго фиксирована, например как в записи из сформированной электронной таблицы.

### 1.1.3. Словарь (dictionary)

Словарь – одна из самых важных встроенных в Python структур данных. Его также называют *хешем*, *отображением* или *ассоциативным массивом*. Он представляет собой коллекцию пар ключ-значение переменного размера, в которой и ключ, и значение – объекты Python. Создать словарь можно с помощью фигурных скобок {}, отделяя ключи от значений двоеточием.

Давайте создадим пустой словарь.

```
# создаем пустой словарь
empty_dict = {}
```

А теперь создадим словарь, в котором ключами будут имена переменных, а значениями – значения переменных. Такие словари часто используются в циклах, например когда по каждой переменной требуется вычислить среднее значение, положить его в словарь (ключом для значения будет соответствующая переменная), а потом извлечь среднее значение для замены пропущенного значения в соответствующей переменной.

```
# создаем еще один словарь
d = {'age': 25, 'income': 20000}
```

Словарь также можно создать с помощью функции `dict()`. Вместо двоеточия уже используется знак равенства `=`, названия ключей берутся без кавычек.

```
# создаем словарь с помощью функции dict()
d = dict(age=25, income=20000)
d
{'age': 25, 'income': 20000}
```

Словарь можно создать с помощью метода `.fromkeys()`.

```
# создаем словарь с помощью метода .fromkeys()
d = dict.fromkeys(['a', 'b'], 25)
d
{'a': 25, 'b': 25}
```

А теперь создадим словарь, в котором ключами будут строковые значения, а значения будут представлять собой списки. Такие словари используются для поиска оптимальных значений гиперпараметров моделей предварительной подготовки и моделей машинного обучения.

```
# создаем словарь, где значениями являются списки, используется
# для поиска оптимальных значений гиперпараметров
d = {'max_depth': [2, 4, 6], 'max_features': [3, 6, 9]}
```

Словарь можно создать на основе двух последовательностей, рассматриваемых как ключи и значения.

```
# создаем список ключей
key_list = ['a', 'b', 'c']
# создаем список значений
value_list = [10, 20, 30]

# создаем словарь на основе двух списков
mapping = {}
for key, value in zip(key_list, value_list):
```

```
mapping[key] = value
```

```
# смотрим словарь
```

```
mapping
```

```
{'a': 10, 'b': 20, 'c': 30}
```

```
# или так
```

```
dict(zip(key_list, value_list))
```

```
{'a': 10, 'b': 20, 'c': 30}
```

Для доступа к элементам, вставки и присваивания применяется такой же синтаксис, как в случае списка или кортежа.

```
# выполняем присваивание значения
```

```
d = {'a': 55, 'b': 35}
```

```
d['b'] = 30
```

```
d
```

```
{'a': 55, 'b': 30}
```

```
# вставляем новую пару ключ-значение
```

```
d['c'] = 10
```

```
d
```

```
{'a': 55, 'b': 30, 'c': 10}
```

Проверка наличия ключа в словаре тоже производится, как для кортежа или списка.

```
# проверяем наличие ключа 'c' в словаре
```

```
'c' in d
```

```
True
```

Методы `.keys()` и `.values()` возвращают соответственно ключи и значения. Хотя точный порядок пар ключ-значение не определен, эти методы возвращают ключи и значения в одном и том же порядке.

```
# возвращаем ключи с помощью метода .keys()
```

```
d.keys()
```

```
dict_keys(['a', 'b', 'c'])
```

```
# возвращаем значения с помощью метода .values()
```

```
d.values()
```

```
dict_values([55, 30, 10])
```

Обратите внимание, что возвращаемые объекты не являются обычными списками. Речь идет о динамическом представлении элементов словаря. Чтобы получить списки ключей или списки значений, нужно воспользоваться генераторами списков `[k for k in d.keys()]` или `[k for k in d.values()]`.

```
# получаем список из ключей словаря
[k for k in d.keys()]

['a', 'b', 'c']

# получаем список из значений словаря
[k for k in d.values()]

[55, 30, 10]
```

Метод `.items()` возвращает пары ключ-значение.

```
# возвращаем пары ключ-значение
d.items()

dict_items([('a', 55), ('b', 30), ('c', 10)])
```

Давайте извлечем первую и последнюю пары.

```
# извлечем первую пару
list(d.items())[0]

('a', 55)

# извлечем последнюю пару
list(d.items())[-1]

('c', 10)
```

Два словаря можно объединить в один с помощью метода `.update()`.

```
# объединяем два словаря в один
d.update({'c': 10, 'd': 12})
d

{'a': 55, 'b': 30, 'c': 10, 'd': 12}
```

Для удаления элемента по ключу можно использовать и ключевое слово `del`.

```
# удаляем элемент по ключу с помощью del
del d['a']
d

{'b': 30, 'c': 10, 'd': 12}
```

Метод `.pop()` удаляет элемент по ключу и возвращает соответствующее ключу значение.

```
# удаляем элемент по ключу с помощью .pop()
print(d.pop('b'))
print(d)

30
{'c': 10, 'd': 12}
```

Метод `.popitem()` удаляет последний элемент словаря и возвращает удаленный ключ и значение.

```
# удаляем последний элемент с помощью .popitem()
print(d.popitem())
print(d)

{'d': 12}
{'c': 10}
```

Метод `.get()` возвращает значение словаря по ключу.

```
# с помощью .get() возвращаем значение по ключу
d.get('c')

10
```

### 1.1.4. Множество (set)

Множество – это неупорядоченный набор уникальных элементов. Множества не упорядочены, они не хранят ни позицию элемента, ни порядок вставки. Поэтому наборы не поддерживают ни обращение к элементам по индексам, ни срезы, ни какое-либо другое поведение, присущее последовательностям. Множества создаются в абсолютно случайном порядке каждый раз. Вы можете размещать элементы как вам будет угодно, но они все равно будут расположены впоследствии в случайном порядке. Во-вторых, множества не могут иметь повторяющихся элементов, поэтому все элементы, которые будут одинаковыми, не будут выведены повторно.

Их очень удобно использовать, если вы хотите удалить повторяющиеся элементы из списка.

Множество можно создать с помощью функции `set()` или фигурных скобок `{}` (однако создать пустое множество с помощью фигурных скобок нельзя, вместо пустого множества вы создадите пустой словарь).

```
# создаем множество с помощью функции set()
a = set([2, 2, 2, 1, 3, 3])
a

{1, 2, 3}

# создаем множество с помощью фигурных скобок {}
a = {2, 2, 2, 1, 3, 3}
a

{1, 2, 3}
```

С помощью метода `.add()` можно добавить элемент в множество.

```
# добавляем элемент в множество с помощью метода .add()
a.add(4)
a

{1, 2, 3, 4}
```

С помощью метода `.remove()` можно удалить элемент из множества.

```
# удаляем элемент из множества с помощью метода .remove()
a.remove(1)
a
{2, 3, 4}
```

Для проверки, содержит ли множество интересующий элемент, используется оператор `in`.

```
# проверяем, содержит ли множество элемент 5
5 in a
False
```

Метод `.clear()` очищает множество.

```
# с помощью метода .clear() очищаем множество
a.clear()
a
set()
```

С помощью метода `.union()` можно найти все уникальные элементы, входящие либо в первое, либо во второе множества.

```
# создаем множества a и b
a = set([1, 2, 3, 6])
b = set([4, 6, 2, 8])

# найдем все уникальные элементы,
# входящие либо в a, либо в b
a.union(b)
```

```
{1, 2, 3, 4, 6, 8}
```

```
# это эквивалентно синтаксису
a | b
```

```
{1, 2, 3, 4, 6, 8}
```

С помощью метода `.intersection()` можно найти все элементы, входящие и в первое, и во второе множества.

```
# находим все элементы, входящие и в a, и в b
a.intersection(b)
{2, 6}
```

```
# это эквивалентно синтаксису
a & b
```

```
{2, 6}
```

С помощью метода `.difference()` можно найти все элементы, входящие в первое множество, но не входящие во второе.

```
# находим все элементы, входящие в a, но не входящие в b
a.difference(b)
```

```
{1, 3}
```

```
# это эквивалентно синтаксису
```

```
a - b
```

```
{1, 3}
```

С помощью метода `.symmetric_difference()` можно найти элементы, входящие либо в первое множество, либо во второе множество, но не в первое и второе множества одновременно.

```
# находим элементы, входящие либо в a, либо в b,
# но не в a и b одновременно
a.symmetric_difference(b)
```

```
{1, 3, 4, 8}
```

```
# это эквивалентно синтаксису
```

```
a ^ b
```

```
{1, 3, 4, 8}
```

С помощью метода `.update()` можно добавить в первое множество все элементы из второго множества.

```
# добавляем в множество a все элементы из множества b
a.update(b)
a
```

```
{1, 2, 3, 4, 6, 8}
```

Можно также проверить, является ли множество подмножеством или надмножеством другого множества.

```
# создаем множества
```

```
a = {1, 2, 3, 4, 5}
```

```
b = {1, 2, 3}
```

```
# проверяем, является ли множество b
# подмножеством множества a
```

```
b.issubset(a)
```

```
True
```

```
# проверяем, является ли множество a надмножеством множества b
# (мы проверяем, содержит ли множество a в себе множество b)
```

```
a.issuperset(b)
```

```
True
```

Теперь от структур данных перейдем к функциям и классам.

## Задача с собеседования (SQL)

1. Создайте с помощью языка SQL таблицу `employees`.

id	name	age	department
1	David	22	B
2	Paul	33	B
3	Jeremy	26	B
4	Jack	21	C
5	John	36	A
6	David	45	A

## 1.2. Функция

Функция в Python – это объект, выполняющий определенное действие. Функции могут принимать параметры, т.е. некоторые значения, передаваемые функции для того, чтобы она что-либо *сделала* с ними.

Параметры указываются как имена в скобках при объявлении функции и разделяются запятыми. Аналогично мы передаем значения при вызове функции. Обратите внимание на терминологию: имена, указанные в объявлении функции, называются **параметрами**, тогда как значения, которые вы передаёте в функцию при её вызове, – **аргументами**. Обычно функция определяется с помощью инструкции `def`.

Инструкция `return` говорит, что нужно вернуть значение.

Например, мы можем написать функцию `square()`, она принимает параметры `width` и `height`.

```
# выясним, что является параметрами, а что – аргументами
# здесь имена width и height, указанные в скобках
# при объявлении функции square(), – это параметры
def square(width, height):
    return width * height
```

Когда функция `square()` вызывается, то ей передаются аргументы. В примере ниже мы задали глобальные переменные `w` и `h` и передали в функцию `square()`.

Однако на самом деле передаются не эти переменные, а их значения. В данном случае числа 10 и 20. Другими словами, мы могли бы писать `square(10, 20)`. Разницы не было бы.

```
# значения, которые передаем при вызове
# функции square(), – аргументы
w = 10
h = 20
```

```
square(w, h)
```

```
200
```

```
# а еще можно было так
```

```
square(10, 20)
```

```
200
```

## Задача с собеседования (математика)

1. Является ли число 3599 простым?

### 1.3. ПОЛЕЗНЫЕ ВСТРОЕННЫЕ ФУНКЦИИ

#### 1.3.1. Функция enumerate()

Функция `enumerate()` применяется для итерируемых коллекций (строки, списки, словари и др.) и возвращает кортежи, состоящие из двух элементов – индекса элемента и самого элемента. Она используется для упрощения прохода по коллекциям в цикле, когда кроме самих элементов требуется их индекс. У нее – два параметра. Параметр `iterable` задает объект, элементы которого будем перебирать (список, множество, словарь). Параметр `start` задает начальное значение индекса. По умолчанию начальное значение равно 0.

```
# создаем список
```

```
a = [10, 20, 30, 40]
```

```
# с помощью цикла for и функции enumerate() возвращаем
```

```
# кортеж (индекс значения, само значение)
```

```
for i in enumerate(a):
```

```
    print(i)
```

```
(0, 10)
```

```
(1, 20)
```

```
(2, 30)
```

```
(3, 40)
```

Здесь мы с помощью `enumerate()` перебираем весь список `a`. При переборе мы каждый раз получаем кортеж, состоящий из двух элементов: индекс (порядковый номер) элемента в списке и значение элемента.

```
# а еще можно так
```

```
for num, val in enumerate(a):
```

```
    print(num, val)
```

```
0 10
```

```
1 20
```

```
2 30
```

```
3 40
```

Выполняем аналогичную операцию. В переменной `num` будет индекс элемента в списке, в переменной `val` – значение элемента.

Приведем еще один пример.

```
# еще один пример
data = [2, 5, 3, 4, 1, 5]
for num, val in enumerate(data, 1):
    print(str(num) + '-е значение равно ' + str(val))
1-е значение равно 2
2-е значение равно 5
3-е значение равно 3
4-е значение равно 4
5-е значение равно 1
6-е значение равно 5
```

С помощью `enumerate()` мы перебираем весь список `data`. При переборе мы опять получали кортеж, состоящий из двух элементов. В переменной `num` будет индекс элемента в списке. В переменной `val` – значение элемента. В качестве второго аргумента в функции `enumerate()` мы использовали цифру 1, чтобы индексы начинались с единицы, а не с нуля.

### 1.3.2. Функция `sorted()`

Функция `sorted()` возвращает новый отсортированный список, построенный из элементов произвольной последовательности.

```
# получаем отсортированный массив
sorted([7, 1, 2, 6, 0, 3, 2])
[0, 1, 2, 2, 3, 6, 7]
```

### 1.3.3. Функция `zip()`

Функция `zip()` «сшивает» (от англ. *zip* – застегивать на молнию) элементы нескольких списков, кортежей или других последовательностей в пары, создавая список кортежей.

```
# создаем два списка
a = [10, 20, 30, 40]
b = ['a', 'b', 'c', 'd', 'e']

# сшиваем элементы списков в кортежи
zipped = list(zip(a, b))
zipped

[(10, 'a'), (20, 'b'), (30, 'c'), (40, 'd')]

# "сшиваем" с помощью цикла for и функции zip()
for i, j in zip(a, b):
    print(i, j)

10 a
20 b
30 c
40 d
```

Функция `zip()` принимает любое число аргументов, а количество порождаемых ей кортежей определяется длиной *самой короткой* последовательности. В нашем случае это количество определяется длиной списка `a`. Часто функцию

`zip()` применяют в связке с функцией `enumerate()`, чтобы дополнительно получить индексы.

```
# "сшиваем" с помощью цикла for,
# функций zip() и enumerate()
for i, (x, y) in enumerate(zip(a, b)):
    print(i, x, y)
```

```
0 10 a
1 20 b
2 30 c
3 40 d
```

Мы можем сразу «сшить» два списка и отсортировать элементы с помощью функций `zip()` и `sorted()`.

```
# создаем списки
letters = ['b', 'a', 'd', 'c']
numbers = [2, 4, 3, 1]

# "сшиваем" списки и сортируем элементы
data = sorted(zip(letters, numbers))
data

[('a', 4), ('b', 2), ('c', 1), ('d', 3)]
```

Функцию `zip()` можно использовать для выполнения арифметических операций.

```
# создаем списки - продажи и затраты
total_sales = [52000.00, 51000.00, 48000.00]
prod_cost = [46800.00, 45900.00, 43200.00]
# вычисляем и печатаем значения прибыли
for sales, costs in zip(total_sales, prod_cost):
    profit = sales - costs
    print(f"Total profit: {profit}")
```

```
Total profit: 5200.0
Total profit: 5100.0
Total profit: 4800.0
```

Если у вас уже есть список кортежей и вы хотите выделить элементы каждого кортежа в отдельные последовательности (то есть хотите «распороть» имеющуюся последовательность), вы можете воспользоваться функцией `zip()` в сочетании с оператором распаковки `*`.

```
# создаем список кортежей
pairs = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
# "распарываем" на два списка с помощью функции zip()
# и оператора распаковки *
numbers, letters = zip(*pairs)
```

```
numbers
```

(1, 2, 3, 4)

letters

('a', 'b', 'c', 'd')

## Задача с собеседования (математика)

1. Что больше,  $\log_2(3)$  или  $\log_3(5)$ ?

### 1.4. Класс

Класс – тип, описывающий устройство объектов. Создание класса в Python начинается с инструкции `class`.

Вот так будет выглядеть минимальный класс.

```
# создаем класс  
class C:  
    pass
```

Класс состоит из объявления (инструкция `class`), имени класса (в нашем случае это имя `c`, в Python конвенция указывает на то, что название класса должно начинаться с заглавной буквы и иметь мужской род) и тела класса, которое содержит атрибуты и методы. Обратите внимание, что в нашем минимальном классе есть только одна инструкция `pass`. Она представляет собой пустую операцию (и она ничего не выполняет). Вы можете использовать `pass` в любом месте, где интерпретатор ожидает встретить настоящую инструкцию. Сейчас мы не готовы определить все детали класса `c`, поэтому использовали `pass`, чтобы избежать любых синтаксических ошибок, которые появились бы при попытке создать класс без какого-либо кода в его теле.

Класс может содержать атрибуты. Ниже представлен класс, содержащий атрибуты `color` (цвет), `width` (ширина), `height` (высота). Атрибут – это переменная, содержащая определенную характеристику объекта.

```
# у класса есть атрибуты  
class Rectangle:  
    color = 'green'  
    width = 100  
    height = 100
```

Чтобы воспользоваться возможностями класса, нужно создать объект, и эта операция называется созданием экземпляра или инстанса класса (инстатированием).

```
# создаем экземпляр класса  
rect1 = Rectangle()
```

Доступ к атрибуту класса можно получить следующим образом: имя\_объекта.атрибут.

```
# получаем доступ к атрибуту
print(rect1.color)
```

```
green
```

Добавим к нашему классу метод. Метод – это функция, находящаяся внутри класса (или, говорят, «функция, принадлежащая объекту», этим объектом может быть экземпляр класса). Он выполняет определенное действие, которое чаще всего предполагает доступ к атрибутам созданного объекта. Метод объекта вызываем так: `имя_объекта.имя_метода()`. Метод часто обозначают так: `.имя_метода()`, например `.fit()`.

Скобки говорят о том, что мы имеем дело с функцией, а точка говорит о принадлежности функции к определенному объекту.

Например, в наш класс `Rectangle` можно добавить метод, вычисляющий площадь прямоугольника. Для того чтобы метод в классе знал, с каким объектом он работает (это нужно для того, чтобы получить доступ к атрибутам: ширине (`width`) и высоте (`height`)), первым аргументом ему следует передать параметр `self`, через который он может получить доступ к своим данным. Параметр `self` – это ссылка на конкретный экземпляр класса. С помощью него мы указываем, что нас интересует не ширина и высота вообще, а ширина и высота конкретного объекта – созданного нами экземпляра класса. Параметр – это наш внешний способ обратиться к атрибуту, находящемуся внутри класса. Вспомним, что параметры указываются как имена в скобках при объявлении функции и разделяются запятыми. Также вспомним, что функцию мы определяем с помощью инструкции `def`.

```
# добавляем в класс метод square
class Rectangle:
    атрибуты { color = 'green'
              width = 100
              height = 100

    метод { def square(self):
            return self.width * self.height
```

Обратите внимание, не все атрибуты могут быть доступными извне или публичными (`public`), они бывают защищенными (`protected`) и обозначаются через символ нижнего подчеркивания `_`, они могут быть частными (`private`) и обозначаются через двойной символ нижнего подчеркивания `__`. Методы также могут быть публичными, защищенными и частными.

Однако, строго говоря, в Python, в отличие от Java, C++, нет механизма, который эффективно ограничивал бы доступ к любой переменной или методу экземпляра. В Python просто принято соглашение о префиксе имени переменной/метода с одним или двойным символом подчеркивания для эмуляции поведения спецификаторов защищенного и частного доступа. Зачем же тогда нужны подчеркивания? Культура программирования Python заключается в том, что имена, начинающиеся с подчеркивания, означают «не трогайте данный атрибут/метод, если вы действительно не знаете, какую задачу он выполняет». Например, если вы видите частный метод `__loss`, то вы должны

понять, что, скорее всего, он нужен для правильной внутренней работы класса, используется внутри класса, например для того, чтобы вычислять логистическую функцию потерь на основе обновленных вероятностей положительного класса, получаемых с помощью обновленных весов в ходе градиентного спуска. Поэтому, в отличие от публичного метода `.fit()`, вам совершенно не нужно вызывать его отдельно.

Вернемся к нашему классу `Rectangle`. Давайте поработаем с атрибутами и методами класса.

```
# создаем экземпляр класса
rect2 = Rectangle()
# взглянем на значение атрибута color
print(rect2.color)

green

# вычислим площадь с помощью метода .square()
print(rect2.square())

10000

# заново создаем экземпляр класса
rect3 = Rectangle()
# зададим новое значение атрибута width
rect3.width = 200
# зададим новое значение атрибута color
rect3.color = 'brown'
# взглянем на значение атрибута color
print(rect3.color)

brown

# заново вычислим площадь с помощью метода .square()
print(rect3.square())

20000
```

Конструктор класса позволяет задать определенные параметры объекта при его создании. Таким образом появляется возможность создавать объекты с уже заранее заданными атрибутами. Конструктором класса является метод `__init__`. Мы используем его для инициализации атрибутов класса.

Например, для того чтобы иметь возможность задать цвет, длину и ширину прямоугольника при его создании, добавим к классу `Rectangle` следующий конструктор:

```

# добавляем в класс конструктор
class Rectangle:
    def __init__(self, color='green', width=100, height=100):
        self.color = color
        self.width = width
        self.height = height

    def square(self):
        return self.width * self.height

```

Метод `init`, задающий конструктор класса (инициализация атрибутов)

Метод `square()`, вычисляющий площадь

Добавляем параметры `color`, `width` и `height` в определение метода `__init__`

Используем значения `color`, `width` и `height` для инициализации атрибутов класса (которые представлены как `self.color`, `self.width`, `self.height`)

Метод `square()` вычисляет площадь

Давайте воспользуемся новым классом.

```

# заново создаем экземпляр класса
rect4 = Rectangle()
# взглянем на значение атрибута color
print(rect4.color)

green

# заново вычислим площадь
print(rect4.square())

10000

# заново создаем экземпляр класса, передав
# конкретные значения параметров
rect5 = Rectangle('yellow', 23, 34)
# взглянем на значение атрибута color
print(rect5.color)

yellow

# заново вычислим площадь
print(rect5.square())

782

```

Когда мы хотим, чтобы наш класс использовал функционал родительского класса, необходимо наследование. В организации наследования участвуют как минимум два класса: класс-родитель и класс-потомок. При этом возможно множественное наследование, в этом случае у класса-потомка есть несколько родителей. Родительский класс помещается в скобки после имени класса.

```

# создаем родительский класс
class Figure:
    def __init__(self, color):
        self.color = color
    def get_color(self):
        return self.color

# создаем дочерний класс
class Rectangle(Figure):
    def __init__(self, color, width=100, height=100):
        super().__init__(color)
        self.width = width
        self.height = height

```

```
def square(self):
    return self.width * self.height
```

В данном случае с помощью функции `super()` мы вызовем родительский конструктор. В противном случае мы не сможем воспользоваться атрибутом `color`. Функция `super()` используется, когда нужно обратиться к атрибутам и методам родительского класса.

```
# заново создаем экземпляр класса,
# задав значение параметра
rect6 = Rectangle('blue')
# получим значение color
print(rect6.get_color())
```

blue

```
# вычислим площадь
print(rect6.square())
```

10000

```
# заново создаем экземпляр класса,
# задав значения параметров
rect7 = Rectangle('red', 25, 70)
# получим значение color
print(rect7.get_color())
```

red

```
# вычислим площадь
print(rect7.square())
```

1750

## Задачи с собеседований (Python)

1. Расскажите об операторах принадлежности `in` и `not in`.
2. Расскажите про операторы тождественности `is` и `is not`.
3. Дано `a = [1, 2, 3]` и `b = [1, 2, 3]`:
  - какое значение получим, применив оператор `a == b`?
  - какое значение получим, применив оператор `a is b`?
 Необходимо пояснить причину получения данных значений.
4. Какое значение получим, применив оператор `a is b`:
  - если `a = 123` и `b = 123`?
  - если `a = 100500` и `b = 100500`?
 Необходимо пояснить причину получения данных значений.
5. Преобразуйте список `['one', 'two', 'three', 'four']` в строку.
6. У нас есть список менеджеров, и по каждому менеджеру известен результат его работы – объем продаж, нужно получить реестр вида:

```
Petrov 200000
Sidorov 100000
Ivanov 50000
```

7. Как в Python узнать, в каком каталоге мы сейчас находимся?
8. Создайте список [1, 3, 5, 7, 9] с помощью генератора списков (нужно воспользоваться одной строкой программного кода).
9. Обратите порядок элементов в списке [1, 3, 5, 7, 9].
10. У нас есть список под названием *a* вида [9, 7, 5, 3, 1], какой элемент вернет программный код `a[-2]`?
11. У нас есть два списка ['Petrov', 'Sidorov', 'Ivanov'] и [200000, 100000, 50000]. Нужно превратить их в словарь вида {'Petrov': 200000, 'Sidorov': 100000, 'Ivanov': 50000}.
12. Как убрать дубликаты из списка [1, 2, 1, 3, 4, 2]?
13. Как работает отрицательный индекс?
14. Необходимо получить индекс по каждому элементу списка ['Petrov', 'Ivanov', 'Sidorov'] в следующем виде:

```
(0, 'Petrov')
(1, 'Ivanov')
(2, 'Sidorov')
```

15. Дан список признаков ['age', 'credit', 'debt'] и список регрессионных коэффициентов [0.4, -0.2, -0.13]. Нужно получить список кортежей вида [('age', 0.4), ('credit', -0.2), ('debt', -0.13)].
16. У нас есть набор данных, записанный в файле *alfa\_python\_test.csv*.
  - а) Для каждого уникального *id* оставьте только одну строку с самой поздней датой.
  - б) Сколько теперь в каждом городе находится уникальных *id*?
  - в) Сколько теперь *id* содержится в каждом множестве городов?

## 1.5. ЗНАКОМСТВО С ANACONDA

Для предварительной подготовки данных и построения моделей в Python нам потребуется ряд библиотек: NumPy, SciPy, matplotlib, pandas, IPython и scikit-learn. Настоятельно рекомендуем воспользоваться дистрибутивом Anaconda, который уже включает все необходимые библиотеки. Есть версии для Mac OS, Windows и Linux.

Anaconda Distribution можно загрузить с веб-сайта Continuum Analytics по адресу <https://www.anaconda.com/download/>. Веб-сервер определит операционную систему вашего браузера и предоставит вам соответствующий вашей системе файл загрузки.

При открытии этого URL-адреса в вашем браузере вы увидите страницу примерно следующего вида:

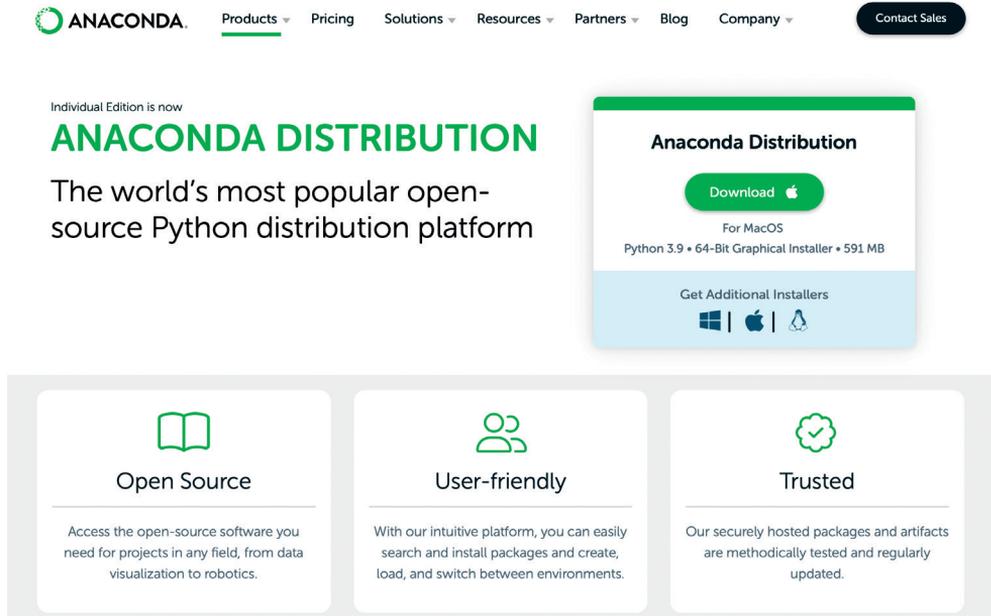


Рис. 1 Стартовая страница Anaconda Distribution

Загрузите инсталлятор для Python 3.9. Запустите инсталлятор и установите Anaconda Distribution.

Теперь, когда у нас установлено все необходимое, давайте перейдем к использованию IPython и Jupyter Notebook.

## 2. IPYTHON И JUPYTER NOTEBOOK

IPython – это альтернативная оболочка для интерактивной работы с Python. Она предлагает несколько усовершенствований для REPL, поставляемой по умолчанию. REPL – это форма организации простой интерактивной среды программирования в рамках средств интерфейса командной строки (REPL, от англ. *read-eval-print loop* – цикл «чтение–вычисление–вывод»), которая поставляется вместе с Python.

Чтобы запустить IPython, просто выполните команду `ipython` из командной строки/терминала.

```

Python: C:\Users\Gewissta
(base) C:\Users\Gewissta>ipython
Python 3.6.5 [Anaconda, Inc.] (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 6.4.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 1 + 1
Out[1]: 2

In [2]:

```

Рис. 2 Командная строка

Командная строка ввода показывает `In[1]:`. Каждый раз, когда вы будете вводить инструкцию в REPL IPython, число в командной строке будет увеличиваться.

Аналогично вывод для какой-либо конкретной записи будет предваряться `Out[x]:`, где `x` соответствует номеру `In[x]:`.

Данная нумерация операций ввода и вывода будет важна для примеров, поскольку все примеры будут предваряться `In[x]:` и `Out[x]`, и таким образом можно будет отследить последовательность выполнения операций.

Обратите внимание, что эти числа являются строго последовательными. Если вы запускаете программный код и при вводе возникают ошибки, или вы вводите дополнительные инструкции, нумерация перестанет быть последовательной (ее можно сбросить, выйдя и перезапустив IPython).

Jupyter Notebook – это веб-оболочка для IPython (ранее называлась IPython Notebook). Это веб-приложение с открытым исходным кодом, которое позволяет создавать и обмениваться документами, содержащими живой код, уравнения, визуализацию и разметку.

Первоначально IPython Notebook ограничивался лишь Python в качестве единственного языка. Jupyter Notebook позволил использовать многие языки программирования, включая Python, R, Julia, Scala и F#. Если вы хотите глубже познакомиться с Jupyter Notebook, перейдите на <http://jupyter.org/>, где вы увидите страницу следующего вида:

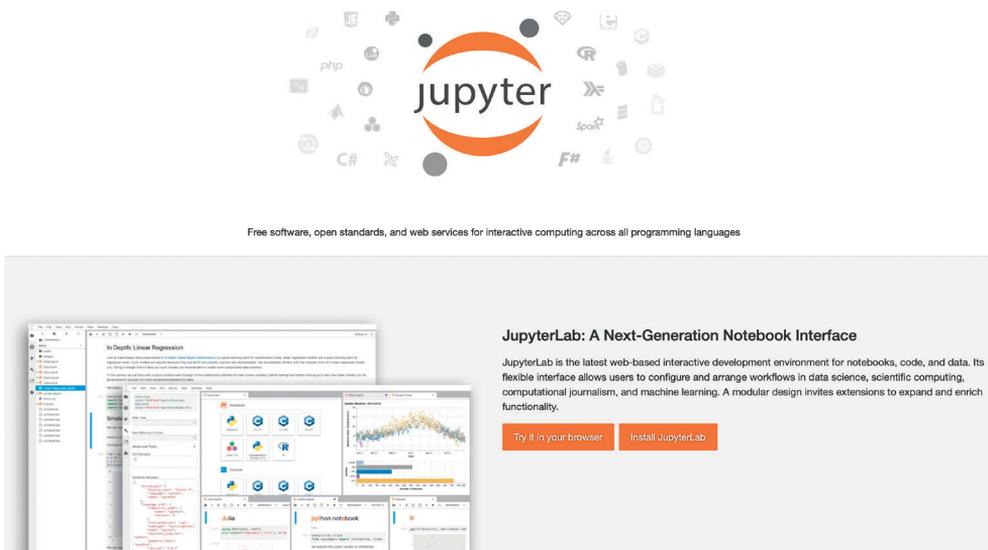


Рис. 3 Стартовая страница сайта <https://jupyter.org>

Jupyter Notebook можно скачать и использовать независимо от Python. Anaconda устанавливает его по умолчанию. Чтобы запустить Jupyter Notebook, введите в Anaconda Prompt следующую команду:

```
jupyter notebook
```

```

Anaconda Prompt - jupyter notebook
(base) C:\Users\Gewissta>jupyter notebook
[W 11:31:18.931 NotebookApp] Terminals not available (error was DLL load failed: %1 не является приложением Win32.)
[I 11:31:18.974 NotebookApp] JupyterLab beta preview extension loaded from C:\Anaconda3\lib\site-packages\jupyterlab
[I 11:31:18.974 NotebookApp] JupyterLab application directory is C:\Anaconda3\share\jupyter\lab
[I 11:31:19.125 NotebookApp] Serving notebooks from local directory: C:\Users\Gewissta
[I 11:31:19.125 NotebookApp] 0 active kernels
[I 11:31:19.125 NotebookApp] The Jupyter Notebook is running at:
[I 11:31:19.126 NotebookApp] http://localhost:8888/?token=bf8697304186f897d91cd92bc13553350ea34f93da97a060
[I 11:31:19.126 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 11:31:19.145 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
http://localhost:8888/?token=bf8697304186f897d91cd92bc13553350ea34f93da97a060&token=bf8697304186f897d91cd92bc13553350ea34f93da97a060
[I 11:31:19.271 NotebookApp] Accepting one-time-token-authenticated connection from ::1

```

Рис. 4 Ввод команды jupyter notebook

Откроется страница браузера, отображающая домашнюю страницу Jupyter Notebook (<http://localhost:8888/tree>). Если щелкнуть по файлу с расширением `.ipynb`, откроется страница с тетрадкой (блокнотом).

```

jupyter Сборник статей... Минимально достаточный арсенал средств библиотеки pandas (unsaved changes) Logout
File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3
In [1]: import pandas as pd
df = pd.read_csv('Data/sample_data2.csv', sep='\t', index_col=0)
df
Out[1]:
   state color favorite food age height score count
Jane  NY  blue      Steak  30   165   4.6    10
Niko  TX  green     Lamb   2    70   8.3     4
Aaron FL  red      Mango  12   120   9.0     3
Penelope AL white   Apple  4    80   3.3    12
Dean  AK  gray     Cheese  32   180   1.8     8
Christina TX black  Melon  33   172   9.5    99
Cornelia TX  red     Beans  69   150   2.2    44
In [2]: df['state']
Out[2]: Jane      NY
Niko      TX
Aaron     FL
Penelope  AL
Dean      AK
Christina TX
Cornelia  TX
Name: state, dtype: object

```

Рис. 5 Тетрадка Jupyter

Отображаемая тетрадка представляет собой HTML-документ, который был создан Jupyter и IPython. Он состоит из нескольких ячеек, которые могут быть одного из трех типов: **Code** (активный программный код), **Markdown** (текст, поясняющий код, более развернутый, чем комментарий), **Raw NBConvert** (пассивный программный код).

Jupyter запускает ядро IPython для каждой тетрадки. Ячейки, содержащие код Python, выполняются внутри этого ядра, и результаты добавляются в тетрадку в формате HTML. Двойной щелчок по любой из этой ячеек позволит отредактировать ее. По завершении редактирования содержимого ячейки нажмите **Shift+Enter**, после чего Jupyter/IPython проанализирует содержимое и отобразит результаты.



Рис. 6 Панель инструментов в Jupyter

Панель инструментов в верхней части браузера предоставляет ряд возможностей по работе с тетрадкой. К ним относятся добавление, удаление и перемещение ячеек вверх и вниз в тетрадке. Также доступны команды для запуска ячеек, перезапуска ячеек и перезапуска основного ядра IPython.

Чтобы создать новую тетрадку, перейдите в меню **File | New Notebook | Python 3**:



Рис. 7 Создание новой тетрадки в Jupyter

Страница новой тетрадки будет создана в новой вкладке браузера. Ее имя по умолчанию будет `Untitled`.

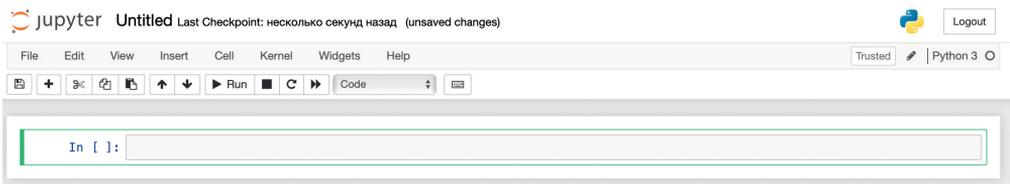


Рис. 8 Новая тетрадка в Jupyter

Тетрадка состоит из одной ячейки Code, которая готова к вводу программного кода Python. Введите `1 + 1` в ячейку и нажмите **Shift+Enter** для выполнения.

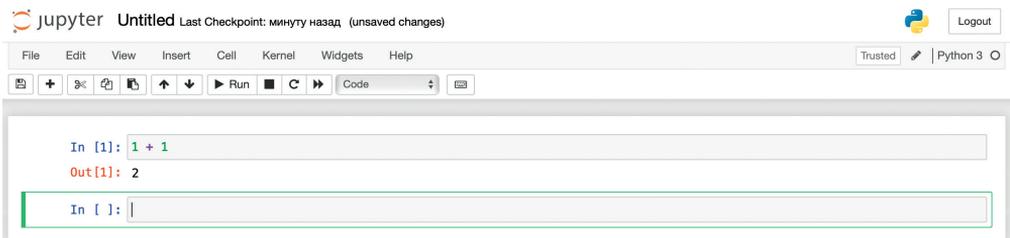


Рис. 9 Операции с ячейками в Jupyter

Ячейка выполнена, и результат показан как `Out[1]:`. Jupyter также создал новую ячейку, чтобы вы могли снова ввести код или разметку.

В ячейке Markdown мы можем вводить и форматировать текст.

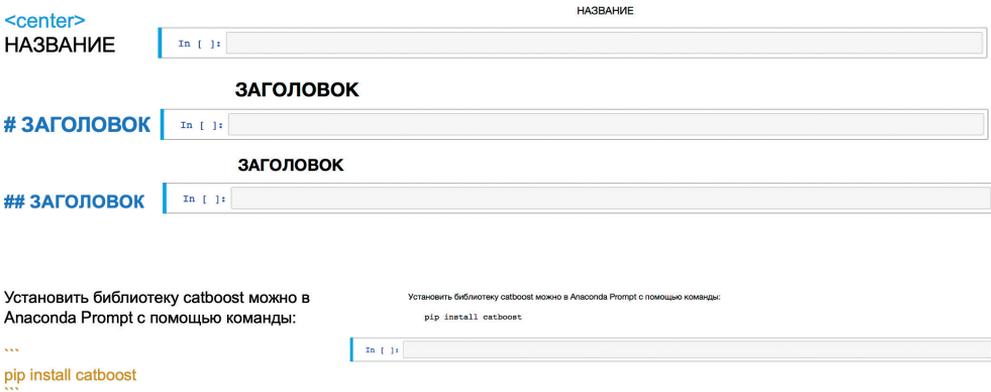


Рис. 10 Редактирование текста в Jupyter

Полезно знать магические функции Jupyter Notebook. Все magic-функции (их еще называют magic-командами) начинаются со знака %, если функция применяется к одной строке, и `%%`, если применяется ко всей ячейке Jupyter.

Чтобы получить представление о времени, которое потребуется для выполнения функции, приведенной выше, мы можем воспользоваться magic-функциями `%timeit`, `%%timeit`, `%time`, `%%time`, созданными специально для работы с тетрадками Jupyter. `%time` однократно запускает строку кода. `%timeit` выполняет строку кода несколько раз, а затем выдает среднее значение. `%%time` однократно запускает блок кода в ячейке. `%%timeit` выполняет блок кода в ячейке несколько раз, а затем выдает среднее значение.

```
%time sum(range(100))
```

```

CPU times: user 4 µs, sys: 0 ns, total: 4 µs
Wall time: 5.96 µs

```

```
4950
```

```
%timeit sum(range(100))
```

```
753 ns ± 22.5 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
# импортируем необходимые библиотеки
```

```
import numpy as np
```

```
# создаем массивы значений
```

```
x = np.random.randn(10000000)
```

```
y = np.random.randn(10000000)
```

```
%%time
```

```

nx = len(x)
result = 0.0
count = 0
for i in range(nx):
    result += x[i] - y[i] count += 1
    result / count

```

CPU times: user 3.37 s, sys: 6.55 ms, total: 3.38 s  
Wall time: 3.38 s

-0.00017262501625817198

```
%%timeit
```

```

nx = len(x)
result = 0.0
count = 0
for i in range(nx):
    result += x[i] - y[i] count += 1
    result / count

```

2.8 s ± 75.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

`%matplotlib notebook` и `%matplotlib inline` позволяют выводить графики непосредственно в тетрадке.

На экранах с высоким разрешением типа Retina графики в тетрадках Jupyter по умолчанию выглядят размытыми, поэтому для улучшения резкости используйте `%config InlineBackend.figure_format = 'retina'` ПОСЛЕ `%matplotlib inline`.

*# импортируем необходимые библиотеки*

```

import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

```

*# загружаем данные*

```

data = pd.read_csv('Data/Visualizations.csv',
                  encoding='cp1251', sep=';')

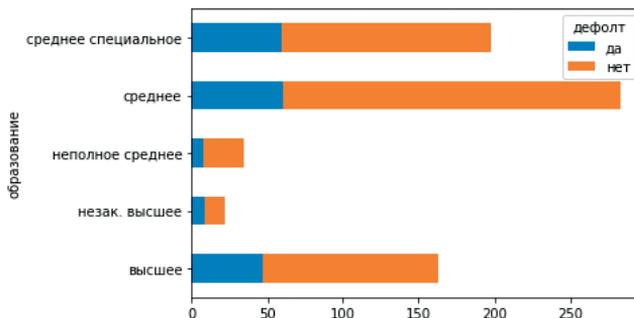
```

*# строим состыкованную столбиковую диаграмму*

```

pd.crosstab(data['образование'], data['дефолт']).plot.barh(
    stacked=True);

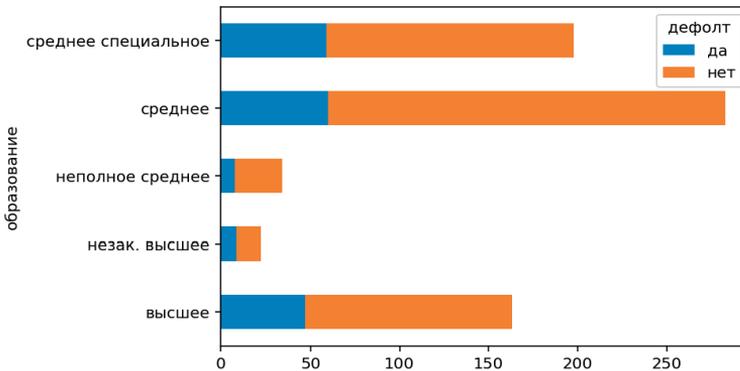
```



Видим, что наш график выглядит размытым.

```
# включаем режим 'retina', если у вас экран Retina
%config InlineBackend.figure_format = 'retina'
```

```
# снова строим состыкованную столбиковую диаграмму
pd.crosstab(data['образование'], data['дефолт']).plot.barh(
    stacked=True);
```



## Задача с собеседования (теория вероятности)

1. В урне находится 15 белых, 5 красных и 10 черных шаров. Наугад извлекается 1 шар. Найти вероятность того, что он будет: а) белым; б) красным; в) черным.

## 3. NumPy

### 3.1. СОЗДАНИЕ МАССИВОВ NUMPY

NumPy (произносится как нампай) – это один из основных пакетов для вычислений в Python. Он содержит функциональные возможности для работы с многомерными массивами и различными математическими функциями.

Основа NumPy – это объект `ndarray`,  $n$ -мерный массив. В Python массив NumPy – это базовая структура данных. Библиотека `scikit-learn`, с помощью которой мы будем строить модели, требует, чтобы данные были записаны в виде массивов NumPy. Датафреймы `pandas`, с которыми мы познакомимся позднее, также будут внутренне преобразованы библиотекой `scikit-learn` в массивы NumPy. Массивы похожи на списки Python, за исключением того, что элементы массива должны иметь одинаковый тип данных, как `float` и `int`. С массивами можно проводить числовые операции с большим объемом информации в разы быстрее и, главное, намного эффективнее, чем со списками. При работе с массивами NumPy полезно помнить, что индексация элементов начинается с 0. Массив NumPy можно создать с помощью функции `array()`. Она имеет общий вид:

```
numpy.array(object, dtype=None, copy=True, order='K', ndmin=0)
```

Параметр	Предназначение
object	Задаёт объект, подобный массиву. Список или кортеж, а также любая функция или метод объекта, возвращающие список или кортеж
dtype	Определяет тип данных выходного массива
copy	Задаёт копирование объекта
order	Определяет, в каком порядке массивы должны храниться в памяти: строчном C-стиле или столбчатом Fortran-стиле. Если объект не является массивом NumPy, то созданный массив будет находиться в памяти в строковом C-порядке; если указать значение 'F', то будет храниться в столбчатом Fortran-порядке. Если объект – это массив NumPy, то в зависимости от флага происходит следующее:

Порядок	нет копирования	copy=True
'K'	сохраняет порядок исходного массива	сохраняет C-порядок или F-порядок, либо устанавливает самый близкий по структуре
'A'	сохраняет порядок исходного массива	установит F-порядок, если массив по стилю похож на столбчатый Fortran-стиль, в противном случае будет задан C-порядок
'C'	C-порядок	C-порядок
'F'	F-порядок	F-порядок

По умолчанию задано значение 'K'

ndmin Определяет минимальное количество измерений результирующего массива

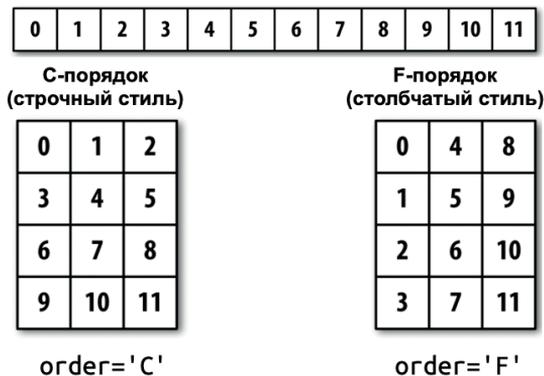


Рис. 11 Порядок хранения массивов в памяти

Импорт всех имен из большого пакета, каким является NumPy (`from numpy import *`), считается среди разработчиков на Python дурным тоном, поэтому

с помощью `np.aggau()` мы указываем, что нас интересует функция `aggau()` библиотеки NumPy, которая у нас импортирована как `np`. Далее все функции NumPy мы будем обозначать `np.имя_функции`.

```
# импортируем библиотеку numpy
```

```
import numpy as np
```

```
# создаем массив NumPy
```

```
a = np.aggau([1, 4, 5, 8], float)  – эквивалентно → a = np.aggau(object=[1, 4, 5, 8],  
print("массив NumPy:\n{}".format(a))                                dtype=float)
```

```
массив NumPy:
```

```
[1. 4. 5. 8.]
```

↙  
Первый элемент будет иметь индекс 0

С помощью функции `type()` убедимся, что перед нами – массив NumPy. Функция `type()` определяет тип переданного аргумента.

```
# выясняем тип объекта
```

```
type(a)
```

```
numpy.ndarray
```

Поскольку NumPy ориентирован прежде всего на численные расчеты, тип данных, если он не указан явно, во многих случаях предполагается `float64`. `float64` – это числовой тип данных в NumPy, который используется для хранения чисел с плавающей точкой двойной точности. Стандартное значение с плавающей точкой двойной точности (хранящееся во внутреннем представлении объекта Python типа `float`) занимает 8 байт, или 64 бита. Поэтому соответствующий тип в NumPy называется `float64`.

Свойство `dtype` как раз возвращает тип значений, хранящихся в массиве.

```
# смотрим тип значений, хранящихся в массиве
```

```
a.dtype
```

```
dtype('float64')
```

**Таблица 1** Список некоторых поддерживаемых NumPy типов данных

Функция	Код типа	Описание
<code>int8, uint8</code>	<code>i1, u1</code>	Знаковое и беззнаковое 8-разрядное (1 байт) целое
<code>int16, uint16</code>	<code>i2, u2</code>	Знаковое и беззнаковое 16-разрядное (2 байта) целое
<code>int32, uint32</code>	<code>i4, u4</code>	Знаковое и беззнаковое 32-разрядное (4 байта) целое
<code>int64, uint64</code>	<code>i8, u8</code>	Знаковое и беззнаковое 64-разрядное (8 байта) целое
<code>float16</code>	<code>f2</code>	С плавающей точкой половинной точности
<code>float32</code>	<code>f4</code>	Стандартный тип с плавающей точкой одинарной точности. Совместим с типом <code>C float</code>

Функция	Код типа	Описание
float64	f8 или d	Стандартный тип с плавающей точкой двойной точности. Совместим с типом C double и с типом Python float
float128	f16	C плавающей точкой расширенной точности
complex64, complex128, complex256	c8, c16, c32	Комплексные числа, вещественная и мнимая части которых представлены соответственно типами float32, float64 и float128
bool	?	Булев тип, способный хранить значения True и False
object	O	Тип объекта Python
string_	S	Тип строки фиксированной длины (1 байт на символ). Например, строка длиной 10 имеет тип S10
unicode_	U	Тип Unicode-строки фиксированной длины (количество байтов на символ зависит от платформы). Семантика такая же, как у типа string_ (например, U10)

На рисунке ниже показана иерархия класса dtype в Python.

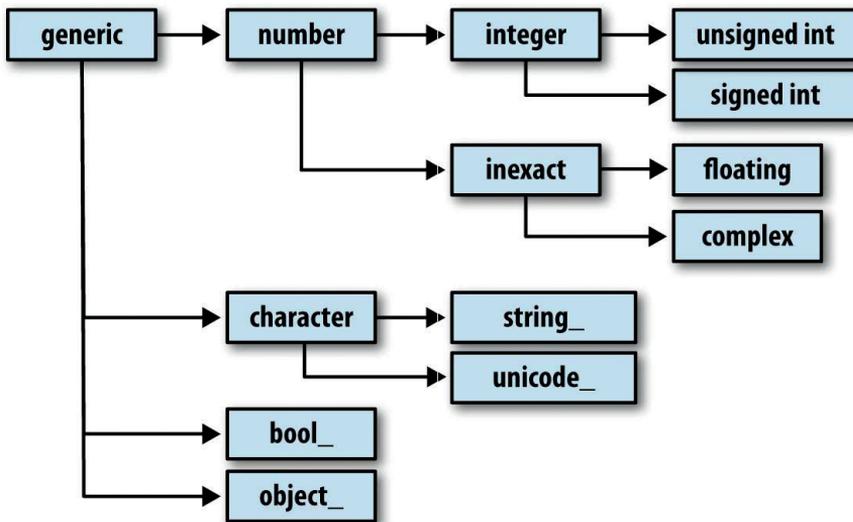


Рис. 12 Иерархия класса dtype в NumPy

Ранее мы создали массив NumPy с помощью функции `np.array()`. Но мы можем использовать и другие функции для создания массивов NumPy, например `np.asarray()`. Она преобразует входные данные в `ndarray`, но не копирует, если на вход уже подан `ndarray`.

```
# воспользуемся функцией asarray()
a = np.asarray([1, 4, 5, 8], float)
a
```

```
array([1., 4., 5., 8.]
```

Функция `np.zeros()` генерирует массив, состоящий из одних нулей, форма массива и тип значений определяются параметрами `shape` и `dtype`.

```
# создаем одномерный массив, состоящий из 5 нулей
Z = np.zeros(5)
Z
array([0., 0., 0., 0., 0.]
```

Функция `np.ones()` генерирует массив, состоящий из одних единиц, форма массива и тип значений определяются параметрами `shape` и `dtype`.

```
# создаем одномерный массив, состоящий из 5 единиц
O = np.ones(5)
O
array([1., 1., 1., 1., 1.]
```

Функции `np.eye()` и `np.identity()` создают единичную квадратную матрицу  $N \times N$  (элементы на главной диагонали равны 1, все остальные – 0).

```
# создаем единичную квадратную матрицу 3x3 (элементы
# на главной диагонали равны 1, все остальные - 0)
E = np.eye(3)
E
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

```
# создаем единичную квадратную матрицу 3x3 (элементы
# на главной диагонали равны 1, все остальные - 0)
I = np.identity(3)
I
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

С помощью метода `.fill()` мы можем заполнить массив одинаковым значением. Операция выполняется на месте.

```
# заполняем значения нулями
I.fill(0)
I
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Функция `np.empty()` создает новые массивы, выделяя под них память, но, в отличие от `np.ones()` и `np.zeros()`, не инициализирует элементы. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (то есть от того мусора, что в ней хранится).

```
# pr.empty() не инициализирует элементы
pr.empty(2)

array([-5.73021895e-300, 8.04338871e-320])
```

## 3.2. ОБРАЩЕНИЕ К ЭЛЕМЕНТАМ МАССИВА

Вернемся к нашему массиву `a`.

```
# вернемся к массиву a
a

array([1., 4., 5., 8.])
```

К элементам массива можно получить доступ, создавать срезы и обращаться с ними так же, как со списками.

```
# смотрим первые 2 элемента
a[:2]

array([1., 4.])
```

```
# смотрим элемент с индексом 3
# (индексация с 0)
a[3]

8.0
```

```
# присваиваем первому элементу (элементу
# с индексом 0) значение 5.
a[0] = 5.
# смотрим результат
a

array([5., 4., 5., 8.])
```

Массивы могут быть многомерными. В отличие от списков, доступ к различным осям можно получить с помощью запятых внутри квадратных скобок. Здесь у нас пример двумерного массива (т.е. матрицы):

```
# создаем двумерный массив
a = pr.array([[1, 2, 3],
             [4, 5, 6]],
            float)
```

Ось 1 (ось столбцов)

		0	1	2
Ось 0 (ось строк)	0	1.	2.	3.
	1	4.	5.	6.

```
a

array([[1., 2., 3.],
       [4., 5., 6.]])
```

Давайте обратимся к элементу с индексом 0 по оси строк и оси столбцов.

```
# обратимся к элементу с индексом 0
# по оси строк и оси столбцов
a[0, 0]
```

1.0

		Ось 1 (ось столбцов)		
		0	1	2
Ось 0 (ось строк)	0	1.	2.	3.
	1	4.	5.	6.

Давайте обратимся к элементу с индексом 0 по оси строк и индексом 1 по оси столбцов.

```
# обратимся к элементу с индексом 0
# по оси строк и оси столбцов
a[0, 1]
```

2.0

		Ось 1 (ось столбцов)		
		0	1	2
Ось 0 (ось строк)	0	1.	2.	3.
	1	4.	5.	6.

Срезы многомерных массивов можно создавать точно так же, как срезы одномерных массивов. Спецификация среза работает как фильтр для заданного измерения. Одиночный символ : для измерения задает использование всех элементов в этом измерении. Индекс с запятой перед символом : будет означать отбор строки с соответствующим индексом. Индекс с запятой после символа : будет означать отбор столбца с соответствующим индексом. Отрицательный знак перед индексом будет обозначать позицию с конца (-1 будет обозначать первую с конца, т.е. последнюю строку/столбец).

Давайте обратимся ко второй строке, т.е. строке с индексом 1.

```
# обращаемся ко второй строке
a[1, :]
```

array([4., 5., 6.])

		Ось 1 (ось столбцов)		
		0	1	2
Ось 0 (ось строк)	0	1.	2.	3.
	1	4.	5.	6.

Теперь обратимся к третьему столбцу, т.е. столбцу с индексом 2.

```
# обращаемся к третьему столбцу
a[:, 2]
```

array([3., 6.])

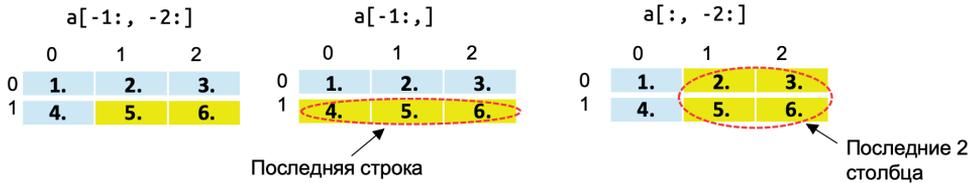
		Ось 1 (ось столбцов)		
		0	1	2
Ось 0 (ось строк)	0	1.	2.	3.
	1	4.	5.	6.

А теперь сначала отберем последнюю строку и в ней отберем значения по двум последним столбцам.

```
# сначала отбираем последнюю строку
# и в ней отбираем значения по двум
# последним столбцам
a[-1:, -2:]
```

array([[5., 6.]])

Происходящее «под капотом» можно визуализировать следующим образом.



Подход Python к использованию именованных переменных работает и для массивов. Метод `.copy()` используется для создания новой отдельной копии массива в памяти.

```
# создаем массив a
a = np.array([1, 2, 3], float)
# присваиваем массиву a имя b
b = a
# создаем c - копию массива a
c = a.copy()
# первому элементу массива a
# присваиваем значение 0
a[0] = 0

# смотрим массив b
b
array([0., 2., 3.])

# смотрим массив c
c
array([1., 2., 3.])
```

### 3.3. ПОЛУЧЕНИЕ КРАТКОЙ ИНФОРМАЦИИ О МАССИВЕ

Свойство `shape` возвращает двухэлементный кортеж с количеством строк и столбцов массива.

```
# смотрим информацию о количестве
# строк и столбцов в массиве
a.shape

(2, 3)
```

Свойство `size` возвращает количество элементов массива.

```
# смотрим информацию о количестве
# элементов массива
a.size

6
```

Свойство `ndim` возвращает размерность массива.

```
# смотрим размерность массива
```

```
a.ndim
```

```
1
```

Функция `len()` возвращает длину первой оси.

```
# смотрим длину первой оси
```

```
len(a)
```

```
2
```

Оператор `in` используется для проверки на наличие элемента в массиве.

```
# проверяем, есть ли значение 2 в массиве
```

```
2 in a
```

```
True
```

```
# проверяем, есть ли значение 0 в массиве
```

```
0 in a
```

```
False
```

### 3.4. ИЗМЕНЕНИЕ ФОРМЫ МАССИВА

Можно изменить форму массива с помощью кортежей, задающих новые размерности. В следующем примере мы превратим одномерный массив с 10 элементами в двумерный массив, у которого первая ось будет содержать 5 элементов, а вторая ось – 2 элемента.

```
# создаем одномерный массив с 10 элементами
```

```
a = np.aggay(range(10), float)
```

```
a
```

```
aggay([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Обратите внимание, что здесь мы воспользовались универсальной функцией `range()`, которая используется для создания списков, содержащих арифметическую прогрессию. Вы ее еще часто увидите в циклах `for`.

Итак, выполняем преобразование. В этом нам поможет функция `np.reshape()`. Она принимает два аргумента: массив, который нужно преобразовать в массив нужной формы, и кортеж с двумя целочисленными значениями, определяющий форму (соответственно параметры `a` и `newshape`). Если передан кортеж с одним целочисленным значением, то результатом будет одномерный массив данной длины. Если значением будет `-1`, то значение определяется длиной массива и оставшимися размерностями.

```
# превратим одномерный массив с 10 элементами
```

```
# в двумерный массив, первая ось будет содержать
```

```
# 5 элементов, а вторая ось – 2 элемента
```

```
a = np.reshape(a, (5, 2))
a
```

двухэлементный кортеж (количество строк, количество столбцов)

```
array([[0., 1.],
       [2., 3.],
       [4., 5.],
       [6., 7.],
       [8., 9.]])
```

Теперь выполним обратное преобразование.

```
# преобразовываем обратно
```

```
a = np.reshape(a, 10)
a
```

```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]])
```

Эквивалентом функции `np.reshape()` будет метод `.reshape()`.

```
# снова превратим одномерный массив с 10 элементами
```

```
# в двумерный массив, первая ось будет содержать
```

```
# 5 элементов, а вторая ось - 2 элемента,
```

```
# только уже с помощью метода .reshape()
```

```
a = a.reshape((5, 2))
a
```

```
array([[0., 1.],
       [2., 3.],
       [4., 5.],
       [6., 7.],
       [8., 9.]])
```

Из массивов можно создавать списки с помощью метода `.tolist()` и функции `list()`.

```
# создаем массив
```

```
a = np.array([1, 2, 3], float)
```

```
# преобразовываем в список
```

```
# с помощью метода .tolist()
```

```
a.tolist()
```

```
[1.0, 2.0, 3.0]
```

```
# преобразовываем в список
```

```
# с помощью функции list()
```

```
list(a)
```

```
[1.0, 2.0, 3.0]
```

Можно преобразовать массив с сырыми данными в бинарную строку (т.е. в машиночитаемый формат), используя метод `.tobytes()` (он пришел на сме-

ну методу `.tostring()`). Функция `np.frombuffer()` работает для обратного преобразования. Раньше использовалась функция `np.fromstring()`, но в ряде случаев она вела себя непредсказуемо. Эти операции иногда полезны для сохранения большого объема данных в файлах, которые могут быть считаны в будущем.

```
# создаем массив
a = np.array([1, 2, 3], float)
# переводим массив в бинарную строку
s = a.tobytes()
# смотрим результат
s

b'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x08@'
```

*# выполняем обратное преобразование*  
`np.frombuffer(s)`

```
array([1., 2., 3.]
```

Транспонирование массивов также возможно, при этом создается новый массив с двумя измененными осями.

```
# создаем массив из 6 элементов,
# 2 строки и 3 столбца
a = np.array(range(6), float).reshape((2, 3))
a

array([[0., 1., 2.],
       [3., 4., 5.]])

# выполняем транспонирование,
# получаем 3 строки и 2 столбца
a_t = a.transpose()
# смотрим транспонированный массив
a_t

array([[0., 3.],
       [1., 4.],
       [2., 5.]])

# можно покороче
a_T = a.T
a_T

array([[0., 3.],
       [1., 4.],
       [2., 5.]])
```

Многомерный массив можно преобразовать в одномерный с помощью методов `.flatten()` и `.ravel()`. Однако между ними есть различия. Метод `.ravel()` возвращает ссылку/представление исходного массива. Если вы модифицируете выходной массив, то и исходный массив изменяется. Метод `.flatten()` возвращает копию исходного массива. Если вы меняете значение выходного массива, исходный массив остается неизменным. Метод `.ravel()` быстрее, чем метод `.flatten()`.

```

# создаем многомерный массив
a = np.array([[1, 2, 3],
              [4, 5, 6]],
              float)

a

array([[1., 2., 3.],
       [4., 5., 6.]])

# преобразовываем многомерный массив в одномерный
a_f = a.flatten()
a_f

array([1., 2., 3., 4., 5., 6.])

# модифицируем выходной массив
a_f[0] = 0

# смотрим выходной и исходный массивы
print(a_f)
print("")
print(a)

[0. 2. 3. 4. 5. 6.]

[[1. 2. 3.]
 [4. 5. 6.]]

# преобразовываем многомерный массив в одномерный
a_r = np.ravel(a)
a_r

array([1., 2., 3., 4., 5., 6.])

# модифицируем выходной массив
a_r[0] = 0

# смотрим выходной и исходный массивы
print(a_r)
print("")
print(a)

[0. 2. 3. 4. 5. 6.]

[[0. 2. 3.]
 [4. 5. 6.]]

```

## 3.5. КОНКАТЕНАЦИЯ МАССИВОВ

Два или больше массивов можно сконкатенировать (соединить) при помощи функции `np.concatenate()`. Мы передаем в функцию кортеж конкатенируемых массивов.

```

# создаем массивы для конкатенации
a = np.array([1, 2], float)

```

```

b = np.array([3, 4, 5, 6], float)
c = np.array([7, 8, 9], float)
# выполняем конкатенацию массивов,
# создавая новый массив
d = np.concatenate((a, b, c))
# смотрим получившийся массив
d
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])

```

Если массив не является одномерным, можно задать ось, по которой будет происходить конкатенация. По умолчанию (если не задавать значения оси) конкатенация будет происходить по оси строк.

```

# создаем массивы для конкатенации
a = np.array([[1, 2], [3, 4]], float)
b = np.array([[5, 6], [7, 8]], float)
# выполняем конкатенацию массивов, создавая новый массив
# (по умолчанию выполняется конкатенация по оси строк)
c = np.concatenate((a,b))

```

```

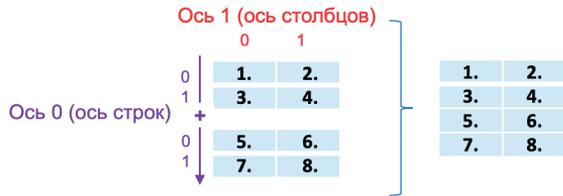
# смотрим получившийся массив
c

```

```

array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])

```



Это эквивалентно варианту конкатенации по оси строк, когда мы явно указали `axis=0`.

```

# выполняем конкатенацию массивов
# по оси строк, указав axis=0
c = np.concatenate((a, b), axis=0)
# смотрим получившийся массив
c

```

```

array([[1., 2.],
       [3., 4.],
       [5., 6.],
       [7., 8.]])

```

Теперь выполним конкатенацию по оси столбцов, указав `axis=1`.

```

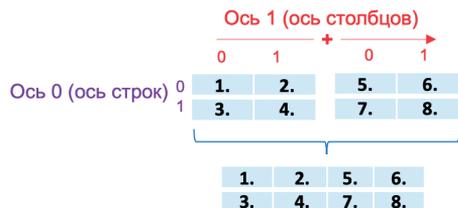
# выполняем конкатенацию массивов
# по оси столбцов, указав axis=1
d = np.concatenate((a, b), axis=1)
# смотрим получившийся массив
d

```

```

array([[1., 2., 5., 6.],
       [3., 4., 7., 8.]])

```



Существуют и другие функции, выполняющие конкатенацию.

Функция `np.vstack()` конкатенирует массивы вертикально (по оси строк). Она эквивалентна конкатенации вдоль первой оси (оси строк). Одномерные массивы соединяются построчно в двумерные массивы.

```
# выполняем конкатенацию одномерных
# массивов с помощью np.vstack()
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
c = np.vstack((a, b))
c
```

```
array([[1, 2, 3],
       [2, 3, 4]])
```

```
# конкатенируем двумерные массивы
# с помощью np.vstack()
a = np.array([[1, 2, 3],
              [3, 4, 6]])
b = np.array([[2, 3, 4],
              [5, 7, 1]])
c = np.vstack((a, b))
c
```

```
array([[1, 2, 3],
       [3, 4, 6],
       [2, 3, 4],
       [5, 7, 1]])
```



Аналогом функции `np.vstack()` является `np.row_stack()`.

```
# выполняем конкатенацию одномерных массивов
# с помощью np.row_stack()
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
c = np.row_stack((a, b))
c
```

```
array([[1, 2, 3],
       [2, 3, 4]])
```

```
# конкатенируем двумерные массивы
# с помощью np.row_stack()
a = np.array([[1, 2, 3],
              [3, 4, 6]])
b = np.array([[2, 3, 4],
              [5, 7, 1]])
c = np.row_stack((a, b))
c
```

```
array([[1, 2, 3],
       [3, 4, 6],
       [2, 3, 4],
       [5, 7, 1]])
```

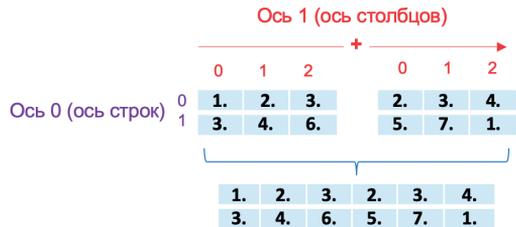
Функция `np.hstack()` конкатенирует массивы горизонтально (по оси столбцов). Она эквивалентна конкатенации вдоль второй оси (оси столбцов). Одномерные массивы просто соединяются.

```
# выполняем конкатенацию одномерных
# массивов с помощью np.hstack()
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
c = np.hstack((a, b))
c
```

```
array([1, 2, 3, 2, 3, 4])
```

```
# конкатенируем двумерные массивы
# с помощью np.hstack()
a = np.array([[1, 2, 3],
              [3, 4, 6]])
b = np.array([[2, 3, 4],
              [5, 7, 1]])
c = np.hstack((a, b))
c
```

```
array([[1, 2, 3, 2, 3, 4],
       [3, 4, 6, 5, 7, 1]])
```



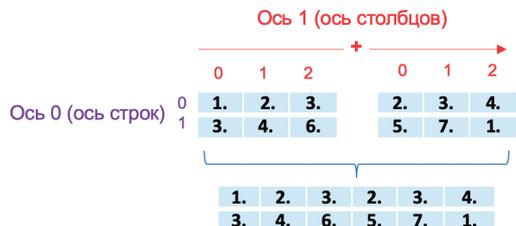
Функция `np.column_stack()` берет одномерные массивы, превращает их в столбцы двумерного массива и соединяет в двумерный массив. Двумерные массивы соединяются так же, как при использовании `np.hstack()`.

```
# выполняем конкатенацию одномерных
# массивов с помощью np.column_stack()
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
c = np.column_stack((a, b))
c
```

```
array([[1, 2],
       [2, 3],
       [3, 4]])
```

```
# конкатенируем двумерные массивы
# с помощью np.column_stack()
a = np.array([[1, 2, 3],
              [3, 4, 6]])
b = np.array([[2, 3, 4],
              [5, 7, 1]])
c = np.column_stack((a, b))
c
```

```
array([[1, 2, 3, 2, 3, 4],
       [3, 4, 6, 5, 7, 1]])
```



## 3.6. ФУНКЦИИ МАТЕМАТИЧЕСКИХ ОПЕРАЦИЙ, ЗНАКОМСТВО С ПРАВИЛАМИ ТРАНСЛИРОВАНИЯ

При использовании стандартных математических операций с массивами выполняется поэлементный принцип: арифметические операции, в которых участвует скаляр, применяются к каждому элементу массива. В свою очередь, это означает, что массивы должны быть одинакового размера во время сложения, вычитания и тому подобных операций.

```
# создаем массивы для выполнения арифметических операций
a = np.array([1, 2, 3], float)
b = np.array([5, 2, 6], float)

# выполняем сложение          # выполняем вычитание          # выполняем умножение
a + b                          a - b                          a * b

array([6., 4., 9.])          array([-4., 0., -3.])          array([ 5., 4., 18.])

# выполняем деление          # выполняем возведение в степень
a / b                          b ** a

array([0.2, 1. , 0.5])      array([ 5., 4., 216.])
```

Для двумерных массивов умножение остается поэлементным и не соответствует умножению матриц. Для этого существуют специальные функции, которые будут рассмотрены позже.

```
# создаем массивы для выполнения арифметических операций
a = np.array([[1, 2],
              [3, 4]],
              float)
b = np.array([[2, 0],
              [1, 3]],
              float)

# выполняем умножение
a * b

array([[ 2.,  0.],
       [ 3., 12.]])
```

Однако если массивы не совпадают по размерности (количеству осей), к ним будет применено *укладывание*, или *транслирование (broadcasting)*, для выполнения математических операций. Транслирование в библиотеке NumPy следует строгому набору правил, определяющему взаимодействие двух массивов.

Если в каком-либо измерении размеры массивов различаются и ни один не равен 1, генерируется ошибка.

Если размерности двух массивов отличаются, форма массива с меньшей размерностью дополняется единицами с ведущей (левой) стороны.

Если форма двух массивов не совпадает в каком-то измерении, массив с формой, равной 1 в данном измерении, растягивается вплоть до соответствия форме другого массива.

Создаем массивы, отличающиеся по размеру: в одном – 3 элемента, в другом – 2 элемента. При этом ни один не равен 1.

```
# при несоответствии размеров массивов
# выбрасываются ошибки
a = np.array([1, 2, 3], float)
b = np.array([4, 5], float)
a + b
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-62-83283f8a8602> in <module>
      2 a = np.array([1, 2, 3], float)
      3 b = np.array([4, 5], float)
----> 4 a + b
```

**ValueError:** operands could not be broadcast together with shapes (3,) (2,)

Создаем массивы, отличающиеся по размерности: один массив является двумерным, а второй – одномерным.

```
# создаем массивы, различающиеся по размерности
a = np.array([[1, 2], [3, 4], [5, 6]], float)
b = np.array([-1, 3], float)

# выполняем сложение
a + b
array([[0., 5.],
       [2., 7.],
       [4., 9.]])
```

К массиву *b* применяется *транслирование*, для того чтобы мы могли выполнить математическую операцию. В нашем случае меньший массив *b* будет использован несколько раз (продублирован) для каждой строки массива *a*. В итоге одномерный массив *b* был транслирован в двумерный так, чтобы он соответствовал форме массива *a*.



Библиотека NumPy предлагает большой набор функций для выполнения стандартных математических операций над массивами (функции `np.abs`, `np.sign`, `np.sqrt`, `np.log`, `np.log10`, `np.exp`, `np.sin`, `np.cos`, `np.tan`, `np.arcsin`, `np.arccos`, `np.arctan`, `np.sinh`, `np.cosh`, `np.tanh`, `np.arcsinh`, `np.arccosh` и `np.arctanh`).

```
# создаем массив
a = np.array([1, 4, 9], float)
```

```
# берем квадратный корень
```

```
np.sqrt(a)
```

```
array([1., 2., 3.]
```

Функция `np rint()` округляет до ближайшего целого числа по общепринятым правилам, функция `np.ceil()` – до ближайшего целого, но только всегда в большую сторону, а функция `np.floor()` – до ближайшего целого, но только всегда в меньшую сторону.

```
# создаем массив
```

```
a = np.array([1.1, 1.5, 1.9], float)
```

```
# функция rint() округляет до ближайшего целого
```

```
# числа по общепринятым правилам
```

```
np rint(a)
```

```
array([1., 2., 2.]
```

```
# функция ceil() округляет до ближайшего целого,
```

```
# но только всегда в большую сторону
```

```
np.ceil(a)
```

```
array([2., 2., 2.]
```

```
# функция floor() округляет до ближайшего целого,
```

```
# но только всегда в меньшую сторону
```

```
np.floor(a)
```

```
array([1., 1., 1.]
```

В NumPy включены две важные математические константы.

```
# выведем число пи и e - основание натурального логарифма
```

```
print(np.pi)
```

```
print(np.e)
```

```
3.141592653589793
```

```
2.718281828459045
```

Подробнее поговорим о функциях и методах, выполняющих базовые операции над массивами.

```
# создаем массив
```

```
a = np.array([2, 4, 3], float)
```

Например, мы можем просуммировать или перемножить элементы массива с помощью `.sum()/np.sum()` и `.prod()/np.prod()` соответственно.

```
# суммируем
```

```
print(a.sum())
```

```
print(np.sum(a))
```

9.0

9.0

```
# перемножаем
```

```
print(a.prod())
```

```
print(np.prod(a))
```

24.0

24.0

С помощью `.mean()/mean()`, `.std()/np.std()` и `.var()/np.var()` мы можем вычислить среднее, стандартное отклонение и дисперсию – квадрат стандартного отклонения.

```
# создаем массив
```

```
a = np.aggay([2, 1, 9], float)
```

```
# вычисляем среднее
```

```
print(a.mean())
```

```
print(np.mean(a))
```

4.0

4.0

```
# вычисляем стандартное отклонение
```

```
print(a.std())
```

```
print(np.std(a))
```

3.559026084010437

3.559026084010437

```
# вычисляем дисперсию
```

```
print(a.var())
```

```
print(np.var(a))
```

12.666666666666666

12.666666666666666

Мы можем найти минимум и максимум в массиве.

```
# находим минимум
```

```
print(a.min())
```

```
print(np.min(a))
```

1.0

1.0

```
# находим максимум
```

```
print(a.max())
```

```
print(np.max(a))
```

9.0

9.0

Функции `np.argmin()` и `np.argmax()` возвращают индекс минимального или максимального элемента.

```
# находим индекс минимального элемента
```

```
np.argmin(a)
```

```
1
```

```
# находим индекс максимального элемента
```

```
np.argmax(a)
```

```
2
```

	0	1	2
np.argmin(a)	2.	1.	9.

	0	1	2
np.argmax(a)	2.	1.	9.

При работе с многомерными массивами у функции будет дополнительный параметр `axis` (по умолчанию задано значение `None` – вычисляется среднее по всем элементам массива, как если бы он был плоским массивом). В зависимости от его значения функция выполнит операцию по заданной оси и поместит результаты выполнения в возвращаемом массиве. Можно сказать, что операции по оси строк – это операции, выполняемые по вертикали, операции по оси столбцов – это операции, выполняемые по горизонтали.

Посмотрим это на примере функции `np.mean()`.

```
# создаем двумерный массив
```

```
a = np.array([[0, 2],
              [3, -1],
              [3, 5]],
             float)
```

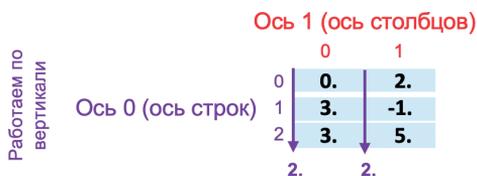
```
a
```

```
array([[ 0.,  2.],
       [ 3., -1.],
       [ 3.,  5.]])
```

```
# вычисляем среднее по оси строк (оси 0)
```

```
np.mean(a, axis=0)
```

```
array([2., 2.])
```



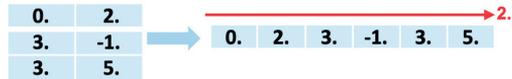
```
# вычисляем среднее по оси столбцов (оси 1)
```

```
np.mean(a, axis=1)
```

```
array([1., 1., 4.])
```



```
# вычисляем среднее по умолчанию
# эквивалентно np.mean(a, axis=None)
np.mean(a)
```



2.0

## 3.7. ОБРАБОТКА ПРОПУСКОВ

Теперь мы создадим двумерный массив с пропущенными значениями.

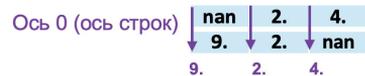
```
# создаем двумерный массив
# с пропущенными значениями
a = np.array([[np.nan, 2, 4],
              [9, 2, np.nan]],
             float)
a
```

```
array([[nan, 2., 4.],
       [9., 2., nan]])
```

Для вычисления среднего при наличии пропусков можно воспользоваться функцией `np.nanmean()`. Она позволяет вычислить среднее, игнорируя пропуски.

```
# вычисляем среднее по оси строк (оси 0)
np.nanmean(a, axis=0)
```

```
array([9., 2., 4.])
```



```
# вычисляем среднее по оси столбцов (оси 1)
np.nanmean(a, axis=1)
```

```
array([3. , 5.5])
```



```
# вычисляем среднее по умолчанию
# эквивалентно np.nanmean(a, axis=None)
np.nanmean(a)
```



4.25

С помощью функции `np.nan_to_num()` можно заменить пропуски определенным значением. Параметр `nan` позволяет задать значение для замены. Сейчас мы заменим все пропуски в массиве значением 5.

```
# создаем двумерный массив
# с пропущенными значениями
a = np.array([[np.nan, 2, 4],
              [9, 2, np.nan]],
             float)
```

```
# заменяем все пропуски значением 5
a = np.nan_to_num(a, nan=5)
a
```

```
aggay([[5.] 2., 4.],
       [9., 2., 5.]])
```

Сейчас с помощью функции `np.nan_to_num()` мы заменим пропуски в отдельном столбце – столбце с индексом 0.

*# заменяем пропуски в столбце с индексом 0*

```
a = np.aggay([[np.nan, 2, 4],
              [9, 2, np.nan]],
             float)
a[:, 0] = np.nan_to_num(a[:, 0], nan=5)
a
```

```
aggay([[5.] 2., 4.],
       [9., 2., nan]])
```

Теперь с помощью функции `np.nan_to_num()` мы заменим пропуски в отдельной строке – строке с индексом 1.

*# заменяем пропуски в строке с индексом 1*

```
a = np.aggay([[np.nan, 2, 4],
              [9, 2, np.nan]],
             float)
a[1, :] = np.nan_to_num(a[1, :], nan=5)
a
```

```
aggay([[nan, 2., 4.],
       [9., 2., 5.]])
```

Теперь выполним замену пропусков по столбцам, при этом у каждого столбца будет свое заменяющее значение. Мы создадим двумерный массив с пропусками, у нас будет две строки и три столбца. Затем создадим словарь, в котором будет три пары «ключ-значение», ключом будет индекс столбца, а значением – значение, на которое нужно заменить пропуск в данном столбце. Для итерирования по столбцам часто применяют цикл `for` в сочетании с функцией `range()`, аргументом будет количество столбцов массива (для этого можно взять второй элемент кортежа, возвращаемого свойством `shape`, например `a.shape[1]`).

*# создаем массив с пропусками*

```
a = np.aggay([[np.nan, 2, 4],
              [9, np.nan, np.nan]],
             float)
a
```

```
aggay([[nan, 2., 4.],
       [9., nan, nan]])
```

*# создаем словарь со значениями для замены*

```
enc_dict = {0: 5, 1: 6, 2: 7}
```

*# по каждому столбцу массива заменяем пропуски*

*# значением из словаря*

```
for col in range(a.shape[1]):
```

```

a[:, col] = np.nan_to_num(a[:, col],
                        nan=enc_dict[col])
a
array([[ 5.,  2.,  4.],
       [ 9.,  6.,  7.]])

```

А сейчас выполним замену пропусков по строкам, у каждой строки будет свое заменяющее значение. Вновь создадим точно такой же двумерный массив с пропусками, у нас опять будет две строки и три столбца. Затем создадим словарь, в котором будет две пары «ключ-значение», ключом будет индекс строки, а значением – значение, на которое нужно заменить пропуск в данной строке. Для итерирования по столбцам опять применяем цикл `for` в сочетании с функцией `range()`, аргументом уже будет количество строк массива (для этого можно взять первый элемент кортежа, возвращаемого свойством `shape`, например `a.shape[0]`).

```

# создаем массив с пропусками
a = np.array([[np.nan, 2, 4],
             [9, np.nan, np.nan]],
            dtype=float)

# создаем словарь со значениями для замены
enc_dict = {0: 5, 1: 10}

# по каждой строке массива заменяем пропуски
# значением из словаря
for col in range(a.shape[0]):
    a[col, :] = np.nan_to_num(a[col, :],
                            nan=enc_dict[col])
a
array([[ 5.,  2.,  4.],
       [ 9., 10., 10.]])

```

Приведем еще ряд полезных функций NumPy.

### 3.8. ФУНКЦИЯ `np.linspace()`

Функция `np.linspace()` возвращает одномерный массив из указанного количества элементов (по умолчанию 50), значения которых равномерно распределены внутри заданного интервала. Она имеет вид:

```
np.linspace(start, stop, num=50, endpoint=True)
```

Параметр	Предназначение
<b>start</b> (вещественное число)	Задаёт начальное значение последовательности
<b>stop</b> (вещественное число)	Задаёт последнее значение последовательности, если для параметра <code>endpoint</code> задано значение <code>True</code> . Если <code>endpoint=False</code> , то данное значение не включается в интервал, при этом значение шага между элементами последовательности изменяется
<b>num</b> (целое положительное число, необязательный)	Определяет количество элементов последовательности (по умолчанию 50)
<b>endpoint</b> ( <code>True</code> или <code>False</code> , необязательный)	Определяет включение последнего значения последовательности ( <code>stop</code> ) в интервал. Если <code>endpoint=True</code> , то значение <code>stop</code> включается в интервал и является последним. В противном случае <code>stop</code> не входит в интервал. По умолчанию <code>endpoint=True</code>
<b>Возвращает</b>	
<b>ndarray</b> (массив Numpy)	Одномерный массив из указанного количества равномерно распределённых элементов

Сейчас мы создадим одномерный массив из 50 элементов, значения которых равномерно распределены внутри интервала от 0 до 1.

```
# создаем одномерный массив из 50 элементов,
# значения которых равномерно распределены
# внутри интервала от 0 до 1
np.linspace(start=0, stop=1)

array([0.          , 0.02040816, 0.04081633, 0.06122449, 0.08163265,
       0.10204082, 0.12244898, 0.14285714, 0.16326531, 0.18367347,
       0.20408163, 0.2244898 , 0.24489796, 0.26530612, 0.28571429,
       0.30612245, 0.32653061, 0.34693878, 0.36734694, 0.3877551 ,
       0.40816327, 0.42857143, 0.44897959, 0.46938776, 0.48979592,
       0.51020408, 0.53061224, 0.55102041, 0.57142857, 0.59183673,
       0.6122449 , 0.63265306, 0.65306122, 0.67346939, 0.69387755,
       0.71428571, 0.73469388, 0.75510204, 0.7755102 , 0.79591837,
       0.81632653, 0.83673469, 0.85714286, 0.87755102, 0.89795918,
       0.91836735, 0.93877551, 0.95918367, 0.97959184, 1.          ])
```

Теперь создадим одномерный массив из 6 элементов, значения которых равномерно распределены внутри интервала от 1 до 6, и посмотрим, как параметр `endpoint` влияет на результат.

```
# создаем массив из 6 элементов, значения которых равномерно
# распределены внутри интервала от 1 до 6,
# включаем/отключаем параметр endpoint
print(np.linspace(start=1, stop=6, num=6, endpoint=True))
print(np.linspace(start=1, stop=6, num=6, endpoint=False))
```

```
[1. 2. 3. 4. 5. 6.]
```

```
[1. 1.83333333 2.66666667 3.5 4.33333333 5.16666667] ← Если endpoint=False, последний элемент не включается
```

### 3.9. ФУНКЦИЯ `np.logspace()`

Функция `np.logspace()` возвращает одномерный массив из указанного количества элементов, значения которых равномерно распределены по логарифмической шкале внутри заданного интервала. Она имеет вид:

```
np.logspace(start, stop, num=50, endpoint=True, base=10.0)
```

Параметр	Предназначение
<b>start</b> (вещественное число)	Задаёт начальное значение последовательности, которое равно $\text{base}^{**} \text{start}$ ( $\text{base}$ в степени $\text{start}$ )
<b>stop</b> (вещественное число)	Задаёт последнее значение последовательности ( $\text{base}^{**} \text{stop}$ ), если для параметра <code>endpoint</code> задано значение <code>True</code> . Если <code>endpoint=False</code> , то данное значение не включается в интервал, при этом значение шага между элементами последовательности изменяется
<b>num</b> (целое положительное число, необязательный)	Определяет количество элементов последовательности (по умолчанию 50)
<b>endpoint</b> ( <code>True</code> или <code>False</code> , необязательный)	Определяет включение последнего значения последовательности ( $\text{base}^{**} \text{stop}$ ) в интервал. Если <code>endpoint=True</code> , то значение $\text{base}^{**} \text{stop}$ включается в интервал и является последним. В противном случае $\text{base}^{**} \text{stop}$ не входит в интервал. По умолчанию <code>endpoint=True</code>
<b>base</b> (вещественное число, обязательный)	Задаёт основание логарифмической шкалы (по умолчанию $\text{base}=10$ )
<b>Возвращает</b>	
<b>ndarray</b> (массив Numpy)	Одномерный массив из указанного количества элементов, значения которых равномерно распределены по логарифмической шкале

Сейчас мы создадим одномерный массив из 20 элементов, значения которых равномерно распределены по логарифмической шкале внутри интервала, стартовым значением которого будет 10 в степени  $-1$  ( $0,1$ ), а последним значением будет 10 в степени  $1$  ( $10$ ).

```
# создадим одномерный массив из 20 элементов, значения которых равномерно
# распределены по логарифмической шкале внутри интервала, стартовым
# значением которого будет 10 в степени -1 (0,1), а последним
# значением будет 10 в степени 1 (10)
np.logspace(start=-1, stop=1, num=20)

array([ 0.1          ,  0.1274275  ,  0.16237767,  0.20691381,  0.26366509,
        0.33598183,  0.42813324,  0.54555948,  0.6951928  ,  0.88586679,
        1.12883789,  1.43844989,  1.83298071,  2.33572147,  2.97635144,
        3.79269019,  4.83293024,  6.15848211,  7.8475997  , 10.          ])
```

На практике с помощью функций `np.linspace()` и `np.logspace()` часто создают массивы значений гиперпараметров для моделей предварительной подготовки данных и моделей машинного обучения.

## 3.10. ФУНКЦИЯ NP.DIGITIZE()

Функция `np.digitize()` возвращает индексы числовых интервалов (бинов), в которые входит каждое значение элементов массива. Она имеет вид:

```
np.digitize(x, bins, right=False)
```

Параметр	Предназначение
<b>x</b> (массив NumPy или объект, подобный массиву)	Задаёт входные данные. Многомерные массивы сжимаются до одной оси
<b>bins</b> (массив NumPy или объект, подобный массиву)	Задаёт одномерный массив, соседние значения которого задают границы полуоткрытых интервалов. Значения должны быть возрастающими
<b>right</b> (True или False, необязательный)	Определяет, какой край интервала включать в интервал ( <code>right=False</code> включает крайнее левое значение и не включает крайнее правое, интервал закрыт слева и открыт справа; <code>right=True</code> включает крайнее правое значение и не включает крайнее левое, интервал закрыт справа и открыт слева; поскольку либо левый, либо правый край интервала будет открытым, мы поэтому и называем интервал полуоткрытым)

### Возвращает

**ndarray of ints**  
(массив NumPy, состоящий из целых чисел)

Массив индексов интервалов той же формы, что и `x`

Давайте создадим массив вещественных чисел.

```
# создаем массив
a = np.array([3, 9, 15], float)
a
array([3., 9., 15.]
```

Теперь мы создадим список с границами и вернем индекс бина для каждого вещественного числа массива `a`, когда `right=False` и когда `right=True`.

```
# создаем список с границами
lst_bin = [3, 9, 15]
# возвращаем индексы бинов, когда right=False
print(np.digitize(x=a, bins=lst_bin, right=False))
# возвращаем индексы бинов, когда right=True
print(np.digitize(x=a, bins=lst_bin, right=True))

[1 2 3]
[0 1 2]
```

3. 9. 15.

right=False

- 0 [-inf, 3) включает наименьшее значение, но не включает 3
- 1 [3, 9) включает 3, но не включает 9
- 2 [9, 15) включает 9, но не включает 15
- 3 [15, +inf) включает 15, но не включает наибольшее значение

[1 2 3]

3. 9. 15.

right=True

- 0 (-inf, 3] не включает наименьшее значение, но включает 3
- 1 (3, 9] не включает 3, но включает 9
- 2 (9, 15] не включает 9, но включает 15
- 3 (15, +inf] не включает 15, но включает наибольшее значение

[0 1 2]

Функция `np.digitize()` не раз нам пригодится для выполнения биннинга с целью улучшения качества линейных моделей.

### 3.11. ФУНКЦИЯ `NP.SEARCHSORTED()`

Функция `np.searchsorted()` возвращает индексы, в которые должны быть вставлены указанные элементы, чтобы порядок сортировки был сохранен. Она имеет вид:

```
np.searchsorted(arr, v, side='left', sorter=None)
```

Параметр	Предназначение
<b>arr</b> (массив <code>NumPy</code> или объект, подобный массиву)	Задаёт одномерный исходный массив. Предполагается, что массив уже является отсортированным. Данный массив может быть и не отсортирован, но индексы возвращаются именно для отсортированной версии. Если <code>sorter</code> равен <code>None</code> , он должен быть отсортирован в порядке возрастания, в противном случае <code>sorter</code> должен быть массивом целочисленных индексов, который сортирует исходный массив <code>arr</code>

Возьмем первый элемент массива со значением 3. `right=False` включает крайнее левое значение и не включает крайнее правое, интервал закрыт слева и открыт справа. Круглая скобка означает, что соответствующий конец не включается (открыт), а квадратная – что включается (закрыт). Мы видим, что все интервалы закрыты слева и открыты справа (у них квадратная скобка слева и круглая скобка справа). Интервал с индексом 0 включает элементы со значениями меньше 3, поэтому элемент со значением 3 попадает в интервал с индексом 1.

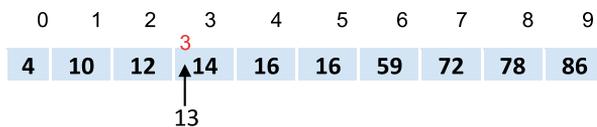
Возьмем первый элемент массива со значением 3. `right=True` не включает крайнее левое значение, но включает крайнее правое. Мы видим, что все интервалы открыты слева и закрыты справа (у них круглая скобка слева и квадратная скобка справа). Интервал с индексом 0 не включает наименьшее значение, но включает 3, поэтому элемент со значением 3 попадает в интервал с индексом 0.

Параметр	Предназначение
<b>v</b> (массив NumPy или объект, подобный массиву)	Задаёт элементы, которые необходимо вставить в массив <code>arr</code> (массив NumPy или объект, подобный массиву)
<b>side</b> (строка 'left' или 'right', необязательный)	Если задано значение 'left', то возвращаем индекс для вставки элемента слева, если задано значение 'right', то возвращаем индекс для вставки элемента справа
<b>sorter</b> (массив NumPy или объект, подобный массиву, необязательный)	Задаёт опциональный массив целочисленных индексов, который сортирует исходный массив <code>arr</code> . Такой массив может быть получен с помощью функции <code>np.argsort()</code>
<b>Возвращает</b>	
<b>int или array of ints</b> (целое число или массив NumPy, состоящий из целых чисел)	Массив индексов, в которые должны быть вставлены указанные элементы, той же формы, что и <code>v</code> , или целое число, если <code>v</code> – это скаляр

Сейчас мы создадим массив целочисленных чисел и найдем индекс, в который должен быть вставлен элемент 13.

```
# создаем массив
a = np.array([4, 10, 12, 14, 16, 16, 59, 72, 78, 86])
# находим индекс, в который должен быть вставлен элемент 13
np.searchsorted(a, 13)
```

3



Данная функция вернет индекс для отсортированной версии массива, даже если он сам не отсортирован (под капотом она выполняет предварительную сортировку массива и поддерживает работу со значениями `nan`).

Сейчас мы создадим неотсортированный массив целочисленных чисел и найдем индекс, в который должен быть вставлен элемент 5.

```
# создаем массив
a = np.array([9, 3, 7, 8, 3, 7, 9])
# находим индекс, в который должен быть вставлен элемент 5
np.searchsorted(a, 5)
```

2



Функция обладает эквивалентным методом класса `ndarray`, т.е. вызов `np.searchsorted(a)` равносильно вызову метода `a.searchsorted()`.

### 3.12. ФУНКЦИЯ `NP.BINCOUNT()`

Функция `np.bincount()` возвращает количество вхождений значений в массиве. Длина выходного массива равна `np.max(x) + 1`. Данная функция выполняет подсчет только целых неотрицательных чисел. Если на вход подан массив с числами типа `float` или `complex`, то будет вызвано исключение `TypeError`. Функция имеет вид:

```
np.bincount(x, weights=None, minlength=0)
```

Параметр	Предназначение
<b>x</b> (массив <code>NumPy</code> или объект, подобный массиву)	Задаёт одномерную последовательность целых неотрицательных чисел. В случае недопустимых значений параметра будет вызвано исключение <code>ValueError</code>
<b>weights</b> (массив <code>NumPy</code> или объект, подобный массиву, необязательный)	Задаёт массив, который имеет ту же длину, что и <code>x</code> , и позволяет назначить весовые коэффициенты каждого его элемента
<b>minlength</b> (целое положительное число или 0, необязательный)	Позволяет задать минимальное значение длины выходного массива. Если указано недопустимое значение, то будет вызвано исключение <code>ValueError</code>
<b>Возвращает</b>	
<b>ndarray of ints</b> (массив <code>NumPy</code> , состоящий из целых чисел)	Массив положительных чисел, указывающих на количество вхождений значений его индекса в исходный массив. Длина выходного массива равна <code>np.max(x) + 1</code>

Давайте создадим массив целых неотрицательных чисел и посмотрим количество вхождений каждого значения.

```
# создаем массив
a = np.array([0, 2, 2, 1, 1, 1])
# смотрим количество вхождений каждого значения
np.bincount(a)

array([1, 3, 2])
```

Видим, что длина равна 3, поскольку максимальное значение в массиве равно 2 и к нему прибавляем единицу. У нас – одно вхождение значения 0, три вхождения значения 1, два вхождения значения 2.

Приведем еще пример.

```
# создаем массив
a = np.array([0, 2, 5, 5])
# смотрим количество вхождений каждого значения
np.bincount(a)

array([1, 0, 1, 0, 0, 2])
```

Видим, что длина равна 6, поскольку максимальное значение в массиве равно 5 и к нему прибавляем единицу. У нас – одно вхождение значения 0, ноль вхождений значения 1, одно вхождение значения 2, ноль вхождений значения 3, ноль вхождений значения 4, два вхождения значения 5.

### 3.13. ФУНКЦИЯ NP.APPLY\_ALONG\_AXIS()

Функция np.apply\_along\_axis() применяет заданную функцию к 1-D срезу массива вдоль указанной оси.

Давайте создадим массив.

```
# создаем массив NumPy
a = np.array([[0, 0, 0, 1],
              [1, 1, 1, 0],
              [0, 1, 1, 1],
              [0, 1, 1, 0]])
```

Теперь с помощью функции np.apply\_along\_axis() применяем по оси строк функцию np.bincount() для подсчета количества вхождений значений, а затем функцию np.argmax() для поиска индекса максимального элемента.

```
# применяем по оси строк функцию подсчета количества
# вхождений значений, затем функцию поиска индекса
# максимального элемента
np.apply_along_axis(lambda x: np.argmax(np.bincount(x)),
                    axis=0,
                    arr=a)

array([0, 1, 1, 0])
```

0	0	0	0	1
1	1	1	1	0
2	0	1	1	1
3	0	1	1	0

Ось 0 (ось строк)

результат np.bincount() → [3, 1] [1, 3] [1, 3] [2, 2]

результат np.argmax() → 0 1 1 0

### 3.14. ФУНКЦИЯ np.INSERT()

Функция `np.insert()` вставляет указанные элементы перед указанными индексами на указанной оси. Функция имеет вид:

```
np.insert(arr, obj, values, axis=None)
```

Параметр	Предназначение
<b>arr</b> (массив NumPy или объект, подобный массиву)	Массив NumPy или любой объект, который может быть преобразован в массив NumPy
<b>obj</b> (срез, целое число или последовательность целых чисел)	Определяет позицию, перед которой необходимо вставить объект. В случае множественных вставок необходимо использовать последовательность целых чисел
<b>values</b> (подобный массиву объект)	Значение, которое необходимо вставить в массив. При множественной вставке необходимо использовать последовательность значений, причем длина <code>values</code> должна соответствовать длине последовательности <code>obj</code> . Тип данных <code>values</code> всегда приводится к типу данных массива <code>arr</code>
<b>axis</b> (целое число, необязательный)	Задаёт ось, вдоль которой нужно вставить значения. Если задано значение <code>None</code> , то <code>arr</code> сначала становится плоским
<b>Возвращает</b>	
<b>ndarray</b> (массив NumPy)	Копия массива <code>arr</code> со вставленными значениями <code>values</code>

Самое частое применение данной функции – вставка строки или столбца в массив. Например, получили метки кластеров с помощью кластерного анализа, и их нужно добавить в качестве столбца в массив признаков, или получили новое наблюдение, его нужно добавить в качестве строки в массив признаков. Проиллюстрируем на нескольких примерах.

```
# создает массив
a = np.array([[0.1, 0.2, 0.5],
              [0.2, 0.7, 0.9],
              [1.3, 3.8, 2.2],
              [2.8, 1.5, 1.9]])
```

*# создаем массив, который нужно добавить как строку*

```
row = np.array([0.4, 0.7, 0.7])
```

*# вставляем строку в конец массива*

```
a = np.insert(a, 4, row, axis=0)
```

```
a
```

```
array([[0.1, 0.2, 0.5],
       [0.2, 0.7, 0.9],
       [1.3, 3.8, 2.2],
       [2.8, 1.5, 1.9],
       [0.4, 0.7, 0.7]])
```

*# создаем массив, который нужно добавить как столбец*

```
column = np.array([0, 1, 1, 0, 0])
```

*# вставляем столбец в конец массива*

```
a = np.insert(a, 3, column, axis=1)
```

```
a
```

```
array([[0.1, 0.2, 0.5, 0. ],
       [0.2, 0.7, 0.9, 1. ],
       [1.3, 3.8, 2.2, 1. ],
       [2.8, 1.5, 1.9, 0. ],
       [0.4, 0.7, 0.7, 0. ]])
```

## 3.15. ФУНКЦИЯ NP.REPEAT()

Функция `np.repeat()` повторяет элементы массива. Функция имеет вид:

```
np.repeat(arr, repeats, axis=None)
```

Параметр	Предназначение
<b>arr</b> (массив NumPy или объект, подобный массиву)	Массив NumPy или любой объект, который может быть преобразован в массив NumPy
<b>repeats</b> (целое число или массив целых чисел)	Количество повторов каждого элемента. Повторы транслируются так, чтобы соответствовать форме данной оси
<b>axis</b> (целое число, не-обязательный)	Задаёт ось, элементы которой нужно повторять. По умолчанию <code>axis=None</code> , т.е. входной массив превращается в плоский, и создается такой же итоговый плоский массив из повторов элементов первого
<b>Возвращает</b>	
<b>ndarray</b> (массив NumPy)	Массив, который образован повторами массива <code>arr</code> вдоль указанной оси и имеет ту же форму, что и <code>arr</code> , кроме данной оси

Самое частое применение данной функции – инициализация константных значений.

Допустим, 4 раза повторим элемент 3.

```
# 4 раза повторяем элемент 3
```

```
np.repeat(3, 4)
```

```
array([3, 3, 3, 3])
```

Два раза повторим элементы двумерного массива. Убеждаемся, что возвращается плоский массив.

```
# создаем двумерный массив
```

```
x = np.array([[1,2],
              [3,4]])
```

```
# два раза повторяем элементы
```

```
np.repeat(x, 2)
```

```
array([1, 1, 2, 2, 3, 3, 4, 4])
```

Теперь три раза повторим элементы двумерного массива по оси 0 и по оси 1.

```
# три раза повторим элементы
```

```
# двумерного массива по оси 0
```

```
np.repeat(x, 3, axis=0)
```

```
array([[1, 2],
       [1, 2],
       [1, 2],
       [3, 4],
       [3, 4],
       [3, 4]])
```

```
# три раза повторим элементы
```

```
# двумерного массива по оси 1
```

```
np.repeat(x, 3, axis=1)
```

```
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
```

## 3.16. Функция np.UNIQUE()

Функция `np.unique()` находит уникальные элементы массива и возвращает их в отсортированном массиве.

В зависимости от установленных параметров данная функция может возвращать:

- индексы входного массива, которые соответствуют его уникальным элементам;
- индексы уникального массива, которые позволяют восстановить входной массив;
- количество вхождений каждого уникального элемента во входном массиве.

Параметр	Предназначение
<b>arr</b> (массив NumPy или объект, подобный массиву)	Массив NumPy или любой объект, который может быть преобразован в массив NumPy. Если входной массив не является одномерным и не указана ось, вдоль которой нужно искать уникальные элементы, то данный массив будет сжат до одной оси
<b>return_index</b> (True или False, необязательный)	Если задано True, то помимо самих уникальных элементов также будут возвращаться их индексы во входном массиве. По умолчанию <code>return_index=False</code>
<b>return_inverse</b> (True или False, необязательный)	Если задано True, то помимо самих уникальных элементов также будут возвращаться индексы уникального массива, которые можно использовать для восстановления входного массива. По умолчанию <code>return_inverse=False</code>
<b>return_counts</b> (True или False, необязательный)	Если задано True, то помимо самих уникальных элементов также будет возвращаться количество вхождений каждого из них во входном массиве. По умолчанию <code>return_counts=False</code>
<b>axis</b> (целое число или None, необязательный)	Определяет ось, по которой необходимо найти уникальные элементы. Если <code>axis=None</code> (по умолчанию), то входной массив будет сжат до одной оси
<b>Возвращает</b>	
<b>ndarray</b> (массив NumPy)	Массив уникальных элементов
<b>ndarray</b> (массив NumPy)	Массив индексов первых вхождений уникальных элементов (если <code>return_index=True</code> )
<b>ndarray</b> (массив NumPy)	Массив индексов всех вхождений уникальных элементов (если <code>return_inverse=True</code> )
<b>ndarray</b> (массив NumPy)	Массив с количеством вхождений уникальных элементов (если <code>return_counts=True</code> )

Создадим одномерный массив и извлечем из него уникальные элементы.

```
# извлечем уникальные элементы
np.unique([1, 2, 2, 2, 3, 3, 3, 3, 3])

array([1, 2, 3])
```

Теперь посмотрим встречаемость каждого уникального значения.

```
# создаем массив
a = np.array([1, 2, 2, 2, 3, 3, 3, 3, 3])

# выведем частоты встречаемости уникальных элементов
values, counts = np.unique(a, return_counts=True)
dict(zip(values, counts))

{1: 1, 2: 3, 3: 5}
```

Теперь получим массив индексов первых вхождений уникальных элементов.

```
# получим массив индексов первых
# вхождений уникальных элементов
_, indices = np.unique(a, return_index=True)
indices

array([0, 1, 4])
```

0	1	2	3	4	5	6	7	8
1	2	2	2	3	3	3	3	3

### 3.17. ФУНКЦИЯ NP.TAKE\_ALONG\_AXIS()

Функция `np.take_along_axis()` сопоставляет одномерным массивам индексов с соответствующими полными срезами исходного массива вдоль указанной оси и возвращает найденные элементы.

Параметр	Предназначение
<b>arr</b> (массив NumPy или объект, подобный массиву)	Исходный массив
<b>indices</b> (массив NumPy, необязательный)	Массив индексов, который должен быть либо транслируемым по массиву <code>arr</code> , либо содержать столько же одномерных массивов, сколько их в индексированном массиве вдоль указанной в параметре <code>axis</code> оси
<b>axis</b> (целое число или <code>None</code> , необязательный)	Определяет ось, вдоль которой извлекаются элементы с указанными в одномерных массивах индексами. По умолчанию <code>axis=None</code> , что соответствует извлечению элементов из сжатого до одной оси представления массива <code>arr</code>
<b>Возвращает</b>	
<b>ndarray</b> (массив NumPy)	Массив элементов исходного массива, выбранных в соответствии с индексами одномерных массивов из полного среза вдоль указанной оси исходного массива

Самый типичный пример использования этой функции – выбор топ  $k$  значений из массива.

Давайте создадим массив NumPy.

```
# создаем массив X
np.random.seed(0)
X = np.round(np.random.rand(10, 10), 2)
X
array([[0.55, 0.72, 0.6 , 0.54, 0.42, 0.65, 0.44, 0.89, 0.96, 0.38],
       [0.79, 0.53, 0.57, 0.93, 0.07, 0.09, 0.02, 0.83, 0.78, 0.87],
       [0.98, 0.8 , 0.46, 0.78, 0.12, 0.64, 0.14, 0.94, 0.52, 0.41],
       [0.26, 0.77, 0.46, 0.57, 0.02, 0.62, 0.61, 0.62, 0.94, 0.68],
       [0.36, 0.44, 0.7 , 0.06, 0.67, 0.67, 0.21, 0.13, 0.32, 0.36],
       [0.57, 0.44, 0.99, 0.1 , 0.21, 0.16, 0.65, 0.25, 0.47, 0.24],
       [0.16, 0.11, 0.66, 0.14, 0.2 , 0.37, 0.82, 0.1 , 0.84, 0.1 ],
       [0.98, 0.47, 0.98, 0.6 , 0.74, 0.04, 0.28, 0.12, 0.3 , 0.12],
       [0.32, 0.41, 0.06, 0.69, 0.57, 0.27, 0.52, 0.09, 0.58, 0.93],
       [0.32, 0.67, 0.13, 0.72, 0.29, 0.18, 0.59, 0.02, 0.83, 0. ]])
```

Затем получим отсортированные индексы по каждой строке.

```
# получим отсортированные индексы по строке
order = X.argsort(axis=1)
order

array([[9, 4, 6, 3, 0, 2, 5, 1, 7, 8],
       [6, 4, 5, 1, 2, 8, 0, 7, 9, 3],
       [4, 6, 9, 2, 8, 5, 3, 1, 7, 0],
       [4, 0, 2, 3, 6, 5, 7, 9, 1, 8],
       [3, 7, 6, 8, 0, 9, 1, 4, 5, 2],
       [3, 5, 4, 9, 7, 1, 8, 0, 6, 2],
       [7, 9, 1, 3, 0, 4, 5, 2, 6, 8],
       [5, 7, 9, 6, 8, 1, 3, 4, 0, 2],
       [2, 7, 5, 0, 1, 6, 4, 8, 3, 9],
       [9, 7, 2, 5, 4, 0, 6, 1, 3, 8]])
```

Находим топ 3 индексов, т.е. получаем индексы 3 наблюдений с наибольшими значениями.

```
# найдем топ 3 индексов
top_idx = order[:, -3:][:, ::-1]
top_idx

array([[8, 7, 1],
       [3, 9, 7],
       [0, 7, 1],
       [8, 1, 9],
       [2, 5, 4],
       [2, 6, 0],
       [8, 6, 2],
       [2, 0, 4],
       [9, 3, 8],
       [8, 3, 1]])
```

Теперь с помощью функции np.take\_along\_axis() из каждой строки извлекаем 3 наибольших значения.

```
# из каждой строки достанем 3 наибольших значения
top_k = np.take_along_axis(X, top_idx, axis=1)
top_k

array([[0.96, 0.89, 0.72],
       [0.93, 0.87, 0.83],
       [0.98, 0.94, 0.8 ],
       [0.94, 0.77, 0.68],
       [0.7 , 0.67, 0.67],
       [0.99, 0.65, 0.57],
       [0.84, 0.82, 0.66],
       [0.98, 0.98, 0.74],
       [0.93, 0.69, 0.58],
       [0.83, 0.72, 0.67]])
```

## 3.18. ФУНКЦИЯ `np.ARRAY_SPLIT()`

Функция `np.array_split()` разбивает массив на несколько подмассивов. Чаще всего используется в операциях, которые нужно распараллелить. Допустим, есть сложная функция, которая применяется к массиву построчно. Чтобы ускорить вычисления, можно использовать параллельные процессы. Поясним на примере.

```
# импортируем Parallel и delayed для распараллеливания
from joblib import Parallel, delayed
```

```
# пишем медленную функцию
```

```
def slow_fn(x, n=5):
    """
    Медленная функция.

    Параметры
    -----
    x: np.ndarray
        Массив значений.
    n: int
        Количество итераций
        выполнения операции.

    Возвращает
    -----
    results: np.ndarray
        Массив значений.
    """
    for i in range(n):
        np.random.seed(0)
        rnd = np.random.rand(*x.shape)
        x = x + rnd

    return x
```

```
# пишем функцию с распараллеливанием
```

```
def parallel_map(fn, x, n_jobs=4, *args, **kwargs):
    """
    Функция с распараллеливанием.

    Параметры
    -----
    fn: callable
        Вызываемая функция.
    x: np.ndarray
        Входной массив.
    n_jobs: int
        Количество процессов.
    args: list
        Неименованные аргументы для
        передачи в функцию.
    kwargs: dict
        Именованные аргументы для
        передачи в функцию.
```

```

Возвращает
-----
results: np.ndarray
    Массив значений.
"""
with Parallel(n_jobs) as p:
    res = p(delayed(fn)(part, *args, **kwargs)
            for part in np.array_split(x, n_jobs))

return np.concatenate(res)

# создаем массив
np.random.seed(0)
X = np.random.rand(100000, 100)

%%time

slow_fn(X, n=100)

CPU times: user 7.11 s, sys: 637 ms, total: 7.74 s
Wall time: 7.75 s

array([[ 55.4301639 ,  72.234126  ,  60.87910098, ...,  2.03086216,
         83.72294295,   0.4742431  ],
       [ 68.45947022,  27.27080529,  74.25459623, ...,  25.69000466,
         5.86094519,  43.87607918],
       [ 31.49138408,  70.33069237,  38.15293577, ...,  87.08134326,
         98.26486839,  97.04430046],
       ...,
       [100.58449943,  55.26692164,  97.40958124, ...,  97.61717411,
         86.8888155  ,  42.39754644],
       [ 27.53629179,   7.55992853,  92.99295104, ...,  68.78146004,
         34.7257013  ,   5.88385571],
       [ 66.54489318,  25.22817557,  49.20478964, ...,  70.86996313,
         46.07244422,  14.69914449]])

%%time

parallel_map(slow_fn, X, n_jobs=4, n=100)

CPU times: user 139 ms, sys: 158 ms, total: 297 ms
Wall time: 3.36 s

array([[ 55.4301639 ,  72.234126  ,  60.87910098, ...,  2.03086216,
         83.72294295,   0.4742431  ],
       [ 68.45947022,  27.27080529,  74.25459623, ...,  25.69000466,
         5.86094519,  43.87607918],
       [ 31.49138408,  70.33069237,  38.15293577, ...,  87.08134326,
         98.26486839,  97.04430046],
       ...,
       [ 31.76407696,  42.2681517  , 100.35156516, ...,  41.88772581,
         99.50098923,  49.18558749],
       [ 56.78399612,  25.69477427,  94.95402764, ...,  95.01045663,
         6.75348646,  23.41179416],
       [ 83.17514864,  31.92025962,  57.17805087, ...,  46.36804628,
         30.0120986  ,  74.90787463]])

```

Для получения подробной информации о библиотеке NumPy рекомендуется ознакомиться со справочным руководством по NumPy на русском языке [https://pyprog.pro/reference\\_manual.html](https://pyprog.pro/reference_manual.html).

## Задача с собеседования (SQL)

- Из таблицы `employees` получите с помощью SQL-запросов:
  - список сотрудников с именем 'David';
  - сотрудника с минимальным возрастом;
  - сотрудника с максимальным возрастом;
  - список сотрудников старше 30 лет;
  - список сотрудников, у которых в имени содержится буква 'j';
  - средний возраст сотрудника в каждом отделе;
  - количество сотрудников в каждом отделе;
  - список сотрудников моложе 27 и работающих в отделе 'B'.

id	name	age	department
1	David	22	B
2	Paul	33	B
3	Jeremy	26	B
4	Jack	21	C
5	John	36	A
6	David	45	A

## 4. БИБЛИОТЕКИ NUMBA, DATATABLE, VOTTLNECK ДЛЯ УСКОРЕНИЯ ВЫЧИСЛЕНИЙ

### 4.1. Numba

Numba (произносится как намба) – библиотека с открытым исходным кодом, которая позволяет создавать быстрые функции для массивов NumPy. Ее можно установить с помощью `pip` или `conda`: `pip install numba` или `conda install numba`. Numba использует JIT-компилятор (компилятор «на лету», от «just in time» – «на лету») на основе инфраструктуры LLVM (Low-Level Virtual Machine), позволяющий с помощью аннотирования транслировать массивоориентированный и математически нагруженный питоновский код в машинный код во время исполнения программы, то есть «на лету» (just-in-time, отсюда и название). Таким образом, получаем код, аналогичный по производительности языкам C, C++, при этом можно избежать переключения языков или интерпретаторов Python. Numba поддерживает компиляцию Python в машинный код на любом CPU или GPU и предназначен для интеграции со стеком научного программного обеспечения Python.

Обратите внимание, что Numba можно использовать с pandas. Установив Numba, в некоторых методах pandas вы можете задать ключевое слово `engine='numba'`. С вычислительной точки зрения первый запуск функции с использованием движка Numba будет медленным, поскольку у Numba будут некоторые накладные расходы на компиляцию функции. Однако функции, скомпилированные JIT, кешируются, и последующие вызовы будут выполняться быстро.

Кроме того, можно задать свою собственную функцию Python с декоратором `@jit` и передать массив NumPy, созданный на основе объектов Series или Dataframe (используйте `to_numpy()`), в эту функцию.

Также обратите внимание, что библиотека Numba может выполнить любую функцию, однако ускорить она может лишь определенные функции.

Когда мы передаем функцию, использующую только те операции, которые Numba знает, как ускорить, библиотека работает в режиме `nopython`, т.е. библиотека выполняет компиляцию. Если библиотеке Numba компиляция не удалась (такое возможно, если передать функцию, которая использует то, с чем библиотека Numba не умеет работать, например функция использует список, содержащий элементы разного типа), она переключается в режим `object`. В режиме `object` Numba выполнит ваш программный код, но при этом значительно увеличения производительности не произойдет. Так происходит по умолчанию (по умолчанию `numba` передан аргумент `nopython=False`).

Опционно библиотека Numba может выдать ошибку, если не сможет скомпилировать функцию так, чтобы ускорить выполнение вашего программного кода. Для этого передайте Numba аргумент `nopython=True` (например, `@numba.jit(nopython=True)`, что эквивалентно `@numba.njit`). Для получения более подробной информации о самой библиотеке смотрите документацию по библиотеке Numba <http://numba.pydata.org/>.

Numba поддерживает создание и возвращение списков из JIT-скомпилированных функций, а также всех связанных методов и операций. Списки должны быть строго однородными: Numba отклонит любой список, содержащий объекты разных типов, даже если эти типы совместимы (например, `[1, 2.5]` отклоняется, так как содержит `int` и `float`). В Numba для этих целей появился `numba.typed.List` (типизированный список). Numba поддерживает генераторы списков. Все методы и операции над множествами поддерживаются в JIT-скомпилированных функциях. Множества должны быть строго однородными: Numba отклонит любой набор, содержащий объекты разных типов, даже если типы совместимы (например, `{1, 2.5}` отклоняется, поскольку объект содержит `int` и `float`). Что касается словарей, функция `dict()` до недавнего времени не поддерживалась, теперь ее можно использовать, но без каких-либо аргументов, такое использование по смыслу эквивалентно `{}` и `numba.typed.Dict()`. В результате мы имеем экземпляр `numba.typed.Dict`, в котором пары ключ-значение будут определены позже при использовании.

Сейчас мы напишем функцию на чистом Python, которая вычисляет среднее расстояние между двумя значениями с помощью цикла `for`.

```
# пишем функцию, вычисляющую среднее расстояние
# между двумя значениями
def mean_distance(x, y):
```

```

nx = len(x)
result = 0.0
count = 0
for i in range(nx):
    result += x[i] - y[i]
    count += 1
return result / count

```

Создаем два массива значений.

```

# создаем массивы
x = np.random.randn(10000000)
y = np.random.randn(10000000)

```

Давайте сравним нашу функцию `mean_distance()` и функцию `np.mean()` с точки зрения скорости вычислений.

```

# проверяем скорость выполнения нашей функции
%timeit mean_distance(x, y)

```

4.44 s ± 181 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```

# проверяем скорость выполнения функции np.mean()
%timeit np.mean(x - y)

```

29.7 ms ± 3 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

Видим, что функция NumPy более чем в 100 раз быстрее нашей функции `mean_distance()`.

Теперь импортируем библиотеку `numba`, с помощью функции `numba.jit()` превращаем нашу функцию в компилируемую функцию `Numba` и проверяем ее скорость работы.

```

# импортируем numba
import numba as nb

```

```

# превращаем нашу функцию в компилируемую функцию
# Numba с помощью функции numba.jit()
numba_mean_distance = nb.jit(mean_distance)

```

```

# проверяем скорость выполнения функции
# numba_mean_distance()
%timeit numba_mean_distance(x, y)

```

12.8 ms ± 1.53 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Полученная функция работает быстрее векторизированной функции NumPy. Кроме того, мы можем взять программный код нашей функции, приведенный выше, и аннотировать его с помощью декоратора `@jit`.

```

# воспользуемся декоратором @jit
@nb.jit

```

```
def nb_mean_distance(x, y):
    nx = len(x)
    result = 0.0
    count = 0
    for i in range(nx):
        result += x[i] - y[i]
        count += 1
    return result / count
```

```
# проверяем скорость выполнения
# функции nb_mean_distance()
%timeit nb_mean_distance(x, y)
```

11.7 ms ± 898 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Декоратор `@vectorize` позволяет питоновским функциям, принимающим скалярные аргументы, использоваться в качестве универсальных функций NumPy. Написание универсальной функции NumPy не является самым простым процессом и включает в себя написание некоторого кода на C. Numba делает это легко. Используя декоратор `@vectorize`, Numba может скомпилировать чистую функцию Python в универсальную функцию, которая работает с массивами NumPy так же быстро, как традиционные универсальные функции, написанные на C. Итак, следует помнить, что, используя `@vectorize`, вы пишете функцию, выполняющую операцию над скалярами, а не над массивами.

Рассмотрим следующий игрушечный пример, в котором мы умножаем все значения в массиве `x` на 2.

```
# воспользуемся декоратором @vectorize
@nb.vectorize()

def nb_square(x):
    return x ** 2
# проверяем скорость выполнения функции nb_square()
%timeit nb_square(x)
```

15.8 ms ± 218 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Сложные последовательности математических операций можно распараллелить с помощью параметра `parallel` декоратора `@jit`, тем самым сократив время выполнения программного кода. Сравним вариант без распараллеливания и вариант с распараллеливанием.

```
# генерируем ряд значений
data = np.random.randn(10000000)

# воспользуемся декоратором @jit без распараллеливания
@nb.jit()

def f(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2
```

```
%timeit f(data)
127 ms ± 7.25 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

*# воспользуемся декоратором @jit с распараллеливанием*

```
@nb.jit(parallel=True)
```

```
def f(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2
```

```
%timeit f(data)
```

22.6 ms ± 540 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)

С помощью декоратора `@jit` с включенным распараллеливанием (`parallel=True`) и функцией `numba.prange()` вы можете распараллелить операции в цикле. Использование же декоратора `@jit` с отключенным распараллеливанием (`parallel=False`, используется по умолчанию) и функцией `numba.prange()` будет эквивалентно применению функции `range()`.

*# воспользуемся декоратором @jit без*

*# распараллеливания и функцией prange()*

```
@nb.jit()
```

```
def compute(x):
    s = 0
    for i in nb.prange(x.shape[0]):
        s += x[i]
    return s
```

```
%timeit compute(data)
```

10.7 ms ± 335 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

*# воспользуемся декоратором @jit без*

*# распараллеливания и функцией prange()*

```
@nb.jit(parallel=True)
```

```
def compute(x):
    s = 0
    for i in nb.prange(x.shape[0]):
        s += x[i]
    return s
```

```
%timeit compute(data)
```

2.97 ms ± 146 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

Посмотрим, как Numba может ускорить работу методов `pandas` на примере вычисления скользящего среднего с шириной окна 10. Простое скользящее среднее (simple moving average – SMA) – полезный признак для временных рядов. Его формула незатейлива:

$$SMA = \frac{A_1 + A_2 + \dots + A_n}{n}$$

В рамках подхода «скользящее окно» библиотека `pandas` вычисляет статистику по «окну» данных, представляющему определенный период времени. Затем окно смещается на определенный интервал времени, и статистика постоянно вычисляется для каждого нового окна до тех пор, пока окно охватывает

даты временного ряда. На рисунке ниже показано вычисление скользящего среднего с шириной окна 3.

```
data['rolling_mean3'] = data['sales'].rolling(window=3).mean()
```

sales			rolling_mean3
date			
2018-01-09	2400	$\frac{2400 + 2800 + 2500}{3} = 2566.67$	NaN
2018-01-10	2800		NaN
2018-01-11	2500		2566.66667
2018-01-12	2890	$\frac{2800 + 2500 + 2890}{3} = 2730.00$	2730.000000
2018-01-13	2610		2666.66667
2018-01-14	2500	$\frac{2500 + 2890 + 2610}{3} = 2666.67$	2666.66667
2018-01-15	2750		2620.000000
2018-01-16	2700		2650.000000
2018-01-17	2250		2566.66667
2018-01-18	2350		2433.333333
2018-01-19	2550		2383.333333
2018-01-20	3000		2633.333333

Рис. 13 Вычисление скользящего среднего с шириной окна 3

Итак, давайте импортируем необходимые библиотеки и создадим экспериментальные данные.

```
# импортируем библиотеки pandas и numpy
import pandas as pd
import numpy as np

# создаем серию со 100 000 значений
series = pd.Series(np.random.randn(100000))

# задаем объект Rolling
roll = series.rolling(10)
# пишем функцию вычисления среднего
def f(x):
    return np.mean(x)
```

Делаем вычисления с помощью методов pandas с движком Numba и без него.

```
# запускаем в первый раз, время компиляции
# повлияет на производительность, здесь и
# далее если raw=True, то переданная в метод .apply()
# функция вместо серии получает массив NumPy,
# -r(epeat) - сколько раз повторять таймер,
# -n(umber) - сколько раз выполнять команду
%timeit -r 1 -n 1 roll.apply(f, engine='numba', raw=True)
```

434 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)

```
# функция кешируется, и производительность улучшается
%timeit roll.apply(f, engine='numba', raw=True)
```

14.5 ms ± 223 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```
%timeit roll.apply(f, raw=True)
```

690 ms ± 23.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Видим, что использование движка Numba позволило сократить время вычислений.

## 4.2. DATATABLE

Datatable – это питоновская библиотека, которая аналогично pandas предназначена для предварительной подготовки данных, но в большей степени ориентирована на обеспечение высокой скорости обработки данных и на поддержку больших наборов данных. Библиотеку с определенным допущением можно назвать аналогом пакета R `data.table`. С помощью `datatable` можно работать как с данными, которые полностью помещаются в оперативной памяти, так и с данными, размер которых превышает объём доступной RAM. Спонсором разработки `datatable` является компания H2O.ai. `Datatable` можно легко установить с помощью команды `pip install datatable`.

Давайте сравним скорость загрузки файла размером 291 Мб (набор данных с 1 787 571 наблюдением и 34 столбцами) в `pandas` и `datatable`.

```
%%time
# загружаем данные с помощью pandas
dataframe = pd.read_csv('Data/train.csv', sep=',')

CPU times: user 5.88 s, sys: 1.15 s, total: 7.03 s
Wall time: 5.45 s

%%time
# загружаем данные с помощью datatable
datatable_df = dt.fread('Data/train.csv', sep=',')

CPU times: user 3.89 s, sys: 415 ms, total: 4.3 s
Wall time: 390 ms
```

Видим, что загрузка данных с помощью пакета `datatable` происходит значительно быстрее. Убедимся, что мы получили объект `Frame` пакета `datatable`.

```
# убедимся, что перед нами - объект
# Frame библиотеки datatable
type(datatable_df)

datatable.Frame
```

Итак, мы получили объект `Frame` пакета `datatable`. При желании его можно преобразовать в объект `DataFrame` библиотеки `pandas` или массив `NumPy` с помощью методов `.to_pandas()` и `.to_numpy()` соответственно.

Попробуем преобразовать наш фрейм `datatable` в объект `DataFrame` библиотеки `pandas` и посмотрим на то, сколько это займёт времени.

```
%%time
```

```
# преобразовываем фрейм в датафрейм pandas
datatable_pandas = datatable_df.to_pandas()
```

```
CPU times: user 14.2 s, sys: 1.35 s, total: 15.5 s
Wall time: 2.84 s
```

Преобразование фрейма в объект DataFrame библиотеки pandas занимает меньше времени, чем загрузка данных в DataFrame средствами pandas.

Рассмотрим основные свойства метода объекта Frame библиотеки datatable.

С помощью свойства shape можно узнать количество наблюдений и количество столбцов фрейма.

```
# смотрим количество строк и столбцов
datatable_df.shape
```

```
(1787571, 34)
```

С помощью метода .head() можно вывести первые 10 наблюдений.

```
# смотрим первые 10 наблюдений
datatable_df.head()
```

	ID	SK_DATE_DECISION	DEF	NUM_SOURCE	CREDIT_ACTIVE	CREDIT_COLLATERAL	CREDIT_CURRENCY	DTIME_CREDIT	CREDIT_DAY_OVERDUE
0	24368	20150901	0	1	0	0	eur	27.03.2011	0
1	24368	20150901	0	3	0	0	eur	03.11.2011	0
2	24368	20150901	0	4	0	0	eur	03.11.2011	0
3	24368	20150901	0	3	0	0	eur	24.02.2012	0
4	24368	20150901	0	4	0	0	eur	24.02.2012	0
5	24368	20150901	0	3	2	0	eur	31.01.2014	0
6	24368	20150901	0	4	2	0	eur	31.01.2014	0
7	24368	20150901	0	3	0	0	eur	31.05.2015	0
8	24368	20150901	0	3	1	0	eur	17.06.2015	0
9	24368	20150901	0	4	1	0	eur	17.06.2015	0

Цвета имен столбцов указывают на тип данных. Красным цветом обозначены строки, зелёным – целые числа, синим – числа с плавающей точкой.

С помощью свойств .names() и .stypes() можно вывести типы переменных.

```
# смотрим типы переменных
for col in range(len(datatable_df.names)):
    print(datatable_df.names[col], ':', datatable_df.stypes[col])
```

```
ID : stype.int32
SK_DATE_DECISION : stype.int32
DEF : stype.bool8
NUM_SOURCE : stype.int32
CREDIT_ACTIVE : stype.int32
CREDIT_COLLATERAL : stype.bool8
CREDIT_CURRENCY : stype.str32
DTIME_CREDIT : stype.str32
CREDIT_DAY_OVERDUE : stype.int32
DTIME_CREDIT_ENDDATE : stype.str32
DTIME_CREDIT_ENDDATE_FACT : stype.str32
```

```

CREDIT_FACILITY : stype.int32
AMT_CREDIT_MAX_OVERDUE : stype.float64
CNT_CREDIT_PROLONG : stype.int32
AMT_CREDIT_SUM : stype.float64
AMT_CREDIT_SUM_DEBT : stype.float64
AMT_CREDIT_SUM_LIMIT : stype.float64
AMT_CREDIT_SUM_OVERDUE : stype.float64
CREDIT_SUM_TYPE : stype.bool8
CREDIT_TYPE : stype.int32
DTIME_CREDIT_UPDATE : stype.str32
CREDIT_DELAY30 : stype.int32
CREDIT_DELAY5 : stype.int32
CREDIT_DELAY60 : stype.int32
CREDIT_DELAY90 : stype.int32
CREDIT_DELAY_MORE : stype.int32
AMT_REQ_SOURCE_HOUR : stype.int32
AMT_REQ_SOURCE_DAY : stype.int32
AMT_REQ_SOURCE_WEEK : stype.int32
AMT_REQ_SOURCE_MON : stype.int32
AMT_REQ_SOURCE_QRT : stype.int32
AMT_REQ_SOURCE_YEAR : stype.int32
AMT_ANNUITY : stype.float64
TEXT_PAYMENT_DISCIPLINE : stype.str32

```

Статистики по столбцам можно получить с помощью следующих методов:

```

datatable_df.sum()      datatable_df.nunique()
datatable_df.sd()      datatable_df.max()
datatable_df.mode()    datatable_df.min()
datatable_df.nmodal()  datatable_df.mean()

```

Для отбора строк и столбцов используем квадратные скобки.

```

# отбираем столбец NUM_SOURCE
# и выводим первые 5 наблюдений
datatable_df[:, 'NUM_SOURCE'].head(5)

```

	NUM_SOURCE
0	1
1	3
2	4
3	3
4	4

5 rows x 1 column

```

# отбираем первые 3 строки столбца NUM_SOURCE
datatable_df[:3, 'NUM_SOURCE']

```

	DEF
0	0
1	0
2	0

3 rows x 1 column

```
# отбираем первые 3 строки столбца NUM_SOURCE
# (столбца с индексом 2)
datatable_df[:, 2]
```

	DEF
0	0
1	0
2	0

3 rows x 1 column

```
# отбираем первые 3 строки первых двух столбцов
# (столбцов ID и SK_DATE_DECISION)
datatable_df[:, :2]
```

	ID	SK_DATE_DECISION
0	24368	20150901
1	24368	20150901
2	24368	20150901

3 rows x 2 columns

С помощью метода `.countna()` можно вывести количество пропусков по каждой переменной.

```
# выведем количество пропусков по каждой переменной
datatable_df.countna()
```

	ID	SK_DATE_DECISION	DEF	NUM_SOURCE	CREDIT_ACTIVE	CREDIT_COLLATERAL	CREDIT_CURRENCY	DTIME_CREDIT
0	0		0	0	0	0	0	0

Теперь посмотрим, как можно вычислить групповые статистики. Сейчас мы выведем средние суммы задолженности по первым 5 клиентам.

```
# выведем средние суммы задолженности по первым 5 клиентам
datatable_df[:, dt.mean(dt.f.AMT_CREDIT_SUM_DEBT), dt.by('ID')].head(5)
```

	ID	AMT_CREDIT_SUM_DEBT
0	24368	13974.8
1	24369	9350.87
2	24370	807.371
3	24372	0
4	24373	5860.14

5 rows x 2 columns

С помощью ключевого слова `del` можно удалять переменные. Например, удалим переменную `ID`.

```
# удаляем столбец ID
del datatable_df[:, 'ID']
```

Содержимое объекта `Frame` можно записать в CSV-файл, что позволяет использовать данные в будущем.

```
# сохраняем фрейм в CSV-файл
datatable_df.to_csv('datatable_results.csv')
```

Библиотека `datatable` работает быстрее библиотеки `pandas`, однако пока главный минус `datatable` в сравнении с `pandas` – ограниченный функционал.

## 4.3. BOTTLENECK

`Bottleneck` – это коллекция быстрых функций `NumPy`, написанных на языке С. Пакет хорошо знаком исследователям, работающим с временными рядами. В модуле `move` пакета `bottleneck` можно найти быстрые реализации скользящих статистик.

**Таблица 2** Скользящие статистики в `pandas`

Функция	Описание
<code>move_mean()</code>	Среднее значение в окне
<code>move_std</code>	Стандартное отклонение в окне
<code>move_var()</code>	Дисперсия в окне
<code>move_min()</code>	Минимальное значение в окне
<code>move_max()</code>	Максимальное значение в окне
<code>move_median()</code>	Медиана в окне
<code>move_sum()</code>	Сумма в окне

Разберем функцию `bn.move_mean()`, вычисляющую скользящее среднее. Функция не работает с объектом `Series`, ей нужно передать массив `NumPy`. С помощью параметра `window` можно задать ширину окна. Параметр `min_count` работает следующим образом: если количество непропущенных значений в окне меньше `min_count`, окну присваивается значение `NaN`. По умолчанию для параметра `min_count` установлено значение `None`, это значит, что параметру `min_count` мы назначаем значение параметра `window`.

Теперь мы создадим серию с 100 000 значений, сравним вычисление скользящего среднего с помощью функции `rolling().mean()` библиотеки `pandas` и с помощью функции `move_mean()` пакета `bottleneck`.

```
# импортируем необходимые библиотеки
import bottleneck as bn
import pandas as pd
import numpy as np

# создаем серию со 100 000 значений
series = pd.Series(np.random.randn(100000))

%%time

roll_mean = series.shift(1).rolling(
    min_periods=1, window=4).mean()

CPU times: user 4.91 ms, sys: 2.57 ms, total: 7.48 ms
Wall time: 6.51 ms

%%time
```

```
roll_mean_bn = bn.move_mean(series.shift(1),
                             window=4, min_count=1)
```

```
CPU times: user 772 µs, sys: 387 µs, total: 1.16 ms
Wall time: 597 µs
```

Видим, что функция `bn.move_mean()` пакета `bottleneck` работает быстрее.

## Задачи с собеседований (теория вероятности)

1. В урне имеется 3 белых и 4 черных шара. Из урны вытягиваются 3 шара. Найти вероятность, что хотя бы один из них окажется белым.
2. Игральный кубик бросается 6 раз. Найти вероятность, что выпадет хотя бы одна шестерка.

## 5. SciPy

SciPy (произносится как сайпай) представляет собой набор математических и статистических функций для научных вычислений в Python. Библиотека SciPy позволяет:

- выполнять статистические тесты;
- находить минимумы и максимумы функций;
- вычислять интегралы функций;
- поддерживать специальные функции;
- выполнять обработку сигналов;
- выполнять обработку изображений;
- работать с генетическими алгоритмами;
- решать обыкновенные дифференциальные уравнения.

Базовая структура данных – массив NumPy.

Чаще всего пакет SciPy используется для тестирования статистических гипотез.

Гипотеза в переводе с греческого обозначает «предположение». Статистическая гипотеза, которую мы выдвигаем, представляет собой некоторое предположение о генеральной совокупности, проверяемое по наблюдаемой выборке данных. Она называется нулевой и обозначается через  $H_0$ . Например, мы выдвигаем нулевую гипотезу о том, что средние значения двух генеральных совокупностей, из которых извлечены сравниваемые зависимые выборки, не отличаются друг от друга. Наряду с выдвинутой гипотезой  $H_0$  рассматривают и альтернативную ей гипотезу  $H_1$  (средние значения двух генеральных совокупностей, из которых извлечены сравниваемые зависимые выборки, различны).

В итоге проверки гипотез могут быть приняты неправильные решения, т.е. могут быть допущены ошибки двух родов.

Ошибка первого рода состоит в том, что мы отвергаем нулевую гипотезу  $H_0$ , когда она верна. Вероятность ошибки I рода (вероятность ложного отклонения нулевой гипотезы) обозначают  $p$  и называют  $p$ -значением ( $p$ -value). Можно сказать, что  $p$ -значение – вероятность того, что случайная величина, имеющая

распределение выбранной статистики при условии верности нулевой гипотезы, примет значение, не меньшее, чем вычисленное значение этой статистики.

Пороговую (критическую) вероятность совершить ошибку первого рода принято обозначать  $\alpha$  или уровнем значимости (significance level). Уровень значимости  $\alpha$  связан с доверительной вероятностью  $1 - \alpha$ .

Традиционно выделяют три уровня значимости:

- $p \leq 0,05$  ( $\alpha = 0,05$ ) – обычный уровень статистической значимости (для наглядности традиционно обозначается одной звездочкой \*);
- $p \leq 0,01$  ( $\alpha = 0,01$ ) – высокий уровень значимости (для наглядности традиционно обозначается двумя звездочками \*\*);
- $p \leq 0,001$  ( $\alpha = 0,001$ ) – очень высокий уровень значимости (обозначается тремя звездочками \*\*\*).

Наиболее часто уровень значимости принимают равным 0,05 или 0,01. Если, например, мы приняли уровень значимости, равный 0,05, доверительная вероятность будет составлять 0,95.

Вы всегда должны помнить, что в статистике «вероятность» относится не к гипотезам, а к величинам, которые являются гипотетическими частотами появления паттернов или закономерностей в рамках предполагаемой статистической модели.

Например, выше мы выдвинули нулевую гипотезу о том, что средние значения двух генеральных совокупностей, из которых извлечены сравниваемые зависимые выборки, не отличаются друг от друга. Допустим, мы получили  $p$ -значение 0,05, это означает, что в 5 случаях из 100 мы допускаем ошибку первого рода – отвергаем нулевую гипотезу, когда она верна (т. е. средние значения генеральных совокупностей равны). Однако это не обозначает, что вероятность того, что между средними значениями генеральных совокупностей есть разница, составляет 95 %.

Ошибка второго рода состоит в том, что мы делаем вывод об отсутствии оснований отвергнуть нулевую гипотезу  $H_0$ , когда она неверна. Ошибку II рода обозначают  $\beta$ .

Обратите внимание, фраза «нет оснований отклонить нулевую гипотезу» не тождественна фразе «принять нулевую гипотезу», которая является неверной. Нулевая гипотеза обычно имеет очень конкретную формулировку. В нашем случае она звучит так: нет разницы между средними значениями генеральной совокупности № 1 и генеральной совокупности № 2. Если мы не можем отклонить нулевую гипотезу, значит ли это, что данные значения равны? Вовсе не обязательно. То, что нам не удалось найти статистически значимую разницу, совершенно не означает, что мы доказали равенство двух величин. Мы можем лишь сказать «у нас недостаточно оснований отклонить нулевую гипотезу». Здесь полезно привести цитату Роналда Фишера, собственно, и предложившего идею «нулевой гипотезы»<sup>2</sup>: «...нулевая гипотеза никогда не доказывается и не принимается, но лишь с определенной вероятностью опровергается в ходе экспериментов. Можно сказать, что каждый эксперимент существует только для того, чтобы дать фактам шанс опровергнуть нулевую гипотезу».

<sup>2</sup> Она была предложена Фишером в рамках эксперимента «леди, дегустирующая чай»: [https://ru.wikipedia.org/wiki/Леди,\\_дегустирующая\\_чай](https://ru.wikipedia.org/wiki/Леди,_дегустирующая_чай).

Кроме того, следует помнить: результаты применения статистических критериев зависят от величины различий и от размера выборки, и одинаковые различия на выборках разного размера могут оказаться в одном случае незначимыми (например, если есть две выборки по 20 наблюдений), а в другом (когда наблюдений будет по 1000) – значимыми на том же уровне значимости. Здесь можно привести шуточную цитату Эндрю Гельмана: «Маловероятно, что маленькое  $n$  найдет маленькие  $p$ -значения. Большое  $n$ , вероятно, что-то да найдет. Ну а огромное  $n$  почти наверняка найдет множество маленьких  $p$ -значений».

С вероятностью ошибки II рода тесно связана другая величина, имеющая большое статистическое значение, – мощность критерия. Она вычисляется по формуле  $(1 - \beta)$  и характеризует способность критерия выявлять различия там, где они есть. Таким образом, чем выше мощность, тем меньше вероятность совершить ошибку II рода (проигнорировать различия там, где они есть).

**Таблица 3** Ошибка I рода и ошибка II рода

Решение относительно нулевой гипотезы $H_0$	Нулевая гипотеза $H_0$	
	Верна	Не верна
Нет оснований отклонить нулевую гипотезу $H_0$	Правильный вывод (Вероятность = $1 - \alpha$ )	<b>Ошибка II рода</b> (Вероятность = $\beta$ )
Отклонить нулевую гипотезу $H_0$	<b>Ошибка I рода</b> (Вероятность = $\alpha$ )	Правильный вывод (Вероятность = $1 - \beta$ )

Уменьшая  $\alpha$ , мы уменьшаем вероятность ошибки I рода (вероятность найти несуществующие различия), но повышаем вероятность ошибки II рода (вероятность проигнорировать различия, когда они есть). Таким образом, вероятности обеих ошибок обратно зависят друг от друга, и их нельзя минимизировать одновременно. По мере уменьшения вероятности одной ошибки увеличивается вероятность другой, и наоборот.

Для принятия решения о том, можно ли отклонить нулевую гипотезу и принять альтернативную, используют статистические критерии. Статистический критерий включает в себя метод расчета определенного показателя, на основании которого принимается решение об отклонении нулевой гипотезы, а также условия принятия решения. Этот рассчитываемый показатель называется эмпирическим (или экспериментальным) значением критерия. Найденное эмпирическое значение сравнивается с известным (например, заданным таблично или определенным с помощью той или иной статистической программы) эталонным числом, именуемым критическим значением критерия. В статистических таблицах критические значения приводятся, как правило, для нескольких уровней значимости: 5 % (0,05), 1 % (0,01) и др. Статистический критерий требует выполнения ряда предпосылок. Например, многие статистические критерии требуют нормальности распределения данных.

Статистический критерий зависит также от числа степеней свободы.

Количество степеней свободы – это количество значений в итоговом вычислении статистики, способных варьироваться. В третьем издании «Statistics in Plain English» дается следующее определение степеней свободы. «Грубо говоря, это минимальный объем данных, необходимый для расчета статистики. На практике это число или числа, используемые для аппроксимации количества наблюдений в наборе данных с целью определения статистической значимости». Количество степеней свободы рассчитывается как количество независимых значений, использованных при расчете статистики, минус количество рассчитанных статистик:

$$\text{количество степеней свободы} = \text{количество независимых значений} - \text{количество статистик.}$$

Например, у нас 50 наблюдений, и мы хотим вычислить выборочную статистику, например среднее. Все 50 наблюдений используются в вычислениях, и у нас одна статистика, таким образом, количество степеней свободы для расчета среднего будет равно  $50 - 1 = 49$ .

Нетрудно понять, что количество степеней свободы линейно зависит от объема выборки (например,  $df = n - 1$ ), а также от количества признаков или их градаций: чем больше эти показатели, тем больше число степеней свободы. Не существует единой формулы для определения числа степеней свободы для всех возможных случаев, поэтому статистический критерий также устанавливает формулу для расчета числа степеней свободы.

Для большинства статистических критериев действует следующее правило: если эмпирическое значение критерия для данного числа степеней свободы оказывается строго больше критического значения, соответствующего выбранному уровню значимости, то нулевая гипотеза отклоняется в пользу альтернативной с соответствующей достоверностью.

Критерий бывает параметрическим или непараметрическим. Параметрический критерий основывается на оценке параметров (таких как среднее или стандартное отклонение) распределения интересующей величины. Примерами параметрических критериев являются  $t$ -критерий Стьюдента, хи-квадрат Пирсона и др. Непараметрические методы не основываются на оценке параметров распределения. Примерами непараметрических критериев являются  $W$ -критерий Уилкоксона,  $Q$ -критерий Кохрена.

Критерий может быть предназначен для независимых выборок и зависимых выборок. Если можно установить гомоморфную пару (то есть когда одному случаю из выборки  $X$  соответствует один и только один случай из выборки  $Y$  и наоборот) для каждого случая в двух выборках (и это основание взаимосвязи является важным для измеряемого на выборках признака), такие выборки называются **зависимыми**. Примеры зависимых выборок: пары близнецов, два измерения какого-либо признака до и после экспериментального воздействия, мужа и жены. В случае если такая взаимосвязь между выборками отсутствует, то эти выборки считаются **независимыми**, например мужчины и женщины, психологи и математики.

Таким образом, общая процедура проверки статистической гипотезы включает в себя следующие шаги:

1. Сформулировать задачу.
2. Сформулировать нулевую и альтернативную гипотезы.
3. Выбрать требуемый уровень значимости.

4. Выбрать соответствующий статистический критерий (критерий проверки гипотезы), убедившись в выполнении условий его применимости.

5. Определить количество степеней свободы.

6. Вычислить эмпирическое значение критерия.

7. Сравнить эмпирическое значение критерия с критическим значением.

Давайте потренируемся в проверке статистических гипотез с помощью SciPy. У нас есть задача снизить смертность от сахарного диабета. Для оценки эффективности нового гипогликемического средства были проведены измерения уровня глюкозы в крови пациентов, страдающих сахарным диабетом, до и после приема препарата.

Давайте импортируем необходимые библиотеки и функции, загрузим данные (10 наблюдений) и взглянем на них.

```
# импортируем библиотеки, функции
import numpy as np
import pandas as pd
from scipy.stats import chi2_contingency, ttest_rel

# загружаем и смотрим данные
data = pd.read_csv('Data/glucose.csv', sep=';')
data
```

	before	after
0	9.6	5.7
1	8.1	4.2
2	8.8	6.4
3	7.9	5.5
4	9.2	5.3
5	8.0	4.2
6	8.4	5.1
7	10.1	5.9
8	7.8	7.5
9	8.1	5.0

Речь идет об измерениях признака (уровня глюкозы в крови) у пациента до и после экспериментального воздействия, и мы можем установить гомоморфную пару для каждого случая в двух выборках. Таким образом, речь идет о зависимых выборках.

Нулевая гипотеза  $H_0$  звучит так: средние значения двух генеральных совокупностей, из которых извлечены сравниваемые зависимые выборки, не отличаются друг от друга. Альтернативная гипотеза  $H_1$  звучит так: средние значения двух генеральных совокупностей, из которых извлечены сравниваемые зависимые выборки, отличаются друг от друга.

Задаем уровень значимости 0,05. Таким образом, если эмпирическое значение выбранного критерия для зависимых выборок равно или больше критического значения, соответствующего уровню значимости 0,05 (находим по

таблице критических значений соответствующего критерия), отклоняем нулевую гипотезу. Делаем вывод о наличии статистически значимых различий содержания глюкозы в крови до и после приема нового препарата при уровне значимости 0,05. Если значение рассчитанного критерия для зависимых выборок меньше табличного, значит, у нас нет достаточных оснований отклонить нулевую гипотезу. Делаем вывод об отсутствии статистически значимых различий содержания глюкозы в крови до и после приема нового препарата.

Мы применяем  $t$ -критерий Стьюдента для зависимых выборок, убедившись, что соблюдены условия его применимости. Критерий вычисляется по формуле:

$$t = \frac{M_d}{\sigma_d / \sqrt{n}},$$

где:

$M_d$  – среднее разностей показателей, измеренных до и после;

$\sigma_d$  – стандартное отклонение разностей показателей;

$n$  – количество исследуемых.

Для применения критерия необходимо, чтобы исходные данные имели нормальное распределение. Мы убедились в этом. Мы посмотрели скос (асимметрию) распределения и эксцесс (остроту пика распределения), необходимо, чтобы они были близки к нулю. Еще мы посмотрели медиану и среднее, для нормального распределения они будут совпадать.

*# проверяем данные на нормальность*

```
print('Скос признака before:', data['before'].skew())
print('Эксцесс признака before:', data['before'].kurtosis())
print('')
print('Скос признака after:', data['after'].skew())
print('Эксцесс признака after:', data['after'].kurtosis())
```

Скос признака before: 0.9058504758902429

Эксцесс признака before: -0.4489742150083589

Скос признака after: 0.6767808127701386

Эксцесс признака after: 0.8331238763353594

*# еще дополнительная проверка на нормальность*

```
data[['before', 'after']].describe()
```

	before	after
count	10.000000	10.000000
mean	8.600000	5.480000
std	0.794425	0.990847
min	7.800000	4.200000
25%	8.025000	5.025000
50%	8.250000	5.400000
75%	9.100000	5.850000
max	10.100000	7.500000

Теперь нужно найти число степеней свободы  $df$  по формуле:  $df = n - 1$ . В нашем случае будет 9 степеней свободы.

После этого по таблице критических значений определяем критическое значение  $t$ -критерия Стьюдента на пересечении строки – вычисленного числа степеней свободы  $df$  и столбца – требуемого уровня значимости. Нам нужно вычислить эмпирическое значение и сравнить его с критическим значением.

Итак, вычисляем эмпирическое значение. Сначала вычисляем разность каждой пары значений.

```
# вычисляем разность каждой пары значений
diff = data['before'] - data['after']
```

Вычисляем среднее разностей:

$$M_d = \frac{\sum d}{n}$$

```
# вычисляем среднее разностей
mean_diff = sum(diff) / len(data)
mean_diff
```

3.12

Вычисляем стандартное отклонение разностей от среднего:

$$\sigma_d = \sqrt{\frac{\sum (M_d - d)^2}{n-1}}$$

```
# вычисляем стандартное отклонение разностей от среднего
std_diff = np.sqrt((sum((mean_diff - diff) ** 2)) / (len(data) - 1))
std_diff
```

1.1773793875477105

```
# еще можно так
std_diff = diff.std()
std_diff
```

1.1773793875477105

Вычисляем  $t$ -критерий Стьюдента для зависимых выборок:

$$t = \frac{M_d}{\sigma_d / \sqrt{n}}$$

```
# вычисляем t-критерий Стьюдента для зависимых выборок
t = mean_diff / (std_diff / np.sqrt(len(data)))
t
```

8.379887064504546

Сравним эмпирическое значение 8,38 с критическим значением – табличным значением, которое при числе степеней свободы  $df$ , равном  $10 - 1 = 9$ , и уровне значимости 0,05 (доверительной вероятности 0,95) составляет 2,262

(выделено красным овалом). Видим, что полученное эмпирическое значение больше критического значения для уровня значимости 0,05.

Таблица 4 Таблица критических значений  $t$ -критерия Стьюдента

Число степеней свободы		Уровень значимости			
$df = n - 1$	$n$	0,1	0,05	0,01	0,001
1	2	6,31	12,71	63,66	636,62
2	3	2,92	4,30	9,92	31,60
3	4	2,35	3,18	5,84	12,92
4	5	2,13	2,78	4,60	8,61
5	6	2,02	2,57	4,03	6,87
6	7	1,94	2,45	3,71	5,96
7	8	1,89	2,36	3,50	5,41
8	9	1,86	2,31	3,36	5,04
9	10	1,83	2,26	3,25	4,78

Полученное эмпирическое значение больше критического, поэтому отклоняем нулевую гипотезу, т.е. делаем вывод о наличии статистически значимых различий содержания глюкозы в крови до и после приема нового препарата при  $\alpha = 0,05$ .

А теперь автоматически вычислим  $t$ -критерий Стьюдента для зависимых выборок с помощью функции `ttest_rel()` библиотеки SciPy.

```
# можно воспользоваться функцией ttest_rel()
# библиотеки SciPy
stats.ttest_rel(data['before'], data['after'])
```

```
Ttest_relResult(statistic=8.379887064504544, pvalue=1.5250840961461116e-05)
```

Видим, что наше эмпирическое значение соответствует  $p$ -значению 0,000015. Вероятность того, что  $t$ -статистика примет значение, не меньшее, чем вычисленное значение 8,38, когда средние значения генеральных совокупностей не отличаются друг от друга, составляет 0,000015. В 0,0015 % случаев из 100 мы рискуем допустить ошибку первого рода – отвергаем нулевую гипотезу, когда она верна (т.е. средние значения генеральных совокупностей равны).

Итак, наше  $p$ -значение меньше  $\alpha = 0,05$ . Отклоняем нулевую гипотезу, т.е. делаем вывод о наличии статистически значимых различий содержания глюкозы в крови до и после приема нового препарата при уровне значимости  $\alpha = 0,05$ .

Допустим, мы строим скоринговую модель, у нас есть признак *Семейное положение* с категориями Холост и Женат и зависимая переменная *Наличие просрочки* с категориями Нет просрочки и Есть просрочка. Мы хотим выяснить, существует ли связь между семейным положением и риском просрочки. Нулевая гипотеза звучит так: в генеральной совокупности категории признака не отличаются друг от друга с точки зрения распределения категорий зависимой переменной. Альтернативная гипотеза заключается в том, что в генеральной совокупности категории признака отличаются друг от друга с точки зрения распределения категорий зависимой переменной. Мы применим критерий хи-квадрат Пирсона.

Можно выделить три условия применимости критерия хи-квадрат:

- случайный выбор наблюдений;
- ожидаемые частоты менее 5 должны встречаться не более чем в 20 % ячейках таблицы;
- суммы по строкам и столбцам всегда должны быть больше нуля.

Критерий хи-квадрат подчиняется распределению хи-квадрат со степенями свободы  $df = (R - 1)(C - 1)$ , где  $R$  и  $C$  – количество строк и столбцов в таблице сопряженности. В нашем случае количество степеней свободы будет равно  $df = (2 - 1)(2 - 1) = 1$ .

Давайте вычислим эмпирическое значение критерия.

Мы строим двухходовую таблицу сопряженности, где строки являются категориями признака *Семейное положение*, а столбцы – категориями зависимой переменной *Наличие просрочки*. Для каждой ячейки таблицы фиксируем наблюдаемую частоту  $O$  (от *observed*). Затем для каждой ячейки фиксируем ожидаемую частоту  $E$  (от *expected*) согласно нулевой гипотезе. В итоге для каждой ячейки вычисляем квадрат разности между наблюдаемой и ожидаемой частотой, поделенный на ожидаемую частоту.

<b><i>O</i>(bserved)</b> Наблюдаемые частоты			
	Нет просрочки	Есть просрочка	Всего
Холост	20	13	33
Женат	30	37	67
Всего	50	50	100

<b><i>E</i>(xpected)</b> Ожидаемые частоты согласно нулевой гипотезе			
	Нет просрочки	Есть просрочка	Всего
Холост	16,50	16,50	33
Женат	33,50	33,50	67
Всего	50	50	100

$\frac{(O - E)^2}{E}$ для каждой ячейки		
	Нет просрочки	Есть просрочка
Холост	0,74	0,74
Женат	0,37	0,37

Нулевая гипотеза звучит так: категории предиктора не отличаются друг от друга с точки зрения распределения категорий зависимой переменной. Альтернативная гипотеза заключается в том, что категории предиктора все же отличаются друг от друга с точки зрения распределения категорий зависимой переменной.

**Рис. 14** Наблюдаемые и ожидаемые частоты для вычисления хи-квадрат

Складываем результаты, вычисленные по каждой ячейке, и получаем эмпирическое значение хи-квадрат ( $\chi^2$ ). Таким образом, критерий хи-квадрат Пирсона – это показатель, вычисляющий степень суммарного расхождения наблюдаемых (реальных) и ожидаемых частот:

$$\chi^2 = \sum \frac{(O_i - E_i)^2}{E_i},$$

где

$O_i$  – наблюдаемая частота  $i$ -й ячейки;

$E_i$  – ожидаемая частота  $i$ -й ячейки.

$$\chi^2 = \frac{(20-16,50)^2}{16,50} + \frac{(13-16,50)^2}{16,50} + \frac{(30-33,50)^2}{33,50} + \frac{(37-33,50)^2}{33,50} = 0,74 + 0,74 + 0,37 + 0,37 = 2,216.$$

Сравним эмпирическое значение 2,216 с критическим значением – табличным значением, которое при числе степеней свободы  $df$ , равном 1, и уровне значимости 0,05 (доверительной вероятности 0,95) составляет 3,84 (выделено красным овалом).

Таблица 5 Таблица критических значений критерия хи-квадрат

Степени свободы	$\alpha$								0.05	0.01	0.001
	0.95	0.90	0.80	0.70	0.50	0.30	0.20	0.10			
1	0.004	0.02	0.06	0.15	0.46	1.07	1.64	2.71	<b>3.84</b>	6.64	10.83
2	0.10	0.21	0.45	0.71	1.39	2.41	3.22	4.60	<b>5.99</b>	9.21	13.82
3	0.35	0.58	1.01	1.42	2.37	3.66	4.64	6.25	<b>7.82</b>	11.34	16.27
4	0.71	1.06	1.65	2.20	3.36	4.88	5.99	7.78	<b>9.49</b>	13.38	18.47
5	1.14	1.61	2.34	3.00	4.35	6.06	7.29	9.24	<b>11.07</b>	15.09	20.52
6	1.63	2.20	3.07	3.83	5.35	7.23	8.56	10.64	<b>12.59</b>	16.81	22.46
7	2.17	2.83	3.82	4.67	6.35	8.38	9.80	12.02	<b>14.07</b>	18.48	24.32
8	2.73	3.49	4.59	5.53	7.34	9.52	11.03	13.36	<b>15.51</b>	20.09	26.12
9	3.32	4.17	5.38	6.39	8.34	10.66	12.24	14.68	<b>16.92</b>	21.67	27.88
10	3.94	4.86	6.18	7.27	9.34	11.78	13.44	15.99	<b>18.31</b>	23.21	29.59

Полученное эмпирическое значение меньше критического значения, поэтому у нас нет оснований отклонить нулевую гипотезу. Можно сделать вывод, что категории переменной *Семейное положение* действительно не отличаются друг от друга с точки зрения распределения клиентов без просрочки и клиентов с просрочкой при уровне значимости  $\alpha = 0,05$ .

Если бы мы воспользовались SciPy, то быстро бы вычислили, что в нашем случае значение хи-квадрат 2,216 с одной степенью свободы соответствует  $p$ -значению 0,1366. Вероятность найти различия по зависимой переменной, когда их нет, составляет 0,1366. Это превышает уровень значимости  $\alpha = 0,05$ . Значит, у нас нет оснований отвергнуть нулевую гипотезу.

Использование распределения хи-квадрат для интерпретации критерия хи-квадрат Пирсона требует от нас предположения, что дискретное распределение наблюдаемых частот можно аппроксимировать непрерывным распределением хи-квадрат. Это предположение не вполне корректно и дает определенную ошибку.

Для уменьшения ошибки аппроксимации английский статистик Фрэнк Йейтс предложил поправку на непрерывность. Ее суть в следующем: аппроксимация распределения критерия хи-квадрат может быть улучшена понижением абсолютного значения разности между наблюдаемой и ожидаемой частотами на величину 0,5 перед возведением в квадрат.

$$\chi_{Yates}^2 = \sum \frac{(|O_i - E_i| - 0,5)^2}{E_i}$$

Это позволяет снизить значение хи-квадрат и таким образом увеличить  $p$ -значение.

Поправка Йейтса призвана избежать переоценки статистической значимости наибольших данных. Эта формула преимущественно используется тогда, когда хотя бы одна из ячеек в таблице имеет ожидаемую частоту меньше 5.

Давайте автоматически вычислим критерий хи-квадрат Пирсона с помощью функции `chi2_contingency()` модуля `stats` библиотеки SciPy. Функция `chi2_contingency()` имеет вид:

```
scipy.stats.chi2_contingency(observed,
                             correction=True,
                             lambda_=None)
```

Параметр `observed` задает таблицу сопряженности (массив NumPy или подобный массиву объект). Параметр `correction` задает поправку Йейтса (если задано значение `True` и число степени свободы равно 1). Параметр `lambda_` задает вычисляемый критерий (по умолчанию вычисляется критерий хи-квадрат Пирсона).

Функция возвращает значение критерия хи-квадрат, соответствующее  $p$ -значение, число степеней свободы и ожидаемые частоты (в виде массива NumPy).

```
# создаем двумерный массив - таблицу сопряженности
obs = np.array([[20, 13], [30, 37]])
# вычисляем значение критерия хи-квадрат, соответствующее
# p-значение, число степеней свободы и ожидаемые частоты,
# без поправки Йейтса
chi2_contingency(obs, correction=False)

(2.2161917684305745, 0.13656956347453514, 1, array([[16.5, 16.5], [33.5, 33.5]]))

# а еще можно было так
chi2, p, dof, ex = chi2_contingency(obs, correction=False)
```

Однако важно понимать, что мы проверяем статистические гипотезы не для того, чтобы просто получить  $p$ -значение. Как правило, исследователя интересует размер эффекта, то есть степень отклонения данных от нулевой гипотезы. Например, если эксперимент связан с проверкой способностей предсказателя будущего, то размер эффекта – это вероятность верного предсказания. Если проверяется эффективность лекарства, то размер эффекта – это вероятность выздоровления пациента, который это лекарство принимает, за вычетом эффекта плацебо. При запуске программы лояльности для пользователей интернет-магазина размер эффекта – это последующее увеличение среднего чека.

Размер эффекта – это величина, определенная на генеральной совокупности. Но, как правило, у исследователя есть только небольшая выборка из нее, а оценка размера эффекта по выборке – это случайная величина. Например,

высокий уровень значимости является показателем того, что такую оценку размера эффекта, какая получена по выборке, с очень маленькой вероятностью можно было получить случайно.

Достижимый уровень значимости зависит не только от размера эффекта, но и от объема выборки, по которой оценивается эффект. Вспоминаем вышеприведенное шутовское высказывание Эндрю Гельмана. Если выборка небольшая, скорее всего, нулевая гипотеза на ней не отвергается (если только она не слишком неразумная). Однако с ростом объема выборки начинают проявляться все более тонкие отклонения данных от нулевой гипотезы. Велика вероятность, что на достаточно большой выборке значительная часть разумных нулевых гипотез будет отвергнута. Именно поэтому, даже если нулевая гипотеза отвергнута, это еще не значит, что полученный эффект имеет какую-то практическую значимость, ее нужно оценивать отдельно.

Например, было проведено большое исследование, в рамках которого на протяжении трех лет у большой выборки женщин измеряли вес, а также оценивали, насколько активно они занимаются спортом. По итогам исследования выяснилось, что женщины, которые в течение этого времени упражнялись не меньше часа в день, набрали значительно меньше веса, чем женщины, которые упражнялись менее 20 минут в день. Статистическая значимость этого результата была достаточно высока:  $p < 0,001$ . Проблема заключалась в размере эффекта: разница в набранном весе между двумя исследуемыми группами женщин составила всего 150 граммов. 150 граммов за 3 года – это не очень много. Крайне сомнительно, что этот эффект имеет какую-то практическую значимость.

В медицинской практике часто досрочно прерывают испытания лекарств, если, например, обнаруживается, что прием препарата ведет к значимому увеличению опасных заболеваний (допустим, повышает риск инфаркта на 0,07 %). Эффект статистически значим, но при этом на первый взгляд кажется, что размер эффекта ничтожен. Однако если пересчитать размер эффекта на всю популяцию людей, которым этот препарат может быть выписан, результатом могут быть тысячи умерших. Поэтому разработчики такой препарат немедленно запрещают и снимают с рынка.

Этот пример показывает, что практическую значимость результата нельзя определить на глаз. В идеале она должна определяться человеком, разбирающимся в предметной области.

## **Задачи с собеседований (математическая статистика)**

1. В ходе исследования были опрошены 1000 человек. 20 % из них заинтересовались новым продуктом. Вычислите 95%-ный доверительный интервал для реальной доли заинтересованных в продукте.

2. Вычислите объем выборки, предельная ошибка которой составит 4 %. При этом мы принимаем 95%-ный доверительный уровень, генеральная совокупность значительно больше выборки, доля респондентов с наличием исследуемого признака равна 0,5. Принять, что выборка является простой случайной, объем выборки значительно меньше генеральной совокупности.

## 6. PANDAS

### 6.1. ПОЧЕМУ PANDAS?

pandas – одна из самых популярных библиотек для исследования данных с открытым исходным кодом, доступных в настоящее время. Она дает своим пользователям возможность исследовать, манипулировать, запрашивать, агрегировать и визуализировать табличные данные. Табличные данные относятся к двумерным данным, состоящим из строк и столбцов. Обычно мы называем такую организованную структуру данных таблицей. pandas – это инструмент, который мы будем использовать для анализа данных почти в каждом разделе этой книги.

Библиотека pandas была создана Уэсом МакКинни в 2008 году, когда он работал в хедж-фонде AQR. В финансовом мире принято называть табличные данные «панельными данными» (panel data), с которыми не всегда удобно работать, поскольку они часто являются громоздкими и неповоротливыми, как панды.

### 6.2. БИБЛИОТЕКА PANDAS ПОСТРОЕНА НА NUMPY

Все данные в pandas хранятся в массивах NumPy. Можно представить pandas как более высокоуровневый, более простой и удобный в использовании интерфейс для анализа данных, надстроенный над NumPy. Однако за это удобство приходится платить скоростью. Библиотека pandas стала сложной, избыточной, для одной и той же процедуры существуют десятки способов с разной вычислительной эффективностью, не решен ряд проблем, связанных с ложным срабатыванием предупреждений. Поэтому хорошая идея заключается в том, чтобы изучить основы NumPy, поработать в библиотеке pandas, найти задачи, которые быстро решаются в pandas, и задачи, которые решаются в pandas хуже, медленнее, и для таких задач применять NumPy, частично пожертвовав удобством в пользу скорости и уже более основательно изучив NumPy.

### 6.3. PANDAS РАБОТАЕТ С ТАБЛИЧНЫМИ ДАННЫМИ

Существует множество форматов данных, таких как XML, JSON, CSV, Parquet, текст и многие другие. Библиотека pandas умеет считывать данные, записанные в различных форматах, и всегда преобразовывает их в табличную форму. Библиотека pandas создана только для анализа этой прямоугольной, обманчиво нормальной концепции хранения данных. pandas не является подходящей библиотекой для обработки данных более чем в двух измерениях. Основное внимание уделяется данным, которые являются одномерными или двумерными.

### 6.4. ОБЪЕКТЫ DATAFRAME И SERIES

Объекты DataFrame и Series – это два основных объекта pandas, которые мы будем использовать в этой книге.

Объект Series – одно измерение данных. Его называют серией. Он аналогичен одному столбцу данных или одномерному массиву.

Объект DataFrame – это двумерная структура, таблица, похожая на электронную таблицу Excel со строками и столбцами. Для простоты эту таблицу называют датафреймом. В отличие от библиотеки NumPy, которая требует, чтобы все элементы в массиве были одного и того же типа, каждый столбец датафрейма (объект Series) может иметь отдельный тип, то есть в столбцах могут быть записаны строковые значения, даты, целые числа, числа с плавающей точкой. Датафрейм имеет две размерности, ось строк 0 (двигаемся по датафрейму вертикально) и ось столбцов 1 (двигаемся по датафрейму горизонтально). У датафрейма есть индекс, как правило, это последовательность целых чисел, начинающаяся с 0. Значения индекса не ограничиваются целыми значениями. Строки – это распространенный тип, который используется в индексе и обеспечивает более описательные метки.

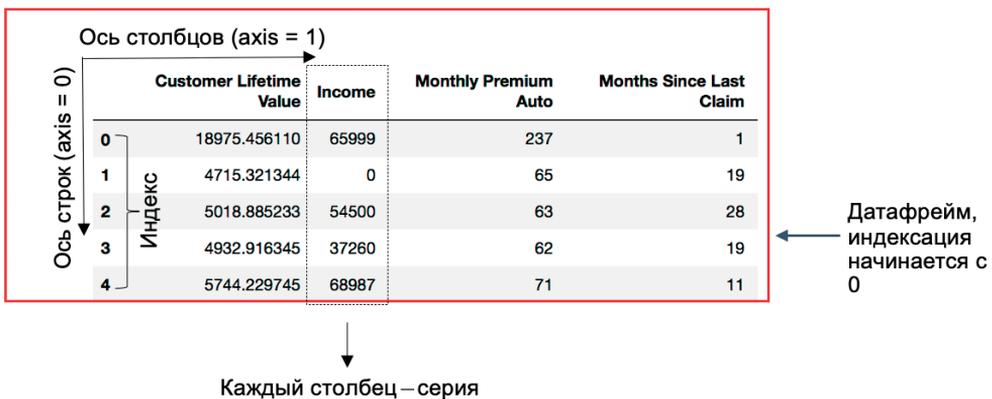


Рис. 15 Структура датафрейма pandas

При работе с различными методами и функциями pandas нам нужно будет указать ось, к которой будет применен метод или функция. Поясним на конкретном примере.

Давайте создадим датафрейм из двух столбцов.

*# импортируем библиотеки pandas и numpy*

```
import pandas as pd
import numpy as np
```

*# создаем датафрейм*

```
df = pd.DataFrame({'Empl': [10, 20],
                  'Age': [30, 40]})
```

```
df
```

	Empl	Age
0	10	30
1	20	40

Теперь с помощью метода `.mean()` вычислим среднее по строкам и вычислим среднее по столбцам (указываем `axis=0` и `axis=1` соответственно).

```
# вычисляем среднее по строкам
```

```
df.mean(axis=0)
```

```
Emp1    15.0
Age     35.0
dtype: float64
```

```
# вычисляем среднее по столбцам
```

```
df.mean(axis=1)
```

```
0     20.0
1     30.0
dtype: float64
```

Видим, что в зависимости от выбранной оси мы получаем разные результаты.

## 6.5. ЗАДАЧИ, ВЫПОЛНЯЕМЫЕ PANDAS

В `pandas` вам будут доступны следующие операции:

- чтение данных;
- доступ к строкам и столбцам;
- фильтрация данных;
- агрегация данных;
- очистка данных;
- изменение формы данных;
- анализ временных рядов;
- визуализация.

## 6.6. КРАТКО О ТИПАХ ДАННЫХ

Ниже приведены наиболее распространенные типы данных, которые часто применяются в датафреймах:

- `boolean` – только два возможных логических значения, `True` и `False`;
- `integer` – целые числа без десятичных знаков;
- `float` – числа с десятичными знаками (числа с плавающей точкой);
- `object` – почти всегда строки, но технически может содержать любой объект Python;
- `datetime` – конкретная дата и время с точностью до наносекунды.

Тип данных `object` является наиболее запутанным и заслуживает более подробного обсуждения. Каждое значение в столбце типа `object` может быть любым объектом Python. Столбцы типа `object` могут содержать целые числа, числа с плавающей запятой или даже структуры данных, такие как списки или словари. Что угодно может содержаться в столбцах объектов. Но почти всегда столбцы с типом данных `object` содержат только строки. Когда вы видите столбец с типом данных `object`, вы должны ожидать, что значения будут строками. Если у вас есть строки в значениях вашего столбца, тип данных будет `object`, но при этом вам не гарантируется, что все значения будут строками.

До выпуска pandas версии 1.0 не существовало выделенного типа данных `string`. Это было огромным ограничением и вызывало множество проблем. В pandas по-прежнему есть тип данных `object`, который может хранить строки.

С добавлением типа данных `string` мы гарантируем, что каждое значение будет строкой в столбце со строковым типом данных. Этот новый тип данных все еще помечен как «экспериментальный» в документации pandas, поэтому пока лучше не использовать его для серьезной работы. Есть много ошибок, которые необходимо исправить и отрегулировать поведение, прежде чем он будет готов к использованию. Поэтому в этой книге по-прежнему будет использоваться тип `object` для столбцов, содержащих строки.

## 6.7. ПРЕДСТАВЛЕНИЕ ПРОПУСКОВ

Наборы данных часто содержат пропущенные значения, и для их идентификации требуется некоторое представление. Pandas использует объекты `NaN` и `NaT` для представления пропусков:

- `NaN` (Not a Number) – «не является числом»;
- `NaT` (Not a Time) – «не является временем».

Представление пропущенного значения зависит от типа данных в столбце:

- `boolean` – нет представления пропуска;
- `integer` – нет представления пропуска;
- `float` – `NaN`;
- `object` – `NaN`;
- `datetime` – `NaT`.

Знание того, что столбец является либо `boolean`, либо `integer`, гарантирует, что в этом столбце нет пропусков, поскольку pandas не допускает их. Если, например, вы хотите поместить пропущенное значение в столбец типа `boolean` или `integer`, pandas преобразует столбец в столбец типа `float`. Это связано с тем, что столбец типа `float` может содержать пропуски. Когда логические значения преобразуются в числа с плавающей запятой, `False` становится равным 0, а `True` становится равным 1.

В pandas 1.0 теперь стали доступны новые типы данных: тип `integer`, допускающий значения `NULL`, тип `boolean`, допускающий значения `NULL`, тип `float`, допускающий значения `NULL`. Это совершенно новые типы данных, отличающиеся от исходных типов `integer`, `boolean`, `float`, и их поведение немного отличается. Основное отличие состоит в том, что они имеют представление пропущенного значения.

Раньше библиотека pandas использовала библиотеку Numpy для главного представления пропуска в виде `NaN`, которое продолжает существовать. С выпуском версии 1.0 разработчики pandas создали собственное представление пропуска `NA`. Это новое и экспериментальное дополнение, поэтому его поведение может измениться.

Главная рекомендация для pandas 1.0 заключается в том, чтобы с большой осторожностью использовать новый тип `string`, тип `integer`, допускающий значение `NULL`, тип `boolean`, допускающий значение `NULL`, а также `NA`, пока их не доработают. Они все еще являются экспериментальными, и их поведение может измениться.

## 6.8. КАКУЮ ВЕРСИЮ PANDAS ИСПОЛЬЗОВАТЬ?

Библиотека pandas находится в постоянном развитии и регулярно выпускает новые версии. В настоящее время pandas находится в основной версии 1, которая была выпущена в январе 2020 года. До основной версии 1 pandas была в версии 0. Библиотеки Python используют форму abc для нумерации версий, где a представляет номер мажорной версии. Он увеличивается всякий раз, когда происходят серьезные изменения, некоторые из которых несовместимы с предыдущими версиями. b представляет номер минорной версии и увеличивается с внесением небольших изменений и улучшений, совместимых с предыдущими версиями. c представляет номер микроверсии и увеличивается в основном при исправлении багов.

Часто, говоря о версии pandas, пишут только мажорную и минорную версии, поскольку микроверсия не так уж важна. Обычно в год выходит несколько минорных версий. Чтобы запустить код в этой книге, вам нужно запустить pandas 1.0 или более позднюю версию.

```
# смотрим версию
```

```
pd.__version__
```

```
'1.4.2'
```

## 6.9. ПОДРОБНО ЗНАКОМИМСЯ С ТИПАМИ ДАННЫХ

Прежде чем приступить к работе с данными, полезно подробно изучить типы данных, доступные в библиотеке pandas. Все значения в серии относятся к одному и тому же типу данных. Точно так же все значения отдельного столбца датафрейма относятся к одному и тому же типу данных. В этом разделе мы будем активно использовать метод `.astype()` для изменения типов данных.

### 6.9.1. Типы данных для работы с числами и логическими значениями

Начнем с типов данных, предназначенных для работы с числами и логическими значениями.

#### 6.9.1.1. Тип данных integer (тип для целых чисел, целочисленный тип), 'int64' или 'int32'

Давайте создадим серию, передав в функцию `pd.Series()` список целочисленных значений.

```
# создаем серию целочисленных значений
```

```
s_int = pd.Series([10, 35, 130])
```

```
s_int
```

```
0    10
```

```
1    35
```

```
2   130
```

```
dtype: int64
```

Вывод показывает тип данных для значений серии. В данном случае речь идет об `int64`, который формально представляет собой 64-битное целое число. Этот тип данных унаследован непосредственно от NumPy и позволяет целым числам иметь размер 8, 16, 32 или 64 бита. С помощью 64 бит мы можем представлять только целые числа от  $-9223372036854775808$  до  $9223372036854775807$ . В NumPy есть функция `np.iinfo()`, которая возвращает точную информацию о минимальном и максимальном целых числах для каждого целочисленного типа данных. Нужно просто передать в функцию нужный тип данных в виде строки.

```
# выводим диапазон чисел для типа int64
np.iinfo('int64')

iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)
```

Аналогично мы можем найти диапазон для 8-битовых целочисленных значений.

```
# выводим диапазон чисел для типа int8
np.iinfo('int8')

iinfo(min=-128, max=127, dtype=int8)
```

Диапазон чисел для `int8` охватывает от  $-128$  до  $127$ , или 256 чисел. Это эквивалентно двойке, возведенной в 8-ю степень.

С помощью метода `.astype()` сменим тип наших данных на `int8`.

```
# сменим тип на int8
s_int.astype('int8')

0    10
1    35
2   -126
dtype: int8
```

Обратите внимание, что третье значение теперь отображается как  $-126$  вместо исходного значения 130. Мы уже знаем, что максимальное 8-битное целое число равно 127. Библиотека NumPy предполагает, что вы знаете, что делаете, и не проверяет, что число 130 превышает максимум. Теперь наше число 130 представлено третьим целым числом, которое больше минимального значения  $-128$  и равно  $-126$ .

Тип целочисленного значения по умолчанию будет зависеть от операционной системы, в которой вы работаете. Для 32-разрядных машин Linux, macOS и Windows используются 32 бита. Для 64-разрядных машин Linux и macOS используются 64 бита. Для 64-разрядных машин Windows будут использоваться 32 бита.

### 6.9.1.2. Тип данных `unsigned integer` (тип для целых чисел без знака), `'uint64'` или `'uint32'`

Целочисленные типы данных по умолчанию делят половину своего диапазона на отрицательные и положительные целые числа. Можно ограничить ваши целые числа только неотрицательными целыми числами, используя тип `unsigned integer`, сокращенно `uint`. Доступны варианты 8, 16, 32 и 64 бита. Давайте преобразуем исходную серию `s_int` в тип `uint8`.

```
# сменим min на uint8
s_int.astype('uint8')
```

```
0    10
1    35
2   130
dtype: uint8
```

Последнее значение верно записано как 130, так как диапазон нашего нового типа данных составляет от 0 до 255. Давайте проверим это с помощью метода `.iinfo()`.

```
# выводим диапазон чисел для типа uint8
np.iinfo('uint8')
```

```
iinfo(min=0, max=255, dtype=uint8)
```

Целые числа без знака используются редко, но они доступны и могут быть полезны в определенных ситуациях, когда вы хотите сэкономить память. В остальных случаях в их использовании нет необходимости, поэтому использование целочисленного типа данных по умолчанию должно сработать.

### 6.9.1.3. Тип данных `nullable integer` (тип для целых чисел, допускающий значения `NULL`), `'Int64'`

С выпуском `pandas` версии 0.24 в конце 2019 года пользователям `pandas` стал доступен новый целочисленный тип данных, допускающий значение `NULL`. Этот новый тип данных допускает наличие пропусков в столбце целых чисел. Это отдельный тип данных, отличающийся от обычных целочисленных типов данных. Исходные целочисленные типы данных все еще существуют и не могут содержать пропуски. Давайте проверим это, попытавшись создать ряд целых чисел с пропусками. Обратите внимание, что мы используем параметр `dtype`, чтобы попытаться установить тип данных `int64`.

```
# создаем серию muna int64
pd.Series([10, 35, 130, np.nan], dtype='int64')
```

```
ValueError: cannot convert float NaN to integer
```

Если не использовать параметр `dtype`, то серия будет создана, но при этом будет задействован более гибкий тип данных `float64`, который допускает пропущенные значения.

```
# серии с пропусками будет присвоен тип float64
pd.Series([10, 35, 130, np.nan])

0    10.0
1    35.0
2   130.0
3     NaN
dtype: float64
```

Целочисленный тип данных, допускающий значения `NULL`, представлен строковым значением `'Int'` (в отличие от `'int'`). Важным отличием является первая заглавная буква `I`. Доступны те же четыре размера: 8, 16, 32 и 64. Давайте создадим серию целых чисел, допускающих значение `NULL`, используя строковое значение `Int64`.

```
# создаем серию с типом nullable integer (Int64)
s_nullable_int = pd.Series([10, 35, 130, np.nan],
                           dtype='Int64')

s_nullable_int

0     10
1     35
2    130
3  <NA>
dtype: Int64
```

Пропуск визуально представлен как значение `<NA>`, которое отличается от значения `NaN`, когда серия имела тип `float64`. Библиотека `pandas` предложила свой собственный объект `NA` для представления пропусков, который отличается от `NaN` библиотеки `numpy`. Библиотека `pandas` преобразует любой пропуск в серии целых чисел, допускающих значение `NULL`, в собственный объект `NA`. Давайте воспользуемся объектом `NA` библиотеки `pandas` непосредственно при создании серии, чтобы показать, что создается одна и та же серия.

```
# создаем серию с типом nullable integer (Int64)
pd.Series([10, 35, 130, pd.NA], dtype='Int64')

0     10
1     35
2    130
3  <NA>
dtype: Int64
```

Целочисленный тип данных, допускающий значения `NULL`, помечен как «экспериментальный», что указывает на то, что его поведение может измениться в будущем. Кроме того, встречаются некоторые ошибки, связанные с этим типом данных. Здесь можно порекомендовать с осторожностью ис-

пользовать этот тип данных для серьезной работы, пока он не перестанет быть экспериментальным. Помните, что этот тип доступен только в pandas, в NumPy этого типа нет.

Целочисленный тип данных, допускающий значения NULL, ведет себя иначе, чем обычный целочисленный тип данных. При попытке создать серию со значениями, которые не находятся в пределах ее диапазона, будет выброшено исключение вместо попытки вычисления значения, как это было сделано выше. Это, вероятно, наилучший вариант для предотвращения ошибок.

```
# создаем серию с типом nullable integer (Int8)
pd.Series([10, 35, 130], dtype='Int8')
```

**TypeError:** cannot safely cast non-equivalent int64 to int8

#### 6.9.1.4. Тип данных nullable unsigned integer (тип для целых чисел без знака, допускающий значения NULL), 'UInt64'

Целочисленный тип без знака, допускающий значения NULL, можно задать с помощью строкового значения 'UInt' (заглавные буквы U и I).

```
# создаем серию с типом nullable unsigned integer (UInt8)
pd.Series([10, 35, 130, pd.NA], dtype='UInt8')
```

```
0      10
1      35
2     130
3     <NA>
dtype: UInt8
```

#### 6.9.1.5. Тип данных float (тип для чисел с плавающей точкой), 'float64' или 'float32'

Столбцы с плавающей точкой содержат числа с десятичными знаками. По умолчанию используется 64-битовый тип float. Это числовой тип данных в NumPy, который используется для хранения чисел с плавающей точкой двойной точности (double precision). Стандартное значение с плавающей точкой двойной точности (хранящееся во внутреннем представлении объекта Python типа float) занимает 8 байт, или 64 бита. Поэтому соответствующий тип в NumPy называется float64. В NumPy есть дополнительные 16-битовый и 32-битовый типы float. Все типы чисел с плавающей запятой могут содержать пропущенные значения. Давайте создадим серию чисел с плавающей точкой, содержащую один пропуск, и проверим тип данных.

```
# создаем серию с типом float64
s_float = pd.Series([5.26, 1234.56789, np.nan])
s_float
```

```
0      5.26000
1    1234.56789
2           NaN
dtype: float64
```

Мы можем присвоить серии тип `float32`, который используется для хранения чисел с плавающей точкой одинарной точности (single precision).

```
# присвоим типу float32
s_float.astype('float32')

0      5.260000
1    1234.567871
2           NaN
dtype: float32
```

Опять с помощью функции `np.finfo()` получим информацию о типе `float`. Тип `float32` гарантирует точность 6 значащих цифр, как это можно увидеть с помощью атрибута `resolution` ниже.

```
# выведем диапазон чисел и точность для типа float32
np.finfo('float32')

finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)
```

Тип `float16`, который используется для хранения чисел с плавающей точкой половинной точности (half precision), гарантирует только 3 цифры точности.

```
# выведем диапазон чисел и точность для типа float16
np.finfo('float16')

finfo(resolution=0.001, min=-6.55040e+04, max=6.55040e+04, dtype=float16)
```

Переход к этому типу данных существенно изменяет второе фактическое значение из-за его ограниченной точности.

```
# присвоим типу float16
s_float.astype('float16')

0      5.261719
1    1235.000000
2           NaN
dtype: float16
```

Мы можем изменить тип с `float` на `integer` и наоборот. Ниже мы попытаемся перейти от `float64` к `int64`. Сделать это не удастся, так как обычный целочисленный тип данных не допускает пропущенных значений.

```
# переведем из float64 в int64
s_float.astype('int64')
```

```
IntCastingNaNError: Cannot convert non-finite values (NA or inf) to integer
```

Удалив пропуски, мы сможем выполнить преобразование. Десятичные знаки отсекаются, и числа не округляются.

```
# удалим пропуски и переведем из float64 в int64
```

```
s_float.dropna().astype('int64')
```

```
0      5
1    1234
dtype: int64
```

Поведение типа `nullable integer` отличается. Он не позволяет выполнить преобразование, если есть какие-либо числа с десятичными знаками.

```
# присвоим mun nullable integer (Int64)
```

```
s_float.astype('Int64')
```

```
TypeError: cannot safely cast non-equivalent float64 to int64
```

Если удалить десятичные знаки (с помощью округления), то преобразование в тип `nullable integer` станет возможным.

```
# округлим и присвоим mun nullable integer (Int64)
```

```
s_float.round(0).astype('Int64')
```

```
0      5
1    1235
2    <NA>
dtype: Int64
```

Теперь выполним преобразование из типа `int` в тип `float`.

```
# преобразовываем из мунa int64 в мун float64
```

```
s_int.astype('float64')
```

```
0    10.0
1    35.0
2    130.0
dtype: float64
```

Преобразование типа `nullable integer` в тип `float` тоже возможно. Поскольку тип `float` является типом данных NumPy, он использует для представления пропусков значение `NaN` вместо `pd.NA`.

```
# преобразовываем из мунa nullable integer (Int64)
```

```
# в мун float64
```

```
s_nullable_int.astype('float64')
```

```
0    10.0
1    35.0
2    130.0
3     NaN
dtype: float64
```

### 6.9.1.6. Тип данных nullable float (тип для чисел с плавающей точкой, допускающий значения NULL), 'Float64'

С выходом pandas 1.2 (декабрь 2020 г.) в библиотеке появился тип nullable float. Его можно задать с помощью строкового значения 'float' (заглавная F), за которым следует размер в битах – 16 или 32. Сейчас мы выполним преобразование из типа float в тип nullable float.

```
# преобразовываем из numpy float
# в numpy nullable float (Float64)
nullable_float = s_float.astype('Float64')
nullable_float

0          5.26
1    1234.56789
2          <NA>
dtype: Float64
```

### 6.9.1.7. Тип данных boolean (логический тип, булев тип), 'bool'

Логические значения имеют один 8-битовый тип данных в NumPy. Давайте создадим серию логических значений.

```
# создадим серию логических значений
s_bool = pd.Series([True, False])
s_bool

0      True
1     False
dtype: bool
```

Мы можем выполнить преобразование из типов int и float в тип bool и наоборот. Единственное значение, которое будет преобразовано в False, – это 0. Все остальные значения будут преобразованы в True. Используйте строковое значение 'bool' для преобразования в логический тип. Давайте создадим серию с типом данных integer и выполним преобразование в логический тип.

```
# создаем серию с типом integer
s = pd.Series([0, 1, 59, -35])

# преобразовываем в numpy boolean
s.astype('bool')

0     False
1      True
2      True
3      True
dtype: bool
```

Давайте создадим серию с типом данных float и выполним преобразование в логический тип. Здесь тоже только значение 0 будет преобразовано в False. Любое другое значение оценивается как True.

```
# создаем серию с типом float
s = pd.Series([0, 0.0001, -3.99])

# преобразовываем в тип boolean
s.astype('bool')

0    False
1     True
2     True
dtype: bool
```

Преобразование серии логических значений в серию с целыми числами или числами с плавающей точкой превратит все значения True в значения 1, а все значения False – в значения 0.

```
# преобразуем из типа boolean в тип integer
s_bool.astype('int64')

0    1
1    0
dtype: int64
```

Использование типа int64 для хранения логического значения является излишним. Для экономии памяти можно воспользоваться наименьшим целочисленным типом, int8 (или uint8).

#### 6.9.1.8. Тип данных nullable boolean (логический тип, допускающий значения NULL), 'Boolean'

С выпуском pandas 1.0 для поддержки пропусков стал доступен новый логический тип, допускающий значения NULL. Тип nullable boolean есть только в pandas, исходный тип boolean по-прежнему существует, но не поддерживает пропуски. Давайте убедимся, что исходный тип boolean не может содержать пропуски.

```
# создаем серию с типом boolean
s = pd.Series([True, False, np.nan], dtype='bool')
s

0     True
1    False
2     True
dtype: bool
```

Выполнение кода не приводит к ошибке, вместо этого объект nan библиотеки NumPy превращается в значение True. Это соответствует правилу, согласно которому каждое ненулевое значение и пропуск оцениваются как значение True для логических значений.

Если взять серию логических значений и присвоить одному из значений значение nan, то вся серия получит тип object.

```
# присвоение значения nan одному из логических
# значений дает серию с типом object
s.loc[0] = np.nan
s
```

```
0    NaN
1    False
2     True
dtype: object
```

Новый тип `nullable boolean` использует строковое значение `'boolean'` вместо `'bool'`. Давайте создадим серию с типом `nullable boolean`.

```
# создаем серию с типом nullable boolean
s = pd.Series([True, False, np.nan], dtype='boolean')
s

0     True
1    False
2    <NA>
dtype: boolean
```

Выполнение арифметических операций с серией может изменить тип данных в полученной серии. Деление всегда преобразует серию с данными типа `integer` в серию с данными типа `float`, даже если результатом являются целые числа.

```
# создаем серию с типом integer
s = pd.Series([-15, 45])
s

0    -15
1     45
dtype: int64

# выполняем деление, получаем
# серию с типом float
s / 15

0    -1.0
1     3.0
dtype: float64
```

Используя деление с округлением до целого значения вниз (`floor division`), мы получим результат в виде целого значения, пока делитель является целым числом.

```
# используем деление с округлением
# до целого значения вниз
s // 77

0    -1
1     0
dtype: int64
```

Умножение серии целочисленных значений на значение с плавающей точкой дает серию значений с плавающей точкой, даже если все результирующие значения являются целыми числами.

```
# выполняем умножение
```

```
s * 4.4
0    -66.0
1    198.0
dtype: float64
```

Все преобразования типов данных в этом разделе были выполнены с использованием строковых значений типа 'int8'. Существует альтернативный подход. Вместо строкового значения вы можете использовать сам фактический объект, доступный непосредственно из NumPy или pandas. Например, мы можем использовать `np.int8` вместо строкового значения 'int8', чтобы указать тип данных.

```
# задаем min int8 max
```

```
pd.Series([10, 50]).astype(np.int8)
0     10
1     50
dtype: int8
```

```
# а еще можно так
```

```
pd.Series([10, 50]).astype('int8')
0     10
1     50
dtype: int8
```

Все типы данных NumPy имеют то же самое имя, что и их аналоги в виде строковых значений. Однако для типов данных pandas это не выполняется. Типы данных pandas заканчиваются словом 'dtype'. Например, для преобразования в 32-битовый тип nullable integer вы можете использовать `pd.Int32Dtype()`.

```
# создаем серию с типом nullable integer (Int32)
```

```
pd.Series([10, 50, np.nan]).astype(pd.Int32Dtype())
0     10
1     50
2    <NA>
dtype: Int32
```

Для преобразования в 64-битовый тип nullable float вы можете использовать `pd.Float64Dtype()`.

```
# создаем серию с типом nullable float (Float64)
```

```
pd.Series([7.3, 5.8, np.nan], dtype=pd.Float64Dtype())
0     7.3
1     5.8
2    <NA>
dtype: Float64
```

Ниже приводится таблица типов данных для работы с числами и логическими значениями, которые унаследованы библиотекой pandas от NumPy, и таблица

типов данных для работы с числами и логическими значениями, имеющих только в pandas.

**Таблица 6** Типы данных для работы с числами и логическими значениями, унаследованные библиотекой pandas от NumPy

Название	Короткое название в виде строкового значения по умолчанию	Размеры (количество битов)
<b>Boolean</b>	bool	8
<b>Integer</b>	int64 или int32	8, 16, 32, 64
<b>Unsigned Integer</b>	uint64 или uint32	8, 16, 32, 64
<b>Float</b>	float64	16, 32, 64

Пропуски доступны в типе float в виде np.nan.

**Таблица 7** Типы данных для работы с числами и логическими значениями, имеющиеся только в pandas

Название	Короткое название в виде строкового значения по умолчанию	Название объекта pandas по умолчанию	Размеры (количество битов)
<b>Nullable Boolean</b>	Boolean	pd.BooleanDtype()	8
<b>Nullable Integer</b>	Int64	pd.Int64Dtype()	8, 16, 32, 64
<b>Nullable Unsigned Integer</b>	UInt64	pd.UInt64Dtype()	8, 16, 32, 64
<b>Nullable Float</b>	Float64	pd.Float64Dtype()	32, 64

Пропуски доступны во всех типах в виде pd.NA.

Теперь разберем типы данных для работы со строками.

## 6.9.2. Типы данных для работы со строками

Теперь разберем типы данных для работы со строками.

### 6.9.2.1. Тип данных object (объектный тип), 'object'

До выхода версии 1.0 у pandas не было специального строкового типа данных. Вместо этого использовался тип данных object для хранения строк. Как упоминалось ранее, тип данных object не имеет ограничений относительно того, какой объект Python может быть внутри него. По сути, это универсальное средство для любого элемента, который вы хотите разместить в датафрейме, который не принадлежит к другим конкретным типам данных.

У типа данных object нет определенного размера в битах. Не существует object64, есть только один тип данных object. Каждый элемент может быть разного типа и, следовательно, разного размера.

Хотя тип данных object может содержать любой объект Python, в основном он используется для хранения строк. Давайте создадим серию с парой строковых значений.

```
# создаем серию со строковыми значениями
s_object = pd.Series(['some', 'strings'])
s_object

0      some
1    strings
dtype: object
```

Как видно из вывода, тип данных – 'object'. Если проверить тип, мы увидим в выводе dtype('O'). Тип данных object тоже унаследован непосредственно от NumPy, в которой используется обозначение 'O' вместо полного названия.

```
# проверим тип
s_object.dtype

dtype('O')
```

Поскольку тип 'object' является наиболее гибким типом, серии с любым типом данных можно присвоить тип 'object'. Ниже мы присвоим серии с целыми числами тип 'object'.

```
# присвоим серии с целыми числами тип object
s = pd.Series([5, 10])
s.astype('object')

0      5
1     10
dtype: object
```

Однако сами значения по-прежнему являются целыми числами. Мы убедимся в этом, найдя тип первого значения.

```
# значения – по-прежнему целые числа
type(s.loc[0])

numpy.int64
```

Серия с типом object может содержать все, что угодно. Серия ниже включает в себя список, логическое значение, строку, число с плавающей точкой и словарь.

```
# серия с типом object может содержать все, что угодно
garbage_series = pd.Series([[1,2], True, 'some string',
                           4.5, {'key': 'value'}])
garbage_series

0      [1, 2]
1         True
2    some string
3         4.5
4  {'key': 'value'}
dtype: object
```

```
# элементом серии с типом object
# может быть все, что угодно
print(type(garbage_series.loc[0]))
print(type(garbage_series.loc[1]))
print(type(garbage_series.loc[2]))
print(type(garbage_series.loc[3]))
print(type(garbage_series.loc[4]))
```

```
<class 'list'>
<class 'bool'>
<class 'str'>
<class 'float'>
<class 'dict'>
```

Несмотря на то что вы можете разместить любой объект Python в серии, обычно это считается плохой практикой. Серии с типом данных `object` предназначены для хранения строк.

### 6.9.2.2. Тип данных `Categorical` (категориальный тип), `'category'`

Теперь познакомимся с категориальным типом данных, который есть в `pandas` и отсутствует в `NumPy`. Категориальный тип данных часто используется, когда столбец данных имеет известные, ограниченные и дискретные значения.

Давайте загрузим данные и отберем для манипуляций столбец `job_position`.

```
# записываем CSV-файл в объект DataFrame
credit = pd.read_csv('Data/credit_train.csv',
                    encoding='cp1251',
                    decimal=',',
                    sep=';')
```

```
# выводим первые 5 наблюдений датафрейма
credit.head()
```

	client_id	gender	age	marital_status	job_position	credit_sum	credit_month	tariff_id	score_shk	education	living_region	monthly_income
0	1	M	NaN	NaN	UMN	59998.00	10	1.6	NaN	GRD	КРАСНОДАРСКИЙ КРАЙ	30000.0
1	2	F	NaN	MAR	UMN	10889.00	6	1.1	NaN	NaN	МОСКВА	NaN
2	3	M	32.0	MAR	SPC	10728.00	12	1.1	NaN	NaN	ОБЛ САРАТОВСКАЯ	NaN
3	4	F	27.0	NaN	SPC	12009.09	12	1.1	NaN	NaN	ОБЛ ВОЛГОГРАДСКАЯ	NaN
4	5	M	45.0	NaN	SPC	NaN	10	1.1	0.421385	SCH	ЧЕЛЯБИНСКАЯ ОБЛАСТЬ	NaN

```
# смотрим частоты категорий job_position
job_position = credit['job_position']
job_position.value_counts()
```

```
SPC    134680
UMN    17674
BIS     5591
PNA    4107
DIR    3750
ATP    2791
WRK     656
```

```

NOR      537
WOI      352
INP      241
BIU      126
WRP      110
PNI      65
PNV      40
PNS      12
HSK      8
INV      5
ONB      1
Name: job_position, dtype: int64

```

Общее количество категорий должно быть **известно**. Вероятность появления новых категорий в будущем должна быть низкой. Общее количество категорий **ограничено** и намного меньше количества наблюдений. Значения должны быть **дискретными**. Если эти условия соблюдаются, рассматриваемый столбец можно перевести в категориальный тип.

Самый простой способ присвоить серии категориальный тип – передать строковое значение 'category' методу .astype().

```

# присваиваем тип Categorical
job_position_cat = job_position.astype('category')
job_position_cat.head()

0    UMN
1    UMN
2    SPC
3    SPC
4    SPC
Name: job_position, dtype: category
Categories (18, object): ['ATP', 'BIS', 'BIU', 'DIR', ..., 'UMN', 'WOI', 'WRK', 'WRP']

```

Убедимся в том, что серии присвоен тип Categorical.

```

# смотрим тип серии
job_position_cat.dtype

CategoricalDtype(categories=['ATP', 'BIS', 'BIU', 'DIR', 'HSK', 'INP', 'INV', 'NOR',
                             'ONB', 'PNA', 'PNI', 'PNS', 'PNV', 'SPC', 'UMN', 'WOI',
                             'WRK', 'WRP'],
                 , ordered=False)

```

Чем полезен тип Categorical?

Категориальные данные хранятся намного эффективнее, чем объектные. Каждое уникальное значение в столбце типа Categorical сохраняется один раз независимо от того, сколько раз оно повторяется в серии, и каждое из уникальных значений имеет целочисленный код, который на него ссылается. Именно эти целые числа хранятся в памяти для представления данных.

Столбцы типа object хранят каждое значение в уникальной локации памяти. Например, строка 'SPC' появляется более 134 000 раз в серии job\_position\_cat. Каждая из этих строк хранится в уникальной локации памяти. Использование

целых чисел для представления категорий может сэкономить огромное количество памяти.

Давайте создадим упрощенный пример, чтобы показать, как pandas хранит категориальные данные внутри, используя списки Python. В этом примере у нас будет три уникальных строковых значения. Они сохраняются ровно один раз в списке `cats` ниже. Фактические данные хранятся в списке значений, содержащем значения 0, 1 и 2.

```
# создаем 2 списка
cats = ['Python', 'Java', 'Scala']
vals = [1, 1, 0, 2, 0, 1, 2, 2, 1, 2, 1]
```

Список `cats`, по сути, работает как сопоставление целочисленной локации со строковым значением. Целое число 0 соответствует 'Python', 1 – 'Java' и 2 – 'Scala'. Мы можем преобразовать каждое значение в списке `vals` в соответствующую категорию, используя генератор списков.

```
# выполняем сопоставление
[cats[val] for val in vals]
```

```
['Java',
 'Java',
 'Python',
 'Scala',
 'Python',
 'Java',
 'Scala',
 'Scala',
 'Scala',
 'Java',
 'Scala',
 'Java']
```

Уникальную последовательность категорий можно получить с помощью средства доступа (аксессора) `.cat` и атрибута `categories`.

```
# выведем уникальный список категорий
job_position_cat.cat.categories

Index(['ATP', 'BIS', 'BIU', 'DIR', 'HSK', 'INP', 'INV', 'NOR', 'ONB', 'PNA',
      'PNI', 'PNS', 'PNV', 'SPC', 'UMN', 'WOI', 'WRK', 'WRP'],
      dtype='object')
```

Соответствующие целочисленные коды для категорий можно извлечь с помощью атрибута `codes`. Обратите внимание: для хранения используется тип `int8`.

```
# смотрим целочисленные коды
job_position_cat.cat.codes.head()
```

```
0    14
1    14
2    13
3    13
```

```
4 13
dtype: int8
```

Одним из самых больших преимуществ использования категориальных столбцов является экономия памяти. Вместо использования строки под каждое значение используется целочисленный код. Целые числа занимают значительно меньше места, чем строки. Кроме того, библиотека pandas использует наименьший размер целочисленного типа для хранения кодов. Например, если категорий меньше 128, используется `int8`.

С помощью метода `.memory_usage()` можно выяснить, сколько памяти позволяет сэкономить использование типа `Categorical`. Чтобы получить точные данные об объеме использованной памяти, для параметра `deep` нужно задать значение `True`.

```
# объем памяти для хранения серии типа object
orig_mem = job_position.memory_usage(deep=True)
orig_mem
```

```
10244888
```

```
# объем памяти для хранения серии типа Categorical
cat_mem = job_position_cat.memory_usage(deep=True)
cat_mem
```

```
172510
```

Сравним скорость выполнения операции приравнивания для обеих серий.

```
# выполним операцию приравнивания
# для серии типа object
%timeit -n 5 -r 2 job_position == 'SPC'
```

```
9.24 ms ± 266 µs per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

```
# выполним операцию приравнивания
# для серии типа Categorical
%timeit -n 5 -r 2 job_position_cat == 'SPC'
```

The slowest run took 4.77 times longer than the fastest. This could mean that an intermediate result is being cached.

```
231 µs ± 151 µs per loop (mean ± std. dev. of 2 runs, 5 loops each)
```

Любой столбец, независимо от его типа данных, может быть преобразован в категориальный. Целые числа – это основной нестроковый тип данных, который используется для представления категориальных данных. Вот несколько примеров целочисленных категориальных данных:

- рейтинг фильма/отеля/ресторана с учетом того, что диапазон известен, например целые числа (1–5);
- почтовые индексы определенного города;
- категория силы урагана (1–5).

Давайте присвоим серии с целочисленными значениями тип `Categorical`.

```
# присвоим серии с целочисленными значениями тип Categorical
credit_month_cat = credit['credit_month'].astype('category')
credit_month_cat.head(10)
```

```
0    10
1     6
2    12
3    12
4    10
5    10
6     6
7    10
8    12
9    10
```

```
Name: credit_month, dtype: category
Categories (31, int64): [3, 4, 5, 6, ..., 30, 31, 32, 36]
```

### 6.9.2.3. Тип данных string (строковый тип), 'string'

С выходом версии 1.0 в pandas стал доступен новый тип данных string. Этот тип есть только в pandas и отсутствует в NumPy. Он может содержать только строки и пропуски. Опять же, используйте его с осторожностью, пока он является экспериментальным.

Для создания серии с этим типом мы можем передать строковое значение 'string' в метод .astype(). Вы также можете напрямую использовать объект pandas pd.StringDtype. Обе серии будут идентичны.

```
# создаем серию с типом string
s_string = pd.Series(['Python', 'Java', 'Scala', pd.NA],
                    dtype='string')
```

```
s_string

0    Python
1     Java
2    Scala
3     <NA>
dtype: string
```

```
# создаем серию с типом string
s_string = pd.Series(['Python', 'Java', 'Scala', pd.NA],
                    dtype=pd.StringDtype())
```

```
s_string

0    Python
1     Java
2    Scala
3     <NA>
dtype: string
```

Предполагаемая цель строкового типа данных состоит в том, чтобы, наконец, предложить пользователям pandas тип данных, который гарантированно будет содержать только строки (и пропуски). Это должно уменьшить количество ошибок, поскольку тип данных object может содержать все, что угодно.

*# серия с типом string может содержать только строки и пропуски*

```
garbage_series = pd.Series([[1,2], True, 'some string', 4.5,
                           {'key': 'value'}])
garbage_series = garbage_series.astype('string')
garbage_series
```

```
0          [1, 2]
1             True
2         some string
3              4.5
4   {'key': 'value'}
dtype: object
```

*# значения уже будут строками*

```
print(type(garbage_series.loc[0]))
print(type(garbage_series.loc[1]))
print(type(garbage_series.loc[2]))
print(type(garbage_series.loc[3]))
print(type(garbage_series.loc[4]))
```

```
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
<class 'str'>
```

Вместе с тем функциональность обоих типов данных будет очень похожей. Здесь мы применим средство доступа (аксессора) `.str`, чтобы сделать строки прописными.

*# сделаем буквы заглавными*

```
s_string.str.upper()
```

```
0   PYTHON
1     JAVA
2     SCALA
3     <NA>
dtype: string
```

Строки, полностью состоящие из чисел, можно преобразовать либо в целое число, либо в число с плавающей точкой. Давайте создадим серию строк, которые выглядят точно так же, как числа с плавающей точкой. Библиотека `pandas` всегда использует тип `object` в качестве типа данных по умолчанию для строк.

*# создаем серию со строками, выглядящими как числа*

```
s = pd.Series(['4.5', '3.19'])
s
```

```
0    4.5
1    3.19
dtype: object
```

Кавычки для строк отсутствуют в выводе, поэтому строки кажутся значениями с плавающей точкой. Но можно заметить, что десятичные дроби не выровнены и каждое значение имеет разное количество цифр после запятой. Давайте создадим фактический столбец типа `float`, чтобы вы могли увидеть разницу в визуальном отображении. Обратите внимание, что десятичные дроби всегда будут выровнены.

```
# переводим в тип float64
```

```
s.astype('float64')
```

```
0    4.50
1    3.19
dtype: float64
```

Теперь представьте, у вас есть серия строковых значений, некоторые из которых могут быть преобразованы в числовые, а другие – нет. В этой ситуации невозможно использовать метод `.astype()`.

```
# создаем серию со строковыми значениями
```

```
s = pd.Series(['4.5', '3.19', 'NO ANSWER'])
```

```
s
0      4.5
1      3.19
2    NO ANSWER
dtype: object
```

```
# переводим в тип float64
```

```
s.astype('float64')
```

```
ValueError: could not convert string to float: 'NO ANSWER'
```

Вместо этого нужно обратиться к функции `to_numeric()`, которая работает аналогично методу `.astype()`, но при этом у нее есть дополнительная возможность принудительно выполнить преобразование. Это можно сделать, задав для параметра `errors` значение `'coerce'`. Любое значение, которое нельзя преобразовать, будет записано как пропуск.

```
# выполняем преобразование в тип float64
```

```
pd.to_numeric(s, errors='coerce')
```

```
0    4.50
1    3.19
2     NaN
dtype: float64
```

Вы можете преобразовать все значения в строки с помощью строкового значения `'str'` или встроенного класса `str`. Давайте создадим серию с целыми числами, а затем преобразуем их в строки с помощью строкового значения `'str'`. Серия получит тип `object`.

```
# серии с целыми числами присваиваем тип object
# с помощью строкового значения str
s = pd.Series([10, 20, 99])
s.astype('str')

0    10
1    20
2    99
dtype: object
```

С помощью атрибута `values`, который возвращает массив NumPy, проверим, являются ли наши значения строками.

```
# проверим, являются ли наши значения строками
s.astype('str').values

array(['10', '20', '99'], dtype=object)
```

Мы можем воспользоваться строковым значением `'string'` для преобразования в **новый тип string**.

```
# преобразовываем в тип string
s.astype('string')

0    10
1    20
2    99
dtype: string
```

Ниже приводится таблица типов данных для работы со строками.

**Таблица 8** Типы данных Object, Categorical и String

Название	Короткое название в виде строкового значения по умолчанию	Размеры (количество битов)	Замечания
<b>Object</b>	object str	Любой	Может содержать любой питоновский объект
<b>String</b>	string	Любой	Может содержать только строки
<b>Categorical</b>	category	Наименьший по размеру тип Integer, позволяющий хранить все имеющиеся категории	

## 6.10. ЧТЕНИЕ ДАННЫХ

Функция `pd.read_csv()` может считывать данные, хранящиеся в виде обычного текста, разделенного разделителем. По умолчанию разделителем является запятая. Ниже приведены ее основные параметры.

```
pandas.read_csv(filepath_or_buffer, ← путь к файлу
                sep=',', ← символ – разделитель полей (по умолчанию ,)
                delimiter=',', ← символ – разделитель полей (по умолчанию ,)
                header='infer', ← номер строки, содержащей имена столбцов (если имена не передаются,
                                аналогично header=0, и имена берутся из первой строки файла)
                names=None, ← список с именами столбцов
                index_col=None, ← столбец, значения которого будут использоваться в качестве меток строк датафрейма
                usecols=None, ← подмножество столбцов
                squeeze=False, ← если прочитанные данные содержат лишь один столбец, возвращает объект Series
                prefix=None, ← добавляет префикс к номерам столбцов без имени (например, 'X' для X0, X1)
                dtype=None, ← тип данных в столбцах (например, {'a': np.float64, 'b': np.int32, 'c': 'Int64'})
                skiprows=None, ← список с номерами строк (индексация с 0) или количество строк
                                (целочисленное значение), которое нужно пропустить с начала файла
                skipfooter=0, ← количество строк (целочисленное значение), которое нужно пропустить с конца файла
                nrows=None, ← количество строк (целочисленное значение), которое нужно прочитать, полезно при
                                чтении больших файлов частями
                na_values=None ← список со строковыми значениями, которые нужно распознать как NA/NaN (можно передать
                                словарь, где ключом будет словарь, значением – строковое значение для пропуска)
                parse_dates=None, ← выполняет парсинг дат
                date_parser ← парсер дат
                decimal=',', ← задает символ – десятичный разделитель (по умолчанию .)
                encoding=None) ← задает тип кодировки
```

объявлены  
устаревшими

Обратите внимание, что параметр `squeeze`, использующийся для превращения датафрейма с одним столбцом в серию (актуально при работе с данными, представляющими временной ряд), объявлен устаревшим. Теперь к функции нужно будет добавить метод `.squeeze('columns')`.

```
# загружаем ежемесячные данные
# о продажах автомобилей
cars = pd.read_csv('Data/monthly_car_sales.csv',
                  header=0,
                  index_col=0,
                  squeeze=True,
                  parse_dates=True)
cars.head()
```

```
# загружаем ежемесячные данные
# о продажах автомобилей
cars = pd.read_csv('Data/monthly_car_sales.csv',
                  header=0,
                  index_col=0,
                  parse_dates=True).squeeze('columns')
cars.head()
```

```
Month
1960-01-01    6550
1960-02-01    8728
1960-03-01   12026
1960-04-01   14395
1960-05-01   14587
Name: Sales, dtype: int64
```

Давайте с помощью функции `pd.read_csv()` прочитаем общедоступные данные об использовании велосипедов в городе Чикаго в датафрейм `pandas` с именем `bikes`.

По каждому наблюдению (поездке) фиксируются следующие переменные (характеристики):

- количественная переменная *Пол* [`gender`];
- переменная даты и времени *Дата и время начала поездки* [`starttime`];
- переменная даты и времени *Дата и время конца поездки* [`stoptime`];
- количественная переменная *Продолжительность поездки* [`tripduration`];
- категориальная переменная *Название станции – начала поездки* [`from_station_name`];
- категориальная переменная *Название станции – конца поездки* [`to_station_name`];
- количественная переменная *Емкость в стартовой точке* [`start_capacity`];
- количественная переменная *Емкость в конечной точке* [`end_capacity`];
- количественная переменная *Температура* [`temperature`];
- количественная переменная *Скорость ветра* [`wind_speed`];
- категориальная переменная *Тип погодного явления во время поездки* [`events`].

С помощью метода `.head()` выведем первые 3 наблюдения.

```
# загружаем данные
```

```
bikes = pd.read_csv('Data/bikes.csv')
```

```
bikes.head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	11.0	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	31.0	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	15.0	Dearborn St & Monroe St	23.0	73.0	16.1	mostlycloudy

Последняя строка блока кода часто будет заканчиваться методом `.head()`. По умолчанию этот метод возвращает первые пять строк `DataFrame` или `Series`. Цель этого метода – ограничить вывод, чтобы он легко умещался на экране или странице книги. Если метод `.head()` не используется, то `pandas` по умолчанию отображает первые и последние 5 строк данных (или все строки, если `DataFrame` содержит 60 строк или меньше). Чтобы еще больше сократить вывод (для экономии места на экране), методу `.head()` можно передать целое число (обычно 3). Это целое число определяет количество возвращаемых строк.

## 6.11. ПОЛУЧЕНИЕ ОБЩЕЙ ИНФОРМАЦИИ О ДАТАФРЕЙМЕ

С помощью свойства `shape` выведем информацию о количестве наблюдений и количестве переменных.

```
# смотрим количество наблюдений
```

```
# и количество переменных
```

```
print(bikes.shape)
```

```
(50089, 11)
```

С помощью функции `len()` выведем информацию о количестве наблюдений.

```
# смотрим количество наблюдений
print(len(bikes))
```

```
50089
```

С помощью свойства `dtypes` выведем информацию о типе данных.

```
# смотрим типы данных
bikes.dtypes
```

```
gender           object
starttime        object
stoptime         object
tripduration     int64
from_station_name object
start_capacity   float64
to_station_name  object
end_capacity     float64
temperature      float64
wind_speed       float64
events           object
dtype: object
```

По умолчанию `pandas` читает столбцы, содержащие строки, как столбцы типа `object`.

Из визуализации датафрейма видно, что столбцы `starttime` и `stoptime` являются датой и временем. Однако результаты выше показывают, что переменные имеют тип `object`. К сожалению, функция `pd.read_csv()` не считывает эти столбцы автоматически как дату и время. Она требует, чтобы вы передали список столбцов, которые являются `datetime`, параметру `parse_dates`, иначе функция будет считывать эти переменные как строки. Давайте перечитаем данные, используя параметр `parse_dates`.

```
# заново читаем данные, парсим даты
bikes = pd.read_csv('Data/bikes.csv',
                   parse_dates=['starttime', 'stoptime'])
bikes.dtypes.head()
```

```
gender           object
starttime        datetime64[ns]
stoptime         datetime64[ns]
tripduration     int64
from_station_name object
dtype: object
```

С помощью метода `.info()` выведем информацию о типе данных и количестве непропущенных наблюдений для каждой переменной.

```
# выведем информацию о типе данных и количестве
# непропущенных наблюдений для каждой переменной
```

```
bikes.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50089 entries, 0 to 50088
Data columns (total 11 columns):
#   Column                Non-Null Count  Dtype
---  -
0   gender                 50089 non-null  object
1   starttime              50089 non-null  object
2   stoptime               50089 non-null  object
3   tripduration          50089 non-null  int64
4   from_station_name     50089 non-null  object
5   start_capacity        50083 non-null  float64
6   to_station_name       50089 non-null  object
7   end_capacity          50077 non-null  float64
8   temperature           50089 non-null  float64
9   wind_speed            50089 non-null  float64
10  events                50089 non-null  object
dtypes: float64(4), int64(1), object(6)
memory usage: 4.2+ MB
```

С помощью свойства `columns` можно вывести информацию об именах столбцов.

```
# выведем имена столбцов
print(bikes.columns.tolist())
```

```
['gender', 'starttime', 'stoptime', 'tripduration', 'from_station_name', 'start_capacity',
'to_station_name', 'end_capacity', 'temperature', 'wind_speed', 'events']
```

## 6.12. ИЗМЕНЕНИЕ НАСТРОЕК ВЫВОДА С ПОМОЩЬЮ ФУНКЦИИ `GET_OPTIONS()`

С помощью функции `get_options()` можно настроить максимальное количество отображаемых столбцов, максимальное количество отображаемых строк, максимальную ширину столбца.

```
# максимальное количество столбцов
pd.get_option('display.max_columns')

20

# максимальное количество строк
pd.get_option('display.max_rows')

60

# максимальная ширина столбца
pd.get_option('display.max_colwidth')

50

# задаем новые настройки
pd.set_option('display.max_columns', 30,
              'display.max_rows', 100)
```

## 6.13. ЗНАКОМСТВО С ИНДЕКСАТОРАМИ [], loc и iloc

Библиотека pandas предлагает индексоы [], loc и iloc для выбора подмножеств данных:

- df[]
- df.loc[]
- df.iloc[]

Одной из самых распространенных ошибок при использовании loc и iloc является добавление к ним круглых скобок вместо квадратных. Одна из основных причин этой ошибки заключается в том, что loc и iloc кажутся методами, а все методы вызываются в круглых скобках. И loc, и iloc не являются методами, но доступ к ним осуществляется так же, как и к методам, через точечную нотацию, что приводит к ошибке.

В Python квадратные скобки являются универсальным оператором для выбора подмножеств данных независимо от типа объекта. Квадратные скобки выбирают подмножества списков, строк и выбирают одно значение в словаре. Массивы NumPy используют оператор квадратных скобок для выбора подмножества. Помните, что если вы делаете выбор подмножества, скорее всего, вам нужны квадратные, а не круглые скобки.

Давайте отберем один столбец.

```
# отберем один столбец
bikes['gender']

0          Male
1          Male
2          Male
3          Male
4          Male
...
50084      Male
50085      Male
50086      Male
50087      Female
50088      Male
Name: gender, Length: 50089, dtype: object
```

Давайте извлечем несколько столбцов, передав индексоору [] список. Обратите внимание, что внутренние квадратные скобки мы используем для списка, а внешние квадратные скобки – для отбора подмножества.

```
# извлекаем несколько столбцов, передав
# индексоору [] список
bikes[['gender', 'tripduration']]
```

	gender	tripduration
0	Male	993
1	Male	623
2	Male	1040
3	Male	667
4	Male	130
...	...	...
50084	Male	1625
50085	Male	585
50086	Male	824
50087	Female	178
50088	Male	214

50089 rows x 2 columns

Индексатор `loc` в первую очередь выбирает подмножества по **меткам** строк и столбцов. Он также делает выбор с помощью логического отбора.

В `loc` мы можем передать:

- отдельную метку;
- список меток;
- срез с метками;
- булеву серию.

Давайте отберем первые две строки в столбцах `gender` и `tripduration`. Для этого передаем в индексатор `loc` список меток строк (в нашем случае они являются целочисленными), а затем список имен столбцов.

```
# отбираем первые две строки столбцов
# start_capacity и tripduration, передав
# в loc список строк, список столбцов
bikes.loc[[0, 1], ['start_capacity', 'tripduration']]
```

	start_capacity	tripduration
0	11.0	993
1	31.0	623

В Python существует нотация среза, которая используется для выбора подмножеств из некоторых основных объектов Python, таких как списки, кортежи и строки. Нотация среза всегда состоит из трех компонентов – `start` (стартовое значение), `stop` (конечное значение) и `step` (шаг). Синтаксически каждый компонент отделяется двоеточием следующим образом: `start:stop:step`. Все компоненты нотации среза являются необязательными, и их необязательно включать. У каждого компонента есть значение по умолчанию, если оно не включено в нотацию. У компонента `start` есть стартовое значение, у компонента `stop` – последнее значение, компонент `step` задает размер шага 1. По сути, работаем как с диапазоном, который всегда задается внутри квадратных скобок.

Давайте отберем первые четыре строки в столбцах `gender` по `tripduration`, передав в `loc` диапазон строк и диапазон столбцов. Обратите внимание, как мы

задали диапазон строк 0:3, это эквивалентно 0:3:1, потому что у step всегда есть значение по умолчанию 1 и мы можем опустить его 0:3:1. Аналогично с диапазоном столбцов.

```
# отбираем первые четыре строки столбцов
# с gender по tripduration, передав
# в loc диапазон строк, диапазон столбцов
bikes.loc[0:3, 'gender':'tripduration']
```

	gender	starttime	stoptime	tripduration
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667

Давайте отберем каждую 2-ю строку каждого 2-го столбца, передав в loc диапазон строк и диапазон столбцов. Обратите внимание, как мы задали диапазон строк 0::2, это эквивалентно 0:50088:2, потому что у stop всегда есть конечное значение по умолчанию (последняя строка с меткой индекса 50088) и мы можем опустить его 0:50088:2.

```
# отберем каждую 2-ю строку каждого 2-го столбца,
# передав в loc диапазон строк, диапазон столбцов
bikes.loc[0::2, 'gender':'events':2]
```

	gender	stoptime	from_station_name	to_station_name	temperature	events
0	Male	2013-06-28 19:17:00	Lake Shore Dr & Monroe St	Michigan Ave & Oak St	73.9	mostlycloudy
2	Male	2013-06-30 15:01:00	Sheffield Ave & Kingsbury St	Dearborn St & Monroe St	73.0	mostlycloudy
4	Male	2013-07-01 11:18:00	Damen Ave & Pierce Ave	Damen Ave & Pierce Ave	73.0	partlycloudy
6	Male	2013-07-02 17:56:00	Clark St & Randolph St	Ravenswood Ave & Irving Park Rd	66.0	cloudy
8	Male	2013-07-03 15:42:00	Clinton St & Washington Blvd	Wood St & Division St	71.1	cloudy
...	...	...	...	...	...	...
50080	Male	2017-12-29 15:51:00	Cityfront Plaza Dr & Pioneer Ct	Mies van der Rohe Way & Chestnut St	14.0	snow
50082	Male	2017-12-30 10:53:00	Larrabee St & Oak St	Halsted St & Blackhawk St (*)	3.9	mostlycloudy
50084	Male	2017-12-30 13:34:00	State St & Pearson St	Clark St & Elm St	5.0	partlycloudy
50086	Male	2017-12-30 13:48:00	Kingsbury St & Kinzie St	Halsted St & Blackhawk St (*)	5.0	partlycloudy
50088	Male	2017-12-31 15:26:00	Clarendon Ave & Leland Ave	Clifton Ave & Lawrence Ave	10.9	partlycloudy

25045 rows x 6 columns

```
# а можно было так
```

```
bikes.loc[0:50088:2, 'gender': 'events':2]
```

	gender	stoptime	from_station_name	to_station_name	temperature	events
0	Male	2013-06-28 19:17:00	Lake Shore Dr & Monroe St	Michigan Ave & Oak St	73.9	mostlycloudy
2	Male	2013-06-30 15:01:00	Sheffield Ave & Kingsbury St	Dearborn St & Monroe St	73.0	mostlycloudy
4	Male	2013-07-01 11:18:00	Damen Ave & Pierce Ave	Damen Ave & Pierce Ave	73.0	partlycloudy
6	Male	2013-07-02 17:56:00	Clark St & Randolph St	Ravenswood Ave & Irving Park Rd	66.0	cloudy
8	Male	2013-07-03 15:42:00	Clinton St & Washington Blvd	Wood St & Division St	71.1	cloudy
...	...	...	...	...	...	...
50080	Male	2017-12-29 15:51:00	Cityfront Plaza Dr & Pioneer Ct	Mies van der Rohe Way & Chestnut St	14.0	snow
50082	Male	2017-12-30 10:53:00	Larrabee St & Oak St	Halsted St & Blackhawk St (*)	3.9	mostlycloudy
50084	Male	2017-12-30 13:34:00	State St & Pearson St	Clark St & Elm St	5.0	partlycloudy
50086	Male	2017-12-30 13:48:00	Kingsbury St & Kinzie St	Halsted St & Blackhawk St (*)	5.0	partlycloudy
50088	Male	2017-12-31 15:26:00	Clarendon Ave & Leland Ave	Clifton Ave & Lawrence Ave	10.9	partlycloudy

25045 rows x 6 columns

А сейчас мы выполним отбор, начиная с пятой строки и столбца *from\_station\_name*, передав в *loc* диапазон строк и диапазон столбцов. Мы задали диапазон строк 4:, это эквивалентно 4:50088:1. Поскольку у *stop* и *step* есть значения по умолчанию, мы можем опустить их 4:50088:1. Мы задали диапазон столбцов 'from\_station\_name:', это эквивалентно 'from\_station\_name':'events':1. Вновь, поскольку у *stop* и *step* есть значения по умолчанию (последний столбец 'events' и размер шага 1), мы можем опустить их 'from\_station\_name':'events':1.

```
# отбираем, начиная с пятой строки и столбца
```

```
# from_station_name, передав в loc диапазон
```

```
# строк, диапазон столбцов
```

```
bikes.loc[4:, 'from_station_name':]
```

	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
4	Damen Ave & Pierce Ave	19.0	Damen Ave & Pierce Ave	19.0	73.0	17.3	partlycloudy
5	California Ave & 21st St	15.0	Clark St & Wrightwood Ave	15.0	73.0	17.3	mostlycloudy
6	Clark St & Randolph St	31.0	Ravenswood Ave & Irving Park Rd	19.0	66.0	15.0	cloudy
7	State St & Van Buren St	27.0	Franklin St & Jackson Blvd	27.0	64.0	5.8	cloudy
8	Clinton St & Washington Blvd	31.0	Wood St & Division St	15.0	71.1	0.0	cloudy
...	...	...	...	...	...	...	...
50084	State St & Pearson St	27.0	Clark St & Elm St	27.0	5.0	16.1	partlycloudy
50085	Halsted St & 35th St (*)	16.0	Union Ave & Root St	11.0	5.0	16.1	partlycloudy
50086	Kingsbury St & Kinzie St	31.0	Halsted St & Blackhawk St (*)	20.0	5.0	16.1	partlycloudy
50087	Clinton St & Lake St	23.0	Kingsbury St & Kinzie St	31.0	7.0	11.5	partlycloudy
50088	Clarendon Ave & Leland Ave	15.0	Clifton Ave & Lawrence Ave	15.0	10.9	15.0	partlycloudy

50085 rows x 7 columns

Теперь мы отберем столбцы *start\_capacity* и *tripduration* (т.е. все строки в столбцах *start\_capacity* и *tripduration*), передав в *loc* список нужных столбцов

после нотации двоеточия с запятой. Двоеточие перед списком столбцов как раз обозначает отбор всех строк.

```
# отбираем столбцы start_capacity
# и tripduration, передав в loc список
# столбцов после двоеточия с запятой
bikes.loc[:, ['start_capacity', 'tripduration']]
```

	start_capacity	tripduration
0	11.0	993
1	31.0	623
2	15.0	1040
3	19.0	667
4	19.0	130
...	...	...
50084	27.0	1625
50085	16.0	585
50086	31.0	824
50087	23.0	178
50088	15.0	214

50089 rows × 2 columns

Отберем диапазон столбцов, передав в loc диапазон столбцов после нотации двоеточия с запятой.

```
# отбираем диапазон столбцов с помощью loc
bikes.loc[:, 'gender':'tripduration']
```

	gender	starttime	stoptime	tripduration
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623
2	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667
4	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130
...	...	...	...	...
50084	Male	2017-12-30 13:07:00	2017-12-30 13:34:00	1625
50085	Male	2017-12-30 13:34:00	2017-12-30 13:44:00	585
50086	Male	2017-12-30 13:34:00	2017-12-30 13:48:00	824
50087	Female	2017-12-31 09:30:00	2017-12-31 09:33:00	178
50088	Male	2017-12-31 15:22:00	2017-12-31 15:26:00	214

50089 rows × 4 columns

Двоеточие можно использовать и для отбора нужных строк.

Давайте отберем строки с метками индекса 1, 5 и 6, передав в loc список строк перед запятой с двоеточием. Двоеточие после списка строк уже будет обозначать отбор всех столбцов.

```
# отберем строки с метками индекса 1, 5 и 6
bikes.loc[[1, 5, 6], :]
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
1	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	31.0	Wells St & Walton St	19.0	69.1	6.9	partlycloudy
5	Male	2013-07-01 12:37:00	2013-07-01 12:48:00	660	California Ave & 21st St	15.0	Clark St & Wrightwood Ave	15.0	73.0	17.3	mostlycloudy
6	Male	2013-07-02 17:47:00	2013-07-02 17:56:00	565	Clark St & Randolph St	31.0	Ravenswood Ave & Irving Park Rd	19.0	66.0	15.0	cloudy

Давайте отберем диапазон строк, передав в loc диапазон строк перед запятой с двоеточием.

```
# отберем каждую 10-ю строку
bikes.loc[0::10, :]
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
0	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	11.0	Michigan Ave & Oak St	15.0	73.9	12.7	mostlycloudy
10	Male	2013-07-04 17:17:00	2013-07-04 17:42:00	1523	Morgan St & 18th St	15.0	Damen Ave & Pierce Ave	19.0	79.0	9.2	mostlycloudy
20	Female	2013-07-09 17:39:00	2013-07-09 17:55:00	943	State St & Van Buren St	27.0	State St & 16th St	15.0	82.9	9.2	mostlycloudy
30	Male	2013-07-12 12:32:00	2013-07-12 12:41:00	512	Jefferson St & Monroe St	19.0	Morgan St & Lake St	15.0	81.0	4.6	partlycloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	19.0	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
...	...	...	...	...	...	...	...	...	...	...	...
50040	Male	2017-12-22 11:11:00	2017-12-22 11:27:00	961	Ada St & Washington Blvd	15.0	Dearborn St & Adams St	19.0	39.0	4.6	cloudy
50050	Male	2017-12-22 16:33:00	2017-12-22 16:45:00	708	Lincoln Ave & Roscoe St	19.0	Damen Ave & Charleston St	11.0	37.0	6.9	hazy
50060	Male	2017-12-23 15:31:00	2017-12-23 15:38:00	412	Western Ave & Winnebago Ave	15.0	Spaulding Ave & Armitage Ave	15.0	26.1	11.5	partlycloudy
50070	Male	2017-12-27 20:02:00	2017-12-27 20:06:00	275	Halsted St & Roscoe St	15.0	Clark St & Wellington Ave	15.0	1.0	0.0	partlycloudy
50080	Male	2017-12-29 15:28:00	2017-12-29 15:51:00	1378	Cityfront Plaza Dr & Pioneer Ct	23.0	Mies van der Rohe Way & Chestnut St	19.0	14.0	6.9	snow

5009 rows x 11 columns

Индексатор iloc очень похож на индексатор loc, но использует целочисленную позицию для создания своего подмножества. Само слово iloc обозначает integer location (целочисленная позиция) и напомним, что оно делает.

Целочисленная позиция – это термин, используемый для ссылки на строку или столбец. На первую строку/столбец ссылается целое число 0. На каждую последующую строку/столбец ссылается следующее целое число. На последнюю строку/столбец ссылается число  $n - 1$ , где  $n - 1$  – количество строк/столбцов.

В iloc мы можем передать:

- отдельное целочисленное значение;
- список целочисленных значений;
- срез с целочисленными значениями.

Однако, в отличие от loc, в iloc невозможен логический отбор.

Давайте используем нотацию среза для отбора строк с целочисленными позициями 2 и 3 и список для отбора столбцов с целочисленными позициями 2 и 3. Обратите внимание, что в `iloc` целочисленная позиция `step` всегда **исключается**. В `loc` целочисленная позиция `step` всегда **включена**. Итак, передаем диапазон строк и диапазон столбцов (срез с целочисленными значениями строк и срез с целочисленными значениями столбцов) в `iloc`.

```
# отбираем строки с индексами 2 и 3
# и столбцы с индексами 2 и 3
bikes.iloc[2:4, 2:4]
```

	stoptime	tripduration
2	2013-06-30 15:01:00	1040
3	2013-07-01 10:16:00	667

Теперь отберем столбцы с индексами 3 и 5, передав в `iloc` список столбцов после двоеточия с запятой. Двоеточие перед списком столбцов вновь обозначает отбор всех строк.

```
# отбираем столбцы с индексами 3 и 5, передав в
# iloc список столбцов после двоеточия с запятой
bikes.iloc[:, [3, 5]]
```

	tripduration	start_capacity
0	993	11.0
1	623	31.0
2	1040	15.0
3	667	19.0
4	130	19.0
...	...	...
50084	1625	27.0
50085	585	16.0
50086	824	31.0
50087	178	23.0
50088	214	15.0

50089 rows × 2 columns

Теперь отберем строки с индексами 3 и 5, передав в `iloc` список строк перед запятой с двоеточием. Напомним, что двоеточие после списка строк обозначает отбор всех столбцов.

```
# отбираем строки с индексами 3 и 5, передав в
# iloc список строк перед запятой с двоеточием
bikes.iloc[[3, 5], :]
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
3	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	19.0	Clark St & Randolph St	31.0	72.0	16.1	mostlycloudy
5	Male	2013-07-01 12:37:00	2013-07-01 12:48:00	660	California Ave & 21st St	15.0	Clark St & Wrightwood Ave	15.0	73.0	17.3	mostlycloudy

Библиотека `pandas` предлагает еще два редко встречающихся индекса: `at` и `iat`. Эти индексы аналогичны `loc` и `iloc` соответственно, но выбирают только одну ячейку датафрейма. Поскольку они выбирают только одну ячейку, вы должны передать строку и столбец либо в виде метки (`loc`), либо в виде целочисленной позиции (`iloc`).

Давайте с помощью `at` отберем строку с меткой индекса 3 и столбец `tripduration`.

```
# отбираем строку с меткой индекса 3 и
# столбец tripduration
bikes.at[3, 'tripduration']
```

667

Давайте с помощью `at` отберем строку с индексом 3 и столбец с индексом 3 (это будет столбец `tripduration`).

```
# отбираем строку с индексом 3 и
# столбец с индексом 3
bikes.iat[3, 3]
```

## 6.14. ФИЛЬТРАЦИЯ ДАННЫХ

Библиотека `pandas` может отфильтровать строки датафрейма в зависимости от того, соответствуют ли значения в этой строке условию. Например, мы можем выбрать только те поездки, продолжительность которых превышает 5000 (секунд).

### 6.14.1. Одно условие

Ниже мы приведем пример фильтрации на основе одного условия, которое проверяется для каждой строки. Возвращаются только те строки, которые удовлетворяют этому условию.

```
# отбор по одному условию
filt = bikes['tripduration'] > 5000
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	35.0	Millennium Park	35.0	79.0	13.8	cloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	19.0	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	15.0	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy

## 6.14.2. Несколько условий

Мы можем отфильтровать данные на основе нескольких условий. В следующем примере мы отберем поездки женщин с продолжительностью поездки более 5000 (секунд).

```
# отбор по нескольким условиям
filt1 = bikes['tripduration'] > 5000
filt2 = bikes['gender'] == 'Female'
filt = filt1 & filt2
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	19.0	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	15.0	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy
1954	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	LaSalle St & Washington St	15.0	Theater on the Lake	15.0	44.1	12.7	clear

Итак, мы отобрали поездки женщин с продолжительностью поездки более 5000 (секунд).

В следующем примере несколько условий, но требуется, чтобы только одно из условий было истинным. Мы возвращаем все строки, в которых либо поездку на велосипеде совершает женщина, либо продолжительность поездки превышает 5000.

```
# только одно из условий является истинным
filt = filt1 | filt2
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
9	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	Lakeview Ave & Fullerton Pkwy	19.0	Racine Ave & Congress Pkwy	19.0	81.0	12.7	mostlycloudy
14	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	Morgan St & Lake St	15.0	Aberdeen St & Jackson Blvd	15.0	82.0	5.8	mostlycloudy
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	35.0	Millennium Park	35.0	79.0	13.8	cloudy

## 6.14.3. Несколько условий в одном столбце

Мы можем отфильтровать данные на основе нескольких условий одного столбца.

```
# несколько условий в одном столбце events
filt = ((bikes['events'] == 'rain') |
        (bikes['events'] == 'snow') |
        (bikes['events'] == 'tstorms') |
        (bikes['events'] == 'sleet'))
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	15.0	State St & Harrison St	19.0	82.9	5.8	rain
78	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	Michigan Ave & Pearson St	23.0	Millennium Park	35.0	82.4	11.5	tstorms
79	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	Carpenter St & Huron St	19.0	Carpenter St & Huron St	19.0	82.4	11.5	tstorms

Как вариант можно воспользоваться методом `.isin()`.

```
# несколько условий в одном столбце events,
# используем isin
filt = bikes['events'].isin(['rain', 'snow',
                             'tstorms', 'sleet'])
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	15.0	State St & Harrison St	19.0	82.9	5.8	rain
78	Male	2013-07-21 16:35:00	2013-07-21 17:06:00	1809	Michigan Ave & Pearson St	23.0	Millennium Park	35.0	82.4	11.5	tstorms
79	Male	2013-07-21 16:47:00	2013-07-21 17:03:00	999	Carpenter St & Huron St	19.0	Carpenter St & Huron St	19.0	82.4	11.5	tstorms

Можно скомбинировать метод `.isin()` с каким-нибудь другим фильтром.

```
# сочетание isin и дополнительного фильтра
filt1 = bikes['events'].isin(['rain', 'snow',
                              'tstorms', 'sleet'])
filt2 = bikes['tripduration'] > 2000
filt = filt1 & filt2
bikes[filt].head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
2344	Female	2014-03-19 07:23:00	2014-03-19 08:00:00	2181	Seeley Ave & Roscoe St	11.0	Franklin St & Lake St	23.0	43.0	6.9	rain
7697	Male	2014-09-12 14:20:00	2014-09-12 14:57:00	2213	Damen Ave & Pierce Ave	19.0	California Ave & Division St	15.0	52.0	12.7	rain
8357	Male	2014-09-30 08:21:00	2014-09-30 08:58:00	2246	Damen Ave & Melrose Ave	11.0	Wood St & Taylor St	15.0	46.9	11.5	rain

## 6.14.4. Использование метода `.query()`

Метод `.query()` предоставляет альтернативный и часто более читаемый способ фильтрации данных, чем описанные выше. Строка с условием просто передается в метод `.query()` для фильтрации данных.

```
# отбор по одному условию
bikes.query('tripduration > 5000').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	35.0	Millennium Park	35.0	79.0	13.8	cloudy
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	19.0	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	15.0	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy

```
# отбор по нескольким условиям
bikes.query('tripduration > 5000 and gender=="Female"').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
40	Female	2013-07-14 14:08:00	2013-07-14 15:53:00	6274	Wabash Ave & Roosevelt Rd	19.0	Lake Shore Dr & Monroe St	11.0	87.1	8.1	partlycloudy
77	Female	2013-07-21 11:35:00	2013-07-21 13:54:00	8299	State St & 19th St	15.0	Sheffield Ave & Kingsbury St	15.0	82.9	5.8	mostlycloudy
1954	Female	2013-12-28 11:37:00	2013-12-28 13:34:00	7050	LaSalle St & Washington St	15.0	Theater on the Lake	15.0	44.1	12.7	clear

# только одно из условий является истинным

```
bikes.query('tripduration > 5000 or gender=="Female").head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
9	Female	2013-07-04 15:00:00	2013-07-04 15:16:00	922	Lakeview Ave & Fullerton Pkwy	19.0	Racine Ave & Congress Pkwy	19.0	81.0	12.7	mostlycloudy
14	Female	2013-07-06 12:39:00	2013-07-06 12:49:00	610	Morgan St & Lake St	15.0	Aberdeen St & Jackson Blvd	15.0	82.0	5.8	mostlycloudy
18	Male	2013-07-09 13:12:00	2013-07-09 14:42:00	5396	Canal St & Jackson Blvd	35.0	Millennium Park	35.0	79.0	13.8	cloudy

Мы можем проверить, равно ли каждое значение в интересующем столбце одному или нескольким другим заданным значениям, используя слово `in` в своем запросе. Используем синтаксис для создания списка в строке запроса, чтобы он содержал все значения, которые нужно проверить. Сейчас мы отберем поездки, совершенные, когда шел снег или дождь.

# отбор с помощью слова `in`

```
bikes.query('events in ["snow", "rain"]').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
45	Male	2013-07-15 16:43:00	2013-07-15 16:55:00	727	Greenwood Ave & 47th St	15.0	State St & Harrison St	19.0	82.9	5.8	rain
112	Male	2013-07-26 19:10:00	2013-07-26 19:33:00	1395	Larrabee St & Kingsbury St	27.0	Damen Ave & Pierce Ave	19.0	66.9	12.7	rain
124	Male	2013-07-30 18:53:00	2013-07-30 19:00:00	442	Canal St & Jackson Blvd	35.0	Racine Ave & Congress Pkwy	19.0	69.1	3.5	rain

Мы можем инвертировать результат, поставив `not` перед `in`. Сейчас отберем поездки, которые были совершены в дни, когда не было облачно, частично облачно или преимущественно облачно.

# отбор с помощью слова `not in`

```
bikes.query('events not in ["cloudy", "partlycloudy", "mostlycloudy"]').head(3)
```

	gender	starttime	stoptime	tripduration	from_station_name	start_capacity	to_station_name	end_capacity	temperature	wind_speed	events
25	Female	2013-07-11 08:17:00	2013-07-11 08:31:00	830	Wabash Ave & Roosevelt Rd	19.0	Daley Center Plaza	47.0	73.9	8.1	clear
26	Male	2013-07-12 01:07:00	2013-07-12 01:24:00	1043	State St & Harrison St	19.0	Racine Ave & 18th St	15.0	64.9	0.0	clear
33	Male	2013-07-12 17:22:00	2013-07-12 17:34:00	730	Clark St & Congress Pkwy	27.0	Racine Ave & Congress Pkwy	19.0	79.0	10.4	clear

К сожалению, метод `.query()` не дает нам возможности выбрать подмножество столбцов при фильтрации данных. Вам нужно будет сделать обычный отбор столбца после вызова метода. Здесь мы используем только скобки, чтобы выбрать два столбца, после того как найдем все поездки, когда шел снег или дождь.

```
# отберем три столбца для поездок, совершенные,
# когда шел снег или дождь
cols = ['starttime', 'temperature', 'events']
bikes.query('events in ["snow", "rain"]')[cols].head(3)
```

	starttime	temperature	events
45	2013-07-15 16:43:00	82.9	rain
112	2013-07-26 19:10:00	66.9	rain
124	2013-07-30 18:53:00	69.1	rain

## 6.15. АГРЕГИРОВАНИЕ ДАННЫХ

### 6.15.1. Группировка и агрегирование с помощью одного столбца

Взгляните на рисунок ниже. Мы сгруппируем наши исходные данные в независимые датафреймы на основе уникальных значений одного или нескольких столбцов. Наши группы основаны на значении столбца *Dept*. Как только данные будут разбиты на эти независимые датафреймы, для каждого мы выполним агрегирование. Значения столбца *Salary* агрегируются с помощью функции *sum*, а столбец *Experience* агрегируется с помощью функции *mean*.

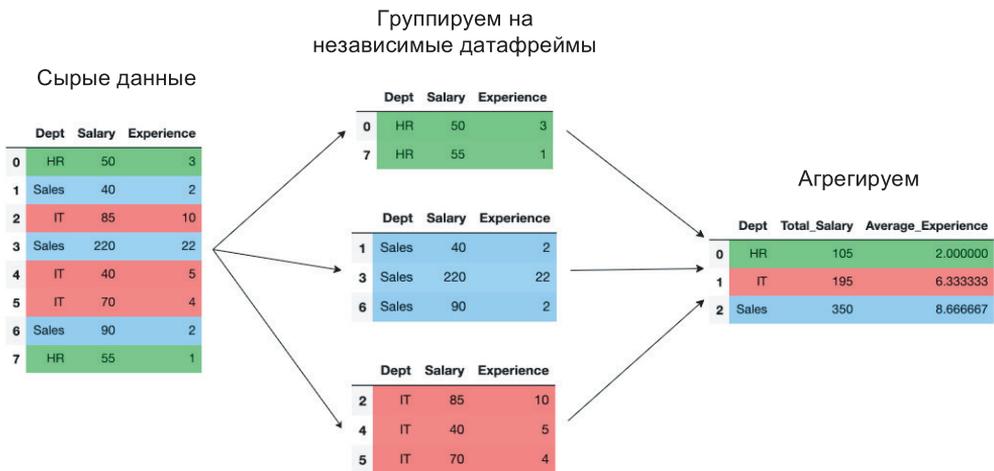


Рис. 16 Схема группировки данных в pandas

Группировка данных – чрезвычайно распространенный метод, используемый в анализе данных, который может помочь нам ответить на множество вопросов. Вот несколько примеров:

- Какова максимальная заработная плата для каждого отдела в компании?
- Какова средняя температура и количество осадков за каждый месяц в разных городах?
- Какие рубашки входят в пятерку самых продаваемых в каждом магазине?

Большинство задач, связанных с группировкой данных в `pandas`, решаются с помощью метода `.groupby()`. Этот единственный метод отвечает за группировку данных на независимые датафреймы и выполнение агрегирования. Обычно это делается в одной строчке кода.

Наиболее распространенным типом действия, выполняемым над каждой группой, является агрегирование, хотя можно манипулировать данными в каждой группе любым удобным для вас способом.

Процедура агрегирования с помощью метода `.groupby()` имеет три отдельных компонента: группирующий столбец, агрегируемый столбец и функцию агрегирования.

- **Группирующий столбец.** Каждое отдельное значение в этом столбце образует отдельную группу.
- **Агрегируемый столбец.** Столбец, к которому мы применяем агрегирующую функцию. Этот столбец обычно является количественным.
- **Агрегирующая функция.** Функция, применяемая к агрегируемому столбцу.

Как правило, мы будем использовать цепочку, состоящую из метода `.groupby()` и метода, который, собственно, возвращает нужный результат. Поэтому процедура агрегирования будет включать два этапа. Во-первых, с помощью метода `.groupby()` мы сообщаем `pandas`, как мы хотели бы группироваться, а затем добавляем метод `.agg()`, чтобы сообщить `pandas`, как агрегировать. Общий синтаксис имеет следующий вид:

```
df.groupby('группирующий столбец').agg(
    new_column=('агрегируемый столбец',
               'агрегирующая функция'))
```

Группирующий столбец, агрегируемый столбец и агрегирующая функция агрегирования могут быть предоставлены в виде строковых значений. Имя нового столбца `new_column` мы определяем в методе `.agg()` и задаем его равным кортежу из двух элементов – агрегируемого столбца и агрегирующей функции.

Выполним нашу первую процедуру агрегации – вычислим среднюю длительность поездки в зависимости от погоды во время поездки.

```
# вычислим среднюю длительность поездки
# в зависимости от погоды во время поездки
bikes.groupby('events').agg(
    avg_tripduration=('tripduration', 'mean'))
```

avg_tripduration	
events	
clear	767.718240
cloudy	690.291346
fog	570.557377
hazy	691.301724
mostlycloudy	736.609963
partlycloudy	725.389928
rain	633.748906
sleet	541.250000
snow	592.860515
tstorms	636.160377
unknown	635.750000

При выполнении агрегирования с помощью метода `.groupby()` важно идентифицировать каждый компонент агрегирования. Для вышеприведенного примера мы имеем:

- группирующий столбец – `events`;
- агрегируемый столбец – `tripduration`;
- агрегирующая функция – функция `mean`.

Ниже приведены допустимые строковые имена агрегирующих функций:

- `sum`;
- `min`;
- `max`;
- `mean`;
- `median`;
- `std`;
- `var`;
- `count` – количество непропущенных значений;
- `size` – количество всех элементов;
- `first` – первое значение в группе;
- `last` – последнее значение в группе;
- `idxmax` – индекс максимального значения в группе;
- `idxmin` – индекс минимального значения в группе.

## 6.15.2. Группировка и агрегирование с помощью нескольких столбцов

Мы можем выполнить группировку на основе нескольких столбцов, передав в метод `.groupby()` список группирующих столбцов. Давайте вычислим среднюю длительность поездки в зависимости от комбинации пола и типа погодного явления во время поездки.

```
# вычислим среднюю длительность поездки
# в зависимости от комбинации пола
# и погоды во время поездки
bikes.groupby(['gender', 'events']).agg(
    avg_tripduration=('tripduration', 'mean'))
```

		avg_tripduration
gender	events	
Female	clear	889.229955
	cloudy	764.428671
	fog	698.933333
	hazy	797.823529
	mostlycloudy	819.058638
	partlycloudy	839.990322
	rain	646.034568
	sleet	781.333333
	snow	613.750000
	tstorms	709.925000
unknown	240.000000	
Male	clear	730.481688
	cloudy	667.281823
	fog	528.695652
	hazy	656.874525
	mostlycloudy	708.844771
	partlycloudy	684.834807
	rain	630.252284
	sleet	485.846154
	snow	589.534826
	tstorms	611.365546
unknown	767.666667	

Столбцы *gender* и *events* больше не являются столбцами и были помещены в индекс. Речь идет о многоуровневом индексе (*MultiIndex*), теперь *gender* и *events* считаются уровнями индекса.

На практике многоуровневый индекс не всегда добавляет особой ценности и может мешать процессу обучения. Обычно переходят к обычному индексу. Часто это обусловлено тем, что агрегирование – промежуточная процедура, вы получили агрегированные данные (например, создавали признаки – групповые средние) и затем хотите присоединить их к исходным данным, многоуровневый индекс здесь вам только мешает. Поэтому с помощью метода `.reset_index()` можно вернуть группирующие столбцы из уровней в столбцы.

```
# вычислим среднюю длительность поездки в зависимости
# от комбинации пола и погоды во время поездки,
# избавимся от многоуровневого индекса
bikes.groupby(['gender', 'events']).agg(
    avg_tripduration=('tripduration', 'mean')).reset_index()
```

	gender	events	avg_tripduration
0	Female	clear	889.229955
1	Female	cloudy	764.428671
2	Female	fog	698.933333
3	Female	hazy	797.823529
4	Female	mostlycloudy	819.058638
5	Female	partlycloudy	839.990322
6	Female	rain	646.034568
7	Female	sleet	781.333333
8	Female	snow	613.750000
9	Female	tstorms	709.925000
10	Female	unknown	240.000000
11	Male	clear	730.481688
12	Male	cloudy	667.281823
13	Male	fog	528.695652
14	Male	hazy	656.874525
15	Male	mostlycloudy	708.844771
16	Male	partlycloudy	684.834807
17	Male	rain	630.252284
18	Male	sleet	485.846154
19	Male	snow	589.534826
20	Male	tstorms	611.365546
21	Male	unknown	767.666667

Чтобы агрегировать несколько столбцов, мы передаем в метод `.agg()` для каждого столбца новое имя и приравниваем его к кортежу из двух элементов, содержащему агрегируемый столбец и агрегирующую функцию. Сейчас мы вычислим среднюю продолжительность поездки и среднюю температуру для каждого типа погодного явления.

```
# мы вычислим среднюю продолжительность поездки
# и среднюю температуру для каждого типа
# погодного явления, избавимся от
# многоуровневого индекса
bikes.groupby('events').agg(
    avg_tripduration=('tripduration', 'mean'),
    avg_temp=('temperature', 'mean')).reset_index()
```

	events	avg_tripduration	avg_temp
0	clear	767.718240	59.531476
1	cloudy	690.291346	56.621143
2	fog	570.557377	50.235246
3	hazy	691.301724	55.594253
4	mostlycloudy	736.609963	67.278551
5	partlycloudy	725.389928	65.444558
6	rain	633.748906	57.066247
7	sleet	541.250000	31.243750
8	snow	592.860515	26.654506
9	tstorms	636.160377	74.200943
10	unknown	635.750000	-2462.250000

При желании мы можем задать несколько группирующих столбцов, несколько агрегируемых столбцов, а также несколько агрегирующих функций. Мы немного упростим задачу: вычислим среднюю длительность поездки, максимальную длительность поездки, среднюю температуру в зависимости от типа погоды во время поездки, при этом избавимся от многоуровневого индекса.

```
# вычислим среднюю длительность поездки,
# максимальную длительность поездки, среднюю температуру
# в зависимости от типа погоды во время поездки,
# избавимся от многоуровневого индекса
bikes.groupby('events').agg(
    avg_tripduration=('tripduration', 'mean'),
    max_tripduration=('tripduration', 'max'),
    avg_temp=('temperature', 'mean')).reset_index()
```

	events	avg_tripduration	max_tripduration	avg_temp
0	clear	767.718240	73591	59.531476
1	cloudy	690.291346	86188	56.621143
2	fog	570.557377	1776	50.235246
3	hazy	691.301724	7739	55.594253
4	mostlycloudy	736.609963	63155	67.278551
5	partlycloudy	725.389928	85442	65.444558
6	rain	633.748906	28994	57.066247
7	sleet	541.250000	1257	31.243750
8	snow	592.860515	8309	26.654506
9	tstorms	636.160377	2868	74.200943
10	unknown	635.750000	1325	-2462.250000

### 6.15.3. Группировка с помощью сводных таблиц

С помощью метода `.pivot_table()` мы можем построить сводную таблицу. Чтобы воспользоваться методом `.pivot_table()`, нам нужно задать следующие параметры:

- `index` – столбец вертикальной группировки (столбцы, которые будут размещены по вертикали);
- `columns` – столбец горизонтальной группировки (столбцы, которые будут размещены по горизонтали);
- `values` – агрегируемый столбец;
- `aggfunc` – агрегирующая функция (по умолчанию используется среднее).

Загрузим данные американской автостраховой компании StateFarm (добавлены категориальные признаки). Они представляют собой записи о 8293 клиентах, классифицированных на два класса: 0 – отклика нет на предложение автостраховки (7462 клиента) и 1 – отклик есть на предложение автостраховки (831 клиент). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- количественный признак *Пожизненная ценность клиента* [*Customer Lifetime Value*];
- категориальный признак *Вид страхового покрытия* [*Coverage*];
- категориальный признак *Образование* [*Education*];
- категориальный признак *Тип занятости* [*EmploymentStatus*];
- категориальный признак *Пол* [*Gender*];
- количественный признак *Доход клиента* [*Income*];
- количественный признак *Размер ежемесячной автостраховки* [*Monthly Premium Auto*];
- количественный признак *Количество месяцев со дня подачи последнего страхового требования* [*Months Since Last Claim*];
- количественный признак *Количество месяцев с момента заключения страхового договора* [*Months Since Policy Inception*];
- количественный признак *Количество открытых страховых обращений* [*Number of Open Complaints*];
- количественный признак *Количество полисов* [*Number of Policies*];
- бинарная зависимая переменная *Отклик на предложение автостраховки* [*Response*].

*# загружаем данные*

```
ins = pd.read_csv('Data/StateFarm_missing.csv', sep=';')
ins.head()
```

	Customer Lifetime Value	Coverage	Education	EmploymentStatus	Gender	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN	32.0	5.0	NaN	1.0	No
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN	13.0	42.0	NaN	NaN	No
2	NaN	NaN	NaN	Employed	F	48767.0	108.0	NaN	38.0	0.0	NaN	No
3	7645.861827	Basic	Bachelor	NaN	NaN	0.0	106.0	18.0	NaN	NaN	7.0	No
4	2813.692575	Basic	Bachelor	NaN	M	43836.0	73.0	12.0	NaN	NaN	1.0	No

С помощью `pivot_table()` посмотрим, как варьирует средний доход клиента по комбинациям пола и образования.

*# смотрим, как варьирует средний доход клиента*

*# по комбинациям пола и образования*

```
ins.pivot_table(index='Education',
```

```
columns='Gender',
values='Income',
aggfunc='mean')
```

	Gender	F	M
<b>Education</b>			
<b>Bachelor</b>		37972.366023	37040.250000
<b>College</b>		37740.598055	36781.846543
<b>Doctor</b>		45731.160256	39342.279503
<b>High School or Below</b>		36211.804149	35891.237416
<b>Master</b>		44329.254190	45259.220000

Теперь избавимся от шума, с помощью метода `.round()` округлим значения дохода до ближайшей тысячи, а с помощью метода `.astype()` превратим их в целые числа.

```
# смотрим, как варьирует средний доход клиента
# по комбинациям пола и образования,
# округлим и превратим в целые числа
ins.pivot_table(
    index='Education',
    columns='Gender',
    values='Income',
    aggfunc='mean').round(-3).astype('int')
```

	Gender	F	M
<b>Education</b>			
<b>Bachelor</b>		38000	37000
<b>College</b>		38000	37000
<b>Doctor</b>		46000	39000
<b>High School or Below</b>		36000	36000
<b>Master</b>		44000	45000

Те же самые результаты мы могли бы получить, используя агрегацию с методом `.groupby()`.

```
# все то же самое можно получить, используя
# агрегацию с groupby
ins.groupby(['Gender', 'Education']).agg(
    mean_salary=('Income', 'mean')) \
    .round(-3).astype('int64')
```

		mean_salary
Gender	Education	
<b>F</b>	<b>Bachelor</b>	38000
	<b>College</b>	38000
	<b>Doctor</b>	46000
	<b>High School or Below</b>	36000
	<b>Master</b>	44000
<b>M</b>	<b>Bachelor</b>	37000
	<b>College</b>	37000
	<b>Doctor</b>	39000
	<b>High School or Below</b>	36000
	<b>Master</b>	45000

Заметим, что расположение всех данных по вертикали немного затрудняет сравнение.

Сводные таблицы создают широкие данные с новыми столбцами для каждого уникального значения одного из группирующих столбцов. Широкие данные обычно легче читать и принимать решения. Метод `.groupby()` возвращает длинные данные с результатами по каждой группе в одном столбце, что затрудняет сравнение.

По умолчанию для параметра `aggfunc` используются значение `'mean'`. Для сводных таблиц доступны все те же строковые названия агрегирующих функций, что и для метода `.groupby()`. Давайте найдем максимальные значения дохода по комбинациям пола и образования.

```
# смотрим максимальный доход клиента
# по комбинациям пола и образования
ins.pivot_table(index='Education',
                 columns='Gender',
                 values='Income',
                 aggfunc='max')
```

	Gender	F	M
<b>Education</b>			
<b>Bachelor</b>		99803.0	99981.0
<b>College</b>		99961.0	99816.0
<b>Doctor</b>		98912.0	99443.0
<b>High School or Below</b>		99841.0	99874.0
<b>Master</b>		99875.0	99960.0

Программа Microsoft Excel хорошо известна своими сводными таблицами, которые создаются путем перетаскивания разных столбцов в разные поля, тем самым мы можем «поворачивать» данные. В pandas вам придется изменить значения параметров и снова вызвать метод `.pivot_table()`, чтобы получить тот же эффект. Давайте повернем таблицу, разместив пол по индексу, а образование – по столбцам.

```
# смотрим максимальный доход клиента
# по комбинациям пола и образования,
# поменяли Education и Gender местами
ins.pivot_table(index='Gender',
                 columns='Education',
                 values='Income',
                 aggfunc='max')
```

	Education Bachelor	College	Doctor	High School or Below	Master
Gender					
F	99803.0	99961.0	98912.0	99841.0	99875.0
M	99981.0	99816.0	99443.0	99874.0	99960.0

Вы можете задать стиль датафрейма, изменив цвет текста, цвет фона, шрифт и некоторые другие элементы с помощью свойства `style`.

Давайте посмотрим среднюю пожизненную ценность клиента по комбинациям типа занятости и образования, результаты переведем в целые числа.

```
# посмотрим среднюю пожизненную ценность клиента
# по комбинациям типа занятости и образования,
# результаты переводим в целые числа
emp_edu_mean_clv = ins.pivot_table(
    index='EmploymentStatus',
    columns='Education',
    values='Customer Lifetime Value',
    aggfunc='mean').astype('int64')
emp_edu_mean_clv
```

	Education Bachelor	College	Doctor	High School or Below	Master
EmploymentStatus					
Disabled	6729	6756	7272	9983	7964
Employed	8224	8047	7538	8566	8042
Medical Leave	7708	6952	11690	8235	7834
Retired	7846	7051	4518	5860	12442
Unemployed	7103	7681	6966	7739	8807

Метод `.highlight_max()` выделяет максимальное значение в каждом столбце или строке. По умолчанию он выделяет максимальное значение каждого столбца.

```
# подсветим максимальные значения в каждом столбце
emp_edu_mean_clv.style.highlight_max()
```

	Education Bachelor	College	Doctor	High School or Below	Master
EmploymentStatus					
Disabled	6729	6756	7272	9983	7964
Employed	8224	8047	7538	8566	8042
Medical Leave	7708	6952	11690	8235	7834
Retired	7846	7051	4518	5860	12442
Unemployed	7103	7681	6966	7739	8807

Мы можем выделить максимальное значение в каждой строке, установив для параметра `axis` значение `'columns'` или `1`. А если для параметра `axis` задать значение `None`, будет подсвечена ячейка с максимальным значением во всей таблице.

```
# подсветим максимальные значения в каждой строке
emp_edu_mean_clv.style.highlight_max(axis='columns')
```

	Education	Bachelor	College	Doctor	High School or Below	Master
EmploymentStatus						
Disabled	6729	6756	7272	9983	7964	
Employed	8224	8047	7538	8566	8042	
Medical Leave	7708	6952	11690	8235	7834	
Retired	7846	7051	4518	5860	12442	
Unemployed	7103	7681	6966	7739	8807	

С помощью метода `.background_gradient()` задаем интенсивность фона ячейки в зависимости от значения в ячейке.

```
# задаем интенсивность фона ячейки
# в зависимости от значения в ячейке
emp_edu_mean_clv.style.background_gradient(cmap='Oranges')
```

	Education	Bachelor	College	Doctor	High School or Below	Master
EmploymentStatus						
Disabled	6729	6756	7272	9983	7964	
Employed	8224	8047	7538	8566	8042	
Medical Leave	7708	6952	11690	8235	7834	
Retired	7846	7051	4518	5860	12442	
Unemployed	7103	7681	6966	7739	8807	

Мы можем подсветить минимальное и максимальное значения каждого столбца, используя разные цвета.

```
# подсветим минимальные и максимальные
# значения в каждом столбце
emp_edu_mean_clv.style.highlight_max(color='yellow') \
    .highlight_min(color='lightblue')
```

	Education	Bachelor	College	Doctor	High School or Below	Master
EmploymentStatus						
Disabled	6729	6756	7272	9983	7964	
Employed	8224	8047	7538	8566	8042	
Medical Leave	7708	6952	11690	8235	7834	
Retired	7846	7051	4518	5860	12442	
Unemployed	7103	7681	6966	7739	8807	

Можно использовать метод `.pivot_table()`, чтобы получить информацию о размере каждой комбинации группирующих столбцов. При этом нет необходимости использовать агрегируемый столбец (параметр `values`). Библиотека `pandas` знает, что размер группы не зависит от того, что она агрегирует, поэтому вам не нужно задавать агрегируемый столбец. Сейчас мы вычислим размер каждой уникальной комбинации типа занятости и образования.

```
# вычислим размер каждой уникальной комбинации
# типа занятости и образования
ins.pivot_table(index='EmploymentStatus',
                columns='Education',
                aggfunc='size')
```

	Education Bachelor	College	Doctor	High School or Below	Master
<b>EmploymentStatus</b>					
<b>Disabled</b>	114	83	21	107	37
<b>Employed</b>	1553	1499	230	1412	492
<b>Medical Leave</b>	113	133	15	106	25
<b>Retired</b>	79	91	1	62	19
<b>Unemployed</b>	635	614	50	710	86

Для параметра `margins` можно установить значение `True`, чтобы добавить одну дополнительную строку и столбец в сводную таблицу, здесь с помощью агрегирующей функции мы вычисляем одну и ту же статистику для всей строки или столбца. В сводной таблице ниже мы видим, что средний доход всех бакалавров (`Bachelor`) составляет 37 521.5, а средний доход всех пенсионеров (`Retired`) – 20 489.5. Кроме того, мы сравним средние доходы для каждой комбинации со средним доходом в целом по выборке.

```
# смотрим, как варьирует средний доход клиента
# по комбинациям типа занятости и образования,
# добавляем среднюю зарплату для
# всей строки и всего столбца
ins.pivot_table(index='EmploymentStatus',
                columns='Education',
                values='Income',
                aggfunc='mean',
                margins=True)
```

	Education Bachelor	College	Doctor	High School or Below	Master	All
<b>EmploymentStatus</b>						
<b>Disabled</b>	19981.508772	20604.397590	20397.904762	19145.635514	21065.675676	20012.226519
<b>Employed</b>	56272.710238	55899.775851	55413.704348	57315.307365	56674.939024	56448.846703
<b>Medical Leave</b>	19720.584071	20586.947368	18364.933333	21003.962264	19227.400000	20278.237245
<b>Retired</b>	21276.468354	20543.802198	19186.000000	19570.354839	20025.263158	20489.503968
<b>Unemployed</b>	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
<b>All</b>	37521.522855	37236.148347	42486.334385	36052.390071	44802.223065	37782.136962

```
# вычислим средний доход
print(ins['Income'].mean())
```

```
37785.17199372814
```

В сводной таблице может быть и один группирующий столбец. Сейчас мы найдем средний доход для каждого типа занятости.

```
# задаем один группирующий столбец
ins.pivot_table(index='EmploymentStatus',
                 values='Income',
                 aggfunc='mean').round(-3)
```

Income	
EmploymentStatus	
Disabled	20000.0
Employed	56000.0
Medical Leave	20000.0
Retired	20000.0
Unemployed	0.0

При использовании одного группирующего столбца результат будет точно таким же, как и при агрегации с помощью метода `.groupby()`. Преимущество использования агрегации с помощью метода `.groupby()` заключается в возможности переименовать результирующий столбец.

```
# тот же результат с .groupby()
ins.groupby('EmploymentStatus').agg(
    average_income=('Income', 'mean')).round(-3)
```

average_income	
EmploymentStatus	
Disabled	20000.0
Employed	56000.0
Medical Leave	20000.0
Retired	20000.0
Unemployed	0.0

Мы можем повернуть эту таблицу с одним столбцом, используя параметр `columns` и отказавшись от параметра `index`.

```
# поворачиваем таблицу с одним столбцом
ins.pivot_table(columns='EmploymentStatus',
                 values='Income',
                 aggfunc='mean').round(-3)
```

EmploymentStatus	Disabled	Employed	Medical Leave	Retired	Unemployed
Income	20000.0	56000.0	20000.0	20000.0	0.0



При создании сводной таблицы лучше сохранять многоуровневый индекс. Несколько уровней при использовании сводной таблицы (если вам случится их создавать). Это противоположно совету, который мы рекомендовали при агрегировании с помощью метода `.groupby()`. Причина заключается в том, что, скорее всего, сводные таблицы будут конечным продуктом, который вы будете использовать в презентации или отчете, и этот продукт не предполагает дальнейших манипуляций (например, объединения с исходными данными).

Мы можем задать несколько агрегируемых столбцов. Мы вычислим среднюю пожизненную ценность клиента и средний доход по комбинациям образования и пола, дополнительно добавим итоговые значения по строкам и столбцам.

```
# зададим два агрегируемых столбца
# (пожизненная ценность клиента и доход)
ins.pivot_table(
    index='Education',
    columns='Gender',
    values=['Income', 'Customer Lifetime Value'],
    aggfunc='mean', margins=True)
```

Gender	Customer Lifetime Value			Income		
	F	M	All	F	M	All
<b>Education</b>						
<b>Bachelor</b>	7980.275871	7675.774393	7833.763532	37972.366023	37040.250000	37539.099439
<b>College</b>	7757.306941	7868.564624	7811.832401	37740.598055	36781.846543	37270.730579
<b>Doctor</b>	7530.432422	7702.351623	7617.747852	45731.160256	39342.279503	42486.334385
<b>High School or Below</b>	8504.800424	8093.913831	8300.471338	36211.804149	35891.237416	36052.390071
<b>Master</b>	8457.905556	8015.928886	8256.396436	44329.254190	45259.220000	44753.250760
<b>All</b>	8088.051778	7882.111780	7987.666592	38235.288439	37327.506313	37792.791214

Мы можем задать несколько агрегирующих функций. Вычислим минимальный, средний и максимальный доходы по комбинациям образования и пола, дополнительно добавим итоговые значения по строкам и столбцам.

```
# зададим несколько агрегирующих функций
ins.pivot_table(
    index='Education',
    columns='Gender',
    values='Income',
    aggfunc=['min', 'mean', 'max'], margins=True)
```

Gender	min			mean			max		
	F	M	All	F	M	All	F	M	All
Education									
Bachelor	0.0	0.0	0.0	37972.366023	37040.250000	37524.053707	99803.0	99981.0	99981.0
College	0.0	0.0	0.0	37740.598055	36781.846543	37270.730579	99961.0	99816.0	99961.0
Doctor	0.0	0.0	0.0	45731.160256	39342.279503	42486.334385	98912.0	99443.0	99443.0
High School or Below	0.0	0.0	0.0	36211.804149	35891.237416	36052.390071	99841.0	99874.0	99874.0
Master	0.0	0.0	0.0	44329.254190	45259.220000	44753.250760	99875.0	99960.0	99960.0
All	0.0	0.0	0.0	38226.287665	37327.506313	37788.230723	99961.0	99981.0	99981.0

## 6.16. АНАЛИЗ ЧАСТОТ С ПОМОЩЬЮ ТАБЛИЦ СОПРЯЖЕННОСТИ

С помощью метода `.value_counts()` мы можем вывести абсолютные частоты категорий переменной. По умолчанию если есть пропуски, то они опускаются, вывести их можно, задав `dropna=False`.

Давайте выведем абсолютные частоты категорий переменной `Coverage`, посмотрим встречаемость различных видов страхового покрытия.

```
# выведем абсолютные частоты категорий
ins['Coverage'].value_counts(dropna=False)
```

```
Basic      5038
Extended   2501
Premium    749
NaN         5
Name: Coverage, dtype: int64
```

Задав для параметра `normalize` значение `True`, мы выведем уже относительные частоты (абсолютные частоты, поделенные на общее количество наблюдений).

```
# выведем относительные частоты категорий
ins['Coverage'].value_counts(dropna=False,
                              normalize=True)
```

```
Basic      0.607500
Extended   0.301580
Premium    0.090317
NaN        0.000603
Name: Coverage, dtype: float64
```

Абсолютные частоты категорий можно также получить с помощью цепочки методов `.groupby()` и `.size()`. Однако информация по пропускам выведена не будет.

```
# выведем абсолютные частоты категорий
# с помощью .groupby() и .size()
ins.groupby('Coverage').size().reset_index(name='count')
```

	Coverage	count
0	Basic	5038
1	Extended	2501
2	Premium	749

С помощью цепочки методов `.groupby()` и `.size()` можно посмотреть абсолютные частоты комбинаций вида страхового покрытия и пола.

```
# выведем абсолютные частоты категорий
# с помощью .groupby() и .size()
ins.groupby(['Coverage', 'Gender']).size().reset_index(name='count')
```

	Coverage	Gender	count
0	Basic	F	2547
1	Basic	M	2489
2	Extended	F	1308
3	Extended	M	1192
4	Premium	F	392
5	Premium	M	356

Абсолютные частоты можно вывести и с помощью метода `.pivot_table()`.

```
# выведем абсолютные частоты категорий
# с помощью .pivot_table()
ins.pivot_table(index='Coverage',
                 columns='Gender',
                 aggfunc='size')
```

Gender	F	M
Coverage		
Basic	2547	2489
Extended	1308	1192
Premium	392	356

Функция `pd.crosstab()` создана специально для подсчета совместной встречаемости значений двух и более столбцов. Название происходит от слова **cross tabulation** («кросстабуляция»). Кроме того, часто используется термин **contingency tables** («таблицы сопряженности»).

К сожалению, `crosstab` – это функция, а не метод. Это означает, что она не привязана к какому-либо датафрейму, но доступ к ней должен осуществляться непосредственно из `pd`. У нее много таких же параметров, что и у метода `.pivot_table()`, и используется она аналогично. Поскольку она не привязана ни к какому объекту `DataFrame`, в качестве значений параметров мы указываем серии вместо строковых значений. По умолчанию функция вычисляет размер каждой группы, поэтому нет необходимости устанавливать параметр `aggfunc`.

```
# строим таблицу сопряженности вида
# страхового покрытия и пола
```

```
pd.crosstab(index=ins['Coverage'],
            columns=ins['Gender'])
```

Gender	F	M
<b>Coverage</b>		
Basic	2547	2489
Extended	1308	1192
Premium	392	356

Результат идентичен результату, полученному с помощью метода `.pivot_table()`. Однако у функции `pd.crosstab()` есть большое преимущество, а именно ее способность возвращать относительные частоты с помощью параметра нормализации. Это нелегко сделать с помощью `.groupby()` или `.pivot_table()`. Функция `pd.crosstab()` позволяет нормализовать строки, столбцы и итоговые значения. Например, нас интересует относительная частота мужчин в каждом виде страхового покрытия. Нам нужно нормализовать каждую строку, для этого для параметра `normalize` задаем значение `'index'`. Все строки должны в сумме давать 100 %.

```
# вычислим относительные частоты мужчин и женщин
# в каждом виде страхового покрытия,
# нормализация по строкам
pd.crosstab(index=ins['Coverage'],
            columns=ins['Gender'],
            normalize='index').round(3) * 100
```

Gender	F	M
<b>Coverage</b>		
Basic	50.6	49.4
Extended	52.3	47.7
Premium	52.4	47.6

А теперь нас интересуют относительные частоты видов страхового покрытия для мужчин. Нам нужно нормализовать каждый столбец, для этого для параметра `normalize` задаем значение `'columns'`. Все столбцы должны в сумме давать 100 %.

```
# вычислим относительные частоты видов
# страхового покрытия для мужчин и женщин,
# нормализация по столбцам
pd.crosstab(index=ins['Coverage'],
            columns=ins['Gender'],
            normalize='columns').round(3) * 100
```

Gender	F	M
<b>Coverage</b>		
Basic	60.0	61.7
Extended	30.8	29.5
Premium	9.2	8.8

Можно найти относительные частоты с учетом всех данных, установив для параметра `normalize` значение `'all'`.

```
# вычислим относительные частоты
# с учетом всех данных
pd.crosstab(index=ins['Coverage'],
            columns=ins['Gender'],
            normalize='all').round(3) * 100
```

Gender	F	M
<b>Coverage</b>		
<b>Basic</b>	30.7	30.0
<b>Extended</b>	15.8	14.4
<b>Premium</b>	4.7	4.3

Согласно возвращенному датафрейму, 4,1% всех клиентов – это мужчины, оформившие вид страхового покрытия `Premium`.

Теперь вернемся к абсолютным частотам и зададим итоги по строкам и столбцам, установив для параметра `margins` значение `True`.

```
# вернемся к абсолютным частотам, зададим
# итоги по строкам и столбцам
pd.crosstab(index=ins['Coverage'],
            columns=ins['Gender'],
            margins=True)
```

Gender	F	M	All
<b>Coverage</b>			
<b>Basic</b>	2547	2489	5036
<b>Extended</b>	1308	1192	2500
<b>Premium</b>	392	356	748
<b>All</b>	4247	4037	8284

## 6.17. Выполнение SQL-запросов в pandas

Библиотека `pandas` стала удобным инструментом для выполнения SQL-запросов. Давайте загрузим данные и подключимся к базе данных, а затем на конкретных примерах сравним синтаксис SQL-запросов и нативных запросов в `pandas`.

```
# импортируем библиотеки
import pandas as pd
import sqlite3

# загружаем данные
airports = pd.read_csv('Data/airports.csv')

# взглянем на данные
airports.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country	iso_region	municipality	scheduled_service	gps_code
0	6523	00A	heliport	Total Rf Heliport	40.070801	-74.933601	11.0	NaN	US	US-PA	Bensalem	no	00A
1	323361	00AA	small_airport	Aero B Ranch Airport	38.704022	-101.473911	3435.0	NaN	US	US-KS	Leoti	no	00AA
2	6524	00AK	small_airport	Lowell Field	59.949200	-151.695999	450.0	NaN	US	US-AK	Anchor Point	no	00AK
3	6525	00AL	small_airport	Epps Airpark	34.864799	-86.770302	820.0	NaN	US	US-AL	Harvest	no	00AL
4	6526	00AR	closed	Newport Hospital & Clinic Heliport	35.808700	-91.254898	237.0	NaN	US	US-AR	Newport	no	NaN

*# создаем подключение*

```
connection = sqlite3.connect('Data/flights.sqlite')
airports.to_sql('AIRPORTS', connection, if_exists='replace')
```

*# подтверждаем отправку данных*

```
connection.commit()
```

## Отбор столбца

*# отбираем столбец с помощью метода .read\_sql()*

```
query_res = pd.read_sql("SELECT id FROM AIRPORTS;", connection)
query_res.head()
```

	id
0	6523
1	323361
2	6524
3	6525
4	6526

*# отбираем столбец в pandas*

```
pandas_res = airports.loc[:, 'id']
pandas_res.head()
```

```
0    6523
1   323361
2    6524
3    6525
4    6526
```

```
Name: id, dtype: int64
```

## Отбор столбца с условием

*# отбираем столбец с помощью метода .read\_sql()*

```
query_res2 = pd.read_sql("SELECT id FROM AIRPORTS WHERE ident='KLAX';",
                          connection)
```

```
query_res2
```

	id
0	3632

```
# отбираем столбец с условием в pandas
```

```
pandas_res2 = airports[airports['ident'] == 'KLAX']['id']
pandas_res2
```

```
27909 3632
```

```
Name: id, dtype: int64
```

### Отбор столбцов с несколькими условиями

```
# отбираем столбцы с несколькими условиями с помощью метода .read_sql()
```

```
query_res3 = pd.read_sql("SELECT ident, name, municipality FROM AIRPORTS "
                        "WHERE iso_region='US-CA' AND "
                        "type='large_airport';",
                        connection)
```

```
query_res3
```

	<b>ident</b>	<b>name</b>	<b>municipality</b>
0	KBAB	Beale Air Force Base	Marysville
1	KEDW	Edwards Air Force Base	Edwards
2	KLAX	Los Angeles International Airport	Los Angeles
3	KOAK	Metropolitan Oakland International Airport	Oakland
4	KONT	Ontario International Airport	Ontario
5	KSAN	San Diego International Airport	San Diego
6	KSFO	San Francisco International Airport	San Francisco
7	KSJC	Norman Y. Mineta San Jose International Airport	San Jose
8	KSMF	Sacramento International Airport	Sacramento
9	KSNA	John Wayne Airport-Orange County Airport	Santa Ana
10	KSUU	Travis Air Force Base	Fairfield
11	KVBG	Vandenberg Air Force Base	Lompoc

```
# отбираем столбцы с несколькими условиями в pandas
```

```
pandas_res3 = airports[(airports['iso_region'] == 'US-CA') & (
    airports['type'] == 'large_airport')][['ident', 'name', 'municipality']]
pandas_res3
```



	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
index									
37913	2	OM11	small_airport	Abu Dhabi Northeast Airport	24.518900	54.980099	88.0	AS	AE
48826	7	UD21	small_airport	Yerevan Yegvard Airport	40.294102	44.564602	4416.0	AS	AM
20811	9	FN18	small_airport	Matala Airport	-14.727500	15.014000	4120.0	AF	AO
20812	10	FN19	small_airport	Cabo Ledo Airport	-9.653050	13.260600	360.0	AF	AO
37062	11	NZ12	small_airport	Palmer Station Airport	-64.775002	-64.054398	149.0	AN	AQ

*# отбираем все столбцы с условием, упорядочив по id, в pandas*

```
pandas_res4 = airports[airports['type'] == 'small_airport'].sort_values('id')
pandas_res4.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
37913	2	OM11	small_airport	Abu Dhabi Northeast Airport	24.518900	54.980099	88.0	AS	AE
48826	7	UD21	small_airport	Yerevan Yegvard Airport	40.294102	44.564602	4416.0	AS	AM
20811	9	FN18	small_airport	Matala Airport	-14.727500	15.014000	4120.0	AF	AO
20812	10	FN19	small_airport	Cabo Ledo Airport	-9.653050	13.260600	360.0	AF	AO
37062	11	NZ12	small_airport	Palmer Station Airport	-64.775002	-64.054398	149.0	AN	AQ

### Отбор столбцов с условием и обратной сортировкой по выбранной переменной

*# отбираем все столбцы с условием, упорядочив по id*

*# в обратном порядке, с помощью метода .read\_sql()*

```
query_res5 = pd.read_sql("SELECT * FROM AIRPORTS WHERE type='small_airport' "
                        "ORDER BY id DESC;",
                        connection,
                        index_col='index')
```

```
query_res5.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
<b>index</b>									
<b>54990</b>	331866	ZA-0159	small_airport	SKA SA Karoo Landing Strip	-30.693800	21.460800	3427.0	AF	ZA
<b>22598</b>	331857	GB-0632	small_airport	Debach Airfield	52.134134	1.265958	NaN	EU	GB
<b>20189</b>	331856	FAWE	small_airport	Welgevonden Reserve	-24.204600	27.900000	3609.0	AF	ZA
<b>19930</b>	331855	FADC	small_airport	Douglas Cape	-29.074600	23.800900	NaN	AF	ZA
<b>22597</b>	331853	GB-0631	small_airport	Royal Glamorgan Hospital Heliport	51.546332	-3.394245	NaN	EU	GB

```
# отбираем все столбцы с условием, упорядочив по id
```

```
# в обратном порядке, в pandas
```

```
pandas_res5 = airports[airports['type'] == 'small_airport'].sort_values(
    'id', ascending=False)
pandas_res5.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
<b>54990</b>	331866	ZA-0159	small_airport	SKA SA Karoo Landing Strip	-30.693800	21.460800	3427.0	AF	ZA
<b>22598</b>	331857	GB-0632	small_airport	Debach Airfield	52.134134	1.265958	NaN	EU	GB
<b>20189</b>	331856	FAWE	small_airport	Welgevonden Reserve	-24.204600	27.900000	3609.0	AF	ZA
<b>19930</b>	331855	FADC	small_airport	Douglas Cape	-29.074600	23.800900	NaN	AF	ZA
<b>22597</b>	331853	GB-0631	small_airport	Royal Glamorgan Hospital Heliport	51.546332	-3.394245	NaN	EU	GB

### Отбор столбцов со спископодобным условием – учитываем только строковые значения переменной В в списке

```
# отбираем все столбцы со спископодобным условием, учитывая лишь
```

```
# строковые значения переменной type В списке,
```

```
# с помощью метода .read_sql()
```

```
query_res6 = pd.read_sql("SELECT * FROM AIRPORTS WHERE type IN "
    "('heliport', 'balloonport');",
    connection,
    index_col='index')
```

```
query_res6.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
<b>index</b>									
0	6523	00A	heliport	Total Rf Heliport	40.070801	-74.933601	11.0	None	US
9	322658	00CN	heliport	Kitchen Creek Helibase Heliport	32.727374	-116.459742	3350.0	None	US
12	6532	00FD	heliport	Ringhaver Heliport	28.846600	-82.345398	25.0	None	US
15	6535	00GE	heliport	Caffrey Heliport	33.884201	-84.733902	957.0	None	US
16	6536	00HI	heliport	Kaupulehu Heliport	19.832715	-155.980233	43.0	None	US

```
# отбираем все столбцы со спископодобным условием, учитывая
# лишь строковые значения переменной type в списке, в pandas
pandas_res6 = airports[airports['type'].isin(
    ['heliport', 'balloonport'])]
pandas_res6.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
0	6523	00A	heliport	Total Rf Heliport	40.070801	-74.933601	11.0	NaN	US
9	322658	00CN	heliport	Kitchen Creek Helibase Heliport	32.727374	-116.459742	3350.0	NaN	US
12	6532	00FD	heliport	Ringhaver Heliport	28.846600	-82.345398	25.0	NaN	US
15	6535	00GE	heliport	Caffrey Heliport	33.884201	-84.733902	957.0	NaN	US
16	6536	00HI	heliport	Kaupulehu Heliport	19.832715	-155.980233	43.0	NaN	US

### Отбор столбцов со спископодобным условием – учитываем только строковые значения переменной ВНЕ списка

```
# отбираем все столбцы со спископодобным условием,
# учитывая лишь строковые значения переменной type
# ВНЕ списка, с помощью метода .read_sql()
query_res7 = pd.read_sql("SELECT * FROM AIRPORTS WHERE type NOT IN "
    "('heliport', 'balloonport');",
    connection,
    index_col='index')
query_res7.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
<b>index</b>									
1	323361	00AA	small_airport	Aero B Ranch Airport	38.704022	-101.473911	3435.0	None	US
2	6524	00AK	small_airport	Lowell Field	59.949200	-151.695999	450.0	None	US
3	6525	00AL	small_airport	Epps Airpark	34.864799	-86.770302	820.0	None	US
4	6526	00AR	closed	Newport Hospital & Clinic Heliport	35.608700	-91.254898	237.0	None	US
5	322127	00AS	small_airport	Fulton Airport	34.942803	-97.818019	1100.0	None	US

```
# отбираем все столбцы со спископодобным условием, учитывая лишь
# строковые значения переменной type ВНЕ списка, в pandas
pandas_res7 = airports[~airports['type'].isin(
    ['heliport', 'balloonport'])]
pandas_res7.head()
```

	id	ident	type	name	latitude_deg	longitude_deg	elevation_ft	continent	iso_country
1	323361	00AA	small_airport	Aero B Ranch Airport	38.704022	-101.473911	3435.0	NaN	US
2	6524	00AK	small_airport	Lowell Field	59.949200	-151.695999	450.0	NaN	US
3	6525	00AL	small_airport	Epps Airpark	34.864799	-86.770302	820.0	NaN	US
4	6526	00AR	closed	Newport Hospital & Clinic Heliport	35.608700	-91.254898	237.0	NaN	US
5	322127	00AS	small_airport	Fulton Airport	34.942803	-97.818019	1100.0	NaN	US

### Вывод статистик по переменным

```
# выводим статистики по переменной elevation_ft
# с помощью метода .read_sql()
query_res8 = pd.read_sql("SELECT MAX(elevation_ft), MIN(elevation_ft), "
    "AVG(elevation_ft) FROM AIRPORTS;",
    connection)
query_res8
```

	MAX(elevation_ft)	MIN(elevation_ft)	AVG(elevation_ft)
0	22000.0	-1266.0	1245.010533

```
# закрываем подключение
connection.close()
```

```
# выводим статистики по переменной elevation_ft
```

```
pandas_res8 = airports.agg({'elevation_ft': ['min', 'max', 'mean']})
pandas_res8
```

elevation_ft	
min	-1266.000000
max	22000.000000
mean	1245.010533

## Слияние таблиц/датафреймов

```
# импортируем класс date модуля datetime
from datetime import date
```

```
# это наши клиенты
customers = {'CustomerID': [10, 11],
             'Name': ['Mike', 'Marcia'],
             'Address': ['Address for Mike',
                        'Address for Marcia']}
customers = pd.DataFrame(customers)
customers
```

	CustomerID	Name	Address
0	10	Mike	Address for Mike
1	11	Marcia	Address for Marcia

```
# это наши заказы, сделанные клиентами, они связаны
# с клиентами с помощью столбца CustomerID
orders = {'CustomerID': [10, 11, 10],
          'OrderDate': [date(2014, 12, 1),
                       date(2014, 12, 1),
                       date(2014, 12, 1)]}
orders = pd.DataFrame(orders)
orders
```

	CustomerID	OrderDate
0	10	2014-12-01
1	11	2014-12-01
2	10	2014-12-01

```
# создаем подключение
connection = sqlite3.connect('Data/sales.sqlite')
customers.to_sql('CUSTOMERS', connection, if_exists='replace')
orders.to_sql('ORDERS', connection, if_exists='replace')
```

```
# подтверждаем отправку данных
connection.commit()
```

```
# выполняем слияние таблиц CUSTOMERS и ORDERS без
# сохранения идентификатора с помощью метода .read_sql()
query_res9_1 = pd.read_sql("SELECT Name, Address, OrderDate "
                          "FROM CUSTOMERS c "
                          "INNER JOIN ORDERS o "
                          "ON c.CustomerID = o.CustomerID",
                          connection)
query_res9_1
```

	Name	Address	OrderDate
0	Mike	Address for Mike	2014-12-01
1	Mike	Address for Mike	2014-12-01
2	Marcia	Address for Marcia	2014-12-01

```
# выполняем слияние таблиц CUSTOMERS и ORDERS с
# сохранением идентификатора с помощью метода .read_sql()
query_res9_2 = pd.read_sql("SELECT CUSTOMERS.CustomerID, Name, "
                           "Address, OrderDate FROM CUSTOMERS "
                           "INNER JOIN ORDERS "
                           "ON CUSTOMERS.CustomerID=ORDERS.CustomerID",
                           connection)

query_res9_2
```

	CustomerID	Name	Address	OrderDate
0	10	Mike	Address for Mike	2014-12-01
1	10	Mike	Address for Mike	2014-12-01
2	11	Marcia	Address for Marcia	2014-12-01

```
# выполняем слияние датафреймов customers и orders
pandas_res9 = customers.merge(orders)
pandas_res9
```

	CustomerID	Name	Address	OrderDate
0	10	Mike	Address for Mike	2014-12-01
1	10	Mike	Address for Mike	2014-12-01
2	11	Marcia	Address for Marcia	2014-12-01

## Задача с собеседования (теория вероятности)

1. По данным ФБР, около 80 % всех преступлений против собственности остаются нераскрытыми. Предположим, что в вашем городе совершено 3 таких преступления, каждое из которых считается независимым друг от друга. Какова вероятность раскрытия точно одного из трех преступлений? Какова вероятность раскрытия по крайней мере одного преступления?

## Задача с собеседования (математическая статистика)

1. Почему на практике мы, как правило, пользуемся бутстрепированными доверительными интервалами метрик качества (например, бутстрепированным доверительным интервалом AUC-ROC) вместо того чтобы вычислять доверительные интервалы по асимптотическому методу, пользуясь центральной предельной теоремой?

## 7. SCIKIT-LEARN

Библиотека `scikit-learn` (произносится как сайкит-лёрн) – это проект с открытым исходным кодом. Ее можно свободно использовать и распространять.

### 7.1. ОСНОВЫ РАБОТЫ С КЛАССАМИ, СТРОЯЩИМИ МОДЕЛИ ПРЕДВАРИТЕЛЬНОЙ ПОДГОТОВКИ ДАННЫХ И МОДЕЛИ МАШИННОГО ОБУЧЕНИЯ

В библиотеке `scikit-learn` каждая модель предварительной подготовки данных и каждая модель машинного обучения реализована в собственном классе. При этом классы, в которых реализованы модели машинного обучения с учителем, в зависимости от решаемой задачи называются классификаторами (`classifier`) или регрессорами (`regressor`). Классы-классификаторы обычно имеют название `[Метод_машинного_обучения]Classifier`, например `DecisionTreeClassifier`, `RandomForestClassifier`. Классы-регрессоры обычно имеют название `[Метод_машинного_обучения]Regressor`, например `DecisionTreeRegressor`, `RandomForestRegressor`. Если модель машинного обучения с учителем позволяет выполнить только одну задачу – либо задачу регрессии, либо задачу классификации – или речь идет о модели машинного обучения без учителя, то класс носит название метода, например класс `LinearRegression`, потому что линейная регрессия решает только задачу регрессии, или класс `DBSCAN` по названию метода кластеризации `DBSCAN`.



Рис. 17 Классы моделей машинного обучения в `scikit-learn`

При работе с классом – моделью предварительной подготовки – мы выполняем следующие операции:

- импортируем из соответствующего модуля класс, в котором реализована соответствующая модель предварительной подготовки;
- создаем экземпляр класса – объект-модель;
- обучаем модель, т.е. вычисляем параметры, с помощью которых будем выполнять преобразование, – используем метод `.fit()` объекта-модели;
- применяем модель, т.е. выполняем преобразование с помощью найденных параметров, – используем метод `.transform()` объекта-модели;
- либо обучаем и применяем модель сразу – используем метод `.fit_transform()` объекта-модели.

В отличие от классов, в которых реализованы модели машинного обучения, большинство классов, выполняющих предварительную подготовку, будут работать только с массивом признаков, а массив меток не используется.

Самый простой способ реализовать собственный класс, строящий модель предварительной подготовки, – воспользоваться наследованием базовых классов `BaseEstimator` и `TransformerMixin`. Наш класс, как и любой класс предварительной подготовки библиотеки `scikit-learn`, должен иметь методы `__init__`, `.fit()` и `.transform()`. Кроме того, наш класс должен иметь методы `__get_params__` и `__set_params__` (их не нужно специально создавать, они наследуются от базового класса `BaseEstimator`), с помощью этих методов мы можем задавать и получать доступ к параметрам. Например, если вы напишете свой класс, не выполнив наследование класса `BaseEstimator`, и, например, захотите воспользоваться им в конвейере в ходе поиска гиперпараметров, то получите ошибку `'название_вашего_класса' object has no attribute 'set_params'`.

Все атрибуты инициализируются в методе `__init__`. Никаких операций с атрибутами в методе `__init__` не должно быть. Названия параметров должны совпадать с названиями атрибутов.

Несмотря на то что большинство классов, выполняющих предварительную подготовку, будут работать только с массивом признаков, метод `.fit()` должен принимать в качестве аргументов  $x$  и  $y$ . Это требуется для совместимости с конвейерами `scikit-learn`! Обычно для  $y$  задают значение `None`. В методе `.fit()` происходит вычисление параметров модели и всегда возвращается `self`.

Метод `.transform()` должен принимать в качестве аргумента только  $x$ , здесь мы понимаем, что преобразование зависимой переменной нам не требуется. В методе `.transform()` происходит применение вычисленных параметров модели и всегда возвращается  $x$ .

Вспомогательные методы, предназначенные для правильной внутренней работы класса, оформляйте в виде частных и защищенных методов.

Импорт библиотек, классов и модулей внутри методов не допускается.

Базовый класс `TransformerMixin` позволяет нам связать методы `.fit()` и `.transform()` в цепочку (то есть мы можем применить `.fit_transform()`, используя методы нашего класса `.fit()` и `.transform()`). Часто пишут собственный метод `.fit_transform()`.

Мы сейчас напишем класс `MeanImputer`, который будет выполнять замену пропусков средними значениями. Нам потребуется импорт библиотек `pandas`, `numpy` и `math`. Базовые классы `BaseEstimator` и `TransformerMixin` импортировать не будем, поскольку класс не будет использоваться в конвейере и метод `.fit_transform()` мы применять не будем.

```
# импортируем библиотеки pandas, numpy, math
import pandas as pd
import numpy as np
import math
# создаем собственный класс, выполняющий замену
# пропусков средним значением
class MeanImputer():
    """
    Параметры
    -----
    сору: bool, по умолчанию True
           Возвращает копию.

    Возвращает
```

```
-----
X : pandas.DataFrame или numpy.ndarray
    Датафрейм pandas или массив NumPy
    с импутированными значениями.
"""
```

Метод `init`, задающий конструктор класса (инициализация атрибутов)

```
def __init__(self, copy=True):
    # все параметры для инициализации публичных атрибутов
    # должны быть заданы в методе __init__

    # публичный атрибут
    self.copy = copy
```

Добавляем параметр `copy` в определение метода `__init__`

Используем значение `copy` для инициализации публичного атрибута класса, который представлен как `self.copy`

Частный метод `is_numpy`, предназначенный для правильной внутренней работы класса (проверка типа объекта)

```
def __is_numpy(self, X):
    # частный метод, который с помощью функции isinstance()
    # проверяет, является ли наш объект массивом NumPy
    return isinstance(X, np.ndarray)
```

Метод `fit`, задающий обучение модели

```
def fit(self, X, y=None):
    # метод .fit() должен принимать
    # в качестве аргументов X и y

    # создаем пустой словарь, в котором ключами
    # будут имена/целые числа, а значениями - средние
    self._encoder_dict = {}

    # записываем результат метода __is_numpy
    is_np = self.__is_numpy(X)

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # если объект - массив NumPy
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # вычисляем среднее и записываем в словарь
            self._encoder_dict[col] = np.nanmean(X[:, col])
    # если объект - датафрейм pandas
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # вычисляем среднее и записываем в словарь
            self._encoder_dict[col] = X[col].mean()

    # fit возвращает self
    return self
```

Вычисляем единственный параметр модели - среднее значение переменной

Метод transform,  
задающий при-  
менение модели

```
def transform(self, X):
    # transform принимает в качестве
    # аргумента только X

    # выполняем копирование массива во избежание
    # предупреждения SettingWithCopyWarning
    # "A value is trying to be set on a copy of
    # a slice from a DataFrame (Происходит попытка
    # изменить значение в копии среза данных
    # датафрейма)"

    if self.copy:
        X = X.copy()

    # записываем результат метода __is_numpy
    is_np = self.__is_numpy(X)

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # применяем преобразование к X
    # если объект - массив NumPy:
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # заменяем пропуски средним
            # значением из словаря
            X[:, col] = np.nan_to_num(
                X[:, col],
                nan=self._encoder_dict[col])
    # если объект - датафрейм pandas:
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # заменяем пропуски средним
            # значением из словаря
            X[col] = np.where(X[col].isnull(),
                              self._encoder_dict[col],
                              X[col])

    # transform возвращает X
    return X
```

Кратко опишем наш класс `MeanImputer`. В методе `__init__` мы задаем параметр `copy`, выполняющий копирование объекта, используем значение параметра `copy` для инициализации публичного атрибута `self.copy`, задаем частный метод `__is_numpy`, проверяющий, является ли наш объект массивом NumPy, метод `.fit()` вычисляет параметр модели – среднее значение по каждой переменной – и кладет в словарь, метод `.transform()` применяет преобразование – импутацию пропусков с помощью вычисленного параметра модели – среднего значения, которое берет из словаря. Таким образом, для хранения средних значений нам понадобится словарь. Вновь обратите внимание, что метод `.fit()` должен принимать в качестве аргументов `x` и `y`. А вот метод `.transform()` принимает в качестве аргумента только `x`.

Давайте рассмотрим, что происходит в методе `__init__`.

Метод `init`, задающий конструктор класса (инициализация атрибутов)

```
def __init__(self, copy=True):
    # все параметры для инициализации публичных атрибутов
    # должны быть заданы в методе __init__
```

Добавляем параметр `copy` в определение метода `__init__`

Используем значение `copy` для инициализации публичного атрибута класса, который представлен как `self.copy`

Рис. 18 Метод `__init__`, задающий конструктор класса

В методе `__init__` мы выполняем инициализацию публичного атрибута `self.copy`.

Частный метод `__is_numpy` проверяет, является ли наш объект массивом NumPy, для проверки мы используем функцию `isinstance()`.

Частный метод `is_numpy`, предназначенный для правильной внутренней работы класса (проверка типа объекта)

```
def __is_numpy(self, X):
    # частный метод, который с помощью функции isinstance()
    # проверяет, является ли наш объект массивом NumPy
    return isinstance(X, np.ndarray)
```

Рис. 19 Частный метод `__is_numpy` для проверки типа объекта

Теперь более подробно разберем операции, происходящие в теле метода `.fit()`.

Метод `fit`, задающий обучение модели

Вычисляем единственный параметр модели – среднее значение переменной

```
def fit(self, X, y=None):
    # метод .fit() должен принимать
    # в качестве аргументов X и y

    # создаем пустой словарь, в котором ключами
    # будут имена/целые числа, а значениями - средние
    self._encoder_dict = {}

    # записываем результат метода __is_numpy
    is_np = self.__is_numpy(X)

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # если объект - массив NumPy
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # вычисляем среднее и записываем в словарь
            self._encoder_dict[col] = np.nanmean(X[:, col])
    # если объект - датафрейм pandas
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # вычисляем среднее и записываем в словарь
            self._encoder_dict[col] = X[col].mean()

    # fit возвращает self
    return self
```

Рис. 20 Метод `fit` для вычисления параметров

Первым делом мы создаем пустой словарь, в котором будем хранить средние значения по каждой переменной.

```
# создаем пустой словарь, в котором ключами
# будут имена/целые числа, а значениями - средние
self._encoder_dict = {}
```

Ключом будет название переменной (в случае датафрейма `pandas`) или соответствующее ей целочисленное значение (в случае массива `NumPy`), а значением – среднее значение переменной). При этом нам важно, чтобы наш массив `NumPy` был двумерным, поскольку классы библиотеки `scikit-learn` работают с двумерными массивами признаков, если же вы передадите одномерный массив `NumPy` или серию `pandas`, то получите ошибку:

```
# создаем 1D-массив признаков
X_toy = np.array([0.1, 0.4, 0.1, 0.9, 0.5])
```

```
# создаем 1D-массив меток
y_toy = np.array([1, 0, 0, 1, 0])

logreg = LogisticRegression(solver='lbfgs',
                             max_iter=200)
logreg.fit(X_toy, y_toy)

ValueError: Expected 2D array, got 1D array instead:
array=[0.1 0.4 0.1 0.9 0.5].
Reshape your data either using array.reshape(-1, 1) if your data has a single
feature or array.reshape(1, -1) if it contains a single sample.
```

Добавляем проверку формы массива. Если массив является одномерным, применяем метод `.reshape(-1, 1)` и получаем двумерный массив.

```
# если 1D-массив, то переводим в 2D
if len(X.shape) == 1:
    X = X.reshape(-1, 1)
```

Таким образом, класс выполняет «тихое» неявное преобразование, что допустимо для классов, которые вы используете только для внутреннего применения. Если класс применяется в рамках большого и публичного проекта, то лучше сделать поведение класса явным.

Вместо применения метода `.reshape()` мы могли бы с помощью конструкции `assert` проверить корректность данных, с которыми должны работать методы нашего класса.

```
# проверяем, является ли наш массив двумерным
assert len(X.shape) == 2, 'Array must be 2-dim.'
```

В случае одномерного массива NumPy мы получили бы сообщение `'Array must be 2-dim'` и вручную преобразовали бы массив в двумерный.

А еще лучше использовать тип исключения `ValueError`.

```
# проверяем, является ли наш массив двумерным
if len(X.shape) == 1:
    raise ValueError('Array must be 2-dim.')
```

Затем записываем количество столбцов в переменную `ncols`. Для этого берем второй элемент кортежа, возвращаемого свойством `shape` (т.е. элемент с индексом 1, первый элемент нам не нужен, поскольку он содержит информацию о количестве строк).

```
# записываем количество столбцов
ncols = X.shape[1]
```

Затем мы записываем результат метода `__isnumpy` в переменную `is_np`.

```
# записываем результат __is_numpy()
is_np = self.__is_numpy(X)
```

Результат представляет собой булево значение `False/True`, поскольку функция `isinstance()` возвращает `True`, если указанный объект (в нашем случае объект – массив NumPy) является таковым, и `False` в противном случае. Проверка

типа объекта позволяет нам выполнять отдельную стратегию вычисления и применения параметров для массива NumPy и отдельную стратегию вычисления и применения параметров для датафрейма pandas. Качественно написанный класс должен уметь работать и с датафреймами pandas, и с массивами NumPy.

Например, мы не можем использовать для массива NumPy следующий программный код:

```
# по каждому столбцу датафрейма pandas
for col in X.columns:
    # вычисляем среднее и записываем в словарь
    self._encoder_dict[col] = np.mean(X[col])
```

Мы просто получим ошибку о том, что массив NumPy не имеет свойства `columns`.

Наконец, по каждой переменной вычисляем среднее значение и кладем в словарь. Способ итерирования по столбцам определяется значением переменной `is_np`. Для массива NumPy используем цикл `for` с функцией `range()`, в качестве аргумента функции `range()` используем `n_cols`, поэтому ключами словаря будут целочисленные значения, соответствующие переменным, а значениями – средние значения переменных. Вычисление средних значений выполняется с помощью функции библиотеки NumPy `nanmean()`. Для датафрейма pandas используем цикл `for` со списком имен переменных, получаемым с помощью свойства `columns`, поэтому ключами словаря будут имена переменных, а значениями – средние значения переменных. Вычисление средних значений выполняется с помощью метода библиотеки pandas `.mean()`.

Переходим к методу `.transform()`.

Метод `transform`,  
задающий при-  
менение модели

```
def transform(self, X):
    # transform принимает в качестве
    # аргумента только X

    # выполняем копирование массива во избежание
    # предупреждения SettingWithCopyWarning
    # "A value is trying to be set on a copy of
    # a slice from a DataFrame (Происходит попытка
    # изменить значение в копии среза данных
    # датафрейма)"

    if self.copy:
        X = X.copy()

    # записываем результат метода __is_numpy
    is_np = self.__is_numpy(X)

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем количество столбцов
    ncols = X.shape[1]

    # применяем преобразование к X
    # если объект - массив NumPy:
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # заменяем пропуски средним
            # значением из словаря
            X[:, col] = np.nan_to_num(
                X[:, col],
                nan=self._encoder_dict[col])
    # если объект - датафрейм pandas:
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # заменяем пропуски средним
            # значением из словаря
            X[col] = np.where(X[col].isnull(),
                              self._encoder_dict[col],
                              X[col])

    # transform возвращает X
    return X
```

Рис. 21 Метод `transform` для применения параметров

В тело метода `.transform()` мы первым делом добавили копирование объекта. В противном случае произойдет срабатывание предупреждения `SettingWithCopy`. Оно звучит так: **A value is trying to be set on a copy of a slice**

### from a DataFrame (Происходит попытка изменить значение в копии среза данных датафрейма).

Звучит несколько туманно, было бы лучше, если бы сообщение выглядело примерно так: **You are attempting to make an assignment on an object that is either a view or a copy of a DataFrame. This occurs whenever you make a subset selection from a DataFrame and then try to assign new values to this subset (Вы пытаетесь применить операцию присваивания к объекту, который является либо представлением, либо копией объекта DataFrame. Это происходит всякий раз, когда вы отбираете подмножество и затем пытаетесь присвоить этому подмножеству новые значения).** Одним словом, библиотеке pandas важно знать, с чем она работает – копией или представлением. С помощью метода `.copy()` мы избавляемся от этого предупреждения.

Мы вновь проверяем, не является ли массив NumPy одномерным, записываем количество столбцов в переменную `ncols`, записываем результат метода `_isnumpy` в переменную `is_np`.

В заключение по каждой переменной выполняем замену пропусков средним значением, взятым из словаря. Способ итерирования по столбцам определяется значением переменной `is_np`. Вновь для массива NumPy используем цикл `for` с функцией `range()`, в качестве аргумента функции `range()` используем `ncols`. Замена пропусков средними значениями выполняется с помощью функции библиотеки NumPy `np.nan_to_num()`. Для датафрейма pandas используем цикл `for` со списком имен переменных, получаемым с помощью свойства `columns`. Замена пропусков средними значениями выполняется с помощью функции библиотеки NumPy `np.where()` и метода `.isnull()` библиотеки pandas. Первый аргумент функции `np.where` – проверяемое условие, второй аргумент – что возвращаем, если условие выполняется, третий аргумент – что возвращаем, если условие не выполняется.

Теперь создаем игрушечные обучающий и тестовый наборы данных и проверяем на них наш класс. Качественно написанный класс должен работать как с датафреймами pandas (необходимо для удобства работы в pandas при создании первого прототипа, наброска модели), так и с массивами NumPy (необходимо для совместимости с конвейерами scikit-learn для проверки гипотез, связанных с гиперпараметрами моделей предподготовки и моделей машинного обучения). Начнем с игрушечных датафреймов pandas.

*# создаем игрушечный обучающий датафрейм pandas*

```
toy_train = pd.DataFrame(
    {'Balance': [8.3, np.NaN, 10.2, 3.1],
     'Age': [23, 29, 36, np.NaN]})
toy_train
```

	Balance	Age
0	8.3	23.0
1	NaN	29.0
2	10.2	36.0
3	3.1	NaN

```
# создаем игрушечный тестовый датафрейм pandas
toy_test = pd.DataFrame(
    {'Balance': [10.4, np.NaN, 22.5, 1.1],
     'Age': [13, 19, 66, np.NaN]})
toy_test
```

	Balance	Age
0	10.4	13.0
1	NaN	19.0
2	22.5	66.0
3	1.1	NaN

Теперь посмотрим, какими должны быть значения в обучающем и тестовом датафреймах pandas.

```
# смотрим, как будут выглядеть преобразования в игрушечных
# обучающем и тестовом датафреймах pandas
for col in toy_train.columns:
    toy_train[col].fillna(toy_train[col].mean(), inplace=True)
    toy_test[col].fillna(toy_train[col].mean(), inplace=True)
print('обучающий датафрейм')
print(toy_train)
print('')
print('тестовый датафрейм')
print(toy_test)
```

обучающий датафрейм

	Balance	Age
0	8.3	23.000000
1	7.2	29.000000
2	10.2	36.000000
3	3.1	29.333333

тестовый датафрейм

	Balance	Age
0	10.4	13.000000
1	7.2	19.000000
2	22.5	66.000000
3	1.1	29.333333

Создаем экземпляр нашего класса `MeanImputer`, обучаем (вычисляем параметр – среднее значение по каждой переменной) и применяем (заменяем пропуски с помощью найденного параметра – среднего значения по каждой переменной).

```
# создаем экземпляр класса MeanImputer
imp = MeanImputer()
# обучаем модель
imp.fit(toy_train)
# выполняем преобразование игрушечного
# обучающего датафрейма pandas
toy_train = imp.transform(toy_train)
toy_train
```

	Balance	Age
0	8.3	23.000000
1	7.2	29.000000
2	10.2	36.000000
3	3.1	29.333333

```
# выполняем преобразование игрушечного
# тестового датафрейма pandas
toy_test = imp.transform(toy_test)
toy_test
```

	Balance	Age
0	10.4	13.000000
1	7.2	19.000000
2	22.5	66.000000
3	1.1	29.333333

Видим, что значения, полученные с помощью класса `MeanImputer`, совпадают с ожидаемыми. Теперь применим наш класс для отдельной переменной датафрейма, отключив копирование. Обратите внимание на двойные квадратные скобки вокруг переменной `Age`, это необходимо, чтобы получить 2-мерный массив признаков, как того требует библиотека `scikit-learn`.

```
# создаем игрушечный обучающий датафрейм pandas
toy_train = pd.DataFrame(
    {'Balance': [8.3, np.NaN, 10.2, 3.1],
     'Age': [23, 29, 36, np.NaN]})
# создаем экземпляр класса, отключив копирование
imp = MeanImputer(copy=False)
# обучаем модель
imp.fit(toy_train[['Age']])
# применяем модель
toy_train['Age'] = imp.transform(toy_train[['Age']])
toy_train
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:57: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame. Try using
.loc[row_indexer,col_indexer] = value instead See the caveats in the documentation:
http://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy
```

	Balance	Age
0	8.3	23.000000
1	NaN	29.000000
2	10.2	36.000000
3	3.1	29.333333

Видим, что хотя преобразование выполнено, оно сопровождается выдачей предупреждения `SettingWithCopy`.

Давайте включим копирование и сделаем все то же самое.

```
# создаем игрушечный обучающий датафрейм pandas
toy_train = pd.DataFrame(
    {'Balance': [8.3, np.NaN, 10.2, 3.1],
     'Age': [23, 29, 36, np.NaN]})
# создаем экземпляр класса, включив копирование
imp = MeanImputer(copy=True)
# обучаем модель
imp.fit(toy_train[['Age']])
# применяем модель
toy_train['Age'] = imp.transform(toy_train[['Age']])
toy_train
```

	Balance	Age
0	8.3	23.000000
1	NaN	29.000000
2	10.2	36.000000
3	3.1	29.333333

Видим, что предупреждение `SettingWithCopy` не выводится.

Теперь проверим работу класса с массивами `NumPy`.

```
# создаем игрушечный обучающий массив NumPy
np_toy_train = np.array(pd.DataFrame(
    {'Balance': [8.3, np.NaN, 10.2, 3.1],
     'Age': [23, 29, 36, np.NaN]}))
np_toy_train

array([[ 8.3, 23. ],
       [ nan, 29. ],
       [10.2, 36. ],
       [ 3.1, nan]])

# создаем игрушечный тестовый массив NumPy
np_toy_test = np.array(pd.DataFrame(
    {'Balance': [10.4, np.NaN, 22.5, 1.1],
     'Age': [13, 19, 66, np.NaN]}))
np_toy_test

array([[10.4, 13. ],
       [ nan, 19. ],
       [22.5, 66. ],
       [ 1.1, nan]])

# обучаем модель
imp.fit(np_toy_train)
# выполняем преобразование игрушечного
# обучающего массива NumPy
np_toy_train = imp.transform(np_toy_train)
```

```

np_toy_train

array([[ 8.3         , 23.         ],
       [ 7.2         , 29.         ],
       [10.2         , 36.         ],
       [ 3.1         , 29.33333333]])

# выполняем преобразование игрушечного
# тестового массива NumPy
np_toy_test = imp.transform(np_toy_test)
np_toy_test

array([[10.4         , 13.         ],
       [ 7.2         , 19.         ],
       [22.5         , 66.         ],
       [ 1.1         , 29.33333333]])

```

Опять видим, что значения, полученные с помощью класса `MeanImputer`, совпадают с ожидаемыми. А теперь проверим работу нашего класса с одномерным массивом `NumPy`.

```

# создаем 1D-массив NumPy
array = np.array([8.3, np.NaN, 10.2, 3.1])
array

array([ 8.3, nan, 10.2, 3.1])

# проверяем размерность массива
array.ndim

1

# проверяем работу класса
imp.fit(array)
array = imp.transform(array)
array

array([[ 8.3],
       [ 7.2],
       [10.2],
       [ 3.1]])

# проверяем размерность массива
array.ndim

```

2

Часто вам придется работать с уже готовыми классами, выполнять их отладку, улучшать их, а для этого нужно подробно изучить тело класса. Не всегда все операции, происходящие внутри класса, будут очевидны, поэтому можно выполнить декомпозицию класса, разложить его на более простые компоненты. Давайте выполним частичную декомпозицию нашего класса `MeanImputer()`, подробно разберемся, что происходит внутри метода `.fit()`, на примере датафрейма `pandas` и массива `NumPy`.

```

# создаем игрушечный датафрейм pandas
toy_train = pd.DataFrame(
    {'Balance': [8.3, np.NaN, 10.2, 3.1],
     'Age': [23, 29, 36, np.NaN]})
# создаем пустой словарь encoder_dict
encoder_dict = {}
# по каждой переменной
for col in toy_train.columns:
    # печатаем имя
    print(col)
    # вычисляем среднее и записываем в словарь
    encoder_dict[col] = toy_train[col].mean()
    # печатаем словарь
    print(encoder_dict)
    print('')

# печатаем итоговый словарь
print('итоговый словарь', encoder_dict)

Balance
{'Balance': 7.2}

Age
{'Balance': 7.2, 'Age': 29.333333333333332}

итоговый словарь {'Balance': 7.2, 'Age': 29.333333333333332}

```

Видим, что мы создаем пустой словарь, а потом на каждой итерации добавляем пару «ключ-значение», ключами будут имена переменных, а значениями – средние значения.

```

# создаем игрушечный обучающий массив NumPy
np_toy_train = np.array(pd.DataFrame(
    {'Balance': [8.3, np.NaN, 10.2, 3.1],
     'Age': [23, 29, 36, np.NaN]}))
# создаем пустой словарь encoder_dict
encoder_dict = {}
# по каждой переменной
for col in range(np_toy_train.shape[1]):
    # печатаем имя
    print(col)
    # вычисляем среднее и записываем в словарь
    encoder_dict[col] = np.nanmean(np_toy_train[:, col])
    # печатаем словарь
    print(encoder_dict)
    print('')

# печатаем итоговый словарь
print('итоговый словарь', encoder_dict)

0
{0: 7.2}

1
{0: 7.2, 1: 29.333333333333332}

итоговый словарь {0: 7.2, 1: 29.333333333333332}

```

Видим, что мы создали пустой словарь, а потом на каждой итерации добавляем пару «ключ-значение», ключами будут целочисленные значения, соответствующие переменным, а значениями – средние значения.

После построения моделей предварительной подготовки мы строим модель машинного обучения.

При работе с классом – моделью машинного обучения – мы выполняем следующие операции:

- импортируем из соответствующего модуля класс, в котором реализована интересующая модель машинного обучения;
- создаем экземпляр класса – объект-модель;
- обучаем модель, т.е. вычисляем параметры модели, с помощью которых будем получать прогнозы, – используем метод `.fit()` объекта-модели;
- применяем модель, т.е. вычисляем прогнозы:
- для вычисления спрогнозированных значений зависимой переменной – используем метод `.predict()` объекта-модели;
- для вычисления вероятностей классов зависимой переменной – используем метод `.predict_proba()` объекта-модели;
- оцениваем качество модели с помощью полученных прогнозов и/или вероятностей – используем метод `.score()` объекта-модели (по умолчанию метрикой качества для классификаторов будет правильность, для регрессоров – R-квадрат).

Самый простой способ реализовать собственный класс, строящий модель машинного обучения, – воспользоваться наследованием базового класса `BaseEstimator`. Наш класс, как и любой класс предварительной подготовки библиотеки `scikit-learn`, должен иметь методы `__init__`, `.fit()` и `.predict()`. Наш класс должен иметь методы `__get_params__` и `__set_params__` (они наследуются от базового класса `BaseEstimator`), с помощью этих методов мы можем задавать и получать доступ к параметрам. Базовые классы `RegressorMixin` и `ClassifierMixin` позволяют нам воспользоваться методом `.score()`, для задачи регрессии будет вычислен R-квадрат, для задачи классификации – правильность.

Даже если вы будете реализовывать модель машинного обучения без учителя, метод `.fit()` должен принимать в качестве аргументов  $X$  и  $y$ . Это требуется для совместимости с конвейерами `scikit-learn`! Обычно для  $y$  задают значение `None`. В методе `.fit()` происходит вычисление параметров модели и всегда возвращается `self`.

Остальные требования будут такими же, как для классов, строящих модели предварительной подготовки. Все атрибуты инициализируются в методе `__init__`. Никаких операций с атрибутами в методе `__init__` не должно быть. Названия параметров должны совпадать с названиями атрибутов. Вспомогательные методы, предназначенные для правильной внутренней работы класса, оформляются в виде частных методов. Импорт библиотек, классов и модулей внутри методов не допускается.

Давайте посмотрим, как выглядят модели машинного обучения изнутри.

Метод ближайших соседей (`k nearest neighbors` – KNN), возможно, является самым простым методом машинного обучения. Построение модели заключа-

ется в запоминании обучающего набора данных. Для того чтобы сделать прогноз для новой точки данных, метод находит ближайшие к ней точки обучающего набора, то есть находит «ближайших соседей».

Начнем с классификации. В простейшем варианте метод ближайших соседей рассматривает лишь одного ближайшего соседа – точку, ближе всего расположенную к точке, которую мы хотим классифицировать. Прогнозом будет ответ, уже известный для данной точки обучающего набора. Если мы рассматриваем более одного соседа, то для присвоения метки используется *голосование* (*voting*). Это означает, что в случае бинарной классификации для каждой точки тестового набора мы подсчитываем количество соседей, относящихся к классу 0, и количество соседей, относящихся к классу 1. Затем мы присваиваем точке тестового набора наиболее часто встречающийся класс: другими словами, мы выбираем класс, набравший большинство среди  $k$  ближайших соседей.

Для регрессии мы просто используем *усреднение*, т.е. вычисляем среднее значение по меткам соседей.

Давайте напишем класс `KNN_Estimator`, в котором реализуем метод ближайших соседей для классификации и регрессии.

```
# пишем класс KNN-модели
class KNN_Estimator():
    """
    KNN-модель.

    Параметры:
    -----
    k: int, по умолчанию 2
        Количество ближайших соседей, которое определяет
        класс/значение предсказываемого наблюдения.
    task: string, 'classification' по умолчанию
        Тип решаемой задачи.
    """
    # пишем защищенный метод, вычисляющий евклидово расстояние
    def _euclidean_distance(self, x1, x2):
        """
        Вычисляет евклидово расстояние между двумя векторами.
        """
        distance = 0
        for i in range(len(x1)):
            distance += pow((x1[i] - x2[i]), 2)
        return math.sqrt(distance)

    # пишем защищенный метод голосования
    def _vote(self, neighbor_labels):
        """
        Возвращает самый часто встречающийся класс
        среди ближайших соседей.
        """
        # подсчитываем абсолютные частоты классов
        # для каждого наблюдения
        counts = np.bincount(neighbor_labels.astype('int'))
        # возвращаем индекс максимального значения -
        # максимальной абсолютной частоты
        return counts.argmax()
```

```

def __init__(self, k=5, task='classification'):
    # инициализируем k - количество ближайших соседей
    self.k = k
    # решаемая задача
    self.task = task
    # создаем пустой список, в котором будем
    # хранить ближайших соседей для
    # каждого наблюдения набора
    self.k_nearest_neighbors_ = []

def fit(self, X, y):
    # просто запоминаем обучающий массив признаков
    # и обучающий массив меток
    self.X_memorized = X
    self.y_memorized = y

def predict(self, X):
    # создаем массив прогнозов, равный
    # длине тестового набора
    y_pred = np.empty(X.shape[0])
    # для каждого наблюдения тестового набора
    # предсказываем наиболее часто встречающийся
    # класс / среднее значение среди k ближайших соседей
    if self.task == 'classification':
        for i, test_sample in enumerate(X):
            idx = np.argsort([self._euclidean_distance(
                test_sample, x) for x in self.X_memorized]):self.k]
            k_nearest_neighbors = np.array(
                [self.y_memorized[i] for i in idx])
            self.k_nearest_neighbors_.append(k_nearest_neighbors)
            y_pred[i] = self._vote(self.k_nearest_neighbors_[i])

    if self.task == 'regression':
        for i, test_sample in enumerate(X):
            idx = np.argsort([self._euclidean_distance(
                test_sample, x) for x in self.X_memorized]):self.k]
            k_nearest_neighbors = np.array(
                [self.y_memorized[i] for i in idx])
            self.k_nearest_neighbors_.append(k_nearest_neighbors)
            y_pred[i] = np.mean(self.k_nearest_neighbors_[i])

    return y_pred

```

Давайте создадим обучающий массив признаков, обучающий массив меток для классификации, тестовый массив признаков.

```

# создаем обучающий массив признаков
X_trn = np.array([[0.1, 0.2, 0.3],
                  [0.7, 0.5, 0.2],
                  [0.1, 0.2, 0.2],
                  [0.9, 0.7, 3.5],
                  [0.2, 0.4, 1.4],
                  [0.4, 0.1, 0.5]])

# создаем обучающий массив меток для классификации
y_trn = np.array([1, 0, 1, 0, 0, 1])

```

```
# создаем тестовый массив признаков
X_tst = np.array([[0.1, 0.7, 1.1],
                  [0.5, 0.3, 2.8],
                  [0.1, 0.1, 0.2],
                  [0.9, 0.7, 1.5]])
```

Теперь обучаем модель KNN-классификации и получаем прогнозы для тестового массива признаков.

```
# обучаем модель KNN-классификации
knn = KNN_Estimator(k=3, task='classification')
knn.fit(X_trn, y_trn)

# получаем прогнозы для тестового
# массива признаков
pred = knn.predict(X_tst)
pred

array([1., 0., 1., 0.]
```

Посмотрим ближайших соседей по каждому наблюдению тестового массива признаков.

```
# посмотрим ближайших соседей по каждому
# наблюдению тестового массива признаков
knn.k_nearest_neighbors_

[array([0, 1, 1]), array([0, 0, 1]), array([1, 1, 1]), array([0, 1, 0])]
```

Создаем обучающий массив меток для регрессии.

```
# создаем обучающий массив меток для регрессии
y_trn = np.array([1.2, 0.5, 1.4, 2.2, 3.5, 5.9])
```

Теперь обучаем модель KNN-регрессии и получаем прогнозы для тестового массива признаков.

```
# обучаем модель KNN-регрессии
knn = KNN_Estimator(k=3, task='regression')
knn.fit(X_trn, y_trn)

# получаем прогнозы для тестового
# массива признаков
pred = knn.predict(X_tst)
pred

array([3.53333333, 3.86666667, 2.83333333, 3.3      ])
```

Посмотрим ближайших соседей по каждому наблюдению тестового массива признаков.

```
# посмотрим ближайших соседей по каждому
# наблюдению тестового массива признаков
knn.k_nearest_neighbors_
```

```
array([3.5, 5.9, 1.2]),
array([2.2, 3.5, 5.9]),
array([1.4, 1.2, 5.9]),
array([3.5, 5.9, 0.5])]
```

## Задачи с собеседований (Python)

1. Напишите функцию, которая находит общие элементы, кратные 7, в двух списках ниже:

```
a = [2, 4, 7, 9, 14, 20, 21, 22]
b = [3, 5, 8, 10, 14, 20, 21, 30]
```

2. Напишите функцию, возвращающую список из элемента первого списка, отсортированный по элементам второго (порядок элементов по совпадающему «ключу» второго не важен):

```
a = ["a", "b", "c", "d", "e", "f"]
b = [1, 0, 9, 3, 2, 0]
```

## 7.2. СТРОИМ СВОЙ ПЕРВЫЙ КОНВЕЙЕР МОДЕЛЕЙ

Давайте начнем работу с данными. Для этого нужно импортировать необходимые библиотеки, модуль `os`, функцию `train_test_split()` и классы `StandardScaler` и `LogisticRegression`. Модуль `os` позволяет взаимодействовать с операционной системой – узнавать/менять файловую структуру, переменные среды, узнавать имя и права пользователя и др. Функция `train_test_split()` разбивает данные на обучающие и тестовые массивы признаков и меток. Класс `StandardScaler` потребуется для предварительной подготовки. Класс `LogisticRegression` необходим для построения модели машинного обучения – логистической регрессии.

```
# импортируем библиотеки pandas, numpy
import pandas as pd
import numpy as np
# импортируем модуль os и функцию train_test_split()
import os
from sklearn.model_selection import train_test_split
# импортируем класс StandardScaler,
# выполняющий стандартизацию
from sklearn.preprocessing import StandardScaler
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression
```

Взглянем на наш рабочий каталог.

```
# взглянем на наш рабочий каталог
os.getcwd()
```

```
'/Users/artemgruzdev/Documents/GitHub/Data Preprocessing in Python'
```

Если данные лежат в другом каталоге, можем сменить его с помощью метода `os.chdir()`.

Давайте загрузим данные с помощью функции `read_csv()` библиотеки `pandas`. Они записаны в файле `StateFarm.csv`, который находится в каталоге `Data` нашего рабочего каталога `'/Users/artemgruzdev/Documents/GitHub/Data Preprocessing in Python'`.

Взглянем на данные.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm.csv', sep=';')
# смотрим данные
data.head(3)
```

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	18975.456110	65999	237	1	14	0	6	0
1	4715.321344	0	65	19	56	0	3	0
2	5018.885233	54500	63	28	17	0	6	0

Перед нами – данные американской автостраховой компании `StateFarm`. Они представляют собой записи о 8262 клиентах, классифицированных на два класса: 0 – отклика нет на предложение автостраховки (7435 клиентов) и 1 – отклик есть на предложение автостраховки (827 клиентов). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- количественный признак *Пожизненная ценность клиента* [`Customer Lifetime Value`];
- количественный признак *Доход клиента* [`Income`];
- количественный признак *Размер ежемесячной автостраховки* [`Monthly Premium Auto`];
- количественный признак *Количество месяцев со дня подачи последнего страхового требования* [`Months Since Last Claim`];
- количественный признак *Количество месяцев с момента заключения страхового договора* [`Months Since Policy Inception`];
- количественный признак *Количество открытых страховых обращений* [`Number of Open Complaints`];
- количественный признак *Количество полисов* [`Number of Policies`];
- бинарная зависимая переменная *Отклик на предложение автостраховки* [`Response`].

Мы будем учить модель правильно классифицировать клиентов на неоткликнувшихся и откликнувшихся. Метрикой качества будет правильность (accuracy). Правильность – это количество правильно спрогнозированных наблюдений, поделенное на общее количество наблюдений.

Обязательно нужно выполнить проверку (валидацию) модели, т.е. посмотреть, как модель выдает прогнозы на данных, не участвовавших в обучении. Самый простой способ проверки – случайное разбиение на обучающую и тестовую выборки.

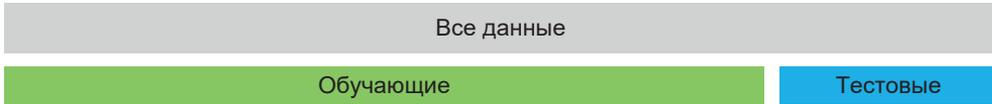


Рис. 22 Разбиение данных на обучающую и тестовую выборки

Сначала мы случайным образом разбиваем имеющиеся данные на две выборки: обучающую и тестовую. Формирование тестовой выборки – это способ преодолеть такие несовершенства неидеального мира, как ограничения в объеме данных и ресурсов, а также невозможность получения дополнительных данных из порождающего распределения. В данном случае тестовая выборка должна представлять собой новые, еще неизвестные модели данные. Важно использовать тестовую выборку лишь однократно. Обычно 2/3 доступных данных назначают в обучающую выборку, а оставшуюся 1/3 данных – в тестовую выборку. Другими популярными методами разбиения на обучающую/тестовую выборки являются 60/40, 70/30, 80/20 или даже 90/10, если набор данных относительно велик.

Затем необходимо построить на обучающей выборке (обучить) модели предварительной подготовки, например модель импутации, модель стандартизации и только потом модель машинного обучения, которая, как мы предполагаем, может оказаться подходящей для решения данной задачи. Таким образом, в реальности мы чаще всего строим конвейер моделей.

После обучения моделей возникает закономерный вопрос: а насколько «хорошо» качество полученного конвейера? И вот теперь наступает время использовать независимую тестовую выборку. Поскольку модели внутри конвейера еще «не видели» эти тестовые данные, такой шаг даст относительно надежную и несмещенную оценку качества на новых, незнакомых данных. Мы берем тестовую выборку и используем последнюю модель конвейера – модель машинного обучения для прогнозирования меток классов зависимой переменной по наблюдениям тестовой выборки. Затем мы берем спрогнозированные метки классов и сравниваем их с фактическими метками классов для оценки качества. Однако есть несколько тонких моментов.

У любой модели есть параметры, которые мы находим в ходе обучения. Например, у нас будет класс `SimpleImputer`, который обучает модель предварительной подготовки – модель импутации (замены) пропущенных значений. Параметрами модели импутации будут статистики, которые мы используем для импутации пропусков (среднее, мода, медиана).

Часто нам придется пользоваться классом `StandardScaler`, строящим модель стандартизации. Самая простая стандартизация подразумевает, что из каждого значения переменной мы вычтем среднее значение и полученный результат разделим на стандартное отклонение (в случае присутствия бинарных переменных для улучшения интерпретируемости делят на два стандартных отклонения):

$$\frac{x_i - \text{mean}(x)}{\text{stdev}(x)}$$

```

from sklearn.preprocessing import StandardScaler(copy=True, ←
with_mean=True, ←
with_std=True) ←

```

Если задано False, пробует избежать копирования и вместо этого выполняет стандартизацию на месте. Стандартизация на месте не всегда гарантируется. Например, если данные не являются массивом NumPy или CSR матрицей из модуля `scipy.sparse`, все равно может быть возвращена копия

Центрирует данные (вычитает из исходного значения переменной среднее значение) перед тем, как поделить на стандартное отклонение

Делит на стандартное отклонение

Параметрами модели стандартизации будут статистики, которые мы используем для стандартизации (среднее и стандартное отклонение).

Стандартизация необходима для некоторых методов машинного обучения, в частности для линейных моделей, которые мы будем строить чуть позже. Она приводит количественные независимые переменные к единому масштабу. Поэтому стандартизацию еще называют масштабированием, не путайте с масштабируемостью – свойством метода или модели продемонстрировать приемлемую или высокую скорость вычислений с увеличением размера набора данных (например, метод градиентного бустинга, реализованный в LightGBM, имеет хорошую масштабируемость, а метод иерархического кластерного анализа имеет плохую масштабируемость, поскольку медленно выполняется на больших наборах данных). Если не привести признаки к единому масштабу, то прогноз будут определять признаки, имеющие наибольший разряд и соответственно наибольшую дисперсию. Различный масштаб признаков приведет к ухудшению сходимости в случае применения градиентного спуска (например, в логистической регрессии градиентный спуск используют для поиска таких регрессионных коэффициентов, при которых мы получаем наименьшее значение логистической функции потерь). Кроме того, единый масштаб позволит нам сравнивать регрессионные коэффициенты при признаках между собой. Категориальные признаки стандартизировать не нужно. Мы подробнее поговорим обо всем этом во втором томе.

Для класса `LogisticRegression`, строящего модель машинного обучения – модель логистической регрессии, параметрами будут регрессионные коэффициенты для соответствующих признаков. Регрессионные коэффициенты мы находим в ходе градиентного спуска, лежащего в основе обучения логистической регрессии.

Для класса `DecisionTreeClassifier`, строящего иную модель машинного обучения – дерево решений CART, параметрами будут правила расщепления (признак расщепления и расщепляющее значение). Правила расщепления мы находим в ходе процедуры рекурсивного разбиения данных на узлы.

Помимо параметров, у модели есть гиперпараметры. Параметры мы находим в ходе обучения модели. А вот гиперпараметры – это параметры, которые нельзя «выучить» в процессе обучения, они сами регулируют ход обучения, их задают перед обучением модели и настраивают на отложенной выборке.

Модель импутации не может самостоятельно выяснить оптимальную стратегию импутации. Поэтому стратегия импутации – это гиперпараметр (гиперпараметр `strategy` класса `SimpleImputer`), который позволяет улучшить качество модели и настраивается на отложенной выборке.

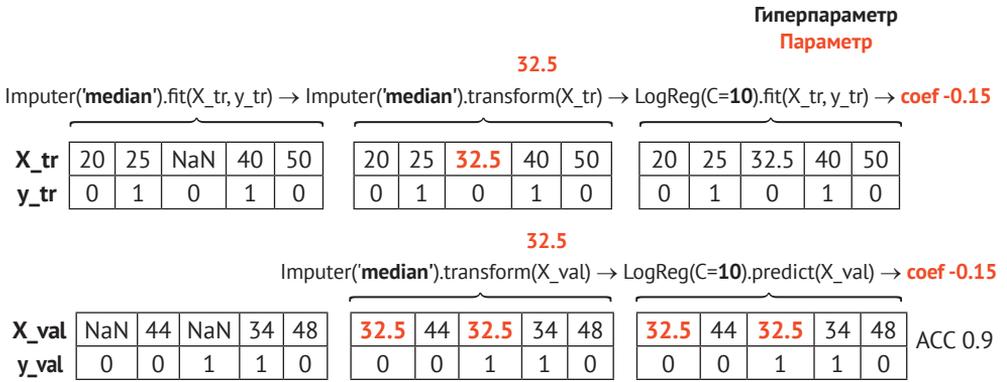
Модель стандартизации не может самостоятельно выяснить оптимальную стратегию стандартизации. Поэтому можно подбирать стандартизацию (например, вместо класса `StandardScaler` попробовать класс `MinMaxScaler`), и это позволяет улучшить качество модели.

Логистическая регрессия не может в процессе обучения самостоятельно выяснить оптимальное значение силы регуляризации. Поэтому сила регуляризации – это гиперпараметр модели (гиперпараметр `C` класса `LogisticRegression`), который позволяет улучшить качество модели и настраивается на отложенной выборке. Дерево решений `CART` не может самостоятельно выяснить оптимальное значение максимальной глубины. Поэтому максимальная глубина – это тоже гиперпараметр (гиперпараметр `max_depth` класса `DecisionTreeClassifier`, строящего дерево `CART`), который мы настраиваем на отложенной выборке.

Не путайте параметры и гиперпараметры с параметрами – именами, которые указываются при объявлении метода `__init__` класса и являются нашими внешними способами обратиться к атрибуту, находящемуся внутри класса.

И вот когда мы строим модели с разными значениями гиперпараметров на обучающей выборке, а проверяем их качество на тестовой выборке, возникает проблема. Мы используем тестовую выборку и для настройки гиперпараметров, и для оценки качества модели. Поскольку мы использовали тестовую выборку для настройки гиперпараметров, мы больше не можем использовать ее для оценки качества модели. Теперь для оценки качества модели нам необходим независимый набор данных, то есть набор, который не использовался для построения модели и настройки ее гиперпараметров и применяется лишь однократно для оценки качества модели. Помним, что тестовая выборка – прообраз новых данных, о которых мы ничего не знаем.

Допустим, у нас есть набор данных из одного признака и зависимой переменной. Мы можем выделить обучающую выборку – для обучения модели, валидационную выборку – для настройки гиперпараметров и тестовую выборку – для итоговой оценки качества выбранной модели. Затем мы создадим конвейер, состоящий, допустим, из модели импутации и модели логистической регрессии (без константы). Несколько раз обучим его на обучающей выборке, используя разные значения гиперпараметров – разные стратегии импутации и разные значения силы регуляризации. Каждый раз будем получать параметры – разные статистики для импутации и разные регрессионные коэффициенты. Ищем лучшую стратегию импутации и лучшее значение силы регуляризации, анализируя качество прогнозов на валидационной выборке. Валидационная выборка нужна только для настройки, это своего рода площадка, на которой кандидаты-гиперпараметры «соревнуются» между собой. Допустим, нашли конвейер с наилучшими гиперпараметрами. Для модели импутации лучшей стратегией оказалась замена медианой, а лучшим значением силы регуляризации для логистической регрессии стало значение 10. У модели импутации будет параметр – значение медианы для признака, у модели логистической регрессии параметром будет регрессионный коэффициент.



**Рис. 23** Обучение конвейера на обучающей выборке и применение к валидационной выборке

Здесь же заметим, что для импутации пропусков в обучающей и валидационной выборках мы можем использовать только медианы признаков, полученные на этапе обучения. И так с любой моделью предварительной подготовки, у которой параметрами будут статистики! Нельзя отдельно вычислить статистики для импутации на обучающей выборке, а затем отдельно вычислить статистики для импутации на валидационной выборке и потом использовать эти значения для импутации признака в соответствующей выборке. Помним, что вычисление статистик для импутации – это тоже модель предварительной подготовки данных, которую мы строим на обучающей выборке и применяем ее к переменным обучающей и валидационной выборок. С помощью валидационной выборки – прообраза новых данных – мы можем проверить эффективность модели импутации (модель импутации средним, модель импутации медианой). Например, если бы мы добавили стандартизацию, то для стандартизации переменных в обучающей и валидационных выборках мы могли бы использовать только среднее и стандартное отклонение переменных в обучающей выборке. Вычисление среднего и стандартного отклонения для каждой стандартизируемой переменной – это тоже модель предварительной подготовки данных, которую мы строим на обучающей выборке и применяем ее к переменным обучающей и валидационной выборок. С помощью валидационной выборки – прообраза новых данных – мы можем проверить эффективность той или иной модели стандартизации (стандартизация с вычитанием из исходного значения переменной среднего и деления на стандартное отклонение, стандартизация с вычитанием из исходного значения переменной минимального значения и деления на разницу между максимальным и минимальным значениями).

Итак, мы нашли наилучшую комбинацию гиперпараметров, нужно проверить, так ли она на самом деле хороша. Затем заново обучим конвейер моделей с найденными наилучшими гиперпараметрами на объединенной обучающе-валидационной выборке. Медиана для импутации и регрессионный коэффициент будут вычислены заново и будут отличаться от медианы и регрессионного коэффициента, полученных при обучении на обучающей выборке.

Для импутации пропусков в обучающе-валидационной и тестовой выборках мы можем использовать только медиану признака, вычисленную на обучающе-валидационной выборке. Получаем итоговую оценку качества конвейера на тестовой выборке.

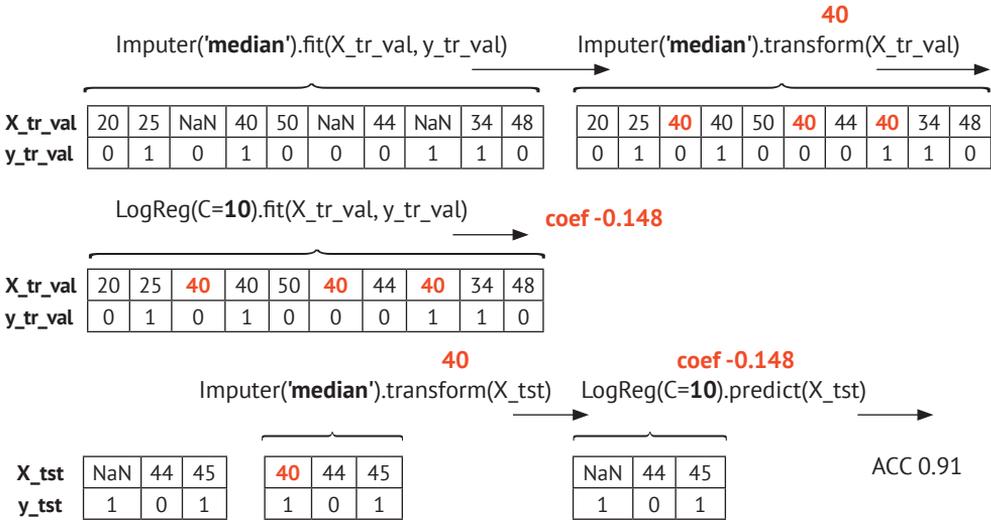


Рис. 24 Обучение конвейера на обучающе-валидационной выборке и применение к тестовой выборке

Если итоговая оценка качества конвейера моделей на тестовой выборке нас устроит, мы объединяем все три выборки, заново обучаем на объединенном наборе наш конвейер с найденными наилучшими гиперпараметрами и применяем к новым данным. Медиана для импутации и регрессионный коэффициент будут вычислены заново и будут отличаться от медианы и регрессионного коэффициента, полученного при обучении на обучающе-валидационной выборке. Для импутации пропусков в объединенном наборе (или, как его еще называют, историческом наборе) и новых данных мы можем использовать только медиану признака, вычисленную на историческом наборе.

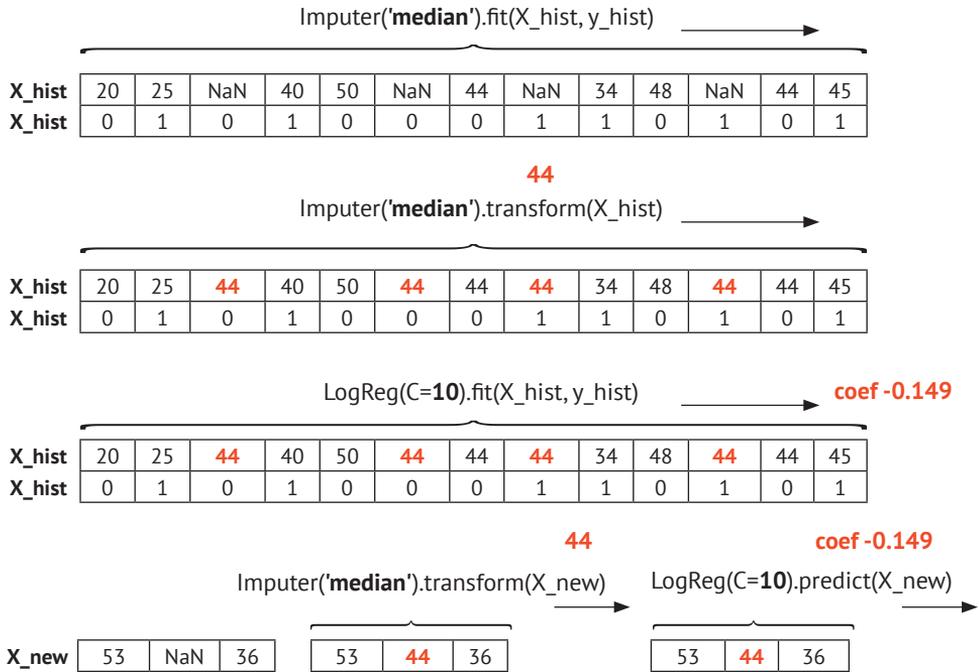


Рис. 25 Обучение конвейера на исторических данных и применение к новым данным

В противном случае мы просто будем настраивать модели нашего конвейера под тестовую выборку, ведь любой выбор, сделанный исходя из метрики на тестовом наборе, «сливает» модели информацию тестового набора. В итоге мы можем получить оптимистичные результаты. Полезно держать в голове картинку: как только вы использовали тестовый набор для оценки качества модели, он «сгорает».

Для простоты пока пренебрежем этим недостатком случайного разбиения на обучающую и тестовую выборки, допустим, наша задача – построить модель стандартизации, а затем модель машинного обучения – модель логистической регрессии, не прибегая к поиску оптимальных гиперпараметров. Такое часто бывает, когда, например, дана задача классификации и нужно сопоставить качество нескольких методов машинного обучения, строят базовые модели логистической регрессии, случайного леса и градиентного бустинга и сравнивают. Мы получим оценку качества модели на тестовых данных, и если качество нас устраивает, мы обучаем модель на всех доступных данных (т.е. объединяем обучающую и тестовую выборки) и применяем модель, обученную на всех доступных данных, к новым данным.

Давайте сделаем случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток. Это можно будет сделать с помощью функции `train_test_split()` модуля `model_selection` библиотеки `scikit-learn`. В `scikit-learn` для массива данных обычно используется заглавная  $X$ , а для массива меток – строчная  $y$ .

```

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    ① data.drop('Response', axis=1),
    ② data['Response'],
    ③ test_size=0.3,
    ④ stratify=data['Response'],
    ⑤ random_state=42)

```

- ① Указываем массив признаков. Для этого мы используем метод `.drop()` библиотеки `pandas`. Этот метод удаляет зависимую переменную `Response`, при удалении мы перемещаемся по оси столбцов, поэтому с помощью параметра `axis` указываем ось 1 (по умолчанию задана ось 0). По умолчанию операция не выполняется на месте (регулируется параметром `inplace`), в противном случае мы удалим зависимую переменную `Response` из датафрейма `data` и уже не сможем создать массив меток.
- ② Затем создаем массив меток, просто указав зависимую переменную.
- ③ С помощью параметра `test_size` настраиваем нужный размер тестовой выборки (в процентах). По умолчанию 0,25.
- ④ С помощью параметра `stratify` (по умолчанию не используется) можно задать стратифицированное разбиение на обучение и тест, чтобы распределение классов зависимой переменной в тестовой выборке соответствовало распределению классов в обучающей.
- ⑤ Поскольку разбиение является случайным, надо позаботиться о воспроизводимости результатов. Для этого с помощью параметра `random_state` задаем стартовое значение генератора псевдослучайных чисел.

Итак, мы получили обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток. Теперь мы можем обучать модель предварительной подготовки данных – модель стандартизации.

Создаем экземпляр класса `StandardScaler`.

```

# создаем экземпляр класса StandardScaler
standardscaler = StandardScaler()

```

В нашем наборе только количественные признаки, если бы здесь были категориальные признаки, мы обязательно превратили бы их в количественные с помощью дамми-кодирования (линейные модели работают только с количественными признаками, и для моделирования мы используем массивы `NumPy`, каждый столбец которого должен быть количественным признаком, помним, что датафреймы `pandas` внутренне преобразовываются в массивы `NumPy`). У нас каждый уровень категориальной переменной стал бы отдельным бинарным столбцом со значениями 0 или 1, и такие переменные не нужно стандартизировать.

С помощью метода `.fit()` мы строим модель `standardscaler` на обучающем массиве признаков. В данном случае метод `.fit()` вычисляет параметры модели – среднее значение и стандартное отклонение для каждой переменной обучающего массива признаков.

```

# обучаем модель стандартизации, т.е. по каждому признаку
# в обучающем массиве признаков вычисляем
# среднее значение признака и стандартное
# отклонение признака для трансформации
standardscaler.fit(X_train)

```

Чтобы применить модель к нашим данным, то есть *отмасштабировать* (*scale*) обучающие и тестовые данные, мы воспользуемся методом `.transform()`. Под капотом метод `.transform()` применяет параметры, найденные с помощью метода `.fit()`, – из каждого значения переменной обучающего и тестового массивов признаков вычитает среднее значение соответствующей переменной в обучающем массиве признаков и делит на стандартное отклонение этой переменной, также взятое по обучающему массиву признаков.

При этом значения NaN обрабатываются как пропущенные значения: игнорируются при обучении и сохраняются в ходе применения.

```
# применяем модель стандартизации к обучающему массиву признаков: из исходного
# значения признака вычитаем среднее значение признака, вычисленное
# по ОБУЧАЮЩЕМУ массиву признаков, и результат делим на стандартное
# отклонение признака, вычисленное по ОБУЧАЮЩЕМУ массиву признаков
X_train_standardscaled = standardscaler.transform(X_train)
# применяем модель стандартизации к тестовому массиву признаков: из исходного
# значения признака вычитаем среднее значение признака, вычисленное
# по ОБУЧАЮЩЕМУ массиву признаков, и результат делим на стандартное
# отклонение признака, вычисленное по ОБУЧАЮЩЕМУ массиву признаков
X_test_standardscaled = standardscaler.transform(X_test)
```

Ранее мы говорили, что для импутации пропусков и стандартизации признаков в обучающей выборке и выборке, которую используем для проверки, можно использовать только статистики, полученные на этапе обучения. Теперь сделаем более широкое обобщение. Если вы используете такие операции, как укрупнение редких категорий по порогу, импутацию пропусков статистиками, стандартизацию, биннинг и конструирование признаков на основе статистик (*frequency encoding*, *likelihood encoding*), т.е. если вы используете *любые операции, предполагающие вычисления по набору данных*, они должны быть осуществлены после разбиения на обучающую и тестовую выборки (если предполагается проверочная выборка, то после разбиения на обучающую и проверочную выборки, а потом после разбиения на обучающе-проверочную и тестовую выборки).

Если мы используем случайное разбиение на обучающую и тестовую выборки и выполняем перечисленные операции до разбиения, получается, что при вычислении статистик для импутации пропусков, среднего и стандартного отклонения для стандартизации, правил биннинга, частот и вероятностей положительного класса зависимой переменной в категориях признака для *frequency encoding* и *likelihood encoding* соответственно использовались все наблюдения набора, часть из которых потом у нас войдут в тестовую выборку (по сути, выборку новых данных). Поэтому получается, что статистики для импутации, статистики для стандартизации, правила биннинга, частоты и вероятности положительного класса в категориях признака, которые мы получили на всем наборе, пришли к нам частично из «будущего» (из новой, тестовой выборки, которой по факту еще нет). Однако мы должны смоделировать наиболее близкую к реальности ситуацию, когда у нас есть только обучающая выборка, а никаких новых данных еще нет.

Теперь обучим модель машинного обучения – модель логистической регрессии, реализованную в классе `LogisticRegression`. Логистическая регрессия используется для прогнозирования бинарной зависимой переменной: не вернет или вернет кредит, не откликнется или откликнется на предложение автостраховки.

Импортируем класс `LogisticRegression`, создаем экземпляр класса `LogisticRegression` и обучаем.

```
# создаем экземпляр класса LogisticRegression
logreg = LogisticRegression(solver='lbfgs', max_iter=200)
# обучаем модель логистической регрессии, т.е.
# находим параметры - регрессионные коэффициенты
logreg.fit(X_train_standardscaled, y_train)
# оцениваем качество модели на обучающих данных
print("Правильность на обучающей выборке: {:.3f}".format(
    logreg.score(X_train_standardscaled, y_train)))
# оцениваем качество модели на тестовых данных
print("Правильность на тестовой выборке: {:.3f}".format(
    logreg.score(X_test_standardscaled, y_test)))
```

```
Правильность на обучающей выборке: 0.900
Правильность на тестовой выборке: 0.900
```

Обратите внимание, что для печати результатов мы воспользовались методом `.format()`. Однако можно применить способ с использованием f-строки или так называемый «быстрый» способ с использованием оператора `%` (позволяет избежать применения фигурных скобок внутри круглых скобок для форматирования количественных значений).

```
# применим для печати f-строки
train_score = logreg.score(X_train_standardscaled, y_train)
test_score = logreg.score(X_test_standardscaled, y_test)
print(f"Правильность на обучающей выборке: {train_score:.3f}")
print(f"Правильность на тестовой выборке: {test_score:.3f}")
```

```
Правильность на обучающей выборке: 0.900
Правильность на тестовой выборке: 0.900
```

```
# применим для печати оператор %
print("Правильность на обучающей выборке: %.3f" % train_score)
print("Правильность на тестовой выборке: %.3f" % test_score)
```

```
Правильность на обучающей выборке: 0.900
Правильность на тестовой выборке: 0.900
```

Теперь получим спрогнозированные классы и вероятности классов (более точно – оценки уверенности модели в том или ином классе зависимой переменной).

```
# вычисляем спрогнозированные значения зависимой переменной
# для тестового массива признаков
logreg_predvalues = logreg.predict(X_test_standardscaled)
logreg_predvalues[:5]

aggay([0, 0, 0, 0, 0])
```

```
# вычисляем вероятности классов зависимой переменной
# для тестового массива признаков
```

```
logreg_probabilities = logreg.predict_proba(
    X_test_standardscaled)
logreg_probabilities[:5]

array([[0.91027855, 0.08972145],
       [0.89847518, 0.10152482],
       [0.88300257, 0.11699743],
       [0.90234211, 0.09765789],
       [0.92554507, 0.07445493]])
```

Давайте взглянем на константу и регрессионные коэффициенты. Для этого нам надо воспользоваться атрибутами `intercept_` и `coef_`.

```
# извлекаем константу
intercept = np.round(logreg.intercept_[0], 3)
intercept

-2.205

# извлекаем коэффициенты
coef = np.round(logreg.coef_, 3)
coef

array([[ -0.022,  0.042,  0.079, -0.056, -0.004, -0.017, -0.109]])
```

С помощью уже знакомой функции `zip()` «сшиваем» константу и коэффициенты с названиями признаков.

```
# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], X_train.columns):
    print(feature, c)
```

```
Константа: -2.205
Регрессионные коэффициенты:
Customer Lifetime Value -0.022
Income 0.042
Monthly Premium Auto 0.079
Months Since Last Claim -0.056
Months Since Policy Inception -0.004
Number of Open Complaints -0.017
Number of Policies -0.109
```

Исходный коэффициент логистической регрессии показывает, насколько в среднем изменится логит – натуральный логарифм шансов наступления события при изменении признака на единицу своего измерения, притом что остальные признаки остаются неизменными.

Стандартизированный коэффициент логистической регрессии показывает, насколько в среднем изменится логит – натуральный логарифм шансов наступления события при изменении признака на одно стандартное отклонение, притом что остальные признаки фиксированы.

Здесь мы использовали логистическую регрессию для прогнозирования отклика на предложение автостраховки и применили стандартизацию. Регрессионный коэффициент при признаке *Количество полисов* [*Number of Policies*] отрицателен и равен  $-0,109$ , это обозначает, что при увеличении количества полисов на одно стандартное отклонение (что примерно составляет 2,368, мы просто вычисляем стандартное отклонение переменной *Количество полисов*) натуральный логарифм шансов отклика уменьшается на 0,109. Регрессионный коэффициент при признаке *Доход клиента* [*Income*] положителен и равен 0,042, это обозначает, что при увеличении дохода клиента на одно стандартное отклонение (что примерно составляет 30652,718, мы просто вычисляем стандартное отклонение переменной *Доход клиента*) натуральный логарифм шансов отклика увеличивается на 0,042.

### 7.3. РАЗБИРАЕМСЯ С ДИЛЕММОЙ СМЕЩЕНИЯ – ДИСПЕРСИИ И ЗНАКОМИМСЯ С БУТСТРЕПОМ

В прогнозном моделировании нам важно построить модель на обучающих данных, а затем получить точные прогнозы для новых, еще не встречавшихся данных, состоящих из тех же самых признаков, что и использованная нами обучающая выборка. Мы помним, что прообразом этих новых данных является валидационная или тестовая выборка. Если модель может выдавать точные прогнозы на ранее не встречавшихся данных, можно сказать, что модель обладает способностью *обобщать* результат на новые данные. Нам требуется построить модель с максимальной *обобщающей способностью* (*generalization*). Для задач классификации в качестве метрики обобщающей способности обычно используется правильность, AUC-ROC, для задач регрессии – корень из среднеквадратичной ошибки, R-квадрат. Определение правильности было дано выше, а смысл других метрик кратко поясним. Если говорить упрощенно, AUC-ROC – это вероятность того, что классификатор присвоит случайно отобранному наблюдению положительного класса более высокий ранг, чем наблюдению отрицательного класса (если пренебречь вероятностью того, что ранг обоих будет одинаковым). Значение AUC-ROC варьирует от 0,5 до 1. Чем выше, тем лучше.  $R^2$  показывает процент вариации или изменчивости зависимой переменной, который объясняется моделью.  $R^2$  принимает значения от 0 до 1. Чем выше, тем лучше. RMSE в среднем измеряет, насколько наши прогнозы по модели отличаются от фактических значений. Метрика RMSE, в отличие от  $R^2$ , не масштабирована в определенном диапазоне, она имеет размерность исследуемой величины (и в этом тоже есть удобство). Чем меньше, тем лучше. В четвертой части мы дадим более точные определения метрик.

Если обучающий набор и новые данные имеют много общего между собой, можно ожидать, что модель будет точно прогнозировать новые данные. Однако в ряде случаев на новых данных модель работает существенно хуже. Почему так происходит?

Причина заключается в излишней сложности модели. На этапе подготовки данных часто отсутствует априорная информация о полезности тех или

иных признаков. Избыточное включение признаков, не несущих новой информации, ведет к тому, что модель становится слишком сложной. Модель может быть слишком сложной и по своей структуре в силу неверно подобранных гиперпараметров: слишком большая глубина деревьев, слишком большое количество слоев в нейронной сети. Очень сложная модель слишком точно подстраивается под особенности обучающего набора, улавливает не только фактические взаимосвязи, но и случайные возмущения обучающих данных. По сути, такая модель восстанавливает не только искомую зависимость, но и выполняет подгонку конкретных наблюдений, фактически осуществляет аппроксимацию погрешностей собственных измерений. В итоге мы получаем модель, которая идеально работает на обучающем наборе, но плохо обобщает результат на новые данные, поскольку описывает случайные возмущения в данных, не имеющие ничего общего с истинной формой связи между зависимой переменной и признаками. Такую ситуацию называют *переобучением (overfitting)*.

С другой стороны, включение недостаточного числа полезных признаков, упрощение структуры модели за счет уменьшения глубины деревьев, количества слоев нейронной сети, наоборот, приводит к тому, что модель не может в достаточной мере уловить фактические зависимости, и качество модели даже на обучающей выборке остается довольно низким. Такую ситуацию называют *недообучением (underfitting)*.

Вы можете диагностировать недообучение по большой ошибке модели в обучающей выборке наряду с большой ошибкой модели в тестовой выборке. Борьба с недообучением будет заключаться в том, что мы усложняем модель с помощью настройки гиперпараметров (добавляем количество деревьев, увеличиваем количество итераций, увеличиваем глубину деревьев, уменьшаем регуляризацию<sup>3</sup>) и/или увеличиваем количество признаков (конструируем новые признаки, обогащаем данные новыми внешними признаками).

Переобучение можно диагностировать по очень низкой ошибке модели в обучающей выборке наряду с более высокой ошибкой модели в тестовой выборке. В таком случае мы упрощаем модель с помощью настройки гиперпараметров (уменьшаем глубину деревьев, уменьшаем количество итераций, увеличиваем регуляризацию) и/или уменьшаем количество признаков.

Мы настраиваем сложность модели вышеописанными способами и проверяем обобщающую способность, используя отложенную выборку.

<sup>3</sup> Для регрессии – приведение коэффициентов к нулевым значениям, для деревьев – случайный отбор признаков, для нейронных сетей – дропаут (случайное обнуление весов нейронов, передаваемых в последующий слой).

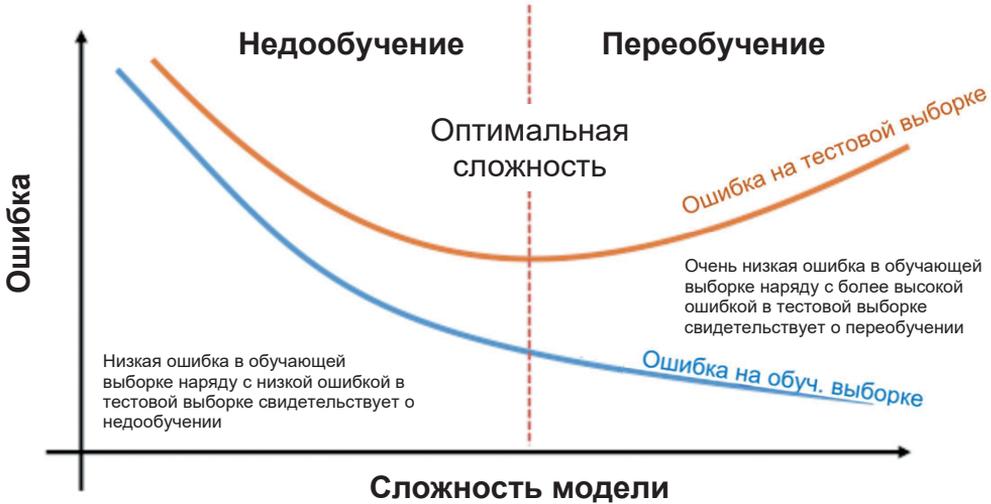


Рис. 26 Классическая иллюстрация недообучения и переобучения

Важнейшим понятием, помогающим понять проблему поиска оптимальной модели, является дилемма смещения–дисперсии. Ниже мы воспользуемся популярным и упрощенным изложением дилеммы дисперсии-смещения, поскольку тема выходит за рамки книги, а наиболее полное и точное изложение дилеммы вы найдете в книге: *Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin. Learning From Data: A Short Course. [United States]: AMLBook.com, 2012. Print.*

Предположим, что у нас есть обучающее множество, состоящее из набора точек  $x_1, \dots, x_n$  и вещественных значений  $y_i$ , связанных с каждой из этих точек  $x_i$ . Мы предполагаем, что есть функция с шумом  $y = f(x) + \varepsilon$ .

Мы хотим найти функцию  $\hat{f}(x; D)$ , которая аппроксимирует истинную функцию  $f(x)$  настолько хорошо, насколько возможно, в смысле некоторого алгоритма обучения, обученного на обучающем наборе (выборке)  $D = \{(x_1, y_1), (x_n, y_n)\}$ . Для оценки качества аппроксимации воспользуемся среднеквадратичной ошибкой между  $y$  и  $\hat{f}(x; D)$ , мы хотим, чтобы значение  $(y - \hat{f}(x; D))^2$  было минимальным как для точек  $x_1, \dots, x_n$ , так и для точек за пределами выборки. Сделать идеально мы не можем, поскольку  $y_i$  содержит шум. Шум по определению является случайной величиной. Это значит, что мы не можем предсказать его точное значение. Поэтому мы должны быть готовы принять неустранимую ошибку в любой функции, с которой будем работать.

Поиск функции  $\hat{f}$ , которая обобщается для точек вне обучающего набора, может быть осуществлен любым из бесконечного числа алгоритмов, используемых для обучения с учителем. Оказывается, что какую бы функцию мы ни выбрали, мы можем разложить ее ожидаемую ошибку следующим образом:

$$E_{D,\varepsilon} \left[ (y - \hat{f}(x; D))^2 \right] = \left( \text{Bias}_D \left[ \hat{f}(x; D) \right] \right)^2 + \text{Var}_D \left[ \hat{f}(x; D) \right] + \sigma^2.$$

После некоторых преобразований получаем:

$$E_{D,\varepsilon} \left[ (y - \hat{f}(x; D))^2 \right] = \left( E_D \left[ \hat{f}(x; D) \right] - f(x) \right)^2 + E_D \left[ \left( E_D \left[ \hat{f}(x; D) \right] - \hat{f}(x; D) \right)^2 \right] + \sigma^2.$$

Математические ожидания варьируют в зависимости от разных вариантов обучающего набора  $x_1, \dots, x_n, y_1, \dots, y_n$  из одного и того же совместного распределения  $P(x, y)$ , их можно получить, например, с помощью бутстрепа. Три члена представляют собой:

- квадрат смещения. Смещение (bias) – это отклонение среднего ответа обученного алгоритма от истинного ответа. Тем самым смещение показывает, насколько далеко наш прогноз оказался от фактического значения зависимой переменной;
- дисперсию. Дисперсия (variance) – это разброс ответов обученного алгоритма относительно среднего ответа. Дисперсия показывает, насколько сильно может изменяться прогноз в зависимости от выборки, иными словами, она характеризует чувствительность метода обучения к изменениям в обучающей выборке;
- неустраняемую ошибку  $\sigma^2$ .

Из всего вышесказанного следует, что выбор сложности модели – это компромисс между смещением и дисперсией. Чем более сложной является модель, тем больше точек данных она пропускает и тем больше будет смещение, а дисперсия будет меньше. Это хорошо иллюстрирует рисунок ниже.

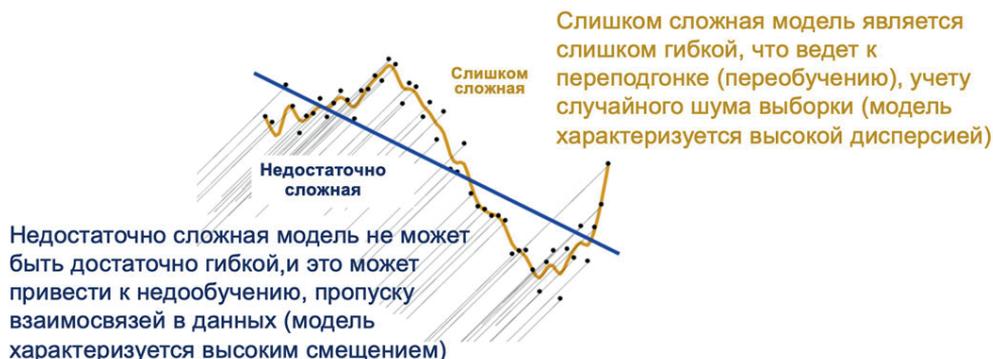


Рис. 27 Недостаточно сложная модель и слишком сложная модель

Таким образом, когда уменьшается смещение, то увеличивается дисперсия, и наоборот. При переобучении смещение падает, а дисперсия возрастает. При недообучении смещение возрастает, зато падает дисперсия. Линейные модели будут характеризоваться большим смещением и низкой дисперсией, а модели на основе деревьев будут характеризоваться большой дисперсией и низким смещением.

Разберем дилемму смещения и дисперсии на примере одиночного дерева решений. Мы можем построить глубокое дерево решений так, что листья будут содержать по одному наблюдению. Например, у нас в лист попало наблюдение с фактической меткой *Хороший клиент* и у нас спрогнозированной меткой будет *Хороший клиент*. Мы получаем однозначный прогноз: у нас будет идеально низкое смещение. При этом мы понимаем, что такое глубокое дерево, по сути, запомнило весь обучающий набор и будет иметь высокую дисперсию. Немного изменив набор данных, мы можем получить совершенно другое дерево.

Хорошим наглядным представлением, которое может помочь в понимании рассматриваемых идей, является мишень с кругами. Наши прогнозы – это выстрелы. Нас интересует удаленность выстрелов от «яблочка» (смещение) и кучность выстрелов (дисперсия).

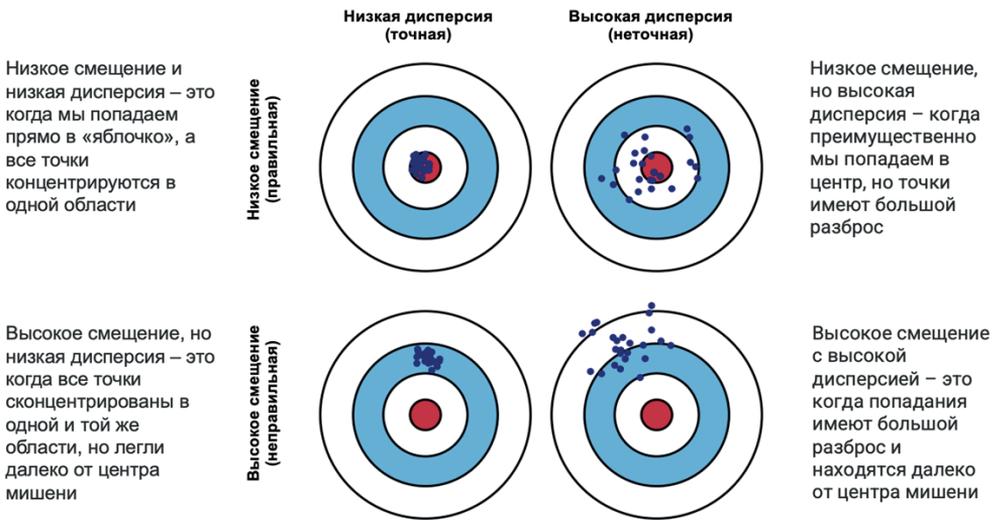


Рис. 28 Дилемма смещения–дисперсии на примере мишени с кругами

Графики кривых обучения и валидации тоже дают наглядное представление о смещении и дисперсии модели.

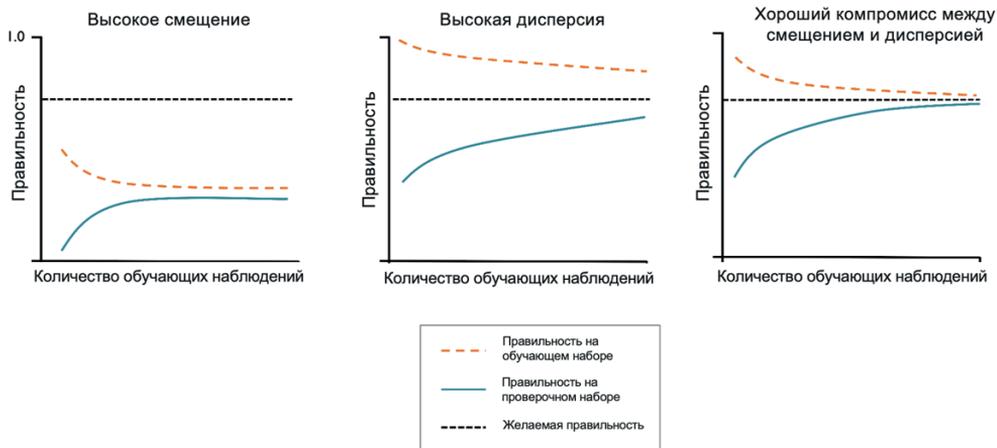


Рис. 29 Дилемма смещения–дисперсии на графиках кривых обучения и валидации

Рассказывая о дилемме смещения–дисперсии, мы упомянули, что разные варианты обучающего набора мы можем получить с помощью бутстрепа. Бутстреп широко используется для валидации и тестирования различных статистических гипотез. Давайте подробнее познакомимся с этим методом.

На основе исходной выборки мы сначала формируем *бутстреп-выборки* наших данных. Для этого из исходной выборки объемом  $n$  наблюдений мы случайным образом выбираем наблюдение с возвращением  $n$  раз (поскольку отбор с возвращением, то одно и то же наблюдение может быть выбрано несколько раз). Мы получаем выборку, которая имеет такой же размер, что и исходная выборка, однако некоторые наблюдения будут отсутствовать в нем (примерно 37 % наблюдений исходного набора), а некоторые попадут в него несколько раз.



Рис. 30 Бутстреп

Здесь сразу возникает вопрос, почему бутстреп-выборка не содержит примерно 37 % наблюдений обучающего набора. Вероятность выбрать одно наблюдение в каждой попытке равна  $\frac{1}{n}$ , соответственно, вероятность не выбрать это наблюдение равна  $1 - \frac{1}{n}$ . У нас всего  $n$  попыток, все попытки являются независимыми, поэтому вероятность не выбрать данное наблюдение при любой из попыток равна  $\left(1 - \frac{1}{n}\right)^n$ . Теперь подумаем, что будет происходить, когда  $n$  будет

увеличиваться. Мы можем взять предел при  $n \rightarrow \infty$ :  $\lim_{n \rightarrow \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e} \approx 0,368$ .

Предположим, что мы хотим создать бутстреп-выборку для списка из 10 наблюдений, проиндексированных от 1 до 10 включительно: ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10']. Первая бутстреп-выборка может выглядеть как ['10', '9', '7', '8', '1', '3', '9', '10', '10', '7']. Второй бутстреп-выборкой может быть ['4', '8', '5', '8', '3', '9', '2', '6', '1', '6']. Каждая бутстреп-выборка содержит ровно столько наблюдений, сколько наблюдений в исходной выборке. Из-за отбора с возвращением какие-то наблюдения могли не попасть в бутстреп-выборку, а какие-то наблюдения попали в бутстреп-выборку несколько раз. Мы видим, что в первой бутстреп-выборке наблюдение 10 встречается три раза. Каждый раз наблюдения, не попавшие в бутстреп-выборку, формируют out-of-bag-выборку, которую можно использовать для валидации модели.

Давайте на игрушечных данных проиллюстрируем, как можно использовать бутстреп для оценки качества модели.

Мы начнем с того, что импортируем необходимые библиотеки, классы и функции. Функция `display()` позволяет визуализировать несколько датафреймов в одном выводе. Класс `DecisionTreeRegressor` строит модель машинного обучения – модель дерева решений.

```
# импортируем библиотеку pandas, numpy
import pandas as pd
import numpy as np
# импортируем функции train_test_split() и display()
from sklearn.model_selection import train_test_split
from IPython.display import display
# импортируем класс StandardScaler,
# выполняющий стандартизацию
from sklearn.preprocessing import StandardScaler
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression
# импортируем класс DecisionTreeRegressor
from sklearn.tree import DecisionTreeRegressor
```

На основе первых 15 наблюдений набора данных от компании StateFarm сформируем игрушечный массив меток и массив признаков. Вместо задачи классификации возьмем задачу регрессии, в качестве зависимой переменной выберем переменную *Customer Lifetime Value*, в качестве признаков – переменные *Income* и *Monthly Premium Auto*. В качестве модели возьмем дерево решений.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm.csv', sep=';')

# сформируем игрушечные массив меток
# и массив признаков
var_lst = ['Income', 'Monthly Premium Auto',
           'Customer Lifetime Value']
toy_data = data[var_lst].head(10)
toy_labels = toy_data.pop('Customer Lifetime Value')
```

Взглянем на них.

```
# смотрим массив признаков
toy_data
```

	Income	Monthly Premium Auto
0	65999	237
1	0	65
2	54500	63
3	37260	62
4	68987	71
5	42305	117
6	65706	91
7	0	90
8	53243	66
9	0	70

```
# смотрим массив меток
toy_labels
```

```
0    18975.456110
1    4715.321344
2    5018.885233
3    4932.916345
4    5744.229745
5    13891.735670
6    7380.976717
7    3090.034104
8    2521.633095
9    2652.061785
Name: Customer Lifetime Value, dtype: float64
```

Теперь покажем, как можно сформировать бутстреп-выборку и out-of-bag-выборку.

```
# получаем индексы наблюдений исходной выборки
sample_indices = np.arange(toy_data.shape[0])

# задаем стартовое значение генератора
# псевдослучайных чисел
```

```

rng = np.random.RandomState(42)
# получаем индексы наблюдений бутстреп-выборки:
# sample_indices - используем индексы исходной выборки
# size - размер тот же, что у исходной выборки
# replace=True - отбор с возвращением
bootstrap_indices = rng.choice(sample_indices,
                               size=sample_indices.shape[0],
                               replace=True)

# получаем бутстреп-выборку
toy_data_boot = toy_data.iloc[bootstrap_indices]
toy_labels_boot = toy_labels.iloc[bootstrap_indices]

display(toy_data_boot)
display(toy_labels_boot)

```

	Income	Monthly Premium Auto
6	65706	91
3	37260	62
7	0	90
4	68987	71
6	65706	91
9	0	70
2	54500	63
6	65706	91
7	0	90
4	68987	71

```

6 7380.976717
3 4932.916345
7 3090.034104
4 5744.229745
6 7380.976717
9 2652.061785
2 5018.885233
6 7380.976717
7 3090.034104
4 5744.229745
Name: Customer Lifetime Value, dtype: float64

```

Видим, что в бутстреп-выборке некоторые наблюдения встретились несколько раз (например, наблюдение 6), а некоторые наблюдения не встретились ни разу. Теперь взглянем на out-of-bag-выборку.

```

# получаем out-of-bag-выборку
toy_data_out_boot = toy_data[~toy_data.index.isin(
    toy_data_boot.index)]
toy_labels_out_boot = toy_labels[~toy_labels.index.isin(
    toy_data_boot.index)]

display(toy_data_out_boot)
display(toy_labels_out_boot)

```

	Income	Monthly Premium Auto
0	65999	237
1	0	65
5	42305	117
8	53243	66

```
0 18975.456110
1 4715.321344
5 13891.735670
8 2521.633095
```

```
Name: Customer Lifetime Value, dtype: float64
```

В out-of-bag-выборке мы находим наблюдения, отсутствующие в бутстрепп-выборке.

Теперь напишем функцию, возвращающую бутстрепп-выборки и out-of-bag-выборки, и с помощью бутстрепа оценим качество модели дерева на наших игрушечных данных.

```
# пишем функцию, возвращающую бутстрепп-выборки
# и out-of-bag-выборки
def generate_bootstrap(rng, X, y, verbose=True):
    # получаем индексы наблюдений исходной выборки
    sample_indices = np.arange(X.shape[0])
    # получаем индексы наблюдений бутстрепп-выборки,
    # бутстрепп-выборка имеет тот же размер,
    # что и исходная, отбор с возвращением
    bootstrap_indices = rng.choice(sample_indices,
                                   size=sample_indices.shape[0],
                                   replace=True)
    X_boot = X.iloc[bootstrap_indices]
    y_boot = y.iloc[bootstrap_indices]
    X_out_boot = X[~X.index.isin(X_boot.index)]
    y_out_boot = y[~y.index.isin(X_boot.index)]
    if verbose:
        print(f"{i}-итерация")
        print(f"индексы в бутстрепп-выборке: {X_boot.index.tolist()}")
        print(f"индексы в out-of-bag-выборке: {X_out_boot.index.tolist()}\n")
    # возвращаем выборки
    return X_boot, y_boot, X_out_boot, y_out_boot
```

Возьмем 3 бутстрепп-выборки (на практике берут от 2000 до 10 000 бутстрепп-выборок). У нас будет три итерации. На каждой итерации на бутстрепп-выборке обучаем модель дерева, а на out-of-bag-выборке оцениваем качество модели, значение метрики кладем в предварительно созданный пустой список. В итоге получим список из трех значений метрики качества (по умолчанию для задачи регрессии вычисляется R-квадрат), вычисляем среднее значение метрики. Обратите внимание: начиная с версий Python 3.4+ мы можем воспользоваться функцией `mean()` модуля `statistics` для вычисления среднего в списке. А начиная с версий Python 3.8+ для значений с плавающей точкой мы можем использовать функцию `fmean()` этого же модуля, которая будет работать быстрее, чем `mean()`.

```

# создаем контейнер для генератора
# псевдослучайных чисел
rng = np.random.RandomState(42)

# создаем экземпляр класса StandardScaler
standardscaler = StandardScaler()

# создаем экземпляр класса DecisionTreeRegressor
tree = DecisionTreeRegressor(random_state=42)

# создаем пустой список, в который
# будем записывать значения R2
test_score_lst = []

# на каждой итерации...
for i in range(1, 4):
    # формируем бустреп-выборку и out-of-bag-выборку
    X_boot, y_boot, X_out_boot, y_out_boot = generate_bootstrap(
        rng, toy_data, toy_labels)
    # обучаем на бустреп-выборке
    tree.fit(X_boot, y_boot)
    # записываем значение R2
    test_score = tree.score(X_out_boot, y_out_boot)
    # кладем значение R2 в список
    test_score_lst.append(test_score)

```

1-итерация

индексы в бустреп-выборке: [6, 3, 7, 4, 6, 9, 2, 6, 7, 4]

индексы в out-of-bag-выборке: [0, 1, 5, 8]

2-итерация

индексы в бустреп-выборке: [3, 7, 7, 2, 5, 4, 1, 7, 5, 1]

индексы в out-of-bag-выборке: [0, 6, 8, 9]

3-итерация

индексы в бустреп-выборке: [4, 0, 9, 5, 8, 0, 9, 2, 6, 3]

индексы в out-of-bag-выборке: [1, 7]

*# смотрим список*

```
print(test_score_lst)
```

```
[-0.043216359884440836, 0.6652528178872501, -2.3683596789799646]
```

*# получаем среднее значение R2*

```
mean_r2 = sum(test_score_lst) / len(test_score_lst)
```

```
print("Среднее значение R2: %.3f" % mean_r2)
```

Среднее значение R2: -0.582

*# среднее значение в списке можно вычислить*

*# с помощью mean() или fmean() пакета statistics*

```
import statistics
```

```
mean_r2 = statistics.fmean(test_score_lst)
```

```
print("Среднее значение R2: %.3f" % mean_r2)
```

Среднее значение R2: -0.582

Если вам нужно оценить качество базовой модели, запускаете бутстреп на всем историческом наборе. Оценкой качества модели будет оценка качества, усредненная по out-of-bag-выборкам. Если нужно оценить гиперпараметры модели, то разбиваете набор на обучающую и тестовую выборки, на обучающей выборке запускаете бутстреп, out-of-bag-выборки выступают в качестве валидационных выборок. Модель с наилучшей комбинацией гиперпараметров, найденной в ходе бутстрепа, обучаете на всей обучающей выборке и применяете к тестовой выборке для итоговой оценки качества.

Если вы планируете перед построением модели осуществить операции предварительной подготовки, предполагающие вычисление статистик (импутацию, стандартизацию и прочее), такие операции должны происходить внутри процедуры бутстрепа. На каждой итерации в бутстреп-выборке вычисляете статистики для импутации и стандартизации, с помощью этих статистик заменяете пропуски и выполняете стандартизацию в бутстреп-выборке и out-of-bag-выборке. Давайте проиллюстрируем этот случай.

```
# создаем контейнер для генератора
# псевдослучайных чисел
rng = np.random.RandomState(42)

# создаем экземпляр класса StandardScaler
standardscaler = StandardScaler()

# создаем экземпляр класса LogisticRegression
logreg = LogisticRegression(solver='lbfgs', max_iter=200)

# создаем пустой список, в который будем
# записывать значения правильности
test_score_lst = []

# на каждой итерации...
for i in range(1000):
    # формируем бутстреп-выборку и out-of-bag-выборку
    X_boot, y_boot, X_out_boot, y_out_boot = generate_bootstrap(
        rng, X_train, y_train, verbose=False)
    # выполняем стандартизацию
    standardscaler.fit(X_boot)
    X_boot_scaled = standardscaler.transform(X_boot)
    X_out_boot_scaled = standardscaler.transform(X_out_boot)
    # обучаем на бутстреп-выборке
    logreg.fit(X_boot_scaled, y_boot)
    X_out_boot_scaled)
    # записываем значение правильности
    test_score = logreg.score(
        X_out_boot_scaled, y_out_boot)
    # кладем значение правильности в список
    test_score_lst.append(test_score)

# получаем среднее значение правильности
mean_acc = statistics.fmean(test_score_lst)
print("Среднее значение правильности: %.3f" % mean_acc)
```

Среднее значение правильности: 0.900

Вернемся к дилемме смещения–дисперсии. На практике полезно разложить функцию потерь на смещение и дисперсию. Это позволяет лучше понять, как настраиваемые нами гиперпараметры влияют на смещение и дисперсию. Давайте выполним разложение квадратичной функции потерь на смещение и дисперсию для конкретного примера с помощью функции `bias_variance_decomp()` из пакета Себастьяна Рашки `mlxtend`.

Мы воспользуемся набором данных `Boston Housing`. Задача, связанная с этим набором данных, заключается в том, чтобы спрогнозировать медианную стоимость домов в нескольких районах Бостона в 1970-е годы на основе такой информации, как уровень преступности, близость к реке Чарльз, удаленность от радиальных магистралей и т. д. Набор данных содержит 506 наблюдений и 13 признаков.

Импортируем необходимые данные.

```
# импортируем необходимые данные
from mlxtend.data import boston_housing_data
```

Теперь создаем обучающий и тестовый массивы признаков и массив значений зависимой переменной.

```
# создаем массив признаков и массив значений зависимой переменной
X, y = boston_housing_data()
# создаем обучающие и тестовые массивы признаков
# и значений зависимой переменной
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.3,
                                                    random_state=123)
```

Нам нужно написать функцию `_draw_bootstrap_sample()`, генерирующую бутстреп-выборки на основе обучающей выборки, во многом аналогичную той, что писали ранее. Она будет использована внутри функции `bias_variance_decomp()`.

```
# пишем функцию, генерирующую бутстреп-выборку
def _draw_bootstrap_sample(rng, X, y):
    # получаем индексы наблюдений исходной выборки
    sample_indices = np.arange(X.shape[0])
    # получаем индексы наблюдений бутстреп-выборки, бутстреп-выборка
    # имеет тот же размер, что и исходная, отбор с возвращением
    bootstrap_indices = rng.choice(sample_indices,
                                   size=sample_indices.shape[0],
                                   replace=True)
    # формируем бутстреп-выборку
    return X[bootstrap_indices], y[bootstrap_indices]
```

Пишем функцию `bias_variance_decomp()`, вычисляющую усредненное ожидаемое значение функции потерь, усредненное смещение, усредненную дисперсию.

```

# пишем функцию, вычисляющую усредненное ожидаемое значение функции
# потерь, усредненное смещение, усредненную дисперсию
def bias_variance_decomp(estimator, X_train, y_train, X_test, y_test,
                        num_rounds=200, random_seed=None):
    """
    Автор: Sebastian Raschka
    https://github.com/rasbt

    Параметры
    -----
    estimator: регрессор.
    X_train: обучающий набор для извлечения бутстреп-выборок.
    y_train: обучающий массив значений зависимой переменной.
    X_test: тестовый набор для вычисления среднеквадратичной функции
            потерь, смещения и дисперсии.
    y_test: тестовый массив значений зависимой переменной.
    num_rounds : количество итераций бутстрепа (целочисленное
                значение, по умолчанию 200).
    random_seed : стартовое значение генератора псевдослучайных
                чисел для создания бутстреп-выборок (целочисленное
                значение, по умолчанию None).

    Возвращает
    -----
    avg_expected_loss, avg_bias, avg_var : усредненное
        ожидаемое значение функции потерь, усредненное смещение,
        усредненную дисперсию (все значения с плавающей точкой),
        усреднение происходит по наблюдениям тестового набора.
    """

    # создаем контейнер для генератора
    # псевдослучайных чисел
    rng = np.random.RandomState(random_seed)
    # создаем массив из нулей, количество строк равно num_rounds,
    # а количество столбцов определяется количеством наблюдений
    # тестового набора, в него мы будем записывать прогнозы
    all_pred = np.zeros((num_rounds, y_test.shape[0]), dtype=int)

    # на каждой итерации...
    for i in range(num_rounds):
        # формируем бутстреп-выборку на основе обучающего набора
        X_boot, y_boot = draw_bootstrap_sample(rng, X_train, y_train)
        # обучаем регрессор на бутстреп-выборке и
        # выдаем прогнозы для тестового набора
        pred = estimator.fit(X_boot, y_boot).predict(X_test)
        # записываем прогнозы в массив all_pred
        all_pred[i] = pred

    # вычисляем усредненное ожидаемое значение функции потерь,
    # в нашем случае среднеквадратичную ошибку, усредненную
    # по среднеквадратичным ошибкам, полученным на каждой
    # итерации (количество итераций задается num_rounds)
    avg_expected_loss = np.apply_along_axis(
        lambda x:
            ((x - y_test) ** 2).mean(),
            axis=1,
            arr=all_pred).mean()

```

```

# вычисляем усредненный прогноз (берем среднее
# по оси 0 массива all_pred)
main_predictions = np.mean(all_pred, axis=0)

# вычисляем усредненное смещение, делим сумму квадратов
# разностей между усредненными прогнозами и фактическими
# значениями зависимой переменной в тестовом наборе на
# количество наблюдений тестового набора
avg_bias = np.sum((main_predictions - y_test) ** 2) / y_test.size
# вычисляем усредненную дисперсию, делим сумму квадратов
# разностей между усредненными прогнозами и прогнозами
# для тестового набора на количество прогнозов
avg_var = np.sum((main_predictions - all_pred) ** 2) / all_pred.size
return avg_expected_loss, avg_bias, avg_var

```

Мы сравним смещение и дисперсию дерева максимальной глубины (строится по умолчанию) со смещением и дисперсией дерева глубины 1 (т.е. дерево имеет один уровень ниже корневого узла).

```

# строим дерево максимальной глубины
tree = DecisionTreeRegressor(random_state=123)

# вычисляем усредненное ожидаемое значение функции потерь,
# усредненное смещение, усредненную дисперсию
avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
    tree, X_train, y_train, X_test, y_test,
    random_seed=123)
# печатаем результаты
ttl = "Усредненное ожидаемое значение функции потерь: %.3f"
print(ttl % avg_expected_loss)
print("Усредненное смещение: %.3f" % avg_bias)
print("Усредненная дисперсия: %.3f" % avg_var)

```

Усредненное ожидаемое значение функции потерь: 31.917  
 Усредненное смещение: 13.814  
 Усредненная дисперсия: 18.102

```

# строим дерево глубины 1
tree2 = DecisionTreeRegressor(random_state=123, max_depth=1)

# вычисляем усредненное ожидаемое значение функции потерь,
# усредненное смещение, усредненную дисперсию
avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
    tree2, X_train, y_train, X_test, y_test,
    random_seed=123)
# печатаем результаты
print(ttl % avg_expected_loss)
print("Усредненное смещение: %.3f" % avg_bias)
print("Усредненная дисперсия: %.3f" % avg_var)

```

Усредненное ожидаемое значение функции потерь: 53.199  
 Усредненное смещение: 41.990  
 Усредненная дисперсия: 11.209

Мы видим, что дерево максимальной глубины характеризуется меньшим смещением и большей дисперсией. И наоборот, короткое дерево имеет высокое смещение и низкую дисперсию.

Для лучшего понимания давайте подробнее посмотрим, что происходит под капотом функции `bias_variance_decomp()`.

Создаем игрушечные обучающие и тестовые массивы признаков и значений зависимой переменной и контейнер для генератора псевдослучайных чисел.

```
# создаем игрушечные обучающие и тестовые массивы
# признаков и значений зависимой переменной
```

```
X_train = np.array([[29.1, 19000.28, 15],
                   [67.3, 48800.81, 45],
                   [77.9, 89800.55, 188]])
X_test = np.array([[11.9, 89900.28, 199],
                  [37.8, 10600.82, 95],
                  [77.2, 99700.22, 87]])
```

```
y_train = np.array([22.6, 89.5, 17.3])
y_test = np.array([12.4, 96.9, 107.9])
```

```
# создаем контейнер для генератора
# псевдослучайных чисел
```

```
rng = np.random.RandomState(123)
```

Создаем массив `all_pred` из нулей, количество строк равно количеству итераций бутстрепа (у нас будет три итерации), а количество столбцов определяется количеством наблюдений тестового набора, в него мы будем записывать прогнозы каждого построенного дерева в целочисленном виде.

```
# создаем массив из нулей, количество строк равно количеству итераций,
# а количество столбцов определяется количеством наблюдений тестового
# набора, в этот массив мы будем записывать прогнозы
all_pred = np.zeros((3, y_test.shape[0]), dtype=int)
all_pred
```

```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

		Ось 1 (ось столбцов) наблюдения тестового набора		
		0	1	2
Ось 0 (ось строк) итерации	0	0	0	0
	1	0	0	0
	2	0	0	0

Теперь запускаем цикл `for`. Извлекаем бутстреп-выборку на основе обучающего набора, строим дерево регрессии по бутстреп-выборке, вычисляем прогнозы для тестового массива и записываем их в массив `all_pred`. И так три раза.

```
# на каждой итерации...
for i in range(3):
    # формируем бутстреп-выборку на основе обучающего набора
    X_boot, y_boot = draw_bootstrap_sample(rng, X_train, y_train)
    # обучаем регрессор на бутстреп-выборке и
    # выдаем прогнозы для тестового набора
    pred = tree.fit(X_boot, y_boot).predict(X_test)
    # записываем прогнозы в массив all_pred
    all_pred[i] = pred
```

Давайте вновь взглянем на массив `all_pred` с прогнозами и тестовый массив значений зависимой переменной.

```
# смотрим массив с прогнозами
all_pred
```

```
array([[17, 89, 89],
       [17, 22, 22],
       [17, 89, 89]])
```

```
# смотрим тестовый массив значений
# зависимой переменной
y_test
```

```
array([ 12.4, 96.9, 107.9])
```

Давайте вычислим среднеквадратичную ошибку по прогнозам, полученным на первой итерации.

```
# вычислим среднеквадратичную ошибку по прогнозам,
# полученным на первой итерации
mse_first_iter = (((17 - 12.4)**2) + ((89 - 96.9)**2) +
                  ((89 - 107.9)**2)) / 3
mse_first_iter

146.92666666666676
```

Разумеется, когда итераций будет много, удобнее применить функцию `np.apply_along_axis()`. Она вычисляет среднеквадратичную ошибку для каждой итерации. Для этого она каждый раз применяет `((x - y_test) ** 2).mean()` вдоль оси 1 (оси столбцов) массива `all_pred`.

```
# вычислим среднеквадратичные ошибки по прогнозам,
# полученным на каждой итерации
```

```
mse = np.apply_along_axis(
    lambda x:
        ((x - y_test)**2).mean(),
    axis=1,
    arr=all_pred)
mse
```

	all_pred			y_test			
	17	89	89	12.4	96.9	107.9	
	17	22	22				
	17	89	89				
	Ось 1 (ось столбцов)						Работаем по горизонтали
	0	1	2				
	$(17-12.4)**2$	$(89-96.9)**2$	$(89-107.9)**2$	$.mean()$	$\rightarrow$	146.9	
	$(17-12.4)**2$	$(22-96.9)**2$	$(22-107.9)**2$	$.mean()$	$\rightarrow$	4336.7	
	$(17-12.4)**2$	$(89-96.9)**2$	$(89-107.9)**2$	$.mean()$	$\rightarrow$	146.9	

```
array([ 146.92666667, 4336.66 , 146.92666667])
```

Вычисляем усредненное ожидаемое значение функции потерь, фактически среднеквадратичную ошибку, усредненную по среднеквадратичным ошибкам, полученным на каждой итерации.

```
# вычисляем усредненное ожидаемое значение функции потерь,
# в нашем случае среднеквадратичную ошибку, усредненную по
# среднеквадратичным ошибкам, полученным на каждой итерации
avg_expected_loss = np.apply_along_axis(
    lambda x:
        ((x - y_test)**2).mean(),
        axis=1,
        arr=all_pred).mean()
avg_expected_loss
```

1543.5044444444445

Ось 1 (ось столбцов)					
0	1	2			
$(17-12.4)**2$	$(89-96.9)**2$	$(89-107.9)**2$	.mean()	→	146.9
$(17-12.4)**2$	$(22-96.9)**2$	$(22-107.9)**2$		→	4336.7
$(17-12.4)**2$	$(89-96.9)**2$	$(89-107.9)**2$		→	146.9
				}	.mean() → 1543.5

Теперь вычисляем усредненный прогноз по каждому наблюдению (берем среднее по оси 0 массива all\_pred).

```
# вычисляем усредненный прогноз по каждому наблюдению
# (берем среднее по оси 0 массива all_pred)
main_predictions = np.mean(all_pred, axis=0)
main_predictions
array([17. , 66.66666667, 66.66666667])
```

		all_pred				
Ось 0 (ось строк)	0	17	89	89		
	1	17	22	22		
	2	17	89	89		
		np.mean()				
		↓	↓	↓		
		17.	66.7	66.7		

Вычисляем усредненное смещение. Для этого делим сумму квадратов разностей между усредненными прогнозами и фактическими значениями зависимой переменной в тестовом наборе на размер тестового набора (количество наблюдений тестового набора).

```
# вычисляем усредненное смещение, делим сумму квадратов разностей
# между усредненными прогнозами и фактическими значениями
# зависимой переменной в тестовом наборе на
# количество наблюдений тестового набора
avg_bias = np.sum((main_predictions - y_test)**2) / y_test.size
avg_bias
```

	<b>main_predictions</b>		<b>y_test</b>
878.4674074074074	17.    66.7    66.7		12.4    96.9    107.9

Вычисляем усредненную дисперсию. Для этого делим сумму квадратов разностей между усредненными прогнозами и прогнозами для тестового набора на количество прогнозов.

```
# вычисляем усредненную дисперсию, делим сумму квадратов разностей
# между усредненными прогнозами и прогнозами для тестового набора
# на количество прогнозов
avg_var = np.sum((main_predictions - all_pred)**2) / all_pred.size
avg_var
```

	main_predictions			all_pred		
665.037037037037	17.	66.7	66.7	17	89	89
				17	22	22
				17	89	89

Давайте проверим, совпадают ли результаты, вычисленные вручную, с результатами, вычисленными с помощью функции `bias_variance_decomp()`.

```
# давайте проверим, совпадают ли результаты, вычисленные вручную, с
# результатами, вычисленными с помощью функции bias_variance_decomp()
# вычисляем усредненное ожидаемое значение функции потерь,
# усредненное смещение, усредненную дисперсию
avg_expected_loss, avg_bias, avg_var = bias_variance_decomp(
    tree, X_train, y_train, X_test, y_test, num_rounds=3,
    random_seed=123)
# печатаем результаты
print(ttl % avg_expected_loss)
print("Усредненное смещение: %.3f" % avg_bias)
print("Усредненная дисперсия: %.3f" % avg_var)
```

Усредненное ожидаемое значение функции потерь: 1543.504  
 Усредненное смещение: 878.467  
 Усредненная дисперсия: 665.037

## 7.4. ОБРАБОТКА ПРОПУСКОВ С ПОМОЩЬЮ КЛАССОВ MISSINGINDICATOR И SIMPLEIMPUTER

В прошлом примере данные, предоставленные компанией StateFarm, не содержали пропущенных значений, однако на практике практически всегда наборы данных содержат пропуски. Давайте разберем полезные классы `MissingIndicator` и `SimpleImputer`, помогающие обработать пропуски.

Класс `MissingIndicator` создает индикатор пропущенных значений. Пропуск переменной заменяется значением `True`, а непропущенное значение – значением `False`.

```
from sklearn.impute import MissingIndicator(missing_values=nan,
                                           number,
                                           string,
                                           None,
                                           strategy='missing-only',
                                           'all')
```

} **Заполнитель пропусков**  
 По умолчанию np.NaN, можно задать число, строку или значение None

} **Создание булевой маски**  
 Либо только для переменных с пропусками, либо для всех переменных

Давайте загрузим данные, сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# импортируем библиотеки pandas и numpy, функцию train_test_split()
# и классы MissingIndicator и SimpleImputer
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import MissingIndicator, SimpleImputer
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Verizon_missing.csv', sep=';')
# выведем первые 3 наблюдения
data.head(3)
```

	longdist	internat	local	age	income	billtype	pay	churn
0	16.0	0.0	5.0	46.0	34805.5	Бюджетный	CC	0
1	0.0	0.0	5.0	59.0	60111.8	NaN	NaN	1
2	13.0	0.0	NaN	NaN	13126.9	Бюджетный	Auto	0

Исходный набор содержит небольшую часть данных американского провайдера Verizon. Он представляет собой записи о 144 клиентах, классифицированных на два класса: 0 – оттока нет (78 клиентов) и 1 – отток есть (66 клиентов). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- количественный признак *Длительность междугородных звонков* [longdist];
- количественный признак *Длительность международных звонков* [internat];
- количественный признак *Длительность местных звонков* [local];
- количественный признак *Возраст* [age];
- категориальный признак *Ежемесячный доход* [income];
- категориальный признак *Тип тарифа* [billtype];
- категориальный признак *Способ оплаты* [pay];
- бинарная зависимая переменная *Факт оттока клиента* [churn].

Давайте разобьем набор на обучающую и тестовую выборки и взглянем на них.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
train, test, y_train, y_test = train_test_split(
    data.drop('churn', axis=1),
    data['churn'],
    test_size=.3,
    stratify=data['churn'],
    random_state=100)
```

Посмотрим на обучающий массив признаков.

```
# взглянем на обучающий массив признаков
train.head(10)
```

	longdist	internat	local	age	income	billtype	pay
18	19.0	NaN	11.0	88.0	66906.60	Бесплатный	CH
19	19.0	NaN	96.0	79.0	37571.10	Бесплатный	CC
66	0.0	0.0	1.0	94.0	13946.80	Бюджетный	NaN
65	9.0	6.0	12.0	93.0	46771.20	Бюджетный	NaN
74	26.0	0.0	NaN	32.0	NaN	Бюджетный	Auto
96	9.0	0.0	NaN	25.0	52366.40	Бюджетный	CC
47	14.0	0.0	48.0	58.0	4123.46	Бесплатный	CC
107	0.0	0.0	3.0	68.0	14955.80	Бесплатный	CH
11	1.0	0.0	9.0	44.0	11861.40	Бюджетный	CD
86	NaN	4.0	13.0	54.0	27376.00	NaN	CC

Посмотрим на тестовый массив признаков.

```
# взглянем на тестовый массив признаков
test.head(10)
```

	longdist	internat	local	age	income	billtype	pay
44	3.0	0.0	9.0	45.0	70239.8	Бесплатный	CC
59	21.0	NaN	NaN	33.0	60170.1	NaN	CH
51	29.0	NaN	110.0	29.0	18861.6	Бюджетный	CC
35	12.0	0.0	55.0	62.0	13965.4	Бесплатный	CH
55	25.0	0.0	77.0	95.0	NaN	NaN	Auto
121	14.0	NaN	167.0	83.0	NaN	Бесплатный	CC
91	19.0	NaN	53.0	30.0	17866.9	Бюджетный	Auto
30	20.0	0.0	29.0	58.0	91446.4	Бюджетный	CC
23	2.0	0.0	99.0	29.0	49127.2	Бесплатный	CC
53	0.0	0.0	1.0	75.0	NaN	Бесплатный	CC

Видим, что массивы содержат пропуски. Также о пропусках можно судить, воспользовавшись методом `.info()` объекта `DataFrame` или цепочкой методов `.isnull()` и `.sum()`. Кроме того, если целочисленная переменная (возраст, количество просрочек) записана как вещественная, это указывает на наличие пропусков (ведь для хранения количественной переменной с пропусками используется тип `float`).

```
# выясняем, есть ли пропуски
print(train.isnull().sum())
print('')
print(test.isnull().sum())
```

```
longdist    5
internat    23
local       12
age         8
income     18
```

```
billtype    13
pay         13
dtype: int64

longdist     3
internat    11
local        2
age          0
income      11
billtype     11
pay          2
dtype: int64
```

Давайте создадим экземпляр класса `MissingIndicator` и обучим модель импутации. В данном случае метод `.fit()` вычисляет единственный параметр модели – булеву маску для пропусков переменной `local` в обучающем массиве признаков.

```
# создаем экземпляр класса MissingIndicator
miss_ind = MissingIndicator()
# обучаем модель импутации – создаем индикатор пропусков
# переменной local в обучающем массиве
miss_ind.fit(train[['local']])
```

Большинство классов библиотеки `scikit-learn` работают с 2D-массивами, поэтому когда работаем с серией, не забываем ее преобразовать в соответствующий массив (используем двойные квадратные скобки).

Теперь применяем модель с помощью метода `.transform()`. Фактически метод `.transform()` создает индикатор пропусков переменной `local` в обучающем и тестовом массивах признаков.

```
# применяем модель импутации к переменной local
# в обучающем массиве признаков:
# создаем индикатор пропусков переменной local
# в обучающем массиве
train['miss_ind_local'] = miss_ind.transform(train[['local']])
# применяем модель импутации к переменной local
# в тестовом массиве признаков:
# создаем индикатор пропусков переменной local
# в тестовом массиве
test['miss_ind_local'] = miss_ind.transform(test[['local']])
train.head()
```

	longdist	internat	local	age	income	billtype	pay	miss_ind_local
<b>18</b>	19.0	NaN	11.0	88.0	66906.6	Бесплатный	CH	False
<b>19</b>	19.0	NaN	96.0	79.0	37571.1	Бесплатный	CC	False
<b>66</b>	0.0	0.0	1.0	94.0	13946.8	Бюджетный	NaN	False
<b>65</b>	9.0	6.0	12.0	93.0	46771.2	Бюджетный	NaN	False
<b>74</b>	26.0	0.0	NaN	32.0	NaN	Бюджетный	Auto	True

Давайте вычислим медиану переменной *local* в обучающем массиве признаков. Мы ее применим для импутации пропусков переменной *local*, воспользовавшись удобным классом `SimpleImputer`.

```
# взглянем на медиану переменной local
# в обучающем массиве признаков
train['local'].median()
```

32.5

Класс `SimpleImputer` осуществляет замену пропущенных значений с помощью статистик – среднего, медианы, моды и константного значения. Среднее и медиану можно задать только для количественных переменных. Моду и константное значение можно применить как к количественным, так и к категориальным переменным (для количественных переменных константой будет 0, для категориальных переменных константой будет строковое значение `'missing_value'`).

```
from sklearn.impute import SimpleImputer(missing_values=nan,
                                         number,
                                         string,
                                         None,
                                         strategy='mean',
                                         'median',
                                         'most_frequent',
                                         'constant',
                                         fill_value=None,
                                         copy=True)
```

**Заполнитель пропусков**  
По умолчанию np.NaN, можно задать число, строку или значение None

**Стратегия импутации**  
По умолчанию используется 'mean'. Значения 'mean' и 'median' – для количественных переменных, 'most\_frequent' и 'constant' – для строковых и количественных переменных. Если выбрано 'constant', нужно задать значение настройки fill\_value

**Значение для импутации, если задано strategy='constant'**  
Если оставить без изменений, fill\_value будет 0 при импутации количественных переменных и 'missing\_value' для категориальных переменных (переменных типов str и object)

Если задано True, всегда создается копия X. Если задано False, импутация будет выполнена на месте в тех случаях, когда это возможно. Обратите внимание, что в следующих случаях всегда будет создана новая копия, даже если задано copy=False:

- если X не является массивом значений с плавающей точкой;
- если X закодирован как CSR-матрица;
- если add\_indicator=True

Давайте создадим экземпляр класса `SimpleImputer` и обучим модель импутации. В данном случае метод `.fit()` вычисляет единственный параметр модели – медиану переменной *local* в обучающем массиве признаков.

```
# создаем экземпляр класса SimpleImputer
simp = SimpleImputer(strategy='median')
# обучаем модель импутации - вычисляем медиану
# переменной local в обучающем массиве
simp.fit(train[['local']]);
```

Теперь применяем модель с помощью метода `.transform()`. Фактически метод `.transform()` заменяет пропуски переменной *local* в обучающем и тестовом массивах признаков медианой переменной *local* в обучающем массиве признаков.

```
# применяем модель импутации к переменной local
# в обучающем массиве признаков:
# заменяем пропуски переменной в обучающем массиве признаков
# медианой переменной, вычисленной по ОБУЧАЮЩЕМУ массиву признаков
```

```

train['local'] = simp.transform(train[['local']])
# еще можно так
# train['local'] = simp.transform(train['local'].values.reshape(-1, 1))

# применяем модель импутации к переменной local
# в тестовом массиве признаков:
# заменяем пропуски переменной в тестовом массиве признаков
# медианой переменной, вычисленной по ОБУЧАЮЩЕМУ массиву признаков
test['local'] = simp.transform(test[['local']])

```

Взглянем на обучающий и тестовый массивы признаков.

```

# взглянем на обучающий массив признаков
train.head(10)

```

	longdist	internat	local	age	income	billtype	pay	miss_ind_local
18	19.0	NaN	11.0	88.0	66906.60	Бесплатный	CH	False
19	19.0	NaN	96.0	79.0	37571.10	Бесплатный	CC	False
66	0.0	0.0	1.0	94.0	13946.80	Бюджетный	NaN	False
65	9.0	6.0	12.0	93.0	46771.20	Бюджетный	NaN	False
74	26.0	0.0	32.5	32.0	NaN	Бюджетный	Auto	True
96	9.0	0.0	32.5	25.0	52366.40	Бюджетный	CC	True
47	14.0	0.0	48.0	58.0	4123.46	Бесплатный	CC	False
107	0.0	0.0	3.0	68.0	14955.80	Бесплатный	CH	False
11	1.0	0.0	9.0	44.0	11861.40	Бюджетный	CD	False
86	NaN	4.0	13.0	54.0	27376.00	NaN	CC	False

```

# взглянем на тестовый массив признаков
test.head(10)

```

	longdist	internat	local	age	income	billtype	pay	miss_ind_local
44	3.0	0.0	9.0	45.0	70239.8	Бесплатный	CC	False
59	21.0	NaN	32.5	33.0	60170.1	NaN	CH	True
51	29.0	NaN	110.0	29.0	18861.6	Бюджетный	CC	False
35	12.0	0.0	55.0	62.0	13965.4	Бесплатный	CH	False
55	25.0	0.0	77.0	95.0	NaN	NaN	Auto	False
121	14.0	NaN	167.0	83.0	NaN	Бесплатный	CC	False
91	19.0	NaN	53.0	30.0	17866.9	Бюджетный	Auto	False
30	20.0	0.0	29.0	58.0	91446.4	Бюджетный	CC	False
23	2.0	0.0	99.0	29.0	49127.2	Бесплатный	CC	False
53	0.0	0.0	1.0	75.0	NaN	Бесплатный	CC	False

Видим, что пропуски были заменены на медианы (выделены красными рамками).

Теперь применим `SimpleImputer` к отдельным спискам переменных. Пропуски в оставшихся количественных переменных заменим медианами, а пропуски в категориальных переменных – модами.

```
# создаем список количественных переменных
numeric_cols = ['longdist', 'internat', 'age', 'income']
# обучаем модель импутации - вычисляем медианы переменных
# longdist, internat, age и income в обучающем массиве признаков
simp.fit(train[numeric_cols])
# применяем модель импутации к указанным переменным
# в обучающем массиве признаков:
# заменяем пропуски каждой переменной в обучающем
# массиве признаков медианой соответствующей переменной,
# вычисленной по ОБУЧАЮЩЕМУ массиву признаков
train[numeric_cols] = simp.transform(train[numeric_cols])
# применяем модель импутации к указанным переменным
# в тестовом массиве признаков:
# заменяем пропуски каждой переменной в тестовом
# массиве признаков медианой соответствующей переменной,
# вычисленной по ОБУЧАЮЩЕМУ массиву признаков
test[numeric_cols] = simp.transform(test[numeric_cols])

# создаем список категориальных переменных
cat_cols = ['billtype', 'pay']
# создаем экземпляр класса SimpleImputer
simp2 = SimpleImputer(strategy='most_frequent')
# обучаем модель импутации - вычисляем моды переменных
# billtype и pay в обучающем массиве признаков
simp2.fit(train[cat_cols])
# применяем модель импутации к указанным переменным
# в обучающем массиве признаков:
# заменяем пропуски каждой переменной в обучающем
# массиве признаков модой соответствующей переменной,
# вычисленной по ОБУЧАЮЩЕМУ массиву признаков
train[cat_cols] = simp2.transform(train[cat_cols])
# применяем модель импутации к указанным переменным
# в тестовом массиве признаков:
# заменяем пропуски каждой переменной в тестовом
# массиве признаков модой соответствующей переменной,
# вычисленной по ОБУЧАЮЩЕМУ массиву признаков
test[cat_cols] = simp2.transform(test[cat_cols])
```

Взглянем на результат и убедимся, что пропуски в указанных переменных импутированы.

```
# взглянем на обучающий массив признаков
train.head(10)
```

	longdist	internat	local	age	income	billtype	pay	miss_ind_local
18	19.0	0.0	11.0	88.0	66906.60	Бесплатный	CH	False
19	19.0	0.0	96.0	79.0	37571.10	Бесплатный	CC	False
66	0.0	0.0	1.0	94.0	13946.80	Бюджетный	CC	False
65	9.0	6.0	12.0	93.0	46771.20	Бюджетный	CC	False
74	26.0	0.0	32.5	32.0	41031.40	Бюджетный	Auto	True
96	9.0	0.0	32.5	25.0	52366.40	Бюджетный	CC	True
47	14.0	0.0	48.0	58.0	4123.46	Бесплатный	CC	False
107	0.0	0.0	3.0	68.0	14955.80	Бесплатный	CH	False
11	1.0	0.0	9.0	44.0	11861.40	Бюджетный	CD	False
86	17.0	4.0	13.0	54.0	27376.00	Бюджетный	CC	False

## 7.5. ВЫПОЛНЕНИЕ ДАММИ-КОДИРОВАНИЯ С ПОМОЩЬЮ КЛАССА ONEHOTENCODER И ФУНКЦИИ GET\_DUMMIES(), ЗНАКОМСТВО С РАЗРЕЖЕННЫМИ МАТРИЦАМИ

Класс `OneHotEncoder` и функция `get_dummies()` осуществляют дамми-кодирование, необходимое для линейных моделей при обработке категориальных признаков. Линейные модели не умеют обрабатывать категориальные признаки «как есть». Каждый уровень категориальной переменной становится отдельным бинарным столбцом со значениями 0 или 1.

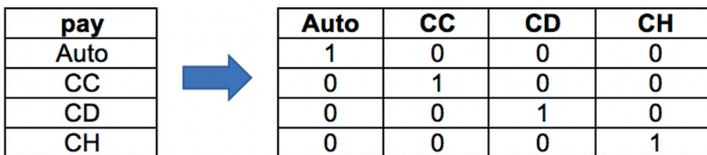


Рис. 31 Дамми-кодирование

Начнем с класса `OneHotEncoder`. Метод `.fit()` определяет схему кодировки категориального признака, а метод `.transform()` применяет эту схему, выполняет дамми-кодирование. Ниже разобраны основные параметры класса `OneHotEncoder`.

```

from sklearn.preprocessing import OneHotEncoder(drop=None,
                                                'first'
                                                'if_binary')

```

Задаёт способ удаления одной из категорий признака. Полезно в ситуациях, когда идеально скоррелированные признаки вызывают проблемы, например, при вводе их в модель линейной регрессии без регуляризации. Однако отбрасывание одной категории нарушает симметрию исходного представления и, следовательно, может вызвать смещение в некоторых моделях, например, в моделях линейной классификации или регрессии с регуляризацией. По умолчанию сохраняет все категории. Если задано значение 'first', удаляется первая категория в каждом признаке, при этом если у признака – одна категория, то такой признак полностью удаляется. Если задано значение 'if\_binary', удаляется первая категория в каждом признаке с двумя категориями, признаки с одной категорией и более чем двумя категориями не удаляются

Если задано значение True, возвращает разреженную матрицу, в противном случае возвращает плотный массив

По умолчанию выдает ошибку, если в ходе преобразования встречается новую категорию. Если задать 'ignore', то все дамми-переменные для этой категории будут представлять собой нули. Если задать 'infrequent\_if\_exist', то когда во время преобразования встретится неизвестная категория, дамми-переменные для этого признака будут сопоставлены с редкой категорией, если она существует. Редкая категория будет сопоставлена с последней позицией кодировки. Во время обратного преобразования неизвестная категория будет сопоставлена с категорией, обозначенной как «редкая», если она существует. Если редкая категория не существует, то методы .transform() и .inverse\_transform() будут обрабатывать неизвестную категорию, как при handle\_unknown='ignore'. Редкие категории определяются согласно min\_frequency и max\_categories

Минимальная частота, ниже которой категория объявляется редкой. Если задано целочисленное значение, категории с меньшей частотой будут считаться редкими. Если задано число с плавающей точкой, категории с меньшей частотой, чем min\_frequency \* n\_samples, будут считаться редкими

Задаёт верхнее предельное количества выходных столбцов для каждого входного столбца при рассмотрении редких категорий

min\_frequency=None,
max\_categories=None)

Наиболее важным параметром класса `OneHotEncoder` является параметр `drop`, который задает способ удаления одной из категорий признака. По умолчанию все категории сохраняются. Если для параметра `drop` задано значение 'first', удаляется первая категория. Это может быть полезно в ситуациях, когда идеально скоррелированные признаки вызывают проблемы, например при вводе их в модель линейной регрессии без регуляризации. Однако отбрасывание одной категории нарушает симметрию исходного представления и, следовательно, может вызвать смещение в некоторых моделях, например в моделях линейной классификации или регрессии с регуляризацией. По умолчанию параметр `drop` сохраняет все категории.

Параметр `handle_unknown` задает способ обработки новой категории в новых данных: 'error' – выдает ошибку (по умолчанию), 'ignore' – игнорирует (дамми-переменные для новой категории будут закодированы нулями), 'infrequent\_if\_exist' – приравнивает к редкой категории (дамми-переменные для новой категории будут закодированы так же, как для редкой категории), редкая категория (infrequent category) определяется параметрами `min_frequency` и `max_categories`.

Давайте выполним дамми-кодирование переменной `pay`, при этом будем игнорировать появление неизвестной категории в новых данных.

```

# импортируем класс OneHotEncoder
from sklearn.preprocessing import OneHotEncoder
# создаем экземпляр класса OneHotEncoder
ohe = OneHotEncoder(sparse=False, handle_unknown='ignore')
# копируем переменную pay в обучающем массиве признаков
ohe_train = train[['pay']].copy()
# обучаем модель дамми-кодирования - определяем дамми для переменной
# pay в обучающем массиве признаков
ohe.fit(ohe_train)
# выполняем дамми-кодирование переменной
# pay в обучающем массиве признаков
ohe_train_transformed = ohe.transform(ohe_train)
# смотрим первые 10 наблюдений
ohe_train_transformed[:10]

```

```
array([[0., 0., 0., 1.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.],
       [0., 1., 0., 0.]])
```

Теперь применим `OneHotEncoder`, когда в тестовом массиве признаков у переменной `rau` появилась новая категория.

```
# копируем переменную rau в тестовом массиве признаков
ohe_test = test[['rau']].copy()
# заменяем значение в переменной rau
ohe_test.iloc[0, 0] = 'new_category'
# выводим первые три наблюдения
ohe_test.head(3)
```

	<b>rau</b>
<b>44</b>	new_category
<b>59</b>	CH
<b>51</b>	CC

```
# выполняем дамми-кодирование переменной
# rau в тестовом массиве признаков
ohe_test_transformed = ohe.transform(ohe_test)
# смотрим первые три наблюдения
ohe_test_transformed[:3]
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 1., 0., 0.]])
```

Видим, что все дамми-переменные для новой категории закодированы нулями.

Теперь создадим обучающий датафрейм с одним столбцом `City`.

```
# создаем обучающий датафрейм с одним столбцом
train = pd.DataFrame(
    {'City': ['MSK', 'MSK', 'MSK', 'SPB',
             'EKB', 'EKB', 'EKB',
             'EKB', 'EKB']})
train
```

City	
0	MSK
1	MSK
2	MSK
3	SPB
4	EKB
5	EKB
6	EKB
7	EKB
8	EKB

Взглянем на частоты категорий переменной *City*.

```
# смотрим частоты категорий City
train['City'].value_counts()
```

```
EKB    5
MSK    3
SPB    1
Name: City, dtype: int64
```

Создаем класс `OneHotEncoder`, задав с помощью параметра `min_frequency` порог для редких категорий, равный 3, т.е. категории с абсолютными частотами менее 3 будут объявлены редкими, и обучаем. Дополнительно с помощью атрибута `infrequent_categories_` выведем обнаруженные редкие категории.

```
# создаем класс OneHotEncoder, задав порог
# для редких категорий, и обучаем
ohe = OneHotEncoder(
    min_frequency=3,
    sparse=False,
    handle_unknown='infrequent_if_exist')
ohe.fit(train)
ohe.infrequent_categories_

[array(['SPB'], dtype=object)]
```

Видим, что редкой категорией является 'SPB', потому что ее частота меньше 3. Теперь выполним дамми-кодирование переменной *City* в обучающем наборе.

```
# выполняем дамми-кодирование
# в обучающем датафрейме
ohe.transform(train)
```

```
array([[0., 1., 0.],
       [0., 1., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

```
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.],
[1., 0., 0.]])
```

Таким образом, для новой категории в новых данных к дамми-переменным будет применена та же схема кодировки, что и для категории 'SPB' (выделена красной рамкой).

Теперь создадим тестовый датафрейм с одним столбцом *City*, у которого помимо знакомых категорий 'SPB', 'MSK' и 'EKB' будет одна новая категория 'NSK'.

```
# создаем обучающий датафрейм с одним столбцом,
# в котором будет новая категория 'NSK'
test = pd.DataFrame(
    {'City': ['NSK', 'MSK', 'NSK', 'MSK',
             'SPB', 'EKB', 'SPB',
             'EKB', 'SPB']})
```

```
test
```

	City
0	NSK
1	MSK
2	NSK
3	MSK
4	SPB
5	EKB
6	SPB
7	EKB
8	SPB

Теперь выполним дамми-кодирование переменной *City* в тестовом наборе.

```
# выполняем дамми-кодирование
# в тестовом датафрейме
ohe.transform(test)
```

```
array([[0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.]])
```

Видим, что дамми-переменные для новой категории 'NSK' в новых данных (тестовом датафрейме) кодируются так же, как для категории 'SPB'.

Типичное применение класса `OneHotEncoder` – в качестве звена в цепочке преобразований внутри конвейера в связке с классами `FunctionTransformer`, `ColumnTransformer` и `Pipeline`. Класс `OneHotEncoder` редко используется для непосредственного дамми-кодирования объектов `Series` и `DataFrame` библиотеки `pandas`. Обычно для этих целей применяется функция `get_dummies()` библиотеки `pandas`.

Функция `get_dummies()` автоматически преобразует заданную категориальную переменную в набор дамми-переменных.

Параметр `drop_first` этой функции задает тип дамми-кодирования. По умолчанию для этого параметра задано значение `False` и выполняется дамми-кодирование по методу неполного ранга, в противном случае будет выполнено дамми-кодирование по методу полного ранга (первая категория объявляется опорной, это необходимо для устранения линейной зависимости столбцов при применении линейных моделей без регуляризации). Пропуски в переменной кодируются нулями во всех дамми-переменных, созданных для данной переменной. С помощью параметра `columns` можно задать список имен столбцов, которые нужно подвергнуть дамми-кодированию. С помощью параметра `dummy_na` можно добавить столбец – индикатор значений `NaN`.

```
# применяем функцию get_dummies()
train = pd.get_dummies(train)
train.head()
```

	longdist	internet	local	age	income	miss_ind_local	billtype_Бесплатный	billtype_Бюджетный	pay_Auto	pay_CC	pay_CD	pay_CH
18	19.0	0.0	11.0	88.0	66906.6	False	1	0	0	0	0	1
19	19.0	0.0	96.0	79.0	37571.1	False	1	0	0	1	0	0
66	0.0	0.0	1.0	94.0	13946.8	False	0	1	0	1	0	0
65	9.0	6.0	12.0	93.0	46771.2	False	0	1	0	1	0	0
74	26.0	0.0	32.5	32.0	41031.4	True	0	1	1	0	0	0

Вовремя избавляйтесь от редких категорий, потому что они могут стать причиной несовпадения количества дамми-переменных в обучающей и тестовой выборках.

У класса `OneHotEncoder` есть параметр `sparse`, возвращающий разреженную матрицу, у функции `get_dummies()` тоже есть параметр `sparse`, возвращающий массив, который используется для хранения разреженных данных. Давайте выясним, что из себя представляет разреженная матрица.

*Разреженной* называют матрицу, состоящую преимущественно из нулевых значений. *Плотной* матрицей называют матрицу, состоящую преимущественно из ненулевых значений. Разреженность можно вычислить, поделив количество нулевых значений на общее количество элементов. В прикладном машинном обучении часто приходится работать с большими разреженными матрицами. Здесь можно привести примеры получения таких матриц, когда:

- мы представляем категориальную переменную в виде бинарных признаков, строя линейную модель;
- в ходе построения рекомендательной системы фиксируем факт просмотра зрителем конкретного фильма в каталоге фильмов, факт покупки клиентом товара в каталоге товаров;

- подсчитываем встречаемость слов в корпусе документов при естественной обработке языка.

Правильная обработка разреженных матриц может привести к сокращению вычислительных затрат.

Для хранения разреженных матриц можно использовать следующие структуры данных.

**Словарь по ключам (dictionary of keys или DOK)** – словарь, в котором ключами являются кортежи индексов строк и столбцов (row, column), а значениями – соответствующие значения. Элементы, отсутствующие в словаре, принимаются за нулевые.

**Список списков (list of lists или LIL)** – каждая строка матрицы хранится как список, каждый подсписок содержит индекс столбца и значение.

**Список координат (coordinate list или COO)** – список кортежей, в котором каждый кортеж хранит индекс строки, индекс столбца и значение (row, column, value).

Кроме того, существуют еще две структуры данных, которые на практике чаще используются для эффективной работы с разреженными матрицами:

- **сжатое хранение строкой (CSR – compressed sparse row, CRS – compressed row storage, Йельский формат);**
- **сжатое хранение столбцом (CSC – compressed sparse column, CCS – compressed column storage).**

При сжатом хранении строкой мы представляем матрицу  $M^{n \times m}$ , содержащую  $N_{NONZERO}$  ненулевых значений, в виде трех одномерных массивов:

- массив значений – массив размера  $N_{NONZERO}$ , в котором хранятся ненулевые значения, взятые подряд из первой непустой строки, затем идут значения из следующей непустой строки и т. д.;
- массив индексов столбцов – массив размера  $N_{NONZERO}$ , который хранит индексы столбцов для соответствующих элементов из массива значений;
- массив индексации строк – массив размера  $n + 1$  (количество строк + 1), который для индекса  $i$  хранит количество ненулевых элементов в строках до  $i - 1$  включительно, при этом последний элемент массива совпадает с  $N_{NONZERO}$ , а первый всегда равен 0, выполняя роль запирающего элемента.

Пусть  $M = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 4 & 0 \\ 0 & 2 & 6 \end{pmatrix}$ , тогда массив значений = (1, 2, 4, 2, 6), массив индексов

столбцов = (0, 1, 1, 1, 2), массив индексов строк = (0, 2, 3, 5).

Как был сформирован массив значений? Мы переписали ненулевые значения, взятые подряд из первой непустой строки, затем ненулевые значения из второй непустой строки, потом ненулевые значения из третьей непустой строки.

Как был сформирован массив индексов столбцов? Мы записали индексы столбцов для ненулевых значений.

Как был сформирован массив индексов строк? Записываем ноль и подсчитываем накопленное количество ненулевых значений в каждой строке. В пер-

вой строке только два ненулевых значения, поэтому записываем 2, во второй строке – одно ненулевое значение, поэтому записываем  $2 + 1 = 3$ , в третьей строке – два ненулевых значения, поэтому записываем  $3 + 2 = 5$ .

Сжатое хранение столбцом похоже на CRS, только строки и столбцы меняются ролями – значения храним по столбцам, по второму массиву можем определить строку, после подсчётов с третьим массивом узнаём столбцы.

Разумеется, могут быть модификации данных форматов.

Сейчас мы определим разреженную матрицу  $4 \times 4$  как плотный массив NumPy, преобразуем ее в формат CRS и потом обратно преобразуем в плотный массив с помощью функции `todense()`.

```
# импортируем функцию csr_matrix()
from scipy.sparse import csr_matrix

# создаем массив из 16 элементов, 4 строки и 4 столбца
A = np.array([[0, 0, 0, 0],
              [5, 8, 0, 0],
              [0, 0, 3, 0],
              [0, 6, 0, 0]])

# печатаем массив
print(A)
# преобразовываем в CRS-представление
print('')
S = csr_matrix(A)
# печатаем CRS-представление
print(S)
# преобразовываем обратно в плотный массив
D = S.todense()
# печатаем плотный массив
print('')
print(D)
```

```
[[0 0 0 0]
 [5 8 0 0]
 [0 0 3 0]
 [0 6 0 0]]
```

```
(1, 0) → 5
(1, 1) → 8
(2, 2) → 3
(3, 1) → 6
```

(1, 0)	5	Массив ненулевых значений
(1, 1)	8	
(2, 2)	3	Массив индексов строк и столбцов для ненулевых значений
(3, 1)	6	

```
[[0 0 0 0]
 [5 8 0 0]
 [0 0 3 0]
 [0 6 0 0]]
```

В заключение проиллюстрируем, как можно ускорить вычисления за счет использования формата для разреженных данных. Загружаем данные с соревнования [Categorical Feature Encoding Challenge II](https://www.kaggle.com/competitions/cat-in-the-dat-ii/overview) на Kaggle <https://www.kaggle.com/competitions/cat-in-the-dat-ii/overview>. Он содержит 600 000 наблюдений и 25 переменных. Метрика качества – AUC-ROC.

```
# загружаем и смотрим данные
```

```
data = pd.read_csv('Data/catfeatures_challenge_II_train.csv')
data.head()
```

id	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3	...	nom_9	ord_0	ord_1	ord_2	ord_3	ord_4	ord_5	day	month	target
0	0.0	0.0	0.0	F	N	Red	Trapezoid	Hamster	Russia	...	02e7c8990	3.0	Contributor	Hot	c	U	Pw	6.0	3.0	0
1	1.0	1.0	0.0	F	Y	Red	Star	Axolotl	NaN	...	f37df64af	3.0	Grandmaster	Warm	e	X	pE	7.0	7.0	0
2	0.0	1.0	0.0	F	N	Red	NaN	Hamster	Canada	...	NaN	3.0	NaN	Freezing	n	P	eN	5.0	9.0	0
3	NaN	0.0	0.0	F	N	Red	Circle	Hamster	Finland	...	f9d456e57	1.0	Novice	Lava Hot	a	C	NaN	3.0	3.0	0
4	0.0	NaN	0.0	T	N	Red	Triangle	Hamster	Costa Rica	...	c5361037c	3.0	Grandmaster	Cold	h	C	OZ	5.0	12.0	0

Зависимой переменной является переменная *target*, бинарные признаки помечены префиксом *bin\_*, номинальные – префиксом *nom\_*, порядковые – префиксом *ord\_*, также есть два циклических признака времени *day* и *month*.

Избавляемся от редких категорий в некоторых переменных. Все категории с частотой 100 наблюдений и менее запишем в отдельную категорию OTHER.

```
# избавляемся от редких категорий
```

```
for col in ['nom_5', 'nom_6', 'nom_7', 'nom_8', 'nom_9']:
    abs_freq = data[col].value_counts(dropna=False)
    data[col] = np.where(
        data[col].isin(abs_freq[abs_freq >= 100].index.tolist()),
        data[col], 'Other')
```

Удаляем идентификатор, формируем массив меток и массив признаков, смотрим типы переменных и количество пропусков.

```
# удаляем идентификатор
```

```
data.drop('id', axis=1, inplace=True)
# формируем массив меток и массив признаков
labels = data.pop('target').values
# смотрим типы переменных и количество пропусков
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 600000 entries, 0 to 599999
Data columns (total 23 columns):
#   Column      Non-Null Count  Dtype
---  -
0   bin_0      582106 non-null  float64
1   bin_1      581997 non-null  float64
2   bin_2      582070 non-null  float64
3   bin_3      581986 non-null  object
4   bin_4      581953 non-null  object
5   nom_0      581748 non-null  object
6   nom_1      581844 non-null  object
7   nom_2      581965 non-null  object
8   nom_3      581879 non-null  object
9   nom_4      581965 non-null  object
10  nom_5      582222 non-null  object
11  nom_6      581869 non-null  object
12  nom_7      581997 non-null  object
13  nom_8      582245 non-null  object
14  nom_9      581927 non-null  object
```

```

15 ord_0 581712 non-null float64
16 ord_1 581959 non-null object
17 ord_2 581925 non-null object
18 ord_3 582084 non-null object
19 ord_4 582070 non-null object
20 ord_5 582287 non-null object
21 day 582048 non-null float64
22 month 582012 non-null float64
dtypes: float64(6), object(17)
memory usage: 105.3+ MB

```

Видим, что каждая переменная имеет пропуск.  
Создаем индикаторы пропусков для каждой переменной.

```

# для всех столбцов создаем индикаторы пропусков
for col in data.columns:
    data[col + '_isnan'] = np.where(data[col].isnull(), 'T', 'F')

```

Теперь создаем две новые переменные на основе переменной *ord5*, просто извлекая первый и второй символы в ее строковых значениях.

```

# создаем две новые переменные на основе переменной ord5,
# просто извлекая первый и второй символы
data['ord_5a'] = data['ord_5'].str[0]
data['ord_5b'] = data['ord_5'].str[1]

```

Формируем список столбцов.

```

# формируем список столбцов
columns = [col for col in data.columns]

```

Разбиваем набор на обучающую и тестовую выборки.

```

# разбиваем набор на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    data,
    labels,
    test_size=0.3,
    stratify=labels,
    random_state=42)

```

С помощью функции `get_dummies()` получим дамми-переменные. Мы попробуем создать массивы с использованием и без использования параметра `sparse`. С помощью значения `True` параметра `drop_first` первую категорию каждой переменной объявляем опорной.

```

# создаем дамми-переменные, не используем
# параметр sparse
X_tr_non_sparse = pd.get_dummies(
    X_train,
    columns=columns,
    drop_first=True,
    sparse=False)
X_tst_non_sparse = pd.get_dummies(

```

```

X_test,
columns=columns,
drop_first=True,
sparse=False)

# создаем дамми-переменные, используем
# параметр sparse
X_tr_sparse = pd.get_dummies(
    X_train,
    columns=columns,
    drop_first=True,
    sparse=True)
X_tst_sparse = pd.get_dummies(
    X_test,
    columns=columns,
    drop_first=True,
    sparse=True)

# смотрим формы массивов
print('non_sparse:', X_tr_non_sparse.shape, X_tst_non_sparse.shape)
print('sparse:', X_tr_sparse.shape, X_tst_sparse.shape)

non_sparse: (420000, 5026) (180000, 5026)
sparse: (420000, 5026) (180000, 5026)

```

Итак, у нас каждый массив содержит по 5026 признаков.

Теперь импортируем функцию `roc_auc_score()` для вычисления AUC-ROC и класс `LogisticRegression`.

```

# импортируем функцию roc_auc_score() для
# вычисления AUC-ROC
from sklearn.metrics import roc_auc_score
# импортируем класс LogisticRegression
from sklearn.linear_model import LogisticRegression

```

Теперь обучаем модель логистической регрессии на массивах, созданных с использованием и без использования параметра `sparse`.

```

%%time

# обучаем модель логистической регрессии
# на массиве признаков, созданном с
# с помощью параметра sparse
logreg = LogisticRegression(solver='liblinear').fit(
    X_tr_sparse, y_train)
print("AUC на обучающей выборке: {:.3f}".format(
    roc_auc_score(y_train, logreg.predict_proba(
        X_tr_sparse[:, 1])))
print("AUC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, logreg.predict_proba(
        X_tst_sparse[:, 1])))

AUC на обучающей выборке: 0.801
AUC на тестовой выборке: 0.787
CPU times: user 11.9 s, sys: 125 ms, total: 12 s

```

Wall time: 12.1 s

```
%%time
```

```
# обучаем модель логистической регрессии
# на массиве признаков, созданном с
# без помощи параметра sparse
logreg = LogisticRegression(solver='liblinear').fit(
    X_tr_non_sparse, y_train)
print("AUC на обучающей выборке: {:.3f}".format(
    roc_auc_score(y_train, logreg.predict_proba(
        X_tr_non_sparse)[: , 1])))
print("AUC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, logreg.predict_proba(
        X_tst_non_sparse)[: , 1])))
```

AUC на обучающей выборке: 0.801

AUC на тестовой выборке: 0.787

CPU times: user 1min 39s, sys: 24.7 s, total: 2min 4s

Wall time: 1min 55s

Обучение на данных в формате разреженных матриц заняло 12 секунд, а обучение на обычных данных заняло уже 2 минуты.

## 7.6. АВТОМАТИЧЕСКОЕ ПОСТРОЕНИЕ КОНВЕЙЕРОВ МОДЕЛЕЙ С ПОМОЩЬЮ КЛАССА PIPELINE

Ранее мы говорили, что на практике строим не одну модель, а конвейер нескольких моделей. Класс `Pipeline` позволяет автоматизировать построение конвейеров.

Он позволяет объединить модели предварительной обработки (например, модели импутации и стандартизации данных) с моделью машинного обучения типа логистической регрессии. Класс `Pipeline` предусматривает методы `.fit()`, `.predict()` и `.score()` и имеет все те же свойства, что и любая модель машинного обучения `scikit-learn`. Конвейер представляет собой список этапов – двухэлементных кортежей. Первый элемент кортежа – имя этапа (любая строка на ваш выбор, за одним исключением: имя не должно содержать символ двойного подчеркивания `_`). Второй элемент кортежа – экземпляр класса.

```
pipe = Pipeline([('imputer', SimpleImputer()), ('forest', RandomForestClassifier())])
```

The diagram illustrates the structure of a Pipeline. It shows a list of two steps. The first step is labeled '1-й этап (2-элементный кортеж)' and consists of the string 'imputer' and the class instance SimpleImputer(). The second step is labeled '2-й этап (2-элементный кортеж)' and consists of the string 'forest' and the class instance RandomForestClassifier(). Below the first step, two arrows point upwards: one from the label 'название этапа' pointing to 'imputer', and another from the label 'экземпляр класса' pointing to SimpleImputer().

Рис. 32 Схема конвейера

Здесь мы создали два этапа: первый этап, названный 'imputer', является экземпляром класса `SimpleImputer`, а второй, названный 'forest', является экземпляром класса `RandomForestClassifier`.

Класс `Pipeline` не ограничивается предварительной обработкой и классификацией, с его помощью можно объединить любое количество моделей. Например, можно создать конвейер, включающий в себя выделение признаков, отбор признаков, масштабирование и классификацию, в общей сложности четыре этапа. Кроме того, последним этапом вместо классификации может быть регрессия или кластеризация.

Единственное требование, предъявляемое к моделям в конвейере, заключается в том, что все этапы, кроме последнего, должны использовать метод `.transform()`, таким образом, они позволяют сгенерировать новое представление данных, которое можно использовать на следующем этапе. Последний этап должен использовать метод `.fit()`.

Допустим, у нас есть конвейер, включающий две модели предварительной подготовки (трансформера) `T1` и `T2` и модель машинного обучения `E`.

Во время вызова `Pipeline.fit` конвейер поочередно вызывает метод `.fit()`, а затем метод `.transform()` каждого этапа, входная информация представляет собой вывод метода `.transform()` для предыдущего этапа. Для последнего этапа конвейера просто вызывается метод `.fit()`.

Во время вызова `Pipeline.predict` конвейер поочередно вызывает метод `.transform()` каждого этапа, входная информация представляет собой вывод метода `.transform()` для предыдущего этапа. Для последнего этапа конвейера просто вызывается метод `.predict()`.

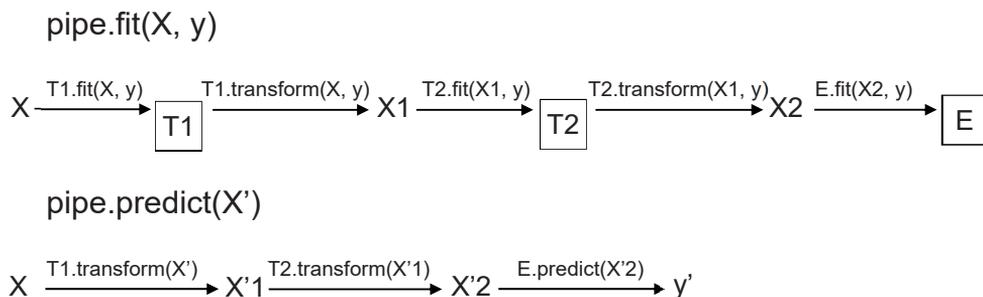


Рис. 33 Схема работы методов `Pipeline.fit()` и `Pipeline.predict()`

Давайте попробуем воспользоваться классом `Pipeline`, для этого импортируем необходимые библиотеки, классы и загрузим данные, которые мы уже использовали ранее для построения логистической регрессии.

```

# импортируем необходимые библиотеки, функцию train_test_split()
# и классы StandardScaler, LogisticRegression, Pipeline
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm.csv', sep=';')
data.head(3)

```

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	18975.456110	65999	237	1	14	0	6	0
1	4715.321344	0	65	19	56	0	3	0
2	5018.885233	54500	63	28	17	0	6	0

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# разбиваем данные на обучающие и тестовые:
# получаем обучающий массив признаков, тестовый
# массив признаков, обучающий массив меток,
# тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)
```

Теперь создаем конвейер.

```
# создаем конвейер - экземпляр класса Pipeline
pipe = Pipeline([('scaler', StandardScaler()),
                 ('logreg', LogisticRegression(
                     solver='lbfgs', max_iter=200))])
```

Здесь мы создали два этапа: первый этап, названный 'scaler', является экземпляром класса StandardScaler, а второй, названный 'logreg', является экземпляром класса LogisticRegression.

Существует удобная функция make\_pipeline(), которая позволяет создать конвейер и автоматически присвоить имя каждому этапу, исходя из его класса (напомним, что каждый этап представляет собой двухэлементный кортеж, содержащий имя и экземпляр класса).

```
# импортируем функцию make_pipeline()
from sklearn.pipeline import make_pipeline
# создаем с ее помощью конвейер
pipe_quick = make_pipeline(
    StandardScaler(),
    LogisticRegression(solver='lbfgs', max_iter=200))
```

Объекты-конвейеры pipe и pipe\_quick выполняют одну и ту же последовательность операций, но в случае с pipe\_quick имена этапов присваиваются автоматически. Мы можем взглянуть на имена этапов с помощью свойства steps.

```
# смотрим этапы конвейера
pipe_quick.steps

[('standardscaler', StandardScaler()),
 ('logisticregression', LogisticRegression(max_iter=200))]
```

Этапам присвоены имена `standardscaler` и `logisticregression`. В общем, имена этапов – это просто названия классов, написанные строчными буквами.

Теперь мы можем обучить конвейер точно так же, как и любую другую модель `scikit-learn`.

```
# обучаем конвейер
pipe.fit(X_train, y_train)
```

В данном случае `pipe.fit` сначала вызывает метод `.fit()` объекта `scaler` (вычисляет среднее и стандартное отклонение для каждого признака), затем метод `.transform()` объекта `scaler` (вычитает из исходного значения каждого признака среднее значение и делит полученный результат на стандартное отклонение, т.е. стандартизирует данные), и, наконец, метод `.fit()` объекта `logreg` (строит модель логистической регрессии на основе стандартизированных данных). Чтобы оценить правильность модели на обучающих и тестовых данных, мы просто вызываем `pipe.score`. Теперь `pipe.score` сначала вызывает метод `.transform()` объекта `scaler` (стандартизирует данные), затем метод `.score()` объекта `logreg` (вычисляет правильность).

```
# оцениваем качество конвейера на обучающих данных
print("Правильность на обучающей выборке: {:.3f}".format(
    pipe.score(X_train, y_train)))
# оцениваем качество конвейера на тестовых данных
print("Правильность на тестовой выборке: {:.3f}".format(
    pipe.score(X_test, y_test)))
```

```
Правильность на обучающей выборке: 0.900
Правильность на тестовой выборке: 0.900
```

Видно, что приведенный вывод идентичен результату, который мы получили, используя программный код в разделе 7.1, когда выполняли преобразования вручную. С помощью конвейера мы сократили программный код, необходимый для нашего процесса «предварительная обработка + классификация».

Теперь извлекаем константу и коэффициенты логистической регрессии – последнего этапа нашего конвейера. Для этого необходимо воспользоваться атрибутом `named_steps`, позволяющим получить доступ к любому этапу конвейера по заданному имени.

```
# извлекаем константу
intercept = np.round(pipe.named_steps['logreg'].intercept_[0], 3)
intercept

-2.205

# извлекаем коэффициенты
coef = np.round(pipe.named_steps['logreg'].coef_, 3)
coef

array([[ -0.022,  0.042,  0.079, -0.056, -0.004, -0.017, -0.109]])
```

Затем задаем список названий признаков и с помощью функции `zip()` «сшиваем» константу и коэффициенты с названиями признаков.

```
# записываем названия признаков
feat_labels = X_train.columns
# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], feat_labels):
    print(feature, c)
```

```
Константа: -2.205
Регрессионные коэффициенты:
Customer Lifetime Value -0.022
Income 0.042
Monthly Premium Auto 0.079
Months Since Last Claim -0.056
Months Since Policy Inception -0.004
Number of Open Complaints -0.017
Number of Policies -0.109
```

## 7.7. ЗНАКОМСТВО С КЛАССОМ COLUMNTRANSFORMER

Часто наши данные содержат как количественные, так и категориальные переменные. Для каждого типа переменных требуется своя предварительная подготовка. Допустим, пропуски в категориальных переменных мы хотим заменить самой часто встречающейся категорией и потом применить к категориальным переменным дамми-кодирование. А пропуски в количественных переменных мы хотим заменить медианами и потом применить к количественным переменным стандартизацию. Можно пойти еще дальше: пропуски в каких-то категориальных переменных заменить самой часто встречающейся категорией, а пропуски в других категориальных переменных записать в отдельную категорию.

Класс `ColumnTransformer` (входящий в новый модуль `compose`) позволяет задать конкретные столбцы или наборы столбцов, которые нужно преобразовать отдельно, и затем признаки, полученные с помощью каждого трансформера, будут объединены вместе и сформируют единое пространство трансформированных признаков.

```
from sklearn.compose import ColumnTransformer(transformers, ← Задаёт список трансформеров –
remainder='drop', ← Задаёт способ формирования трёхэлементных кортежей
'passthrough' ← выходного пространства признаков
n_jobs=None)
← Задаёт количество используемых ядер процессора для распараллеливания
```

Класс `ColumnTransformer` принимает на вход список трехэлементных кортежей. Первое значение в кортеже является **именем** самого кортежа, второе – **экземпляр класса**, третье – **список столбцов**, которые нужно преобразовать. Кортеж будет выглядеть следующим образом: `('name', SomeTransformer(parameters), columns)`.

Столбцы необязательно должны быть именами столбцов, вы можете использовать целочисленные индексы столбцов, массив булевых значений и даже функцию, которая принимает в качестве аргумента весь DataFrame и возвращает набор столбцов.

Необходимо помнить, что порядок столбцов в выходном пространстве трансформированных признаков соответствует тому порядку, в котором столбцы указаны в списке трансформеров. Столбцы исходного пространства признаков, не указанные в списке трансформеров, исключаются из получаемого пространства трансформированных признаков. С этой целью для параметра `remainder` по умолчанию задано значение `'drop'`. Если же указать `remainder='passthrough'`, то оставшиеся столбцы добавляются справа к трансформированным столбцам.

Давайте загрузим данные и потренируемся использовать класс `ColumnTransformer`.

```
# импортируем необходимые библиотеки, функцию train_test_split()
# и классы SimpleImputer, StandardScaler, OneHotEncoder,
# ColumnTransformer, LogisticRegression, Pipeline
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (StandardScaler,
                                   OneHotEncoder)
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')
data.head(5)
```

	Customer Lifetime Value	Coverage	Education	EmploymentStatus	Gender	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN	32.0	5.0	NaN	1.0	No
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN	13.0	42.0	NaN	NaN	No
2	NaN	NaN	NaN	Employed	F	48767.0	108.0	NaN	38.0	0.0	NaN	No
3	7645.861827	Basic	Bachelor	NaN	NaN	0.0	106.0	18.0	NaN	NaN	7.0	No
4	2813.692575	Basic	Bachelor	NaN	M	43836.0	73.0	12.0	NaN	NaN	1.0	No

Исходный набор содержит расширенные данные американской автостраховой компании StateFarm (добавлены категориальные признаки). Он представляет собой записи о 8293 клиентах, классифицированных на два класса: 0 – отклика нет на предложение автостраховки (7462 клиента) и 1 – отклик есть на предложение автостраховки (831 клиент). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- количественный признак *Пожизненная ценность клиента* [*Customer Lifetime Value*];
- категориальный признак *Вид страхового покрытия* [*Coverage*];
- категориальный признак *Образование* [*Education*];
- категориальный признак *Тип занятости* [*EmploymentStatus*];

- категориальный признак *Пол [Gender]*;
- количественный признак *Доход клиента [Income]*;
- количественный признак *Размер ежемесячной автостраховки [Monthly Premium Auto]*;
- количественный признак *Количество месяцев со дня подачи последнего страхового требования [Months Since Last Claim]*;
- количественный признак *Количество месяцев с момента заключения страхового договора [Months Since Policy Inception]*;
- количественный признак *Количество открытых страховых обращений [Number of Open Complaints]*;
- количественный признак *Количество полисов [Number of Policies]*;
- бинарная зависимая переменная *Отклик на предложение автостраховки [Response]*.

Видим, что наши данные содержат как количественные, так и категориальные переменные. Кроме того, у нас есть пропуски.

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)
```

Давайте создадим списки категориальных и количественных переменных.

```
# создаем списки категориальных
# и количественных столбцов
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()
```

Теперь создаем конвейер для количественных переменных и конвейер для категориальных переменных. Для количественных переменных будем применять импутацию медианами и стандартизацию, а для категориальных переменных будем применять импутацию самой часто встречающейся категорией и дамми-кодирование. Конвейеры, ответственные за операции предварительной подготовки (трансформации переменных), так и называют – трансформерами (transformers).

```
# создаем конвейер для количественных переменных
num_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
```

```
# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])
```

Теперь создаем список трансформеров – трехэлементных кортежей. Первый элемент кортежа – название конвейера с преобразованиями для определенного списка переменных (столбцов), второй элемент – собственно конвейер, и третий элемент – соответствующий список переменных (столбцов).

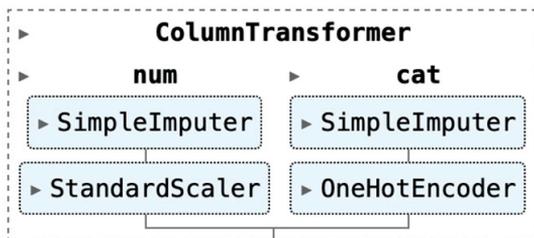
```
# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа – название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
               ('cat', cat_pipe, cat_columns)]
```

Теперь создаем экземпляр класса ColumnTransformer, передав список трансформеров.

```
# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)
```

Давайте взглянем на созданный нами объект. Здесь нас интересует порядок столбцов, который задает экземпляр класса ColumnTransformer. Это нам понадобится для правильного сопоставления весов логистической регрессии названиям признаков.

```
# смотрим трансформер
transformer
```



Вывод представляет собой схему предварительной обработки данных, заданную экземпляром класса ColumnTransformer.

Щелчком по стрелке слева от ColumnTransformer и получим следующий вывод.

```
ColumnTransformer(transformers=[('num',
                                Pipeline(steps=[('imputer',
                                                  SimpleImputer(strategy='median')),
                                                  ('scaler', StandardScaler())]),
                                ['Customer Lifetime Value', 'Income',
                                 'Monthly Premium Auto',
                                 'Months Since Last Claim',
                                 'Months Since Policy Inception',
                                 'Number of Open Complaints',
                                 'Number of Policies']),
                                ('cat',
                                 Pipeline(steps=[('imputer',
                                                  SimpleImputer(strategy='most_frequent')),
                                                  ('ohe',
                                                   OneHotEncoder(handle_unknown='ignore',
                                                                sparse=False))]),
                                ['Coverage', 'Education', 'EmploymentStatus',
                                 'Gender'])])
```

Обращаем внимание на порядок столбцов. Сначала у нас перечисляются имена количественных переменных, а затем имена категориальных переменных, потому что в нашем списке трехэлементных кортежей, который мы передали в `ColumnTransformer`, сначала указан конвейер для количественных переменных, а затем конвейер для категориальных переменных. Этот порядок столбцов отличается от порядка столбцов в исходном датафрейме.

Таким образом, класс `ColumnTransformer` поменял порядок переменных, в новом пространстве признаков сначала будут располагаться количественные переменные, затем категориальные переменные, преобразованные с помощью класса `OneHotEncoder` в дамми-переменные.

Создаем и обучаем итоговый конвейер. Итоговый конвейер содержит список из двух этапов, этап `'transformer'` – список трансформеров, выполняющих преобразования для переменных разного типа, и этап `'logreg'`, строящий модель логистической регрессии.

```
# задаем итоговый конвейер
ml_pipe = Pipeline(
    [('transform', transformer),
     ('logreg', LogisticRegression(solver='lbfgs',
                                   max_iter=200))])

# обучаем итоговый конвейер
ml_pipe.fit(X_train, y_train)
# оцениваем качество конвейера на обучающих данных
print("Правильность на обучающей выборке: {:.3f}".format(
    ml_pipe.score(X_train, y_train)))
# оцениваем качество конвейера на тестовых данных
print("Правильность на тестовой выборке: {:.3f}".format(
    ml_pipe.score(X_test, y_test)))
```

Правильность на обучающей выборке: 0.900

Правильность на тестовой выборке: 0.899

Теперь извлекаем дамми-переменные, созданные классом `OneHotEncoder`. Вспомним об атрибуте `named_steps` класса `Pipeline`, позволяющем получить

доступ к любому этапу конвейера по заданному имени. Итак, с помощью атрибута `named_steps` класса `Pipeline` обращаемся к этапу `'transform'` итогового конвейера `ml_pipe`, затем с помощью свойства `named_transformers_` класса `ColumnTransformer` обращаемся к названию конвейера для категориальных признаков `'cat'`, находящемуся внутри этапа `'transform'`. С помощью атрибута `named_steps` класса `Pipeline` обращаемся к этапу `'ohe'`, находящемуся внутри конвейера для категориальных признаков `'cat'`, затем с помощью метода `.get_feature_names_out()` класса `ColumnTransformer` получаем массив с именами признаков после трансформации и создаем из него список.

```
# извлекаем дамми-переменные, созданные
# классом OneHotEncoder
```

```
cat = ml_pipe.named_steps['transform'].named_transformers_['cat']
onehot_columns = list(cat.named_steps['ohe'].get_feature_names_out(
    input_features=cat_columns))
onehot_columns
```

```
['Coverage_Basic',
 'Coverage_Extended',
 'Coverage_Premium',
 'Education_Bachelor',
 'Education_College',
 'Education_Doctor',
 'Education_High School or Below',
 'Education_Master',
 'EmploymentStatus_Disabled',
 'EmploymentStatus_Employed',
 'EmploymentStatus_Medical Leave',
 'EmploymentStatus_Retired',
 'EmploymentStatus_Unemployed',
 'Gender_F',
 'Gender_M']
```

```
# еще можно применить такой стиль
```

```
cat = ml_pipe.named_steps.transform.named_transformers_.cat
onehot_columns = list(cat.named_steps.ohe.get_feature_names_out(
    input_features=cat_columns))
onehot_columns
```

```
['Coverage_Basic',
 'Coverage_Extended',
 'Coverage_Premium',
 'Education_Bachelor',
 'Education_College',
 'Education_Doctor',
 'Education_High School or Below',
 'Education_Master',
 'EmploymentStatus_Disabled',
 'EmploymentStatus_Employed',
 'EmploymentStatus_Medical Leave',
 'EmploymentStatus_Retired',
 'EmploymentStatus_Unemployed',
 'Gender_F',
 'Gender_M']
```

Добавляем в конец списка количественных переменных список дамми-переменных, созданных классом `OneHotEncoder`, т.е. сохраняем тот же порядок столбцов, что задал класс `ColumnTransformer`.

```
# добавляем в конец списка количественных переменных
# дамми-переменные, созданные OneHotEncoder, т.е.
# сохраняем тот же порядок столбцов, что задал
# ColumnTransformer
all_columns_lst = num_columns + onehot_columns
all_columns_lst
```

```
['Customer Lifetime Value',
 'Income',
 'Monthly Premium Auto',
 'Months Since Last Claim',
 'Months Since Policy Inception',
 'Number of Open Complaints',
 'Number of Policies',
 'Coverage_Basic',
 'Coverage_Extended',
 'Coverage_Premium',
 'Education_Bachelor',
 'Education_College',
 'Education_Doctor',
 'Education_High School or Below',
 'Education_Master',
 'EmploymentStatus_Disabled',
 'EmploymentStatus_Employed',
 'EmploymentStatus_Medical Leave',
 'EmploymentStatus_Retired',
 'EmploymentStatus_Unemployed',
 'Gender_F',
 'Gender_M']
```

Теперь извлекаем константу и коэффициенты логистической регрессии – последнего этапа нашего конвейера. Для этого вновь необходимо воспользоваться атрибутом `named_steps` класса `Pipeline`.

```
# извлекаем константу
intercept = np.round(ml_pipe.named_steps['logreg'].intercept_[0], 3)
intercept
```

```
-1.697
```

```
# извлекаем коэффициенты
coef = np.round(ml_pipe.named_steps['logreg'].coef_, 3)
coef
```

```
array([[ 7.000e-03,  2.300e-02,  1.240e-01, -4.700e-02, -2.900e-02,
        -4.000e-02, -5.900e-02, -3.600e-02,  1.290e-01, -9.400e-02,
        -1.640e-01, -2.100e-02,  4.120e-01, -1.710e-01, -5.700e-02,
         1.000e-03, -5.240e-01, -1.000e-01,  1.648e+00, -1.026e+00,
        -7.000e-03,  6.000e-03]])
```

С помощью функции `zip()` «сшиваем» константу и коэффициенты с названиями признаков.

```
# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], all_columns_lst):
    print(feature, c)
```

```
Константа: -1.697
Регрессионные коэффициенты:
Customer Lifetime Value 0.007
Income 0.023
Monthly Premium Auto 0.124
Months Since Last Claim -0.047
Months Since Policy Inception -0.029
Number of Open Complaints -0.04
Number of Policies -0.059
Coverage_Basic -0.036
Coverage_Extended 0.129
Coverage_Premium -0.094
Education_Bachelor -0.164
Education_College -0.021
Education_Doctor 0.412
Education_High School or Below -0.171
Education_Master -0.057
EmploymentStatus_Disabled 0.001
EmploymentStatus_Employed -0.524
EmploymentStatus_Medical Leave -0.1
EmploymentStatus_Retired 1.648
EmploymentStatus_Unemployed -1.026
Gender_F -0.007
Gender_M 0.006
```

С помощью функций `make_pipeline()` и `make_column_transformer()`, класса `make_column_selector` можно сократить объем программного кода, требуемого для создания экземпляра класса `ColumnTransformer`.

Существует удобная функция `make_column_transformer()`, которая позволяет создать экземпляр класса `ColumnTransformer` и автоматически присвоить имя каждому конвейеру, исходя из номера списка столбцов, к которому он должен быть применен. Например, конвейер, предназначенный для обработки первого списка – списка количественных признаков, будет назван `'pipeline-1'`, а конвейер, предназначенный для обработки второго списка – списка категориальных признаков, будет назван `'pipeline-2'`. Списки признаков определенного типа можно создавать автоматически с помощью класса `make_column_selector`. У класса `make_column_selector` будут два параметра: параметр `dtype_include` задает тип признаков для включения, а параметр `dtype_exclude` – тип признаков для исключения. В итоге мы можем передать в функцию `make_column_transformer()` двухэлементные кортежи, в которых первый элемент – конвейер

ер, создаваемый с помощью функции `make_pipeline()`, а второй элемент – список признаков, создаваемый с помощью класса `make_column_selector`.

Давайте импортируем функции `make_pipeline()` и `make_column_transformer()`, класс `make_column_selector`.

```
# импортируем функции make_pipeline() и make_column_transformer(),
# класс make_column_selector
from sklearn.compose import (make_column_transformer,
                             make_column_selector)
from sklearn.pipeline import make_pipeline
```

Теперь с помощью функции `make_column_transformer()` автоматически создаем экземпляр класса `ColumnTransformer`, при этом автоматически создаем трансформеры с помощью функции `make_pipeline()` и списки признаков с помощью класса `make_column_selector`.

```
# автоматически создаем экземпляр класса ColumnTransformer,
# при этом автоматически создаем трансформеры и списки
# признаков
column_transformer_quick = make_column_transformer(
    (make_pipeline(SimpleImputer(strategy='median'),
                  StandardScaler()),
     make_column_selector(dtype_include=np.number)),
    (make_pipeline(SimpleImputer(strategy='most_frequent'),
                  OneHotEncoder()),
     make_column_selector(dtype_include=object)))
```

Давайте посмотрим получившийся результат (вывод для удобства восприятия изменен).

```
column_transformer_quick

ColumnTransformer(
  transformers=[('pipeline-1', Pipeline(
    steps=[('simpleimputer', SimpleImputer(strategy='median')),
           ('standardscaler', StandardScaler())]),
    <sklearn.compose._column_transformer.make_column_selector object 1>,
  ('pipeline-2', Pipeline(steps=[('simpleimputer',
                                  SimpleImputer(
                                    strategy='most_frequent')),
                                  ('onehotencoder',
                                   OneHotEncoder())]),
    <sklearn.compose._column_transformer.make_column_selector object 2>)]])
```

Теперь с помощью функции `make_pipeline()` автоматически создаем итоговый конвейер.

```
# автоматически создаем итоговый конвейер
pipe_quick = make_pipeline(
    column_transformer_quick,
    LogisticRegression(solver='lbfgs', max_iter=200))
```

Вновь посмотрим получившийся результат (вывод для удобства восприятия изменен).

```
pipe_quick
```

```
Pipeline(
  steps=[('columntransformer',
          ColumnTransformer(
            transformers=[
              ('pipeline-1', Pipeline(steps=[('simpleimputer',
                                             SimpleImputer(strategy='median')),
                                             ('standardscaler',
                                              StandardScaler())])),
              (<sklearn.compose._column_transformer.make_column_selector object 1>,
              ('pipeline-2', Pipeline(steps=[('simpleimputer',
                                             SimpleImputer(
                                               strategy='most_frequent')),
                                             ('onehotencoder',
                                              OneHotEncoder())])),
              (<sklearn.compose._column_transformer.make_column_selector object 2>)])),
          ('logisticregression', LogisticRegression(max_iter=200))]
```

Давайте вновь обучим наш конвейер, оценим качество прогнозов и выведем коэффициенты.

```
# обучаем итоговый конвейер
pipe_quick.fit(X_train, y_train)
# оцениваем качество конвейера на обучающих данных
print("Правильность на обучающей выборке: {:.3f}".format(
    pipe_quick.score(X_train, y_train)))
# оцениваем качество конвейера на тестовых данных
print("Правильность на тестовой выборке: {:.3f}".format(
    pipe_quick.score(X_test, y_test)))
```

```
Правильность на обучающей выборке: 0.900
```

```
Правильность на тестовой выборке: 0.899
```

```
# извлекаем дамми-переменные,
# созданные классом OneHotEncoder
cat = pipe_quick.named_steps['columntransformer']\
    .named_transformers_['pipeline-2']
onehot_columns = list(cat.named_steps['onehotencoder'].get_feature_names_out(
    input_features=cat_columns))
onehot_columns
```

```
['Coverage_Basic',
 'Coverage_Extended',
 'Coverage_Premium',
 'Education_Bachelor',
 'Education_College',
 'Education_Doctor',
 'Education_High School or Below',
 'Education_Master',
 'EmploymentStatus_Disabled',
 'EmploymentStatus_Employed',
 'EmploymentStatus_Medical Leave',
 'EmploymentStatus_Retired',
 'EmploymentStatus_Unemployed',
```

```

'Gender_F',
'Gender_M']

# добавляем в конец списка количественных переменных
# дамми-переменные, созданные OneHotEncoder, т. е.
# сохраняем тот же порядок столбцов, что задал
# ColumnTransformer
all_columns_lst = num_columns + onehot_columns
all_columns_lst

['Customer Lifetime Value',
'Income',
'Monthly Premium Auto',
'Months Since Last Claim',
'Months Since Policy Inception',
'Number of Open Complaints',
'Number of Policies',
'Coverage_Basic',
'Coverage_Extended',
'Coverage_Premium',
'Education_Bachelor',
'Education_College',
'Education_Doctor',
'Education_High School or Below',
'Education_Master',
'EmploymentStatus_Disabled',
'EmploymentStatus_Employed',
'EmploymentStatus_Medical Leave',
'EmploymentStatus_Retired',
'EmploymentStatus_Unemployed',
'Gender_F',
'Gender_M']

# извлекаем константу
logreg_step = pipe_quick.named_steps['logisticregression']
intercept = np.round(logreg_step.intercept_[0], 3)
# извлекаем коэффициенты
coef = np.round(logreg_step.coef_, 3)

# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], all_columns_lst):
    print(feature, c)

```

```

Константа: -1.697
Регрессионные коэффициенты:
Customer Lifetime Value 0.007
Income 0.023
Monthly Premium Auto 0.124
Months Since Last Claim -0.047
Months Since Policy Inception -0.029
Number of Open Complaints -0.04

```

```

Number of Policies -0.059
Coverage_Basic -0.036
Coverage_Extended 0.129
Coverage_Premium -0.094
Education_Bachelor -0.164
Education_College -0.021
Education_Doctor 0.412
Education_High School or Below -0.171
Education_Master -0.057
EmploymentStatus_Disabled 0.001
EmploymentStatus_Employed -0.524
EmploymentStatus_Medical Leave -0.1
EmploymentStatus_Retired 1.648
EmploymentStatus_Unemployed -1.026
Gender_F -0.007
Gender_M 0.006

```

Напомним: для списка трансформеров вместо списков количественных и категориальных переменных можно использовать массив булевых значений без/с оператором побитового отрицания `~`. Для этого вместо метода `.select_dtypes()` надо воспользоваться свойством `dtypes`, чтобы получить массив булевых значений.

```

# создаем массив булевых значений
categorical = X_train.dtypes == object
categorical

```

```

Customer Lifetime Value      False
Coverage                      True
Education                    True
EmploymentStatus              True
Gender                        True
Income                       False
Monthly Premium Auto         False
Months Since Last Claim      False
Months Since Policy Inception False
Number of Open Complaints    False
Number of Policies           False
dtype: bool

```

```

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, ~categorical),
               ('cat', cat_pipe, categorical)]

```

```

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)

```

```

# задаем итоговый конвейер
ml_pipe = Pipeline(
    [('transform', transformer),
     ('logreg', LogisticRegression(solver='lbfgs',
                                   max_iter=200))])

```

```

# обучаем итоговый конвейер

```

```

ml_pipe.fit(X_train, y_train)
# оцениваем качество конвейера на обучающих данных
print("Правильность на обучающей выборке: {:.3f}".format(
    ml_pipe.score(X_train, y_train)))
# оцениваем качество конвейера на тестовых данных
print("Правильность на тестовой выборке: {:.3f}".format(
    ml_pipe.score(X_test, y_test)))

Правильность на обучающей выборке: 0.900
Правильность на тестовой выборке: 0.899

# извлекаем дамми-переменные, созданные классом OneHotEncoder
cat = ml_pipe.named_steps['transform'].named_transformers_['cat']
onehot_cols = list(cat.named_steps['ohe'].get_feature_names_out(
    input_features=X_train.loc[:, categorical].columns))
num_cols = X_train.loc[:, ~categorical].columns.tolist()

# добавляем в конец списка количественных переменных
# дамми-переменные, созданные OneHotEncoder, т.е.
# сохраняем тот же порядок столбцов, что задал
# ColumnTransformer
all_cols_lst = num_cols + onehot_cols

# извлекаем константу
lr_step = ml_pipe.named_steps['logreg']
intercept = np.round(lr_step.intercept_[0], 3)
# извлекаем коэффициенты
coef = np.round(lr_step.coef_, 3)

# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], all_cols_lst):
    print(feature, c)

```

```

Константа: -1.697
Регрессионные коэффициенты:
Customer Lifetime Value 0.007
Income 0.023
Monthly Premium Auto 0.124
Months Since Last Claim -0.047
Months Since Policy Inception -0.029
Number of Open Complaints -0.04
Number of Policies -0.059
Coverage_Basic -0.036
Coverage_Extended 0.129
Coverage_Premium -0.094
Education_Bachelor -0.164
Education_College -0.021
Education_Doctor 0.412
Education_High School or Below -0.171
Education_Master -0.057
EmploymentStatus_Disabled 0.001

```

```

EmploymentStatus_Employed -0.524
EmploymentStatus_Medical Leave -0.1
EmploymentStatus_Retired 1.648
EmploymentStatus_Unemployed -1.026
Gender_F -0.007
Gender_M 0.006

```

## 7.8. КЛАСС FEATUREUNION

Как и класс `ColumnTransformer`, класс `FeatureUnion` принимает на вход список трансформеров и конкатенирует результаты этих трансформеров. Разница между ними заключается в том, что `FeatureUnion` применяет различные трансформеры **ко всем входным данным**, а затем объединяет результаты путем их конкатенации. `ColumnTransformer` применяет разные трансформеры к **разным подмножествам входных данных**, а затем объединяет результаты путем их конкатенации.

Приведем пример применения класса `FeatureUnion` на данных `StateFarm`.

```

# импортируем библиотеку pandas и необходимые классы
import pandas as pd
from sklearn.pipeline import FeatureUnion
from sklearn.decomposition import PCA, TruncatedSVD

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm.csv', sep=';')
data.head(3)

```

	Customer Lifetime Value	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	18975.456110	65999	237	1	14	0	6	0
1	4715.321344	0	65	19	56	0	3	0
2	5018.885233	54500	63	28	17	0	6	0

Сейчас мы создадим массив признаков и массив меток и применим ко всему массиву признаков две процедуры – анализ главных компонент и усеченное сингулярное разложение. Первая процедура даст в итоге два признака, а вторая процедура – один признак, в итоге преобразованный массив признаков будет состоять из трех признаков.

```

# создаем массив меток и массив признаков
label = data.pop('Response')

# проиллюстрируем работу
union = FeatureUnion([('pca', PCA(n_components=1)),
                      ('svd', TruncatedSVD(n_components=2))])
union.fit_transform(data)

array([[ 28273.41204393,  67940.15460167, 10005.84432796],
       [-37806.25947898,  628.90764304,  4673.59827961],
       [ 16694.58513307,  54682.57666392, -2293.32343664],
       ...,
       [-37806.13907195,  631.67210945,  4694.33429531],
       [ 44111.51317356,  81930.31618429, -5364.98348564],
       [-37796.22695042,  862.2607331 ,  6407.91339948]])

```

## 7.9. ВЫПОЛНЕНИЕ ПЕРЕКРЕСТНОЙ ПРОВЕРКИ С ПОМОЩЬЮ ФУНКЦИИ `CROSS_VAL_SCORE()`, ПОЛУЧЕНИЕ ПРОГНОЗОВ ПЕРЕКРЕСТНОЙ ПРОВЕРКИ С ПОМОЩЬЮ ФУНКЦИИ `CROSS_VAL_PREDICT()`, СОХРАНЕНИЕ МОДЕЛЕЙ ПЕРЕКРЕСТНОЙ ПРОВЕРКИ С ПОМОЩЬЮ ФУНКЦИИ `CROSS_VALIDATE()`

Ранее мы говорили о таком простом способе проверки модели, как случайное разбиение на обучающую и тестовую выборки. Еще одним методом проверки является  $k$ -блочная перекрестная проверка. Метод  $k$ -блочной перекрестной проверки (ее еще называют кросс-валидацией) разбивает весь набор данных на  $k$  блоков (обычно 5 или 10) приблизительно равного объема, а затем  $k$  раз на  $k - 1$  блоках осуществляет обучение модели (эти блоки называют обучающими, по сути они формируют обучающую выборку), а блок, не участвовавший в обучении, использует для проверки (этот блок называют тестовым блоком, по сути он является тестовой выборкой, однако если перекрестная проверка применяется внутри комбинированной проверки для настройки гиперпараметров и выбора оптимальной модели, блок называют проверочным блоком, или проверочной выборкой).

Проще говоря, для набора данных, разбитого, допустим, на 5 блоков, будет построено 5 моделей, первая модель обучается по всем данным, за исключением данных из первого блока, вторая модель обучается по всем данным, за исключением данных из второго блока, и т. д. Качество модели проверяется путем применения модели, построенной по обучающим блокам, к тестовому блоку, исключенному из процесса построения данной модели. Затем метрики качества, вычисленные по каждому тестовому блоку, усредняют и получают итоговую метрику качества модели.



Рис. 34 Схема перекрестной проверки

Отметим основную разницу между методом разбиения на обучающую и тестовую выборки и  $k$ -блочной перекрестной проверкой: в  $k$ -блочной перекрестной проверке для обучения и проверки используются все доступные данные, тогда как при разбиении на обучающую и тестовую выборки наши данные, отложенные под соответствующую выборку, словно «заморожены». Идея, лежащая в основе этого метода, состоит в уменьшении пессимистичного смещения за счет исполь-

зования большего количества данных для обучения, вместо того чтобы отложить в качестве тестовой выборки довольно большую часть набора. Кроме того, перекрестная проверка имитирует важное свойство реального мира: вариативность получения обучающего и тестового наборов в силу случайности, тогда как при случайном разбиении на обучающую и тестовую выборки мы работаем с одной определенной обучающей выборкой и одной определенной тестовой выборкой.

Для выполнения перекрестной проверки мы можем воспользоваться функцией `cross_val_score()`.

```
from sklearn.model_selection import cross_val_score(
    estimator, X, y=None, groups=None, scoring=None,
    cv=None, n_jobs=None)

# estimator — Модель, используемая для обучения
# X — Массив признаков
# y=None — Массив меток
# groups=None — Массив меток
# scoring=None — Стратегия перекрестной проверки
# cv=None — Возможные значения:
#               • None, будет использована 5-блочная перекрестная
#               проверка,
#               • целочисленное значение, задает количество блоков
#               для (Stratified)Kfold,
#               • генератор расщеплений на обучающую и тестовую
#               выборки
# n_jobs=None — Для None и целочисленных значений, если модель
#                является классификатором и у является либо бинарной,
#                либо мультиклассовой переменной,
#                используется StratifiedKfold. Во всех остальных
#                случаях используется Kfold
```

Давайте потренируемся использовать функцию `cross_val_score()`. Для этого импортируем необходимые библиотеки, классы и функции.

```
# импортируем необходимые библиотеки, классы SimpleImputer,
# StandardScaler, OneHotEncoder, ColumnTransformer,
# LogisticRegression, Pipeline, функции cross_val_score(),
# cross_val_predict(), cross_validate()
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (StandardScaler,
                                   OneHotEncoder)
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import (GradientBoostingRegressor,
                              GradientBoostingClassifier)
from sklearn.pipeline import Pipeline
from sklearn.model_selection import (cross_val_score,
                                    cross_val_predict,
                                    cross_validate)
```

Начнем с задачи классификации и загрузим уже знакомые данные компании StateFarm с пропусками.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')
data.head(5)
```

	Customer Lifetime Value	Coverage	Education	EmploymentStatus	Gender	Income	Monthly Premium Auto	Months Since Last Claim	Months Since Policy Inception	Number of Open Complaints	Number of Policies	Response
0	2763.519279	Basic	Bachelor	Employed	F	56274.0	NaN	32.0	5.0	NaN	1.0	No
1	NaN	NaN	Bachelor	Unemployed	F	0.0	NaN	13.0	42.0	NaN	NaN	No
2	NaN	NaN	NaN	Employed	F	48767.0	108.0	NaN	38.0	0.0	NaN	No
3	7645.861827	Basic	Bachelor	NaN	NaN	0.0	106.0	18.0	NaN	NaN	7.0	No
4	2813.692575	Basic	Bachelor	NaN	M	43836.0	73.0	12.0	NaN	NaN	1.0	No

С помощью метода `.pop()` сформируем массив признаков и массив меток. Метод `.pop()` возвращает указанную серию и удаляет ее из датафрейма.

```
# создаем массив меток и массив признаков
у = data.pop('Response')
```

Часто начинающие специалисты выполняют операции предварительной подготовки данных, предполагающие вычисления (укрупнение редких категорий по порогу, импутацию пропусков статистиками, стандартизацию, биннинг и конструирование признаков на основе статистик и т. д.) на всем наборе, а потом запускают перекрестную проверку.

Проблема заключается в том, что если мы используем перекрестную проверку и выполняем перечисленные операции до нее, получается, что в каждой итерации перекрестной проверки при вычислении статистик для импутации, среднего и стандартного отклонения по каждому признаку для стандартизации, правил биннинга, частот и вероятностей положительного класса зависимой переменной в категориях признака использовались все наблюдения набора, часть из которых у нас теперь находится в тестовом блоке (по сути, выборке новых данных).

И вот таких случаях в Python нас снова выручают классы `Pipeline` и `ColumnTransformer`.

Мы вновь создадим списки количественных и категориальных переменных, создадим отдельные конвейеры для переменных разного типа, создадим список трансформеров, передадим этот список в `ColumnTransformer` и создадим итоговый конвейер.

```
# создаем списки категориальных
# и количественных столбцов
cat_columns = data.select_dtypes(
    include='object').columns.tolist()
num_columns = data.select_dtypes(
    exclude='object').columns.tolist()

# создаем конвейер для количественных переменных
num_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('ohc', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
                ('cat', cat_pipe, cat_columns)]

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)
```

```
# задаем итоговый конвейер
ml_pipe = Pipeline([
    ('transform', transformer),
    ('logreg', LogisticRegression(solver='lbfgs',
                                  max_iter=200))])
```

Теперь мы можем выполнить перекрестную проверку, разместив итоговый конвейер внутри цикла 5-блочной перекрестной проверки. Таким образом, на каждой итерации перекрестной проверки мы на обучающих блоках строим модели предварительной подготовки, а затем модель логистической регрессии и оцениваем ее качество на тестовом блоке. Затем метрики качества (оценки правильности), полученные по 5 итерациям, усредняем.

```
# выполняем перекрестную проверку, разместив итоговый
# конвейер внутри цикла перекрестной проверки
cv_mean = cross_val_score(ml_pipe, data, y, cv=5).mean()
print("Усредненная правильность cv: {:.3f}".format(cv_mean))
```

Усредненная правильность cv: 0.897

На практике берут не просто усредненное значение метрики, а усредненное значение метрики с поправкой на стандартное отклонение метрики (используем эту статистику как штраф, если метрику максимизируем, то из среднего значения метрики вычитаем стандартное отклонение, а если метрику минимизируем, то к среднему значению добавляем стандартное отклонение).

```
# выполняем перекрестную проверку, разместив итоговый
# конвейер внутри цикла перекрестной проверки
cv_mean = cross_val_score(
    ml_pipe, data, y, scoring='roc_auc', cv=5).mean()
cv_std = cross_val_score(
    ml_pipe, data, y, scoring='roc_auc', cv=5).std()
print("Усредненная оценка AUC-ROC cv: {:.3f}".format(cv_mean))
print("Стандартное отклонение AUC-ROC cv: {:.3f}".format(cv_std))
cv_mean_corr = cv_mean - cv_std
print(f"Усредненная оценка AUC-ROC с поправкой\n"
      f"на стандартное отклонение: {cv_mean_corr:.3f}")
```

Усредненная оценка AUC-ROC cv: 0.635

Стандартное отклонение AUC-ROC cv: 0.026

Усредненная оценка AUC-ROC с поправкой

на стандартное отклонение: 0.609

Обратите внимание, что, как и случайное разбиение на обучающую и тестовую выборки, перекрестную проверку нельзя использовать для выбора модели, если предполагается настройка гиперпараметров. Допустим, мы строим модель случайного леса (или конвейер моделей: например, модель импутации для количественных переменных → модель случайного леса) с разными значениями гиперпараметров на обучающих блоках, а проверяем ее качество на тестовых блоках. Получается, мы используем тестовые блоки и для настройки гиперпараметров, и для итоговой оценки качества модели.

С помощью функции `cross_val_predict()` мы в каждой итерации перекрестной проверки можем вычислить спрогнозированные значения для наблюдений

каждого тестового блока с помощью модели, обученной на обучающих блоках, потом эти спрогнозированные значения конкатенируются, сортируются, и мы получаем спрогнозированные значения для всего набора. Эта функция часто используется в классах, реализующих классический стекинг. В классическом стекинге прогнозы базовых моделей, полученные для тестовых блоков перекрестной проверки, используются в качестве метапризнаков для метамодели.

```
# мы вычисляем спрогнозированные значения для
# наблюдений каждого тестового блока с помощью
# модели, обученной на обучающих блоках
predictions = cross_val_predict(ml_pipe, data, y, cv=5)
predictions

array(['No', 'No', 'No', ..., 'No', 'No', 'No'], dtype=object)
```

А теперь представьте: нам нужно получить коэффициенты логистической регрессии, обученной на каждой итерации перекрестной проверки. Здесь нам понадобится функция `cross_validate()`. С ее помощью мы создаем объект, в который будем сохранять информацию о конвейере моделей, обученном на каждой итерации перекрестной проверки. Для этого вы должны установить значение `True` для параметра `return_estimator` функции `cross_validate()`. Затем создаем пустой список, в который сохраняем коэффициенты логистической регрессии, извлеченные из объекта, хранящего информацию о конвейере моделей, обученном на каждой итерации перекрестной проверки. В итоге у нас будет список списков значений регрессионных коэффициентов.

```
# создаем объект, в который будем сохранять
# результаты перекрестной проверки
cv_results = cross_validate(ml_pipe,
                             data,
                             y,
                             cv=5,
                             return_estimator=True)

# создаем пустой список, в который будем сохранять
# коэффициенты логистической регрессии
coefs = []
# для каждой модели сохраняем коэффициенты в список
for model in cv_results['estimator']:
    coefs.append(list(model[1].intercept_) + list(model[1].coef_[0]))

[[-1.6557542038814281, ← константа 1-й модели
  -0.027229128042908744, ← регрессионные коэффициенты 1-й модели
  -0.005572260533719482],
 [-1.7073464334264254, ← константа 5-й модели
  -0.04804586614059469, ← регрессионные коэффициенты 5-й модели
   0.019724215289614083]]
```

Рис. 35 Список списков coefs

Теперь извлекаем дамми-переменные, созданные классом `OneHotEncoder`, добавляем в конец списка количественных переменных, затем в начало списка добавляем строковое значение `'Intercept'`.

```

# извлекаем дамми-переменные, созданные классом OneHotEncoder
ml_pipe.fit(data, y)
cat = ml_pipe.named_steps['transform'].named_transformers_['cat']
onehot_columns = list(cat.named_steps['ohe'].get_feature_names_out(
    input_features=cat_columns))

# добавляем в конец списка количественных переменных
# дамми-переменные, созданные OneHotEncoder, т.е.
# сохраняем тот же порядок столбцов, что задал
# ColumnTransformer
num_columns.extend(onehot_columns)

# добавляем Intercept в начало списка
num_columns.insert(0, 'Intercept')

```

Теперь сопоставляем коэффициенты переменным в итоговом списке, результат сопоставления оформляем в датафрейм pandas и для удобства транспонируем, чтобы признаки стали строками, а номера итераций перекрестной проверки – столбцами.

```

# теперь сопоставляем признакам коэффициенты модели
lr_coefs = pd.DataFrame(coefs, columns=num_columns).T
# смотрим на результат
lr_coefs

```

	0	1	2	3	4
<b>Intercept</b>	-1.655754	-1.734709	-1.727187	-1.693281	-1.707346
<b>Customer Lifetime Value</b>	-0.027229	0.016621	-0.004951	-0.057115	-0.048046
<b>Income</b>	0.072515	0.058170	0.033085	0.045221	0.058014
<b>Monthly Premium Auto</b>	0.105276	0.092715	0.139402	0.129031	0.129856
<b>Months Since Last Claim</b>	-0.025576	-0.023675	-0.026496	-0.015031	-0.057444
<b>Months Since Policy Inception</b>	-0.016235	0.007449	-0.014484	0.016502	0.012168
<b>Number of Open Complaints</b>	-0.041276	-0.035427	-0.034442	-0.030048	-0.014053
<b>Number of Policies</b>	-0.058914	-0.051552	-0.053842	-0.046859	-0.090626
<b>Coverage_Basic</b>	-0.029730	0.041536	0.062551	-0.005767	0.014594
<b>Coverage_Extended</b>	0.085790	0.083429	0.090402	0.104599	0.107809
<b>Coverage_Premium</b>	-0.056129	-0.125313	-0.153198	-0.098963	-0.123596
<b>Education_Bachelor</b>	-0.104063	-0.142671	-0.220087	-0.129386	-0.114116
<b>Education_College</b>	-0.012814	-0.013205	-0.036655	-0.009937	-0.080164
<b>Education_Doctor</b>	0.301580	0.371795	0.360314	0.311182	0.414890
<b>Education_High School or Below</b>	-0.152756	-0.186809	-0.165951	-0.257480	-0.151275
<b>Education_Master</b>	-0.032017	-0.029459	0.062134	0.085489	-0.070528
<b>EmploymentStatus_Disabled</b>	-0.188859	-0.122412	-0.142471	-0.107659	-0.012759
<b>EmploymentStatus_Employed</b>	-0.611002	-0.522530	-0.526585	-0.534611	-0.528384
<b>EmploymentStatus_Medical Leave</b>	-0.124838	-0.009983	-0.008835	-0.012086	-0.179149
<b>EmploymentStatus_Retired</b>	1.864284	1.606893	1.678562	1.689315	1.770246
<b>EmploymentStatus_Unemployed</b>	-0.939654	-0.952317	-1.000916	-1.035090	-1.051147
<b>Gender_F</b>	0.005503	-0.021311	-0.051113	-0.034866	-0.020917
<b>Gender_M</b>	-0.005572	0.020963	0.050867	0.034734	0.019724

Теперь перейдем к задаче регрессии и загрузим данные о сделках с недвижимостью.

```
# записываем CSV-файл в объект DataFrame
```

```
df = pd.read_csv('Data/Flats_missing.csv', sep=';', decimal=',')
df.head(3)
```

	Rooms_Number	District	Stor	Storeys	Space_Total	Space_Living	Space_Kitchen	Balcon_Num	Lodgee_Num	lat	Long	Cost_KV
0	1	Заяльцовский	13	17.0	54.1	18.0	21.2	0.0	1	55.0725	82.9069	50831.79298
1	1	Заяльцовский	10	17.0	54.5	18.0	21.1	0.0	1	55.0725	82.9069	52000.00000
2	1	Центральный	8	17.0	37.0	0.0	0.0	0.0	0	55.0725	82.9068	87837.83784

Наша задача – предсказать стоимость квадратного метра жилья в Новосибирске (*Cost\_KV*). Метрики качества –  $R^2$  и RMSE.

По каждому наблюдению (сделке) фиксируются следующие переменные (характеристики):

- количественный признак *Количество комнат* [*Rooms\_Number*];
- категориальный признак *Район Новосибирска* [*District*];
- количественный признак *Этаж* [*Stor*];
- количественный признак *Количество этажей* [*Storeys*];
- количественный признак *Общая площадь* [*Space\_Total*];
- количественный признак *Жилая площадь* [*Space\_Living*];
- количественный признак *Площадь кухни* [*Space\_Kitchen*];
- количественный признак *Количество балконов* [*Balcon\_Num*];
- количественный признак *Количество лоджий* [*Lodgee\_Num*];
- количественный признак *Широта* [*lat*];
- количественный признак *Долгота* [*Long*];
- количественная зависимая переменная *Стоимость квадратного метра* [*Cost\_KV*].

Опять с помощью метода `.pop()` сформируем массив признаков и массив меток.

```
# создаем массив меток и массив признаков
```

```
label = df.pop('Cost_KV')
```

Мы вновь зададим списки количественных и категориальных переменных, назначим отдельные конвейеры для переменных разного типа, создадим список трансформеров, передадим этот список в `ColumnTransformer` и создадим итоговый конвейер. Для количественных переменных мы будем применять импутацию медианами и стандартизацию, а для категориальных переменных будем применять импутацию модой и дамми-кодирование.

```
# создаем списки категориальных
```

```
# и количественных столбцов
```

```
cat_cols = df.select_dtypes(
    include='object').columns.tolist()
```

```
num_cols = df.select_dtypes(
    exclude='object').columns.tolist()
```

```
# создаем конвейер для количественных переменных
```

```
num_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
```

```
# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])
```

```
# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа – название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_cols),
                ('cat', cat_pipe, cat_cols)]
```

```
# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)
```

```
# задаем итоговый конвейер
pipe = Pipeline([
    ('transform', transformer),
    ('boost', GradientBoostingRegressor(
        max_depth=6, random_state=46))])
```

Сейчас на каждой итерации перекрестной проверки мы на обучающих блоках строим модели предварительной подготовки, а потом модель градиентного бустинга и оцениваем ее качество на тестовом блоке. Затем метрики качества (оценки RMSE и R2), полученные по 5 итерациям, усредняем. Обратите внимание: для RMSE мы задали 'neg\_mean\_squared\_error', поскольку в scikit-learn все объекты-скореры подчиняются соглашению «чем выше значение, тем лучше», т.е. наша цель всегда заключается в том, чтобы максимизировать значение метрики, поэтому для минимизируемых метрик мы используем метрики, принимающие отрицательные значения. Допустим, мы получили значения RMSE –1000, –500, –200. –200 будет наилучшим значением RMSE. Чем больше отрицательное значение метрики (т.е. чем ближе значение RMSE к нулю), тем лучше.

```
# выполняем перекрестную проверку, разместив итоговый
# конвейер внутри цикла перекрестной проверки
rmse_cv = cross_val_score(pipe,
                           df,
                           label,
                           scoring='neg_mean_squared_error',
                           cv=5)
r2_cv = cross_val_score(pipe, df, label, cv=5)
print("RMSE, усреднение по 5 тестовым блокам cv: {:.3f}".format(np.mean(
    np.sqrt(np.abs(rmse_cv)))))
print("R2, усреднение по 5 тестовым блокам cv: {:.3f}".format(
    np.mean(r2_cv)))
```

```
RMSE, усредненный по 5 тестовым блокам cv: 11864.022
R2, усредненный по 5 тестовым блокам cv: 0.404
```

Методы машинного обучения на основе деревьев (одиночное дерево, случайный лес и градиентный бустинг) позволяют получить важности признаков – оценки полезности признаков. Эту информацию полезно использовать для отбора признаков. Оценки важностей будут более надежными, если их получить с помощью перекрестной проверки. Давайте проиллюстрируем это на простом примере.

Загружаем данные, формируем массив меток и массив признаков, создаем экземпляр класса `GradientBoostingClassifier`, выполняем перекрестную проверку и сохраняем информацию о моделях с помощью функции `cross_validate()`.

```
# загружаем данные
data = pd.read_csv('Data/StateFarm.csv', sep=';')
# создаем массив меток и массив признаков
y = data.pop('Response')
# создаем экземпляр класса GradientBoostingClassifier
boost = GradientBoostingClassifier(random_state=46)
# выполняем перекрестную проверку и сохраняем результат
# с помощью функции cross_validate()
output = cross_validate(boost,
                        data,
                        y,
                        cv=5,
                        return_estimator=True)
```

Создаем список, в который будем сохранять важности признаков, и сохраняем в него важности, полученные для каждой из моделей.

```
# создаем список fi, в который будем сохранять
# важности признаков, и сохраняем в него важности,
# рассчитанные для каждой из моделей
fi = []
for estimator in output['estimator']:
    fi.append(estimator.feature_importances_)
```

Преобразовываем список в датафрейм, в котором индексы – имена наших переменных, а столбцы – значения важностей, полученные на определенной итерации перекрестной проверки.

```
# преобразовываем список в датафрейм, индексы в котором
# будут именами наших переменных
fi = pd.DataFrame(
    np.array(fi).T,
    columns=['importance ' + str(idx) for idx in range(len(fi))],
    index=data.columns)
fi
```

	importance 0	importance 1	importance 2	importance 3	importance 4
<b>Customer Lifetime Value</b>	0.255318	0.234372	0.269660	0.203805	0.161001
<b>Income</b>	0.348616	0.332639	0.296688	0.313324	0.361096
<b>Monthly Premium Auto</b>	0.139044	0.154778	0.167606	0.142860	0.183078
<b>Months Since Last Claim</b>	0.073406	0.084058	0.090921	0.116752	0.092812
<b>Months Since Policy Inception</b>	0.125063	0.121127	0.114898	0.142917	0.115305
<b>Number of Open Complaints</b>	0.025870	0.030423	0.026967	0.031941	0.038452
<b>Number of Policies</b>	0.032684	0.042602	0.033259	0.048403	0.048257

Наконец, получаем усредненные важности признаков и выводим в порядке убывания.

```
# получаем усредненные важности
# и выводим в порядке убывания
fi['mean_importance'] = fi.mean(axis=1)
fi.sort_values('mean_importance', ascending=False)
```

	importance 0	importance 1	importance 2	importance 3	importance 4	mean_importance
<b>Income</b>	0.348616	0.332639	0.296688	0.313324	0.361096	0.330472
<b>Customer Lifetime Value</b>	0.255318	0.234372	0.269660	0.203805	0.161001	0.224831
<b>Monthly Premium Auto</b>	0.139044	0.154778	0.167606	0.142860	0.183078	0.157473
<b>Months Since Policy Inception</b>	0.125063	0.121127	0.114898	0.142917	0.115305	0.123862
<b>Months Since Last Claim</b>	0.073406	0.084058	0.090921	0.116752	0.092812	0.091590
<b>Number of Policies</b>	0.032684	0.042602	0.033259	0.048403	0.048257	0.041041
<b>Number of Open Complaints</b>	0.025870	0.030423	0.026967	0.031941	0.038452	0.030731

Однако вернемся к функции `cross_val_score()`. Она универсальна, потому что в параметр `cv` всегда можно передать экземпляр класса, реализующий ту или иную стратегию перекрестной проверки. Перекрестная проверка в библиотеке `scikit-learn` может быть осуществлена для данных формата «один клиент – одно наблюдение», когда клиент представлен одним наблюдением (классы без префикса `GroupKFold`), и для данных формата «один клиент – несколько наблюдений», когда клиент представлен несколькими наблюдениями, т.е. по сути является группой (классы с префиксом `GroupKFold`). Если решается задача классификации, мы можем соблюдать распределение классов в каждом блоке перекрестной проверки так, чтобы оно соответствовало распределению классов в исходном наборе (классы с префиксом `Stratified`), а можем не соблюдать (классы без префикса `Stratified`). Мы можем разбить данные на блоки, предварительно назначив каждое наблюдение определенному блоку (классы без префикса `Repeated`), а можем выполнять переназначение наблюдений блокам после определенного количества итераций (классы с префиксом `Repeated`). Все классы, кроме класса `TimeSeriesSplit`, предназначены для работы с данными, не относящимися к временным рядам.

## 7.10. ВИДЫ ПЕРЕКРЕСТНОЙ ПРОВЕРКИ ДЛЯ ДАННЫХ ФОРМАТА «ОДИН ОБЪЕКТ – ОДНО НАБЛЮДЕНИЕ» (ОТСУТСТВУЕТ ОСЬ ВРЕМЕНИ)

В этом разделе разберем виды перекрестных проверок, предназначенных для данных формата «один объект – одно наблюдение», когда отсутствует ось времени. Объектом могут быть клиент, товар, услуга.

Любой из нижеописанных видов проверки является корректным подходом только в том случае, если на основании оценки качества на тестовых выборках не производится выбор лучшей модели из множества моделей с разными гиперпараметрами.

Обратите внимание: если данные представляют собой временной ряд или множество временных рядов, мы не можем использовать эти виды проверок.

### 7.10.1. Обычная нестратифицированная $k$ -блочная перекрестная проверка с помощью класса KFold

Когда решается задача классификации, обычная  $k$ -блочная перекрестная проверка делится на стратифицированную и нестратифицированную. В стратифицированной перекрестной проверке мы разбиваем данные на блоки таким образом, чтобы пропорции классов в каждом блоке в точности соответствовали пропорциям классов в наборе данных. Например, если 90 % примеров относятся к классу А, а 10 % примеров – к классу В, то стратифицированная перекрестная проверка выполняется так, чтобы в каждом блоке 90 % примеров принадлежали к классу А, а 10 % примеров – к классу В.

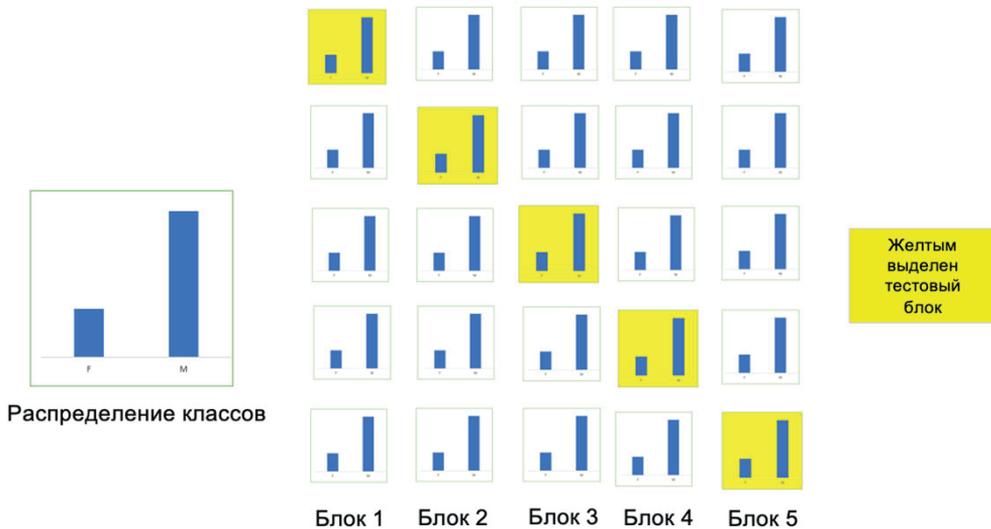


Рис. 36 Распределение классов в стратифицированной перекрестной проверке

Использование для оценки классификатора стратифицированной  $k$ -блочной перекрестной проверки вместо обычной  $k$ -блочной перекрестной является хорошей идеей, поскольку позволяет получить более надежные оценки обобщающей способности. В ситуации, когда лишь 10 % примеров принадлежат к классу В, использование стандартной  $k$ -блочной перекрестной проверки может привести к тому, что один из блоков будет полностью состоять из примеров, относящихся к классу А. Использование этого блока в качестве тестового набора не даст особой информации о качестве работы классификатора.

Стратифицированная перекрестная проверка важна для небольших выборок, несбалансированных наборов данных, многоклассовой классификации, случаев, когда наблюдения отсортированы по метке класса и тогда в блок может попасть один определенный класс.

Обычная  $k$ -блочная перекрестная проверка может быть со случайным перемешиванием перед разбиением на блоки и без случайного перемешивания. Перемешивание также может помочь в случае с наблюдениями, упорядоченными по меткам классов.

Обычную нестратифицированную  $k$ -блочную перекрестную проверку можно выполнить, указав экземпляр класса `KFold` для параметра `cv` функции `cross_val_score()`. Ниже приведены параметры класса `KFold`.

```
from sklearn.model_selection import KFold(n_splits=5, ← Количество итераций
                                             (по умолчанию 5, должно быть не меньше 2)
                                             shuffle=False, ← Случайное перемешивание данных
                                                             перед разбиением на блоки
                                             random_state=None) ← Стартовое значение генератора случайных
                                                                    чисел для воспроизводимости результатов
```

Импортируем необходимые библиотеки, классы и функции и проиллюстрируем работу класса `KFold`.

```
# импортируем необходимые библиотеки
import numpy as np
import pandas as pd
# импортируем наборы
from sklearn import datasets
# импортируем функцию train_test_split() и cross_val_score(),
# классы KFold, StratifiedKFold, RepeatedKFold,
# RepeatedStratifiedKFold, LeaveOneOut, LeavePOut, GroupKFold,
# LeaveOneGroupOut, LeavePGroupsOut, StratifiedGroupKFold
from sklearn.model_selection import (train_test_split,
                                     cross_val_score,
                                     KFold,
                                     StratifiedKFold,
                                     RepeatedKFold,
                                     RepeatedStratifiedKFold,
                                     ShuffleSplit,
                                     LeaveOneOut,
                                     LeavePOut,
                                     GroupKFold,
                                     LeaveOneGroupOut,
                                     LeavePGroupsOut,
                                     StratifiedGroupKFold)

# импортируем класс DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
```

Создадим экземпляр класса `KFold`. У нас будет 5 итераций и перемешивание данных перед разбиением на блоки.

```
# создаем экземпляр класса KFold
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
```

Теперь создадим игрушечные массив признаков и массив меток и взглянем на индексы наблюдений, попавших в обучающую и тестовую выборки, по каждой из 5 итераций.

```
# создаем игрушечные массив признаков и массив меток
X_toy = [0.1, 0.2, 2.2, 2.4, 2.3, 2.55, 2.8, 5.8, 2.9, 1.3]
y_toy = [0, 0, 1, 0, 1, 0, 0, 1, 0, 0]
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из 5 итераций
for train_index, test_index in kfold.split(X_toy, y_toy):
    print("TRAIN:", train_index, "TEST:", test_index)
```







2-я итерация

```
TRAIN:
{0: 32.0, 1: 33.0, 2: 35.0}
TEST:
{0: 36.0, 1: 34.0, 2: 30.0}
```

3-я итерация

```
TRAIN:
{0: 37.0, 1: 32.0, 2: 31.0}
TEST:
{0: 26.0, 1: 36.0, 2: 38.0}
```

Мы видим *разное* распределение классов в обучающей и тестовой выборках при каждом разбиении данных, и это распределение отличается от распределения классов в исходном наборе. Все это непременно ухудшит качество модели машинного обучения.

Теперь возьмем задачу кредитного скоринга – задачу 2-классовой классификации с гораздо большим количеством наблюдений.

```
# считываем данные
data = pd.read_csv('Data/cs-training.csv',
                  index_col=0)
# заменяем пропуски
data.fillna(0, inplace=True)
# создаем массив меток и массив признаков
labels = data.pop('SeriousDlqin2yrs')
```

Взглянем на количество наблюдений и распределение классов.

```
# посмотрим количество наблюдений
# и распределение классов
print(len(data))
print("")
print(labels.value_counts(normalize=True))
```

150000

```
0    0.93316
1    0.06684
```

Name: SeriousDlqin2yrs, dtype: float64

Выполним обычную нестратифицированную 3-блочную перекрестную проверку с перемешиванием и взглянем на распределение классов в обучающей и тестовой выборках по каждой из трех итераций.

```
# взглянем на распределение классов в обучающей
# и тестовой выборках по каждой из трех итераций
for cnt, (train_index, test_index) in enumerate(
    kfold.split(data, labels), 1):
    print(f"\n{cnt}-я итерация")
    train_pcnt = round(labels.iloc[train_index].value_counts(
        normalize=True), 3)
```

```
print(f"\nTRAIN:\n{train_pcnt}\n")
test_pcnt = round(labels.iloc[test_index].value_counts(
    normalize=True), 3)
print(f"TEST:\n{test_pcnt}")
```

1-я итерация

```
TRAIN:
0    0.933
1    0.067
Name: SeriousDlqin2yrs, dtype: float64
```

```
TEST:
0    0.934
1    0.066
Name: SeriousDlqin2yrs, dtype: float64
```

2-я итерация

```
TRAIN:
0    0.933
1    0.067
Name: SeriousDlqin2yrs, dtype: float64
```

```
TEST:
0    0.933
1    0.067
Name: SeriousDlqin2yrs, dtype: float64
```

3-я итерация

```
TRAIN:
0    0.934
1    0.066
Name: SeriousDlqin2yrs, dtype: float64
```

```
TEST:
0    0.932
1    0.068
Name: SeriousDlqin2yrs, dtype: float64
```

В случае с задачей бинарной классификации с большим количеством наблюдений ситуация выглядит иначе: мы видим *примерно одинаковое* распределение классов в обучающем и тестовом блоках при каждом разбиении данных.

Давайте применительно к этой задаче вычислим правильность, усредненную по тестовым блокам обычной перекрестной проверки без стратификации. В качестве модели возьмем дерево решений. Для этого в функцию `cross_val_score()` передадим `kfold` – экземпляр класса `KFold`.

```
# создаем экземпляр класса DecisionTreeClassifier
tree = DecisionTreeClassifier(random_state=152)
```

```
# вычисляем значение правильности, усредненное по тестовым
# блокам обычной нестратифицированной перекрестной проверки
```

```
scores_acc = cross_val_score(tree,
                             data,
                             labels,
                             cv=kfold)

print("Среднее значение правильности: {:.2f}".format(
    scores_acc.mean()))
```

Среднее значение правильности: 0.90

## 7.10.2. Обычная стратифицированная $k$ -блочная перекрестная проверка с помощью класса `StratifiedKFold`

Обычную  $k$ -блочную перекрестную проверку со стратификацией можно выполнить, указав экземпляр класса `StratifiedKFold` для параметра `cv` функции `cross_val_score()`. Ниже приведены параметры класса `StratifiedKFold`.

```
from sklearn.model_selection import StratifiedKFold(n_splits=5, ← Количество итераций
                                                    (по умолчанию 5, должно быть не меньше 2)
                                                    shuffle=False, ← Случайное перемешивание данных
                                                    random_state=None) ← перед разбиением на блоки
                                                    ← Стартовое значение генератора случайных чисел
                                                    для воспроизводимости результатов
```

Давайте применим обычную стратифицированную 3-блочную перекрестную проверку с перемешиванием к набору с данными об ирисах Фишера.

```
# создаем экземпляр класса StratifiedKFold
stratfold = StratifiedKFold(n_splits=3,
                             shuffle=True,
                             random_state=42)

# взглянем на распределение классов в обучающей
# и тестовой выборках по каждой из трех итераций
for cnt, (train_index, test_index) in enumerate(
    stratfold.split(X, y), 1):
    print(f"\n{cnt}-я итерация")
    uniques, counts = np.unique(y[train_index], return_counts=True)
    train_pcnt = dict(zip(uniques, counts * 100 / len(y[train_index])))
    uniques, counts = np.unique(y[test_index], return_counts=True)
    test_pcnt = dict(zip(uniques, counts * 100 / len(y[test_index])))
    print("\nTRAIN:\n", train_pcnt)
    print("TEST:\n", test_pcnt)
```

1-я итерация

```
TRAIN:
{0: 33.0, 1: 33.0, 2: 34.0}
TEST:
{0: 34.0, 1: 34.0, 2: 32.0}
```

2-я итерация

```
TRAIN:
{0: 33.0, 1: 34.0, 2: 33.0}
TEST:
{0: 34.0, 1: 32.0, 2: 34.0}
```

3-я итерация

```
TRAIN:
{0: 34.0, 1: 33.0, 2: 33.0}
TEST:
{0: 32.0, 1: 34.0, 2: 34.0}
```

1-я итерация

```
TRAIN:
{0: 31.0, 1: 35.0, 2: 34.0}
TEST:
{0: 38.0, 1: 30.0, 2: 32.0}
```

2-я итерация

```
TRAIN:
{0: 32.0, 1: 33.0, 2: 35.0}
TEST:
{0: 36.0, 1: 34.0, 2: 30.0}
```

3-я итерация

```
TRAIN:
{0: 37.0, 1: 32.0, 2: 31.0}
TEST:
{0: 26.0, 1: 36.0, 2: 38.0}
```

Классы все равно распределены неравномерно, но разброс стал намного меньше (распределение классов для нестратифицированной проверки с перемешиванием показано справа серым цветом).

Теперь применим обычную стратифицированную 3-блочную перекрестную проверку с перемешиванием к задаче кредитного скоринга.

```
# взглянем на распределение классов в обучающей
# и тестовой выборках по каждой из трех итераций
for cnt, (train_index, test_index) in enumerate(
    stratfold.split(data, labels), 1):
    print(f"\n{cnt}-я итерация")
    train_pcmt = round(labels.iloc[train_index].value_counts(
        normalize=True), 3)
    print(f"\nTRAIN:\n{train_pcmt}\n")
    test_pcmt = round(labels.iloc[test_index].value_counts(
        normalize=True), 3)
    print(f"TEST:\n{test_pcmt}")
```

1-я итерация

```
TRAIN:
0    0.933
1    0.067
Name: SeriousDlqin2yrs, dtype: float64
```

```
TEST:
0    0.933
1    0.067
Name: SeriousDlqin2yrs, dtype: float64
```

2-я итерация

```
TRAIN:
0    0.933
```

```
1 0.067
Name: SeriousDlqin2yrs, dtype: float64
```

```
TEST:
```

```
0 0.933
1 0.067
Name: SeriousDlqin2yrs, dtype: float64
```

3-я итерация

```
TRAIN:
```

```
0 0.933
1 0.067
Name: SeriousDlqin2yrs, dtype: float64
```

```
TEST:
```

```
0 0.933
1 0.067
Name: SeriousDlqin2yrs, dtype: float64
```

Мы видим *одинаковое* распределение классов в обучающей и тестовой выборках при каждом разбиении данных.

Давайте вычислим правильность, усредненную по тестовым выборкам обычной перекрестной проверки со стратификацией. Опять в качестве модели возьмем дерево решений. Для этого в функцию `cross_val_score()` передадим `stratfold` – экземпляр класса `StratifiedKFold`.

```
# вычисляем значение правильности, усредненное по тестовым
# выборкам обычной стратифицированной перекрестной проверки
scores_acc = cross_val_score(tree,
                              data,
                              labels,
                              cv=stratfold)
print("Среднее значение правильности: {:.2f}".format(
    scores_acc.mean()))
```

Среднее значение правильности: 0.90

### 7.10.3. Повторная нестратифицированная *k*-блочная перекрестная проверка с помощью класса `RepeatedKFold`

Еще один часто используемый метод перекрестной проверки – повторная перекрестная проверка (*repeated cross-validation*). В рамках каждого повтора мы по-разному относим наблюдения к блокам.

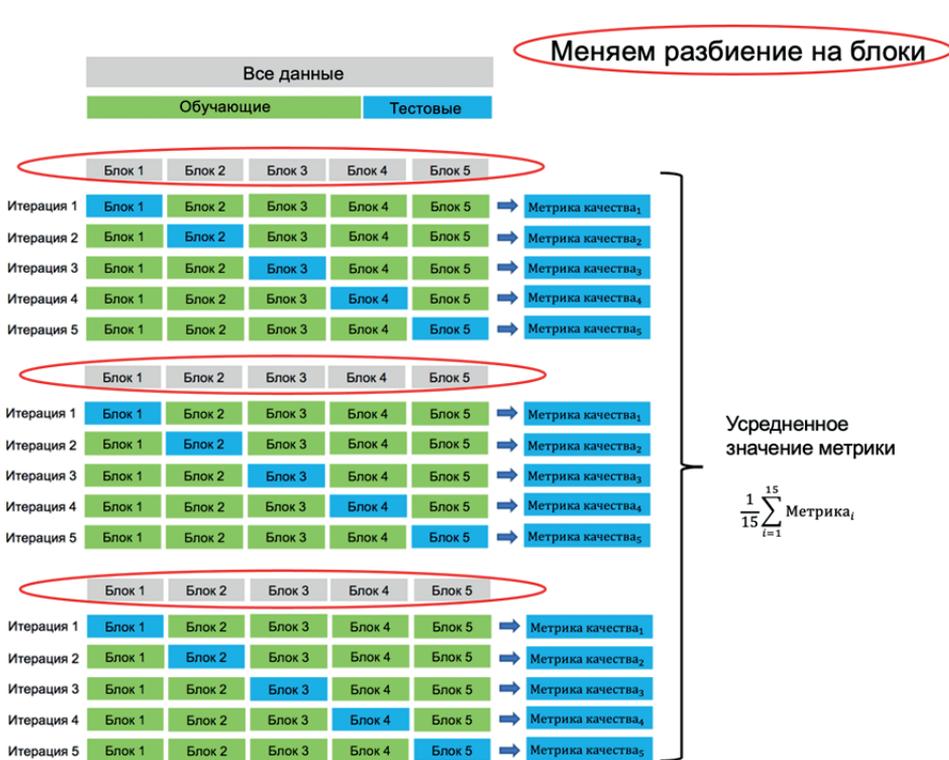


Рис. 37 Повторная перекрестная проверка

Допустим, у нас 10 наблюдений, и мы применяем обычную 3-блочную перекрестную проверку. В блок 1 попадут наблюдения с индексами 0, 1, 5, 8 (выделены синим), в блок 2 попадут наблюдения с индексами 2, 7, 9 (выделены красным), в блок 3 попадут наблюдения с индексами 3, 4, 6 (выделены фиолетовым). В первой итерации блок 2 и блок 3 сформируют обучающий блок, а блок 1 станет тестовым блоком. Во второй итерации блок 1 и блок 3 сформируют обучающий блок, а блок 2 станет тестовым блоком. В третьей итерации блок 1 и блок 2 сформируют обучающий блок, а блок 3 станет тестовым блоком.

```
# создаем экземпляр класса KFold
kfold = KFold(n_splits=3, shuffle=True, random_state=42)
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train_index, test_index in kfold.split(X_toy, y_toy):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [2 3 4 6 7 9] TEST: [0 1 5 8]
TRAIN: [0 1 3 4 5 6 8] TEST: [2 7 9]
TRAIN: [0 1 2 5 7 8 9] TEST: [3 4 6]
```

Однако наблюдения как принадлежали конкретному блоку, так и будут ему принадлежать. Теперь по-другому разнесем наблюдения по блокам. Это будет

второй повтор. В блок 1 попадут наблюдения с индексами 0, 1, 5, 8 (выделены синим), в блок 2 попадут наблюдения с индексами 3, 4, 7 (выделены красным), в блок 3 попадут наблюдения с индексами 2, 6, 9 (выделены фиолетовым). В первой итерации блок 2 и блок 3 сформируют обучающий блок, а блок 1 станет тестовым блоком. Во второй итерации блок 1 и блок 3 сформируют обучающий блок, а блок 2 станет тестовым блоком. В третьей итерации блок 1 и блок 2 сформируют обучающий блок, а блок 3 станет тестовым блоком.

```
# создаем экземпляр класса RepeatedKFold
rkf = RepeatedKFold(n_splits=3, n_repeats=2, random_state=42)
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из двух итераций двух повторов
for train_index, test_index in rkf.split(X_toy, y_toy):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [2 3 4 6 7 9] TEST: [0 1 5 8]
TRAIN: [0 1 3 4 5 6 8] TEST: [2 7 9]
TRAIN: [0 1 2 5 7 8 9] TEST: [3 4 6]
TRAIN: [2 3 4 6 7 9] TEST: [0 1 5 8]
TRAIN: [0 1 2 5 6 8 9] TEST: [3 4 7]
TRAIN: [0 1 3 4 5 7 8] TEST: [2 6 9]
```

} 1-й повтор    блок 1  
                  блок 2  
                  блок 3  
} 2-й повтор

Таким образом, с помощью повторной перекрестной проверки мы моделируем большую изменчивость данных, чем при обычной перекрестной проверке, поэтому для больших наборов данных повторная перекрестная проверка может дать более реалистичные результаты.

Повторную  $k$ -блочную перекрестную проверку без стратификации можно выполнить, указав экземпляр класса `RepeatedKFold` для параметра `cv` функции `cross_val_score()`. Ниже приведены параметры класса `RepeatedKFold`.

```
from sklearn.model_selection import RepeatedKFold(n_splits=5, ← Количество итераций
                                                    (по умолчанию 5, должно быть не меньше 2)
                                                    n_repeats=10, ← Количество повторов (случайных перемешиваний
                                                    данных перед разбиением на блоки)
                                                    random_state=None) ← Стартовое значение генератора случайных чисел
                                                    для воспроизводимости результатов
```

Давайте вычислим правильность, усредненную по тестовым выборкам повторной перекрестной проверки без стратификации.

```
# создаем экземпляр класса RepeatedKFold
rkf = RepeatedKFold(n_splits=5,
                    n_repeats=2,
                    random_state=42)

# вычисляем значение правильности, усредненное по
# тестовым выборкам повторной нестратифицированной
# перекрестной проверки
scores_acc = cross_val_score(tree,
                              data,
                              labels,
                              cv=rkf)

print("Среднее значение правильности: {:.2f}".format(
    scores_acc.mean()))
```

Среднее значение правильности: 0.90

## 7.10.4. Повторная стратифицированная $k$ -блочная перекрестная проверка с помощью класса `RepeatedStratifiedKFold`

Повторную  $k$ -блочную перекрестную проверку со стратификацией можно выполнить, указав экземпляр класса `RepeatedStratifiedKFold` для параметра `cv` функции `cross_val_score()`.

```
from sklearn.model_selection import RepeatedStratifiedKFold(
    n_splits=5, ←———— Количество итераций
                    (по умолчанию 5, должно быть не меньше 2)
    n_repeats=10, ←———— Количество повторов (случайных перемешиваний
                    данных перед разбиением на блоки)
    random_state=None) ←— Стартовое значение генератора случайных чисел
                    для воспроизводимости результатов
```

Давайте вычислим правильность, усредненную по тестовым выборкам повторной перекрестной проверки со стратификацией.

```
# создаем экземпляр класса RepeatedStratifiedKFold
rskf = RepeatedStratifiedKFold(n_splits=5,
                               n_repeats=2,
                               random_state=42)

# вычисляем значение правильности, усредненное по
# тестовым выборкам повторной стратифицированной
# перекрестной проверки
scores_acc = cross_val_score(tree,
                              data,
                              labels,
                              cv=rskf)

print("Среднее значение правильности: {:.2f}".format(
    scores_acc.mean()))
```

Среднее значение правильности: 0.90

Итоговая оценка обобщающей способности, получаемая с помощью перекрестной проверки, характеризуется смещением и дисперсией, которое связано с процентом данных, используемых для проверки. Если вы отложили для проверки 50 % данных, используя 2-блочную перекрестную проверку, ваша итоговая оценка будет более смещенной, чем итоговая оценка, когда вы откладывали для проверки 10 % данных, используя 10-блочную перекрестную проверку. С другой стороны, общепринятое мнение состоит в том, что применение меньшего количества данных для проверки увеличивает дисперсию оценки обобщающей способности, поскольку каждый тестовый блок включает меньшее количество данных, не позволяющее получить устойчивую оценку обобщающей способности (например, RMSE).

Макс Кун провел ряд симуляций, чтобы оценить смещение и дисперсию итоговых оценок метрик качества при использовании различных методов перекрестной проверки (обычная  $k$ -блочная перекрестная проверка, повторная  $k$ -блочная перекрестная проверка, перекрестная проверка по методу

Монте-Карло). Он сгенерировал синтетические данные для задачи регрессии (при этом были известны правильные ответы, чтобы можно было вычислить оценку RMSE). В качестве модели он использовал случайный лес, состоящий из 1000 деревьев, со значениями гиперпараметров по умолчанию. Он сгенерировал 100 различных наборов данных с 500 обучающими наблюдениями. Для каждого набора данных применял каждый метод перекрестной проверки 25 раз, используя разные стартовые значения генератора случайных чисел. В итоге вычислил медианное смещение и медианную дисперсию по каждому методу перекрестной проверки.

Во-первых, давайте посмотрим, как точность измерения метрики изменяется в зависимости от количества наблюдений, откладываемых для проверки, и размера обучающего набора. Мы измеряем дисперсию оценок для различных видов перекрестной проверки.

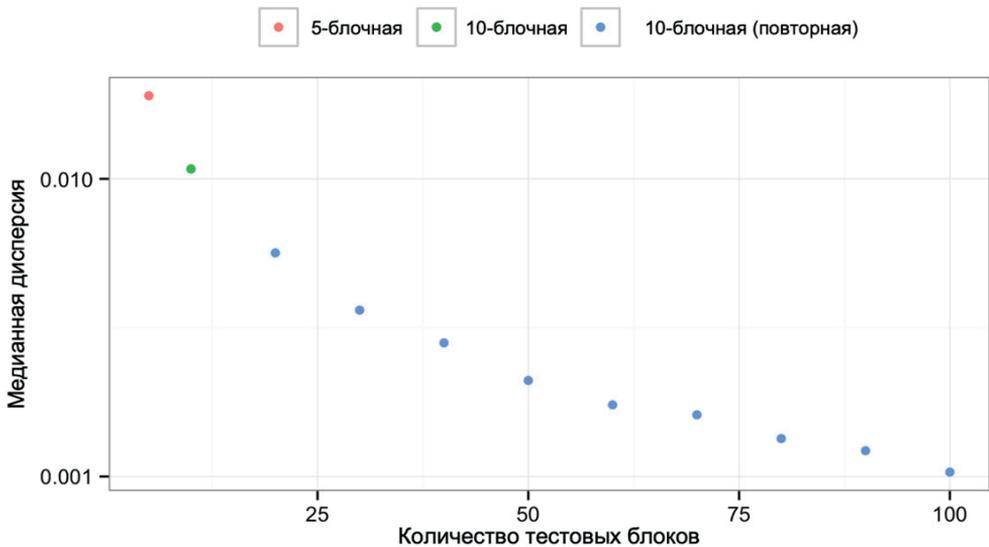


Рис. 38 Медианная дисперсия для разных видов перекрестной проверки

Значение 5 на оси  $x$  соответствует 5-блочной перекрестной проверке, а 10 – 10-блочной перекрестной проверке. Значения больше 10 соответствуют повторной 10-блочной перекрестной проверке (т.е. 20 – это 2-повторная 10-блочная перекрестная проверка, 60 – это 6-повторная 10-блочная перекрестная проверка). В левой части графика (то есть для 5-блочной перекрестной проверки) медианная дисперсия составляет 0,019. Она показывает, как будет варьировать итоговая оценка 5-блочной перекрестной проверки по всем сгенерированным наборам данных.

Результат ожидаем: измеренная дисперсия снижается по мере увеличения повторов 10-блочной перекрестной проверки. В какой-то момент дисперсия выравнивается, но мы все еще можем улучшить точность измерения метрики, повторяя 10-блочную перекрестную проверку более одного раза.

Если посмотреть на первые две точки данных (однократную 5-блочную и 10-блочную перекрестные проверки), то уменьшение дисперсии, вероят-

но, связано с объемом данных, отложенным для проверки (20 % против 10 %), и количеством блоков (5 против 10).

Теперь взглянем на изменение смещения. Общепринято считать, что смещение должно быть меньше в случае применения 10-блочной перекрестной проверки, поскольку в таком случае мы для проверки откладываем меньшее количество наблюдений.

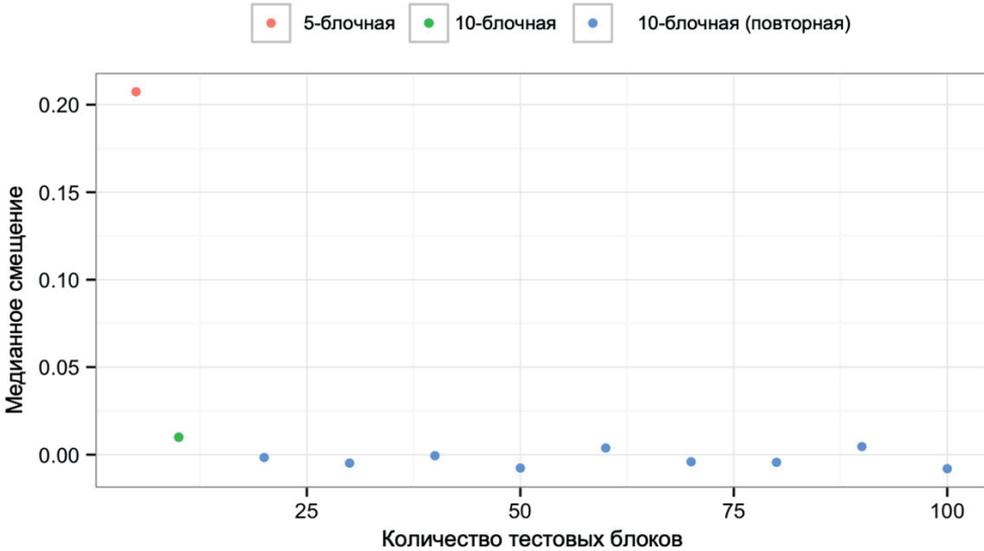


Рис. 39 Медианное смещение для разных видов перекрестной проверки

На рисунке видим, что 5-блочная перекрестная проверка демонстрирует пессимистичное смещение, и смещение снижается, когда мы переходим к 10-блочной перекрестной проверке. Возможно, речь идет о шуме, но похоже, что повторение 10-блочной перекрестной проверки несколько раз может также незначительно уменьшить смещение.

### 7.10.5. $k$ -кратное случайное разбиение на обучающую и тестовую выборки (перекрестная проверка Монте-Карло)

Одним из способов получения более надежной оценки, которая будет в меньшей степени зависеть от того, как вы разобьете набор на обучающую и тестовую выборки, является метод  $k$ -кратного разбиения на обучающую и тестовую выборки с разными стартовыми значениями генератора псевдослучайных чисел с последующим вычислением метрики качества, усредненной по этим  $k$  итерациям.

Этот метод многократного разбиения на обучающую и тестовую выборки, иногда называемый перекрестной проверкой по методу Монте-Карло (Monte Carlo cross-validation), дает более надежную оценку качества модели на случайном тестовом наборе данных по сравнению с обычным методом однократного разбиения на обучающую и тестовую выборки. Кроме того, он дает информацию о стабильности модели, то есть о том, как сильно модель, обученная с помощью алгоритма, меняется при использовании разных разбиений набора данных.

```

# пишем функцию, выполняющую перекрестную проверку
# по методу Монте-Карло
def monte_carlo_crossvalidation(X,
                                y,
                                n_splits=10,
                                test_size=0.3):
    """
    Выполняет перекрестную проверку
    по методу Монте-Карло

    Параметры
    -----
    estimator: экземпляр класса
        Модель машинного обучения.
    X: pandas.DataFrame
        Массив признаков.
    y: pandas.Series
        Массив меток.
    n_splits: int, по умолчанию 10
        Количество случайных разбиений
        на обучающую и тестовую выборки
    test_size: float, по умолчанию 0.3
        Размер тестовой выборки.
    """
    # создадим пустой список, в котором будем
    # записывать значения правильности
    accuracy_list = []
    # выполняем n_split раз случайное разбиение на обучающую
    # и тестовую выборки (размер тестовой выборки определяется
    # с помощью test_size) и печатаем значение правильности
    # для обучающей и тестовой выборок в каждой итерации
    for i in range(n_splits):
        train, test, y_train, y_test = train_test_split(
            X,
            y,
            random_state=i,
            test_size=test_size)
        print("train=%d, test=%d" % (len(train), len(test)))
        estimator.fit(train, y_train)
        train_score = estimator.score(train, y_train)
        test_score = estimator.score(test, y_test)
        print("Правильность на обучающей выборке: {:.3f}".format(
            train_score))
        print("Правильность на тестовой выборке: {:.3f}".format(
            test_score))
        print("")
        accuracy_list.append(test_score)
    # печатаем среднее значение правильности
    print("Среднее значение правильности: {:.3f}".format(
        sum(accuracy_list) / len(accuracy_list)))

```

Применим нашу функцию к задаче кредитного скоринга. Выполним 10 случайных разбиений на обучающую и тестовую выборки, при этом для тестовой выборки будем откладывать 30 % наблюдений.

*# применяем нашу функцию*

```
monte_carlo_crossvalidation(  
    data,  
    labels,  
    n_split=10,  
    test_size=0.3)
```

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.896

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.900

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.897

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.900

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.897

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.896

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.898

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.899

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

Правильность на тестовой выборке: 0.897

```
train=105000, test=45000
```

Правильность на обучающей выборке: 1.000

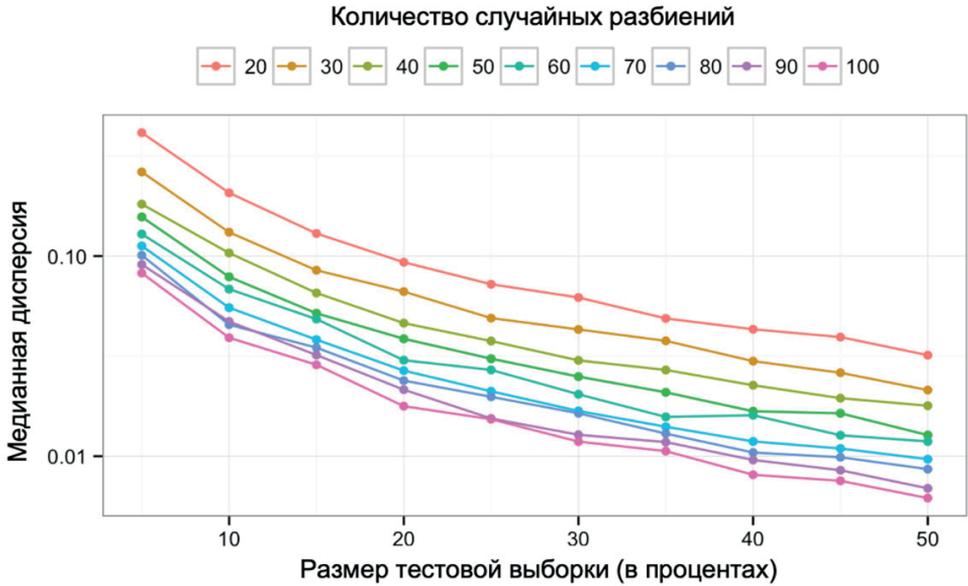
Правильность на тестовой выборке: 0.900

Среднее значение правильности: 0.898

В своих экспериментах Макс Кун также попытался ответить еще на два вопроса:

- как оценка, полученная с помощью перекрестной проверки по методу Монте-Карло, зависит от количества наблюдений, откладываемых для проверки?
- имеет ли перекрестная проверка по методу Монте-Карло преимущества перед  $k$ -блочной перекрестной проверкой?

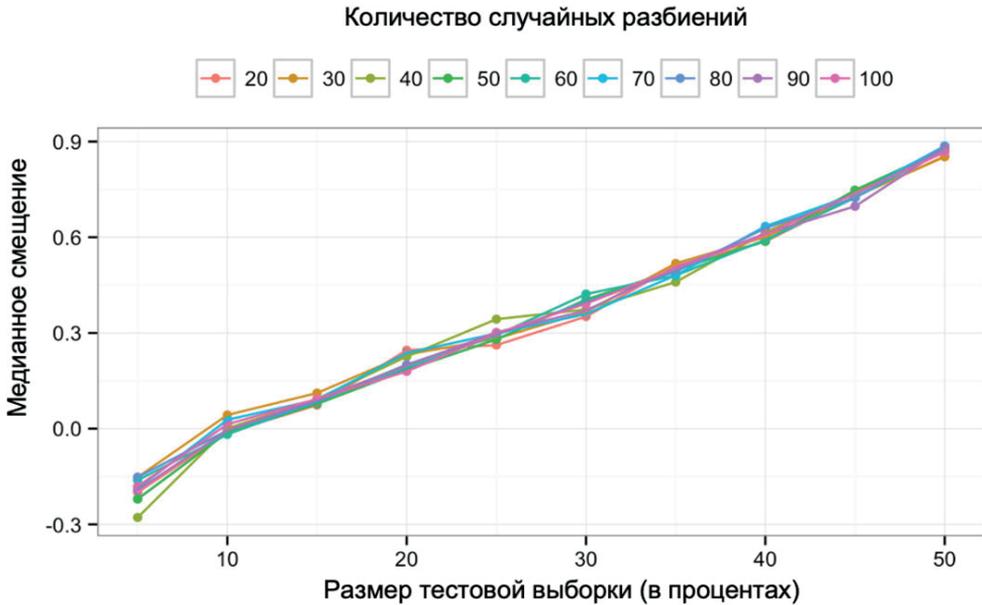
Анализируя дисперсию для перекрестной проверки по методу Монте-Карло, Макс Кун обнаружил интересную закономерность.



**Рис. 40** Зависимость медианной дисперсии от количества случайных разбиений и размера тестовой выборки

Как видим, размер тестовой выборки имеет немного большее влияние на дисперсию, чем количество случайных разбиений. Большой размер тестовой выборки обеспечивает меньшую дисперсию.

Теперь рассмотрим смещение.



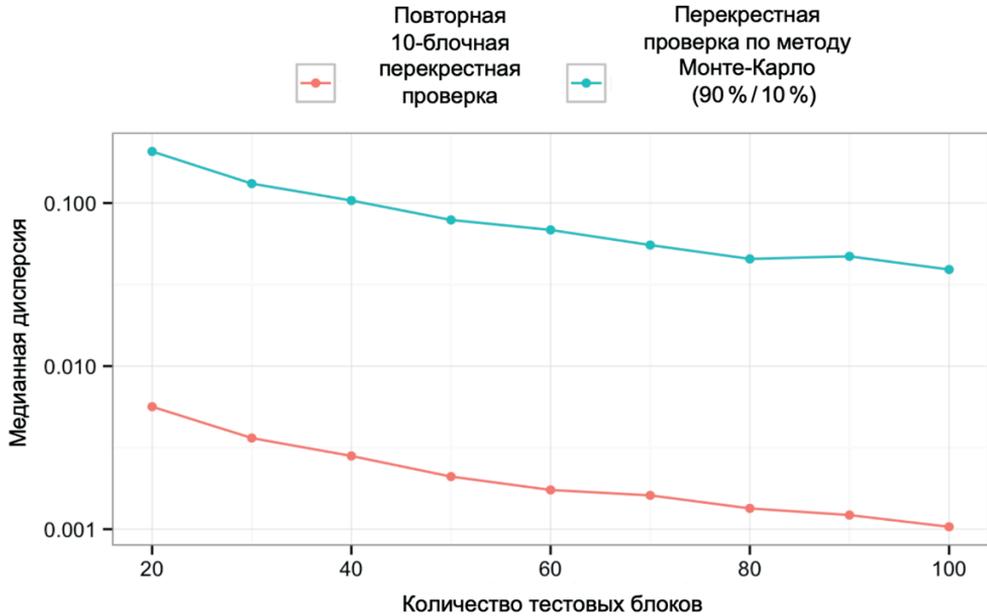
**Рис. 41** Зависимость медианного смещения от количества случайных разбиений и размера тестовой выборки

Как видим, перекрестная проверка по методу Монте-Карло дает чрезмерно пессимистические оценки при увеличении размера тестовой выборки. Причиной этому может служить тот факт, что при уменьшении обучающей выборки случайный лес создает менее эффективные модели.

Следует также отметить, что количество случайных разбиений не оказывает существенного влияния на смещение.

Таким образом, для перекрестной проверки по методу Монте-Карло можно порекомендовать использовать тестовую выборку малого размера (например, 10 %) и выполнять большое количество случайных разбиений, чтобы уменьшить дисперсию. Здесь возникает вопрос: почему же тогда просто не воспользоваться повторной 10-блочной перекрестной проверкой?

Для сравнения перекрестной проверки по методу Монте-Карло и повторной перекрестной проверки рассмотрим результаты тестов, в которых оба метода используют тестовую выборку размером 10 %. На рисунке внизу представлены кривые дисперсии.

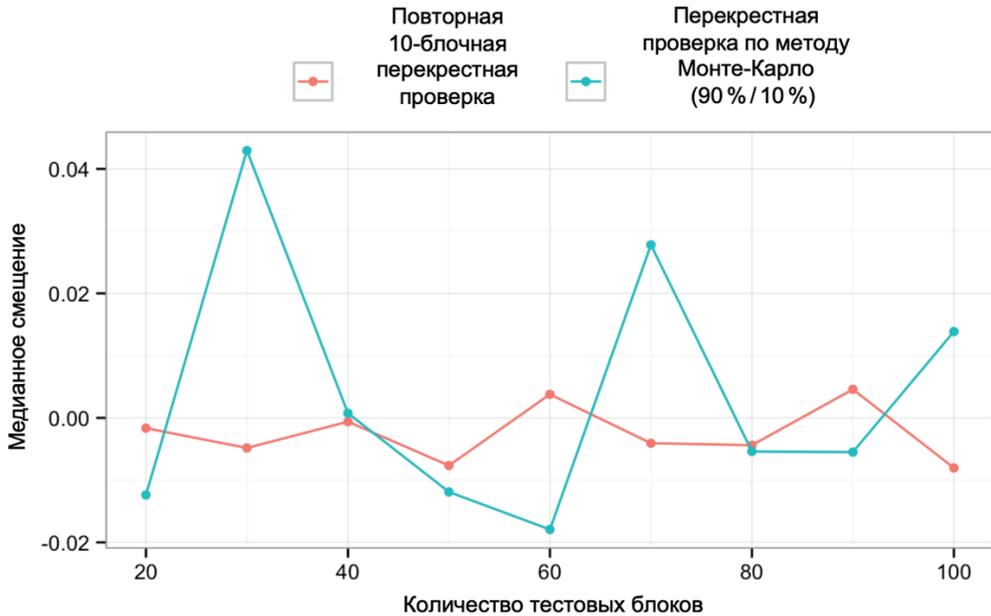


**Рис. 42** Зависимость медианной дисперсии от количества тестовых блоков для повторной 10-блочной перекрестной проверки и перекрестной проверки по методу Монте-Карло (90 % / 10 %)

Значение 20 на оси  $x$  соответствует 2-повторной 10-блочной перекрестной проверке или перекрестной проверке по методу Монте-Карло с 20 разбиениями 90 % / 10 %, а 40 – 4-повторной 10-блочной перекрестной проверке или перекрестной проверке по методу Монте-Карло с 40 разбиениями 90 % / 10 %.

Результат вполне определенный: повторная 10-блочная перекрестная проверка обеспечивает на порядок меньшую дисперсию (при прочих равных условиях).

Теперь посмотрим смещение.



**Рис. 43** Зависимость медианного смещения от количества тестовых блоков для повторной 10-блочной перекрестной проверки и перекрестной проверки по методу Монте-Карло (90 % / 10 %)

Кривые смещения содержат существенное количество шума. Здесь нет существенной разницы в смещении, и этот результат был ожидаем. Размер тестовой выборки для обоих методов составлял 10 %, соответственно, если смещение зависит от этой величины, значит, оба метода должны давать примерно одинаковые результаты.

Общий вывод заключается в том, что повторная 10-блочная перекрестная проверка является лучшим методом проверки с точки зрения минимизации дисперсии и смещения. Как всегда, есть оговорки. При наличии достаточно большого объема данных дисперсия и смещение оценки, получаемой при 10- или даже 5-блочной перекрестной проверке, вполне могут быть приемлемыми.

### 7.10.6. Перекрестная проверка со случайными перестановками при разбиении с помощью класса `ShuffleSplit`

В этом виде проверки каждое из `n_splits` разбиений отбирает `train_size` наблюдений для обучающей выборки и `test_size` наблюдений для тестовой выборки. Чем она отличается от  $k$ -блочной перекрестной проверки? В  $k$ -блочной перекрестной проверке в самом начале происходит разбиение данных на блоки, и каждое наблюдение в течение всей процедуры перекрестной проверки принадлежит одному и тому же блоку. В перекрестной проверке со случайными перестановками принадлежность наблюдения к обучающей или тестовой

выборке меняется от разбиения к разбиению. Для `train_size` и `test_size` можно задать целочисленные значения для указания абсолютных размеров выборок или значения с плавающей точкой для указания долей от общей выборки. Метод позволяет использовать на каждой итерации лишь часть данных (значения `train_size` и `test_size` необязательно должны в сумме давать 1), моделируя прореживание, что может быть полезно при работе с большими наборами данных.

Перекрестную проверку со случайными перестановками при разбиении можно выполнить, указав экземпляр класса `ShuffleSplit` для параметра `cv` функции `cross_val_score()`. Ниже приводятся параметры класса `ShuffleSplit`.

```
from sklearn.model_selection import ShuffleSplit(n_splits=10, ← Количество итераций (по умолчанию 10)
                                                test_size=None, ← Размер тестовой выборки
                                                train_size=None, ← Размер обучающей выборки
                                                random_state=None)
                                                ↙
                                                Стартовое значение генератора случайных чисел для
                                                воспроизводимости результатов
```

Проиллюстрируем данный вид проверки на игрушечных данных.

```
# создаем экземпляр класса ShuffleSplit
shuffle_split = ShuffleSplit(test_size=0.5, train_size=0.5,
                             n_splits=5, random_state=42)
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из 5 итераций
for train_index, test_index in shuffle_split.split(X_toy, y_toy):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [2 9 4 3 6] TEST: [8 1 5 0 7]
TRAIN: [4 7 9 6 2] TEST: [0 1 8 5 3]
TRAIN: [5 3 7 1 4] TEST: [9 2 0 6 8]
TRAIN: [0 3 4 5 9] TEST: [1 7 6 2 8]
TRAIN: [7 6 3 2 9] TEST: [1 5 4 8 0]
```

Видим, что принадлежность каждого наблюдения к обучающей или тестовой выборке меняется от разбиения к разбиению.

Давайте вычислим правильность, усредненную по тестовым выборкам перекрестной проверки со случайными перестановками при разбиении.

```
# создаем экземпляр класса ShuffleSplit
shflsplit = ShuffleSplit(test_size=0.3, train_size=0.7,
                         n_splits=10, random_state=42)

# вычисляем значение правильности, усредненное по
# тестовым выборкам повторной нестратифицированной
# перекрестной проверки
scores_acc = cross_val_score(tree,
                              data,
                              labels,
                              cv=shflsplit)
print("Среднее значение правильности: {:.2f}".format(
    scores_acc.mean()))
```

Среднее значение правильности: 0.90

## 7.10.7. Стратифицированная перекрестная проверка со случайными перестановками при разбиении с помощью класса `StratifiedShuffleSplit`

У обычной перекрестной проверки со случайными перестановками те же проблемы, что и у обычной *k*-блочной перекрестной проверки, распределение классов в выборках будет отличаться от распределения классов в исходном наборе, особенно это будет заметно на небольших выборках и в случаях многоклассовой классификации. Проиллюстрируем, как сработает обычная перекрестная проверка со случайными перестановками на данных об ирисах Фишера.

```
# создаем экземпляр класса ShuffleSplit
shflsplit = ShuffleSplit(test_size=0.5, train_size=0.5,
                        n_splits=3, random_state=42)

# взглянем на распределение классов в обучающей
# и тестовой выборках по каждой из трех итераций
for cnt, (train_index, test_index) in enumerate(
    shflsplit.split(X, y), 1):
    print(f"\n{cnt}-я итерация")
    uniques, counts = np.unique(y[train_index], return_counts=True)
    train_pcnt = dict(zip(uniques, counts * 100 / len(y[train_index])))
    uniques, counts = np.unique(y[test_index], return_counts=True)
    test_pcnt = dict(zip(uniques, counts * 100 / len(y[test_index])))
    print("\nTRAIN:\n", train_pcnt)
    print("TEST:\n", test_pcnt)
```

1-я итерация

```
TRAIN:
{0: 28.0, 1: 36.0, 2: 36.0}
TEST:
{0: 38.666666666666664, 1: 30.666666666666668, 2: 30.666666666666668}
```

2-я итерация

```
TRAIN:
{0: 26.666666666666668, 1: 36.0, 2: 37.333333333333336}
TEST:
{0: 40.0, 1: 30.666666666666668, 2: 29.333333333333332}
```

3-я итерация

```
TRAIN:
{0: 36.0, 1: 26.666666666666668, 2: 37.333333333333336}
TEST:
{0: 30.666666666666668, 1: 40.0, 2: 29.333333333333332}
```

Мы видим *разное* распределение классов в обучающей и тестовой выборках при каждом разбиении данных, и это распределение отличается от распределения классов в исходном наборе.

Теперь запустим в данных об ирисах Фишера стратифицированную перекрестную проверку со случайными перестановками. Ее можно реализовать с помощью класса `StratifiedShuffleSplit`.

```
# создаем экземпляр класса StratifiedShuffleSplit
stratshflsplt = StratifiedShuffleSplit(
    test_size=0.5, train_size=0.5,
    n_splits=3, random_state=42)

# взглянем на распределение классов в обучающей
# и тестовой выборках по каждой из трех итераций
for cnt, (train_index, test_index) in enumerate(
    stratshflsplt.split(X, y), 1):
    print(f"\n{cnt}-я итерация")
    uniques, counts = np.unique(y[train_index], return_counts=True)
    train_pcnt = dict(zip(uniques, counts * 100 / len(y[train_index])))
    uniques, counts = np.unique(y[test_index], return_counts=True)
    test_pcnt = dict(zip(uniques, counts * 100 / len(y[test_index])))
    print("\nTRAIN:\n", train_pcnt)
    print("\nTEST:\n", test_pcnt)
```

1-я итерация

```
TRAIN:
{0: 33.333333333333336, 1: 33.333333333333336, 2: 33.333333333333336}
TEST:
{0: 33.333333333333336, 1: 33.333333333333336, 2: 33.333333333333336}
```

2-я итерация

```
TRAIN:
{0: 33.333333333333336, 1: 33.333333333333336, 2: 33.333333333333336}
TEST:
{0: 33.333333333333336, 1: 33.333333333333336, 2: 33.333333333333336}
```

3-я итерация

```
TRAIN:
{0: 33.333333333333336, 1: 33.333333333333336, 2: 33.333333333333336}
TEST:
{0: 33.333333333333336, 1: 33.333333333333336, 2: 33.333333333333336}
```

Теперь мы получили *одинаковое* распределение классов в обучающей и тестовой выборках при каждом разбиении данных, и это распределение совпадает с распределением классов в исходном наборе.

### 7.10.8. Перекрестная проверка с исключением по одному с помощью класса LeaveOneOut

Теперь рассмотрим перекрестную проверку с исключением по одному (leave-one-out cross-validation). Перекрестную проверку с исключением по одному можно представить в виде  $k$ -блочной перекрестной проверки, в которой каждый блок представляет собой отдельное наблюдение. В каждой итерации мы выбираем одно наблюдение в качестве тестового блока. Этот вид проверки может занимать очень много времени, особенно при работе с большими наборами данных, однако иногда позволяет получить более точные оценки на небольших наборах данных.

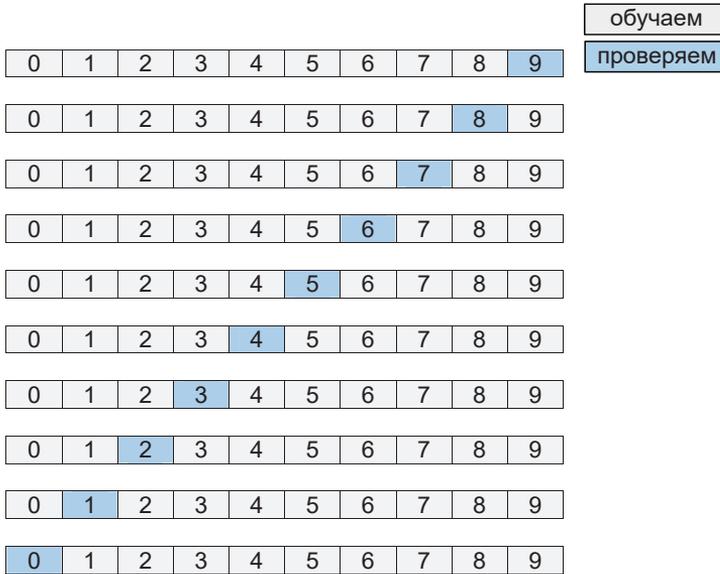


Рис. 44 Перекрестная проверка с исключением по одному

Оценка обобщающей способности, полученная с помощью перекрестной проверки с исключением по одному, является приблизительно несмещенной: пессимистичное смещение перекрестной проверки с исключением по одному ( $k = n$ ) интуитивно ниже в сравнении с  $k < n$ -блочной перекрестной проверкой, поскольку почти все  $(n - 1)$  наблюдения доступны для обучения модели. Хотя перекрестная проверка с исключением по одному дает практически несмещенную оценку, одним из недостатков ее использования по сравнению с  $k$ -блочной перекрестной проверкой при  $k < n$  является большая дисперсия оценки. Можно сказать, что «перекрестная проверка с исключением по одному имеет высокую дисперсию, потому что тестовый набор состоит лишь из одного наблюдения»<sup>4</sup>. Хасти, Тибширани и Фридман сделали следующее замечание: «при  $k = n$  оценка перекрестной проверки является приблизительно несмещенной с точки зрения фактической (ожидаемой) погрешности прогнозов, но может иметь высокую дисперсию, потому что  $n$  «обучающих выборок» очень похожи друг на друга»<sup>5</sup>.

Продемонстрируем перекрестную проверку с исключением по одному на игрушечном наборе. Мы посмотрим на индексы наблюдений, попавших в обучающую и тестовую выборки по каждой итерации. Количество итераций будет равно количеству наблюдений.

```
# создаем экземпляр класса LeaveOneOut
loo = LeaveOneOut()
```

<sup>4</sup> Tan P.-N., Steinbach M., and Kumar V. (2005). In Introduction to Data Mining. Pearson Addison Wesley, Boston.

<sup>5</sup> Hastie T., Tibshirani R., and Friedman J. H. (2009). In The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer, New York.

```
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой итерации,
# количество итераций = количество наблюдений
for train_index, test_index in loo.split(X_toy, y_toy):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [1 2 3 4 5 6 7 8 9] TEST: [0]
TRAIN: [0 2 3 4 5 6 7 8 9] TEST: [1]
TRAIN: [0 1 3 4 5 6 7 8 9] TEST: [2]
TRAIN: [0 1 2 4 5 6 7 8 9] TEST: [3]
TRAIN: [0 1 2 3 5 6 7 8 9] TEST: [4]
TRAIN: [0 1 2 3 4 6 7 8 9] TEST: [5]
TRAIN: [0 1 2 3 4 5 7 8 9] TEST: [6]
TRAIN: [0 1 2 3 4 5 6 8 9] TEST: [7]
TRAIN: [0 1 2 3 4 5 6 7 9] TEST: [8]
TRAIN: [0 1 2 3 4 5 6 7 8] TEST: [9]
```

Давайте возьмем задачу кредитного скоринга с меньшим объемом наблюдений (1500 наблюдений) и вычислим правильность, усредненную по тестовым выборкам перекрестной проверки с исключением по одному.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Bankloan.csv',
                  sep=';', decimal=',')

# создаем массив меток и массив признаков
y = data.pop('default')
X = pd.get_dummies(data)

# печатаем количество наблюдений
print(len(X))

# вычисляем значение правильности, усредненное по тестовым
# блокам перекрестной проверки с исключением по одному
scores_acc = cross_val_score(tree,
                              X,
                              y,
                              cv=loo)
print("Количество итераций cv: ", len(scores_acc))
print("Среднее значение правильности: {:.2f}".format(
    scores_acc.mean()))
```

```
Количество итераций cv: 1500
Среднее значение правильности: 0.67
```

Вновь видим, что количество итераций совпадает с количеством наблюдений, и оно далеко не маленькое.

### 7.10.9. Перекрестная проверка с исключением $p$ наблюдений с помощью класса `LeavePOut`

Перекрестную проверку с исключением  $p$  наблюдений можно представить в виде  $k$ -блочной перекрестной проверки, в которой каждый блок состоит из  $p$  отдельных наблюдений. В каждой итерации мы выбираем  $p$  отдельных наблюдений

в качестве тестового блока. Этот вид проверки также может занимать очень много времени, особенно при работе с большими наборами данных, однако иногда позволяет получить более точные оценки на небольших наборах данных.

Давайте создадим экземпляр класса `LeavePOut` с  $p$ , равным 2, и применим к нашему игрушечному набору.

```
# создаем экземпляр класса LeavePOut
lpo = LeavePOut(2)

# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой итерации
for cnt, (train_index, test_index) in enumerate(
    lpo.split(X_toy, y_toy), 1):
    print(f"\n{cnt}-я итерация")
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
1-я итерация
TRAIN: [2 3 4 5 6 7 8 9] TEST: [0 1]
```

```
2-я итерация
TRAIN: [1 3 4 5 6 7 8 9] TEST: [0 2]
```

```
3-я итерация
TRAIN: [1 2 4 5 6 7 8 9] TEST: [0 3]
```

```
4-я итерация
TRAIN: [1 2 3 5 6 7 8 9] TEST: [0 4]
```

```
. . . . .
43-я итерация
TRAIN: [0 1 2 3 4 5 6 9] TEST: [7 8]
```

```
44-я итерация
TRAIN: [0 1 2 3 4 5 6 8] TEST: [7 9]
```

```
45-я итерация
TRAIN: [0 1 2 3 4 5 6 7] TEST: [8 9]
```

Здесь у нас есть набор из 10 наблюдений, проиндексированных от 0 до 9. Мы задали  $p$  равным 2. Количество 2-значных комбинаций чисел от 0 до 9 составит  $10^2 = 100$ . Нам не нужны повторы (00, 11, ..., 99), поэтому остается 90 комбинаций. Нам не важен порядок, поэтому остается  $90/2 = 45$  комбинаций, следовательно, потребуется 45 итераций. Общей формулой вычисления количества итераций здесь будет формула числа сочетаний из  $n$  объектов по  $k$

$C_n^k = \frac{n!}{(n-k)!k!}$ . Применим ее для нашего примера:

$$C_n^k = \frac{n!}{(n-k)!k!} = \frac{10!}{8!2!} = \frac{3628800}{40320 \times 2} = \frac{3628800}{80640} = 45.$$

## 7.11. ВИДЫ ПЕРЕКРЕСТНОЙ ПРОВЕРКИ ДЛЯ ДАННЫХ ФОРМАТА «ОДИН ОБЪЕКТ – НЕСКОЛЬКО НАБЛЮДЕНИЙ» И СТРАТИФИЦИРОВАННЫХ ДАННЫХ (ОТСУТСТВУЕТ ОСЬ ВРЕМЕНИ)

В этом разделе разберем виды перекрестных проверок, предназначенных для данных формата «один объект – несколько наблюдений» и стратифицированных данных для ситуаций, когда отсутствует ось времени. Объектом могут быть клиент, товар, услуга.

Вновь сделаем несколько предостережений.

Любой из нижеописанных видов проверки является корректным подходом только в том случае, если на основании оценки качества на тестовых выборках не производится выбор лучшей модели из множества моделей с разными гиперпараметрами.

Обратите внимание: если данные представляют собой временной ряд или множество временных рядов, мы не можем использовать эти виды проверок.

### 7.11.1. Перекрестная проверка, учитывающая группы связанных наблюдений, с помощью классов GroupKFold

С помощью класса GroupKFold вы можете выполнить перекрестную проверку, учитывающую группы связанных наблюдений.

```
from sklearn.model_selection import GroupKFold(n_splits=5)
```

← Количество итераций  
(должно быть не меньше 2)

В ходе такой проверки на каждой итерации происходит разбиение на обучающую и тестовую выборки так, что наблюдения, принадлежащие одной и той же группе, не могут появиться в обучающей и тестовой выборках одновременно. На каждой итерации наблюдения, принадлежащие одной и той же группе, попадают либо в обучающую выборку, либо в тестовую выборку. В тестовую выборку могут попасть наблюдения нескольких групп. Количество итераций не может быть больше количества групп. При количестве итераций, равном количеству групп, класс становится эквивалентом класса LeaveOneGroupOut, когда на каждой итерации в тестовую выборку попадают только наблюдения одной группы. Выборки сбалансированы в том смысле, что количество отдельных групп является примерно одинаковым в каждой выборке. В качестве групп чаще всего выступают идентификаторы клиентов.

Этот вид перекрестной проверки применяется, когда данные содержат сильно взаимосвязанные между собой группы. Допустим, вы хотите построить бинарный классификатор, распознающий депрессию («нет депрессии» / «есть депрессия») по фотографиям лиц, и для этого вы собрали набор изображений 100 человек, в котором каждый человек сфотографирован несколько раз, чтобы зафиксировать разные ракурсы. Цель заключается в том, чтобы построить классификатор, который сможет правильно определить состояние людей, не включенных в этот набор изображений. В данном случае для оценки качества работы классификатора вы можете использовать традиционную стратифици-

рованную перекрестную проверку. Однако вполне вероятно, что фотографии одного и того же человека попадут как в обучающую, так и в тестовую выборки. По сравнению с совершенно новым лицом классификатору намного проще будет определить состояние по лицу, которое уже встречалось ему в обучающем наборе. Чтобы точно оценить способность модели обобщать результат на новые лица, необходимо убедиться в том, что обучающий и тестовый наборы содержат изображения разных людей.

Подобные группы данных часто встречаются в медицинской практике, когда у вас, возможно, есть несколько наблюдений по одному и тому же пациенту (например, несколько измерений уровня глюкозы в моче, суточного количества мочи для прогнозирования уровня глюкозы в крови или постановки статуса «не болен диабетом» / «болен диабетом»), но вы заинтересованы в обобщении результатов на новых пациентов. Аналогично в задачах распознавания речи у вас может быть несколько записей одного и того же человека, но вас интересует точность распознавания речи новых людей.

Давайте создадим игрушечные массив признаков и массив меток, массив с индексами групп.

```
# создаем игрушечные массив признаков и массив меток
X_toy = np.array([0.1, 0.2, 2.2, 2.4, 2.3,
                 4.55, 5.8, 8.8, 9, 10])
y_toy = np.array([0, 0, 0, 1, 1, 1, 0, 0, 0, 0])
# создаем идентификатор групп
groups = np.array([1, 1, 1, 2, 2, 2, 3, 3, 3, 3])
```

Взглянем на индексы групп, попавших в обучающую и тестовую выборки на каждой итерации.

```
# создаем экземпляр класса GroupKFold
gkf = GroupKFold(n_splits=3)

# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train, test in gkf.split(X_toy, y_toy, groups=groups):
    print("TRAIN:", groups[train], "TEST:", groups[test])
```

```
TRAIN: [1 1 1 2 2 2] TEST: [3 3 3 3]
TRAIN: [1 1 1 3 3 3] TEST: [2 2 2]
TRAIN: [2 2 2 3 3 3] TEST: [1 1 1]
```

Видим, что наблюдения, принадлежащие одной и той же группе, в каждой итерации попадают либо в обучающую выборку, либо в тестовую выборку. В ходе трех итераций каждая группа побывала в тестовой выборке. Для полной ясности можем непосредственно взглянуть на индексы групп, попавших в обучающую и тестовую выборки на каждой итерации.

## 7.11.2. Перекрестная проверка, учитывающая группы связанных наблюдений с исключением из обучения одной группы, с помощью класса `LeaveOneGroupOut`

Мы помним, что под капотом класса `GroupKFold` на каждой итерации происходит разбиение на обучающую и тестовую выборки так, что наблюдения, при-

надлежащие одной и той же группе, не могут появиться в обучающей и тестовой выборках одновременно. На каждой итерации наблюдения, принадлежащие одной и той же группе, попадают либо в обучающую выборку, либо в тестовую выборку. В тестовую выборку могут попасть наблюдения нескольких групп. Класс `LeaveOneGroupOut` выполняет аналогичную работу с той разницей, что в тестовую выборку попадают наблюдения только одной группы. Отсюда и название класса, так как мы на каждой итерации исключаем из обучения строго одну группу. Поскольку на каждой итерации исключается одна группа, количество итераций будет равно количеству групп. Класс `LeaveOneGroupOut` эквивалентен классу `GroupKFold` с количеством разбиений, равным количеству групп.

Давайте создадим игрушечные массив признаков и массив меток, массив с индексами групп.

```
# создаем игрушечные массив признаков и массив меток
X_toy = np.array(list(range(17)))
y_toy = np.array([0, 0, 1, 1, 1, 1, 1, 1, 0,
                 0, 0, 0, 0, 0, 0, 0])
# создаем идентификатор групп
groups = np.array([1, 1, 2, 2, 3, 3, 3, 4, 5,
                  5, 5, 5, 6, 6, 7, 8, 8])
```

Взглянем на индексы групп в обучающей и тестовой выборках на каждой итерации, которые были созданы классом `GroupKFold`.

```
# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train, test in gkf.split(X_toy, y_toy, groups=groups):
    print("TRAIN:", groups[train], "TEST:", groups[test])

TRAIN: [2 2 3 3 3 4 6 6 7 8 8] TEST: [1 1 5 5 5 5]
TRAIN: [1 1 5 5 5 5 6 6 7 8 8] TEST: [2 2 3 3 3 4]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5] TEST: [6 6 7 8 8]
```

А теперь взглянем на индексы групп в обучающей и тестовой выборках на каждой итерации, которые были созданы классом `LeaveOneGroupOut`.

```
# создаем экземпляр класса LeaveOneGroupOut
logo = LeaveOneGroupOut()

# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки, по каждой из восьми итераций,
# количество итераций = количество групп
for train, test in logo.split(X_toy, y_toy, groups=groups):
    print("TRAIN:", groups[train], "TEST:", groups[test])

TRAIN: [2 2 3 3 3 4 5 5 5 5 6 6 7 8 8] TEST: [1 1]
TRAIN: [1 1 3 3 3 4 5 5 5 5 6 6 7 8 8] TEST: [2 2]
TRAIN: [1 1 2 2 4 5 5 5 5 6 6 7 8 8] TEST: [3 3 3]
TRAIN: [1 1 2 2 3 3 3 5 5 5 5 6 6 7 8 8] TEST: [4]
TRAIN: [1 1 2 2 3 3 3 4 6 6 7 8 8] TEST: [5 5 5 5]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 7 8 8] TEST: [6 6]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 6 6 8 8] TEST: [7]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 6 6 7] TEST: [8 8]
```

Давайте убедимся, что класс `GroupKFold` с количеством итераций, равным количеству групп, работает точно так же, как класс `LeaveOneGroupOut`.

```
# создаем экземпляр класса GroupKFold
new_gkf = GroupKFold(n_splits=8)

# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки, по каждой из восьми итераций,
# количество итераций = количество групп
for train, test in new_gkf.split(X_toy, y_toy, groups=groups):
    print("TRAIN:", groups[train], "TEST:", groups[test])

TRAIN: [1 1 2 2 3 3 3 4 6 6 7 8 8] TEST: [5 5 5 5]
TRAIN: [1 1 2 2 4 5 5 5 5 6 6 7 8 8] TEST: [3 3 3 3]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 6 6 7] TEST: [8 8 8 8]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 7 8 8] TEST: [6 6 6 6]
TRAIN: [1 1 3 3 3 4 5 5 5 5 6 6 7 8 8] TEST: [2 2 2 2]
TRAIN: [2 2 3 3 3 4 5 5 5 5 6 6 7 8 8] TEST: [1 1 1 1]
TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 6 6 8 8] TEST: [7 7 7 7]
TRAIN: [1 1 2 2 3 3 3 5 5 5 5 6 6 7 8 8] TEST: [4 4 4 4]
```

Видим результаты, аналогичные результатам, которые были получены с помощью класса `LeaveOneGroupOut`.

### 7.11.3. Перекрестная проверка, учитывающая группы связанных наблюдений с исключением из обучения $p$ групп, с помощью класса `LeavePGroupsOut`

С помощью класса `LeavePGroupsOut` на каждой итерации можно исключать из обучения определенное количество групп. У него есть единственный параметр  $p$ , с помощью которого задают требуемое количество исключаемых групп. Давайте создадим экземпляр класса `LeavePGroupsOut`, задав  $p$  равным 2, и применим к игрушечному набору. Нас будут интересовать индексы групп, попавших в обучающую и тестовую выборки, на каждой итерации.

```
# создаем экземпляр класса LeavePGroupsOut
lpgo = LeavePGroupsOut(n_groups=2)

# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки, по каждой из 28 итераций
for cnt, (train, test) in enumerate(
    lpgo.split(X_toy, y_toy, groups=groups), 1):
    print(f"\n{cnt}-я итерация")
    print("TRAIN:", groups[train], "TEST:", groups[test])
```

```
1-я итерация
TRAIN: [3 3 3 4 5 5 5 5 6 6 6 7 8 8] TEST: [1 1 2 2]
```

```
2-я итерация
TRAIN: [2 2 4 5 5 5 5 6 6 7 8 8] TEST: [1 1 3 3 3]
```

```
3-я итерация
TRAIN: [2 2 3 3 3 5 5 5 5 6 6 7 8 8] TEST: [1 1 4 4]
```

. . . . .

27-я итерация

TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 7] TEST: [6 6 8 8]

28-я итерация

TRAIN: [1 1 2 2 3 3 3 4 5 5 5 5 6 6] TEST: [7 8 8]

Видим, что действительно каждый раз в тестовую выборку попадают наблюдения двух групп. Общей формулой вычисления количества итераций здесь будет формула числа сочетаний из  $n$  объектов по  $k$   $C_n^k = \frac{n!}{(n-k)!k!}$ . Применим

ее для нашего примера:  $C_n^k = \frac{n!}{(n-k)!k!} = \frac{8!}{6!2!} = \frac{40320}{720 \times 2} = \frac{40320}{1440} = 28$ .

#### 7.11.4. Перекрестная проверка, учитывающая группы связанных наблюдений и распределение классов, с помощью класса StratifiedGroupKFold

Допустим, вам нужно классифицировать клиентов на неоткликнувшихся и откликнувшихся, клиенты сгруппированы по стратам – возрастным группам, уровню лояльности. Вам необходимо создавать блоки так, чтобы все клиенты одной и той же страты находились либо в обучающей выборке, либо в тестовой выборке, и при этом максимально сохранить процент наблюдений каждого класса. В этом случае вам нужен класс StratifiedGroupKFold. Его часто применяют, чтобы оценить дискриминирующую способность модели в разрезе какого-то группирующего фактора.

Разница между классами GroupKFold и StratifiedGroupKFold заключается в том, что GroupKFold пытается создавать блоки так, чтобы количество уникальных групп было одним и тем же в каждом блоке, тогда как StratifiedGroupKFold пытается создавать блоки, которые сохраняют максимально возможный процент каждого класса (с учетом ограничения создавать непересекающиеся группы в ходе разбиения).

Давайте получим разбиения с помощью класса GroupKFold и по каждой итерации посмотрим распределение классов, индексы групп и значения зависимой переменной в обучающей и тестовой выборках.

```
# взглянем на распределение классов, индексы групп
# и значения зависимой переменной в обучающей
# и тестовой выборках по каждой из трех итераций
for cnt, (train, test) in enumerate(
    gkf.split(X_toy, y_toy, groups=groups), 1):
    print(f"\n{cnt}-я итерация")
    uniques, counts = np.unique(y_toy[train], return_counts=True)
    train_pcnt = dict(zip(uniques, counts * 100 / len(y_toy[train])))
    uniques, counts = np.unique(y_toy[test], return_counts=True)
    test_pcnt = dict(zip(uniques, counts * 100 / len(y_toy[test])))
    print("\nTRAIN:\n", train_pcnt)
    print(" groups:", groups[train])
    print("      y:", y_toy[train])
    print("TEST:\n", test_pcnt)
```

```
print(" groups:", groups[test])
print("      y:", y_toy[test])
```

1-я итерация

```
TRAIN:
{0: 45.45454545454545, 1: 54.54545454545455}
groups: [2 2 3 3 3 4 6 6 7 8 8]
      y: [1 1 1 1 1 1 0 0 0 0 0]
TEST:
{0: 100.0}
groups: [1 1 5 5 5 5]
      y: [0 0 0 0 0 0]
```

2-я итерация

```
TRAIN:
{0: 100.0}
groups: [1 1 5 5 5 5 6 6 7 8 8]
      y: [0 0 0 0 0 0 0 0 0 0 0]
TEST:
{1: 100.0}
groups: [2 2 3 3 3 4]
      y: [1 1 1 1 1 1]
```

3-я итерация

```
TRAIN:
{0: 50.0, 1: 50.0}
groups: [1 1 2 2 3 3 3 4 5 5 5]
      y: [0 0 1 1 1 1 1 1 0 0 0]
TEST:
{0: 100.0}
groups: [6 6 7 8 8]
      y: [0 0 0 0 0]
```

Видим, что вроде бы все выглядит нормально: наблюдения, принадлежащие одной группе, попадают либо в обучающую, либо в тестовую выборку. Однако класс `GroupKFold` совершенно не заботится о распределении классов в выборках. Часто наши выборки состоят из наблюдений одного класса. Теперь применим к нашему набору класс `StratifiedGroupKFold`.

```
# создаем экземпляр класса StratifiedGroupKFold
sgkf = StratifiedGroupKFold(n_splits=3)
```

```
# взглянем на распределение классов, индексы групп
# и значения зависимой переменной в обучающей
# и тестовой выборках по каждой из трех итераций
```

```
for cnt, (train, test) in enumerate(
    gkf.split(X_toy, y_toy, groups=groups), 1):
    print(f"\n{cnt}-я итерация")
    uniques, counts = np.unique(y_toy[train], return_counts=True)
    train_pcnt = dict(zip(uniques, counts * 100 / len(y_toy[train])))
    uniques, counts = np.unique(y_toy[test], return_counts=True)
    test_pcnt = dict(zip(uniques, counts * 100 / len(y_toy[test])))
```

```

print("\nTRAIN:\n", train_pcnt)
print(" groups:", groups[train])
print("      y:", y_toy[train])
print("TEST:\n", test_pcnt)
print(" groups:", groups[test])
print("      y:", y_toy[test])

```

1-я итерация

```

TRAIN:
{0: 72.72727272727273, 1: 27.272727272727273}
groups: [1 1 2 2 4 5 5 5 8 8]
      y: [0 0 1 1 1 0 0 0 0 0]
TEST:
{0: 50.0, 1: 50.0}
groups: [3 3 3 6 6 7]
      y: [1 1 1 0 0 0]

```

2-я итерация

```

TRAIN:
{0: 63.63636363636363, 1: 36.36363636363637}
groups: [3 3 3 4 5 5 5 6 6 7]
      y: [1 1 1 1 0 0 0 0 0 0]
TEST:
{0: 66.66666666666667, 1: 33.333333333333336}
groups: [1 1 2 2 8 8]
      y: [0 0 1 1 0 0]

```

3-я итерация

```

TRAIN:
{0: 58.333333333333336, 1: 41.666666666666664}
groups: [1 1 2 2 3 3 3 6 6 7 8 8]
      y: [0 0 1 1 1 1 1 0 0 0 0 0]
TEST:
{0: 80.0, 1: 20.0}
groups: [4 5 5 5 5]
      y: [1 0 0 0 0]

```

Здесь мы видим, что выборки уже не состоят из наблюдений одного класса, действительно, `StratifiedGroupKFold` пытается создавать блоки, которые сохраняют максимально возможный процент каждого класса.

### 7.11.5. Перекрестная проверка со случайными перестановками при разбиении и учитывающая группы связанных наблюдений с помощью класса `GroupShuffleSplit`

В этом виде проверки каждое из `n_splits` разбиений отбирает `train_size` групп для обучающей выборки и `test_size` групп для тестовой выборки. Наблюдения, принадлежащие определенной группе, могут появиться либо в обучающей выборке, либо в тестовой выборке.

Для `train_size` и `test_size` можно задать целочисленные значения для указания абсолютного количества групп в соответствующей выборке или значения с плавающей точкой для указания количества групп как процента от общего количества групп в соответствующей выборке. В нашем игрушечном наборе – 8 групп. Давайте зададим `train_size=6`, `test_size=2` и `n_splits=5`. В каждом из пяти разбиений в обучающую выборку будут попадать наблюдения 6 групп, а в тестовую выборку будут попадать наблюдения 2 групп. Давайте убедимся в этом.

```
# создаем экземпляр класса GroupShuffleSplit
grpshflsplt = GroupShuffleSplit(train_size=6,
                                test_size=2,
                                n_splits=5,
                                random_state=42)

# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки
for cnt, (train, test) in enumerate(
    grpshflsplt.split(X_toy, y_toy, groups=groups), 1):
    print(f"\n{cnt}-я итерация")
    print("TRAIN:", groups[train], "TEST:", groups[test])
```

```
1-я итерация
TRAIN: [1 1 3 3 3 4 5 5 5 5 7 8 8] TEST: [2 2 6 6]
```

```
2-я итерация
TRAIN: [1 1 2 2 3 3 3 5 5 5 5 6 6 7] TEST: [4 8 8]
```

```
3-я итерация
TRAIN: [2 2 3 3 3 4 5 5 5 5 6 6 8 8] TEST: [1 1 7]
```

```
4-я итерация
TRAIN: [1 1 2 2 4 5 5 5 5 7 8 8] TEST: [3 3 3 6 6]
```

```
5-я итерация
TRAIN: [1 1 3 3 3 4 5 5 5 5 6 6 7] TEST: [2 2 8 8]
```

Действительно, каждый раз в обучающую выборку попадают наблюдения 6 групп, а в тестовую выборку попадают наблюдения 2 групп.

Давайте зададим `train_size=0.5`, `test_size=0.5` и `n_splits=5`. В каждом из пяти разбиений в обучающую выборку будут попадать наблюдения групп. Давайте убедимся в этом.

```
# создаем экземпляр класса GroupShuffleSplit
grpshflsplt = GroupShuffleSplit(train_size=0.5,
                                test_size=0.5,
                                n_splits=5,
                                random_state=42)

# взглянем на индексы групп, попавших в обучающую
# и тестовую выборки
for cnt, (train, test) in enumerate(
    grpshflsplt.split(X_toy, y_toy, groups=groups), 1):
    print(f"\n{cnt}-я итерация")
    print("TRAIN:", groups[train], "TEST:", groups[test])
```

1-я итерация

TRAIN: [3 3 3 4 5 5 5 7] TEST: [1 1 2 2 6 6 8 8]

2-я итерация

TRAIN: [2 2 3 3 3 6 6 7] TEST: [1 1 4 5 5 5 8 8]

3-я итерация

TRAIN: [3 3 3 5 5 5 6 6 8 8] TEST: [1 1 2 2 4 7]

4-я итерация

TRAIN: [1 1 2 2 4 5 5 5] TEST: [3 3 3 6 6 7 8 8]

5-я итерация

TRAIN: [1 1 4 5 5 5 6 6] TEST: [2 2 3 3 3 7 8 8]

Каждый раз в обучающую и тестовую выборки попадают наблюдения групп.

## 7.12. ОБЫЧНЫЙ И СЛУЧАЙНЫЙ ПОИСК НАИЛУЧШИХ ГИПЕРПАРАМЕТРОВ ПО СЕТКЕ С ПОМОЩЬЮ КЛАССОВ **GRIDSEARCHCV** И **RANDOMIZEDSEARCHCV**

Мы помним, что перекрестную проверку нельзя использовать для выбора модели, если предполагается настройка гиперпараметров. Получается, что тестовые блоки применяются и для подбора гиперпараметров, и для итоговой оценки качества, которую нужно получать на отдельном тестовом наборе.

Также мы говорили о том, что когда нужно настраивать гиперпараметры, нам требуется валидационная выборка, и в итоге мы из нашего набора выделяем обучающую выборку – для обучения модели, валидационную выборку – для настройки гиперпараметров и тестовую выборку – для итоговой оценки качества выбранной модели. Однако схема наследует обсуждавшиеся недостатки разбиения на обучающую и тестовую выборки – невозможность использовать все имеющиеся обучающие-валидационные данные для обучения и валидации, обусловленное этим пессимистичное смещение, отсутствие вариативности обучающего и валидационного набора.

Для преодоления этих недостатков можно воспользоваться комбинированной проверкой, которая сочетает случайное разбиение на обучающую и тестовую выборки и  $k$ -блочную перекрестную проверку.

Предположим, у нас есть три значения гиперпараметра модели машинного обучения – три значения глубины дерева. Разбиваем набор данных на обучающую и тестовую выборки. Для каждого значения гиперпараметра на обучающей выборке запускается  $k$ -блочная перекрестная проверка, пусть  $k = 5$ . Таким образом, для каждого значения гиперпараметра будет 5 итераций перекрестной проверки. На каждой итерации на 4 обучающих блоках обучаем модель (например, модель дерева решений), на одном проверочном блоке оцениваем качество ее прогнозов. Таким образом, для каждого значения гиперпараметра получаем значение метрики, усредненное по 5 проверочным блокам перекрестной проверки. Находим оптимальное усредненное значение метрики и запоминаем соответствующее значение гиперпараметра, объявляем его

оптимальным. Строим модель с оптимальным значением гиперпараметра на всей обучающей выборке и итоговую оценку качества прогнозов получаем на тестовой выборке, которая не использовалась для обучения и настройки гиперпараметров. Если качество прогнозов модели на тестовой выборке нас устраивает, заново обучаем модель на всей исторической выборке и применяем ее к новым данным.

Аналогично с конвейером. Допустим, у нас есть конвейер из модели импутации и модели дерева решений. У нас три значения гиперпараметра для модели импутации – замена средним, замена медианой и замена константой (например, нулем) и три гиперпараметра для модели дерева решений – все те же три значения глубины. Разбиваем набор данных на обучающую и тестовую выборки. Для каждой комбинации значений гиперпараметров (всего будет  $3 \times 3 = 9$  таких комбинаций) на обучающей выборке запускается  $k$ -блочная перекрестная проверка, пусть  $k = 5$ . Таким образом, для каждой комбинации значений гиперпараметров будет 5 итераций перекрестной проверки. На каждой итерации на 4 обучающих блоках обучаем конвейер, состоящий из модели импутации и модели дерева решений, на одном проверочном блоке оцениваем качество его прогнозов. Таким образом, для каждой комбинации значений гиперпараметров получаем значение метрики, усредненное по 5 проверочным блокам перекрестной проверки. Находим оптимальное усредненное значение метрики и запоминаем соответствующую комбинацию значений гиперпараметров, объявляем ее оптимальной. По сути, это обозначает, что теперь каждую модель в конвейере мы строим с найденным для нее оптимальным значением гиперпараметра на всей обучающей выборке и итоговую оценку качества прогнозов получаем на тестовой выборке, которая не использовалась для обучения и настройки гиперпараметров. Если качество прогнозов конвейера с найденной оптимальной комбинацией значений гиперпараметров на тестовой выборке нас устраивает, заново обучаем этот конвейер на всей исторической выборке и применяем его к новым данным.

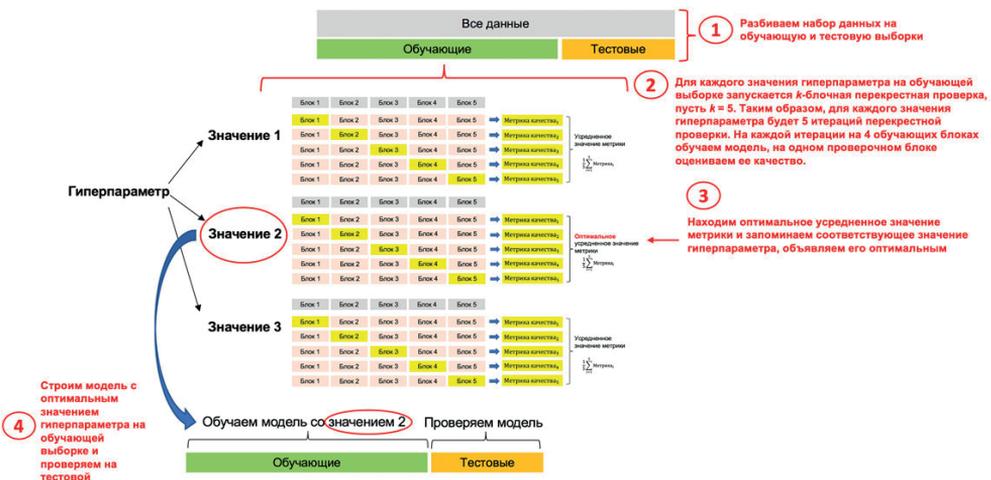


Рис. 45 Схема комбинированной проверки

Обратите внимание, что мы можем проходить все значения гиперпараметров по сетке (обычный поиск, или `grid search`) или выбирать их из сетки случайным образом (случайный поиск, или `random search`).

Комбинируем проверку с обычным поиском значений гиперпараметров по сетке можно реализовать с помощью класса `GridSearchCV`. Ниже подробно разобраны параметры класса `GridSearchCV`.

```

from sklearn.model_selection import GridSearchCV(
    1 estimator,
    2 param_grid,
    3 scoring=None,
    4 n_jobs=None,
    5 refit=True,
    6 cv=None,
    7 return_train_score=False)

```

1 Задаёт объект-модель машинного обучения, т. е. экземпляр класса, в котором реализован соответствующий метод машинного обучения

2 Задаёт словарь или список словарей

3 Словарь, в котором ключами будут названия гиперпараметров (строки) и значениями будут списки значений гиперпараметров

```

param_grid = {'max_depth': [4, 6, 8, 10],
              'max_features': [3, 6, 9, 12]}

```

4 Список словарей, когда каждый словарь в списке выделяется в самостоятельную сетку гиперпараметров

```

param_grid = [{'kernel': ['rbf'],
                'C': [0.001, 0.01, 0.1, 1, 10, 100],
                'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
              {'kernel': ['linear'],
                'C': [0.001, 0.01, 0.1, 1, 10, 100]}]

```

5 Задаёт оптимизируемую метрику

6 Задаёт количество используемых ядер процессора

7 Заново обучает модель с наилучшими значениями гиперпараметров на всем обучающем наборе (по умолчанию задан)

8 Задаёт стратегию перекрестной проверки

```

param_grid = [{'logreg_penalty': ['l1'],
                'logreg_solver': ['saga', 'liblinear'],
                'logreg_C': np.logspace(-2.5, 0.5, 10)},
              {'logreg_penalty': ['elasticnet'],
                'logreg_l1_ratio': [0.3, 0.4, 0.5, 0.6, 0.7],
                'logreg_solver': ['saga'],
                'logreg_C': np.logspace(-2.5, 0.5, 10)}]

```

9 Возвращает значения метрик для обучающей выборки, которые используются для получения информации о том, как различные гиперпараметры влияют на компромисс между переобучением и недообучением. Однако вычисление этих значений может быть дорогостоящим в вычислительном плане и не является строго обязательным для выбора гиперпараметров, обеспечивающих наилучшую обобщающую способность

Также полезно знать атрибуты класса `GridSearchCV`.

Атрибут	Предназначение
<code>cv_results_</code>	Возвращает результаты перекрестной проверки, которые представляют собой словарь массивов NumPy – словарь, в котором ключами являются имена столбцов, а значениями – значения столбцов, его можно перевести в объект <code>DataFrame</code> библиотеки <code>pandas</code>
<code>best_estimator_</code>	Возвращает модель, которая дала наилучшее значение метрики на той части данных, которая не участвовала в обучении
<code>best_score_</code>	Возвращает наилучшее значение метрики по итогам перекрестной проверки, полученное с помощью наилучшей модели
<code>best_params_</code>	Возвращает значения гиперпараметров, давших наилучшие результаты на той части данных, которая не участвовала в обучении

Комбинируем проверку со случайным поиском значений гиперпараметров из сетки можно реализовать с помощью класса `RandomizedSearchCV`. Ниже подробно разобраны параметры класса `RandomizedSearchCV`.

```

from sklearn.model_selection import RandomizedSearchCV(
    1 estimator,
    2 param_distributions,
    3 n_iter=10,
    4 scoring=None,
    5 n_jobs=None,
    6 refit=True,
    7 cv=None,
    8 random_state=None,
    9 return_train_score=False)

```

1 Задаёт объект-модель машинного обучения, т.е. экземпляр класса, в котором реализован соответствующий метод машинного обучения  
 2 Задаёт словарь, в котором ключами будут названия гиперпараметров (строки) и значениями будут списки значений гиперпараметров  
 3 Задаёт количество случайно отбираемых значений гиперпараметров или комбинаций значений гиперпараметров (если перебираем несколько гиперпараметров)  
 4 Задаёт оптимизируемую метрику  
 5 Задаёт количество используемых ядер процессора  
 6 Заново обучает модель с наилучшими значениями гиперпараметров на всем обучающем наборе (по умолчанию задан)  
 7 Задаёт стратегию перекрестной проверки  
 8 Поскольку отбор значений гиперпараметров является случайным, надо позаботиться о воспроизводимости результатов. Для этого с помощью параметра `random_state` задаём стартовое значение генератора псевдослучайных чисел  
 Возвращает значения метрик для обучающей выборки, которые используются для получения информации о том, как различные гиперпараметры влияют на компромисс между переобучением и недообучением. Однако вычисление этих значений может быть дорогостоящим в вычислительном плане и не является строго обязательным для выбора гиперпараметров, обеспечивающих наилучшую обобщающую способность

Как и у класса `GridSearchCV`, у класса `RandomizedSearchCV` есть атрибуты `cv_results_`, `best_estimator_`, `best_score_`, `best_params_`.

### 7.12.1. Обычный поиск оптимальных значений гиперпараметров моделей предварительной подготовки и модели машинного обучения

Итак, давайте начнем с классического перебора значений гиперпараметров моделей предварительной подготовки и моделей машинного обучения.

Давайте импортируем необходимые библиотеки, классы и функции, загрузим данные `StateFarm` с пропусками.

```

# импортируем необходимые библиотеки, функции и классы
import pandas as pd
import numpy as np
from sklearn.model_selection import (train_test_split,
                                     KFold,
                                     ParameterGrid,
                                     cross_val_score,
                                     GridSearchCV,
                                     RandomizedSearchCV)

from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (StandardScaler,
                                   OneHotEncoder)

from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from catboost import CatBoostClassifier
from sklearn.metrics import roc_auc_score
from sklearn.pipeline import Pipeline
from category_encoders import WOEEncoder, SumEncoder
from tqdm import tqdm_notebook

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')

```

Преобразуем строковые значения зависимой переменной в целочисленные, это понадобится для корректной работы классов пакета `category_encoders`.

```
# преобразовываем строковые значения в целочисленные
dct = {'No': 0, 'Yes': 1}
data['Response'] = data['Response'].replace(dct)
```

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)
```

Давайте создадим списки категориальных и количественных переменных.

```
# создаем списки категориальных
# и количественных столбцов
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()
```

Мы вновь зададим списки количественных и категориальных переменных, назначим отдельные конвейеры для переменных разного типа, создадим список трансформеров, передадим этот список в ColumnTransformer и создадим итоговый конвейер. Для количественных переменных мы будем применять импутацию средними (применяется по умолчанию) и стандартизацию, а для категориальных переменных будем применять импутацию константным значением (для категориальных переменных это будет строковое значение 'missing\_value') и дамми-кодирование.

```
# создаем списки количественных и категориальных столбцов
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()

# создаем конвейер для количественных переменных
num_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('scaler', StandardScaler())
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='constant')),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])
```

```
# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
```

```
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
               ('cat', cat_pipe, cat_columns)]

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)

# задаем итоговый конвейер
ml_pipe = Pipeline([
    ('tf', transformer),
    ('logreg', LogisticRegression(solver='lbfgs',
                                 max_iter=200))
])
```

Теперь задаем сетку гиперпараметров. Мы можем перебирать значения гиперпараметров не только для модели машинного обучения, но и для моделей предварительной подготовки. Название гиперпараметра для модели предварительной подготовки будет представлять собой название этапа итогового конвейера\_\_название конвейера с преобразованиями\_\_название этапа конвейера для соответствующего класса, строящего модель предварительной подготовки\_\_название гиперпараметра модели предварительной подготовки. Название гиперпараметра для модели машинного обучения будет представлять собой название этапа итогового конвейера\_\_название гиперпараметра модели машинного обучения.



```
# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imp', SimpleInputer()),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))])

# создаем список преэлементных картелей, в котором
# первый элемент картеля - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
               ('cat', cat_pipe, cat_columns)]

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)

# задаем итоговый конвейер
ml_pipe = Pipeline([('tf', transformer),
                  ('logreg', LogisticRegression(solver='lbfgs', max_iter=200))])
```

Рис. 46 Название гиперпараметра для модели предварительной подготовки

В данном примере мы будем перебирать стратегии импутации пропусков для категориальных и количественных признаков, а также значения силы регуляризации.

```
# задаем сетку гиперпараметров
param_grid = {
    'tf_num_imp_strategy': ['mean', 'median', 'constant'],
    'tf_cat_imp_strategy': ['most_frequent', 'constant'],
    'logreg_C': [.01, .1, .5, 1, 5, 10, 100]
}
```

Теперь создаем экземпляр класса GridSearchCV, передав конвейер, сетку гиперпараметров и указав количество блоков перекрестной проверки.

```
# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество
# блоков перекрестной проверки
```

```
gs = GridSearchCV(ml_pipe,
                  param_grid,
                  cv=5)
```

Запускаем поиск по всем значениям гиперпараметров в сетке точно так же, как и любую другую модель scikit-learn.

```
# выполняем поиск по всем значениям сетки
gs.fit(X_train, y_train)
```

В данном случае `gs.fit()` выполняет поиск по всем значениям гиперпараметров, приведенных в сетке. Для каждой комбинации гиперпараметров (всего будет  $3 \times 2 \times 7 = 42$  комбинации) мы будем 5 раз на 4 обучающих блоках перекрестной проверки обучать последовательность моделей (модель импутации для количественных переменных, модель стандартизации для количественных переменных, модель импутации для категориальных переменных, модель дамми-кодирования для категориальных переменных, модель логистической регрессии) и на одном проверочном блоке вычислять значение правильности. В итоге для каждой последовательности моделей со своей комбинацией значений гиперпараметров мы получим значение правильности, усредненное по 5 проверочным блокам перекрестной проверки. Затем мы выберем последовательность моделей с наилучшей комбинацией значений гиперпараметров, т.е. комбинацией значений гиперпараметров, дающей наибольшее значение правильности, усредненное по 5 проверочным блокам перекрестной проверки. Потом мы обучим эту последовательность моделей с наилучшей комбинацией гиперпараметров на всей обучающей выборке.

Давайте взглянем на наилучшие значения гиперпараметров, наилучшее значение правильности (наибольшее значение правильности, усредненное по 5 проверочным блокам перекрестной проверки), значение правильности на тестовой выборке.

```
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs.best_params_))
# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    gs.best_score_))
# смотрим значение правильности
# на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    gs.score(X_test, y_test)))
```

```
Наилучшие значения гиперпараметров:
{'logreg__C': 0.01,
 'tf_cat_imp_strategy': 'most_frequent',
 'tf_num_imp_strategy': 'mean'}
Наилучшее значение правильности: 0.900
Значение правильности на тестовой выборке: 0.900
```

Извлекаем дамми-переменные, созданные классом `OneHotEncoder`, и добавляем в конец списка количественных переменных. Потом извлекаем константу и коэффициенты наилучшей модели логистической регрессии. Для этого необходимо воспользоваться атрибутами `named_transformers_` и `best_estimator_`.

```

# извлекаем дамми-переменные, созданные классом OneHotEncoder
cat = gs.best_estimator_['tf'].named_transformers_['cat']
onehot_columns = list(cat.named_steps['ohe'].get_feature_names_out(
    input_features=cat_columns))

# добавляем к списку количественных переменных дамми-переменные,
# созданные OneHotEncoder, т.е. сохраняем тот же порядок
# столбцов, что задал ColumnTransformer
all_cols = num_columns + onehot_columns

# извлекаем константу
intercept = np.round(gs.best_estimator_['logreg'].intercept_[0], 3)
intercept

-1.978

# извлекаем коэффициенты
coef = np.round(gs.best_estimator_['logreg'].coef_, 3)
coef

array([[ 0.011,  0.014,  0.094, -0.038, -0.021, -0.026, -0.064, -0.041,
         0.079, -0.038, -0.058,  0.039,  0.083, -0.074,  0.01 ,  0.053,
        -0.215,  0.027,  0.53 , -0.395, -0.007,  0.007]])

```

С помощью функции `zip()` «сшиваем» константу и коэффициенты с названиями признаков.

```

# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], all_cols):
    print(feature, c)

```

```

Константа: -1.978
Регрессионные коэффициенты:
Customer Lifetime Value 0.011
Income 0.014
Monthly Premium Auto 0.094
Months Since Last Claim -0.038
Months Since Policy Inception -0.021
Number of Open Complaints -0.026
Number of Policies -0.064
Coverage_Basic -0.041
Coverage_Extended 0.079
Coverage_Premium -0.038
Education_Bachelor -0.058
Education_College 0.039
Education_Doctor 0.083
Education_High School or Below -0.074
Education_Master 0.01
EmploymentStatus_Disabled 0.053
EmploymentStatus_Employed -0.215
EmploymentStatus_Medical Leave 0.027

```

```

EmploymentStatus_Retired 0.53
EmploymentStatus_Unemployed -0.395
Gender_F -0.007
Gender_M 0.007

```

Теперь результаты перекрестной проверки, сохраненные в атрибуте `cv_results_`, записываем в объект `DataFrame` и с помощью метода `.pivot_table()` превращаем в сводную таблицу.

```

# запишем результаты поиска в DataFrame
results = pd.DataFrame(gs.cv_results_)
# превращаем в сводную таблицу
table = results.pivot_table(
    values=['mean_test_score'],
    index=['param_logreg_C',
           'param_tf_cat_imp_strategy',
           'param_tf_num_imp_strategy'])
print(table)

```

param_logreg_C	param_tf_cat_imp_strategy	param_tf_num_imp_strategy	mean_test_score
0.01	constant	constant	0.899742
		mean	0.899742
		median	0.899742
	most_frequent	constant	0.899742
		mean	0.899742
		median	0.899742
0.10	constant	constant	0.899569
		mean	0.899569
		median	0.899569
	most_frequent	constant	0.899569
		mean	0.899569
		median	0.899569
0.50	constant	constant	0.898363
		mean	0.898363
		median	0.898363
	most_frequent	constant	0.898363
		mean	0.898363
		median	0.898363
1.00	constant	constant	0.898363
		mean	0.898363
		median	0.898363
	most_frequent	constant	0.898363
		mean	0.898363
		median	0.898363
5.00	constant	constant	0.897674
		mean	0.897502
		median	0.897502
	most_frequent	constant	0.897674
		mean	0.897502
		median	0.897502
10.00	constant	constant	0.897847
		mean	0.897502
		median	0.897502
	most_frequent	constant	0.898019
		mean	0.897502
		median	0.897674
100.00	constant	constant	0.897847
		mean	0.897847
		median	0.897847
	most_frequent	constant	0.897847
		mean	0.897847
		median	0.897847

Здесь мы видим 42 последовательности моделей, соответствующие 42 комбинациям значений гиперпараметров (3 стратегии импутации для количественных признаков \* 2 стратегии импутации для категориальных признаков \* 7 значений силы регуляризации).

### 7.12.2. Обычный поиск оптимальных значений гиперпараметров моделей предварительной подготовки и модели машинного обучения с добавлением строки прогресса

Перебор большого количества значений гиперпараметров требует большого времени, и часто необходимо понять, сколько времени осталось до завершения поиска по сетке. Мы можем реализовать поиск по сетке со строкой прогресса (progress bar).

Создаем пустой список `scores`, в который будем записывать гиперпараметры и полученные метрики качества, задаем метрику качества и печатаем строку статуса.

```
# создаем пустой список scores, в который будем записывать
# гиперпараметры и полученные метрики качества
scores = []
# задаем метрику качества
scoring = 'accuarcy'
# печатаем строку статуса
print("Выполняем поиск оптимальных значений гиперпараметров...")
```

Внутри цикла `for` с помощью функции `tqdm()` одноименной библиотеки обучаем конвейер с каждой комбинацией гиперпараметров. Обратите внимание: здесь мы используем класс `ParameterGrid`, для того чтобы в явном виде перебирать гиперпараметры внутри цикла `for`. Класс `ParameterGrid` преобразовывает сетку гиперпараметров, обычно записанную в виде словаря с парами ключ-значение (где ключ – имя гиперпараметра, а значение – значение гиперпараметра), в последовательность. Приведем пример:

```
param_grid = {'max_features': [1, 2, 3], 'min_samples': [1, 3]}
```

```
list(ParameterGrid(param_grid))

[{'max_features': 1, 'min_samples': 1},
 {'max_features': 1, 'min_samples': 3},
 {'max_features': 2, 'min_samples': 1},
 {'max_features': 2, 'min_samples': 3},
 {'max_features': 3, 'min_samples': 1},
 {'max_features': 3, 'min_samples': 3}]
```

В итоге получаем список с результатами, сортируем его, печатаем наилучшие гиперпараметры и наилучшее значение метрики. Затем передаем наилучшие значения гиперпараметров в конвейер и обучаем его с этими значениями гиперпараметров на всей обучающей выборке, печатаем значение правильности на тестовой выборке.

```

# поскольку tqdm работает с итерируемыми объектами,
# то используем класс ParameterGrid, для того чтобы
# в явном виде перебирать гиперпараметры внутри цикла for
for param in tqdm(list(ParameterGrid(param_grid)),
                  desc='Выполнено'):
    # задаем гиперпараметры конвейера
    ml_pipe.set_params(**param)

    # обучаем конвейер с этими гиперпараметрами
    # и сохраняем гиперпараметры и полученные
    # метрики качества в список scores
    scores.append([param, cross_val_score(ml_pipe,
                                          X_train,
                                          y_train,
                                          scoring=scoring,
                                          cv=5)])

    # рассчитываем и добавляем в список scores
    # усредненную метрику качества
    scores[-1].append(sum(scores[-1][1]) / len(scores[-1][1]))

# сортируем список с полученными результатами
scores.sort(reverse=True, key=lambda x: x[2])

# сохраняем и печатаем наилучшие значения гиперпараметров
best_params = scores[0][0]
print("Наилучшие значения гиперпараметров:",
      best_params, sep='\n', end='\n')

# сохраняем и печатаем наилучшее значение метрики качества
best_score = scores[0][2]
print("Наилучшее значение %s: %.3f" % (scoring, best_score))

# передаем наилучшие значения гиперпараметров
# в конвейер и обучаем его с этими значениями
# гиперпараметров на всей обучающей выборке
ml_pipe.set_params(**best_params).fit(X_train, y_train)
# вычисляем правильность для тестовой выборки
test_score = ml_pipe.score(X_test, y_test)
# печатаем значение метрики качества на тестовой выборке
print("Значение %s на тестовой выборке: %.3f" % (scoring, test_score))

```

Выполняем поиск оптимальных значений гиперпараметров...

Выполнено: 100%  42/42 [00:07<00:00, 5.30it/s]

Наилучшие значения гиперпараметров:

```
{'logreg_C': 0.01, 'tf_cat_imp_strategy': 'most_frequent',
  'tf_num_imp_strategy': 'mean'}
```

Наилучшее значение accuracy: 0.900

Значение accuracy на тестовой выборке: 0.900

### 7.12.3. Случайный поиск оптимальных значений гиперпараметров моделей предварительной подготовки и модели машинного обучения

Теперь создаем экземпляр класса `RandomizedSearchCV`, передав конвейер, сетку гиперпараметров и указав количество блоков перекрестной проверки.

```
# создаем экземпляр класса RandomizedSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество блоков перекрестной
# проверки, количество отбираемых значений гиперпараметров,
# задав стартовое значение генератора псевдослучайных чисел для
# воспроизводимости результатов
rs = RandomizedSearchCV(ml_pipe,
                        param_grid,
                        n_iter=10,
                        cv=5,
                        random_state=42)
```

Запускаем поиск по значениям гиперпараметров, случайно отобранным из сетки.

```
# выполняем поиск по случайно
# отобранным значениям из сетки
rs.fit(X_train, y_train)
```

Вновь взглянем на наилучшие значения гиперпараметров, наилучшее значение правильности (наибольшее значение правильности, усредненное по пяти проверочным блокам перекрестной проверки), значение правильности на тестовой выборке.

```
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров: \n{}".format(
    rs.best_params_))
# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    rs.best_score_))
# смотрим значение правильности
# на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    rs.score(X_test, y_test)))
```

```
Наилучшие значения гиперпараметров:
{'tf_num_imp_strategy': 'median',
 'tf_cat_imp_strategy': 'constant',
 'logreg_C': 0.01}
Наилучшее значение правильности: 0.900
Значение правильности на тестовой выборке: 0.900
```

Снова результаты перекрестной проверки, записанные в атрибуте `cv_results_`, записываем в объект `DataFrame` и с помощью метода `.pivot_table()` превращаем в сводную таблицу.

```
# запишем результаты поиска в DataFrame
res = pd.DataFrame(rs.cv_results_)
# превращаем в сводную таблицу
tbl = res.pivot_table(
    values=['mean_test_score'],
    index=['param_logreg__C',
           'param_tf_cat_imp_strategy',
           'param_tf_num_imp_strategy'])
print(tbl)
```

			mean_test_score
param_logreg__C	param_tf_cat_imp_strategy	param_tf_num_imp_strategy	
0.01	constant	median	0.899742
0.10	most_frequent	constant	0.899569
		mean	0.899569
0.50	most_frequent	median	0.898363
1.00	most_frequent	median	0.898363
5.00	constant	constant	0.897674
	most_frequent	constant	0.897674
		median	0.897502
10.00	most_frequent	mean	0.897502
100.00	constant	mean	0.897847

Здесь мы видим 10 последовательностей моделей, соответствующих 10 случайно выбранным комбинациям значений гиперпараметров ( $n\_iter=10$ ).

#### 7.12.4. Обычный поиск оптимальных значений гиперпараметров для CatBoost при обработке категориальных признаков «как есть» (заданы индексы категориальных признаков)

Теперь разберем особенности оптимизации значений гиперпараметров для CatBoost. CatBoost – это библиотека градиентного бустинга, которую в июле 2017 года представила компания Yandex. Ее особенностью является возможность работать с категориальными признаками (под капотом эти переменные будут преобразованы в количественные), и поэтому нам нужно указать индексы категориальных признаков.

Давайте модифицируем наши конвейеры, создадим список трансформеров и передадим этот список в ColumnTransformer.

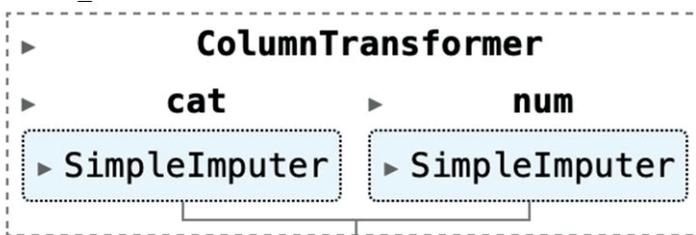
```
# создаем конвейер для категориальных переменных,
# который будем использовать с catboost
catbst_cat_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent'))
])
# создаем конвейер для количественных переменных,
# который будем использовать с catboost
catbst_num_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='median'))
])
```

```
# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
catbst_transformers = [('cat', catbst_cat_pipe, cat_columns),
                      ('num', catbst_num_pipe, num_columns)]

# передаем список трансформеров в ColumnTransformer
catbst_transformer = ColumnTransformer(
    transformers=catbst_transformers)
```

Взглянем на сам объект – экземпляр класса `ColumnTransformer`.

```
# взглянем на объект
catbst_transformer
```



Щелкнем по стрелке слева от `ColumnTransformer` и получим следующий вывод.

```
ColumnTransformer(transformers=[('cat',
                                Pipeline(steps=[('imp',
                                                  SimpleImputer(strategy='most_frequent'))]),
                                ['Coverage', 'Education', 'EmploymentStatus',
                                 'Gender']),
                                ('num',
                                Pipeline(steps=[('imp',
                                                  SimpleImputer(strategy='median'))]),
                                ['Customer Lifetime Value', 'Income',
                                 'Monthly Premium Auto',
                                 'Months Since Last Claim',
                                 'Months Since Policy Inception',
                                 'Number of Open Complaints',
                                 'Number of Policies'])])
```

Обращаем внимание на порядок столбцов. Сначала у нас перечисляются имена категориальных переменных, а затем имена количественных переменных, потому что в нашем списке трехэлементных кортежей, который мы передали в `ColumnTransformer`, сначала указан конвейер для категориальных переменных, а затем конвейер для количественных переменных. Этот порядок столбцов отличается от порядка столбцов в исходном датафрейме. В нашем случае категориальные признаки будут иметь индексы 0, 1, 2, 3, хотя в нашем исходном наборе индекс 0 будет соответствовать количественной переменной *Customer Lifetime Value*.

```
# взглянем на исходный список столбцов
print(data.columns.tolist())
```

```
['Customer Lifetime Value', 'Coverage', 'Education', 'EmploymentStatus', 'Gender', 'Income',
'Monthly Premium Auto', 'Months Since Last Claim', 'Months Since Policy Inception', 'Number
of Open Complaints', 'Number of Policies', 'Response']
```

Теперь зададим правильный список индексов категориальных переменных в пространстве трансформированных признаков, воспользовавшись генератором списка.

```
# записываем индексы категориальных признаков
# в пространстве трансформированных признаков
cat_feat_ind = [col for col in range(len(cat_columns))]
cat_feat_ind

[0, 1, 2, 3]
```

Можно всегда придерживаться данной схемы: в списке трехэлементных кортежей, передаваемом в ColumnTransformer, первым задаем конвейер для категориальных признаков, а затем через генератор списка получаем индексы категориальных признаков, которые передаем в модель CatBoost.

Итак, создаем экземпляр класса CatBoostClassifier, передав ему индексы категориальных признаков, формируем итоговый конвейер, задаем сетку гиперпараметров, создаем экземпляр класса GridSearchCV и выполняем поиск.

```
# создаем экземпляр класса CatBoostClassifier
catbst = CatBoostClassifier(n_estimators=200,
                           logging_level='Silent',
                           random_state=42,
                           cat_features=cat_feat_ind)

# задаем итоговый конвейер
catbst_pipe = Pipeline([('tf', catbst_transformer),
                        ('catbst', catbst)])

# задаем сетку гиперпараметров
catbst_param_grid = {
    'tf_cat_imp_strategy': ['most_frequent', 'constant'],
    'catbst_max_depth': [4, 6, 8]
}

# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество
# блоков перекрестной проверки
catbst_gs = GridSearchCV(catbst_pipe,
                        catbst_param_grid,
                        cv=5)

# выполняем поиск по всем значениям сетки
catbst_gs.fit(X_train, y_train)

# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    catbst_gs.best_params_))

# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    catbst_gs.best_score_))

# смотрим значение правильности на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    catbst_gs.score(X_test, y_test)))
```

Наилучшие значения гиперпараметров:

```
{'catbst__max_depth': 8,
 'tf__cat__imp__strategy': 'constant'}
```

Наилучшее значение правильности: 0.918

Значение правильности на тестовой выборке: 0.924

Вновь результаты перекрестной проверки, сохраненные в атрибуте `cv_results_`, записываем в объект `DataFrame` и с помощью метода `.pivot_table()` превращаем в сводную таблицу.

```
# запишем результаты поиска в DataFrame
results = pd.DataFrame(catbst_gs.cv_results_)
# превращаем в сводную таблицу
table = results.pivot_table(
    values=['mean_test_score'],
    index=['param_catbst__max_depth',
          'param_tf__cat__imp__strategy'])
# печатаем таблицу
print(table)
```

		mean_test_score
param_catbst__max_depth	param_tf__cat__imp__strategy	
4	constant	0.903015
	most_frequent	0.903359
6	constant	0.908183
	most_frequent	0.907666
8	constant	0.917657
	most_frequent	0.916968

### 7.12.5. Отбор оптимальной модели предварительной подготовки данных в рамках отдельного трансформера

С помощью класса `GridSearchCV` вы также можете выбрать оптимальную модель предварительной подготовки данных в рамках одного трансформера, например выбрать наиболее эффективный способ превращения категориальных признаков в количественные.

Давайте заново создадим трансформеры, список трансформеров и итоговый конвейер. Обратите внимание на трансформер для категориальных признаков. Помимо модели импутации, мы помещаем в него несколько классов – моделей, превращающих категориальные признаки в количественные, из пакета `category_encoders`.

```
# создаем конвейер для количественных переменных
num_pipe2 = Pipeline([
    ('imp', SimpleImputer(strategy='mean'))
])

# создаем конвейер для категориальных переменных
cat_pipe2 = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('woe', WOEEncoder(return_df=False)),
    ('sum', SumEncoder(return_df=False)),
    ('ohc', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])
```

```

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers2 = [('num2', num_pipe2, num_columns),
                ('cat2', cat_pipe2, cat_columns)]

# передаем список трансформеров в ColumnTransformer
transformer2 = ColumnTransformer(transformers=transformers2)

# задаем итоговый конвейер
ml_pipe2 = Pipeline([
    ('tf2', transformer2),
    ('boost', GradientBoostingClassifier(random_state=42))
])

```

Теперь задаем сетку гиперпараметров. Мы хотим сделать выбор между различными способами превращения категориальных признаков в количественные для градиентного бустинга, попробуем дамми-кодирование (OneHotEncoder), кодирование контрастами сумм (SumEncoder), кодирование WoE-значениями (WOEncoder). Поскольку эти методы предварительной подготовки могут использовать разные гиперпараметры для настройки, мы можем воспользоваться списком словарей, в котором каждый словарь представляет отдельную сетку гиперпараметров. Чтобы задать модель для этапа, мы должны указать имя этапа в качестве названия гиперпараметра. Если нам нужно пропустить какой-то этап в конвейере, мы можем задать для него значение None. Например, если мы хотим выполнить дамми-кодирование, мы можем задать для этапов, соответствующих кодированию контрастами сумм и кодированию WoE-значениями, значение None. Или мы хотим выполнить кодирование контрастами сумм, тогда для этапов, соответствующих дамми-кодированию и кодированию WoE-значениями, задаем значение None. Таким образом, мы построим модели градиентного бустинга с разной обработкой категориальных признаков.

```

# задаем сетку гиперпараметров
param_grid2 = [
    {'boost__max_depth': [4, 6, 8],
     'tf2__cat2__ohe': [None],
     'tf2__cat2__sum': [None]},
    {'boost__max_depth': [4, 6, 8],
     'tf2__cat2__woe': [None],
     'tf2__cat2__sum': [None]},
    {'boost__max_depth': [4, 6, 8],
     'tf2__cat2__woe': [None],
     'tf2__cat2__ohe': [None]}
]

```

Модели градиентного бустинга с разными значениями глубины, используется WoE-кодирование (применяем экземпляр класса WOEncoder)

Модели градиентного бустинга с разными значениями глубины, используется дамми-кодирование (применяем экземпляр класса OneHotEncoder)

Модели градиентного бустинга с разными значениями глубины, используется кодирование контрастами сумм (применяем экземпляр класса SumEncoder)

В данном случае у наших моделей предварительной подготовки нет гиперпараметров, а если бы мы задали вместо экземпляра класса SumEncoder (этап sum) экземпляр класса TargetEncoder (этап te), у которого есть гиперпараметры min\_samples\_leaf и smoothing, сетка могла бы выглядеть так:

```

# импортируем класс TargetEncoder
from category_encoders import TargetEncoder
# создаем конвейер для категориальных переменных
cat_pipe2 = Pipeline([
    ('imp', SimpleImputer(missing_values=np.nan, strategy='most_frequent')),
    ('woe', WOEEncoder(return_df=False)),
    ('te', TargetEncoder(return_df=False)),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])
# задаем сетку гиперпараметров
param_grid2 = [
    {'boost_max_depth': [4, 6, 8],
     'tf2_cat2_ohe': [None],
     'tf2_cat2_te': [None]},
    {'boost_max_depth': [4, 6, 8],
     'tf2_cat2_woe': [None],
     'tf2_cat2_te': [None]},
    {'boost_max_depth': [4, 6, 8],
     'tf2_cat2_te_min_samples_leaf': [2, 4, 6],
     'tf2_cat2_woe': [None],
     'tf2_cat2_ohe': [None]}
]

```

Вновь создаем экземпляр класса GridSearchCV, передав конвейер, сетку гиперпараметров и указав количество блоков перекрестной проверки.

```

# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество
# блоков перекрестной проверки
gs2 = GridSearchCV(ml_pipe2,
                  param_grid2,
                  cv=5)

```

Запускаем поиск по всем значениям сетки.

```

# выполняем поиск по всем значениям сетки
gs2.fit(X_train, y_train)

```

Взглянем на наилучшие значения гиперпараметров, наилучшее значение правильности (значение правильности, усредненное по пяти проверочным блокам перекрестной проверки), значение правильности на тестовой выборке.

```

# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs2.best_params_))
# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    gs2.best_score_))
# смотрим значение правильности
# на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    gs2.score(X_test, y_test)))

```

Наилучшие значения гиперпараметров:

```
{'boost__max_depth': 8, 'tf2__cat2__ohe': None, 'tf2__cat2__sum': None}
```

Наилучшее значение правильности: 0.932

Значение правильности на тестовой выборке: 0.938

Видим, что оптимальной стала комбинация глубины 8 и WoE-кодирование (отключено дамми-кодирование и кодирование контрастами сумм).

Теперь взглянем на результаты перекрестной проверки, записав их в DataFrame.

```
# увеличиваем ширину столбцов
pd.set_option('max_colwidth', 100)
# записываем результаты перекрестной
# проверки в DataFrame
results2 = pd.DataFrame(gs2.cv_results_)
# отбираем нужные столбцы
table2 = results2.loc[:, ['param_boost__max_depth',
                        'params',
                        'mean_test_score']]
# переименуем столбец для удобства интерпретации
table2.rename(columns={'params': 'param_boost__encoding'},
              inplace=True)
# сортируем по убыванию
table2.sort_values('mean_test_score',
                  ascending=False,
                  inplace=True)

table2
```

	param_boost__max_depth	param_boost__encoding	mean_test_score
2	8	{'boost__max_depth': 8, 'tf2__cat2__ohe': None, 'tf2__cat2__sum': None}	0.932300
8	8	{'boost__max_depth': 8, 'tf2__cat2__ohe': None, 'tf2__cat2__woe': None}	0.929371
5	8	{'boost__max_depth': 8, 'tf2__cat2__sum': None, 'tf2__cat2__woe': None}	0.929199
1	6	{'boost__max_depth': 6, 'tf2__cat2__ohe': None, 'tf2__cat2__sum': None}	0.921792
7	6	{'boost__max_depth': 6, 'tf2__cat2__ohe': None, 'tf2__cat2__woe': None}	0.919552
4	6	{'boost__max_depth': 6, 'tf2__cat2__sum': None, 'tf2__cat2__woe': None}	0.918691
3	4	{'boost__max_depth': 4, 'tf2__cat2__sum': None, 'tf2__cat2__woe': None}	0.906115
6	4	{'boost__max_depth': 4, 'tf2__cat2__ohe': None, 'tf2__cat2__woe': None}	0.905943
0	4	{'boost__max_depth': 4, 'tf2__cat2__ohe': None, 'tf2__cat2__sum': None}	0.905599

Постараемся сделать результаты более понятными.

```
# создаем серию со значением кодировок, не забывая передать
# индекс датафрейма для верного сопоставления
enc = pd.Series(['woe', 'ohe', 'sum',
                'woe', 'sum', 'ohe',
                'ohe', 'sum', 'woe'],
               index = table2.index)
# создаем копию таблицы
table2_copy = table2.copy()
# значения серии становятся значениями переименованного столбца
table2_copy['param_boost__encoding'] = enc
table2_copy
```

	param_boost_max_depth	param_boost_encoding	mean_test_score
2	8	woe	0.932300
8	8	ohe	0.929371
5	8	sum	0.929199
1	6	woe	0.921792
7	6	sum	0.919552
4	6	ohe	0.918691
3	4	ohe	0.906115
6	4	sum	0.905943
0	4	woe	0.905599

Недостаток такого подхода заключается в том, что мы вручную задаем нужные названия кодировок. Давайте сделаем присвоение кодировок автоматическим в зависимости от того, какие логические условия у нас выполняются.

```
# присваиваем кодировки автоматически в зависимости от того,
# какие логические условия у нас выполняются
cond = (table2['param_boost_encoding'].str.contains(
    'tf2_cat2_ohe', regex=False)) & (
    table2['param_boost_encoding'].str.contains(
        'tf2_cat2_sum', regex=False))
cond2 = (table2['param_boost_encoding'].str.contains(
    'tf2_cat2_ohe', regex=False)) & (
    table2['param_boost_encoding'].str.contains(
        'tf2_cat2_woe', regex=False))
cond3 = (table2['param_boost_encoding'].str.contains(
    'tf2_cat2_woe', regex=False)) & (
    table2['param_boost_encoding'].str.contains(
        'tf2_cat2_sum', regex=False))

table2['param_boost_encoding'] = np.where(
    cond, 'woe', (np.where(cond2, 'sum', 'ohe')))
table2
```

	param_boost_max_depth	param_boost_encoding	mean_test_score
2	8	woe	0.932300
8	8	sum	0.929371
5	8	ohe	0.929199
1	6	woe	0.921792
7	6	sum	0.919552
4	6	ohe	0.918691
3	4	ohe	0.906115
6	4	sum	0.905943
0	4	woe	0.905599

### 7.12.6. Отбор оптимального метода машинного обучения среди разных методов машинного обучения (перебор значений гиперпараметров с отдельной предобработкой данных под каждый метод машинного обучения)

С помощью класса `GridSearchCV` вы также можете выбрать оптимальную модель машинного обучения среди моделей, относящихся к разным методам машинного обучения, а также настраивать предварительную подготовку, учитывая последующую модель машинного обучения. Например, логистической регрессии нужна стандартизация, а градиентному бустингу и случайному лесу – нет, импутация пропусков медианами эффективна для линейной регрессии и логистической регрессии, а для градиентного бустинга и случайного леса более эффективной стратегией может быть импутация каким-то большим положительным или отрицательным значением, лежащим вне диапазона значений признака.

Давайте загрузим данные.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Response.csv', sep=';')
data.head(5)
```

	mortgage	life_ins	cre_card	deb_card	mob_bank	curr_acc	internet	perloan	savings	atm_user	markpl	age	cus_leng	response
0	No	No	No	No	No	18.0	less than 3 years	No						
1	Yes	Yes	NaN	NaN	Yes	No	NaN	NaN	NaN	Yes	No	18.0	NaN	Yes
2	Yes	Yes	NaN	Yes	No	No	No	No	No	No	Yes	NaN	from 3 to 7 years	Yes
3	Yes	Yes	Yes	Yes	NaN	Yes	No	No	No	NaN	Yes	18.0	from 3 to 7 years	Yes
4	Yes	Yes	No	Yes	No	No	No	Yes	No	Yes	No	NaN	NaN	No

Данные записаны в файле `Response.csv`. Исходная выборка содержит записи о 30 259 клиентах, классифицированных на два класса: 0 – отклика нет (17 170 клиентов) и 1 – отклик есть (13 089 клиентов). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- категориальный признак *Ипотечный кредит* [`mortgage`];
- категориальный признак *Страхование жизни* [`life_ins`];
- категориальный признак *Кредитная карта* [`cre_card`];
- категориальный признак *Дебетовая карта* [`deb_card`];
- категориальный признак *Мобильный банк* [`mob_bank`];
- категориальный признак *Текущий счет* [`curr_acc`];
- категориальный признак *Интернет-доступ к счету* [`internet`];
- категориальный признак *Индивидуальный займ* [`perloan`];
- категориальный признак *Наличие сбережений* [`savings`];
- категориальный признак *Пользование банкоматом за последнюю неделю* [`atm_user`];
- категориальный признак *Пользование услугами онлайн-маркетплейса за последний месяц* [`markpl`];
- количественный признак *Возраст* [`age`];
- количественный признак *Давность клиентской истории* [`cus_leng`];
- бинарная зависимая переменная *Отклик на предложение новой карты* [`response`].

Задача заключается в том, чтобы классифицировать клиентов на неоткликнувшихся и откликнувшихся. Метрика качества – AUC-ROC.

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# создаем обучающий массив признаков, обучающий массив меток,
# тестовый массив признаков, тестовый массив меток
tr, tst, y_tr, y_tst = train_test_split(
    data.drop('response', axis=1),
    data['response'],
    test_size=.3,
    stratify=data['response'],
    random_state=100)
```

Создаем списки количественных и категориальных переменных, создаем трансформеры для каждого списка переменных, создаем список трансформеров, передаем этот список в ColumnTransformer и формируем итоговый конвейер.

```
# создаем списки категориальных
# и количественных столбцов
categorical_features = tr.select_dtypes(
    include='object').columns.tolist()
numeric_features = tr.select_dtypes(
    exclude='object').columns.tolist()

# создаем трансформеры
numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='constant')),
    ('onehot', OneHotEncoder(sparse=False,
                             handle_unknown='ignore'))
])

# передаем список трансформеров в ColumnTransformer
preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])

# формируем итоговый конвейер
pipe = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(solver='lbfgs',
                                     max_iter=400))
])
```

Теперь задаем сетку гиперпараметров. Мы хотим сделать выбор между логистической регрессией (LogisticRegression) и градиентным бустингом (GradientBoostingClassifier). Поскольку эти методы машинного обучения

используют разные гиперпараметры для настройки, один метод нуждается в предварительной обработке, а другой – нет, мы снова можем воспользоваться списком словарей, в котором каждый словарь представляет отдельную сетку гиперпараметров. Чтобы задать модель для этапа, мы должны указать имя этапа в качестве названия гиперпараметра. Помним, что если нам нужно пропустить какой-то этап в конвейере (например, потому что нам не нужна предварительная обработка для GradientBoostingClassifier), мы можем задать для этапа значение None.

Мы построим модели логистической регрессии с разными значениями силы регуляризации со стандартизацией и без нее, а также модели градиентного бустинга с разными значениями глубины, отключив стандартизацию, поскольку бустингу не требуется стандартизация.

*# задаем сетку гиперпараметров*

```
param_grid = [
    {'classifier': [GradientBoostingClassifier(
        n_estimators=50,
        random_state=42,
        subsample=0.8)],
     'classifier__max_depth': [4, 6, 8],
     'preprocessor__num_scaler': [None]},
    {'classifier': [LogisticRegression(solver='lbfgs',
                                       max_iter=400)],
     'classifier__C': [.05, .01],
     'preprocessor__num_scaler': [None]},
    {'classifier': [LogisticRegression(solver='lbfgs',
                                       max_iter=400)],
     'classifier__C': [.05, .01]}
```

Модели градиентного бустинга с разными значениями глубины, стандартизация отключена (используется ансамбль из 50 деревьев и случайная подвыборка наблюдений для каждого дерева)

Модели логистической регрессии с разными значениями силы регуляризации, стандартизация отключена

Модели логистической регрессии с разными значениями силы регуляризации, используются все операции предварительной подготовки (импутация и стандартизация), эквивалентный программный код:

```
{'classifier': [LogisticRegression(solver='lbfgs',
                                   max_iter=400)],
 'classifier__C': [.05, .01],
 'preprocessor': [preprocessor]}
```

С помощью класса KFold задаем 5-блочную перекрестную проверку со случайным перемешиванием данных перед разбиением на блоки (это может быть полезно, если данные отсортированы по значению какого-то важного признака или по меткам классов зависимой переменной). Поскольку используется случайность, не забываем задать стартовое значение генератора псевдослучайных чисел с помощью параметра random\_state.

*# создаем экземпляр класса KFold*

```
kf = KFold(n_splits=5, shuffle=True, random_state=123)
```

Теперь приступаем к поиску. Еще раз обратите внимание, что оптимизируемой метрикой будет не правильность, а AUC-ROC.

*# создаем экземпляр класса GridSearchCV, передав конвейер,*

*# сетку гиперпараметров, оптимизируемую метрику,*

*# стратегию перекрестной проверки*

```
gs = GridSearchCV(pipe,
                  param_grid,
                  scoring='roc_auc',
                  cv=kf)
```

```

# выполняем поиск по всем значениям сетки
gs.fit(tr, y_tr)
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs.best_params_))
# смотрим наилучшее значение AUC
print("Наилучшее значение AUC-ROC: {:.3f}".format(
    gs.best_score_))
# смотрим значение AUC на тестовой выборке
print("AUC-ROC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_tst, gs.predict_proba(tst)[: , 1])))

```

Наилучшие значения гиперпараметров:

```

{'classifier': GradientBoostingClassifier(max_depth=4,
                                         n_estimators=50,
                                         random_state=42,
                                         subsample=0.8),
 'classifier__max_depth': 4, 'preprocessor__num_scaler': None}
Наилучшее значение AUC-ROC: 0.910
AUC-ROC на тестовой выборке: 0.907

```

Видим, что наилучшей моделью стала модель градиентного бустинга с глубиной деревьев 4.

Разберем еще один пример, когда мы выполняем поиск оптимального метода машинного обучения из нескольких методов и для каждого метода у нас выполняется своя предварительная подготовка данных.

Давайте сначала создадим конвейер для логистической регрессии.

```

# создаем трансформеры для логистической регрессии
numeric_transformer_logreg = Pipeline(
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
)

categorical_transformer_logreg = Pipeline([
    ('imputer', SimpleImputer(strategy='constant')),
    ('onehot', OneHotEncoder(sparse=False,
                             handle_unknown='ignore'))
])

# передаем список трансформеров для логистической
# регрессии в ColumnTransformer
preprocessor_logreg = ColumnTransformer([
    ('num', numeric_transformer_logreg, numeric_features),
    ('cat', categorical_transformer_logreg, categorical_features)
])

# формируем итоговый конвейер для логистической регрессии
pipe_logreg = Pipeline([
    ('preprocessor', preprocessor_logreg),
    ('classifier', LogisticRegression(solver='lbfgs',
                                     max_iter=400))
])

```

Теперь создадим конвейер для градиентного бустинга.

```

# создаем трансформеры для градиентного бустинга
numeric_transformer_boost = Pipeline([
    ('imputer', SimpleImputer(strategy='constant',
                              fill_value=-9999))
])

categorical_transformer_boost = Pipeline([
    ('imputer', SimpleImputer(strategy='constant')),
    ('sum', SumEncoder(return_df=False))
])

# передаем список трансформеров для градиентного
# бустинга в ColumnTransformer
preprocessor_boost = ColumnTransformer([
    ('num', numeric_transformer_boost, numeric_features),
    ('cat', categorical_transformer_boost, categorical_features)
])

# формируем итоговый конвейер для градиентного бустинга
pipe_boost = Pipeline([
    ('preprocessor', preprocessor_boost),
    ('classifier', GradientBoostingClassifier(
        random_state=42, subsample=0.8))
])

```

Теперь задаем сетки значений гиперпараметров для логистической регрессии со стандартизацией, логистической регрессии без стандартизации, градиентного бустинга.

```

# задаем сетки значений гиперпараметров
param_logreg_scaled = [{'classifier__C': [.05, .01]}]
param_logreg_non_scaled = [{'classifier__C': [.05, .01],
                              'preprocessor__num__scaler': [None]}]
param_boost = [{'classifier__max_depth': [2, 4],
                 'classifier__n_estimators': [50, 100]}]

```

Создаем экземпляры класса GridSearchCV и объединяем в список.

```

# создаем экземпляры класса GridSearchCV
gs_logreg_scaled = GridSearchCV(
    pipe_logreg,
    param_logreg_scaled,
    scoring='roc_auc',
    cv=5)

gs_boost = GridSearchCV(
    pipe_boost,
    param_boost,
    scoring='roc_auc',
    cv=5)

gs_logreg_non_scaled = GridSearchCV(
    pipe_logreg,
    param_logreg_non_scaled,
    scoring='roc_auc',
    cv=5)

# объединяем в список
grids = [gs_logreg_scaled, gs_boost, gs_logreg_non_scaled]

```

Создаем словарь для сопоставления индекса с названием метода машинного обучения.

```
# создаем словарь для сопоставления индекса
# с названием метода машинного обучения
grid_dict = {0: 'Логистическая регрессия со стандартизацией',
             1: 'Градиентный бустинг',
             2: 'Логистическая регрессия без стандартизации'}
```

Выполняем поиск по сетке, нужно выбрать оптимальный метод машинного обучения из разных методов машинного обучения со своими значениями гиперпараметров и своей предварительной подготовкой.

```
# печатаем строку статуса
print("Выполняем поиск оптимальных значений гиперпараметров...")
# добавляем пустую строку в вывод
print("")
# создаем пустой список, в который будем записывать наилучшее
# значение AUC по каждому методу машинного обучения
auc_list = []
# здесь будем хранить индекс, соответствующий
# методу машинного обучения
best_clf = 0
# выполняем поиск по сетке
for idx, gs in enumerate(grid_dict):
    # печатаем метод машинного обучения
    print("Метод: %s" % grid_dict[idx])
    gs.fit(tr, y_tr)
    # печатаем наилучшие значения гиперпараметров для этого метода
    print("Наилучшие значения гиперпараметров: %s" % gs.best_params_)
    # печатаем наилучшее значение AUC для этого метода
    print("Наилучшее значение AUC: %.3f" % gs.best_score_)
    # добавляем пустую строку в вывод
    print("")
    # записываем наилучшее значение AUC для метода машинного обучения
    auc_score = gs.best_score_
    # добавляем наилучшее значение AUC в список
    auc_list.append(auc_score)
    # если мы получаем максимальное значение AUC
    if auc_score == max(auc_list):
        # записываем индекс наилучшего метода
        best_clf = idx
        # записываем значения гиперпараметров наилучшего метода
        bst_params = gs.best_params_
        # вычисляем вероятности положительного класса
        proba_tst = gs.predict_proba(tst)[: , 1]
        # вычисляем AUC на тестовой выборке
        auc_tst = roc_auc_score(y_tst, proba_tst)
# добавляем пустую строку в вывод
print("")
# печатаем название лучшего метода, используем словарь
# для сопоставления индекса с названием метода
# машинного обучения
print("Лучший метод машинного обучения: %s" % grid_dict[best_clf])
# печатаем значения гиперпараметров лучшего метода
```

```
print("Значения гиперпараметров лучшего метода: %s" % bst_params)
# печатаем AUC лучшего метода на тестовой выборке
print("AUC лучшего метода на тестовой выборке: %.3f" % auc_tst)
```

Выполняем поиск оптимальных значений гиперпараметров...

Метод: Логистическая регрессия со стандартизацией  
 Наилучшие значения гиперпараметров: {'classifier\_\_C': 0.05}  
 Наилучшее значение AUC-ROC: 0.905

Метод: Градиентный бустинг  
 Наилучшие значения гиперпараметров: {'classifier\_\_max\_depth': 4, 'classifier\_\_n\_estimators': 50}  
 Наилучшее значение AUC-ROC: 0.909

Метод: Логистическая регрессия без стандартизации  
 Наилучшие значения гиперпараметров: {'classifier\_\_C': 0.05, 'preprocessor\_\_num\_scaler': None}  
 Наилучшее значение AUC-ROC: 0.905

Лучший метод машинного обучения: Градиентный бустинг  
 Значения гиперпараметров лучшего метода: {'classifier\_\_max\_depth': 4, 'classifier\_\_n\_estimators': 50}  
 AUC-ROC лучшего метода на тестовой выборке: 0.907

## 7.13. ВЛОЖЕННАЯ ПЕРЕКРЕСТНАЯ ПРОВЕРКА

При использовании комбинированной проверки с помощью классов `GridSearchCV` и `RandomizedSearchCV` мы все еще выполняем всего лишь одно разбиение на обучающую и тестовую выборки, что может привести к получению нестабильных результатов и ставит нас в зависимость от этого единственного разбиения данных. Мы можем пойти дальше и вместо однократного разбиения исходных данных на обучающий и тестовый наборы использовать несколько разбиений. В результате получим *вложенную перекрестную проверку* (*nested cross-validation*). Метод вложенной перекрестной проверки довольно очевиден, поскольку это всего лишь вложение двух циклов  $k$ -блочной перекрестной проверки: внутренний цикл отвечает за выбор модели (настраиваем гиперпараметры), а во внешнем цикле выполняется оценка обобщающей способности.

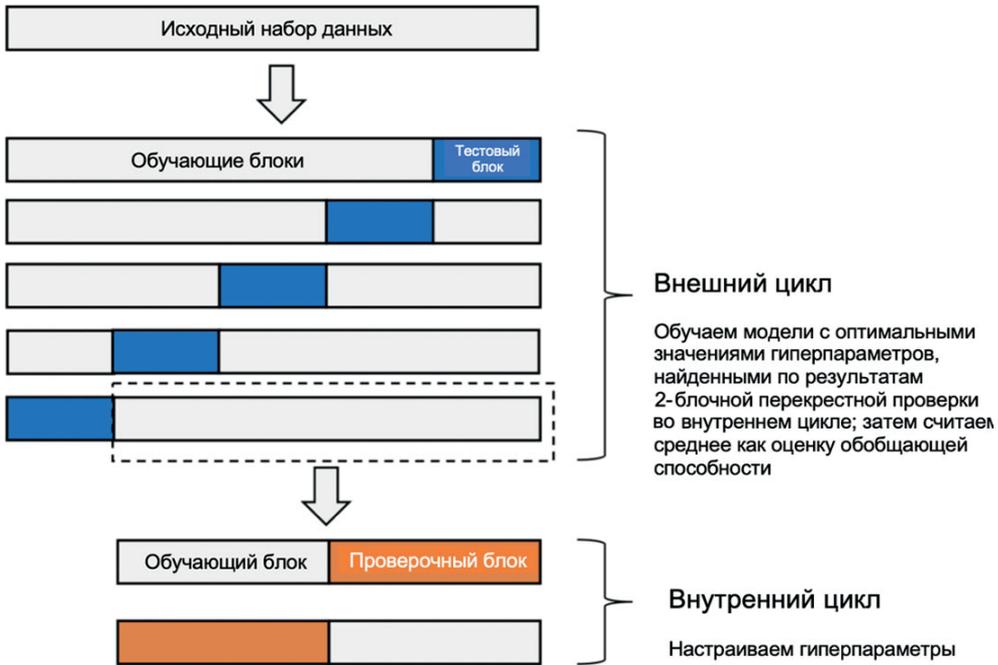


Рис. 47 Вложенная перекрестная проверка

Результатом этой процедуры является не модель, а список значений выбранной метрики (также можно получить список выбираемых лучших комбинаций значений гиперпараметров). Значения метрики указывают нам на обобщающую способность модели с использованием лучших комбинаций значений гиперпараметров, найденных в ходе поиска по сетке. Кроме того, мы можем найти такое значение гиперпараметра или такую комбинацию значений гиперпараметров, которая чаще всего выбирается в качестве наилучшей. Это может быть полезно для поиска гиперпараметров, стабильно дающих высокое качество, при построении моделей с высокой дисперсией типа градиентного бустинга.

Вложенную перекрестную проверку можно реализовать, вызвав функцию `cross_val_score()` и передав ей экземпляр класса `GridSearchCV` в качестве модели. Соответственно, в класс `GridSearchCV` передаем нашу модель машинного обучения или конвейер моделей.

Проиллюстрируем этот вид проверки на задаче кредитного скоринга.

```
# импортируем необходимые библиотеки
import numpy as np
import pandas as pd
# импортируем класс DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# импортируем функцию cross_val_score()
# и классы RepeatedStratifiedKFold,
# ParameterGrid, GridSearchCV
```

```

from sklearn.model_selection import (cross_val_score,
                                     RepeatedStratifiedKFold,
                                     ParameterGrid,
                                     GridSearchCV)

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Bankloan.csv',
                  encoding='cp1251',
                  sep=';', decimal=',')

# создаем массив меток и массив признаков
y = data.pop('default')
X = pd.get_dummies(data)

# создаем экземпляр класса DecisionTreeClassifier
tree = DecisionTreeClassifier(random_state=152)

# задаем сетку гиперпараметров
param_grid = {'max_depth': [4, 8, 12], 'max_features': [3, 6]}

# запускаем вложенную перекрестную проверку
scores = cross_val_score(GridSearchCV(tree, param_grid, cv=5),
                          X, y, cv=5)
print("Значения правильности: ", scores)
print("Среднее значение правильности: ", scores.mean())

```

```

Значения правильности: [0.69333333 0.70333333 0.72333333 0.73666667 0.66333333]
Среднее значение правильности: 0.7040000000000001

```

Допустим, у нас есть набор наблюдений, проиндексированных от 1 до 15 включительно. Мы строим модель дерева, подбираем значения гиперпараметра `max_features` 1, 2, 3. Теперь настраиваем вложенную перекрестную проверку. Мы берем 2-блочную перекрестную проверку во внутреннем цикле и такую же 2-блочную перекрестную проверку во внешнем цикле. Происходящее под капотом вложенной перекрестной проверки можно пояснить нижеприведенной схемой.

```

                                1-я внешняя итерация
outer TRAIN: [ 2  3  4  9 11 12 14] outer TEST: [ 0  1  5  6  7  8 10 13]

                                1-я внутренняя итерация
параметры: {'max_features': 1}
inner TRAIN: [3 9 14] inner TEST: [2 4 11 12]
Качество на inner TRAIN: 0.25

                                2-я внутренняя итерация
параметры: {'max_features': 1}
inner TRAIN: [2 4 11 12] inner TEST: [3 9 14]
Качество на inner TRAIN: 0.3333333333333333

                                1-я внутренняя итерация
параметры: {'max_features': 2}
inner TRAIN: [3 9 14] inner TEST: [2 4 11 12]
Качество на inner TRAIN: 0.5

                                2-я внутренняя итерация
параметры: {'max_features': 2}
inner TRAIN: [2 4 11 12] inner TEST: [3 9 14]
Качество на inner TRAIN: 0.6666666666666666

```

1-я внутренняя итерация  
 параметры: {'max\_features': 3}  
 inner TRAIN: [3 9 14] inner TEST: [2 4 11 12]  
 Качество на inner TRAIN: 0.25

2-я внутренняя итерация  
 параметры: {'max\_features': 3}  
 inner TRAIN: [2 4 11 12] inner TEST: [3 9 14]  
 Качество на inner TRAIN: 1.0

Наилучшее среднее качество по inner TESTs: 0.625  
 Наилучшие гиперпараметры, найденные по inner TESTs: {'max\_features': 3}  
 Качество на outer TEST: 0.625 с гиперпараметрами {'max\_features': 3}

**2-я внешняя итерация**  
 outer TRAIN: [ 0 1 5 6 7 8 10 13] outer TEST: [ 2 3 4 9 11 12 14]

1-я внутренняя итерация  
 параметры: {'max\_features': 1}  
 inner TRAIN: [0 5 7 13] inner TEST: [1 6 8 10]  
 Качество на inner TRAIN: 0.0

2-я внутренняя итерация  
 параметры: {'max\_features': 1}  
 inner TRAIN: [1 6 8 10] inner TEST: [0 5 7 13]  
 Качество на inner TRAIN: 0.25

1-я внутренняя итерация  
 параметры: {'max\_features': 2}  
 inner TRAIN: [0 5 7 13] inner TEST: [1 6 8 10]  
 Качество на inner TRAIN: 0.25

2-я внутренняя итерация  
 параметры: {'max\_features': 2}  
 inner TRAIN: [1 6 8 10] inner TEST: [0 5 7 13]  
 Качество на inner TRAIN: 0.5

1-я внутренняя итерация  
 параметры: {'max\_features': 3}  
 inner TRAIN: [0 5 7 13] inner TEST: [1 6 8 10]  
 Качество на inner TRAIN: 0.5

2-я внутренняя итерация  
 параметры: {'max\_features': 3}  
 inner TRAIN: [1 6 8 10] inner TEST: [0 5 7 13]  
 Качество на inner TRAIN: 0.5

Наилучшее среднее качество по inner TESTs: 0.5  
 Наилучшие гиперпараметры, найденные по inner TESTs: {'max\_features': 3}  
 Качество на outer TEST: 0.429 с гиперпараметрами {'max\_features': 3}

Среднее значение правильности:  
 $(0.625 + 0.429) / 2 = 0.527$

Здесь мы видим, что у нас две внешние итерации, внутри которых на каждое значение гиперпараметра (или каждую комбинацию значений гиперпараметров) приходится две внутренние итерации. Проще говоря, для каждого значения гиперпараметра / каждой комбинации мы получим два значения метрики на основе внутренних тестовых выборок. Мы находим наилучшее значение гиперпараметра (или наилучшую комбинацию значений гиперпараметров), дающее наилучшее значение метрики, усредненное по внутренним тестовым выборкам. С найденным наилучшим значением гиперпараметра (или наилучшей комбинацией значений гиперпараметров) обучаем модель на внешней обучающей выборке и проверяем на внешней тестовой выборке, записываем значение метрики для внешней тестовой выборки. Итоговое среднее значение метрики (в нашем случае – среднее значение правильности) – это значение метрики, усредненное по значениям метрики, полученным для внешних тестовых выборок.

На практике пишут собственную реализацию вложенной перекрестной проверки (обычно во внешнем и внутреннем циклах используют повторную  $k$ -блочную стратифицированную проверку), возвращающую наилучшее значение метрики и наилучшие гиперпараметры для внешних тестовых выборок. Как уже было сказано вначале, здесь нас интересует значение гиперпараметра или комбинация значений гиперпараметров, чаще всего выбираемая в качестве наилучшей.

```
# пишем функцию, выполняющую вложенную
# перекрестную проверку
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    """
    Автор: Andreas Mueller <t3kcit@gmail.com>

    Выполняет вложенную перекрестную проверку.

    Параметры
    -----
    X: numpy.ndarray
        Массив признаков.
    y: numpy.ndarray
        Массив меток.
    inner_cv: int
        Количество итераций внутренней
        перекрестной проверки.
    outer_cv: int
        Количество итераций внешней
        перекрестной проверки.
    Classifier: instance of class sklearn
        Классификатор.
    parameter_grid: dict of str
        Сетка гиперпараметров.

    Возвращает
    -----
    results: pandas.DataFrame
        Датафрейм с наилучшими гиперпараметрами
        и наилучшим значением метрики.
    """
    # собираем наилучшие значения правильности
    # и наилучшие гиперпараметры
    outer_scores_lst = []
    best_params_lst = []
    # для каждого разбиения данных во внешней перекрестной проверке
    # (метод split возвращает индексы)
    for outer_cnt, (training_samples, test_samples) in enumerate(
        outer_cv.split(X, y), 1):
        # находим наилучшие значения гиперпараметров
        # с помощью внутренней перекрестной проверки
        best_params = {}
        best_score = -np.inf
        # итерируем по гиперпараметрам
        for parameters in parameter_grid:
            # собираем значения правильности
            # по всем внутренним разбиениям
            cv_scores = []
```

```

# итерируем по разбиениям внутренней перекрестной проверки
for inner_cnt, (inner_train, inner_test) in enumerate(
    inner_cv.split(X[training_samples], y[training_samples]), 1):
    # строим классификатор с данными значениями гиперпараметров
    # на внутренней обучающей выборке
    clf = Classifier(**parameters)
    clf.fit(X[inner_train], y[inner_train])
    # оцениваем качество на внутренней тестовой выборке
    score = clf.score(X[inner_test], y[inner_test])
    cv_scores.append(score)
# вычисляем среднее значение правильности
# по внутренним тестовым выборкам
mean_score = np.mean(cv_scores)
if mean_score > best_score:
    # если лучше, чем предыдущие, записываем
    # значения гиперпараметров как лучшие
    best_score = mean_score
    best_params = parameters
# строим классификатор с наилучшими значениями
# гиперпараметров на внешней обучающей выборке
clf = Classifier(**best_params)
clf.fit(X[training_samples], y[training_samples])
# оцениваем качество на внешней тестовой выборке
out_score = clf.score(X[test_samples], y[test_samples])
# добавляем лучшие значения метрики
# и лучшие гиперпараметры в списки
outer_scores_lst.append(out_score)
best_params_lst.append(best_params)
# создаем словарь с наилучшими гиперпараметрами
# и наилучшими значениями метрики
results = pd.DataFrame(best_params_lst)
results['score'] = outer_scores_lst
return results

```

Сейчас мы зададим новую сетку гиперпараметров, во внутреннем и внешнем циклах зададим 5-повторную 5-блочную стратифицированную перекрестную проверку, запустим нашу вложенную перекрестную проверку и посмотрим, какая наилучшая комбинация гиперпараметров выбирается чаще всего.

```

# задаем сетку гиперпараметров
param_grid = {'max_features': [1, 2, 3],
              'max_depth': [2, 4, 6]}

# применяем нашу функцию
scores = nested_cv(X.values,
                  y.values,
                  RepeatedStratifiedKFold(n_repeats=5,
                                           n_splits=5,
                                           random_state=42),
                  RepeatedStratifiedKFold(n_repeats=5,
                                           n_splits=5,
                                           random_state=42),
                  DecisionTreeClassifier,
                  ParameterGrid(param_grid))

```

scores

	max_depth	max_features	score
0	6	3	0.693333
1	4	3	0.713333
2	6	3	0.726667
3	4	3	0.713333
4	6	3	0.673333
5	6	3	0.703333
6	4	3	0.733333
7	6	3	0.656667
8	6	3	0.656667
9	6	3	0.710000
10	6	3	0.723333
11	6	3	0.760000
12	6	3	0.676667
13	6	3	0.716667
14	6	2	0.673333
15	4	3	0.690000
16	6	3	0.700000
17	4	3	0.713333
18	6	3	0.713333
19	6	3	0.670000
20	4	3	0.680000
21	6	3	0.706667
22	6	3	0.706667
23	6	3	0.640000
24	6	3	0.723333

Мы видим, что среди наилучших комбинаций никогда не встречается значение `max_features`, равное 1, один раз встречается значение `max_features`, равное 2, никогда не встречается значение `max_depth`, равное 2. Чаще всего выбирается комбинация со значением `max_depth`, равным 6, и значением `max_features`, равным 3.

## 7.14. КЛАССЫ POWERTRANSFORMER, KBINSDISCRETIZER И FUNCTIONTRANSFORMER

Класс `PowerTransformer` позволяет применить преобразование Бокса–Кокса или преобразование Йео–Джонсона к выбранным переменным. Эти преобразования позволяют максимизировать нормальность распределения переменных и тем самым улучшить качество линейных моделей. Задача заключается в том, чтобы найти такое преобразование эмпирического (исходного) распределения, чтобы оно максимально соответствовало теоретическому (нормальному) распределению. Под капотом происходит поиск наилучшего значения параметра (лямбда), соответствующего тому или иному преобразованию (логарифм, квадратный корень и прочие), затем это преобразование применяется к переменным обуча-

ющей и тестовой выборки. Преобразование Бокса–Кокса применяется только к положительным значениям, а преобразование Йео–Джонсона можно применять как к положительным, так и к отрицательным значениям. Кроме того, класс `PowerTransformer` позволяет выполнить еще и стандартизацию (заменяет `StandardScaler`). Ниже подробно разобраны параметры класса `PowerTransformer`.

```
from sklearn.preprocessing import PowerTransformer(method='yeo-johnson', ← Задаёт преобразование
                                                    'box-cox'
                                                    standardize=True, ← Задаёт стандартизацию
                                                    copy=True) ← Если задано False, вычисления осуществляются
                                                         на месте в ходе преобразования
```

Класс `KBinsDiscretizer` позволяет выполнить биннинг (категоризацию) количественных переменных. Например, переменную *Возраст* со значениями от 20 до 60 лет можно разбить на три категории: *моложе 30 лет*, *от 30 до 50 лет включительно*, *старше 50 лет*. Это позволяет учитывать нелинейность, бороться с выбросами и может улучшить качество линейных моделей. Ниже подробно разобраны параметры класса `KBinsDiscretizer`.

```
from sklearn.preprocessing import KBinsDiscretizer(1 n_bins=5,
                                                    2 encode='onehot',
                                                    'onehot-dense'
                                                    'ordinal'
                                                    3 strategy='quantile',
                                                    'uniform'
                                                    'kmeans'
                                                    4 subsample=None)
```

- 1 Задаёт количество бинов (категорий)
- 2 Задаёт способ кодировки. По умолчанию используется 'onehot', применяется дамми-кодирование, возвращающее разреженную матрицу. Если задано 'onehot-dense', применяется дамми-кодирование, возвращающее плотный массив. Если задано 'ordinal', бины кодируются целочисленными значениями
- 3 Задаёт стратегию биннинга. По умолчанию используется 'quantile', все бины имеют примерно одинаковое количество наблюдений. Если задано 'uniform', все бины имеют одинаковую ширину. Если задано 'kmeans', наблюдения в каждом бине имеют один и тот же ближайший центр
- 4 Максимальное количество наблюдений, используемое для вычисления бинов. Применяется, когда strategy='quantile'. subsample=None означает, что все наблюдения обучающей выборки используются при вычислении бинов. Для наборов с очень большим количеством наблюдений рекомендуется использовать подвыборку наблюдений

Класс `FunctionTransformer` позволяет использовать встроенные или пользовательские функции внутри конвейеров. Ниже подробно разобраны параметры класса `FunctionTransformer`.

```
from sklearn.preprocessing import FunctionTransformer(func=None, ← Задаёт функцию
                                                    validate=None)
                                                    Задаёт проверку входного массива X →
                                                    перед вызовом функции. Если задано
                                                    False, проверка входного массива X не
                                                    выполняется. Если задано True, X будет
                                                    преобразован в 2D-массив NumPy или
                                                    разреженную матрицу
```

Теперь потренируемся использовать классы `PowerTransformer`, `KBinsDiscretizer` и `FunctionTransformer`.

```
# импортируем необходимые библиотеки, функции и классы
import pandas as pd
import numpy as np
from sklearn.model_selection import (train_test_split,
                                     GridSearchCV)
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (PowerTransformer,
                                   FunctionTransformer,
                                   KBinsDiscretizer,
                                   OneHotEncoder)
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Vodafone_missing.csv', sep=';')
data.head(5)
```

	region	tenure	age	marital	address	income	employ	retire	gender	reside	custcat	churn
0	Region 2	13.0	44.0	mar	9.0	64.0	5.0	no	f	2.0	cat 1	1
1	Region 3	11.0	33.0	mar	7.0	136.0	5.0	no	f	6.0	cat 4	1
2	Region 3	68.0	52.0	mar	24.0	116.0	29.0	no	NaN	NaN	NaN	0
3	Region 2	NaN	33.0	NaN	12.0	NaN	NaN	no	NaN	1.0	cat 1	1
4	Region 2	23.0	30.0	mar	9.0	30.0	2.0	no	f	4.0	cat 3	0

Исходный набор содержит данные немецкого провайдера Vodafone. Он представляет собой записи о 1000 клиентов, классифицированных на два класса: 0 – оттока нет (726 клиентов) и 1 – отток есть (274 клиента). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- категориальный признак *Географический регион [region]*;
- количественный признак *Время пользования услугами в месяцах [tenure]*;
- количественный признак *Возраст [age]*;
- категориальный признак *Семейное положение [marital]*;
- количественный признак *Время проживания по последнему адресу [address]*;
- количественный признак *Доход домохозяйства в тысячах [income]*;
- количественный признак *Время работы на последнем месте работы [employ]*;
- категориальный признак *Уход на пенсию [retire]*;
- категориальный признак *Пол [gender]*;
- количественный признак *Количество членов домохозяйства [reside]*;
- категориальный признак *Потребительская категория [custcat]*;
- бинарная зависимая переменная *Отказ от пользования услуг [churn]*.

Необходимо классифицировать клиентов на лояльных и ушедших. Метрика качества – правильность.

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('churn', axis=1),
    data['churn'],
    test_size=0.3,
    stratify=data['churn'],
    random_state=42)

```

Мы создадим список категориальных признаков, список количественных признаков, не предназначенных для биннинга, и списки количественных признаков, предназначенных для биннинга.

```

# создаем список категориальных признаков, список
# количественных признаков, не предназначенных для
# биннинга, список количественных признаков,
# предназначенных для биннинга
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()
age_col = ['age']
tenure_col = ['tenure']

```

Теперь напишем собственную функцию `discretize()`, выполняющую биннинг. Функция возвращает целочисленные индексы бинов.

```

# пишем функцию биннинга
def discretize(x):
    bins = [-np.inf, 20, 40, np.inf]
    return np.digitize(x, bins)

```

Создаем конвейеры. При создании конвейера для количественных признаков, которые не будут подвергнуты биннингу, воспользуемся классом `PowerTransformer`, он применит преобразование Йео–Джонсона, а затем выполнит стандартизацию. При создании конвейера для количественного признака `age`, который будет подвергнут биннингу, воспользуемся классом `KBinsDiscretizer`, специально предназначенным для выполнения биннинга. Он сразу выполняет биннинг и дамми-кодирование. При создании конвейера для количественного признака `tenure`, который также будет подвергнут биннингу, воспользуемся классом `FunctionTransformer`, указав написанную функцию `discretize()`, а затем выполняем дамми-кодирование с помощью класса `OneHotEncoder`. Для категориальных признаков мы применяем дамми-кодирование с помощью класса `OneHotEncoder`. Для всех конвейеров на первом этапе применяется импутация пропусков с помощью класса `SimpleImputer`.

```

# создаем конвейер для количественных переменных,
# которые не будут подвергнуты биннингу
num_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('yeo_john', PowerTransformer(method='yeo-johnson',
    standardize=True))
])

```

```

# создаем конвейер для age, которая будет
# подвергнута биннингу
age_bin_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('disc', KBinsDiscretizer(encode='onehot-dense'))
])

# создаем конвейер для tenure, которая будет
# подвергнута биннингу
tenure_bin_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('cust_disc', FunctionTransformer(discretize, validate=False)),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])

```

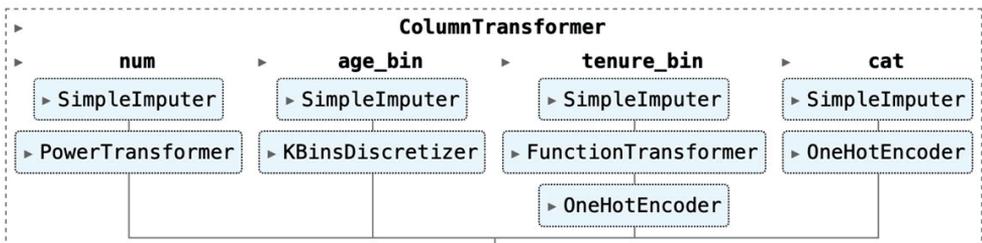
Создаем список трансформеров, передаем этот список в ColumnTransformer и смотрим полученный экземпляр класса ColumnTransformer.

```

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
               ('age_bin', age_bin_pipe, age_col),
               ('tenure_bin', tenure_bin_pipe, tenure_col),
               ('cat', cat_pipe, cat_columns)]

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)
transformer

```



Щелкнем по стрелке слева от `ColumnTransformer` и получаем следующий вывод (модифицирован для удобства восприятия).

```

ColumnTransformer(transformers=[
    ('num', Pipeline(steps=[('imp', SimpleImputer()),
                           ('yeo_john', PowerTransformer())])),
    ('tenure', 'age', 'address', 'income', 'employ', 'reside'),
    ('age_bin', Pipeline(steps=[('imp', SimpleImputer()),

```

```

        ('disc', KBinsDiscretizer(
            encode='onehot-dense'))]),
    ['age']],
    ('tenure_bin', Pipeline(steps=[('imp', SimpleImputer()),
        ('cust_disc', FunctionTransformer(
            func=<function discretize at 0x7fcc2023aca0>)),
        ('ohe', OneHotEncoder(handle_unknown='ignore',
            sparse=False))])),
    ['tenure']],
    ('cat', Pipeline(steps=[('imp', SimpleImputer()),
        ('ohe', OneHotEncoder(handle_unknown='ignore',
            sparse=False))])),
    ['region', 'marital', 'retire', 'gender', 'custcat']]])

```

Видим, что переменные *tenure* и *age* будут использованы как в виде количественных переменных, так и в виде дамми-переменных по итогам биннинга.

Теперь формируем итоговый конвейер.

*# задаем итоговый конвейер*

```

ml_pipe = Pipeline([
    ('tr', transformer),
    ('lr', LogisticRegression(C=0.1,
        solver='lbfgs',
        max_iter=200))
])

```

Задаем сетку гиперпараметров. Обратите внимание, вы можете передать параметры функции, заданной с помощью класса `FunctionTransformer`, в виде гиперпараметров. Название гиперпараметра – параметра функции, заданной с помощью класса `FunctionTransformer`, будет представлять собой название этапа итогового конвейера\_\_название конвейера с классом `FunctionTransformer`\_\_название этапа конвейера, где используется класс `FunctionTransformer`\_\_kw\_args.

```

# создаем конвейер для tenure, которая будет
# подвергнута биннингу
tenure_bin_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('cust_disc', FunctionTransformer(discretize, validate=False)),
    ('ohe', OneHotEncoder(sparse=False, handle_unknown='ignore'))
])

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
                ('age_bin', age_bin_pipe, age_col),
                ('tenure_bin', tenure_bin_pipe, tenure_col),
                ('cat', cat_pipe, cat_columns)]

# передаем список трансформеров в ColumnTransformer
transformer = ColumnTransformer(transformers=transformers)

# задаем итоговый конвейер
ml_pipe = Pipeline([
    ('tr', transformer),
    ('lr', LogisticRegression(C=0.1,
        solver='lbfgs',
        max_iter=200))
])

```

название этапа итогового конвейера

название этапа конвейера, где используется класс FunctionTransformer

название конвейера с классом FunctionTransformer

'tr\_tenure\_bin\_cust\_disc\_kw\_args'

Перебираемые значения гиперпараметров оформляем в список словарей, где ключом будет название гиперпараметра, значением – значение гиперпараметра.

```

[{'hyperparameter_1': 2}, {'hyperparameter_1': 3}]

```

ИЛИ

```
{'hyperparameter_1': 4, 'hyperparameter_2': 1},
 {'hyperparameter_1': 4, 'hyperparameter_2': 2}]
```

```
# задаем сетку гиперпараметров
param_grid = {
    'tr_tenure_bin_cust_disc_kw_args': [
        {'bins': [-np.inf, 25, 40, np.inf]},
        {'bins': [-np.inf, 30, 40, 50, np.inf]},
        {'bins': [-np.inf, 25, 45, 55, np.inf]}
    ],
    'tr_num_imp_strategy': ['mean', 'median', 'constant'],
    'tr_age_bin_disc_n_bins': [2, 3],
    'tr_cat_imp_strategy': ['most_frequent', 'constant']
}
```

Выполняем поиск по сетке и смотрим результаты.

```
# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество
# блоков перекрестной проверки
gs = GridSearchCV(ml_pipe, param_grid, cv=5)
# выполняем поиск по сетке
gs.fit(X_train, y_train)
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs.best_params_))
# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    gs.best_score_))
# смотрим значение правильности на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    gs.score(X_test, y_test)))
```

Наилучшие значения гиперпараметров:

```
{'tr_age_bin_disc_n_bins': 3, 'tr_cat_imp_strategy': 'most_frequent', 'tr_num__
imp_strategy': 'constant', 'tr_tenure_bin_cust_disc_kw_args': {'bins': [-inf, 30,
40, 50, inf]}}
```

Наилучшее значение правильности: 0.763

Значение правильности на тестовой выборке: 0.760

Результаты перекрестной проверки, сохраненные в атрибуте `cv_results_`, записываем в объект `DataFrame` и с помощью метода `.pivot_table()` превращаем в сводную таблицу.

```
# запишем результаты перекрестной
# проверки в DataFrame
results = pd.DataFrame(gs.cv_results_)
# отбираем нужные столбцы
pattern = results.columns[results.columns.str.contains(
    'param_')].tolist() + ['mean_test_score']
table = results[pattern]
```

```
# сортируем по убыванию правильности
```

```
table = table.sort_values(by='mean_test_score',
                          ascending=False)
```

	param_tr_age_bin_disc_n_bins	param_tr_cat_imp_strategy	param_tr_num_imp_strategy	param_tr_tenure_bin_cust_disc_kw_args	mean_test_score
25	3	most_frequent	constant	('bins': [-inf, 30, 40, 50, inf])	0.762857
34	3	constant	constant	('bins': [-inf, 30, 40, 50, inf])	0.757143
8	2	most_frequent	constant	('bins': [-inf, 25, 45, 55, inf])	0.755714
19	3	most_frequent	mean	('bins': [-inf, 30, 40, 50, inf])	0.755714
22	3	most_frequent	median	('bins': [-inf, 30, 40, 50, inf])	0.755714
24	3	most_frequent	constant	('bins': [-inf, 25, 40, inf])	0.755714
20	3	most_frequent	mean	('bins': [-inf, 25, 45, 55, inf])	0.754286
23	3	most_frequent	median	('bins': [-inf, 25, 45, 55, inf])	0.754286
2	2	most_frequent	mean	('bins': [-inf, 25, 45, 55, inf])	0.754286
33	3	constant	constant	('bins': [-inf, 25, 40, inf])	0.754286
29	3	constant	mean	('bins': [-inf, 25, 45, 55, inf])	0.754286
7	2	most_frequent	constant	('bins': [-inf, 30, 40, 50, inf])	0.754286
26	3	most_frequent	constant	('bins': [-inf, 25, 45, 55, inf])	0.754286
21	3	most_frequent	median	('bins': [-inf, 25, 40, inf])	0.752857
18	3	most_frequent	mean	('bins': [-inf, 25, 40, inf])	0.752857
27	3	constant	mean	('bins': [-inf, 25, 40, inf])	0.752857
32	3	constant	median	('bins': [-inf, 25, 45, 55, inf])	0.752857
28	3	constant	mean	('bins': [-inf, 30, 40, 50, inf])	0.752857
35	3	constant	constant	('bins': [-inf, 25, 45, 55, inf])	0.752857
16	2	constant	constant	('bins': [-inf, 30, 40, 50, inf])	0.752857
14	2	constant	median	('bins': [-inf, 25, 45, 55, inf])	0.752857
13	2	constant	median	('bins': [-inf, 30, 40, 50, inf])	0.752857
11	2	constant	mean	('bins': [-inf, 25, 45, 55, inf])	0.752857
17	2	constant	constant	('bins': [-inf, 25, 45, 55, inf])	0.752857
5	2	most_frequent	median	('bins': [-inf, 25, 45, 55, inf])	0.752857
15	2	constant	constant	('bins': [-inf, 25, 40, inf])	0.751429
12	2	constant	median	('bins': [-inf, 25, 40, inf])	0.751429
3	2	most_frequent	median	('bins': [-inf, 25, 40, inf])	0.751429
6	2	most_frequent	constant	('bins': [-inf, 25, 40, inf])	0.751429
30	3	constant	median	('bins': [-inf, 25, 40, inf])	0.751429
31	3	constant	median	('bins': [-inf, 30, 40, 50, inf])	0.751429
1	2	most_frequent	mean	('bins': [-inf, 30, 40, 50, inf])	0.750000
4	2	most_frequent	median	('bins': [-inf, 30, 40, 50, inf])	0.750000
10	2	constant	mean	('bins': [-inf, 30, 40, 50, inf])	0.750000
0	2	most_frequent	mean	('bins': [-inf, 25, 40, inf])	0.750000
9	2	constant	mean	('bins': [-inf, 25, 40, inf])	0.748571

Теперь извлекаем списки переменных, полученные с помощью классов `OneHotEncoder` и `KBinsDiscretizer`, добавляем к списку количественных переменных. Помним, что добавляем списки, сгенерированные трансформерами, в том же порядке, в котором следуют наши трансформеры.

```
# извлекаем дамми-переменные для списка с переменной age
```

```
age = gs.best_estimator_['tr'].named_transformers_['age_bin']
age_ohc_columns = list(age.named_steps['disc'].get_feature_names_out(
    input_features=age_col))
age_ohc_columns
```

```
['age_0.0', 'age_1.0', 'age_2.0']
```

```

# извлекаем дамми-переменные для списка с переменной tenure
tenure = gs.best_estimator_['tr'].named_transformers_['tenure_bin']
tenure_ohc_columns = list(tenure.named_steps['ohc'].get_feature_names_out(
    input_features=tenure_col))
tenure_ohc_columns

['tenure_1', 'tenure_2', 'tenure_3', 'tenure_4']

# извлекаем дамми-переменные для списка
# категориальных признаков
cat = gs.best_estimator_['tr'].named_transformers_['cat']
cat_ohc_columns = list(cat.named_steps['ohc'].get_feature_names_out(
    input_features=cat_columns))
cat_ohc_columns

['region_Region 1',
 'region_Region 2',
 'region_Region 3',
 'marital_mar',
 'marital_unmar',
 'retire_no',
 'retire_yes',
 'gender_f',
 'gender_m',
 'custcat_cat 1',
 'custcat_cat 2',
 'custcat_cat 3',
 'custcat_cat 4']

# добавляем к списку количественных переменных
# остальные списки
all_cols = (num_columns + age_ohc_columns +
            tenure_ohc_columns + cat_ohc_columns)

```

Извлекаем константу, коэффициенты наилучшей модели логистической регрессии и с помощью функции `zip()` «сшиваем» константу и коэффициенты с названиями признаков.

```

# извлекаем константу
intercept = np.round(gs.best_estimator_['lr'].intercept_[0], 3)
# извлекаем коэффициенты
coef = np.round(gs.best_estimator_['lr'].coef_, 3)

# печатаем название "Константа"
print("Константа:", intercept)
# печатаем название "Регрессионные коэффициенты"
print("Регрессионные коэффициенты:")
# для удобства сопоставим каждому названию
# признака соответствующий коэффициент
for c, feature in zip(coef[0], all_cols):
    print(feature, c)

```

```

Константа: -1.173
Регрессионные коэффициенты:
tenure -0.78
age -0.18

```

```

address -0.008
income 0.174
employ -0.244
reside -0.105
age_0.0 -0.03
age_1.0 -0.062
age_2.0 0.092
tenure_1 0.074
tenure_2 -0.112
tenure_3 0.153
tenure_4 -0.114
region_Region 1 -0.033
region_Region 2 0.058
region_Region 3 -0.025
marital_mar 0.044
marital_unmar -0.044
retire_no 0.014
retire_yes -0.014
gender_f -0.029
gender_m 0.029
custcat_cat 1 -0.25
custcat_cat 2 0.348
custcat_cat 3 -0.438
custcat_cat 4 0.34

```

## 7.15. НАПИСАНИЕ СОБСТВЕННЫХ КЛАССОВ ПРЕДВАРИТЕЛЬНОЙ ПОДГОТОВКИ ДЛЯ ПРИМЕНЕНИЯ В КОНВЕЙЕРЕ

До сих пор мы строили модели предварительной подготовки и машинного обучения, пользуясь готовыми классами библиотеки `scikit-learn`. Теперь давайте научимся писать собственные классы и применять их в конвейерах.

Напомним: самый простой способ реализовать класс, строящий модель предварительной подготовки, – воспользоваться наследованием базовых классов `BaseEstimator` и `TransformerMixin`. Класс должен содержать методы `__init__`, `.fit()` и `.transform()`.

Давайте импортируем необходимые библиотеки, классы и функции, загрузим расширенные данные компании `StateFarm` (в качестве метрики качества выберем `AUC-ROC`) и потренируемся в написании классов, выполняющих предварительную подготовку данных.

```

# импортируем необходимые библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.model_selection import (train_test_split,
                                     GridSearchCV)
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (PowerTransformer,
                                   OneHotEncoder,
                                   StandardScaler)
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

```

```
from sklearn.metrics import roc_auc_score

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')
```

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)
```

Часто нам придется пользоваться логарифмическим преобразованием, чтобы приблизить распределение переменной к нормальному. Однако если это преобразование делать «в лоб», с ним будет много хлопот. Во-первых, логарифм отрицательных чисел и нуля не определен, и нужно прибавлять к исходному значению переменной константу, однако если у вас есть отрицательные числа (например, отрицательные остатки на счетах) и при этом в обучающей и тестовой выборках они сильно отличаются друг от друга, то константы, вычисленной для переменной в обучающей выборке, может «не хватить», чтобы перевести отрицательное значение переменной в тестовой выборке в положительное. Например, в обучающей выборке есть значение  $-1$ , взяли логарифм с использованием константы  $1,01$ :  $\log(-1 + 1,01) = \log(0,01) = -1,99$ . А теперь представим, что у этой переменной в тестовой выборке есть значение  $-10$ :  $\log(-10 + 1,01) = \log(-8,99)$ , и получаем значение NaN. Но и это еще не все. Когда мы добавляем к исходному значению константу, выходит, что получаемое распределение будет зависеть от того, насколько исходное значение велико по сравнению с константой  $1,01$ .

Решают эти проблемы следующим образом: пишут свой собственный класс, который с помощью метода `.clip()` библиотеки `pandas` или функции `Numpy` `np.clip()` приравнивает все отрицательные и нулевые значения к определенному нижнему пороговому значению (обычно очень маленькое положительное число), делит полученное значение на среднее и добавляет небольшое значение  $k$  (меньше 1), которое работает как корректор асимметрии распределения. Маленькие значения  $k$  делают данные более скошенными влево, а большие значения  $k$  делают данные менее скошенными. Единственный параметр нашей модели, который нужно вычислить, – это среднее значение признака.  $k$  может быть гиперпараметром, значения которого можно перебирать по сетке.

Итак, пишем класс `CorrLogTransformer`. В методе `__init__` мы выполняем инициализацию атрибутов класса `self.lower` и `self.k`.

```

# создаем собственный класс, выполняющий скорректированное
# логарифмическое преобразование
class CorrLogTransformer(BaseEstimator, TransformerMixin):
    """
    Приравнивает значение признака к нижнему пороговому
    значению – очень маленькому положительному числу,
    делит его на среднее признака и добавляет значение
    от 0 до 1 – коррективу асимметрии, берет
    натуральный логарифм полученного результата.

    Параметры:
    lower: float, по умолчанию 0.001
        Нижнее пороговое значение.
    k: float, по умолчанию 0.2
        Корректировка асимметрии.
    copy: bool, по умолчанию True
        Возвращает копию.
    """
    def __init__(self, lower=0.001, k=0.2, copy=True):
        # все параметры для инициализации публичных атрибутов
        # должны быть заданы в методе __init__

        # публичные атрибуты
        self.lower = lower
        self.k = k
        self.copy = copy

```

Метод `__init__`, задающий конструктор класса (инициализация атрибутов)

Добавляем параметры `lower`, `k` и `copy` в определение метода `__init__`

Используем значения `lower`, `k` и `copy` для инициализации публичных атрибутов класса (которые представлены как `self.lower`, `self.k` и `self.copy`)

Рис. 48 Метод `__init__` класса `CorrLogTransformer`

Метод `.fit()` вычисляет средние значения переменных. Обратите внимание, чтобы проверить, является ли объект датафреймом `pandas`, здесь мы уже используем функцию `isinstance()` в явном виде.

```

# создаем собственный класс, выполняющий скорректированное
# логарифмическое преобразование
class CorrLogTransformer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        # fit должен принимать в качестве аргументов только X и y,
        # даже если ваша модель является моделью машинного обучения
        # без учителя, вы должны принять аргумент y, это требуется
        # для совместимости с конвейерами!

        # обучение модели осуществляется прямо здесь

        # создаем пустой словарь, в котором ключами
        # будут имена/коды числа, а значениями – средние
        self._encoder_dict = {}

        # функция isinstance() проверяет, является
        # объект датафреймом pandas или нет
        if isinstance(X, pd.DataFrame):
            for col in X.columns:
                # вычисляем среднее и записываем в словарь
                self._encoder_dict[col] = np.mean(X[col])
            else:
                for col in range(X.shape[1]):
                    # вычисляем среднее и записываем в словарь
                    self._encoder_dict[col] = np.mean(X[:, col])

        # fit возвращает self
        return self

```

Метод `fit`, задающий обучение модели

Вычисляем единственный параметр модели – среднее значение переменной

Рис. 49 Метод `fit` класса `CorrLogTransformer`

Метод `.transform()` применяет преобразования с помощью вычисленных средних значений.

```

# создаем собственный класс, выполняющий скорректированное
# логарифмическое преобразование
class CorrLogTransformer(BaseEstimator, TransformerMixin):
    def transform(self, X):
        # выполняем копирование массива во избежание
        # предупреждения SettingWithCopyWarning
        # "A value is trying to be set on a copy of a slice
        # from a DataFrame (Происходит попытка изменить
        # значение в копии среза данных датафрейма)"
        if self.copy:
            X = X.copy()

        # проверяем, является ли X pandas DataFrame
        if isinstance(X, pd.DataFrame):
            col = X.columns
            X[col] = (np.log(np.clip(X[col], self.lower,
                                   self.encoder_dict[col] + self.k)))
        else:
            col = (X.shape[0]):
            X[:, col] = (np.log(np.clip(X[:, col], self.lower,
                                       self.encoder_dict[col] + self.k)))

        X

```

Метод transform, задающий применение модели

Применяем модель, т.е. выполняем преобразование с помощью вычисленного параметра модели — среднего значения переменной

Рис. 50 Метод transform класса CorrLogTransformer

Теперь создаем игрушечные обучающий и тестовый наборы данных и проверяем на них наш класс. Качественно написанный класс должен работать как с датафреймами pandas, так и с массивами NumPy. Начнем с игрушечных датафреймов pandas.

```

# создаем игрушечный обучающий датафрейм pandas
train = pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]})
train

```

	Balance	Age
0	8.3	23
1	9.4	29
2	10.2	36
3	0.0	44

```

# создаем игрушечный тестовый датафрейм pandas
test = pd.DataFrame(
    {'Balance': [10.4, 3.1, 22.5, -1],
     'Age': [13, 19, 66, 33]})
test

```

	Balance	Age
0	10.4	13
1	3.1	19
2	22.5	66
3	-1.0	33

Теперь посмотрим, какими должны быть значения в обучающем и тестовом датафреймах pandas. По умолчанию в качестве нижнего порогового значения мы выбрали 0,001, а значение  $k$  установили равным 0,2.

```

# смотрим, как будут выглядеть преобразования в игрушечных
# обучающем и тестовом датафреймах pandas
print(np.log((train['Balance'].clip(0.001) /
              train['Balance'].mean()) + 0.2))
print(np.log((train['Age'].clip(0.001) /
              train['Age'].mean()) + 0.2))
print("")
print(np.log((test['Balance'].clip(0.001) /
              train['Balance'].mean()) + 0.2))
print(np.log((test['Age'].clip(0.001) /
              train['Age'].mean()) + 0.2))

```

```

0    0.329278
1    0.436751
2    0.508242
3   -1.608721
Name: Balance, dtype: float64
0   -0.108733
1    0.075838
2    0.255347
3    0.427444
Name: Age, dtype: float64

```

```

0    0.525343
1   -0.439367
2    1.231337
3   -1.608721
Name: Balance, dtype: float64
0   -0.520978
1   -0.253915
2    0.788457
3    0.182322
Name: Age, dtype: float64

```

Создаем экземпляр нашего класса `CorrLogTransformer`, обучаем и применяем.

```

# создаем экземпляр класса CorrLogTransformer
corrlog = CorrLogTransformer()
# обучаем модель
corrlog.fit(train)
# выполняем преобразование игрушечного
# обучающего датафрейма pandas
train = corrlog.transform(train)
train

```

	Balance	Age
0	0.329278	-0.108733
1	0.436751	0.075838
2	0.508242	0.255347
3	-1.608721	0.427444

```
# выполняем преобразование игрушечного
# тестового датафрейма pandas
test = corrlog.transform(test)
test
```

	Balance	Age
0	0.525343	-0.520978
1	-0.439367	-0.253915
2	1.231337	0.788457
3	-1.608721	0.182322

Видим, что значения, полученные с помощью класса `CorrLogTransformer`, совпадают с ожидаемыми. Теперь применим наш класс для отдельной переменной датафрейма, отключив копирование.

```
# создаем игрушечный обучающий датафрейм pandas
train = pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]})
# создаем экземпляр класса, отключив копирование
corrlog = CorrLogTransformer(copy=False)
# обучаем модель
corrlog.fit(train[['Age']])
# применяем модель
train['Age'] = corrlog.transform(train[['Age']])
train
```

```
/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:57: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame. Try using
.loc[row_indexer,col_indexer] = value instead See the caveats in the documentation:
http://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy
```

	Balance	Age
0	8.3	-0.108733
1	9.4	0.075838
2	10.2	0.255347
3	0.0	0.427444

Видим, что хотя преобразование выполнено, оно сопровождается выдачей предупреждения `SettingWithCopy`.

Давайте включим копирование и проделаем все то же самое.

```
# создаем игрушечный обучающий датафрейм pandas
train = pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]})
# создаем экземпляр класса, включив копирование
corrlog = CorrLogTransformer(copy=True)
```

```
# обучаем модель
corrlog.fit(train[['Age']])
# применяем модель
train[['Age']] = corrlog.transform(train[['Age']])
train
```

	Balance	Age
0	8.3	-0.108733
1	9.4	0.075838
2	10.2	0.255347
3	0.0	0.427444

Видим, что предупреждение `SettingWithCopy` не выводится. Теперь проверим работу класса с массивами NumPy.

```
# создаем игрушечный обучающий массив NumPy
np_train = np.array(pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]}))
np_train

array([[ 8.3, 23. ],
       [ 9.4, 29. ],
       [10.2, 36. ],
       [ 0. , 44. ]])
```

```
# создаем игрушечный тестовый массив NumPy
np_test = np.array(pd.DataFrame({
    'Balance': [10.4, 3.1, 22.5, -1],
    'Age': [13, 19, 66, 33]}))
np_test

array([[10.4, 13. ],
       [ 3.1, 19. ],
       [22.5, 66. ],
       [-1. , 33. ]])
```

```
# обучаем модель
corrlog.fit(np_train)
# выполняем преобразование игрушечного
# обучающего массива NumPy
np_train = corrlog.transform(np_train)
np_train

array([[ 0.32927796, -0.1087332 ],
       [ 0.43675074,  0.07583808],
       [ 0.50824164,  0.25534669],
       [-1.60872132,  0.42744401]])
```

```
# выполняем преобразование игрушечного
# тестового массива NumPy
np_test = corrlog.transform(np_test)
np_test
```

```
array([[ 0.52534339, -0.520978  ],
       [-0.43936666, -0.25391521],
       [ 1.2313369 ,  0.78845736],
       [-1.60872132,  0.18232156]])
```

Опять видим, что значения, полученные с помощью класса `CorrLogTransformer`, совпадают с ожидаемыми.

Кроме того, мы можем сделать для переменных автоматический подбор преобразований, максимизирующих нормальность с помощью класса `PowerTransformer`. Однако если мы захотим воспользоваться преобразованием Бокса–Кокса, то в случае отрицательных и нулевых значений переменных мы получим ошибку `The Box-Cox transformation can only be applied to strictly positive data`. Поэтому перед его применением нужно преобразовать отрицательные и нулевые значения в небольшие положительные. Давайте напишем класс `Replacer`, который будет делать это.

```
# создаем собственный класс, заменяющий отрицательные
# и нулевые значения на небольшие положительные
class Replacer(BaseEstimator, TransformerMixin):
    """
    Заменяет отрицательные и нулевые значения на
    небольшое положительное значение.

    Параметры
    -----
    repl: float, по умолчанию 0.1
        Значение для замены.
    """

    def __init__(self, repl_value=0.1):
        self.repl_value = repl_value

    # fit здесь бездельничает
    def fit(self, X, y=None):
        return self

    # transform выполняет всю работу: применяет преобразование
    # с помощью заданного значения параметра repl_value
    def transform(self, X):
        if isinstance(X, pd.DataFrame):
            X[X <= 0] = self.repl_value
        else:
            X = np.where(X <= 0, self.repl_value, X)
        return X
```

В методе `__init__` мы задали параметр `repl_value` – число, на которое нужно заменить отрицательное или нулевое значение, метод `.fit()` в данном случае «бездельничает», метод `.transform()` применяет преобразование с помощью заданного значения параметра `repl_value`. Это нормальная практика, когда мы не вычисляем статистики, а применяем заранее известные константные преобразования.

Вновь создаем игрушечные обучающий и тестовый датафреймы.

*# создаем игрушечный обучающий датафрейм pandas*

```
train = pd.DataFrame(
    {'Balance': [0, 9.4, 10.2, 0],
     'Income': [23, 29, -1, 44]})
train
```

	Balance	Income
0	0.0	23
1	9.4	29
2	10.2	-1
3	0.0	44

*# создаем игрушечный тестовый датафрейм pandas*

```
test = pd.DataFrame(
    {'Balance': [-2, 3.1, 22.5, -1],
     'Income': [13, -2, 0, 33]})
test
```

	Balance	Income
0	-2.0	13
1	3.1	-2
2	22.5	0
3	-1.0	33

Проверяем работу класса `Replacer` на этих датафреймах.

*# создаем экземпляр класса Replacer*

```
repl = Replacer(repl_value=0.1)
```

*# обучаем модель*

```
repl.fit(train)
```

*# выполняем замены в игрушечном*

*# обучающем датафрейме pandas*

```
train = repl.transform(train)
```

```
train
```

	Balance	Income
0	0.1	23.0
1	9.4	29.0
2	10.2	0.1
3	0.1	44.0

*# выполняем замены в игрушечном*

*# тестовом датафрейме pandas*

```
test = repl.transform(test)
```

```
test
```

	Balance	Income
0	0.1	13.0
1	3.1	0.1
2	22.5	0.1
3	0.1	33.0

Создаем игрушечные обучающий и тестовый массивы NumPy.

```
# создаем игрушечный обучающий массив NumPy
```

```
np_train = np.array(pd.DataFrame(
    {'Balance': [0, 9.4, 10.2, 0],
     'Income': [23, 29, -1, 44]}))
np_train
```

```
array([[ 0., 23. ],
       [ 9.4, 29. ],
       [10.2, -1. ],
       [ 0., 44. ]])
```

```
# создаем игрушечный тестовый массив NumPy
```

```
np_test = np.array(pd.DataFrame(
    {'Balance': [-2, 3.1, 22.5, -1],
     'Income': [13, -2, 0, 33]}))
np_test
```

```
array([[ -2., 13. ],
       [ 3.1, -2. ],
       [22.5,  0. ],
       [ -1., 33. ]])
```

Проверяем работу класса `Replасег` на этих массивах NumPy.

```
# обучаем модель
```

```
repl.fit(np_train)
```

```
# выполняем замены в игрушечном
```

```
# обучающем массиве NumPy
```

```
np_train = repl.transform(np_train)
np_train
```

```
array([[ 0.1, 23. ],
       [ 9.4, 29. ],
       [10.2,  0.1],
       [ 0.1, 44. ]])
```

```
# выполняем замены в игрушечном
```

```
# тестовом массиве NumPy
```

```
np_test = repl.transform(np_test)
np_test
```

```
array([[ 0.1, 13. ],
       [ 3.1,  0.1],
       [22.5,  0.1],
       [ 0.1, 33. ]])
```

Теперь напишем класс `CustomDiscretizer`, выполняющий биннинг количественных переменных. В методе `__init__` мы задали параметр `bins` – список заранее найденных бинов, метод `.fit()` опять «бездельничает», метод `.transform()` выполняет биннинг с помощью заданного значения параметра `bins` – списка бинов. Для выполнения биннинга в методе `.transform()` мы используем функцию NumPy `np.digitize()`, которая возвращает целочисленные индексы бинов, к которым принадлежат исходные значения переменных. Класс рекомендуется применять к отдельной переменной либо к переменным одного и того же масштаба.

*# создаем собственный класс, выполняющий биннинг*

```
class CustomDiscretizer(BaseEstimator, TransformerMixin):
    """
    Выполняет биннинг количественных признаков.

    Параметры
    -----
    bins: list
        Список бинов.
    """
    def __init__(self, bins=[-np.inf, 2, 4, np.inf]):
        self.bins = bins

    # fit опять бездельничает
    def fit(self, X, y=None):
        return self

    # transform выполняет всю работу: применяем
    # преобразование с помощью заданного
    # значения параметра bins
    def transform(self, X):
        if isinstance(X, pd.DataFrame):
            cols = X.columns.tolist()
            X = pd.DataFrame(data=np.digitize(X, self.bins),
                             columns=cols)
        else:
            X = np.digitize(X, self.bins)
        return X
```

Создаем игрушечный обучающий датафрейм.

*# создаем игрушечный обучающий датафрейм pandas*

```
train = pd.DataFrame(
    {'Balance': [0, 1.4, 2.2, 5.5, 4.3],
     'Income': [3, 5, 1, 4, 6]})
train
```

	Balance	Income
0	0.0	3
1	1.4	5
2	2.2	1
3	5.5	4
4	4.3	6

Проверяем работу класса `CustomDiscretizer` на этом обучающем датафрейме.

```
# создаем экземпляр класса CustomDiscretizer
disc = CustomDiscretizer()
# обучаем модель
disc.fit(train)
# выполняем биннинг в игрушечном
# обучающем датафрейме pandas
train = disc.transform(train)
train
```

	Balance	Income
0	1	2
1	1	3
2	2	1
3	3	3
4	3	3

Создаем игрушечный обучающий массив `NumPy`.

```
# создаем игрушечный обучающий массив NumPy
np_train = np.array(pd.DataFrame(
    {'Balance': [0, 1.4, 2.2, 5.5, 4.3],
     'Income': [3, 5, 1, 4, 6]}))
np_train

array([[0. , 3. ],
       [1.4, 5. ],
       [2.2, 1. ],
       [5.5, 4. ],
       [4.3, 6. ]])
```

Проверяем работу класса `CustomDiscretizer` на этом обучающем массиве `NumPy`.

```
# обучаем модель
disc.fit(np_train)
# выполняем биннинг в игрушечном
# обучающем массиве NumPy
np_train = disc.transform(np_train)
np_train

array([[1, 2],
       [1, 3],
       [2, 1],
       [3, 3],
       [3, 3]])
```

Наконец, напишем класс `CustomScaler`, выполняющий стандартизацию. Часто необходимость написания собственного класса стандартизации вызывает недоумение у начинающих специалистов. Однако отметим, что существует довольно много способов стандартизации, которые реализованы в отдельных классах би-

библиотеки `scikit-learn` (классы `StandardScaler`, `MinMaxScaler`, `RobustScaler`), что не совсем удобно при тестировании разных стратегий импутации на месте. Здесь обратите внимание, что для всех переменных, подвергаемых стандартизации, применяется один и тот же способ. Нельзя одну часть переменных стандартизировать с помощью класса `StandardScaler`, а другую часть переменных – с помощью класса `MinMaxScaler`. Кроме того, есть способы стандартизации, не реализованные в виде классов. Мы помним, что в стандартизации нуждаются лишь количественные переменные. Если речь идет о линейных моделях, категориальные переменные будут записаны в виде дамми-переменных со значениями 0 или 1 и дамми-переменные стандартизировать не нужно. В случае присутствия дамми-переменных рекомендуется при выполнении стандартизации количественных переменных делить не на одно, а на два стандартных отклонения (поправка, предложенная Эндрю Гельманом <http://www.stat.columbia.edu/~gelman/research/published/standardizing7.pdf>), чтобы и дамми-переменные, и количественные переменные имели один и тот же масштаб (и мы могли сравнивать коэффициенты при них), стандартные отклонения дамми-переменных и количественных переменных будут примерно равны 0,5. В противном случае стандартные отклонения дамми-переменных будут равны 0,5, а стандартные отклонения количественных переменных – 1, что не позволит корректно сравнивать коэффициенты при дамми-переменных с коэффициентами при количественных переменных.

Наш класс `CustomScaler` будет выполнять стандартизацию либо как в классе `StandardScaler` (когда из исходного значения переменной вычитаем среднее и полученный результат делим на стандартное отклонение), либо как в классе `MinMaxScaler` (когда из исходного значения переменной вычитаем минимальное значение и полученный результат делим на разницу между максимальным и минимальным значениями). Для этого у него будет параметр `strategy`, который может принимать либо значение `'standard_scaler'`, либо значение `'minmax_scaler'`. С помощью параметра `corr` можно задать поправку Гельмана (ее можно задать, только когда `strategy='standard_scaler'`).

При написании класса нужно учесть, что в `NumPy` и `pandas` стандартное отклонение вычисляется по разным формулам (в `pandas` используется

$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$ , а в `NumPy` используется  $s = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$ ). Для эквивалентности результатов установите значение параметра `ddof` функции `np.std()` равным 1.

Класс `CustomScaler` похож по своей структуре на ранее написанный нами класс `MeanImputer`. Мы используем частный метод `__is_numpy`, который с помощью функции `isinstance()` проверяет, является ли наш объект массивом `NumPy`, и в зависимости от результатов проверки вычисляем/применяем статистики соответствующим образом. Если получаем на вход одномерный массив, превращаем его в двумерный (обратите внимание: в ранее созданных классах мы не стали добавлять проверку на одномерный массив). Для хранения статистик мы используем словарь, где ключами будут имена переменных (для датафреймов `pandas`) или целочисленные значения (для массивов `NumPy`), а значениями будут двухэлементные кортежи, для стратегии `'standard_scaler'` первым элементом будет среднее значение, вторым элементом – стандартное

отклонение, для стратегии 'minmax\_scaler' первым элементом будет минимальное значение, вторым элементом – разница между максимальным и минимальным значениями.

*# создаем собственный класс, выполняющий стандартизацию*

```
class CustomScaler(BaseEstimator, TransformerMixin):
    """
    Выполняет стандартизацию признаков.

    Параметры
    -----
    strategy: str, по умолчанию 'standard_scaler'
        Стратегия стандартизации.
    corr: bool, по умолчанию False
        Корректировка по Гельману.
    copy: bool, по умолчанию True
        Выполняет копирование датафрейма.
    """
    def __init__(self, strategy='standard_scaler',
                 copy=True, corr=False):
        self.strategy = strategy
        self.corr = corr
        self.copy = copy

        if corr and strategy == 'minmax_scaler':
            raise ValueError('Corr=True только для ' +
                              'стратегии standard_scaler')

    # частный метод, который с помощью функции isinstance()
    # проверяет, является ли наш объект массивом NumPy
    def __is_numpy(self, X):
        return isinstance(X, np.ndarray)

    def fit(self, X, y=None):
        self.dict = {}

        # если 1D-массив, то переводим в 2D
        if len(X.shape) == 1:
            X = X.reshape(-1, 1)

        # записываем результат __is_numpy()
        is_np = self.__is_numpy(X)

        # записываем количество столбцов
        ncols = X.shape[1]

        if self.strategy == 'standard_scaler':
            # если объект - массив NumPy,
            # выполняем следующие действия:
            if is_np:
                # по каждому столбцу массива NumPy
                for col in range(ncols):
                    # вычисляем среднее
                    mean = np.mean(X[:, col])
                    # вычисляем стандартное отклонение
                    std = np.std(X[:, col], ddof=1)
```

```

        if self.corr:
            std = np.std(X[:, col], ddof=1) * 2
            self.dict[col] = (mean, std)

    # если объект - датафрейм pandas,
    # выполняем следующие действия:
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # вычисляем среднее
            mean = X[col].mean()
            # вычисляем стандартное отклонение
            std = X[col].std()
            if self.corr:
                std = X[col].std() * 2
            self.dict[col] = (mean, std)

if self.strategy == 'minmax_scaler':
    # если объект - массив NumPy,
    # выполняем следующие действия:
    if is_np:
        # по каждому столбцу массива NumPy
        for col in range(ncols):
            # вычисляем минимальное значение
            min_value = np.min(X[:, col])
            # вычисляем разницу между максимальным
            # и минимальным значениями
            rng = np.ptp(X[:, col])
            self.dict[col] = (min_value, rng)

    # если объект - датафрейм pandas,
    # выполняем следующие действия:
    else:
        # по каждому столбцу датафрейма pandas
        for col in X.columns:
            # вычисляем среднее
            min_value = X[col].min()
            # вычисляем разницу между максимальным
            # и минимальным значениями
            rng = X[col].max() - X[col].min()
            self.dict[col] = (min_value, rng)

return self

def transform(self, X):

    if self.copy:
        X = X.copy()

    # если 1D-массив, то переводим в 2D
    if len(X.shape) == 1:
        X = X.reshape(-1, 1)

    # записываем результат __is_numpy()
    is_np = self.__is_numpy(X)

```

```

# записываем количество столбцов
ncols = X.shape[1]

# если объект - массив NumPy,
# выполняем следующие действия:
if is_np:
    # по каждому столбцу массива NumPy
    for col in range(ncols):
        # выполняем стандартизацию
        X[:, col] = ((X[:, col] - self.dict[col][0]) /
                    self.dict[col][1])

# если объект - датафрейм pandas,
# выполняем следующие действия:
else:
    # по каждому столбцу датафрейма pandas
    for col in X.columns:
        # выполняем стандартизацию
        X[col] = ((X[col] - self.dict[col][0]) /
                 self.dict[col][1])

return X

```

Теперь посмотрим, какими должны быть значения в обучающем и тестовом датафреймах pandas для стратегии 'standard\_scaler', когда поправка Гельмана не используется.

```

# создаем игрушечный обучающий датафрейм pandas
train = pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]})
# создаем игрушечный тестовый датафрейм pandas
test = pd.DataFrame(
    {'Balance': [10.4, 3.1, 22.5, -1],
     'Age': [13, 19, 66, 33]})

# смотрим, как должны выглядеть результаты
balance_mean = toy_train['Balance'].mean()
balance_std = toy_train['Balance'].std()
age_mean = toy_train['Age'].mean()
age_std = toy_train['Age'].std()
print((toy_train['Balance'] - balance_mean) / balance_std)
print((toy_train['Age'] - age_mean) / age_std)
print("")
print((toy_test['Balance'] - balance_mean) / balance_std)
print((toy_test['Age'] - age_mean) / age_std)

0    0.281031
1    0.514340
2    0.684019
3   -1.479390
Name: Balance, dtype: float64
0   -1.104315
1   -0.441726
2    0.331295
3    1.214747

```

```
Name: Age, dtype: float64
```

```
0    0.726439
1   -0.821883
2    3.292835
3   -1.691489
```

```
Name: Balance, dtype: float64
```

```
0   -2.208631
1   -1.546041
2    3.644240
3    0.000000
```

```
Name: Age, dtype: float64
```

Проверяем работу класса `CustomScaler` на датафреймах, заодно выведем информацию о стандартных отклонениях стандартизированных переменных.

```
# применяем класс к датафреймам pandas
scaler = CustomScaler(strategy='standard_scaler',
                      corr=False)
scaler.fit(toy_train)
toy_train = scaler.transform(toy_train)
toy_test = scaler.transform(toy_test)
```

```
# смотрим результат в игрушечном
# обучающем датафрейме
toy_train
```

	Balance	Age
0	0.281031	-1.104315
1	0.514340	-0.441726
2	0.684019	0.331295
3	-1.479390	1.214747

```
# смотрим результат в игрушечном
# тестовом датафрейме
toy_test
```

	Balance	Age
0	0.726439	-2.208631
1	-0.821883	-1.546041
2	3.292835	3.644240
3	-1.691489	0.000000

```
# взглянем на стандартные отклонения
for col in toy_train.columns:
    print(col, toy_train[col].std())
```

```
Balance 1.0
Age 1.0
```

Видим, что значения, полученные с помощью класса `CustomScaler`, совпадают с ожидаемыми.

Проверяем работу класса `CustomScaler` на массивах `NumPy`.

```
# создаем игрушечный обучающий массив NumPy
np_toy_train = np.array(pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]}))

# создаем игрушечный тестовый массив NumPy
np_toy_test = np.array(pd.DataFrame(
    {'Balance': [10.4, 3.1, 22.5, -1],
     'Age': [13, 19, 66, 33]}))

# применяем класс к массивам NumPy
scaler.fit(np_toy_train)
np_toy_train = scaler.transform(np_toy_train)
np_toy_test = scaler.transform(np_toy_test)

# смотрим результат в игрушечном
# обучающем массиве NumPy
np_toy_train

array([[ 0.28103104, -1.10431526],
       [ 0.51433982, -0.4417261 ],
       [ 0.68401894,  0.33129458],
       [-1.4793898 ,  1.21474679]])

# смотрим результат в игрушечном
# тестовом массиве NumPy
np_toy_test

array([[ 0.72643872, -2.20863052],
       [-0.82188322, -1.54604137],
       [ 3.29283537,  3.64424036],
       [-1.6914887 ,  0.          ]])
```

Опять видим, что значения, полученные с помощью класса `CustomScaler`, совпадают с ожидаемыми.

Теперь посмотрим, какими должны быть значения в обучающем и тестовом датафреймах `pandas` для стратегии `'standard_scaler'`, когда поправка Гельмана используется.

```
# создаем игрушечный обучающий датафрейм pandas
train = pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]})
# создаем игрушечный тестовый датафрейм pandas
test = pd.DataFrame(
    {'Balance': [10.4, 3.1, 22.5, -1],
     'Age': [13, 19, 66, 33]})
```

```
# смотрим, как должны выглядеть результаты
balance_mean = toy_train['Balance'].mean()
balance_2std = toy_train['Balance'].std() * 2
age_mean = toy_train['Age'].mean()
age_2std = toy_train['Age'].std() * 2
print((toy_train['Balance'] - balance_mean) / balance_2std)
print((toy_train['Age'] - age_mean) / age_2std)
print("")
print((toy_test['Balance'] - balance_mean) / balance_2std)
print((toy_test['Age'] - age_mean) / age_2std)
```

```
0    0.140516
1    0.257170
2    0.342009
3   -0.739695
Name: Balance, dtype: float64
0   -0.552158
1   -0.220863
2    0.165647
3    0.607373
Name: Age, dtype: float64
```

```
0    0.363219
1   -0.410942
2    1.646418
3   -0.845744
Name: Balance, dtype: float64
0   -1.104315
1   -0.773021
2    1.822120
3    0.000000
Name: Age, dtype: float64
```

Проверяем работу класса `CustomScaler` на датафреймах, заодно выведем информацию о стандартных отклонениях стандартизированных переменных.

```
# применяем класс к датафреймам pandas
scaler = CustomScaler(strategy='standard_scaler',
                      corr=False)
scaler.fit(toy_train)
toy_train = scaler.transform(toy_train)
toy_test = scaler.transform(toy_test)
```

```
# смотрим результат в игрушечном
# обучающем датафрейме
toy_train
```

	<b>Balance</b>	<b>Age</b>
<b>0</b>	0.140516	-0.552158
<b>1</b>	0.257170	-0.220863
<b>2</b>	0.342009	0.165647
<b>3</b>	-0.739695	0.607373

```
# смотрим результат в игрушечном
# тестовом датафрейме
toy_test
```

	Balance	Age
0	0.363219	-1.104315
1	-0.410942	-0.773021
2	1.646418	1.822120
3	-0.845744	0.000000

```
# взглянем на стандартные отклонения
for col in toy_train.columns:
    print(col, toy_train[col].std())
```

```
Balance 0.5
Age 0.5
```

Видим, что значения, полученные с помощью класса `CustomScaler`, совпадают с ожидаемыми. Кроме того, как и ожидалось, у нас стандартные отклонения стали равны 0,5.

Проверяем работу класса `CustomScaler` на массивах `NumPy`.

```
# создаем игрушечный обучающий массив NumPy
np_toy_train = np.array(pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]}))

# создаем игрушечный тестовый массив NumPy
np_toy_test = np.array(pd.DataFrame(
    {'Balance': [10.4, 3.1, 22.5, -1],
     'Age': [13, 19, 66, 33]}))

# применяем класс к массивам NumPy
scaler.fit(np_toy_train)
np_toy_train = scaler.transform(np_toy_train)
np_toy_test = scaler.transform(np_toy_test)

# смотрим результат в игрушечном
# обучающем массиве NumPy
np_toy_train

array([[ 0.14051552, -0.55215763],
       [ 0.25716991, -0.22086305],
       [ 0.34200947,  0.16564729],
       [-0.7396949 ,  0.60737339]])

# смотрим результат в игрушечном
# тестовом массиве NumPy
np_toy_test

array([[ 0.36321936, -1.10431526],
       [-0.41094161, -0.77302068],
       [ 1.64641768,  1.82212018],
       [-0.84574435,  0.          ]])
```



```
toy_train = scaler.transform(toy_train)
toy_test = scaler.transform(toy_test)
```

```
# Смотрим результат в игрушечном
# обучающем датафрейме
toy_train
```

	Balance	Age
0	0.813725	0.000000
1	0.921569	0.285714
2	1.000000	0.619048
3	0.000000	1.000000

```
# Смотрим результат в игрушечном
# тестовом датафрейме
toy_test
```

	Balance	Age
0	1.019608	-0.476190
1	0.303922	-0.190476
2	2.205882	2.047619
3	-0.098039	0.476190

Видим, что значения, полученные с помощью класса `CustomScaler`, совпадают с ожидаемыми.

Проверяем работу класса `CustomScaler` на массивах `NumPy`.

```
# создаем игрушечный обучающий массив NumPy
np_toy_train = np.array(pd.DataFrame(
    {'Balance': [8.3, 9.4, 10.2, 0],
     'Age': [23, 29, 36, 44]}))

# создаем игрушечный тестовый массив NumPy
np_toy_test = np.array(pd.DataFrame(
    {'Balance': [10.4, 3.1, 22.5, -1],
     'Age': [13, 19, 66, 33]}))

# применяем класс к массивам NumPy
scaler.fit(np_toy_train)
np_toy_train = scaler.transform(np_toy_train)
np_toy_test = scaler.transform(np_toy_test)

# Смотрим результат в игрушечном
# обучающем массиве NumPy
np_toy_train
```

```
array([[0.81372549, 0.          ],
       [0.92156863, 0.28571429],
       [1.          , 0.61904762],
       [0.          , 1.          ]])
```

```
# смотрим результат в угрушечном
# тестовом массиве NumPy
np_toy_test

array([[ 1.01960784, -0.47619048],
       [ 0.30392157, -0.19047619],
       [ 2.20588235,  2.04761905],
       [-0.09803922,  0.47619048]])
```

Значения, полученные с помощью класса `CustomScaler`, совпадают с ожидаемыми.

А теперь все как обычно: создаем списки переменных, создаем трансформеры, список трансформеров передаем в `ColumnTransformer`, задаем итоговый конвейер. При этом обратите внимание: для количественных переменных *Customer LifeTime Value*, *Number of Policies*, *Monthly Premium Auto* у нас будут отдельные трансформеры. Для переменной *Customer LifeTime Value* мы применим собственный класс `CorrLogTransformer`, для переменных *Number of Policies* и *Monthly Premium Auto* мы применим собственный класс `CustomDiscretizer`. Также применим собственный класс `Replacer` для количественных переменных *Number of Open Complaints*, *Income*, *Months Since Last Claim* и *Months Since Policy Inception*.

Итак, готовим списки признаков. Сначала создаем список количественных признаков и список категориальных признаков. Список категориальных признаков откладываем в сторону, а со списком количественных признаков немного поработаем. Создадим список с одним элементом – признаком *Customer LifeTime Value*, он будет предназначаться для трансформера, обрабатывающего признак *Customer LifeTime Value*. Создадим список с одним элементом – признаком *Number of Policies*, он будет предназначаться для трансформера, обрабатывающего признак *Number of Policies*. Создадим список с одним элементом – признаком *Monthly Premium Auto*, он будет предназначаться для трансформера, обрабатывающего признак *Monthly Premium Auto*. Затем из списка количественных признаков удаляем признаки *Customer LifeTime Value*, *Number of Policies*, *Monthly Premium Auto*.

```
# создаем список категориальных признаков
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
# создаем список количественных признаков
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()
# создаем одноэлементный список для
# признака Customer LifeTime Value
clv_corrlog = ['Customer Lifetime Value']
# создаем одноэлементный список для
# признака Number of Policies
numberofpol_bin = ['Number of Policies']
# создаем одноэлементный список для
# признака Monthly Premium Auto
monthly_bin = ['Monthly Premium Auto']

# удаляем из списка количественных признаков признаки Customer
# LifeTime Value, Number of Policies, Monthly Premium Auto
num_columns = list(set(num_columns).difference(
    set(clv_corrlog + numberofpol_bin + monthly_bin)))
['Number of Open Complaints',
 'Months Since Last Claim',
```

```
'Months Since Policy Inception',
'Income']
```

Теперь создаем трансформеры, сводим в список трансформеров, передаем в ColumnTransformer и смотрим получившийся экземпляр класса ColumnTransformer.

```
# создаем конвейер для количественных признаков
# Number of Open Complaints, Income,
# Months Since Last Claim u
# Months Since Policy Inception
num_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='mean')),
    ('repl', Replacer()),
    ('box_cox', PowerTransformer(method='box-cox',
                                standardize=True))
])

# создаем конвейер для количественной
# переменной Customer Lifetime Value
clv_corrlog_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='mean')),
    ('corrlog', CorrLogTransformer(k=0.001)),
    ('scaler', CustomScaler())
])

# создаем конвейер для количественной переменной
# Number of Policies
numberofpol_bin_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('binn', CustomDiscretizer()),
    ('ohe', OneHotEncoder(sparse=False,
                          handle_unknown='ignore'))
])

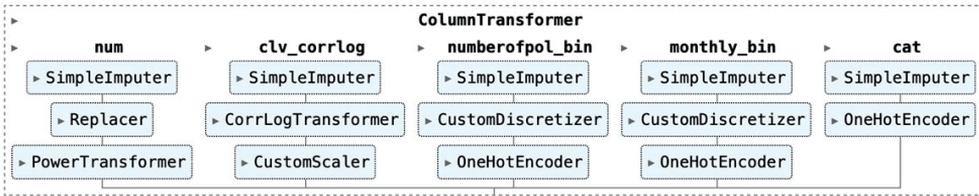
# создаем конвейер для количественной
# переменной Monthly Premium Auto
monthly_bin_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('binn', CustomDiscretizer()),
    ('ohe', OneHotEncoder(sparse=False,
                          handle_unknown='ignore'))
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('ohe', OneHotEncoder(sparse=False,
                          handle_unknown='ignore'))
])

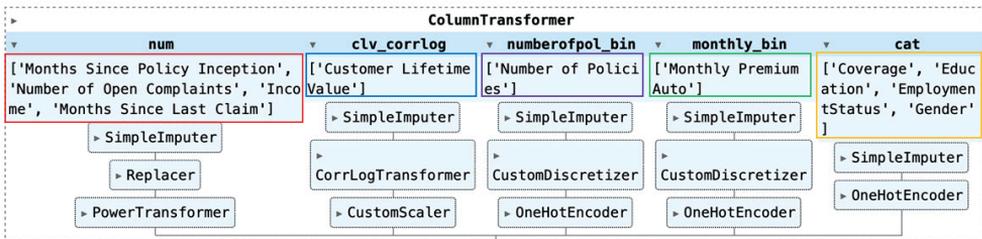
# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название конвейера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
                ('clv_corrlog', clv_corrlog_pipe, clv_corrlog),
                ('numberofpol_bin', numberofpol_bin_pipe, numberofpol_bin),
                ('monthly_bin', monthly_bin_pipe, monthly_bin),
                ('cat', cat_pipe, cat_columns)]
```

*# передаем список трансформеров в ColumnTransformer*

```
transformer = ColumnTransformer(transformers=transformers)
transformer
```



Щелчком по названию каждого списка, чтобы лучше понять, с какими признаками будет работать каждый трансформер.



Теперь создаем итоговый конвейер и задаем сетку значений гиперпараметров. Мы будем перебирать разные значения корректора асимметрии, разные числа, используемые для замены отрицательных и нулевых значений, разные списки бинов для биннинга и, наконец, разные значения силы регуляризации.

*# задаем итоговый конвейер*

```
ml_pipe = Pipeline([
    ('tr', transformer),
    ('lr', LogisticRegression(solver='lbfgs',
                             max_iter=200))
])
```

*# задаем сетку гиперпараметров*

```
param_grid = {
    'tr_num_repl_repl_value': [0.005, 0.01, 0.2],
    'tr_clv_corrlog_corrlog_k': [0.1, 0.5, 0.6, 0.7, 0.8],
    'tr_numberofpol_bin_binn_bins': [[-np.inf, 2, 4, 6, np.inf],
                                     [-np.inf, 3, 6, np.inf]],
    'tr_monthly_bin_binn_bins': [[-np.inf, 100, 150, 200, 250, np.inf],
                                 [-np.inf, 100, 150, 200, np.inf]],
    'lr_C': [0.1, 0.2, 0.3]
}
```

Выполняем поиск по сетке и смотрим результаты.

*# создаем экземпляр класса GridSearchCV, передав конвейер,*

*# сетку гиперпараметров, оптимизируем метрику качества*

*# и указав количество блоков перекрестной проверки*

```
gs = GridSearchCV(ml_pipe,
                  param_grid,
```

```

        scoring='roc_auc',
        cv=5)
# выполняем поиск по сетке
gs.fit(X_train, y_train)

# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs.best_params_))
# смотрим наилучшее значение AUC
print("Наилучшее значение AUC: {:.3f}".format(
    gs.best_score_))
# смотрим значение AUC на тестовой выборке
print("AUC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, gs.predict_proba(X_test)[: , 1])))

```

```

Наилучшие значения гиперпараметров:
{'lr_C': 0.2, 'tr_clv_corrlog_corrlog_k': 0.6,
 'tr_monthly_bin_binn_bins': [-inf, 100, 150, 200, inf],
 'tr_num_repl_repl_value': 0.005,
 'tr_numberofpol_bin_binn_bins': [-inf, 3, 6, inf]}
Наилучшее значение AUC: 0.668
AUC на тестовой выборке: 0.669

```

## 7.16. МОДИФИКАЦИЯ КЛАССОВ БИБЛИОТЕКИ SCIKIT-LEARN ДЛЯ РАБОТЫ С ДАТАФРЕЙМАМИ

Классы библиотеки scikit-learn предназначены для работы с массивами NumPy. Методы `.transform()`, `.predict()` и `.predict_proba()` возвращают нам массив NumPy вместо объекта DataFrame библиотеки pandas, что не всегда бывает удобно для работы.

Давайте импортируем необходимые библиотеки, классы и функции и загрузим расширенные данные StateFarm.

```

# импортируем необходимые библиотеки pandas и numpy,
# функцию train_test_split() и классы StandardScaler,
# OneHotEncoder, TransformerMixin,
# LogisticRegression, Pipeline
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import (StandardScaler,
                                   OneHotEncoder)
from sklearn.base import TransformerMixin
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/StateFarm_missing.csv', sep=';')

```

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```

# разбиваем данные на обучающие и тестовые: получаем обучающий
# массив признаков, тестовый массив признаков, обучающий массив
# меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('Response', axis=1),
    data['Response'],
    test_size=0.3,
    stratify=data['Response'],
    random_state=42)

```

Теперь создаем собственные классы импутации, стандартизации и дамми-кодирования, которые будут возвращать датафрейм pandas.

```

# создаем класс, выполняющий импутацию
class DFImputer(TransformerMixin):
    def __init__(self):
        self.fill = None

    def fit(self, X, y=None):
        # вычисляем моды (берем первую) для
        # категориальных переменных
        # и медианы - для количественных
        self.fill = pd.Series([X[c].mode()[0]
                               if X[c].dtype == np.dtype('O') else X[c].median()
                               for c in X], index=X.columns)

        return self

    def transform(self, X, y=None):
        # заменяем модами пропуски в категориальных переменных
        # и медианами - пропуски в количественных переменных
        Xfill = X.fillna(self.fill)
        return Xfill

# создаем класс, выполняющий стандартизацию
class DFStandardScaler(TransformerMixin):
    def __init__(self):
        pass

    def fit(self, X, y=None):
        self.ss = StandardScaler()
        # выделяем список количественных переменных
        num_cols = X.select_dtypes(
            exclude='object').columns.tolist()
        # вычисляем средние (mean_) и стандартные отклонения (scale_)
        # для количественных переменных
        self.ss.fit(X[num_cols])
        return self

    def transform(self, X):
        # выделяем список количественных переменных
        num_cols = X.select_dtypes(
            exclude='object').columns.tolist()
        # выполняем стандартизацию количественных переменных
        X[num_cols] = self.ss.transform(X[num_cols])
        # преобразовываем массив NumPy в объект DataFrame
        X_scaled = pd.DataFrame(X, index=X.index, columns=X.columns)
        return X_scaled

# создаем класс, выполняющий дамми-кодирование
class DFOneHotEncoder(TransformerMixin):
    def __init__(self):
        pass

```

```

def fit(self, X, y=None):
    # выделяем список категориальных переменных
    cat_cols = X.select_dtypes(
        include='object').columns.tolist()
    # определяем дамми-переменные для
    # категориальных переменных
    self.ohc = OneHotEncoder(sparse=False,
                             handle_unknown='ignore')
    self.ohc.fit(X[cat_cols])
    return self
def transform(self, X):
    # выделяем списки количественных
    # и категориальных переменных
    num_cols = X.select_dtypes(
        exclude='object').columns.tolist()
    cat_cols = X.select_dtypes(
        include='object').columns.tolist()
    # выполняем дамми-кодирование
    # категориальных переменных
    Xt = self.ohc.transform(X[cat_cols])
    # извлекаем имена дамми-переменных
    cols = self.ohc.get_feature_names()
    # превращаем массив NumPy с дамми-переменными
    # в объект DataFrame
    X_dum = pd.DataFrame(Xt, index=X[cat_cols].index,
                         columns=cols)
    # конкатерируем датафрейм, содержащий
    # количественные переменные, с датафреймом,
    # содержащим дамми-переменные
    X_res = pd.concat([X[num_cols], X_dum], axis=1)
    return X_res

```

Вновь проверим созданные классы на упрощенных наборах данных.

```

# загружаем игрушечные наборы для тестирования классов
toy_train = pd.read_csv('Data/toy_train.csv', sep=';')
toy_test = pd.read_csv('Data/toy_test.csv', sep=';')

```

```

# смотрим игрушечный обучающий набор
toy_train

```

	age	income	region
0	NaN	NaN	MSK
1	23.0	4560.55	MSK
2	24.0	NaN	NaN
3	30.0	NaN	EKAT
4	NaN	7888.10	NaN
5	55.0	9000.50	SPB
6	37.0	NaN	SPB

```

# смотрим игрушечный тестовый набор
toy_test

```

	age	income	region
0	89.0	903.33	MSK
1	23.0	4560.55	NSK
2	24.0	NaN	MSK
3	55.0	6700.00	MSK
4	NaN	8999.00	EKAT
5	NaN	5430.00	SPB
6	37.0	NaN	NaN

У нас две количественные переменные *age* и *income* и категориальная переменная *region*, есть пропуски, переменная *region* имеет две моды, переменные имеют разный масштаб, в тестовом наборе у переменной *region* есть категория NSK, которая отсутствует в обучающем наборе.

Давайте вычислим медианы для количественных переменных и моду для категориальной переменной.

```
# вычислим медианы и моды
print(toy_train['age'].median())
print(toy_train['income'].median())
print(toy_train['region'].mode())
```

```
30.0
7888.1
0 MSK
1 SPB
dtype: object
```

Применим класс `DFImputer` и взглянем на результаты.

```
# тестируем класс, выполняющий импутацию,
# на игрушечных данных
imp = DFImputer()
imp.fit(toy_train)
toy_train = imp.transform(toy_train)
toy_test = imp.transform(toy_test)
```

	age	income	region
0	30.0	7888.10	MSK
1	23.0	4560.55	MSK
2	24.0	7888.10	MSK
3	30.0	7888.10	EKAT
4	30.0	7888.10	MSK
5	55.0	9000.50	SPB
6	37.0	7888.10	SPB

```
# смотрим игрушечный тестовый набор
# после импутации
toy_test
```

	age	income	region
0	89.0	903.33	MSK
1	23.0	4560.55	NSK
2	24.0	7888.10	MSK
3	55.0	6700.00	MSK
4	30.0	8999.00	EKAT
5	30.0	5430.00	SPB
6	37.0	7888.10	MSK

Теперь применим класс `DFStandardScaler`.

```
# тестируем класс, выполняющий стандартизацию,
# на игрушечных данных
scaler = DFStandardScaler()
scaler.fit(toy_train)
toy_train = scaler.transform(toy_train)
toy_test = scaler.transform(toy_test)

# смотрим игрушечный обучающий набор
# после стандартизации
toy_train
```

	age	income	region
0	-0.270000	0.245729	MSK
1	-0.966315	-2.338170	MSK
2	-0.866841	0.245729	MSK
3	-0.270000	0.245729	EKAT
4	-0.270000	0.245729	MSK
5	2.216840	1.109526	SPB
6	0.426315	0.245729	SPB

```
# смотрим игрушечный тестовый набор
# после стандартизации
toy_test
```

	age	income	region
0	5.598941	-5.178063	MSK
1	-0.966315	-2.338170	NSK
2	-0.866841	0.245729	MSK
3	2.216840	-0.676851	MSK
4	-0.270000	1.108361	EKAT
5	-0.270000	-1.663027	SPB
6	0.426315	0.245729	MSK

Наконец, воспользуемся классом `DFOneHotEncoder`.

```
# тестируем класс, выполняющий дамми-кодирование,
# на игрушечных данных
ohe = DFOneHotEncoder()
ohe.fit(toy_train)
toy_train = ohe.transform(toy_train)
toy_test = ohe.transform(toy_test)

# смотрим игрушечный обучающий набор
# после дамми-кодирования
toy_train
```

	age	income	x0_EKAT	x0_MSK	x0_SPB
0	-0.270000	0.245729	0.0	1.0	0.0
1	-0.966315	-2.338170	0.0	1.0	0.0
2	-0.866841	0.245729	0.0	1.0	0.0
3	-0.270000	0.245729	1.0	0.0	0.0
4	-0.270000	0.245729	0.0	1.0	0.0
5	2.216840	1.109526	0.0	0.0	1.0
6	0.426315	0.245729	0.0	0.0	1.0

```
# смотрим игрушечный тестовый набор
# после дамми-кодирования
toy_test
```

	age	income	x0_EKAT	x0_MSK	x0_SPB
0	5.598941	-5.178063	0.0	1.0	0.0
1	-0.966315	-2.338170	0.0	0.0	0.0
2	-0.866841	0.245729	0.0	1.0	0.0
3	2.216840	-0.676851	0.0	1.0	0.0
4	-0.270000	1.108361	1.0	0.0	0.0
5	-0.270000	-1.663027	0.0	0.0	1.0
6	0.426315	0.245729	0.0	1.0	0.0

Убедившись в том, что все созданные нами классы отработали правильно, применяем их с помощью класса `Pipeline` к нашим исходным данным.

```
# создаем конвейер
ml_pipe = Pipeline([
    ('impute', DFImputer()),
    ('scaler', DFStandardScaler()),
    ('ohe', DFOneHotEncoder()),
    ('logreg', LogisticRegression(solver='lbfgs',
                                  max_iter=200))
])
```

```

# обучаем итоговый конвейер
ml_pipe.fit(X_train, y_train)
# оцениваем качество модели на обучающих данных
print('Правильность на обучающей выборке: {:.3f}'.format(
    ml_pipe.score(X_train, y_train)))
# оцениваем качество модели на тестовых данных
print('Правильность на тестовой выборке: {:.3f}'.format(
    ml_pipe.score(X_test, y_test)))

```

Правильность на обучающей выборке: 0.900

Правильность на тестовой выборке: 0.899

Обратите внимание, здесь мы не использовали класс `ColumnTransformer`, применяющий соответствующий трансформер для соответствующего списка признаков, наши классы уже «знают», с какими переменными им нужно работать. Подобная модификация классов может быть полезна для клиентов, желающих увидеть, что происходит с данными на каждом этапе сложного конвейера, понять, что может пойти не так в ходе преобразований.

## 7.17. Полный цикл построения конвейера моделей

### В SCIKIT-LEARN

До этого момента мы ограничивались лишь тем, что выполняли очень простую предварительную подготовку данных и выполняли поиск оптимальной комбинации гиперпараметров моделей, входящих в конвейер, с помощью перекрестной проверки, заново обучали конвейер с найденной оптимальной комбинацией гиперпараметров на всей обучающей выборке и проверяли качество прогнозов конвейера на тестовой выборке. В этом разделе мы решим две отдельные задачи, для этого выполним более сложную предварительную подготовку, часть операций предварительной подготовки выполним с помощью специально написанной функции, а другую часть операций выполним с помощью трансформеров конвейера. Здесь мы построим конвейер моделей с подбором гиперпараметров по полному циклу:

- с помощью перекрестной проверки найдем оптимальную комбинацию гиперпараметров моделей, входящих в конвейер, заново обучим конвейер на всей обучающей выборке и проверим качество прогнозов конвейера на тестовой выборке;
- заново обучим конвейер с найденной оптимальной комбинацией гиперпараметров на всей исторической выборке;
- применим конвейер с найденной оптимальной комбинацией гиперпараметров, обученный на всей исторической выборке, к новым данным.

#### 7.17.1. Первая задача

Первая задача заключается в прогнозировании оттока клиентов. Данные об оттоке клиентов записаны в файле `Verizon.csv`. По каждому клиенту (наблюдению) фиксируются следующие переменные:

- количественный признак *Длительность междугородних звонков* [`longdist`];
- количественный признак *Длительность международных звонков* [`internat`];

- количественный признак *Длительность местных звонков* [*local*];
- категориальный признак *Скидка на интернет* [*int\_disc*];
- категориальный признак *Тип счета* [*billtype*];
- категориальный признак *Способ оплаты* [*pay*];
- количественный признак *Возраст клиента* [*age*];
- категориальный признак *Пол клиента* [*gender*];
- категориальный признак *Семейное положение клиента* [*marital*];
- количественный признак *Количество детей у клиента* [*children*];
- количественный признак *Доход клиента* [*income*];
- бинарная зависимая переменная *Отток клиента* [*churn*].

Метрикой качества является AUC-ROC.

Давайте импортируем необходимые библиотеки, классы и функции и увеличим количество отображаемых строк в выводе.

```
# импортируем библиотеки numpy и pandas
import numpy as np
import pandas as pd
# импортируем функцию train_test_split(), с помощью
# которой разбиваем данные на обучающие и тестовые,
# и класс GridSearchCV, позволяющий выполнить
# поиск по сетке
from sklearn.model_selection import (train_test_split,
                                     GridSearchCV)
# импортируем класс ColumnTransformer, позволяющий выполнять
# преобразования для отдельных типов столбцов
from sklearn.compose import ColumnTransformer
# импортируем класс Pipeline, позволяющий создавать конвейеры
from sklearn.pipeline import Pipeline
# импортируем класс SimpleImputer, позволяющий
# выполнить импутацию пропусков
from sklearn.impute import SimpleImputer
# импортируем класс PowerTransformer, позволяющий
# выполнить преобразование Бокса-Кокса/Йео-Джонсона
# и стандартизацию, класс OneHotEncoder, позволяющий
# выполнить дамми-кодирование
from sklearn.preprocessing import (PowerTransformer,
                                   OneHotEncoder)
# импортируем класс LogisticRegression для построения
# логистической регрессии
from sklearn.linear_model import LogisticRegression
# импортируем функцию roc_auc_score()
# для вычисления AUC-ROC
from sklearn.metrics import roc_auc_score
# импортируем собственный класс Replacer
from utils import Replacer
# увеличиваем количество отображаемых строк
pd.set_option('display.max_rows', 200)
```

Загрузим данные и взглянем на первые 5 наблюдений.

```
# записываем CSV-файл в объект DataFrame
data = pd.read_csv('Data/Verizon.csv', sep=';')
# смотрим первые 5 наблюдений
data.head()
```

	longdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income	churn
0	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0
1	21.27	0.0	218,12	Нет	Бюджетный	СН	60.0	NaN	_Одинокий	2.0	95930,6	1
2	6,13	0.0	NaN	Да	NaN	NaN	25.0	Женский	NaN	2.0	295,34	1
3	16.46	0.0	57,66	Да	Бесплатный	NaN	93.0	Женский	Одинокий	0.0	NaN	1
4	NaN	0.0	16,01	Да	Бесплатный	СН	68.0	Женский*	NaN	0.0	99832,9	1

Видим, что в количественных переменных *longdist*, *internat*, *local* и *income* в качестве десятичного разделителя используется как точка (американский стандарт), так и запятая (европейский стандарт), есть пропуски, в строковых значениях категориальных переменных *gender* и *marital* встречаются причудливые символы.

Предварительную подготовку начинаем с того, что ищем дублирующиеся наблюдения. Здесь нам нужно воспользоваться методом `.duplicated()`.

```
# посмотрим наличие дублей
data[data.duplicated(keep=False)]
```

	longdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income	churn
0	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0
13	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0
14	8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	_Женат	0.0	33935.8	0

Видим три одинаковых наблюдения. С помощью метода `.drop_duplicates()` можно удалить дубли, при этом параметр `keep` позволяет оставить либо первое встретившееся наблюдение ('first'), либо последнее встретившееся наблюдение ('last'). Мы оставим первое встретившееся наблюдение.

```
# удаляем дубли на месте, оставляя первое
# встретившееся наблюдение в паттерне дубля
data.drop_duplicates(subset=None, keep='first',
                    inplace=True)
```

Затем ищем бесполезные переменные, т.е. переменные с одним уникальным значением и категориальные переменные, у которых категорий столько же, сколько уникальных значений. Заодно взглянем на типы переменных. С этой целью воспользуемся методом `nunique()` и свойством `dtypes`.

```
# создаем список переменных
cols_lst = data.columns.tolist()
# записываем количество уникальных
# значений по каждой переменной
uniq = [data[col].nunique() for col in cols_lst]
# записываем тип каждой переменной
types = data.dtypes
pd.DataFrame({'type': types, 'n_uniq': uniq})
```

	type	n_uniq
<b>longdist</b>	object	1081
<b>internat</b>	object	218
<b>local</b>	object	1372
<b>int_disc</b>	object	2
<b>billtype</b>	object	2
<b>pay</b>	object	4
<b>age</b>	float64	80
<b>gender</b>	object	4
<b>marital</b>	object	5
<b>children</b>	float64	3
<b>income</b>	object	1434
<b>churn</b>	int64	2

Похоже, что бесполезных переменных нет, но есть проблемы с количественными переменными *longdist*, *internat*, *local* и *income*, из-за десятичного разделителя – запятой – они записаны как категориальные. Давайте с помощью метода `str.replace()` заменим в переменных запятые на точки, а затем с помощью метода `.astype()` присвоим тип `float`.

```
# заменяем запятые на точки и преобразуем в тип float
for col in ['longdist', 'internat', 'local', 'income']:
    data[col] = data[col].str.replace(',', '.').astype('float')
```

Проверим типы переменных с помощью метода `.info()`, заодно посмотрим количество непропущенных значений.

```
# посмотрим типы переменных и количество
# непропущенных значений
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1475 entries, 0 to 1476
Data columns (total 12 columns):
#   Column      Non-Null Count  Dtype
---  -
0   longdist    1467 non-null   float64
1   internat    1469 non-null   float64
2   local       1466 non-null   float64
3   int_disc    1471 non-null   object
4   billtype    1450 non-null   object
5   pay         1470 non-null   object
6   age         1473 non-null   float64
7   gender      1469 non-null   object
8   marital     1471 non-null   object
9   children    1474 non-null   float64
10  income      1471 non-null   float64
```

```
11 churn      1475 non-null  int64
dtypes: float64(6), int64(1), object(5)
memory usage: 149.8+ KB
```

В более явном виде информацию о пропусках можно получить с помощью цепочки методов `.isnull()` и `.sum()`.

```
# смотрим пропуски в переменных
data.isnull().sum()
```

```
longdist      8
internat      6
local         9
int_disc      4
billtype     25
pay           5
age           2
gender        6
marital       4
children      1
income        4
churn         0
dtype: int64
```

Видим, что количественным переменным *longdist*, *internat*, *local* и *income* присвоен тип `float64`. Все переменные, кроме зависимой переменной *churn*, имеют пропуски.

Теперь приступаем к нормализации строковых значений. С помощью цикла `for` и метода `.unique()` выведем уникальные значения по каждой категориальной переменной.

```
# создаем список категориальных переменных
cat_cols = data.select_dtypes(
    include=['object']).columns.tolist()
# смотрим уникальные значения этих переменных
for col in cat_cols:
    print(col, data[col].unique())
```

```
int_disc ['Нет' 'Да' nan]
billtype ['Бюджетный' nan 'Бесплатный']
pay ['CH' nan 'CC' 'CD' 'Auto']
gender ['Мужской' nan 'Женский' 'Женский&*' 'Мужской&*']
marital ['_Женат' '_Одинокий' nan 'Одинокий' 'Женат' 'Же&нат']
```

Избавляемся от лишних символов `*&_` с помощью метода `str.replace()`.

```
# удаляем лишние символы в категориях
# переменных gender и marital
for col in ['gender', 'marital']:
    data[col] = data[col].str.replace('[*&_]', '')
# проверяем
for col in ['gender', 'marital']:
    print(col, data[col].unique())

gender ['Мужской' nan 'Женский']
marital ['Женат' 'Одинокий' nan]
```

Теперь выведем частоты категорий по категориальным переменным, чтобы выявить редкие категории.

```
# смотрим частоты по категориальным переменным,
# чтобы выявить редкие категории
for col in cat_cols:
    print(data[col].value_counts(dropna=False))
    print('')
```

```
Нет      1015
Да       456
NaN       4
Name: int_disc, dtype: int64
```

```
Бюджетный    731
Бесплатный   719
NaN          25
Name: billtype, dtype: int64
```

```
CC      846
CH      324
Auto    297
NaN      5
CD       3
Name: pay, dtype: int64
```

```
Женский    743
Мужской    726
NaN         6
Name: gender, dtype: int64
```

```
Женат      872
Одинокий   599
NaN         4
Name: marital, dtype: int64
```

У нас есть редкая категория 'CD' в переменной *pay*. Если обработка редких категорий заключается в том, чтобы отнести ее к отдельной категории на основе определенного порога (статистики), то такую обработку нужно делать после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки. Если обработка редких категорий опирается на априорные знания, здравый смысл и бизнес-логику, то такую обработку можно делать до разбиения на обучающую и тестовую выборки или до цикла перекрестной проверки. В нашем случае мы понимаем, что, скорее всего, категория 'CD' появилась в результате ошибки ввода, поэтому заменяем ее модой до разбиения на обучающую и тестовую выборки. Если у вас нет априорных знаний, объясняющих причину появления редкой категории, появление редкой категории нельзя объяснить здравым смыслом и бизнес-логикой, то обработку редких категорий (отнесение к отдельной категории по порогу или с помощью метода CHAID, случайное отнесение к одной из существующих категорий, которые мы не считаем редкими) делайте после разбиения на обучающую и тестовую выборки или внутри цикла перекрестной проверки.

```
# заменяем редкую категорию модой
data.loc[data['pay'] == 'CD', 'pay'] = 'CC'
```

Теперь создадим переменную – взаимодействие переменных *gender* и *marital*, по сути делаем конъюнкцию строковых значений переменных *gender* и *marital*.

Пол	Семейное положение	Взаимодействие
Мужской	Женат	Мужской + Женат
Мужской	Женат	Мужской + Женат
Мужской	Холост	Мужской + Холост
Женский	Замужем	Женский + Замужем

Обязательно нужно предусмотреть обработку пропусков у переменных, участвующих во взаимодействии.

```
# создаем переменную - результат конъюнкции
data['gender_marital'] = np.where(
    (data['gender'].isnull()) | (data['marital'].isnull()),
    np.NaN,
    data.apply(lambda x: f"{x['gender']} + {x['marital']}", axis=1))
```

В данном случае если хотя бы в одной из переменных есть пропуск, во взаимодействие мы записываем пропуск. Такая стратегия используется, когда пропуски редки, как в нашем случае. Здесь значение NaN является временным, после разбиения на обучающую и тестовую выборки мы заменим ее модой.

Теперь создадим переменные-отношения.

```
# поделим возраст на длительность междугородних звонков в минутах
cond = (data['age'] == 0) | (data['longdist'] == 0)
data['ratio1'] = np.where(cond, 0, data['age'] / data['longdist'])
# поделим длительность междугородних звонков в минутах на
# длительность международных звонков в минутах
cond = (data['longdist'] == 0) | (data['internat'] == 0)
data['ratio2'] = np.where(cond, 0, data['longdist'] / data['internat'])
# поделим доход на возраст
cond = (data['income'] == 0) | (data['age'] == 0)
data['ratio3'] = np.where(cond, 0, data['income'] / data['age'])
# поделим возраст на количество детей
cond = (data['age'] == 0) | (data['children'] == 0)
data['ratio4'] = np.where(cond, 0, data['age'] / data['children'])
```

Обратите внимание: все преобразования, которые мы применили, и все переменные, которые мы создали, не предполагают вычисления статистик, поэтому мы их выполнили до разбиения на обучающую и тестовую выборки. Небольшим исключением здесь является обработка редких категорий, однако тут причина появления редкой категории была очевидной.

Теперь разбиваем данные на обучающую и тестовую выборки.

```
# разбиваем данные на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('churn', axis=1),
    data['churn'],
```

```
test_size=0.3,
stratify=data['churn'],
random_state=42)
```

И создаем список категориальных признаков и список количественных признаков.

```
# создаем список категориальных признаков
# и список количественных признаков
cat_columns = X_train.select_dtypes(
    include='object').columns.tolist()
num_columns = X_train.select_dtypes(
    exclude='object').columns.tolist()
```

Далее подготавливаем трансформеры, передаем их в ColumnTransformer. Созданный экземпляр класса ColumnTransformer вместе с экземпляром класса LogisticRegression() передаем в итоговый конвейер, обучаем его, не перебирая гиперпараметры, и оцениваем качество его прогнозов. По сути, строим и оцениваем конвейер базовых моделей.

```
# создаем трансформер для категориальных переменных
cat_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('ohe', OneHotEncoder(sparse=False,
                          handle_unknown='ignore'))
])

# создаем трансформер для количественных переменных
num_pipe = Pipeline([
    ('imp', SimpleImputer()),
    ('repl', Replacer(repl_value=0.1)),
    ('boxcox', PowerTransformer(method='box-cox',
                                standardize=True))
])

# создаем список трехэлементных кортежей, в котором
# первый элемент кортежа - название трансформера с
# преобразованиями для определенного типа признаков
transformers = [('num', num_pipe, num_columns),
                ('cat', cat_pipe, cat_columns)]

# передаем список в ColumnTransformer
ct = ColumnTransformer(transformers=transformers)

# задаем итоговый конвейер
ml_pipe = Pipeline([
    ('tr', ct),
    ('logreg', LogisticRegression(solver='liblinear'))
])

# обучаем конвейер базовых моделей
ml_pipe.fit(X_train, y_train)
# оцениваем качество конвейера на обучающих данных
print("AUC-ROC на обучающей выборке: {:.3f}".format(
    roc_auc_score(y_train, ml_pipe.predict_proba(X_train)[: , 1])))
```

```
# оцениваем качество конвейера на тестовых данных
print("AUC-ROC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, ml_pipe.predict_proba(X_test)[: , 1])))
```

AUC-ROC на обучающей выборке: 0.885

AUC-ROC на тестовой выборке: 0.861

Теперь задаем сетку гиперпараметров (перебираем стратегии импутации для категориальных и количественных признаков, заменяющее значение, силу регуляризации), создаем экземпляр класса GridSearchCV, выполняем обычный поиск по сетке и печатаем результаты поиска.

```
# задаем сетку значений гиперпараметров
param_grid = {
    'tr_num_imp_strategy': ['mean', 'median', 'constant'],
    'tr_num_repl_repl_value': [0.1, 0.2, 0.3],
    'tr_cat_imp_strategy': ['most_frequent', 'constant'],
    'logreg_C': np.logspace(-2, 1, 10)
}
# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров, количество блоков
# перекрестной проверки, метрику
gs = GridSearchCV(ml_pipe,
                  param_grid,
                  cv=5,
                  scoring='roc_auc')
# выполняем поиск по сетке
gs.fit(X_train, y_train)
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs.best_params_))
# смотрим наилучшее значение AUC-ROC
print("Наилучшее значение AUC-ROC: {:.3f}".format(
    gs.best_score_))
# смотрим AUC-ROC на тестовой выборке
print("Значение AUC-ROC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, gs.predict_proba(X_test)[: , 1])))
```

Наилучшие значения гиперпараметров:

```
{'logreg_C': 0.46415888336127775,
 'tr_cat_imp_strategy': 'constant',
 'tr_num_imp_strategy': 'constant',
 'tr_num_repl_repl_value': 0.3}
```

Наилучшее значение AUC-ROC: 0.879

Значение AUC-ROC на тестовой выборке: 0.866

```
# запишем результаты поиска в DataFrame
results = pd.DataFrame(gs.cv_results_)
# превращаем в сводную таблицу
table = results.pivot_table(
    values=['mean_test_score'],
    index=['param_logreg_C',
           'param_tr_num_imp_strategy',
           'param_tr_num_repl_repl_value',
           'param_tr_cat_imp_strategy'])
```

```
table.sort_values('mean_test_score',
                  ascending=False,
                  inplace=True)
table
```

					mean_test_score
param_logreg_C	param_tr_num_imp_strategy	param_tr_num_repl_repl_value	param_tr_cat_imp_strategy		
0.464159	constant	0.3	constant	0.879293	
1.000000	constant	0.3	constant	0.879219	
10.000000	constant	0.3	most_frequent	0.879077	
4.641589	constant	0.3	most_frequent	0.879038	
2.154435	constant	0.3	most_frequent	0.878941	
1.000000	constant	0.3	most_frequent	0.878845	
0.464159	constant	0.3	most_frequent	0.878823	
0.215443	constant	0.3	constant	0.878776	
1.000000	median	0.3	constant	0.878671	

Мы нашли оптимальные значения гиперпараметров, теперь нам надо обучить конвейер с этими оптимальными значениями на всей исторической выборке.

Заново загружаем данные.

```
# записываем CSV-файл в объект DataFrame
fulldata = pd.read_csv('Data/Verizon.csv', sep=';')
```

Теперь пишем функцию предварительной подготовки `rpreprocessing()`, в которой процедуры обработки данных идут в том же порядке, в каком мы их выполняли ранее. Функция гарантирует, что данные в исторической выборке и в новых данных будут обработаны единообразно. В функцию мы помещаем только те операции преобразования данных и конструирования признаков, которые не использовали статистики и были выполнены до разбиения на обучающую и тестовую выборки. Функцию пишем только после того, как убедились с помощью перекрестной проверки, что именно данная комбинация процедур, которую мы помещаем в функцию, дает лучшее качество. Кроме того, в функции нужно предусмотреть возможность обработки новых категорий. Например, мы знаем, что в переменной *rau* могут появиться новые категории (например, клиент заполняет данное поле самостоятельно, и могут появиться ошибки ввода). Все новые категории заменяем модой. В категориях переменных *gender* и *marital* у нас есть странные символы. Исходя из априорных знаний, мы полагаем, что это могут быть символы *\*&\_* и цифры. Тогда мы просто будем удалять символы, не являющиеся буквами, и цифры. Операции предварительной подготовки, использующие статистики, в данной задаче выполняются в трансформерах конвейера, которые запускаются после вызова функции `rpreprocessing()`.

```
# записываем CSV-файл в объект DataFrame
fulldata = pd.read_csv('Data/Verizon.csv', sep=';')
```

```
# пишем функцию, выполняющую предварительную
# обработку всех исторических данных
```

```
def preprocessing(df):
    # удаляем дубли на месте, оставляя первое
    # встретившееся наблюдение в паттерне дубли
    df.drop_duplicates(subset=None, keep='first', inplace=True)
    # заменяем запятые на точки и преобразуем в тип float
    for i in ['longdist', 'internat', 'local', 'income']:
        df[i] = df[i].str.replace(',', '.').astype('float')
    # удаляем возможные лишние символы и цифры в категориях
    # переменных gender и marital
    for i in ['gender', 'marital']:
        df[i] = df[i].str.replace('[\d+W_]', '')
    # все новые категории переменной pay заменяем модой
    lst = ['CC', 'Auto', 'CH', np.NaN]
    replace_new_values = lambda x: 'CC' if x not in lst else x
    df['pay'] = df['pay'].map(replace_new_values)
    # создаем переменную - результат конъюнкции
    df['gender_marital'] = np.where(
        (df['gender'].isnull()) | (df['marital'].isnull()),
        np.NaN,
        df.apply(lambda x: f"{x['gender']} + {x['marital']}", axis=1))
    # поделим возраст на длительность междугородних звонков в минутах
    cond = (df['age'] == 0) | (df['longdist'] == 0)
    df['ratio'] = np.where(cond, 0, df['age'] / df['longdist'])
    # поделим длительность междугородних звонков в минутах на
    # длительность международных звонков в минутах
    cond = (df['longdist'] == 0) | (df['internat'] == 0)
    df['ratio2'] = np.where(cond, 0, df['longdist'] / df['internat'])
    # поделим доход на возраст
    cond = (df['income'] == 0) | (df['age'] == 0)
    df['ratio3'] = np.where(cond, 0, df['income'] / df['age'])
    # поделим возраст на количество детей
    cond = (df['age'] == 0) | (df['children'] == 0)
    df['ratio4'] = np.where(cond, 0, df['age'] / df['children'])
    return df
```

Применяем функцию предварительной подготовки preprocessing() к нашим историческим данным и выводим первые 5 наблюдений.

```
# применяем функцию предварительной обработки
# ко всем историческим данным
fulldata = preprocessing(fulldata)
fulldata.head()
```

ngdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income	churn	gender_marital	ratio	ratio2	ratio3	ratio4
8.62	NaN	8.49	Нет	Бюджетный	СН	43.0	Мужской	Женат	0.0	33935.80	0	Мужской + Женат	4.988399	NaN	789.204651	0.0
21.27	0.0	218.12	Нет	Бюджетный	СН	60.0	NaN	Одинокий	2.0	95930.60	1	NaN	2.820874	0.0	1598.843333	30.0
6.13	0.0	NaN	Да	NaN	NaN	25.0	Женский	NaN	2.0	295.34	1	NaN	4.078303	0.0	11.813600	12.5
16.46	0.0	57.66	Да	Бесплатный	NaN	93.0	Женский	Одинокий	0.0	NaN	1	Женский + Одинокий	6.650061	0.0	NaN	0.0
NaN	0.0	16.01	Да	Бесплатный	СС	68.0	Женский	NaN	0.0	99832.90	1	NaN	NaN	0.0	1468.130882	0.0

Теперь создаем массив меток и массив признаков.

```
# создаем массив меток и массив признаков
y_fulldata = fulldata.pop('churn')
```

С помощью атрибута `best_params_` класса `GridSearchCV` извлекаем оптимальные гиперпараметры. С помощью метода `.set_params()` и оператора `**` передаем в конвейер оптимальные гиперпараметры и обучаем конвейер уже на всей исторической выборке.

```
# записываем оптимальные значения гиперпараметров
best_params = gs.best_params_
# присваиваем итоговому конвейеру оптимальные
# значения гиперпараметров
ml_pipe.set_params(**best_params)
# обучаем итоговый конвейер с оптимальными значениями
# гиперпараметров на всех исторических данных
ml_pipe.fit(fulldata, y_fulldata);
```

Теперь давайте загрузим новые данные и взглянем на них.

```
# записываем CSV-файл, содержащий новые данные,
# в объект DataFrame
newdata = pd.read_csv('Data/Verizon_new.csv', sep=';')
newdata.head()
```

	longdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income
0	19,46	7,11	64,51	Нет	Бюджетный	СН	28.0	Женский7	Одинокий	1.0	43787
1	18,42	6,46	28,07	Да	Бесплатный	СН	35.0	Женский&*	Женат	2.0	47396,2
2	10,97	0	60	Да	Бесплатный	СН	45.0	Мужской\$	Женат	0.0	30057,4
3	22,18	0	149,73	Нет	Бесплатный	НН	41.0	Мужской\$	Одинокий	1.0	21126,7
4	0	0	4,15	Да	Бесплатный	СН	40.0	Женский&*	_Одинокий	2.0	81000,9

Применяем функцию предварительной подготовки `preprocessing()` к новым данным и выводим первые 5 наблюдений.

```
# применяем функцию предварительной обработки
# к новым данным
newdata = preprocessing(newdata)
newdata.head()
```

	longdist	internat	local	int_disc	billtype	pay	age	gender	marital	children	income	gender_marital	ratio	ratio2	ratio3	ratio4
0	19.46	7.11	64.51	Нет	Бюджетный	СН	28.0	Женский	Одинокий	1.0	43787.0	Женский + Одинокий	1.438849	2.736990	1563.821429	28.0
1	18.42	6.46	28.07	Да	Бесплатный	СН	35.0	Женский	Женат	2.0	47396.2	Женский + Женат	1.900109	2.851393	1354.177143	17.5
2	10.97	0.00	60.00	Да	Бесплатный	СН	45.0	Мужской	Женат	0.0	30057.4	Мужской + Женат	4.102097	0.000000	667.942222	0.0
3	22.18	0.00	149.73	Нет	Бесплатный	СС	41.0	Мужской	Одинокий	1.0	21126.7	Мужской + Одинокий	1.848512	0.000000	515.285366	41.0
4	0.00	0.00	4.15	Да	Бесплатный	СН	40.0	Женский	Одинокий	2.0	81000.9	Женский + Одинокий	0.000000	0.000000	2025.022500	20.0

Теперь пишем функцию `check_vars_order()`, проверяющую порядок признаков в наборах. В ряде ситуаций несовпадение порядка столбцов в историческом наборе и наборе новых данных может дать некорректные результаты. Класс `Pipeline` также предупредит, что порядок столбцов на этапе применения отличается от порядка столбцов на этапе обучения: `feature names must be in the same order as they were in fit`.

```
# пишем функцию, проверяющую порядок переменных в наборах
def check_vars_order(hist_data, new_data):
    if hist_data == new_data:
        print("Одни и те же списки переменных, " +
              "один и тот же порядок.")
    elif sorted(hist_data) == sorted(new_data):
        print("Одни и те же списки переменных, " +
              "разный порядок. ")
    else:
        print("Совершенно разные списки переменных.")
```

Итак, выполняем проверку совпадения порядка столбцов в историческом наборе и наборе новых данных.

```
# создаем списки переменных для исторического набора
# и набора новых данных
fulldata_cols = fulldata.columns.tolist()
newdata_cols = newdata.columns.tolist()

# выполняем проверку совпадения порядка столбцов
# в историческом наборе и наборе новых данных
check_vars_order(fulldata_cols, newdata_cols)
```

Одни и те же списки переменных, один и тот же порядок.

Порядок столбцов совпадает.

Используя конвейер с оптимальными гиперпараметрами, обученный на всей исторической выборке, мы вычисляем вероятности классов для новых данных.

```
# при помощи итогового конвейера с оптимальными значениями
# гиперпараметров, обученного на всей исторической выборке,
# вычисляем вероятности классов для новых данных
proba = ml_pipe.predict_proba(newdata)
# выведем вероятности классов для первых пяти наблюдений
proba[:5]

array([[0.04534913, 0.95465087],
       [0.05034639, 0.94965361],
       [0.9233456 , 0.0766544 ],
       [0.9238732 , 0.0761268 ],
       [0.03992666, 0.96007334]])
```

## 7.17.2. Вторая задача

Приступаем ко второй задаче. Речь идет о соревновании Categorical Feature Encoding Challenge II на Kaggle <https://www.kaggle.com/c/cat-in-the-dat-ii/overview>. Исторический набор содержит 600 000 наблюдений и 25 переменных. Нужно придумать такой способ обработки категориальных признаков, который даст наилучшее качество с точки зрения AUC-ROC. Импортируем необходимые библиотеки, классы, функции и загружаем данные.

```
# импортируем библиотеки numpy и pandas
import numpy as np
```

```
import pandas as pd
# импортируем функцию train_test_split(), с помощью
# которой разбиваем данные на обучающие и тестовые,
# и класс GridSearchCV, позволяющий выполнить
# поиск по сетке
from sklearn.model_selection import (train_test_split,
                                     GridSearchCV)

# импортируем класс Pipeline, позволяющий
# создавать конвейеры
from sklearn.pipeline import Pipeline
# импортируем класс OneHotEncoder, позволяющий
# выполнить дамми-кодирование
from sklearn.preprocessing import OneHotEncoder
# импортируем класс LogisticRegression для построения
# логистической регрессии
from sklearn.linear_model import LogisticRegression
# импортируем функцию roc_auc_score()
# для вычисления AUC-ROC
from sklearn.metrics import roc_auc_score
# импортируем собственную функцию check_vars_order
# для проверки порядка столбцов в наборах
from utils import check_vars_order
# увеличиваем количество отображаемых столбцов
pd.set_option('display.max_columns', 50)

# загружаем и смотрим данные
data = pd.read_csv('Data/catfeatures_challenge_II_train.csv')
data.head()
```

	nom_2	nom_3	nom_4	nom_5	nom_6	nom_7	nom_8	nom_9	ord_0	ord_1	ord_2	ord_3	ord_4	ord_5	day	month	target
Hamster	Russia	Bassoon	de4c57ee2	a64bc7ddf	598080a91	0256c7a4b	02e7c8990	3.0	Contributor	Hot	c	U	Pw	6.0	3.0	0	
Axolotl	NaN	Theremin	2bb3c3e5c	3a3a936e8	1dddb8473	52ead350c	f37df64af	3.0	Grandmaster	Warm	e	X	pE	7.0	7.0	0	
Hamster	Canada	Bassoon	b574c9841	708248125	5ddc9a726	745b909d1	NaN	3.0	NaN	Freezing	n	P	eN	5.0	9.0	0	
Hamster	Finland	Theremin	673bdf1f6	23eddb8da3	3a33ef960	bdaa56dd1	f9d456e57	1.0	Novice	Lava Hot	a	C	NaN	3.0	3.0	0	
Hamster	Costa Rica	NaN	777d1ac2c	3a7975e46	bc9cc2a94	NaN	c5361037c	3.0	Grandmaster	Cold	h	C	OZ	5.0	12.0	0	

Зависимой переменной является переменная *target*, бинарные признаки помечены префиксом *bin\_*, номинальные – префиксом *nom\_*, порядковые – префиксом *ord\_*, также есть два циклических признака времени *day* и *month*.

Взглянем на типы переменных и количество непропущенных значений в каждой переменной.

```
# смотрим типы переменных и количество
# непропущенных наблюдений
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 600000 entries, 0 to 599999
Data columns (total 25 columns):
#   Column      Non-Null Count  Dtype
---  -
0    id           600000 non-null  int64
1    bin_0       582106 non-null  float64
2    bin_1       581997 non-null  float64
3    bin_2       582070 non-null  float64
```

```

4 bin_3 581986 non-null object
5 bin_4 581953 non-null object
6 nom_0 581748 non-null object
7 nom_1 581844 non-null object
8 nom_2 581965 non-null object
9 nom_3 581879 non-null object
10 nom_4 581965 non-null object
11 nom_5 582222 non-null object
12 nom_6 581869 non-null object
13 nom_7 581997 non-null object
14 nom_8 582245 non-null object
15 nom_9 581927 non-null object
16 ord_0 581712 non-null float64
17 ord_1 581959 non-null object
18 ord_2 581925 non-null object
19 ord_3 582084 non-null object
20 ord_4 582070 non-null object
21 ord_5 582287 non-null object
22 day 582048 non-null float64
23 month 582012 non-null float64
24 target 600000 non-null int64
dtypes: float64(6), int64(2), object(17)
memory usage: 114.4+ MB

```

Поскольку задача является конкурсной, мы немного отойдем от правил. Здесь мы сразу напишем функцию предварительной подготовки данных `preprocessing()`. Помним, что до разбиения на обучающую и тестовую выборки мы можем применять только те преобразования, которые не предполагают вычисление статистик. Сейчас мы укрупним редкие категории до разбиения на обучающую и тестовую выборки. В бизнес-подходе вы так можете делать только в том случае, если у вас есть априорные знания, которым вы доверяете. Кроме того, мы создадим индикаторы пропусков, которые могут быть весьма эффективны для линейных моделей. Однако опять же в бизнес-подходе индикаторы пропусков как самостоятельные признаки используются редко и носят временный характер для проверки гипотез, касающихся причин появления пропусков.

```

# пишем функцию предварительной подготовки
def preprocessing(df, newdf=False):
    # избавляемся от редких категорий
    for col in ['nom_5', 'nom_6', 'nom_7', 'nom_8', 'nom_9']:
        abs_freq = df[col].value_counts(dropna=False)
        df[col] = np.where(df[col].isin(
            abs_freq[abs_freq >= 50].index.tolist()),
            df[col], 'Other')

    # удаляем идентификатор
    df.drop('id', axis=1, inplace=True)

    if newdf:
        # для всех столбцов создаем
        # индикаторы пропусков
        for col in df.columns:
            df[col + '_isnan'] = np.where(
                df[col].isnull(), 'T', 'F')

```

```

else:
    # для всех столбцов, кроме target,
    # создаем индикаторы пропусков
    for col in df.columns.difference(['target']):
        df[col + '_isnan'] = np.where(
            df[col].isnull(), 'T', 'F')

# создаем две новые переменные на основе переменной ord5,
# просто извлекая первый и второй символы
df['ord_5a'] = df['ord_5'].str[0]
df['ord_5b'] = df['ord_5'].str[1]

if newdf:
    # присваиваем всем переменным тип str, пропуски
    # формируют отдельную категорию
    for col in df.columns:
        df[col] = df[col].astype('str')
else:
    # присваиваем всем переменным, кроме target, тип str,
    # пропуски формируют отдельную категорию
    for col in df.columns.difference(['target']):
        df[col] = df[col].astype('str')
return df

```

Применяем нашу функцию предварительной обработки `preprocessing()` и снова смотрим типы переменных и количество непропущенных значений в переменных.

```

# применяем нашу функцию предобработки
data = preprocessing(data)
data.head()

```

	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3	nom_4	nom_5	nom_6	nom_7	nom_8	nom_9	ord_0
0	0.0	0.0	0.0	F	N	Red	Trapezoid	Hamster	Russia	Bassoon	de4c57ee2	a64bc7ddf	598080a91	0256c7a4b	02e7c8990	3.0
1	1.0	1.0	0.0	F	Y	Red	Star	Axolotl	nan	Theremin	2bb3c3e5c	3a3a936e8	1dddb8473	52ead350c	f37df64af	3.0
2	0.0	1.0	0.0	F	N	Red	nan	Hamster	Canada	Bassoon	b574c9841	708248125	5ddc9a726	745b909d1	nan	3.0
3	nan	0.0	0.0	F	N	Red	Circle	Hamster	Finland	Theremin	673bdf1f6	23edb8da3	3a33ef960	bdaa56dd1	f9d456e57	1.0
4	0.0	nan	0.0	T	N	Red	Triangle	Hamster	Costa Rica	nan	777d1ac2c	3a7975e46	bc9cc2a94	nan	c5361037c	3.0

```

# снова смотрим типы переменных и
# количество непропущенных значений
data.info()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 600000 entries, 0 to 599999
Data columns (total 49 columns):
#   Column          Non-Null Count  Dtype
---  -
0   bin_0           600000 non-null  object
1   bin_1           600000 non-null  object
2   bin_2           600000 non-null  object
3   bin_3           600000 non-null  object
4   bin_4           600000 non-null  object
5   nom_0           600000 non-null  object
6   nom_1           600000 non-null  object

```

```

7  nom_2      600000 non-null object
8  nom_3      600000 non-null object
9  nom_4      600000 non-null object
10 nom_5      600000 non-null object
11 nom_6      600000 non-null object
12 nom_7      600000 non-null object
13 nom_8      600000 non-null object
14 nom_9      600000 non-null object
15 ord_0      600000 non-null object
16 ord_1      600000 non-null object
17 ord_2      600000 non-null object
18 ord_3      600000 non-null object
19 ord_4      600000 non-null object
20 ord_5      600000 non-null object
21 day        600000 non-null object
22 month      600000 non-null object
23 target     600000 non-null int64
24 bin_0_isnan 600000 non-null object
25 bin_1_isnan 600000 non-null object
26 bin_2_isnan 600000 non-null object
27 bin_3_isnan 600000 non-null object
28 bin_4_isnan 600000 non-null object
29 day_isnan   600000 non-null object
30 month_isnan 600000 non-null object
31 nom_0_isnan 600000 non-null object
32 nom_1_isnan 600000 non-null object
33 nom_2_isnan 600000 non-null object
34 nom_3_isnan 600000 non-null object
35 nom_4_isnan 600000 non-null object
36 nom_5_isnan 600000 non-null object
37 nom_6_isnan 600000 non-null object
38 nom_7_isnan 600000 non-null object
39 nom_8_isnan 600000 non-null object
40 nom_9_isnan 600000 non-null object
41 ord_0_isnan 600000 non-null object
42 ord_1_isnan 600000 non-null object
43 ord_2_isnan 600000 non-null object
44 ord_3_isnan 600000 non-null object
45 ord_4_isnan 600000 non-null object
46 ord_5_isnan 600000 non-null object
47 ord_5a     600000 non-null object
48 ord_5b     600000 non-null object
dtypes: int64(1), object(48)
memory usage: 224.3+ MB

```

Разбиваем набор данных на обучающую и тестовую выборки.

```

# разбиваем данные на обучающие и тестовые: получаем
# обучающий массив признаков, тестовый массив признаков,
# обучающий массив меток, тестовый массив меток
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis=1),
    data['target'],
    test_size=0.3,
    stratify=data['target'],
    random_state=42)

```

Создаем простой конвейер из классов `OneHotEncoder` и `LogisticRegression`, сетку гиперпараметров, экземпляр класса `GridSearchCV` и выполняем обычный поиск по сетке. По сути, мы выполняем дамми-кодирование с помощью класса и строим логистическую регрессию, используя разные значения силы регуляризации, задача – найти значение силы регуляризации, дающее наивысшее значение AUC по результатам перекрестной проверки.

```
# создаем итоговый конвейер
ml_pipe = Pipeline([
    ('ohe', OneHotEncoder(sparse=True,
                          handle_unknown='ignore')),
    ('logreg', LogisticRegression(solver='liblinear'))
])
# задаем сетку гиперпараметров
param_grid = {'logreg__C': [0.01, 0.05, 0.1, 0.5, 1]}

# создаем экземпляр класса GridSearchCV
gs = GridSearchCV(ml_pipe,
                  param_grid,
                  scoring='roc_auc',
                  cv=5)
# выполняем поиск по всем значениям сетки
gs.fit(X_train, y_train);
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров: {}".format(
    gs.best_params_))
# оцениваем наилучшее значение AUC
print("Наилучшее значение AUC: {:.3f}".format(
    roc_auc_score(y_train, gs.predict_proba(X_train)[:, 1])))
# оцениваем AUC модели на тестовой выборке
print("AUC на тестовой выборке: {:.3f}".format(
    roc_auc_score(y_test, gs.predict_proba(X_test)[:, 1])))

Наилучшие значения гиперпараметров: {'logreg__C': 0.05}
Наилучшее значение AUC: 0.798
AUC-ROC на тестовой выборке: 0.790
```

Мы нашли оптимальное значение силы регуляризации, теперь нам надо обучить конвейер с этим оптимальным значением на всей исторической выборке. Заново загружаем данные.

```
# загружаем данные
fulldata = pd.read_csv('Data/catfeatures_challenge_II_train.csv')
```

Применяем функцию предварительной подготовки данных, формируем массив меток и массив признаков.

```
# применяем нашу функцию предобработки
fulldata = preprocessing(fulldata)
fulldata.head()
```

	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3	nom_4	nom_5	nom_6	nom_7	nom_8	nom_9	ord_0
0	0.0	0.0	0.0	F	N	Red	Trapezoid	Hamster	Russia	Bassoon	de4c57ee2	a64bc7ddf	598080a91	0256c7a4b	02e7c8990	3.0
1	1.0	1.0	0.0	F	Y	Red	Star	Axolotl	nan	Theremin	2bb3c3e5c	3a3a936e8	1dddb8473	52ead350c	f37df64af	3.0
2	0.0	1.0	0.0	F	N	Red	nan	Hamster	Canada	Bassoon	b574c9841	708248125	5ddc9a726	745b909d1	nan	3.0
3	nan	0.0	0.0	F	N	Red	Circle	Hamster	Finland	Theremin	673bdf1f6	23edb8da3	3a33ef960	bdaa56dd1	f9d456e57	1.0
4	0.0	nan	0.0	T	N	Red	Triangle	Hamster	Costa Rica	nan	777d1ac2c	3a7975e46	bc9cc2a94	nan	c5361037c	3.0

```
# создаем массив меток и массив признаков
y_fullldata = fullldata.pop('target')
```

Теперь обучаем итоговый конвейер с найденным оптимальным значением силы регуляризации на всей исторической выборке.

```
# записываем оптимальные значения гиперпараметров
best_params = gs.best_params_
# присваиваем итоговому конвейеру оптимальные
# значения гиперпараметров
ml_pipe.set_params(**best_params)
# обучаем итоговый конвейер с оптимальными значениями
# гиперпараметров на всех исторических данных
ml_pipe.fit(fullldata, y_fullldata)
```

Теперь с помощью конвейера с оптимальным значением силы регуляризации, обученного на всей исторической выборке, мы вычисляем вероятности положительного класса для новых данных, формируем посылку и отправляем ее на Kaggle.

Давайте загрузим новые данные и взглянем на них.

```
# загружаем и смотрим новые данные
newdata = pd.read_csv('Data/catfeatures_challenge_II_test.csv')
newdata.head()
```

	id	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3	nom_4	nom_5	nom_6	nom_7	nom_8	nom_9	ord_0
0	600000	0.0	0.0	0.0	F	Y	Blue	Polygon	Axolotl	Finland	Piano	52f6dd16c	147d704e4	8d857a0a1	ca9ad1d4b	fcde9e114	3.0
1	600001	0.0	0.0	0.0	F	Y	Red	Circle	Lion	Russia	Bassoon	691ebee8	8653dcc2e	67a8d4ebb	060a21580	7ca8775da	1.0
2	600002	0.0	0.0	0.0	F	Y	Blue	Circle	Axolotl	Russia	Theremin	81f792c16	6cdda499e	69403e18c	165e81a00	5940334c9	1.0
3	600003	1.0	0.0	0.0	F	N	Red	Polygon	Axolotl	Costa Rica	Bassoon	c9134205b	acbc4a4827	cb681246b	77d41330d	6fbdeefc8	1.0
4	600004	0.0	0.0	1.0	F	Y	Red	Circle	NaN	Finland	Theremin	f0f100f57	6f800b9af	cd9feb5c6	2218d9dfe	2a27c8fde	1.0

Сохраним идентификатор для посылки.

```
# сохраним ID для посылки
ident = newdata['id']
```

Смотрим типы переменных и количество непропущенных значений в переменных.

```
# снова смотрим типы переменных и
# количество непропущенных значений
newdata.info()
```

```
class 'pandas.core.frame.DataFrame'>
```

```

RangeIndex: 400000 entries, 0 to 399999
Data columns (total 24 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   id           400000 non-null  int64
1   bin_0       388099 non-null  float64
2   bin_1       387962 non-null  float64
3   bin_2       388028 non-null  float64
4   bin_3       388049 non-null  object
5   bin_4       388049 non-null  object
6   nom_0       387938 non-null  object
7   nom_1       388053 non-null  object
8   nom_2       387821 non-null  object
9   nom_3       387824 non-null  object
10  nom_4       388007 non-null  object
11  nom_5       388088 non-null  object
12  nom_6       387988 non-null  object
13  nom_7       387997 non-null  object
14  nom_8       388044 non-null  object
15  nom_9       387940 non-null  object
16  ord_0       388107 non-null  float64
17  ord_1       387833 non-null  object
18  ord_2       387895 non-null  object
19  ord_3       387947 non-null  object
20  ord_4       388067 non-null  object
21  ord_5       387953 non-null  object
22  day         387975 non-null  float64
23  month       388016 non-null  float64
dtypes: float64(6), int64(1), object(17)
memory usage: 73.2+ MB

```

Применяем нашу функцию предварительной обработки `preprocessing()` и снова смотрим типы переменных и количество непропущенных значений в переменных.

*# применяем нашу функцию предобработки к новым данным*

```

newdata = preprocessing(newdata, newdf=True)
newdata.head()

```

	bin_0	bin_1	bin_2	bin_3	bin_4	nom_0	nom_1	nom_2	nom_3	nom_4	nom_5	nom_6	nom_7	nom_8	nom_9	ord_0	ord_1
0	0.0	0.0	0.0	F	Y	Blue	Polygon	Axolotl	Finland	Piano	52f6dd16c	147d704e4	8d857a0a1	ca9ad1d4b	fcde9e114	3.0	Novice
1	0.0	0.0	0.0	F	Y	Red	Circle	Lion	Russia	Bassoon	691ebeeae8	8653dcc2e	67a8d4ebb	060a21580	7ca8775da	1.0	Novice
2	0.0	0.0	0.0	F	Y	Blue	Circle	Axolotl	Russia	Theremin	81f792c16	6cdda499e	69403e18c	165e81a00	5940334c9	1.0	Expert
3	1.0	0.0	0.0	F	N	Red	Polygon	Axolotl	Costa Rica	Bassoon	c9134205b	acbaa4827	cb681246b	77d41330d	6fbdeefc8	1.0	Expert
4	0.0	0.0	1.0	F	Y	Red	Circle	nan	Finland	Theremin	f0f100f57	6f800b9af	cd9feb5c6	2218d9dfe	2a27c8fde	1.0	Contributor

*# снова смотрим типы переменных и*

*# количество непропущенных значений*

```

newdata.info()

```

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400000 entries, 0 to 399999
Data columns (total 48 columns):
#   Column      Non-Null Count  Dtype
---  ---

```

```
0 bin_0      400000 non-null object
1 bin_1      400000 non-null object
2 bin_2      400000 non-null object
3 bin_3      400000 non-null object
4 bin_4      400000 non-null object
5 nom_0      400000 non-null object
6 nom_1      400000 non-null object
7 nom_2      400000 non-null object
8 nom_3      400000 non-null object
9 nom_4      400000 non-null object
10 nom_5     400000 non-null object
11 nom_6     400000 non-null object
12 nom_7     400000 non-null object
13 nom_8     400000 non-null object
14 nom_9     400000 non-null object
15 ord_0     400000 non-null object
16 ord_1     400000 non-null object
17 ord_2     400000 non-null object
18 ord_3     400000 non-null object
19 ord_4     400000 non-null object
20 ord_5     400000 non-null object
21 day       400000 non-null object
22 month     400000 non-null object
23 bin_0_isnan 400000 non-null object
24 bin_1_isnan 400000 non-null object
25 bin_2_isnan 400000 non-null object
26 bin_3_isnan 400000 non-null object
27 bin_4_isnan 400000 non-null object
28 nom_0_isnan 400000 non-null object
29 nom_1_isnan 400000 non-null object
30 nom_2_isnan 400000 non-null object
31 nom_3_isnan 400000 non-null object
32 nom_4_isnan 400000 non-null object
33 nom_5_isnan 400000 non-null object
34 nom_6_isnan 400000 non-null object
35 nom_7_isnan 400000 non-null object
36 nom_8_isnan 400000 non-null object
37 nom_9_isnan 400000 non-null object
38 ord_0_isnan 400000 non-null object
39 ord_1_isnan 400000 non-null object
40 ord_2_isnan 400000 non-null object
41 ord_3_isnan 400000 non-null object
42 ord_4_isnan 400000 non-null object
43 ord_5_isnan 400000 non-null object
44 day_isnan  400000 non-null object
45 month_isnan 400000 non-null object
46 ord_5a    400000 non-null object
47 ord_5b    400000 non-null object
dtypes: object(48)
memory usage: 146.5+ MB
```

Теперь создаем списки переменных для исторического набора и набора новых данных и выполняем проверку совпадения порядка столбцов в историческом наборе и наборе новых данных.

```
# создаем списки переменных для исторического набора
# и набора новых данных
fulldata_cols = fulldata.columns.tolist()
newdata_cols = newdata.columns.tolist()
```

```
# выполняем проверку совпадения порядка столбцов
# в историческом наборе и наборе новых данных
check_vars_order(fulldata_cols, newdata_cols)
```

Одни и те же списки переменных, разный порядок.

Порядок столбцов не совпадает, поэтому приведем порядок столбцов в новых данных к порядку столбцов в исторических данных.

```
# порядок столбцов отличается от порядка
# столбцов в обучающей выборке, приведем
# порядок столбцов в новых данных
# к порядку столбцов в исторических данных
cols_lst = fulldata.columns.tolist()
newdata = newdata[cols_lst]
```

Снова выполняем проверку совпадения порядка столбцов в историческом наборе и наборе новых данных.

```
# создаем списки переменных для исторического набора
# и набора новых данных
fulldata_cols = fulldata.columns.tolist()
newdata_cols = newdata.columns.tolist()
```

```
# выполняем проверку совпадения порядка столбцов
# в историческом наборе и наборе новых данных
check_vars_order(fulldata_cols, newdata_cols)
```

Одни и те же списки переменных, один и тот же порядок.

Взглянем на порядок переменных в наборе новых данных с помощью метода `.info()`.

```
# снова смотрим типы переменных и
# количество непропущенных значений
newdata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400000 entries, 0 to 399999
Data columns (total 48 columns):
#   Column      Non-Null Count  Dtype
---  -
0   bin_0       400000 non-null  object
1   bin_1       400000 non-null  object
2   bin_2       400000 non-null  object
3   bin_3       400000 non-null  object
4   bin_4       400000 non-null  object
5   nom_0       400000 non-null  object
6   nom_1       400000 non-null  object
7   nom_2       400000 non-null  object
8   nom_3       400000 non-null  object
```

```

9  nom_4      400000 non-null object
10 nom_5      400000 non-null object
11 nom_6      400000 non-null object
12 nom_7      400000 non-null object
13 nom_8      400000 non-null object
14 nom_9      400000 non-null object
15 ord_0      400000 non-null object
16 ord_1      400000 non-null object
17 ord_2      400000 non-null object
18 ord_3      400000 non-null object
19 ord_4      400000 non-null object
20 ord_5      400000 non-null object
21 day        400000 non-null object
22 month      400000 non-null object
23 bin_0_isnan 400000 non-null object
24 bin_1_isnan 400000 non-null object
25 bin_2_isnan 400000 non-null object
26 bin_3_isnan 400000 non-null object
27 bin_4_isnan 400000 non-null object
28 day_isnan  400000 non-null object
29 month_isnan 400000 non-null object
30 nom_0_isnan 400000 non-null object
31 nom_1_isnan 400000 non-null object
32 nom_2_isnan 400000 non-null object
33 nom_3_isnan 400000 non-null object
34 nom_4_isnan 400000 non-null object
35 nom_5_isnan 400000 non-null object
36 nom_6_isnan 400000 non-null object
37 nom_7_isnan 400000 non-null object
38 nom_8_isnan 400000 non-null object
39 nom_9_isnan 400000 non-null object
40 ord_0_isnan 400000 non-null object
41 ord_1_isnan 400000 non-null object
42 ord_2_isnan 400000 non-null object
43 ord_3_isnan 400000 non-null object
44 ord_4_isnan 400000 non-null object
45 ord_5_isnan 400000 non-null object
46 ord_5a     400000 non-null object
47 ord_5b     400000 non-null object

```

```
dtypes: object(48)
```

```
memory usage: 146.5+ MB
```

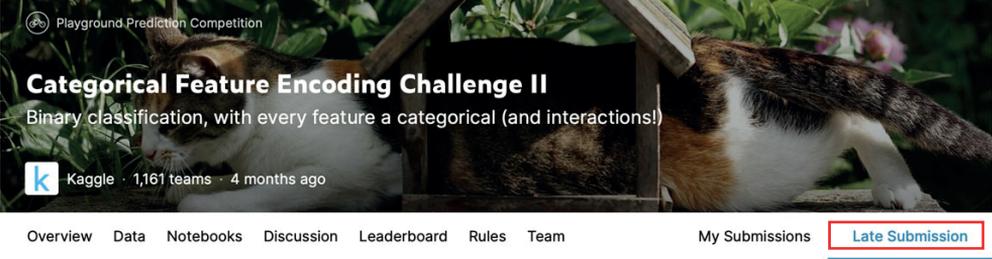
С помощью конвейера, обученного на всей исторической выборке, вычисляем вероятности положительного класса для новых данных и формируем посылку.

```

# вычисляем вероятности с помощью конвейера
proba = ml_pipe.predict_proba(newdata)[: , 1]
# формируем посылку для Kaggle
subm = pd.DataFrame(proba, index=ident, columns=['target'])
subm.to_csv('subm.csv')

```

Заходим на страницу соревнования <https://www.kaggle.com/c/cat-in-the-dat-ii/overview> и выбираем раздел **Late Submission**.



Playground Prediction Competition

## Categorical Feature Encoding Challenge II

Binary classification, with every feature a categorical (and interactions!)

Kaggle · 1,161 teams · 4 months ago

Overview Data Notebooks Discussion Leaderboard Rules Team My Submissions **Late Submission**

Затем надо загрузить посылку, дать ей название и нажать кнопку **Make Submission**.

### Step 1

Upload submission file



#### File Format

Your submission should be in CSV format. You can upload this in a zip/gz/rar/7z archive, if you prefer.

#### Number of Predictions

We expect the solution file to have 400000 prediction rows. This file should have a header row. Please see sample submission file on the [data page](#).

### Step 2

Describe submission



Briefly describe your submission

**Make Submission**

Выполняется процедура оценки и выдается результат.

**Score: 0.78567**

Private score: 0.78719

## 7.18. КАЛИБРОВКА МОДЕЛИ

### 7.18.1. Актуальность калибровки

При выполнении классификации часто требуется не только предсказать класс, но и получить вероятность соответствующего класса. Эта вероятность дает вам некоторую уверенность в прогнозе. Проблема заключается в том, что некоторые модели могут давать плохие оценки вероятностей классов,

а некоторые даже не позволяют вычислить вероятности (например, линейный классификатор опорных векторов, реализованный в классе `LinearSVC`). Модуль калибровки в `skikit-learn` позволяет лучше калибровать вероятности используемой модели.

Хорошо откалиброванные классификаторы – это вероятностные классификаторы, для которых выходные данные метода `.predict_proba()` могут напрямую интерпретироваться как некоторый уровень уверенности. Например, хорошо откалиброванный (бинарный) классификатор должен классифицировать наблюдения так, чтобы среди наблюдений, которым он присвоил вероятность положительного класса, близкую к 0,8, приблизительно 80 % наблюдений действительно принадлежали положительному классу. Некоторые методы машинного обучения, такие как наивный байесовский классификатор, а в последнее время и глубокие нейронные сети, известны своей сверхуверенностью. Несмотря на то что они могут иметь хорошую (или даже превосходную) дискриминирующую способность, полученные вероятности располагаются ближе к 1 и 0, чем следует. Например, среди наблюдений, которым наивный байесовский классификатор присвоил вероятность 0,99, только 75 % могут действительно принадлежать положительному классу.

Случайный лес часто демонстрирует пики в районе вероятностей 0,2 и 0,9, в то время как вероятности, близкие к 0 или 1, очень редки. Объяснение этому дают Niculescu-Mizil и Caruana в своей работе «Predicting Good Probabilities with Supervised Learning»<sup>6</sup>: «Такие методы, как бэггинг и случайные леса, которые усредняют прогнозы по набору базовых моделей, могут иметь трудности при прогнозировании 0 и 1, поскольку дисперсия, лежащая в основе базовых моделей, будет смещать прогнозы, близкие к нулю или единице, относительно этих значений».

Метод опорных векторов для низких значений вероятности дает высокие значения, а для высоких значений вероятности – низкие и поэтому также настоятельно нуждается в калибровке. Собственно, один из методов калибровки (калибровка Платта) появился для того, чтобы откалибровать ответы метода опорных векторов.

Итак, выяснив актуальность процедуры калибровки, для начала импортируем необходимые библиотеки, классы и функции.

```
# импортируем необходимые библиотеки, классы и функции
import numpy as np
import pandas as pd
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import (StandardScaler,
                                   OneHotEncoder)
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.ensemble import (GradientBoostingClassifier,
                              RandomForestClassifier)
from sklearn.linear_model import LogisticRegression
```

<sup>6</sup> <https://www.cs.cornell.edu/~alexn/papers/calibration.icml05.crc.rev3.pdf>.



```
print("фактические вероятности:      ", prob_true)
print("спрогнозированные вероятности:", prob_pred)
```

```
фактические вероятности:      [0.  0.5  1. ]
спрогнозированные вероятности: [0.22666667 0.5375      0.82666667]
```

Видим, что в первом и втором бинах фактическая вероятность ниже спрогнозированной, а в третьем бине фактическая вероятность выше спрогнозированной.

Теперь подробно выясним, как были получены фактические и спрогнозированные вероятности. Для подробного разбора функции `calibration_curve()` мы напишем свой немного упрощенный вариант функции, назвав ее `custom_calibration_curve()`.

```
def custom_calibration_curve(y_true, y_prob, normalize=False, n_bins=5,
                             strategy='uniform'):
    if normalize: # нормализуем значения y_prob в интервал [0, 1]
        y_prob = (y_prob - y_prob.min()) / (y_prob.max() - y_prob.min())
    elif y_prob.min() < 0 or y_prob.max() > 1:
        raise ValueError("y_prob имеет значения вне диапазона [0, 1], а для "
                          "параметра normalize задано значение False.")
    labels = np.unique(y_true)
    if len(labels) > 2:
        raise ValueError("Поддерживается только бинарная классификация. "
                          "Полученные метки %s." % labels)
    if strategy == 'quantile': # определяем границы бинов по распределению
        quantiles = np.linspace(0, 1, n_bins + 1)
        bins = np.percentile(y_prob, quantiles * 100)
        bins[-1] = bins[-1] + 1e-8
    elif strategy == 'uniform':
        bins = np.linspace(0., 1. + 1e-8, n_bins + 1)
    else:
        raise ValueError("Неверное значение для параметра 'strategy'. "
                          "Значение должно быть либо "
                          "'quantile', либо 'uniform'.")
    binids = np.digitize(y_prob, bins) - 1
    bin_sums = np.bincount(binids, weights=y_prob, minlength=len(bins))
    bin_true = np.bincount(binids, weights=y_true, minlength=len(bins))
    bin_total = np.bincount(binids, minlength=len(bins))
    nonzero = bin_total != 0
    prob_true = bin_true[nonzero] / bin_total[nonzero]
    prob_pred = bin_sums[nonzero] / bin_total[nonzero]
    return prob_true, prob_pred
```

Начнем по порядку разбираться, что происходит внутри функции, а заодно посмотрим на некоторые полезные функции NumPy в действии.

Если на вход в качестве вероятностей поданы значения ниже 0 или выше 1 (например, вы передали значения решающей функции), выполняется процедура нормализации, в ходе которой мы из исходного значения вычитаем минимальное значение, а затем полученный результат делим на разницу между максимальным и минимальным значениями. В итоге преобразовываем значения в отрезок [0, 1]. Блок нормализации в следующих версиях библиотеки `scikit-learn` будет удален, и нужно будет явно выполнять нормализацию значений, рассматриваемых в качестве вероятностей.

С помощью функции `np.unique()` создаем массив с уникальными фактическими значениями зависимой переменной – метками классов. Это необходимо для проверки количества уникальных меток. Если количество меток больше 2, выбрасывается ошибка.

Потом запускается схема биннинга. Мы можем формировать бины одинаковой ширины (значение `'uniform'` для параметра `strategy`) или бины с одинаковым количеством наблюдений (значение `'quantile'` для параметра `strategy`). В данном случае будем формировать бины одинаковой ширины.

Давайте фиксируем количество бинов (допустим, у нас будет 3 бина) и с помощью функции `np.linspace()` создаем последовательность, начальное значение – 0, последнее значение – 1 (к нему добавляем очень маленькое положительное значение), количество элементов последовательности будет равно количеству бинов + 1. Элементы последовательности будут равноудалены друг от друга, таким образом, ширина бинов будет одинаковой. По сути, мы получили границы бинов.

```
# фиксируем количество бинов
n_bins = 3
# создаем последовательность, начальное значение - 0,
# последнее значение - 1 (к нему добавляем очень
# маленькое положительное значение), количество
# элементов последовательности равно n_bins + 1
bins = np.linspace(0., 1. + 1e-8, n_bins + 1)
bins

array([0.          , 0.33333334, 0.66666667, 1.00000001])
```

С помощью функции `np.digitize()` возвращаем индексы бинов, в которые входит каждое значение массива `y_prob`, и уменьшаем на 1. Уменьшение на 1 необходимо, потому что `np.digitize()` возвращает целочисленные индексы бинов, начиная с 1. Например, для наибольшего значения `y_prob` получаем индекс 3, уменьшаем его на 1 и получаем 2, а для наименьшего значения `y_prob` получаем индекс 1, уменьшаем его на 1 и получаем 0.

```
# возвращаем индексы бинов, в которые входит каждое
# значение массива y_prob, и уменьшаем на 1, например
# для наименьшего значения y_prob получаем индекс 1,
# уменьшаем его на 1 и получаем 0
binids = np.digitize(y_prob, bins) - 1
binids

array([0, 0, 0, 1, 1, 1, 1, 2, 2, 2])
```

bins	0	0.333	0.666	1						
y_prob	0.15	0.21	0.32	0.45	0.49	0.55	0.66	0.71	0.85	0.92
binids	0	0	0	1	1	1	1	2	2	2

Рис. 51 Получение индексов бинов

Теперь с помощью функции `np.bincount()` вычисляем количество вхождений индексов бинов в массиве `binids`, но поскольку с помощью параметра `weights` заданы весовые коэффициенты (в данном случае ими будут вероятности положительного класса), будет подсчитываться сумма весовых коэффициентов (веро-

ятностей) по каждому индексу, например по индексу 0 эта сумма составит  $0,15 + 0,21 + 0,32 = 0,68$ . С помощью параметра `minlength` фиксируем минимальную длину массива, она определяется здесь количеством границ бинов (в нашем случае – 4 границы, значит, массив будет из 4 элементов), в итоге у нас будет три значения (три суммы) по трем индексам бинов, а четвертое будет нулем. По сути, мы вычислили суммы вероятностей положительного класса в каждом бине.

```
# функция np.bincount() вычисляет количество вхождений индексов
# в массиве binids, но поскольку заданы весовые коэффициенты
# (в данном случае - вероятности), будет подсчитываться сумма
# весовых коэффициентов по каждому индексу, например по индексу 0
# эта сумма составит 0.15 + 0.21 + 0.32 = 0.68, с помощью minlength
# фиксируем минимальную длину массива, она определяется здесь
# количеством границ бинов (в нашем случае - 4 границы, значит, массив
# будет из 4 элементов), в итоге у нас будет три значения по трем
# индексам, а четвертое будет нулем
bin_sums = np.bincount(binids, weights=y_prob, minlength=len(bins))
bin_sums

array([0.68, 2.15, 2.48, 0. ])
```

bins	0			0.333			0.666			1
y_prob	0.15	0.21	0.32	0.45	0.49	0.55	0.66	0.71	0.85	0.92
binids	0	0	0	1	1	1	1	2	2	2
bin_sums	0.68			2.15			2.48			

Рис. 52 Вычисление сумм вероятностей положительного класса в каждом бине

Опять с помощью функции `np.bincount()` вычисляем количество вхождений индексов бинов в массиве `binids`, только теперь весовыми коэффициентами будут фактические значения зависимой переменной, вновь будет подсчитываться сумма весовых коэффициентов по каждому индексу, например по индексу 1 эта сумма составит  $0 + 1 + 1 + 0 = 2$ . Снова с помощью параметра `minlength` фиксируем минимальную длину массива, она определяется здесь количеством границ бинов (в нашем случае – 4 границы, значит, массив будет из 4 элементов), в итоге у нас будет три значения (три суммы) по трем индексам бинов, а четвертое будет нулем. По сути, мы вычислили количество фактических наблюдений положительного класса в каждом бине.

```
# функция np.bincount() вычисляет количество вхождений индексов
# в массиве binids, но поскольку заданы весовые коэффициенты
# (в данном случае - фактические значения зависимой переменной),
# будет подсчитываться сумма весовых коэффициентов по каждому
# индексу, например по индексу 1 эта сумма составит 0 + 1 + 1 + 0 = 2,
# с помощью minlength фиксируем минимальную длину массива, она
# определяется здесь количеством границ бинов (в нашем случае
# - 4 границы, значит, массив будет из 4 элементов), в итоге
# у нас будет три значения по трем индексам, а четвертое будет нулем
bin_true = np.bincount(binids, weights=y_true, minlength=len(bins))
bin_true

array([0., 2., 3., 0.] )
```

bins	0	0.333			0.666			1		
y_prob	0.15	0.21	0.32	0.45	0.49	0.55	0.66	0.71	0.85	0.92
binids	0	0	0	1	1	1	1	2	2	2
y_true	0	0	0	0	1	1	0	1	1	1
bin_true	0			2			3			

Рис. 53 Вычисление количества фактических наблюдений положительного класса в каждом бине

Теперь с помощью функции `np.bincount()` вычисляем количество вхождений индексов бинов в массиве `binids`, но теперь веса не используются. По-прежнему с помощью параметра `minlength` фиксируем минимальную длину массива, она определяется здесь количеством границ бинов (в нашем случае – 4 границы, значит, массив будет из 4 элементов), в итоге у нас будет три значения (три суммы) по трем индексам бинов, а четвертое будет нулем. По сути, мы вычислили общее количество наблюдений в каждом бине.

```
# функция np.bincount() вычисляет количество вхождений индексов
# в массиве binids, веса не используем, с помощью minlength
# фиксируем минимальную длину массива, она определяется
# здесь количеством границ бинов
bin_total = np.bincount(binids, minlength=len(bins))
bin_total
```

bins	0	0.333			0.666			1		
y_prob	0.15	0.21	0.32	0.45	0.49	0.55	0.66	0.71	0.85	0.92
binids	0	0	0	1	1	1	1	2	2	2
y_true	0	0	0	0	1	1	0	1	1	1
bin_total	3			4			3			

Рис. 54 Вычисление общего количества наблюдений в каждом бине

На основе `bin_total` создаем булеву маску, она принимает значение `TRUE` для ненулевого количества наблюдений в бине и `FALSE` – для нулевого количества наблюдений в бине. Она нужна для того, чтобы в дальнейших вычислениях фактических и спрогнозированных вероятностей положительного класса участвовали бины с ненулевым общим количеством наблюдений.

```
# создаем булеву маску, TRUE для ненулевого количества
# наблюдений, FALSE - для нулевого количества наблюдений
nonzero = bin_total != 0
nonzero

array([ True,  True,  True, False])
```

Теперь вычисляем фактическую вероятность положительного класса в каждом бине как количество фактических наблюдений положительного класса в бине, поделенное на общее количество наблюдений в бине, при условии что бин содержит ненулевое общее количество наблюдений.

```
# получаем фактические вероятности
# положительного класса в бинах
prob_true = bin_true[nonzero] / bin_total[nonzero]
prob_true

array([0. , 0.5, 1. ])
```

Наконец, вычисляем спрогнозированную вероятность положительного класса в каждом бине как сумму вероятностей положительного класса в бине, поделенную на общее количество наблюдений в бине, при условии что бин содержит ненулевое общее количество наблюдений.

```
# получаем спрогнозированные вероятности
# положительного класса в бинах
prob_pred = bin_sums[nonzero] / bin_total[nonzero]
prob_pred

array([0.22666667, 0.5375 , 0.82666667])
```

	bins	0	0.333	0.666	1						
y_prob		0.15	0.21	0.32	0.45	0.49	0.55	0.66	0.71	0.85	0.92
bin_ids		0	0	0	1	1	1	1	2	2	2
y_true		0	0	0	0	1	1	0	1	1	1
сумма вероятностей в бине			0.68			2.15			2.48		bin_sums
количество фактических положительных наблюдений в бине			0			2			3		bin_true
общее количество наблюдений в бине			3			4			3		bin_total
фактическая вероятность положительного класса в бине			0/3=0			2/4=0.5			3/3=1		prob_true
спрогнозированная вероятность положительного класса в бине			0.68/3=0.23			2.15/4=0.54			2.48/3=0.83		prob_pred

**Рис. 55** Вычисление фактической вероятности положительного класса и спрогнозированной вероятности положительного класса в каждом бине

Спрогнозированную вероятность положительного класса в бине еще называют средним спрогнозированным значением (`mean_predicted_value`).

Теперь применяем функцию `custom_calibration_curve()` и автоматически вычисляем фактические и спрогнозированные вероятности положительного класса в каждом бине.

```
# применяем нашу функцию
prob_true, prob_pred = custom_calibration_curve(y_true,
                                              y_prob,
                                              n_bins=3,
                                              strategy='uniform')

print("фактические вероятности:      ", prob_true)
print("спрогнозированные вероятности:", prob_pred)

фактические вероятности:      [0. 0.5 1. ]
спрогнозированные вероятности: [0.22666667 0.5375 0.82666667]
```

Видим, что результаты, возвращенные функцией `custom_calibration_curve()`, совпадают с результатами, вычисленными вручную, и результатами функции `calibration_curve()`.

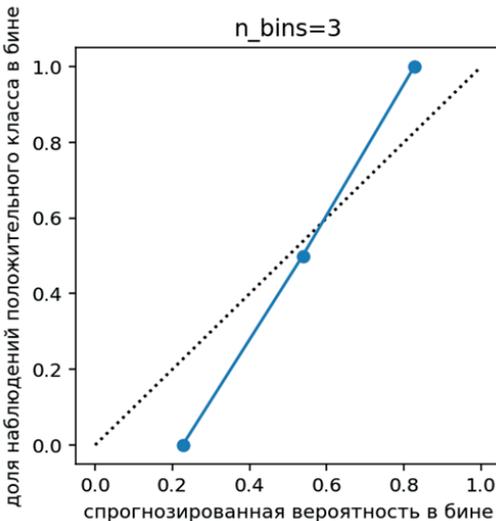
Теперь пишем функцию `plot_calibration_curve()`, строящую график калибровки. По сути, мы сейчас визуализируем результаты, полученные с помощью функции `custom_calibration_curve()`.

```
# пишем функцию, строящую график калибровки
def plot_calibration_curve(y_true, y_prob, n_bins=3):
    # вычисляем фактические и спрогнозированные
    # вероятности положительного класса
    # с помощью функции calibration_curve()
    prob_true, prob_pred = calibration_curve(
        y_true, y_prob, n_bins=n_bins)
    # метод .gca() возвращает текущие области рисования
    # для текущего изображения (gca - это сокращение
    # от get current axes)
    ax = plt.gca()
    # строим график идеальной калибровки (диагональ)
    ax.plot([0, 1], [0, 1], ':', c='k')
    # строим кривую
    curve = ax.plot(prob_pred, prob_true, marker="o")

    # задаем заголовки осей
    ax.set_xlabel("спрогнозированная вероятность в бине")
    ax.set_ylabel("доля наблюдений положительного класса в бине")
    # задаем укладку
    ax.set(aspect='equal')
    return curve
```

Давайте применим нашу функцию `plot_calibration_curve()`.

```
# строим график калибровки
plot_calibration_curve(y_true, y_prob)
plt.title("n_bins=3")
plt.tight_layout()
```



По оси абсцисс откладывается спрогнозированная вероятность положительного класса в бине, а по оси ординат – доля фактических наблюдений положительного класса в бине (фактическая вероятность положительного класса в бине). В идеале они должны совпадать. Логика идеальной калибровки следующая: когда вы уверены на 20 %, вы на 20 % правы (например, у вас в бине 100 наблюдений, средняя спрогнозированная вероятность составила 0,2, и из 100 наблюдений 20 оказались наблюдениями положительного класса).

Диагональ соответствует идеальной калибровке. Если точка кривой лежит над диагональю, речь идет о недопрогнозе, наши спрогнозированные вероятности являются слишком маленькими. Если точка кривой лежит под диагональю, речь идет о перепрогнозе, наши спрогнозированные вероятности являются слишком большими.

Возьмем первый бин от 0 до 0,333. Спрогнозированная вероятность равна 0,23, а фактическая вероятность равна 0. Точка лежит под диагональю. Наши вероятности 0,15, 0,21, 0,32 являются слишком большими. У нас – перепрогноз. Например, если бы наши вероятности были бы меньше, допустим 0,01, 0,02, 0,03, то спрогнозированная вероятность равнялась бы  $(0,01 + 0,02 + 0,03)/3 = 0,02$ , и это было бы гораздо ближе к фактической вероятности 0.

Возьмем третий бин от 0,666 до 1. Спрогнозированная вероятность равна 0,83, а фактическая вероятность равна 1. Точка лежит над диагональю. Наши вероятности 0,71, 0,85, 0,92 являются слишком маленькими. У нас – недопрогноз. Например, если бы наши вероятности были бы больше, допустим 0,91, 0,92, 0,95, то спрогнозированная вероятность равнялась бы  $(0,91 + 0,92 + 0,95)/3 = 0,93$ , и это было бы гораздо ближе к фактической вероятности 1. Таким образом, калибровка будет заключаться в такой деформации вероятностей, чтобы спрогнозированная вероятность положительного класса совпадала с фактической.

### 7.18.3. Оценка Брайера

Для оценки качества калибровки вероятностей используется оценка Брайера (Бриера) – среднеквадратичная разница между спрогнозированной вероятностью и фактическим прогнозом. Оценка Брайера была предложена Гленном Брайером в 1950 году. Чаще всего она вычисляется по формуле

$$Brier = \frac{1}{N} \sum_{i=1}^N (p_i - a_i)^2,$$

где  $p_i$  – спрогнозированная вероятность для  $i$ -го наблюдения;  
 $a_i$  – фактический прогноз для  $i$ -го наблюдения (измеряемый как 0 или 1).

Следовательно, чем ниже оценка Брайера, тем лучше откалиброваны вероятности. Обратите внимание, что оценка Брайера, рассчитываемая по формуле выше, принимает значение от нуля до единицы, единица указывает на наибольшую возможную разницу между спрогнозированной вероятностью (которая должна быть между 0 и 1) и фактическим прогнозом (который может принимать значения только 0 или 1). В оригинальной формулировке оценки Брайера 1950 года диапазон удваивается, от 0 до 2.

Предположим, вы прогнозируете вероятность дождя на завтра.

Если вероятность составляет 100 % ( $P = 1$ ) и идет дождь (1), то оценка Брайера составляет 0, получаем наилучшую, т. е. наименьшую, оценку Брайера.

Если вероятность составляет 100 % ( $P = 1$ ) и дождя нет (0), то оценка Брайера составляет 1, получаем наихудшую, т.е. наибольшую, оценку Брайера.

Если вероятность составляет 70 % ( $P = 0,7$ ) и идет дождь (1), то оценка Брайера составляет  $(0,70 - 1)^2 = 0,09$ .

Если вероятность составляет 30 % ( $P = 0,3$ ) и идет дождь (1), то оценка Брайера составляет  $(0,30 - 1)^2 = 0,49$ .

Если вероятность составляет 50 % ( $P = 0,5$ ), то оценка Брайера составляет  $(0,50 - 1)^2 = (1 - 0,50)^2 = 0,25$  независимо от того, идет дождь или нет.

Давайте с помощью функции `brier_score_loss()` и потом вручную вычислим оценку Брайера для нашего примера.

```
# вычислим оценку Брайера с помощью функции brier_score_loss()
```

```
brier_score = brier_score_loss(y_true, y_prob)
```

```
brier_score
```

```
0.13827
```

```
# вычислим оценку Брайера вручную
```

```
manual_brier_score = (sum(np.square(y_true - y_prob))) / len(y_true)
```

```
manual_brier_score
```

```
0.13827
```

Хотя вышеприведенная формула является наиболее широко используемой, первоначальная формула оценки Брайера предназначалась для многоклассовых прогнозов (например, предсказываем холодно/нормально/жарко):

$$Brier = \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^R (p_{it} - a_{it}),$$

где  $R$  – количество возможных классов, к которым можно отнести наблюдение;  $N$  – общее количество наблюдений всех классов.

Оценка Брайера становится неадекватной для очень редких (или очень часто встречающихся) событий, поскольку недостаточно чувствительно дискриминирует небольшие изменения, которые важны при работе с редкими событиями. Уилкс в своей работе отмечает, что «для высокоточных прогнозов касательно редких событий требуются достаточно большие размеры выборки, т.е.  $n > 1000$ , в то время как для несложных прогнозов обычных событий требуются лишь довольно скромные размеры выборок»<sup>7</sup>. Кроме того, оценка Брайера не учитывает разницу между вероятностями в разных частях диапазона значений вероятности. Интуитивно мы понимаем, что при прогнозировании дефолта разница между вероятностями 0 % и 10 % сильно отличается от разницы между вероятностями 40 % и 50 % (когда мы приближаемся к возможному порогу принятия решений), однако для оценки Брайера эта разница является одинаковой.

Отметим, что помимо оценки Брайера в качестве метрики качества калибровки модели используется также логистическая функция потерь (подробнее о ней читайте во втором томе).

<sup>7</sup> Wilks D. S. (2010). Sampling distributions of the Brier score and Brier skill score under serial dependence. Quarterly Journal of the Royal Meteorological Society. 136 (1): 2109–2118.

## 7.18.4. Оценка качества калибровки моделей до применения калибратора

Сейчас мы сравним качество калибровки вероятности положительного класса для трех различных классификаторов: логистической регрессии, градиентного бустинга и случайного леса.

Данные записаны в файле *Response.csv*. Исходная выборка содержит записи о 30 259 клиентах, классифицированных на два класса: 0 – отклика нет (17 170 клиентов) и 1 – отклик есть (13 089 клиентов). По каждому наблюдению (клиенту) фиксируются следующие переменные (характеристики):

- категориальный признак *Ипотечный кредит* [*mortgage*];
- категориальный признак *Страхование жизни* [*life\_ins*];
- категориальный признак *Кредитная карта* [*cre\_card*];
- категориальный признак *Дебетовая карта* [*deb\_card*];
- категориальный признак *Мобильный банк* [*mob\_bank*];
- категориальный признак *Текущий счет* [*curr\_acc*];
- категориальный признак *Интернет-доступ к счету* [*internet*];
- категориальный признак *Индивидуальный займ* [*perloan*];
- категориальный признак *Наличие сбережений* [*savings*];
- категориальный признак *Пользование банкоматом за последнюю неделю* [*atm\_user*];
- категориальный признак *Пользование услугами онлайн-маркетплейса за последний месяц* [*markpl*];
- количественный признак *Возраст* [*age*];
- количественный признак *Давность клиентской истории* [*cus\_leng*];
- категориальная зависимая переменная *Отклик на предложение новой карты* [*response*].

*# загружаем данные*

```
data = pd.read_csv('Data/Response.csv', sep=';')
data.head(3)
```

	mortgage	life_ins	cre_card	deb_card	mob_bank	curr_acc	internet	perloan	savings	atm_user	markpl	age	cus_leng	response
0	No	No	No	No	No	18.0	less than 3 years	No						
1	Yes	Yes	NaN	NaN	Yes	No	NaN	NaN	NaN	Yes	No	18.0	NaN	Yes
2	Yes	Yes	NaN	Yes	No	No	No	No	No	No	Yes	NaN	from 3 to 7 years	Yes

Теперь меняем строковые метки зависимой переменной на целочисленные.

*# меняем строковые метки на целочисленные*

```
data['response'] = np.where(data['response'] == 'Yes', 1, 0)
```

Разбиваем данные на обучающую и тестовую выборки.

*# разбиваем данные на обучающую и тестовую выборки*

```
X_tr, X_tst, y_tr, y_tst = train_test_split(
    data.drop('response', axis=1),
    data['response'],
    test_size=0.3,
    stratify=data['response'],
    random_state=42)
```

Создаем конвейеры для соответствующего классификатора.

```
# создаем списки категориальных
# и количественных столбцов
cat_cols = X_tr.select_dtypes(
    include='object').columns.tolist()
num_cols = X_tr.select_dtypes(
    exclude='object').columns.tolist()

# создаем конвейер для количественных переменных
# (для применения с логистической регрессией)
logreg_num_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# создаем конвейер для количественных переменных
# (для применения с ансамблями на основе деревьев)
tree_num_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='constant',
        fill_value=-999))
])

# создаем конвейер для категориальных переменных
cat_pipe = Pipeline([
    ('imp', SimpleImputer(strategy='most_frequent')),
    ('ohe', OneHotEncoder(sparse=False,
        handle_unknown='ignore'))
])

# создаем список трансформеров для логистической регрессии
logreg_transformers = [('num', logreg_num_pipe, num_cols),
    ('cat', cat_pipe, cat_cols)]
logreg_transformer = ColumnTransformer(
    transformers=logreg_transformers)

# создаем список трансформеров для ансамблей на основе деревьев
tree_transformers = [('num', tree_num_pipe, num_cols),
    ('cat', cat_pipe, cat_cols)]
tree_transformer = ColumnTransformer(
    transformers=tree_transformers)

# создаем итоговый конвейер для логистической регрессии
lr_pipe = Pipeline([
    ('logreg_tf', logreg_transformer),
    ('lr', LogisticRegression(solver='liblinear'))
])

# создаем итоговый конвейер для градиентного бустинга
gb_pipe = Pipeline([
    ('tree_tf', tree_transformer),
    ('gb', GradientBoostingClassifier(n_estimators=100,
        random_state=42))
])
```

```

# создаем итоговый конвейер для случайного леса
rf_pipe = Pipeline([
    ('tree_tf', tree_transformer),
    ('rf', RandomForestClassifier(n_estimators=100,
                                random_state=42))
])

# создаем итоговый конвейер для линейного SVM
svc_pipe = Pipeline([
    ('logreg_tf', logreg_transformer),
    ('svc', LinearSVC())
])

```

Теперь пишем функцию для построения графика калибровки и гистограммы распределения средних спрогнозированных значений по бинам.

```

# пишем функцию для построения графика калибровки и
# гистограммы распределения средних спрогнозированных
# значений по бинам
def extended_calibration_plot(X_train, X_test, y_train, y_test):
    # создаем таблицу графиков
    plt.figure(figsize=(9, 9))
    ax1 = plt.subplot2grid((3, 1), (0, 0), rowspan=2)
    ax1.grid()
    ax2 = plt.subplot2grid((3, 1), (2, 0))
    ax2.grid()
    ax1.plot([0, 1], [0, 1], "k:", label="Идеальная калибровка")

    # обучаем конвейеры
    for pipe, name in [(lr_pipe, "Логистическая регрессия"),
                       (gb_pipe, "Обычный градиентный бустинг"),
                       (rf_pipe, "Случайный лес"),
                       (svc_pipe, "Линейный SVM")]:
        pipe.fit(X_train, y_train)

        # если объект имеет атрибут 'predict_proba'
        if hasattr(pipe, 'predict_proba'):
            # записываем вероятности положительного класса
            prob_pos = pipe.predict_proba(X_test)[:, 1]
            # если объект не имеет атрибут 'predict_proba'
        else:
            # используем решающую функцию
            prob_pos = pipe.decision_function(X_test)
            prob_pos = ((prob_pos - prob_pos.min()) /
                        (prob_pos.max() - prob_pos.min()))

        # получаем фактические и спрогнозированные вероятности
        fraction_of_positives, mean_predicted_value = calibration_curve(
            y_test, prob_pos, n_bins=10)
        # строим график калибровки
        ax1.plot(mean_predicted_value, fraction_of_positives, "s-",
                label="%s" % (name, ))
        # строим гистограмму
        ax2.hist(prob_pos, range=(0, 1), bins=10, label=name,
                histtype='step', lw=2)

```

```

# вычисляем оценку Брайера, AUC и логистическую функцию потерь
brier_score = brier_score_loss(y_test, prob_pos)
auc_score = roc_auc_score(y_test, prob_pos)
log_loss_score = log_loss(y_test, prob_pos)
# печатаем название метода, оценку Брайера, AUC
# и логистическую функцию потерь
print("%s:" % name)
print("\tОценка Брайера: %1.3f" % (brier_score))
print("\tAUC: %1.3f" % (auc_score))
print("\tЛогистическая функция потерь: %1.3f" % (log_loss_score))

# задаем заголовок оси y
ax1.set_ylabel("Доля наблюдений положительного класса")
# задаем диапазон значений оси y
ax1.set_ylim([-0.05, 1.05])
# задаем расположение легенды
ax1.legend(loc="lower right")
# задаем заголовок графика
ax1.set_title("График калибровки (кривая надежности) и гистограмма")
# задаем заголовок оси x
ax2.set_xlabel("Усредненное спрогнозированное значение")
# задаем заголовок оси y
ax2.set_ylabel("Частота")
# задаем расположение легенды
ax2.legend(loc="upper center", ncol=2)
# выполняем укладку
plt.tight_layout()

```

Применяем нашу функцию.

```

# применяем нашу функцию
extended_calibration_plot(X_tr, X_tst, y_tr, y_tst)

```

Логистическая регрессия:

```

Оценка Брайера: 0.123
AUC: 0.907
Логистическая функция потерь: 0.380

```

Обычный градиентный бустинг:

```

Оценка Брайера: 0.120
AUC: 0.911
Логистическая функция потерь: 0.373

```

Случайный лес:

```

Оценка Брайера: 0.152
AUC: 0.873
Логистическая функция потерь: 0.874

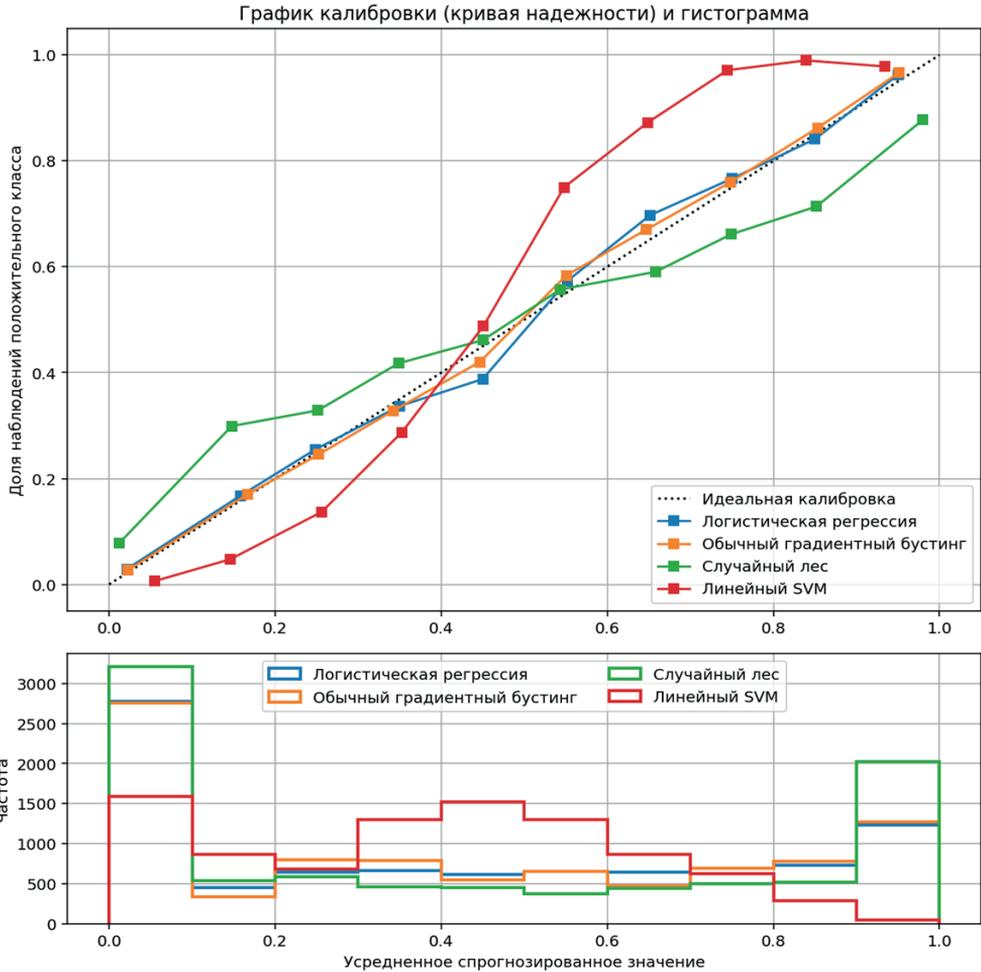
```

Линейный SVM:

```

Оценка Брайера: 0.142
AUC: 0.907
Логистическая функция потерь: 0.441

```



В выводе видно, что хуже всего откалиброваны вероятности случайного леса и линейного SVM. Случайный лес и линейный SVM имеют наибольшие оценки Брайера и логистической функции потерь. Кривая калибровки для SVM имеет характерный вид сигмоиды. На практике среди логистической регрессии, случайного леса и градиентного бустинга именно случайный лес и метод опорных векторов будут больше всего нуждаться в калибровке. Для градиентного бустинга калибровка будет в большей степени необходима, когда для задачи классификации вместо минимизации логистической функции потерь будет использоваться минимизация квадратичной функции потерь (здесь можно привести в качестве примера кейс компании H2O для PayPal: использование калибровки для квадратичной функции потерь позволило уменьшить оценку Брайера и логистическую функцию потерь при построении моделей градиентного бустинга).

## 7.18.5. Класс CalibratedClassifierCV

Давайте попробуем откалибровать вероятности классификаторов с помощью класса `CalibratedClassifierCV` (для краткости мы иногда будем называть его калибратором). Класс `CalibratedClassifierCV` осуществляет калибровку вероятностей с помощью параметрического подхода, основанного на сигмоидной модели Платта, и непараметрического подхода, основанного на изотонической регрессии.

Класс `CalibratedClassifierCV` запускает на обучающей выборке перекрестную проверку, на каждой итерации он строит модель на обучающих наблюдениях и калибрует на отложенных наблюдениях. Уже обученные классификаторы могут быть откалиброваны с помощью параметра `cv="prefit"`. В этом случае пользователь должен вручную позаботиться о том, чтобы данные для обучения модели и калибровки не пересекались.

`CalibratedClassifierCV` также может работать с несколькими классами, если классификатор, вероятности которого должны быть откалиброваны (`base_estimator`), поддерживает многоклассовую классификацию. В этом случае классификатор сначала калибруется для каждого класса отдельно по схеме «один против остальных». Когда нужно спрогнозировать вероятности для новых данных, для каждого класса отдельно прогнозируются откалиброванные вероятности. Поскольку эти вероятности не обязательно должны быть равны единице, для их нормализации выполняется постобработка. Отметим, что калибровку вероятностей нужно рассматривать как настройку гиперпараметра (могут быть разные способы калибровки, у сложных способов калибровки могут быть свои настройки), и поэтому ее необходимо выполнять на новых данных, не используемых для обучения модели, что, собственно, и делает класс `CalibratedClassifierCV`. Ниже приведены параметры класса `CalibratedClassifierCV`.

```
from sklearn.calibration import CalibratedClassifierCV(base_estimator=None,
method='sigmoid',
'isotonic'
cv=None)
```

Задаёт метод калибровки: сигмоидную калибровку Платта или изотоническую калибровку. Не рекомендуется использовать изотоническую калибровку на небольшом количестве наблюдений (< 1000), так как она имеет тенденцию к переобучению. В этом случае используйте сигмоидную калибровку Платта

Задаёт классификатор, вероятности которого должны быть откалиброваны. Если для параметра `cv` задано значение `'prefit'`, классификатор уже должен быть обучен

Задаёт стратегию перекрестной проверки

Теперь разберем методы калибровки, которые предлагает класс `CalibratedClassifierCV`.

Калибровка Платта изначально предназначалась для преобразования ответов алгоритма опорных векторов (SVM) в вероятности, пропустив первые через сигмоиду. Обозначим ответы алгоритма как  $f(x)$  и пропускаем их через сигмоиду:

$$P(y = 1|f) = \frac{1}{1 + \exp(Af + B)},$$

где  $A$  и  $B$  – неизвестные параметры, которые определяются методом максимального правдоподобия на отложенной выборке.

Градиентный спуск применяется для поиска таких значений, чтобы решение удовлетворяло условию:

$$\operatorname{argmin}_{A,B} \left( -\sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \right),$$

где  $p_i = \frac{1}{1 + \exp(Af_i + B)}$ .

Альтернативой калибровки Платта является изотоническая регрессия. Изотоническая регрессия выступает более общим методом, в отличие от калибровки Платта, поскольку лишь предполагает, что  $y_i = m(f_i) + \varepsilon_i$ , где  $m$  – это изотоническая (монотонно возрастающая) функция. На отложенной выборке мы ищем такую кусочно-постоянную функцию  $\hat{m}$ , которая удовлетворяет условию  $\hat{m} = \underset{z}{\operatorname{argmin}} \sum (y_i - z(f_i))^2$  (речь идет о минимизации среднеквадратичной

ошибки между спрогнозированными вероятностями положительного класса и фактическими значениями зависимой переменной).

Сигмоидная калибровка Платта предполагает, что калибровочная кривая имеет вид сигмоиды, тогда как изотоническая калибровка может справиться с любым видом калибровочной кривой при условии достаточного объема данных для калибровки.

### 7.18.6. Оценка качества калибровки моделей после применения калибратора

Давайте для каждого метода создадим конвейер, в который передадим экземпляр класса `CalibratedClassifierCV`, а ему передадим соответственно экземпляр класса – модель машинного обучения.

```
# создаем итоговый конвейер для логистической регрессии
lr_pipe = Pipeline([
    ('logreg_tf', logreg_transformer),
    ('lr', CalibratedClassifierCV(
        LogisticRegression(solver='liblinear'), cv=10))
])

# создаем итоговый конвейер для градиентного бустинга
gb_pipe = Pipeline([
    ('tree_tf', tree_transformer),
    ('gb', CalibratedClassifierCV(GradientBoostingClassifier(
        n_estimators=100, random_state=42), cv=10))
])

# создаем итоговый конвейер для случайного леса
rf_pipe = Pipeline([
    ('tree_tf', tree_transformer),
    ('rf', CalibratedClassifierCV(RandomForestClassifier(
        n_estimators=100, random_state=42), cv=10))
])

# создаем итоговый конвейер для линейного SVM
svc_pipe = Pipeline([
    ('logreg_tf', logreg_transformer),
    ('svc', CalibratedClassifierCV(LinearSVC(), cv=10))
])
```

Вновь строим график калибровки и гистограмму распределения средних спрогнозированных значений по бинам.

# вновь строим график калибровки и гистограмму распределения  
 # средних спрогнозированных значений по бинам  
 extended\_calibration\_plot(X\_tr, X\_tst, y\_tr, y\_tst)

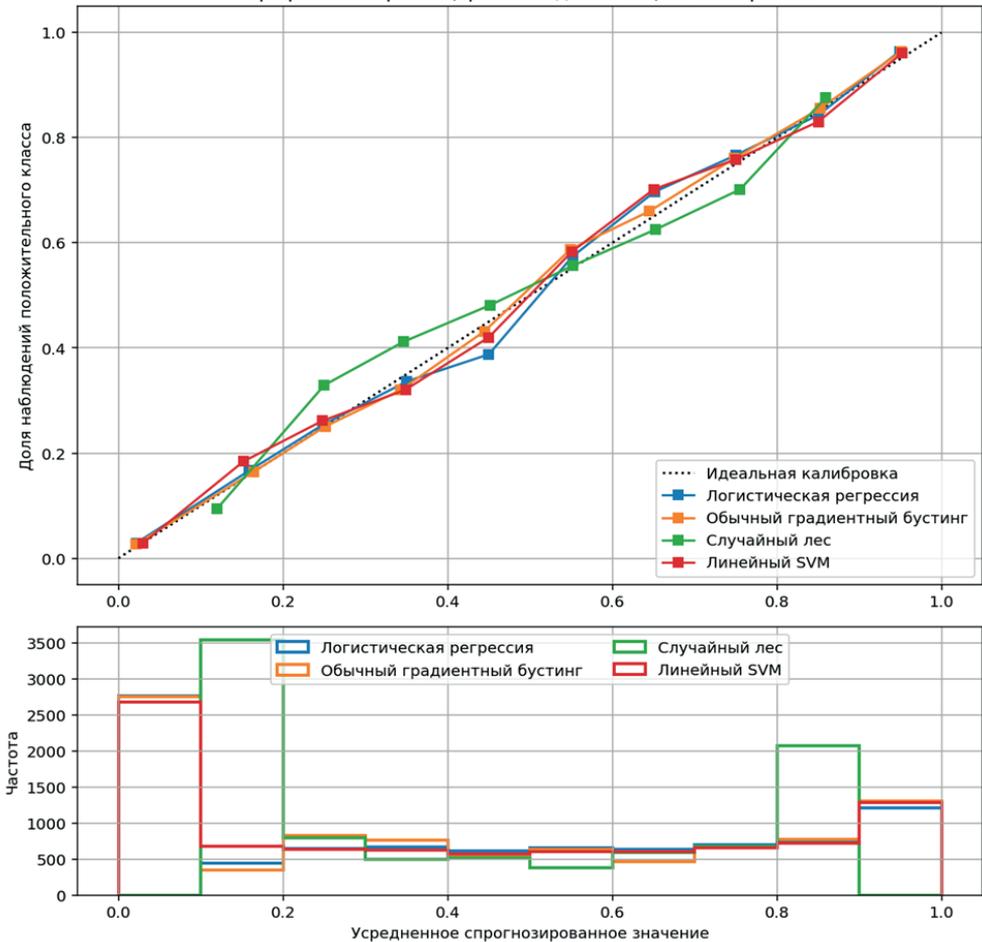
Логистическая регрессия:  
 Оценка Брайера: 0.123  
 AUC: 0.907  
 Логистическая функция потерь: 0.380

Обычный градиентный бустинг:  
 Оценка Брайера: 0.120  
 AUC: 0.911  
 Логистическая функция потерь: 0.372

Случайный лес:  
 Оценка Брайера: 0.145  
 AUC: 0.878  
 Логистическая функция потерь: 0.451

Линейный SVM:  
 Оценка Брайера: 0.123  
 AUC: 0.907  
 Логистическая функция потерь: 0.381

График калибровки (кривая надежности) и гистограмма



Для случайного леса и метода опорных векторов калибровка позволила улучшить оценки Брайера и логистической функции потерь. При этом необходимо следить, чтобы в результате калибровки оценка AUC не уменьшалась. В ряде случаев после калибровки оценка AUC может измениться (строго говоря, функции калибровки монотонны и не влияют на порядок ранжирования, от которого зависит AUC, однако бывают ситуации, когда после калибровки наблюдения получают одну и ту же вероятность, тогда их сортировка зависит от начального положения в наборе и из-за этого AUC может меняться), здесь важную роль будет играть размер калибровочного набора.

### 7.18.7. Оценка качества калибровки моделей после применения калибратора с уже обученным классификатором

Попробуем еще один способ калибровки. Разобьем данные на обучающую, проверочную (калибровочную) и тестовую выборки. Подготовим данные. На обучающей выборке обучим модель случайного леса, получим вероятности для тестовой выборки, построим график калибровки для них и вычислим оценку Брайера, логистическую функцию потерь и AUC-ROC.

```
# разбиваем обучающую выборку на обучающую и проверочную
```

```
X_tr_sub, X_val, y_tr_sub, y_val = train_test_split(
    X_tr, y_tr,
    stratify=y_tr,
    random_state=42)
```

```
# печатаем информацию о пропусках
```

```
print(X_tr_sub.isnull().sum().sum())
print(X_val.isnull().sum().sum())
print(X_tst.isnull().sum().sum())
```

```
61
36
122
```

```
# печатаем размеры выборок
```

```
print(X_tr_sub.shape)
print(X_val.shape)
print(X_tst.shape)
```

```
(15885, 13)
(5296, 13)
(9078, 13)
```

```
# выполняем импутацию переменных
```

```
imp = SimpleImputer(strategy='mean')
imp.fit(X_tr_sub[num_cols])
X_tr_sub[num_cols] = imp.transform(X_tr_sub[num_cols])
X_val[num_cols] = imp.transform(X_val[num_cols])
X_tst[num_cols] = imp.transform(X_tst[num_cols])
```

```
imp2 = SimpleImputer(strategy='most_frequent')
```

```
imp2.fit(X_tr[cat_cols])
X_tr_sub[cat_cols] = imp2.transform(X_tr_sub[cat_cols])
```

```
X_val[cat_cols] = imp2.transform(X_val[cat_cols])
X_tst[cat_cols] = imp2.transform(X_tst[cat_cols])
```

```
# выполняем дамми-кодирование
```

```
X_tr_sub = pd.get_dummies(X_tr_sub)
X_val = pd.get_dummies(X_val)
X_tst = pd.get_dummies(X_tst)
```

```
# печатаем размеры выборок
```

```
print(X_tr_sub.shape)
print(X_val.shape)
print(X_tst.shape)
```

```
(15885, 26)
```

```
(5296, 26)
```

```
(9078, 26)
```

```
# обучаем случайный лес на обучающей выборке
```

```
rf = RandomForestClassifier(n_estimators=100, random_state=42).fit(
    X_tr_sub, y_tr_sub)
```

```
# вычисляем вероятности для тестовой выборки
```

```
scores = rf.predict_proba(X_tst)[:, 1]
```

```
# построим график калибровки для вероятностей на
```

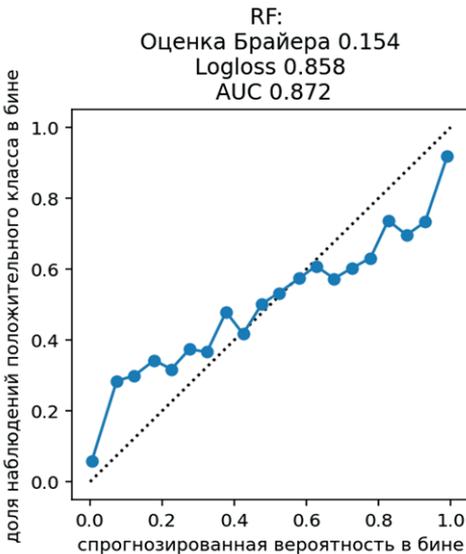
```
# тестовой выборке и вычисляем логистическую
```

```
# функцию потерь, оценку Брайера и AUC
```

```
plot_calibration_curve(y_tst, scores, n_bins=20)
```

```
string = "{}: \n Оценка Брайера {:.3f} \n Logloss {:.3f} \n AUC {:.3f}"
```

```
plot = plt.title(string.format("RF",
                               brier_score_loss(y_tst, scores),
                               log_loss(y_tst, scores),
                               roc_auc_score(y_tst, scores)))
```

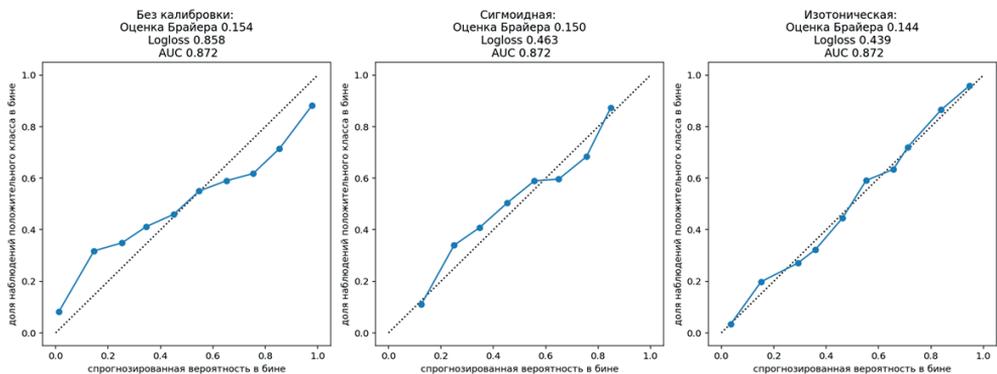


Затем обучаем разные калибраторы с уже обученной моделью случайного леса на калибровочной выборке. Таким образом, пробуем разные способы калибровки на калибровочной выборке, которая не использовалась для обучения модели случайного леса, и получаем откалиброванные вероятности для тестовой выборки.

```
# создаем экземпляр класса CalibratedClassifierCV, передав уже
# обученную модель случайного леса и применив сигмоидную калибровку
cal_rf_sigm = CalibratedClassifierCV(rf, cv='prefit', method='sigmoid')
# обучаем калибратор на калибровочной выборке
cal_rf_sigm.fit(X_val, y_val)
# вычисляем с помощью калибратора вероятности на тестовой выборке
scores_sigm = cal_rf_sigm.predict_proba(X_tst)[: , 1]
# создаем экземпляр класса CalibratedClassifierCV, передав уже
# обученную модель случайного леса и применив изотоническую калибровку
cal_rf_iso = CalibratedClassifierCV(rf, cv='prefit', method='isotonic')
# обучаем калибратор на калибровочной выборке
cal_rf_iso.fit(X_val, y_val)
# вычисляем с помощью калибратора вероятности на тестовой выборке
scores_iso = cal_rf_iso.predict_proba(X_tst)[: , 1]
```

Наконец, выводим графики калибровки и оценки метрик для случая без калибровки, калибровки Платта и изотонической регрессии. Здесь нам понадобятся неоткалиброванные вероятности для тестовой выборки и вероятности для тестовой выборки, откалиброванные с помощью калибровки Платта и изотонической регрессии.

```
# выводим графики калибровки и оценки метрик для случая без
# калибровки, калибровки Платта и изотонической регрессии
fig, axes = plt.subplots(1, 3, figsize=(15, 15))
for name, s, ax in zip(["Нет калибровки", "Сигмоидная", "Изотоническая"],
                      [scores, scores_sigm, scores_iso], axes):
    plot_calibration_curve(y_tst, s, n_bins=10, ax=ax)
    ax.set_title(string.format(name,
                              brier_score_loss(y_tst, s),
                              log_loss(y_tst, s),
                              roc_auc_score(y_tst, s)))
plt.tight_layout()
```



Меньшую дискриминирующую способность модели случайного леса можно объяснить тем, что она была обучена на меньшем количестве наблюдений, в отличие от подхода, основанного на перекрестной проверке, нам пришлось отдельно сформировать обучающую, калибровочную и тестовую выборки. В рамках же перекрестной проверки мы эффективно используем все данные как для обучения модели, так и для калибровки. Однако здесь нужно сделать предостережение, которое состоит в том, что оценки, используемые для калибровки, приходят из слегка различных моделей, каждая из которых, в свою очередь, будет немного отличаться от окончательной модели, к которой калибровка применяется на практике. Эти различия можно уменьшить, выбрав большее число блоков и тем самым увеличив время, необходимое для вычисления. Для линейных моделей достаточно пятиблочной перекрестной проверки, для случайного леса и градиентного бустинга рекомендуется использовать 10-блочную перекрестную проверку.

### 7.18.8. Калибровка на основе сплайнов

Для получения калибровочной кривой можно использовать не только сигмоиду или кусочно-постоянную функцию, но и сплайны. Сплайн – функция, область определения которой разбита на конечное число отрезков, на каждом из которых она совпадает с некоторым алгебраическим многочленом (полиномом). Максимальная из степеней использованных полиномов называется степенью сплайна. Например, непрерывная ломаная – это сплайн степени 1.

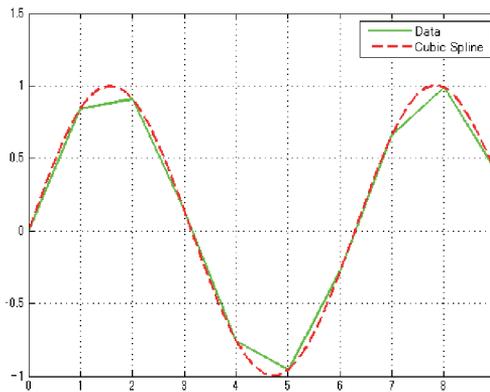


Рис. 56 Пример применения кубического сплайна

Общераспространенным сплайном является натуральный кубический сплайн – сплайн степени 3.

Слово «натуральный» означает, что вторые производные полиномов сплайна установлены равными нулю в конечных точках интервала интерполяции. Кусочные сплайны из многочленов невысокого порядка очень удобны для интерполяции, так как они не требуют больших вычислительных затрат и не вызывают численных отклонений, свойственных многочленам высокого порядка. Кубический сплайн удобен еще и тем, что это кривая наименьшего порядка, допускающая точки перегиба и изгиб в пространстве.

Итак, переходим к использованию сплайна. Рассмотрим предсказание одномерного ответа  $y$  от одномерного признака  $x$ . В сплайновой регрессии при подгонке гладкой кривой к нашим данным мы стремимся найти некоторую функцию  $f(x)$ , которая хорошо описывает эти данные, т. е. мы хотим получить наименьшее значение  $RSS = \sum_{i=1}^n (y_i - f(x_i))^2$  на тестовой выборке. Однако у этого

подхода есть одна проблема. Не накладывая никаких ограничений на  $f(x)$ , мы всегда можем сделать  $RSS$  равной нулю, просто выбрав функцию  $f$ , которая интерполирует все значения  $y_i$ .

Чем больше узлов мы используем, тем лучше интерполируем обучающие данные, тем выше риск переобучения. В случае переобучения функция оказалась бы чрезмерно гибкой. На самом же деле мы хотим, чтобы кривая хорошо интерполировала обучающие данные (с точки зрения  $RSS$  на тестовой выборке), но была не до такой степени сложной, чтобы у нас получилась очень волнистая, нереалистичная кривая с низкой обобщающей способностью.

Решением, позволяющим обеспечить гладкость, является нахождение такой функции  $f$ , которая минимизирует

$$\sum_{i=1}^n (y_i - f(x_i))^2 + \lambda \int \{f''(t)\}^2 dt \quad (1)$$

для заданного набора  $(x_i, y_i)$ .

В этой формуле  $\lambda$  – это некоторый неотрицательный гиперпараметр. Функция  $f$ , минимизирующая выражение выше, известна как сглаживающий сплайн. Сглаживающий сплайн позволяет избежать проблемы выбора узлов, используя все значения  $x$  в качестве узлов и регуляризацию с помощью наложения штрафа на проинтегрированную вторую производную.

Кроме того, помимо критерия наименьших квадратов, в большинстве случаев достаточно просто применить логистическую функцию потерь. Находим такую функцию  $f$ , которая минимизирует

$$-\sum_{i=1}^n [y_i \log(f(x_i)) + (1 - y_i) \log(1 - f(x_i))] + \frac{1}{2} \lambda \int \{f''(t)\}^2 dt \quad (2)$$

для заданного набора. Этот подход еще называют *непараметрической логистической регрессией*.

Теперь подробнее разберем формулу (2).

Член  $-\sum_{i=1}^n [y_i \log(f(x_i)) + (1 - y_i) \log(1 - f(x_i))]$  представляет собой логистическую функцию потерь. Член  $\frac{1}{2} \lambda \int \{f''(t)\}^2 dt$  – это штрафное слагаемое, которое ограничивает варибельность  $f$ . Нотация  $f''(t)$  обозначает вторую производную функции  $f$ . Первая производная  $f'(t)$  отражает угол наклона функции в точке  $t$ , а вторая производная соответствует скорости, с которой этот угол изменяется. Следовательно, в общих чертах вторая производная некоторой функции – это мера ее извилистости: она принимает высокие значения, когда  $f(t)$  является

очень извилистой, и близка к нулю в остальных случаях. (Вторая производная прямой линии равна нулю, заметьте, что в этом случае линия является идеально гладкой). Символ  $\int$  обозначает интеграл, о котором мы можем думать как об операции суммирования в пределах интервала значений  $t$ . Иными словами,  $\int \{f''(t)\}^2 dt$  – это просто обобщенная мера изменчивости функции  $f'(t)$  на всем интервале ее значений. Если  $f$  является очень гладкой, то  $f'(t)$  будет почти неизменной и  $\int \{f''(t)\}^2$  примет некоторое низкое значение. Однако если  $f$  очень извилиста и изменчива, то  $f'(t)$  тоже будет значительно варьировать и  $\int \{f''(t)\}^2$  примет некоторое высокое значение. Следовательно, в формуле 2 слагаемое  $1/2 \int \{f''(t)\}^2$  заставляет функцию  $f$  становиться гладкой. Чем выше значение  $\lambda$ , чем более гладкой будет  $f$ .

Гиперпараметр  $\lambda$  определяет, насколько сильно штрафовать кривизну функции. При  $\lambda = 0$  штрафное слагаемое в формуле не имеет никакого эффекта, и функция  $f$  окажется очень извилистой и будет в точности интерполировать обучающие наблюдения. При  $\lambda \rightarrow \infty$  кривизна не допускается. Как видим,  $\lambda$  контролирует соотношение между смещением и дисперсией сглаживающего сплайна. Для нахождения приемлемого значения  $\lambda$  можно воспользоваться перекрестной проверкой.

Итак, мы берем независимый калибровочный набор, строим непараметрическую логистическую регрессию на паре  $\hat{y}_{ca}$  и  $y_{ca}$  и получаем калибровочную функцию  $f$ . Ниже описывается базовый алгоритм калибровки на сплайнах, предложенный Брайаном Люсьеной<sup>8</sup>.

## Алгоритм 1. SplineCalib

**Ввод:** векторы спрогнозированных вероятностей  $\hat{y}$  и соответствующих фактических значений  $[0; 1] y$

**Вывод:** калибровочная функция  $f: [0; 1] \rightarrow [0; 1]$

(1) Выбрать набор узлов размера  $k$  из уникальных значений  $\hat{y}$  (можно использовать все узлы, но обычно достаточно выборки размером 200).

(2) Пусть  $X$  – разложение значений  $\hat{y}$  по натуральному базису с использованием заданного набора узлов. ( $X$  будет матрицей размера  $n \times k$ ).

(3) Обучить логистическую регрессию с L2-регуляризацией с использованием перекрестной проверки на паре  $(X, y)$  по диапазону возможных значений  $\lambda$ . Выбрать  $\lambda^*$  в качестве значения, которое дает наилучшее значение логистической функции потерь по результатам перекрестной проверки.

(4) Заново обучить логистическую регрессию с L2-регуляризацией на паре  $(X, y)$ , используя значение  $\lambda^*$ , найденное на предыдущем шаге.

(5) Вернуть калибровочную функцию  $f(z): [0,1] \rightarrow [0,1]$ , полученную в результате выполнения операций: (a) разложения по натуральному базису  $z$  и (b) использования модели логистической регрессии, полученной на шаге 4, для прогнозирования вероятности.

<sup>8</sup> <https://arxiv.org/abs/1809.07751>.

Выше мы говорили, что некоторые методы машинного обучения, такие как наивный байесовский классификатор и глубокие нейронные сети, известны своей сверхуверенностью. Несмотря на то что они могут иметь хорошую (или даже превосходную) дискриминирующую способность, полученные оценки, если рассматривать их как вероятности, располагаются ближе к 1 и 0, чем следует.

Для решения этой задачи Брайан предложил функцию, выполняющую преобразование  $[0, 1] \rightarrow [0, 1]$ , под названием «функция компактного логита», которая определяется как

$$G_\epsilon(x) = \begin{cases} \frac{(1-2\epsilon)}{2 * \log((1-\epsilon)/\epsilon)} \log\left(\frac{x}{1-x}\right) + \frac{1}{2}, & \text{если } x \in [\epsilon, 1-\epsilon] \\ x, & \text{если } x \in [0, \epsilon] \cup [1-\epsilon, 1]. \end{cases}$$

График этой функции для нескольких значений  $\epsilon$  показан на рисунке ниже. По сути, представленная функция является логит-функцией, усеченной, масштабированной и сдвинутой, так чтобы области определения и значений были в диапазоне  $[\epsilon, 1 - \epsilon]$ . Второй случай просто делает ее непрерывной функцией с областями определения и значений в диапазоне  $[0, 1]$ . Легко проверить, что  $G_\epsilon(\epsilon) = \epsilon$ ,  $G_\epsilon(1 - \epsilon) = 1 - \epsilon$  и  $G_\epsilon(1/2) = 1/2$ . Идея ее использования заключается в том, чтобы выбрать такое значение  $\epsilon$ , при котором не было бы никакого содержательного «сигнала» в интервалах  $[0, \epsilon]$  и  $[1 - \epsilon, 1]$ . Брайан в своей статье предложил использовать эвристику  $\epsilon = 10^{r-1}$ , где  $r = \lfloor \log_{10}(\min(1 - p_i)) \rfloor$ , а  $p_i$  – это некалиброванные вероятности (при этом вероятности, которые точно равны 1, игнорируются).

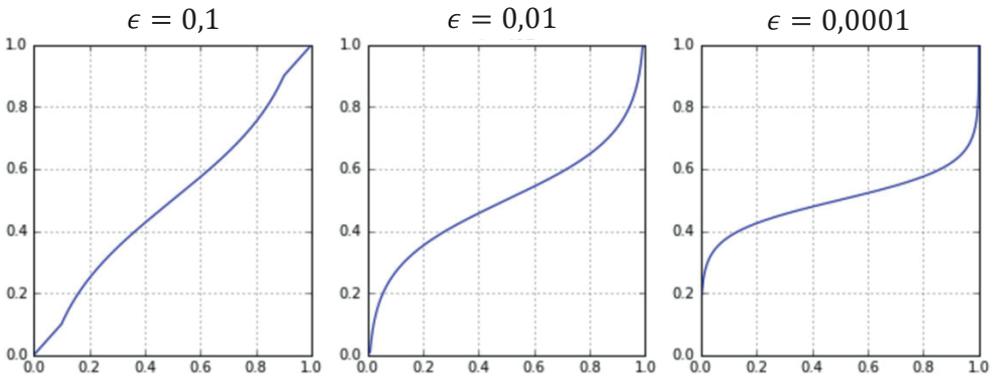


Рис. 57 Компактная логит-функция для для  $\epsilon = 0,1$ ,  $\epsilon = 0,01$  и  $\epsilon = 0,0001$

Ниже приведен алгоритм SplineCalib с преобразованием на основе компактного логита.

## Алгоритм 2. SplineCalib с преобразованием на основе компактного логита

**Ввод:** векторы спрогнозированных вероятностей  $\hat{\mathbf{y}}$  и соответствующих фактических значений  $[0; 1]$   $\mathbf{y}$  (из независимого калибровочного набора)

**Вывод:** калибровочная функция  $f: [0; 1] \rightarrow [0; 1]$

(1) Пусть  $\mathbf{y}' = G_\epsilon(\hat{\mathbf{y}})$ .

(2) Применить алгоритм 1 к  $\mathbf{y}'$  и  $\mathbf{y}$ , чтобы получить функцию  $f_1(x)$ .

(3) Вернуть функцию.

Алгоритм калибровки на сплайнах реализован в классе `SplineCalibratedClassifierCV`, который можно найти в скрипте `advanced_calibration.py`. Автором программного кода является Брайан Люсьена. Ниже приведены параметры класса `SplineCalibratedClassifierCV`.

```
from advanced_calibration import SplineCalibratedClassifierCV(base_estimator=None, ← Задаёт классификатор,
                                                                method='logistic', ← вероятности которого
                                                                'ridge' ← нужно откалибровать.
                                                                Если для параметра cv
                                                                задано значение 'prefit',
                                                                классификатор уже
                                                                должен быть обучен.
                                                                )
                                                                transform_type='none', ← Задаёт преобразование логит-
                                                                функции (применение функции
                                                                компактного логита)
                                                                'cl'
                                                                verbose=True ← Задаёт стратегию перекрестной проверки
                                                                cv=5) ← Выводит историю поиска оптимальных значений
                                                                alpha (для метода 'ridge') и C (для метода
                                                                'logistic')
```

Давайте последовательно применим различные способы калибровки на сплайнах. У нас есть уже обученная модель случайного леса, осталось запустить калибратор на калибровочном наборе.

*# создаем экземпляр класса SplineCalibratedClassifierCV, передав уже*

*# обученную модель случайного леса и применив метод ridge*

```
cal_rf_spline_ridge = SplineCalibratedClassifierCV(
    base_estimator=rf,
    cv='prefit',
    transform_type='none',
    method='ridge')
# обучаем калибратор на калибровочной выборке
cal_rf_spline_ridge.fit(X_val, y_val)
# вычисляем с помощью калибратора
# вероятности на тестовой выборке
scores_spline_ridge = cal_rf_spline_ridge.predict_proba(X_tst)[: , 1]
```

Determining Calibration Function

Originally there were 1651 knots. Reducing to 225 while preserving first and last knot.

Trying 71 values of alpha between 1e-07 and 10000000.0

Best value found alpha = 0.06309573444801943

*# создаем экземпляр класса SplineCalibratedClassifierCV, передав уже*

*# обученную модель случайного леса и применив метод logistic*

```
cal_rf_spline_logistic = SplineCalibratedClassifierCV(
    base_estimator=rf,
    cv='prefit',
    transform_type='none',
    method='logistic')
```

```
# обучаем калибратор на калибровочной выборке
cal_rf_spline_logistic.fit(X_val, y_val)
# вычисляем с помощью калибратора
# вероятности на тестовой выборке
scores_spline_logistic = cal_rf_spline_logistic.predict_proba(X_tst)[: , 1]
```

Determining Calibration Function

Originally there were 1651 knots. Reducing to 225 while preserving first and last knot.  
Trying 61 values of C between 1e-07 and 100000.0  
Best value found C = [10000.]

```
# создаем экземпляр класса SplineCalibratedClassifierCV, передав уже
# обученную модель случайного леса, применив метод logistic
# и преобразование с помощью компактного логита
cal_rf_spline_logistic_cl = SplineCalibratedClassifierCV(
    base_estimator=rf,
    cv='prefit',
    transform_type='cl',
    method='logistic')
# обучаем калибратор на калибровочной выборке
cal_rf_spline_logistic_cl.fit(X_val, y_val)
# вычисляем с помощью калибратора
# вероятности на тестовой выборке
scores_spline_logistic_cl = cal_rf_spline_logistic_cl.predict_proba(
    X_tst)[: , 1]
```

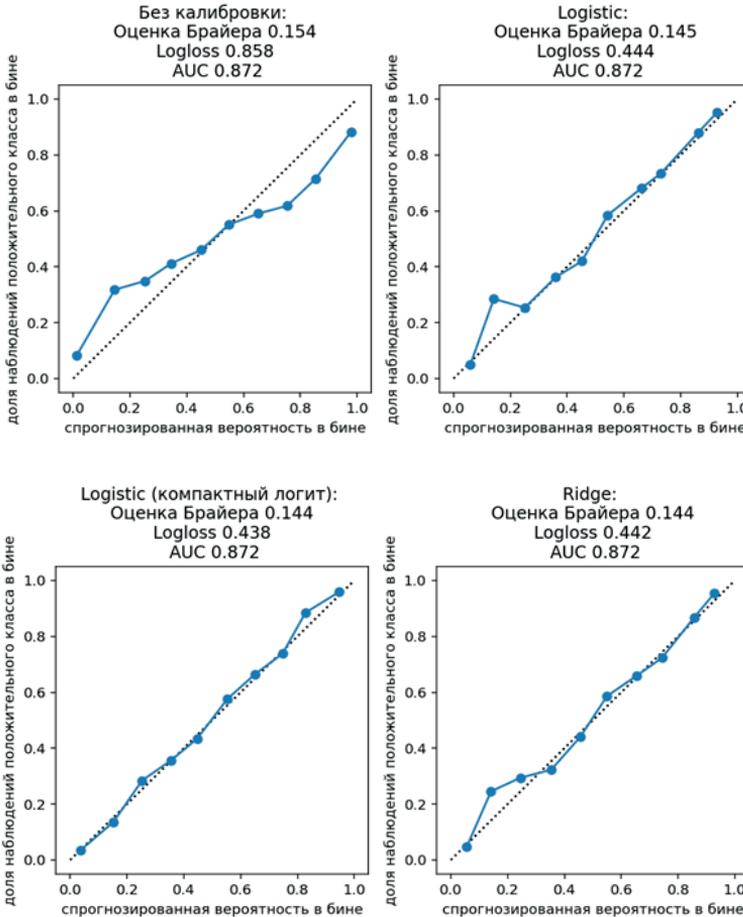
Determining Calibration Function

Originally there were 1642 knots. Reducing to 224 while preserving first and last knot.  
Trying 61 values of C between 1e-07 and 100000.0  
Best value found C = [25118.8643151]

Выводим графики калибровки и оценки метрик для трех способов калибровки на сплайнах.

```
# выводим графики калибровки и оценки метрик для
# трех способов калибровки на сплайнах
fig, axes = plt.subplots(1, 4, figsize=(15, 15))
for name, s, ax in zip(["Без калибровки", "Logistic",
                      "Logistic (компактный логит)", "Ridge"],
                      [scores, scores_spline_logistic,
                       scores_spline_logistic_cl,
                       scores_spline_ridge], axes):
    plot_calibration_curve(y_tst, s, n_bins=10, ax=ax)
    ax.set_title(string.format(name,
                               brier_score_loss(y_tst, s),
                               log_loss(y_tst, s),
                               roc_auc_score(y_tst, s)))

plt.tight_layout()
```



А теперь выполним калибровку на сплайнах, воспользовавшись перекрестной проверкой. Перед калибровкой выполним небольшую предварительную подготовку, а именно откажемся от импутации пропусков, удалив из исходной выборки наблюдения с пропусками. Кроме того, обучим модель случайного леса на обучающей выборке и получим обычные вероятности для тестовой выборки, которые мы сравним с вероятностями, полученными для тестовой выборки с помощью нашего калибратора.

```
# разбиваем данные на обучающую и тестовую выборки
X_tr, X_tst, y_tr, y_tst = train_test_split(
    data.drop('response', axis=1),
    data['response'],
    test_size=0.3,
    stratify=data['response'],
    random_state=42)

# выполняем дамми-кодирование
X_tr = pd.get_dummies(X_tr)
X_tst = pd.get_dummies(X_tst)
```

```

# из исходной выборки удаляем наблюдения с пропусками
data.dropna(inplace=True)

# разбиваем данные на обучающую и тестовую выборки
X_tr, X_tst, y_tr, y_tst = train_test_split(
    data.drop('response', axis=1),
    data['response'],
    test_size=0.3,
    stratify=data['response'],
    random_state=42)

# печатаем размеры выборок
print(X_tr.shape)
print(X_tst.shape)

(21153, 13)
(9066, 13)

# выполняем дамми-кодирование
X_tr = pd.get_dummies(X_tr)
X_tst = pd.get_dummies(X_tst)

# печатаем размеры выборок
print(X_tr.shape)
print(X_tst.shape)

(21153, 26)
(9066, 26)

# обучаем случайный лес на обучающей выборке
rf = RandomForestClassifier(n_estimators=100, random_state=42).fit(
    X_tr, y_tr)
# вычисляем вероятности для тестовой выборки
scores = rf.predict_proba(X_tst)[:, 1]

# создаем экземпляр класса SplineCalibratedClassifierCV,
# задав перекрестную проверку и применив метод ridge
cal_rf_spline_ridge = SplineCalibratedClassifierCV(
    base_estimator=rf,
    cv=5,
    transform_type='none',
    method='ridge')
# обучаем модель и выполняем калибровку
cal_rf_spline_ridge.fit(X_tr, y_tr)
# вычисляем с помощью калибратора
# вероятности на тестовой выборке
scores_spline_ridge = cal_rf_spline_ridge.predict_proba(X_tst)[:, 1]

training fold 1 of 5
training fold 2 of 5
training fold 3 of 5
training fold 4 of 5
training fold 5 of 5
Training Full Model
Determining Calibration Function
Originally there were 6114 knots. Reducing to 226 while preserving first and last knot.

```

Trying 71 values of alpha between 1e-07 and 10000000.0  
 Best value found alpha = 0.00039810717055349735

```
# создаем экземпляр класса SplineCalibratedClassifierCV,
# задав перекрестную проверку и применив метод logistic
cal_rf_spline_logistic = SplineCalibratedClassifierCV(
    base_estimator=rf,
    cv=5,
    transform_type='none',
    method='logistic')
# обучаем модель и выполняем калибровку
cal_rf_spline_logistic.fit(X_tr, y_tr)
# вычисляем с помощью калибратора
# вероятности на тестовой выборке
scores_spline_logistic = cal_rf_spline_logistic.predict_proba(X_tst)[: , 1]
```

```
training fold 1 of 5
training fold 2 of 5
training fold 3 of 5
training fold 4 of 5
training fold 5 of 5
Training Full Model
Determining Calibration Function
Originally there were 6114 knots. Reducing to 226 while preserving first and last knot.
Trying 61 values of C between 1e-07 and 100000.0
Best value found C = [25118.8643151]
```

```
# создаем экземпляр класса SplineCalibratedClassifierCV,
# задав перекрестную проверку, применив метод logistic
# и преобразование с помощью компактного логита
cal_rf_spline_logistic_cl = SplineCalibratedClassifierCV(
    base_estimator=rf,
    cv=5,
    transform_type='cl',
    method='logistic')
# обучаем модель и выполняем калибровку
cal_rf_spline_logistic_cl.fit(X_tr, y_tr)
# вычисляем с помощью калибратора
# вероятности на тестовой выборке
scores_spline_logistic_cl = cal_rf_spline_logistic_cl.predict_proba(
    X_tst)[: , 1]
```

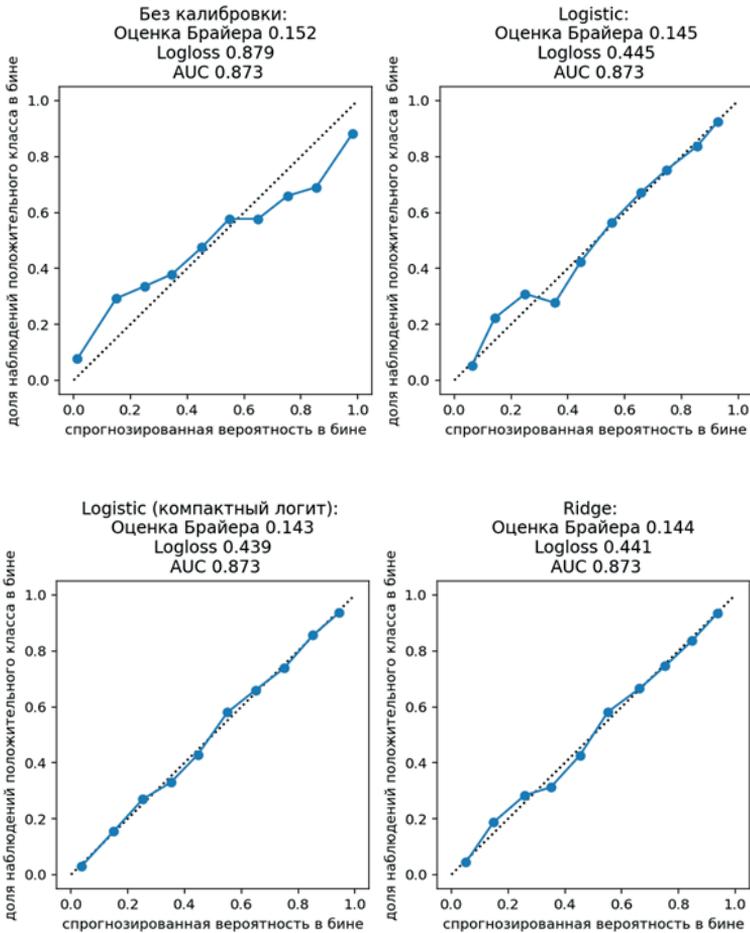
```
training fold 1 of 5
training fold 2 of 5
training fold 3 of 5
training fold 4 of 5
training fold 5 of 5
Training Full Model
Determining Calibration Function
Originally there were 5907 knots. Reducing to 225 while preserving first and last knot.
Trying 61 values of C between 1e-07 and 100000.0
Best value found C = [100000.]
```

Опять выводим графики калибровки и оценки метрик для трех способов калибровки на сплайнах.

```

# выводим графики калибровки и оценки метрик для
# трех способов калибровки на слайдах
fig, axes = plt.subplots(1, 4, figsize=(15, 15))
for name, s, ax in zip(["Без калибровки", "Logistic",
                       "Logistic (компактный логит)", "Ridge"],
                       [scores, scores_spline_logistic,
                       scores_spline_logistic_cl,
                       scores_spline_ridge], axes):
    plot_calibration_curve(y_tst, s, n_bins=10, ax=ax)
    ax.set_title(string.format(name,
                               brier_score_loss(y_tst, s),
                               log_loss(y_tst, s),
                               roc_auc_score(y_tst, s)))
plt.tight_layout()

```



## 7.19. ПОЛЕЗНЫЕ КЛАССЫ COUNTVECTORIZER И TFIDFVECTORIZER ДЛЯ РАБОТЫ С ТЕКСТОМ

До этого момента мы работали со структурированными данными. Теперь давайте посмотрим, как можно обрабатывать такой тип неструктурированных данных, как текст.

Текстовые данные – это данные, которые состоят из фраз или предложений. Примерами таких данных могут быть твиты, логи чата или отзывы о гостинице, а также собрание сочинений Шекспира, содержание Википедии или проекта «Гутенберг», включающего 50 000 электронных книг. Все эти коллекции содержат информацию, представленную преимущественно в виде предложений, составленных из слов.

С точки зрения анализа текста набор данных часто называют *корпусом* (*corpus*), и каждое наблюдение, представленное в виде отдельного текста, называется *документом* (*document*).

Один из самых простых, но эффективных и широко используемых способов подготовки текста для машинного обучения – это представление текстовой информации в виде «мешка слов» (*bag-of-words*). Используя это представление, мы удаляем структуру исходного текста, например главы, параграфы, предложения, форматирование, и лишь подсчитываем частоту встречаемости каждого слова в каждом документе корпуса. Удаление структуры и подсчет частоты каждого слова позволяет получить образное представление текста в виде «мешка слов». Получение представления «мешок слов» включает следующие три этапа:

- 1) *токенизация (tokenization)*. Разбиваем каждый документ на слова, которые встречаются в нем (*токены*), например с помощью пробелов и знаков пунктуации;
- 2) *построение словаря (vocabulary building)*. Собираем словарь всех слов, которые появляются в любом из документов, и пронумеровываем их (например, в алфавитном порядке);
- 3) *создание разреженной матрицы (sparse matrix encoding)*. Для каждого документа подсчитываем, как часто каждое из слов, занесенное в словарь, встречается в документе.

Сейчас давайте посмотрим, как мы можем применить обработку данных «мешок слов», используя библиотеку *scikit-learn*. Рисунок иллюстрирует процесс на примере строки «This is how you get war». В итоге каждый документ можно представить в виде вектора частот слов. Для каждого слова, записанного в словаре, мы подсчитываем частоту его встречаемости в каждом документе. Это означает, что в нашем числовом представлении каждый признак соответствует каждому уникальному слову набора данных. Обратите внимание: порядок слов в исходной строке абсолютно не имеет никакого значения для представления признаков «мешок слов».



Рис. 58 Представление «мешок слов»

Давайте импортируем привычные библиотеки pandas и NumPy, классы Pipeline, GridSearchCV, LogisticRegression и функцию train\_test\_split() и создадим игрушечный набор данных, состоящий из двух примеров.

```

# импортируем библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression

# создаем набор из двух примеров
toy_corpus = ['Машина едет по дороге',
              'Грузовик едет по шоссе']
  
```

Модель «мешка слов» реализована в классе CountVectorizer, который выполняет соответствующее преобразование. Сейчас мы импортируем этот класс, создадим экземпляр класса и обучим модель на наших игрушечных данных.

```

# импортируем класс CountVectorizer
from sklearn.feature_extraction.text import CountVectorizer
# создаем экземпляр класса CountVectorizer
vect = CountVectorizer()
# обучаем CountVectorizer
vect.fit(toy_corpus)
  
```

Процесс обучения CountVectorizer включает в себя токенизацию обучающих данных и построение словаря, к которому мы можем получить доступ с помощью атрибута vocabulary\_.

```
# смотрим размер и содержимое словаря
print("Размер словаря: {}".format(len(vect.vocabulary_)))
print("Содержимое словаря:\n {}".format(vect.vocabulary_))
```

```
Размер словаря: 6
Содержимое словаря:
{'машина': 3, 'едет': 2, 'по': 4, 'дороге': 1, 'грузовик': 0, 'шоссе': 5}
```

Ключами словаря являются слова (их еще называют *термами*), а значениями – соответствующие индексы в матрице признаков. Словарь состоит из 6 слов, начинается со слова «грузовик» (индекс 0) и заканчивается словом «шоссе» (индекс 5).

Еще один способ получить доступ к словарю – это использование метода `.get_feature_names_out()`. Он возвращает удобный список, в котором каждый элемент соответствует одному признаку.

```
# вычислим количество признаков
# и посмотрим список признаков
feature_names = vect.get_feature_names()
print("Количество признаков: {}".format(len(feature_names)))
print("Признаки:", feature_names)
```

```
Количество признаков: 6
Признаки: ['грузовик', 'дороге', 'едет', 'машина', 'по', 'шоссе']
```

Чтобы получить представление «мешок слов» для обучающих данных, нужно вызвать метод `.transform()`.

```
# получаем представление "мешок слов"
bag_of_words = vect.transform(toy_corpus)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

```
bag_of_words: <2x6 sparse matrix of type '<class 'numpy.int64'>'
with 8 stored elements in Compressed Sparse Row format>
```

Представление «мешок слов» записывается в формате разреженной матрицы SciPy, которая хранит только ненулевые элементы. В нашем случае матрица имеет форму 2×6, в ней каждая строка соответствует примеру, а каждый столбец (признак) соответствует слову, записанному в словаре. Эту матрицу еще называют матрицей «документ–терм». Использование разреженной матрицы обусловлено тем, что документы, как правило, содержат лишь небольшое количество слов, записываемое в словарь, таким образом, большая часть элементов массива будет равна 0. Хранение всех этих нулей – ненужные затраты памяти.

Чтобы взглянуть на фактическое содержимое разреженной матрицы, мы можем преобразовать ее в «плотный» массив NumPy (который помимо ненулевых элементов также хранит все нулевые элементы) с помощью метода `.toarray()`. Заметим: это возможно лишь благодаря тому, что мы используем игрушечный набор данных, который содержит лишь 6 слов. Если бы мы взяли реальный набор данных, то получили бы исключение `MemoryError`.

```
# получаем плотное представление "мешок слов"
print("Плотное представление bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

Плотное представление bag\_of\_words:

```
[[0 1 1 1 1 0]
 [1 0 1 0 1 1]]
```

Видно, что матрица имеет две строки и шесть столбцов. Проясняется смысл фразы «8 stored elements», матрица хранит 8 «единичек», частоты слов равны либо 0, либо 1. Ни одна из двух строк массива `toy_corpus` не содержит слов, которые встречались бы дважды в одной строке. Давайте разберемся, как нужно работать с этими векторами признаков. Первая строка («Машина едет по дороге») соответствует первому ряду элементов, и слово «грузовик», записанное в словаре первым, встречается в ней ноль раз. Второе слово «дороге» встречается в этой строке один раз. Третье слово «едет» встречается один раз. Взглянув на оба ряда, мы можем увидеть, что третье слово «едет», пятое слово «по» встречаются в обеих строках («Машина едет по дороге» и «Грузовик едет по шоссе»).

Теперь давайте подробнее посмотрим, как происходит процесс токенизации. Когда мы обучали модель, мы получили следующий вывод.

```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0,
                max_features=None, min_df=1, ngram_range=(1, 1),
                preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\b\w+\b',
                tokenizer=None, vocabulary=None)
```

Нас будет интересовать значение параметра `token_pattern`. `CountVectorizer` извлекает токены с помощью регулярных выражений. По умолчанию используется регулярное выражение «`(?u)\b\w+\b`».

Для тех, кто незнаком с регулярными выражениями, поясним: это выражение включает флаг `re.U` (`re.UNICODE`) и позволяет найти все последовательно-сти символов, которые состоят как минимум из двух букв или цифр (`\w`) и отделены друг от друга границами слов (`\b`).

Затем `CountVectorizer` преобразует все слова в строчные (если для параметра `lowercase` задано значение `True`, оно выставлено по умолчанию), поэтому, например, слова «слон», «Слон» и «сЛон» будут соответствовать одному и тому же токenu (и, следовательно, одному и тому же признаку). Этот простой принцип достаточно хорошо работает на практике, однако, как мы уже видели ранее, можно получить массу неинформативных признаков (например, числа). Один из способов решить эту проблему – использовать только те токены, которые встречаются по крайней мере в двух документах (или по крайней мере в пяти документах и т. д.).

С помощью параметра `min_df` мы можем задать минимальную частоту документа (`minimum document frequency`). Речь идет о минимальном количестве / минимальной доле документов, в которых должен появиться терм. Термы, которые встречаются в количестве документов (доле документов), меньшем(ей), чем заданное значение, игнорируются.

Сейчас мы извлечем только те слова, которые встретились не менее, чем в двух документах (`min_df=2`).

```
# создаем экземпляр класса CountVectorizer, будем
# извлекать слова, которые встретились не менее,
# чем в двух документах
vect = CountVectorizer(min_df=2)
# обучаем CountVectorizer
vect.fit(toy_corpus)
# смотрим содержимое словаря
print("Содержимое словаря:\n {}".format(vect.vocabulary_))
```

```
Содержимое словаря:
{'едет': 0, 'по': 1}
```

Видим, что теперь словарь содержит слова «едет» и «по», которые встречаются в обеих строках («Машина **едет по** дороге» и «Грузовик **едет по** шоссе»).

С помощью параметра `max_df` мы можем задать максимальную частоту документа (maximum document frequency). С помощью параметра `max_df` мы можем задать максимальное количество (максимальную долю) документов, в которых должен появиться терм. Термы, которые встречаются в количестве документов (доле документов), большем(ей), чем заданное значение, игнорируются.

Сейчас мы извлечем только те слова, которые встретились не более, чем в одном документе (`max_df=1`).

```
# создаем экземпляр класса CountVectorizer, будем
# извлекать слова, которые встретились не более,
# чем в одном документе
vect = CountVectorizer(max_df=1)
# обучаем CountVectorizer
vect.fit(toy_corpus)
# смотрим содержимое словаря
print("Содержимое словаря:\n {}".format(vect.vocabulary_))
```

```
Содержимое словаря:
{'машина': 2, 'дороге': 1, 'грузовик': 0, 'шоссе': 3}
```

Кроме того, с помощью параметра `stop_words` можно задать список стоп-слов, которые нужно исключить из анализа. Если задано значение `'english'`, используется встроенный список стоп-слов для английского языка. Если задан список, то предполагается, что этот список содержит стоп-слова, и все они будут удалены из полученных токенов. Применяется, если признаки извлекаются из слов (`analyzer='word'`). Если задано `None`, стоп-слова не будут использоваться. Для `max_df` может быть задано значение в диапазоне `[0,7, 1,0)` для автоматического обнаружения и фильтрации стоп-слов, исходя из встречаемости термов в документах корпуса.

Итак, обозначим в качестве стоп-слова предлог `по` и заново обучим `CountVectorizer`, таким образом, данный предлог не попадет в словарь (`stop_words=['по']`).

```
# создаем экземпляр класса CountVectorizer,
# будем извлекать слова, кроме предлога по
vect = CountVectorizer(stop_words=['no'])
# обучаем CountVectorizer
vect.fit(toy_corpus)
# смотрим содержимое словаря
print("Содержимое словаря:\n {}".format(vect.vocabulary_))
```

Содержимое словаря:

```
{'машина': 3, 'едет': 2, 'дороге': 1, 'грузовик': 0, 'шоссе': 4}
```

С помощью параметра `max_features` мы можем задать топ-`max_features` признаков (например, топ-10 или топ-20 признаков), упорядоченных по частоте встречаемости в корпусе.

Сейчас мы создадим новый набор из трех примеров и извлечем три самых часто встречаемых слова.

```
# создаем набор из трех примеров
toy_corpus = ['Машина едет по дороге',
              'Машина столкнулась на дороге с автобусом',
              'Машина сбила на дороге пешехода']
# создаем экземпляр класса CountVectorizer, будем
# извлекать три самых часто встречающихся слова
vect = CountVectorizer(max_features=3)
# обучаем CountVectorizer
vect.fit(toy_corpus)
# смотрим содержимое словаря
print("Содержимое словаря:\n {}".format(vect.vocabulary_))
```

Содержимое словаря: {'машина': 1, 'дороге': 0, 'на': 2}

Теперь поговорим об  $n$ -граммах.  $n$ -грамма – это последовательность из  $n$  элементов. Элементом может быть символ или слово, таким образом, может быть  $n$ -грамма символов или  $n$ -грамма слов. Последовательность из одного элемента называют униграммой, последовательность из двух элементов – биграммой, последовательность из трёх элементов – триграммой.

Параметр `analyzer` класса `CountVectorizer` определяет способ извлечения признаков – на основе  $n$ -грамм, состоящих из слов (значение `'word'` используется по умолчанию), или  $n$ -грамм, состоящих из символов (значение `'char'`). Когда задано значение `'char_wb'`,  $n$ -граммы из символов создаются только на основе текста внутри границ слов. Таким образом, признаками могут быть не только слова, но и символы.

Параметр `ngram_range` задает в виде двухэлементного кортежа (`min_n`, `max_n`) нижнюю и верхнюю границы диапазона значений  $n$  для разных  $n$ -грамм из слов и  $n$ -грамм из символов. Все значения  $n$ , удовлетворяющие `min_n <= n <= max_n`, будут использованы. Например, `(1, 1)` означает создание только униграмм, `(1, 2)` означает создание униграмм и биграмм и `(2, 2)` означает создание только биграмм.

Сейчас создадим новый набор и извлечем униграммы, биграммы и триграммы.

```
# создаем новый набор
toy_corpus = ['Серебристая машина едет по разбитой дороге',
              'Грязный грузовик мчится по широкому шоссе']
# создаем экземпляр класса CountVectorizer, будем извлекать
# униграммы, биграмммы и триграммы
vect = CountVectorizer(ngram_range=(1, 3))
vect.fit(toy_corpus)
print("Признаки:", vect.get_feature_names())
```

```
Признаки: ['грузовик', 'грузовик мчится', 'грузовик мчится по', 'грязный', 'грязный грузо-
вик', 'грязный грузовик мчится', 'дороге', 'едет', 'едет по', 'едет по разбитой', 'машина',
'машина едет', 'машина едет по', 'мчится', 'мчится по', 'мчится по широкому', 'по', 'по раз-
битой', 'по разбитой дороге', 'по широкому', 'по широкому шоссе', 'разбитой', 'разбитой до-
роге', 'серебристая', 'серебристая машина', 'серебристая машина едет', 'широкому', 'широкому
шоссе', 'шоссе']
```

В выводе униграммы выделены желтым фоном, биграмммы – голубым фоном, а триграммы – зеленым фоном.

Теперь применим класс `CountVectorizer` для классификации отзывов о ресторанах на отрицательные и положительные.

Наши данные записаны в tsv-файле `Restaurant_Reviews.tsv`. Для естественной обработки языка TSV-формат (от tab-separated values – значения, разделенные символами табуляции) может быть более подходящим, чем CSV-формат, потому что запятые, скорее всего, будут частью предложения, и в CSV-формате они могут быть распознаны как разделители, а вот символы табуляции вряд ли будут частью предложения. При чтении файла с помощью параметра `sep` нужно задать соответствующий разделитель, а с помощью значения параметра `quoting=3` – игнорировать двойные кавычки.

```
# загружаем данные
data = pd.read_csv('Data/Restaurant_Reviews.tsv',
                  sep='\t',
                  quoting=3)
data.head()
```

	Review	Liked
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1

Выполним случайное разбиение данных на обучающую и тестовую выборки: сформируем обучающий массив признаков, тестовый массив признаков, обучающий массив меток, тестовый массив меток.

```
# разбиваем данные на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(
    data['Review'],
    data['Liked'],
```

```
test_size=0.3,
stratify=data['Liked'],
random_state=42)
```

Давайте снова создадим экземпляр класса CountVectorizer и обучим на наших данных, вычислим количество признаков и посмотрим список первых 100 признаков.

```
# создаем экземпляр класса CountVectorizer
vect = CountVectorizer()
# обучаем CountVectorizer
vect.fit(X_train)
# вычислим количество признаков и посмотрим список первых 100 признаков
feature_names = vect.get_feature_names()
print("Количество признаков: {}".format(len(feature_names)))
print("Первые 100 признаков:\n{}".format(feature_names[:100]))
```

Количество признаков: 1640

Первые 100 признаков:

```
['00' '10' '100' '11' '12' '15' '17' '1979' '20' '2007' '23' '30' '35'
'40' '45' '4ths' '5lb' '70' '99' 'about' 'above' 'absolute' 'absolutely'
'absolutley' 'accordingly' 'accountant' 'ache' 'acknowledged' 'across'
'actual' 'actually' 'added' 'affordable' 'after' 'afternoon' 'again'
'ago' 'airline' 'airport' 'ala' 'albondigas' 'all' 'allergy' 'almost'
'alone' 'also' 'although' 'always' 'am' 'amazing' 'ambiance' 'ambience'
'amount' 'an' 'and' 'andddd' 'angry' 'annoying' 'another' 'anticipated'
'any' 'anymore' 'anyone' 'anytime' 'anyways' 'apart' 'apology' 'app'
'appalling' 'apparently' 'appealing' 'appetizer' 'apple' 'are' 'area'
'aren' 'aria' 'around' 'array' 'arrived' 'article' 'as' 'ask' 'asked'
'asking' 'assure' 'at' 'ate' 'atmosphere' 'atrocious' 'attack'
'attention' 'attentive' 'attitudes' 'authentic' 'average' 'avocado'
'avoid' 'avoided' 'away']
```

Скорее всего, из-за проклятия размерности мы получим сильное переобучение, кроме того, это переобучение может стать еще более выраженным, если мы добавим биграммы и триграммы, поэтому необходимо применять регуляризацию в логистической регрессии и контролировать максимальное количество признаков при выполнении векторизации.

Во избежание протечек надежнее всего выполнять векторизацию (вспомним, что представление bag-of-words предполагает вычисление статистик – частот слов) и построение модели логистической регрессии в конвейере. Давайте зададим список стоп-слов, а также создадим конвейер из экземпляров классов CountVectorizer и LogisticRegression и попробуем найти оптимальный диапазон значений  $n$  для  $n$ -грамм и оптимальное значение регуляризации.

```
# задаем список стоп-слов
stop_wrds = ['be', 'is', 'are', 'the', 'a',
             'an', 'on', 'of', 'and', 'in']

# задаем конвейер
pipe = Pipeline([
    ('vectorizer', CountVectorizer(stop_words=stop_wrds)),
    ('logreg', LogisticRegression(solver='liblinear'))
])
```

```
# задаем сетку гиперпараметров
param_grid = {
    'vectorizer__ngram_range': [(1, 1), (1, 2), (2, 2)],
    'logreg__C': [.01, .1, .5, 1, 5, 10, 100, 150]
}
```

Теперь выполняем обычный поиск по сетке.

```
# создаем экземпляр класса GridSearchCV, передав конвейер,
# сетку гиперпараметров и указав количество
# блоков перекрестной проверки
gs = GridSearchCV(pipe,
                  param_grid,
                  cv=10)
# выполняем поиск по всем значениям сетки
gs.fit(X_train['Review'], y_train)
# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs.best_params_))
# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    gs.best_score_))
# смотрим значение правильности на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    gs.score(X_test['Review'], y_test)))
```

```
Наилучшие значения гиперпараметров:
{'logreg__C': 100, 'vectorizer__ngram_range': (1, 1)}
Наилучшее значение правильности: 0.796
Значение правильности на тестовой выборке: 0.803
```

Помимо исключения несущественных признаков, повысить качество можно с помощью масштабирования признаков в зависимости от степени их информативности. Одним из наиболее распространенных способов такого масштабирования является метод *частота термина – обратная частота документа* (term frequency-inverse document frequency, tf-idf). Идея этого метода заключается в том, чтобы присвоить большой вес терму, который часто встречается в конкретном документе, но при этом редко встречается в остальных документах корпуса. Если слово часто появляется в конкретном документе, но при этом редко встречается в остальных документах, оно, вероятно, будет описывать содержимое данного документа лучше.

Частота термина – это частота интересующего нас слова в каждом документе корпуса. Это отношение количества появлений интересующего слова в документе к общему количеству слов в данном документе. Оно увеличивается по мере увеличения количества вхождений этого слова в документе. У каждого документа будет своя частота термина.

$$tf(i) = \frac{\text{количество появлений термина } i \text{ в документе } j}{\text{общее количество термов в документе } j}$$

Обратная частота термина – это отношение общего количества документов к количеству документов, содержащих интересующий терм.

$$idf(i) = \frac{\text{общее количество документов}}{\text{количество документов, содержащих терм } i}$$

Часто берут логарифм отношения:

$$idf(i) = \log\left(\frac{\text{общее количество документов}}{\text{количество документов, содержащих терм } i}\right)$$

Также есть модификация формулы, когда к знаменателю дроби добавляют единицу:

$$idf(i) = \log\left(\frac{\text{общее количество документов}}{1 + \text{количество документов, содержащих терм } i}\right)$$

В итоге получаем значение tf-idf для термина  $i$  в документе  $j = tf(i) * idf(i)$ . Приведем пример для более ясного понимания. Допустим, у нас есть два предложения.

Предложение А. Этот автомобиль уже едет по скоростной дороге.

Предложение В. Этот грузовик уже едет по скоростной трассе.

Напомним: каждое предложение является документом.

**Таблица 9** Значения tf, idf и tf\*idf для термов нашего игрушечного корпуса

Терм	TF		IDF	TF*IDF	
	А	В		А	В
Этот	1/7	1/7	$\log(2/2)=0$	0	0
автомобиль	1/7	0	$\log(2/1)=0,3$	0,043	0
грузовик	0	1/7	$\log(2/1)=0,3$	0	0,043
уже	1/7	1/7	$\log(2/2)=0$	0	0
едет	1/7	1/7	$\log(2/2)=0$	0	0
по	1/7	1/7	$\log(2/2)=0$	0	0
скоростной	1/7	1/7	$\log(2/2)=0$	0	0
дороге	1/7	0	$\log(2/1)=0,3$	0,043	0
трассе	0	1/7	$\log(2/1)=0,3$	0	0,043

Из таблицы выше мы видим, что значения tf-idf слов, встретившихся в обоих предложениях, равны нулю, что говорит об их незначимости. С другой стороны, значения tf-idf слов «автомобиль», «грузовик», «дороге», «трассе» не равны нулю. Они имеют большую значимость.

В библиотеке scikit-learn метод tf-idf реализован в двух классах: TfidfTransformer, который принимает на вход разреженную матрицу, полученную с помощью CountVectorizer, и преобразует ее, и TfidfVectorizer, который принимает на вход текстовые данные и выполняет как выделение признаков в представление «мешок слов», так и преобразование tf-idf. Для преобразования tf-idf существует несколько вариантов взвешивания частот, о которых вы можете прочитать в Вики-

педии. Значение tf-idf для слова  $w$  в документе  $d$  вычисляется с помощью классов `TfidfTransformer` и `TfidfVectorizer` по модифицированным формулам:

$$tfidf(w, d) = \text{tfln}\left(\frac{N+1}{df(t)+1}\right) + 1 \quad \text{или} \quad tfidf(w, d) = \text{tfln}\left(\frac{N}{df(t)}\right) + 1,$$

где

$tf$  – частота термина, т.е. частота встречаемости данного термина в рассматриваемом документе;

$N$  – общее количество документов в корпусе;

$df(x)$  – количество документов обучающего набора, в которых встретился данный терм.

По умолчанию используется формула слева.

Давайте импортируем классы `TfidfTransformer` и `TfidfVectorizer`.

Теперь мы создадим экземпляр класса `TfidfVectorizer`, создадим игрушечный корпус из двух документов – предложений и выполним масштабирование tf-idf.

```
# импортируем классы TfidfVectorizer и TfidfTransformer
from sklearn.feature_extraction.text import (TfidfVectorizer,
                                             TfidfTransformer)

# создаем экземпляр класса TfidfVectorizer
tdidfvectorizer = TfidfVectorizer(smooth_idf=True)
# создаем игрушечный корпус из двух документов - предложений
toy_corpus = ['Этот автомобиль едет', 'Этот грузовик едет']
# выполним масштабирование tf-idf
tfidf = tdidfvectorizer.fit_transform(toy_corpus)
tfidf.toarray()
```

```
array([[0.70490949, 0.          , 0.50154891, 0.50154891],
       [0.          , 0.70490949, 0.50154891, 0.50154891]])
```

Давайте выясним, как были получены эти значения.

Итак, у нас есть два предложения.

Предложение А. Этот автомобиль едет.

Предложение В. Этот грузовик едет.

Сначала по каждому предложению мы получаем вектор сырых значений tf-idf.

Терм	TF		IDF	Raw TF*IDF	
	A	B		A	B
автомобиль	1/3	0	$\ln\left(\frac{2+1}{1+1}\right)+1=1,405$	0,468	0
грузовик	0	1/3	$\ln\left(\frac{2+1}{1+1}\right)+1=1,405$	0	0,468
едет	1/3	1/3	$\ln\left(\frac{2+1}{2+1}\right)+1=1$	0,333	0,333
этот	1/3	1/3	$\ln\left(\frac{2+1}{2+1}\right)+1=1$	0,333	0,333

Затем к векторам сырых значений tf-idf применяется L2-регуляризация. Другими словами, мы масштабируем векторизованное представление каждого документа с помощью евклидовой нормы:

$$v_{norm} = \frac{v}{v_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}}.$$

Подобное масштабирование означает, что длина документа (количество слов) не меняет его векторизованное представление.

Давайте вычислим вектор нормализованных значений tf-idf для первого предложения:

$$v_{1, norm} = \frac{[0, 468, 0, 0, 333, 0, 333]}{\sqrt{0, 468^2 + 0^2 + 0, 333^2 + 0, 333^2}} = \frac{[0, 468, 0, 0, 333, 0, 333]}{0, 664} = [0, 705, 0, 0, 502, 0, 502].$$

Видим, что он совпал с результатами, полученными автоматически.

```
array([[0.70490949, 0.          , 0.50154891, 0.50154891],
       [0.          , 0.70490949, 0.50154891, 0.50154891]])
```

Давайте добавим TfidfTransformer в наш конвейер и снова выполним поиск по сетке. При этом попробуем, помимо силы регуляризации, включения биграмм, ограничивать максимальное количество признаков.

```
# задаем новую сетку гиперпараметров
param_grid2 = {
    'vectorizer_ngram_range': [(1, 1), (1, 2), (2, 2)],
    'vectorizer_max_features': [2000, 2500, 3000, 3500, 4000,
                                4500, 5000, 6000, 7000],
    'logreg_C': [.01, .1, .5, 1, 5, 10,
                 100, 150, 200, 300]
}

# задаем конвейер
pipe2 = Pipeline([
    ('vectorizer', CountVectorizer(stop_words=stop_wrds)),
    ('tfidftransformer', TfidfTransformer()),
    ('logreg', LogisticRegression(solver='liblinear'))
])

# создаем экземпляр класса GridSearchCV
gs2 = GridSearchCV(pipe2,
                   param_grid2,
                   cv=10)

# выполняем поиск по всем значениям сетки
gs2.fit(X_train['Review'], y_train)

# смотрим наилучшие значения гиперпараметров
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs2.best_params_))

# смотрим наилучшее значение правильности
print("Наилучшее значение правильности: {:.3f}".format(
    gs2.best_score_))
```

```
# смотрим значение правильности на тестовой выборке
print("Значение правильности на тестовой выборке: {:.3f}".format(
    gs2.score(X_test['Review'], y_test)))
```

Наилучшие значения гиперпараметров:

```
{'logreg__C': 5, 'vectorizer__max_features': 4000, 'vectorizer__ngram_range': (1, 2)}
```

Наилучшее значение правильности: 0.800

Значение правильности на тестовой выборке: 0.807

Как видим, масштабирование tf-idf в сочетании с регуляризацией и ограничением максимального количества признаков позволило немного улучшить качество модели.

Теперь для улучшения качества применим метод back translation, т.е. переведем отзывы с английского на русский, а затем снова на английский, в процессе перевода у нас могут появиться синонимы, которые будут новыми признаками для модели. Загружаем необходимую библиотеку googletrans, использующую Google Translate API (установить ее можно с помощью команды `pip install googletrans`), и создаем экземпляр класса `Translator`.

```
# импортируем библиотеку googletrans
import googletrans
from googletrans import Translator
translator = Translator()
```

Пишем функцию обратного перевода текста.

```
# функция обратного перевода текста
def back_translation(text):
    ru_text = translator.translate(text, dest='ru', src='en').text
    en_text = translator.translate(ru_text, dest='en', src='ru').text

    return en_text
```

Выполняем обратный перевод для обучающего набора (только для обучающего набора!).

```
# выполняем обратный перевод для обучающего набора
X_train_back = X_train.apply(back_translation)
```

Для воспроизводимости результатов рекомендуется сохранить набор с переводами, потому что при повторном обращении к Google Translate API вы можете получить немного отличающиеся результаты.

Посмотрим размер исходного обучающего набора и переведенного обучающего набора.

```
# смотрим размер исходного обучающего набора
# и переведенного обучающего набора
print(X_train.shape, X_train_back.shape)
```

```
(700,) (700,)
```

Давайте сравним первые пять текстов исходного обучающего набора и переведенного обучающего набора.

```
# смотрим первые пять текстов исходного
# обучающего набора
print(X_train.head())
print("")
# смотрим первые пять текстов переведенного
# обучающего набора
print(X_train_back.head())

284    I would definitely recommend the wings as well...
5      Now I am getting angry and I want my damn pho.
368    The staff are great, the ambiance is great.
670                                         dont go here.
413    I can assure you that you won't be disappointed.
Name: Review, dtype: object

284    I definitely recommend wings, as well as pizza.
5      Now I am angry and want my damned FOR.
368    The staff is magnificent, the atmosphere is ma...
670                                         Do not go here.
413    I can assure you that you will not be disappoi...
Name: Review, dtype: object
```

Тексты некоторых отзывов в исходном обучающем и переведенном обучающем наборах могут совпадать. Давайте объединим исходный и переведенный обучающие наборы и избавимся от одинаковых записей.

```
# объединяем исходный и переведенный обучающие наборы
X_train_new = pd.concat([X_train, X_train_back])
y_train_new = pd.concat([y_train, y_train])

# избавляемся от записей с одинаковым текстом обзора
y_train_new = y_train_new[~X_train_new.duplicated()]
X_train_new.drop_duplicates(inplace=True)
print(y_train_new.shape, X_train_new.shape)
(1349,) (1349,)
```

Проверяем, нет ли в тестовом наборе текстов, которые есть в новом обучающем наборе. Такая ситуация редко бывает для длинных текстов, однако типична для коротких отзывов.

```
# проверяем, нет ли в тестовом наборе текстов,
# которые есть в новом обучающем наборе
np.intersect1d(X_train_new.values, X_test.values)

array(['The food was terrible.'], dtype=object)
```

Видим, что такой текст у нас есть. Давайте сформируем тестовый набор так, чтобы в нем не было текстов, имеющих в новом обучающем наборе.

```
# конкатенируем тестовые массив признаков и массив меток
test_data = pd.concat([X_test, y_test], axis=1)
```

```
# формируем тестовый набор так, чтобы в нем не было текстов,
# имеющих в новом обучающем наборе
test_data = test_data[~test_data['Review'].isin(X_train_new)]
# формируем новый тестовый массив признаков
X_test = test_data['Review']
# формируем новый тестовый массив меток
y_test = test_data['Liked']
```

Давайте снова выполним обычный поиск по сетке, используя уже новый обучающий набор.

```
# выполняем поиск по всем значениям сетки
gs2.fit(X_train_new, y_train_new)
print("Наилучшие значения гиперпараметров:\n{}".format(
    gs2.best_params_))
print("Наилучшее значение правильности: {:.3f}".format(
    gs2.best_score_))
print("Значение правильности на тестовой выборке: {:.3f}".format(
    gs2.score(X_test, y_test)))
```

Наилучшие значения гиперпараметров:

```
{'logreg__C': 150, 'vectorizer__max_features': 7000, 'vectorizer__ngram_range': (1, 2)}
```

Наилучшее значение правильности: 0.961

Значение правильности на тестовой выборке: 0.833

С помощью метода `back translation` мы добились повышения качества.

## 7.20. СРАВНЕНИЕ МОДЕЛЕЙ, ПОЛУЧЕННЫХ В ХОДЕ ПОИСКА ПО СЕТКЕ, С ПОМОЩЬЮ СТАТИСТИЧЕСКИХ ТЕСТОВ

До этого момента мы выбирали наилучшую модель по итогам перекрестной проверки в ходе поиска по сетке, не привлекая аппарат математической статистики. Вопрос, является ли разница между качеством наилучшей модели и модели-конкурента статистически значимой, остается открытым. Две модели могут иметь разные стоимости внедрения, разный уровень сложности интерпретации, но при этом между этими моделями с точки зрения качества прогнозов может отсутствовать статистически значимая разница.

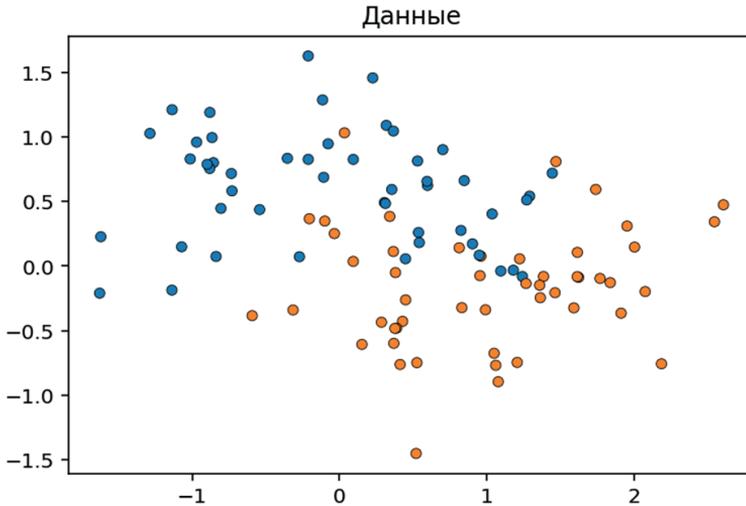
В этом разделе мы проиллюстрируем, как с помощью статистических тестов можно сравнить качество моделей, полученных в ходе поиска по сетке с помощью класса `GridSearchCV`. Мы в адаптированном виде изложим несколько подходов, представленных авторами библиотеки `scikit-learn` в этом материале: [https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_grid\\_search\\_stats.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_stats.html).

Начнем с симуляции набора данных `make_moons` (в котором идеальное разделение между классами является нелинейным), добавив в него умеренную степень шума. Точки данных будут принадлежать одному из двух возможных классов, предсказываемым по двум признакам. Симулируем по 50 наблюдений для каждого класса.

```
# импортируем библиотеки
import matplotlib.pyplot as plt
%matplotlib inline
```

```
%config InlineBackend.figure_format = 'retina'
import seaborn as sns

# импортируем make_moons
from sklearn.datasets import make_moons
X, y = make_moons(noise=0.352, random_state=1, n_samples=100)
sns.scatterplot(
    x=X[:, 0], y=X[:, 1], hue=y,
    marker='o', s=25, edgecolor='k', legend=False
).set_title("Данные")
plt.show()
```



### 7.20.1. Простое сравнение всех построенных моделей

Сравним качество классификаторов на основе метода опорных векторов SVC с разными значениями параметра `kernel`, чтобы выяснить, какое значение этого гиперпараметра предсказывает наши симулированные данные лучше всего. Мы будем оценивать качество моделей, используя класс `RepeatedStratifiedKFold`, повторяя 10 раз 10-блочную стратифицированную перекрестную проверку, используя различную рандомизацию данных при каждом повторе. Качество модели будет оцениваться с помощью функции `roc_auc_score()`. Таким образом, метрикой качества будет AUC-ROC.

```
# импортируем необходимые классы
from sklearn.model_selection import (GridSearchCV,
                                     RepeatedStratifiedKFold)
from sklearn.svm import SVC

# задаем сетку гиперпараметров
param_grid = [
    {'kernel': ['linear']},
    {'kernel': ['poly'], 'degree': [2, 3]},
    {'kernel': ['rbf']}
]
```

```

# задаем экземпляр класса SVC
svc = SVC(random_state=0)

# задаем экземпляр класса RepeatedStratifiedKFold,
# т.е. задаем повторную стратифицированную
# перекрестную проверку
cv = RepeatedStratifiedKFold(
    n_splits=10, n_repeats=10, random_state=0
)

# выполняем поиск по сетке
search = GridSearchCV(
    estimator=svc, param_grid=param_grid,
    scoring='roc_auc', cv=cv
)
search.fit(X, y);

```

Теперь можем посмотреть результаты нашего поиска по сетке, которые отсортированы по убыванию метрики, усредненной по тестовым блокам `mean_test_score`.

```

# импортируем pandas
import pandas as pd

# записываем результаты перекрестной
# проверки в виде датафрейма
results_df = pd.DataFrame(search.cv_results_)
results_df = results_df.sort_values(by=['rank_test_score'])
results_df = (
    results_df
    .set_index(results_df['params'].apply(
        lambda x: '_'.join(str(val) for val in x.values()))
    )
    .rename_axis('kernel')
)

# отбираем нужные столбцы
results_df[
    ['params', 'rank_test_score',
     'mean_test_score', 'std_test_score']
]

```

	params	rank_test_score	mean_test_score	std_test_score
<b>kernel</b>				
<b>rbf</b>	{'kernel': 'rbf'}	1	0.9400	0.079297
<b>linear</b>	{'kernel': 'linear'}	2	0.9300	0.077846
<b>3_poly</b>	{'degree': 3, 'kernel': 'poly'}	3	0.9044	0.098776
<b>2_poly</b>	{'degree': 2, 'kernel': 'poly'}	4	0.6852	0.169106

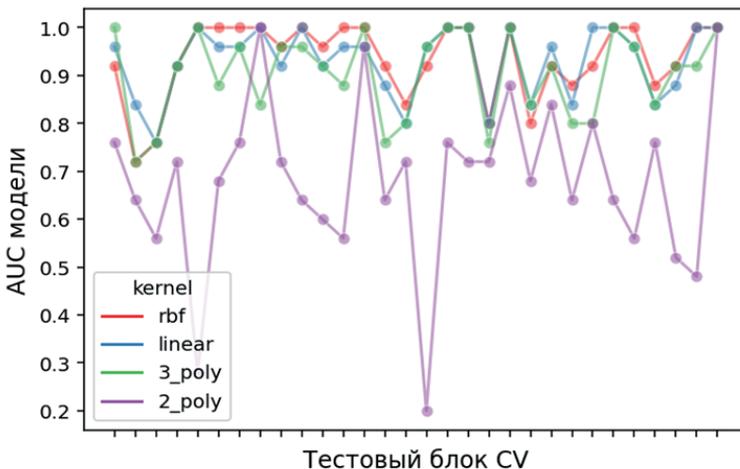
Видно, что лучше всего сработал классификатор, использующий гауссово ядро с радиальной базовой функцией 'rbf', за ним следует линейное ядро 'linear'. Обе оценки с полиномиальным ядром 'poly' сработали хуже: та, ко-

торая использовала полином второй степени, имеет гораздо меньшее качество, чем все остальные модели.

Обычно анализ на этом просто заканчивается, но половина истории упускается. Класс `GridSearchCV` не дает информацию о степени уверенности в найденных различиях между моделями. Мы не знаем, являются ли различия **статистически** значимыми. Чтобы оценить это, нужно провести статистический тест. В частности, чтобы сопоставить качество двух моделей, нужно статистически сравнить их оценки AUC (площадь под ROC-кривой). У нас 100 выборок (100 значений AUC) для каждой модели, поскольку мы повторили 10 раз 10-блочную перекрестную проверку.

Однако оценки моделей не являются независимыми: все модели оцениваются на **одних и тех же** 100 выборках, что увеличивает корреляцию между оценками качества моделей. Поскольку некоторые разбиения данных могут упростить или усложнить задачу разделения классов для всех моделей, оценки моделей будут варьировать соответствующим образом. Поэтому полезно проверить эффект разбиения данных, построив график оценки качества всех моделей в каждом блоке и вычислив корреляцию между моделями по блокам.

```
# создаем датафрейм с поблочными оценками модели,
# отсортированными по убыванию качества
model_scores = results_df.filter(regex=r'split\d*_test_score')
# выводим на график 30 примеров зависимости
# оценки AUC от блока перекрестной проверки
fig, ax = plt.subplots()
sns.lineplot(
    data=model_scores.transpose().iloc[:30],
    dashes=False, palette='Set1', marker='o', alpha=.5, ax=ax
)
ax.set_xlabel("Тестовый блок CV", size=12, labelpad=10)
ax.set_ylabel("AUC модели", size=12)
ax.tick_params(bottom=True, labelbottom=False)
plt.show()
# выводим корреляции между оценками AUC в разрезе блоков
print(f"Корреляция моделей:\n {model_scores.transpose().corr()}")
```



Корреляция моделей:

kernel	rbf	linear	3_poly	2_poly
kernel				
rbf	1.000000	0.882561	0.783392	0.351390
linear	0.882561	1.000000	0.746492	0.298688
3_poly	0.783392	0.746492	1.000000	0.355440
2_poly	0.351390	0.298688	0.355440	1.000000

Можно заметить, что качество моделей сильно зависит от блока перекрестной проверки.

Как следствие, если предположить независимость между выборками, будет недооцениваться дисперсия, вычисленная в наших статистических тестах, увеличивая количество ложноположительных ошибок (т.е. обнаруживая значительную разницу между моделями, когда таковой не существует) [1].

Для этих случаев было разработано несколько статистических тестов с поправкой на дисперсию. В этом примере мы покажем, как реализовать один из них (так называемый скорректированный  $t$ -критерий Надо–Бенжио) в соответствии с двумя разными статистическими подходами: частотным и байесовским.

## 7.20.2. Сравнение двух моделей: частотный подход

Сравнение можно начать с вопроса: «Является ли первая модель значительно лучше, чем вторая модель (когда используется сортировка по `mean_test_score`)?»

Чтобы ответить на этот вопрос, используя частотный подход, можно было бы запустить парный  $t$ -критерий и вычислить  $p$ -значение. В литературе по прогнозированию он также известен как тест Диболда–Мариано [5]. Многие варианты такого  $t$ -критерия были разработаны для учета «проблемы независимости выборок», описанной в предыдущем разделе. Мы воспользуемся одним из вариантов  $t$ -критерия, который, как было доказано, позволяет получить максимально воспроизводимые оценки качества (он оценивает, насколько схоже качество модели при ее оценке на разных случайных выборках из одного и того же набора данных), сохраняя при этом низкий уровень как ложноположительных, так и ложноотрицательных ошибок, – скорректированным  $t$ -критерием Надо и Бенжио [2], использующим 10-повторную 10-блочную перекрестную проверку [3].

Этот скорректированный парный  $t$ -критерий рассчитывается так:

$$t = \frac{\frac{1}{k \cdot r} \sum_{i=1}^k \sum_{j=1}^r X_{ij}}{\sqrt{\left( \frac{1}{k \cdot r} + \frac{n_{test}}{n_{train}} \right) \hat{\sigma}^2}},$$

где  $k$  – количество блоков,  $r$  – количество повторов в перекрестной проверке,  $x$  – разница в оценке качества моделей,  $n_{test}$  – количество наблюдений, используемых для тестирования,  $n_{train}$  – это количество наблюдений, используемых для обучения, и  $\hat{\sigma}^2$  представляет собой дисперсию наблюдаемых различий.

Давайте реализуем скорректированный правосторонний парный  $t$ -критерий, чтобы оценить, значительно ли качество первой модели лучше, чем у вто-

рой модели. Наша нулевая гипотеза состоит в том, что вторая модель работает по крайней мере так же хорошо, как первая.

```
import numpy as np
from scipy.stats import t
```

Нам нужно написать две функции: функцию `corrected_std()`, которая корректирует стандартное отклонение, используя метод Надо–Бенжио (по сути, вычисляет знаменатель вышеприведенной формулы), и функцию `compute_corrected_ttest()`, которая вычисляет правосторонний парный  $t$ -критерий с поправкой на дисперсию.

```
# пишем функцию, вычисляющую знаменатель формулы
def corrected_std(differences, n_train, n_test):
    """
    Функция корректирует стандартное отклонение,
    используя метод Надо–Бенжио.

    Параметры
    -----
    differences : ndarray формы (n_samples, 1)
        Вектор, содержащий разности оценок
        качества двух моделей.
    n_train : int
        Количество наблюдений в обучающей выборке.
    n_test : int
        Количество наблюдений в тестовой выборке.

    Возвращает
    -----
    corrected_std : int
        Стандартное отклонение набора различий
        с поправкой на дисперсию.
    """
    # kr = k x r, r-повторная k-блочная перекрестная проверка,
    # kr - это сколько раз оценивалась наша модель
    kr = len(differences)
    corrected_var = (np.var(differences, ddof=1) *
                    (1 / kr + n_test / n_train))
    corrected_std = np.sqrt(corrected_var)
    return corrected_std

# пишем функцию, вычисляющую правосторонний парный
# t-критерий с поправкой на дисперсию
def compute_corrected_ttest(differences, df, n_train, n_test):
    """
    Функция вычисляет правосторонний парный t-критерий
    с поправкой на дисперсию.

    Параметры
    -----
    differences : списокподобный объект формы (n_samples, 1)
        Вектор, содержащий разности оценок
        качества двух моделей.
    df : int
        Количество степеней свободы.
```

```

n_train : int
    Количество наблюдений в обучающей выборке.
n_test : int
    Количество наблюдений в тестовой выборке.

Возвращает
-----
t_stat : float
    t-статистику с поправкой на дисперсию.
p_val : float
    p-значение с поправкой на дисперсию.
"""
mean = np.mean(differences)
std = corrected_std(differences, n_train, n_test)
t_stat = mean / std
# правосторонний t-критерий
p_val = t.sf(np.abs(t_stat), df)
return t_stat, p_val

```

Сначала запишем оценки качества наилучшей модели.

```

# оценки наилучшей модели
model_1_scores = model_scores.iloc[0].values
model_1_scores

array([0.92, 0.72, 0.76, 0.92, 1. , 1. , 1. , 1. , 0.96, 1. , 0.96,
       1. , 1. , 0.92, 0.84, 0.92, 1. , 1. , 0.8 , 1. , 0.8 , 0.92,
       0.88, 0.92, 1. , 1. , 0.88, 0.92, 1. , 1. , 0.8 , 0.96, 0.84,
       1. , 1. , 1. , 1. , 0.96, 0.92, 1. , 1. , 0.92, 1. , 0.92,
       1. , 0.76, 1. , 1. , 1. , 1. , 1. , 0.84, 1. , 1. , 1. ,
       0.72, 0.92, 1. , 1. , 1. , 0.92, 0.92, 1. , 1. , 0.8 , 0.88,
       1. , 0.92, 0.96, 1. , 0.96, 0.92, 0.84, 0.92, 1. , 1. , 1. ,
       0.88, 0.92, 0.92, 1. , 0.92, 0.96, 1. , 0.72, 1. , 1. , 1. ,
       1. , 0.76, 0.96, 0.88, 1. , 0.72, 0.92, 1. , 0.96, 1. , 1. ,
       0.84])

```

Теперь запишем оценки второй по качеству модели.

```

# оценки второй по качеству модели
model_2_scores = model_scores.iloc[1].values
model_2_scores

array([0.96, 0.84, 0.76, 0.92, 1. , 0.96, 0.96, 1. , 0.92, 1. , 0.92,
       0.96, 0.96, 0.88, 0.8 , 0.96, 1. , 1. , 0.8 , 1. , 0.84, 0.96,
       0.84, 1. , 1. , 0.96, 0.84, 0.88, 1. , 1. , 0.84, 0.92, 0.84,
       1. , 0.96, 1. , 1. , 0.92, 0.88, 1. , 0.96, 0.92, 1. , 0.92,
       1. , 0.76, 1. , 0.92, 1. , 1. , 1. , 0.8 , 0.96, 0.96, 1. ,
       0.68, 0.96, 0.92, 1. , 1. , 0.92, 0.92, 1. , 0.8 , 0.84, 0.88,
       0.96, 0.92, 0.96, 1. , 0.96, 0.88, 0.8 , 0.84, 1. , 0.96, 1. ,
       0.92, 0.92, 0.92, 0.96, 1. , 0.92, 1. , 0.68, 1. , 1. , 1. ,
       1. , 0.76, 0.96, 0.92, 1. , 0.76, 0.92, 1. , 0.92, 0.92, 1. ,
       0.84])

```

Вычисляем разности между оценками.

```

# вычисляем разности между оценками
differences = model_1_scores - model_2_scores

```

Теперь нужно записать информацию о количестве разностей. По сути, речь идет о количестве полученных оценок качества или количестве тестовых выборок перекрестной проверки (ведь именно по ним мы получаем наши оценки качества).

```
# количество разностей, по сути количество полученных
# оценок качества или количество тестовых выборок
# перекрестной проверки
n = differences.shape[0]
n

100
```

Зная количество разностей, мы можем вычислить количество степеней свободы.

```
# количество степеней свободы
df = n - 1
df

99
```

Наконец, запишем информацию о количестве наблюдений, используемых для обучения, и о количестве наблюдений, используемых для тестирования.

```
# количество наблюдений, используемых для обучения
n_train = len(list(cv.split(X, y))[0][0])
n_train

90

# количество наблюдений, используемых для тестирования
n_test = len(list(cv.split(X, y))[0][1])
n_test

10
```

Сейчас мы можем вычислить значение скорректированного  $t$ -критерия и  $p$ -значение для него.

```
# вычисляем значение скорректированного
# t-критерия и p-значение для него
t_stat, p_val = compute_corrected_ttest(differences,
                                       df,
                                       n_train,
                                       n_test)

print(f"Значение скорректированного t-критерия: {t_stat:.3f}\n"
      f"p-значение для скорректированного t-критерия: {p_val:.3f}")
```

```
Значение скорректированного t-критерия: 0.750
p-значение для скорректированного t-критерия: 0.227
```

Можем сравнить значение скорректированного  $t$ -критерия и  $p$ -значение для него со значением нескорректированного  $t$ -критерия и соответствующим  $p$ -значением.

```
# вычисляем значение нескорректированного
# t-критерия и p-значения для него
t_stat_uncorrected = (
    np.mean(differences) / np.sqrt(np.var(differences, ddof=1) / n)
)
p_val_uncorrected = t.sf(np.abs(t_stat_uncorrected), df)

print(f"Значение нескорректированного "
      f"t-критерия: {t_stat_uncorrected:.3f}\n"
      f"p-значение для нескорректированного "
      f"t-критерия: {p_val_uncorrected:.3f}")
```

Значение нескорректированного t-критерия: 2.611  
 p-значение для нескорректированного t-критерия: 0.005

Используя общепринятый уровень значимости  $p = 0,05$ , мы наблюдаем, что нескорректированный t-критерий позволяет сделать вывод о том, что первая модель значительно лучше, чем вторая.

Напротив, при использовании скорректированного подхода эта разница не обнаруживается.

Однако в последнем случае частотный подход не позволяет сделать вывод, что первая и вторая модели имеют одинаковое качество. Если мы хотим сделать подобное утверждение, нам нужно использовать байесовский подход.

### 7.20.3. Сравнение двух моделей: байесовский подход

Мы можем воспользоваться байесовской оценкой для вычисления вероятности того, что первая модель лучше второй. Байесовская оценка даст распределение, которому подчиняется среднее значение разницы между метриками качества двух моделей.

Чтобы получить апостериорное распределение, нужно сперва определить априорное, моделирующее наши представления о распределении среднего значения перед анализом данных, и умножить его на функцию правдоподобия, которая вычисляет, насколько вероятны наши наблюдаемые различия с учетом значений, которые могло бы принять среднее различий.

Чтобы ответить на этот вопрос, мы могли бы выполнить разные варианты байесовского оценивания, но в этом примере реализуем подход, предложенный Бенаволи и его коллегами [4].

Один из способов определения нашего апостериорного распределения в замкнутой форме – выбрать априорное распределение, сопряженное с функцией правдоподобия. Бенаволи и его коллеги [4] пришли к выводу, что при сравнении качества двух классификаторов мы можем моделировать априорное распределение как нормальное гамма-распределение (с неизвестными средним значением и дисперсией), сопряженное с функцией правдоподобия для нормального распределения, чтобы, таким образом, выразить апостериорное распределение как нормальное распределение.

Отбросив дисперсию из этого нормального апостериорного распределения, мы можем определить апостериорное распределение среднего как t-распределение Стьюдента. В частности:

$$St\left(\mu; n-1, \bar{x}, \left(\frac{1}{n} + \frac{n_{test}}{n_{train}}\right)\sigma^2\right),$$

где  $n$  – общее количество наблюдений,  $\bar{x}$  – средняя разница оценок,  $n_{test}$  – количество наблюдений, использованных для тестирования,  $n_{train}$  – это количество наблюдений, используемых для обучения, а  $\sigma^2$  представляет собой дисперсию наблюдаемых различий.

Обратите внимание, что в байесовском подходе также используется дисперсия, скорректированная по методу Надо–Бенжио.

Давайте получим и визуализируем апостериорное распределение. Сначала с помощью класса  $t$  инициализируем случайную величину, подчиняющуюся  $t$ -распределению Стьюдента. Значением параметра сдвига будет среднее значение разностей метрик, а значением параметра масштаба будет скорректированная дисперсия разностей метрик, полученная с помощью функции `corrected_std()`.

*# получим и визуализируем апостериорное распределение*

*# инициализируем случайную величину*

```
t_post = t(
    df, loc=np.mean(differences),
    scale=corrected_std(differences, n_train, n_test)
)
```

С помощью метода `.ppf()` получаем значения, соответствующие вероятностям нашего распределения.

*# с помощью квантильной функции получаем значения,*

*# соответствующие вероятностям распределения*

```
x = np.linspace(t_post.ppf(0.001), t_post.ppf(0.999), 100)
```

Мы используем функцию процентной точки (`ppf` – percent point function), обратную к кумулятивной функции распределения (`cdf` – cumulative distribution function). Часто функцию процентной точки называют процентильной функцией, квантильной функцией. Из университетского курса статистики вспомним, что кумулятивная функция распределения вещественной случайной величины  $X$ , или просто функция распределения  $X$ , оцененная в  $x$ , – это вероятность того, что  $X$  примет значение, меньшее или равное значению  $x$ :

$$\Pr(X \leq x) = F(x).$$

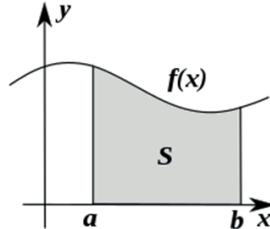
Функция принимает в качестве аргумента  $x$  и возвращает значения в интервале  $[0, 1]$  – вероятности, которые мы обозначим как  $p$ . Если кумулятивная функция распределения является строго возрастающей и непрерывной, то обратная функция кумулятивного распределения сообщит нам, какое значение  $x$ , являющееся уникальным вещественным числом, заставит  $F(x)$  вернуть значение  $p$ :

$$F^{-1}(p) = x.$$

Теперь с помощью метода `.pdf()` строим график функции плотности вероятности или функции плотности распределения (`pdf` – probability distribution function).

```
# строим график функции плотности вероятности
plt.plot(x, t_post.pdf(x))
```

Вспомним классический рисунок, иллюстрирующий полезное применение функции плотности вероятности.

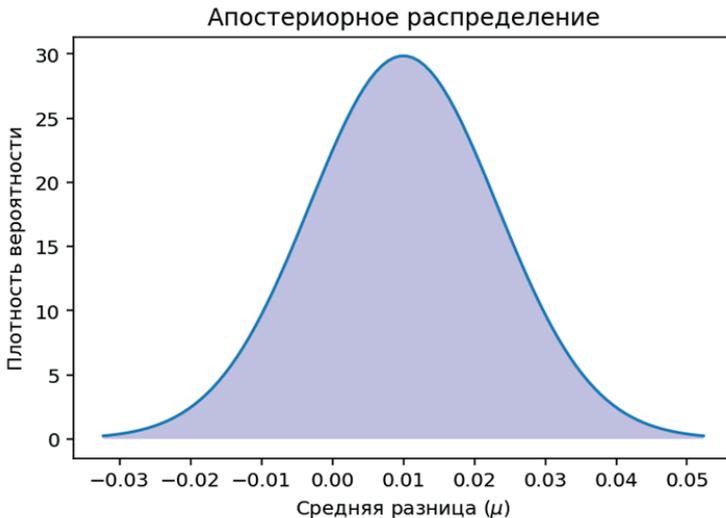


**Рис. 59** Вероятность  $P$  попадания случайной величины в интервал между  $a$  и  $b$  равна площади  $S$  под графиком функции плотности вероятности  $f(x)$

С помощью функции `plt.fill_between()` и метода `.pdf()` визуализируем синим цветом площадь под кривой плотности вероятности на интервале, полученном с помощью квантильной функции, далее подписываем оси и задаем заголовок.

```
# задаем диапазон значений и цену деления для оси X,
# диапазон определяем по первому и последнему значениям
# массива, полученного с помощью квантильной функции
plt.xticks(np.arange(-0.04, 0.06, 0.01))
# визуализируем площадь под кривой распределения
plt.fill_between(x, t_post.pdf(x), 0, facecolor='blue', alpha=.2)
plt.ylabel("Плотность вероятности")
plt.xlabel(r"Средняя разница ( $\mu$ )")
plt.title("Апостериорное распределение")

plt.show()
```



Можно вычислить вероятность того, что первая модель лучше, чем вторая, вычислив площадь под кривой апостериорного распределения от нуля до бесконечности. И наоборот, мы можем вычислить вероятность того, что вторая модель лучше первой, вычислив площадь под кривой от минус бесконечности до нуля. Здесь нужно воспользоваться кумулятивной функцией распределения нашей случайной величины.

```
# вычисляем вероятность того, что
# первая модель лучше второй
better_prob = 1 - t_post.cdf(0)

print(f"Вероятность того, что {model_scores.index[0]} "
      f"имеет более высокое качество, чем "
      f"{model_scores.index[1]}: {better_prob:.3f}")
print(f"Вероятность того, что {model_scores.index[1]} "
      f"имеет более высокое качество, чем "
      f"{model_scores.index[0]}: {1 - better_prob:.3f}")
```

```
Вероятность того, что gbf имеет более высокое качество, чем linear: 1.000
Вероятность того, что linear имеет более высокое качество, чем gbf: 0.000
```

В отличие от частотного подхода, мы смогли вычислить вероятность того, что одна модель лучше другой.

Обратите внимание, что мы получили те же результаты, что и при частотном подходе: первая модель лучше, чем вторая. Учитывая наш выбор априорных распределений, мы по сути выполняем одни и те же вычисления, но можем сделать разные выводы.

В отличие от частотного подхода, байесовский вывод не основан на статистической значимости, когда эффекты проверяются относительно «нуля». Байесовский подход предлагает вероятностный взгляд на параметры, что позволяет оценить связанную с ними неопределенность. Таким образом, вместо того чтобы делать вывод о наличии эффекта, когда он отличается от нуля, мы могли бы прийти к выводу, что вероятность выхода за пределы определенного диапазона, который можно рассматривать как «практическое отсутствие эффекта» (т.е. пренебрежимо малую величину), является достаточной. Этот диапазон называется областью практической эквивалентности (ROPE). Идея, лежащая в основе ROPE, состоит в том, чтобы позволить пользователю определить область вокруг нулевого значения, включающую значения, которые эквивалентны нулевому значению с практической точки зрения.

При построении моделей машинного обучения нас интересует определение вероятностей того, что сравниваемые модели будут иметь эквивалентное качество, где понятие «эквивалентности» определяется практическим способом. Наивный подход [4] состоял бы в том, чтобы определять классификаторы как практически эквивалентные, если они отличаются друг от друга менее чем на 1 % с точки зрения метрики качества. Но мы можем также определить эту практическую эквивалентность с точки зрения решаемой проблемы. Например, разница в метрике качества, равная 5 %, будет означать увеличение продаж на 1000 долларов, и мы считаем любую величину выше этого значения значимой для нашего бизнеса.

В этом примере мы определим область практической эквивалентности (Region of Practical Equivalence – ROPE) как  $[-0,01, 0,01]$ . Таким образом, мы будем считать две модели практически эквивалентными, если они отличаются по своему качеству менее чем на 1 %.

Чтобы вычислить вероятности того, что классификаторы будут практически эквивалентными, вычислим площадь под кривой апостериорного распределения на ROPE-интервале.

Задаем ROPE-интервал, вычисляем вероятность эквивалентности моделей с учетом заданного ROPE-интервала.

```
# задаем ROPE-интервал
rope_interval = [-0.01, 0.01]
# вычисляем вероятность эквивалентности моделей
# с учетом заданного ROPE-интервала
rope_prob = (t_post.cdf(rope_interval[1]) -
            t_post.cdf(rope_interval[0]))

print(f"Вероятность того, что {model_scores.index[0]} "
      f"и {model_scores.index[1]} практически "
      f"эквивалентны: {rope_prob:.3f}")
```

Вероятность того, что `gbf` и `linear` практически эквивалентны: 0.432

Теперь визуализируем апостериорное распределение на ROPE-интервале.

```
# получаем массив значений в заданном интервале
x_rope = np.linspace(rope_interval[0], rope_interval[1], 100)

# визуализируем апостериорное распределение
# в соответствии с ROPE
plt.plot(x, t_post.pdf(x))
plt.xticks(np.arange(-0.04, 0.06, 0.01))
plt.vlines([-0.01, 0.01], ymin=0,
           ymax=(np.max(t_post.pdf(x)) + 1))
plt.fill_between(x_rope, t_post.pdf(x_rope), 0,
                 facecolor='blue', alpha=.2)
plt.ylabel("Плотность вероятности")
plt.xlabel(r"Средняя разница ( $\mu$ )")
plt.title("Апостериорное распределение в соответствии с ROPE")
plt.show()
```

Как было предложено в [4], мы можем дополнительно интерпретировать эти вероятности, используя те же самые критерии, что и в частотном подходе: вероятность попадания в ROPE-интервал выше 95 % (уровень значимости 5 %)? В таком случае мы можем сделать вывод, что обе модели практически эквивалентны. Кроме того, байесовский подход к оценке также позволяет вычислить степень неопределенности нашей оценки разности метрик. Ее можно рассчитать с помощью байесовских доверительных интервалов (credible intervals). Для заданной вероятности они показывают диапазон значений, которые может принимать оцениваемая величина, в нашем случае речь идет о средней разнице между метриками качества моделей. Например, 50%-ный байесовский доверительный интервал  $[x, y]$  говорит, что существует 50%-ная

вероятность того, что истинная (средняя) разница в качестве моделей находится между  $x$  и  $y$ .

Давайте определим байесовские доверительные интервалы наших данных, используя 50 %, 75 % и 95 %.

```
# задаем пустой список, в который будем добавлять
# байесовские доверительные интервалы - списки
# из нижнего и верхнего значений
cred_intervals = []
# задаем список значений
intervals = [0.5, 0.75, 0.95]
# заполняем список
for interval in intervals:
    cred_interval = list(t_post.interval(interval))
    cred_intervals.append([interval, cred_interval[0], cred_interval[1]])
# на основе полученных результатов создаем датафрейм
cred_int_df = pd.DataFrame(
    cred_intervals,
    columns=['интервал', 'нижнее значение', 'верхнее значение']
).set_index('интервал')
cred_int_df
```

	нижнее значение	верхнее значение
интервал		
<b>0.50</b>	0.000977	0.019023
<b>0.75</b>	-0.005422	0.025422
<b>0.95</b>	-0.016445	0.036445

Как показано в таблице, существует 50%-ная вероятность того, что истинная средняя разница в качестве моделей будет между 0.000977 и 0.019023, 70%-ная вероятность того, что она будет между  $-0.005422$  и  $0.025422$ , и 95%-ная вероятность того, что она будет между  $-0.016445$  и  $0.036445$ .

## 7.20.4. Попарное сравнение всех моделей: частотный подход

Кроме того, нас могло бы заинтересовать сравнение качества всех наших моделей, оцененных с помощью класса `GridSearchCV`. В этом случае мы будем проводить статистический тест несколько раз, что приведет нас к проблеме множественных сравнений: [https://en.wikipedia.org/wiki/Multiple\\_comparisons\\_problem](https://en.wikipedia.org/wiki/Multiple_comparisons_problem).

Когда проводится много статистических тестов, возрастает вероятность совершения хотя бы одной ошибки I рода (ложное отклонение нулевой гипотезы) в любом из сравнений. Допустим, выполняем 5 тестов. Групповая вероятность ошибки (вероятность получения ложноположительного вывода, ошибки I рода) будет намного выше индивидуальной вероятности ошибки. Она вычисляется по формуле  $1 - (1 - \alpha_{\text{per comparison}})^m$ , где  $m$  – число выполняемых тестов (проверяемых гипотез). Например, если индивидуальная вероятность ошибки

( $\alpha_i$ ) по каждому тесту составляет 0,05, то групповая вероятность ошибки составит  $1 - (1 - 0,05)^5 = 0,226$ .

Если мы хотим, чтобы групповая вероятность ошибки не превышала определенный уровень значимости  $\alpha$  (например, 0,05), то, согласно методу Бонферрони, мы должны сравнить каждое из полученных  $p$ -значений не с  $\alpha$ , а с  $\alpha/m$ , где  $m$  – количество выполняемых тестов (количество проверяемых гипотез). Деление исходного уровня значимости  $\alpha$  на  $m$  и будет поправкой Бонферрони: [https://en.wikipedia.org/wiki/Bonferroni\\_correction](https://en.wikipedia.org/wiki/Bonferroni_correction). Таким образом, если мы, например, использовали желаемый уровень значимости 0,05, то теперь у нас будет 5 тестов,  $p$ -значения которых мы будем сравнивать с уровнем значимости  $0,05/5 = 0,01$ . Вместо деления изначально принятого уровня значимости на число проверяемых гипотез мы могли бы умножить каждое из исходных  $p$ -значений на это число, а затем сравнить такие скорректированные  $p$ -значения с  $\alpha$ . В ряде случаев при умножении исходных  $p$ -значений на число проверяемых гипотез результат может превысить 1. По определению, вероятность не может быть больше 1, и если это происходит, то получаемое значение просто приравнивают к 1.

Неявно поправка Бонферрони предполагает, что эти тестовые статистики независимы. Поправка Бонферрони уменьшает количество ложных отклонений, но также увеличивает количество случаев, когда нулевая гипотеза не отвергается, хотя на самом деле должна быть отклонена. Таким образом, корректировка Бонферрони может снизить мощность обнаружения важного эффекта.

Давайте выполним попарное сравнение всех моделей с помощью скорректированного  $t$ -критерия на основе частотного подхода.

```
# импортируем функции combinations() и factorial()
from itertools import combinations
from math import factorial

# количество сравнений
n_comparisons = (
    factorial(len(model_scores))
    / (factorial(2) * factorial(len(model_scores) - 2))
)

# список, в который будем записывать
# результаты парного t-теста
pairwise_t_test = []

# по каждой комбинации
for model_i, model_k in combinations(range(len(model_scores)), 2):
    # получаем оценки первой модели
    model_i_scores = model_scores.iloc[model_i].values
    # получаем оценки второй модели
    model_k_scores = model_scores.iloc[model_k].values
    # вычисляем разности
    differences = model_i_scores - model_k_scores
    # вычисляем скорректированный парный t-тест
    t_stat, p_val = compute_corrected_ttest(
        differences, df, n_train, n_test
    )
    # задаем поправку Бонферрони
    p_val *= n_comparisons
```

```

# поправка Бонферрони может выдавать p-значения большие, чем 1,
# оговариваем, что максимальное p-значение не должно превышать 1
p_val = 1 if p_val > 1 else p_val
# добавляем результаты скорректированного парного
# t-теста с поправкой Бонферрони в список
pairwise_t_test.append(
    [model_scores.index[model_i], model_scores.index[model_k],
     t_stat, p_val]
)

# на основе полученных результатов создаем датафрейм
pairwise_comp_df = pd.DataFrame(
    pairwise_t_test,
    columns=['model_1', 'model_2', 't_stat', 'p_val']
).round(3)
pairwise_comp_df

```

	model_1	model_2	t_stat	p_val
0	rbf	linear	0.750	1.000
1	rbf	3_poly	1.657	0.302
2	rbf	2_poly	4.565	0.000
3	linear	3_poly	1.111	0.807
4	linear	2_poly	4.276	0.000
5	3_poly	2_poly	3.851	0.001

Видно, что после корректировки множественных сравнений единственная модель, которая значительно отличается от других, – это '2\_poly'. Модель 'rbf', занимающая первое место по итогам поиска по сетке с помощью класса GridSearchCV, существенно не отличается от 'linear' или '3\_poly'.

## 7.20.5. Попарное сравнение всех моделей: байесовский подход

Давайте выполним попарное сравнение всех моделей с помощью скорректированного  $t$ -критерия теперь уже на основе байесовского подхода. Заметьте: при использовании байесовской оценки для сравнения нескольких моделей нам не нужно вводить поправку на множественные сравнения (по причинам, указанным в [4]).

```

# создаем пустой список, в который будем записывать результаты
# попарного сравнения согласно байесовскому подходу
pairwise_bayesian = []

# по каждой комбинации
for model_i, model_k in combinations(range(len(model_scores)), 2):
    # получаем оценки первой модели
    model_i_scores = model_scores.iloc[model_i].values
    # получаем оценки второй модели
    model_k_scores = model_scores.iloc[model_k].values
    # вычисляем разности

```

```

differences = model_i_scores - model_k_scores
# инициализируем случайную величину
t_post = t(
    df, loc=np.mean(differences),
    scale=corrected_std(differences, n_train, n_test)
)
# вычисляем вероятность того, что первая модель хуже второй
worse_prob = t_post.cdf(rope_interval[0])
# вычисляем вероятность того, что первая модель лучше второй
better_prob = 1 - t_post.cdf(rope_interval[1])
# вычисляем вероятность эквивалентности моделей
# с учетом заданного ROPE-интервала
rope_prob = t_post.cdf(rope_interval[1]) - t_post.cdf(rope_interval[0])
# добавляем результаты в список
pairwise_bayesian.append([worse_prob, better_prob, rope_prob])

# на основе полученных результатов создаем датафрейм
pairwise_bayesian_df = (pd.DataFrame(
    pairwise_bayesian,
    columns=['worse_prob', 'better_prob', 'rope_prob']
)).round(3)

# присоединяем к датафрейму с результатами попарного сравнения
# моделей на основе частотного подхода датафрейм с результатами
# попарного сравнения моделей на основе байесовского подхода
pairwise_comp_df = pairwise_comp_df.join(pairwise_bayesian_df)
pairwise_comp_df

```

	model_1	model_2	t_stat	p_val	worse_prob	better_prob	rope_prob
0	rbf	linear	0.750	1.000	0.068	0.500	0.432
1	rbf	3_poly	1.657	0.302	0.018	0.882	0.100
2	rbf	2_poly	4.565	0.000	0.000	1.000	0.000
3	linear	3_poly	1.111	0.807	0.063	0.750	0.187
4	linear	2_poly	4.276	0.000	0.000	1.000	0.000
5	3_poly	2_poly	3.851	0.001	0.000	1.000	0.000

Используя байесовский подход, можно вычислить вероятность того, что модель работает лучше, хуже или практически эквивалентна другой модели.

Вероятность того, что модель 'rbf', занявшая первое место по результатам GridSearchCV, хуже, чем 'linear', составляет 6.8 %. Вероятность того, что модель 'rbf' хуже, чем '3\_poly', составляет 1.8 %. Вероятность того, что 'rbf' и 'linear' практически эквивалентны, составляет 43 %, в то время как вероятность того, что 'rbf' и '3\_poly' практически эквивалентны, составляет всего 10 %.

Подобно выводам, полученным с использованием частотного подхода, все модели имеют 100%-ную вероятность быть лучше, чем '2\_poly', и ни одна из них не имеет практической эквивалентности с последней по качеству.

## 7.20.6. Итоговые выводы

Можно сделать некоторые итоговые выводы.

- Небольшие различия в оценках качества могут легко оказаться просто случайностью, но не потому, что одна модель систематически предсказывает лучше, чем другая. Как показано в этом примере, статистика может сказать, насколько это вероятно.
- При статистическом сравнении качества двух моделей, оцененных в ходе поиска по сетке с помощью класса `GridSearchCV`, необходимо скорректировать вычисленную дисперсию, которая может быть недооценена, поскольку оценки моделей не являются независимыми друг от друга.
- Частотный подход, в котором используется парный  $t$ -критерий (с поправкой на дисперсию), позволяет выяснить, лучше ли качество одной модели по сравнению с другой, со степенью уверенности выше случайного угадывания.
- Байесовский подход позволяет вычислить вероятности того, что одна модель будет лучше, хуже или практически эквивалентна другой. Он также позволяет сказать, какова уверенность в том, что истинная разница в качестве наших моделей попадает в определенный диапазон значений.
- Если несколько моделей сравниваются статистически, требуется корректировка множественных сравнений при использовании частотного подхода.

[1] *Dietterich T. G.* (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural computation*, 10 (7).

[2] *Nadeau C., & Bengio Y.* (2000). Inference for the generalization error. In *Advances in neural information processing systems*.

[3] *Bouckaert R. R., & Frank E.* (2004). Evaluating the replicability of significance tests for comparing learning algorithms. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*.

[4] *Benavoli A., Corani G., Demšar J., & Zaffalon M.* (2017). Time for a change: a tutorial for comparing multiple classifiers through Bayesian analysis. *The Journal of Machine Learning Research*, 18 (1).

[5] *Diebold F. X. & Mariano R. S.* (1995). Comparing predictive accuracy *Journal of Business & economic statistics*, 20 (1), 134–144.

## 7.21. РАЗБИЕНИЕ НА ОБУЧАЮЩУЮ, ПРОВЕРОЧНУЮ И ТЕСТОВУЮ ВЫБОРКИ С УЧЕТОМ ВРЕМЕННОЙ СТРУКТУРЫ ДЛЯ ВАЛИДАЦИИ ВРЕМЕННЫХ РЯДОВ

До этого момента мы выполняли валидацию для обычных данных. В наборе данных, представляющем временной ряд, наблюдения расположены в хронологическом порядке, т.е. присутствует ось времени. Теперь подумаем, что произойдет, если применить случайное разбиение на обучающую и тестовую выборки. В наши обучающую и тестовую выборки попадут наблюдения, относящиеся к разным моментам времени, как к более ранним, так и к более поздним. Получается, что при обучении модель получит «подсказки из будущего», а сама тестовая выборка будет неправдоподобной, потому что в ней перемешаются данные из более ранних и более поздних моментов времени, тогда как в реальности новые данные всегда будут относиться к более позднему периоду времени. Поэтому для валидации временного ряда нужно применять разбиение на обучающую и тестовую выборки с учетом временной структуры данных. Размер тестовой выборки определяется горизонтом прогнозирования.

Давайте импортируем необходимые библиотеки, классы и функции, а также загрузим игрушечные данные, содержащие временной ряд.

```
# импортируем необходимые библиотеки
import re
import bottleneck as bn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

# импортируем необходимые классы
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Ridge
from catboost import CatBoostRegressor

# импортируем из модуля sklearn.model_selection
# функцию train_test_split()
from sklearn.model_selection import train_test_split

# импортируем класс TimeSeriesSplit, реализующий
# стратегии перекрестной проверки
# для временных рядов
from sklearn.model_selection import TimeSeriesSplit

# импортируем функцию mean_squared_error()
from sklearn.metrics import mean_squared_error

# импортируем модуль warnings
import warnings

# записываем CSV-файл в объект DataFrame
flat_data = pd.read_csv('Data/Flats_timeseries.csv',
                       encoding='UTF-8',
                       sep=';')
```

```
# выводим первые 5 наблюдений датафрейма
flat_data.head()
```

	Date_Create	Value_abs
0	22.08.2016	1450000
1	23.08.2016	1650000
2	24.08.2016	2250000
3	25.08.2016	1960000
4	26.08.2016	1950000

Игрушечный набор содержит данные о продажах квартир, собранные агентством недвижимости. По каждому наблюдению (сделке) фиксируются следующие переменные (характеристики):

- признак даты *Дата* [*Data\_Create*];
- количественная зависимая переменная *Продажи в рублях* [*Value\_abs*].

Задача заключается в том, чтобы разработать модель, прогнозирующую продажи квартир.

Нам нужно сначала отсортировать данные по дате (от самой ранней к самой поздней) и разбить их на обучающую и тестовую выборки. Вспомним, что размер тестовой выборки определяется горизонтом прогнозирования. Если нужно прогнозировать продажи на 7 дней вперед, горизонт прогнозирования – 7 дней, тестовая выборка должна состоять из 7 более поздних наблюдений (более поздних моментов времени). Если нужно прогнозировать продажи на 30 дней вперед, горизонт прогнозирования – 30 дней, тестовая выборка должна состоять из 30 более поздних наблюдений (более поздних моментов времени). Допустим, в нашем случае горизонт прогнозирования составляет 6 дней. Мы разбиваем отсортированные по дате данные на обучающую и тестовую выборки с помощью функции `train_test_split()`, отключив перемешивание данных (для параметра `shuffle` задаем значение `False`) и задав для параметра `test_size` значение 6. Параллельно выполним некоторую предобработку данных: столбцу `Date_Create` присвоим тип `datetime` (по сути, распознаем строковые значения как даты, выполняя парсинг), сформируем на основе столбца `Date_Create` некоторые календарные признаки – день недели и индикатор выходного дня, и, наконец, назначим столбец `Date_Create` индексом. Вообще говоря, старайтесь при чтении данных в датафрейм сразу парсить столбец с датами и на его основе формировать индекс. Затем на основе индекса можно создать полезные календарные признаки. Разумеется, можно создать календарные признаки и на основе столбца с распарсенными датами, однако часто перед моделированием начинающие специалисты забывают удалить столбец с датами, который не используется в моделировании (обычно если мы хотим включить информацию о датах в модель, мы переводим их в UNIX-время, но сами даты не рассматриваются в качестве признака), и возникают ошибки при построении модели.

```
# преобразовываем в формат даты
```

```

flat_data['Date_Create'] = pd.to_datetime(
    flat_data['Date_Create'], format='%d.%m.%Y')
# сортируем данные по дате сделки
# (от самой ранней к самой поздней)
flat_data = flat_data.sort_values(
    by='Date_Create', ascending=1)
# создадим некоторые календарные признаки
flat_data['dayofweek'] = pd.DatetimeIndex(
    flat_data['Date_Create']).dayofweek
flat_data['weekend'] = np.where(
    flat_data['dayofweek'].isin([5, 6]), 1, 0)
# сделаем столбец Date_Create индексом
flat_data = flat_data.set_index('Date_Create')
flat_data

```

**Value\_abs dayofweek weekend**

**Date\_Create**

Date_Create	Value_abs	dayofweek	weekend
2016-08-22	1450000	0	0
2016-08-23	1650000	1	0
2016-08-24	2250000	2	0
2016-08-25	1960000	3	0
2016-08-26	1950000	4	0
2016-08-27	1700000	5	1
2016-08-28	1550000	6	1
2016-08-29	2330000	0	0
2016-08-30	1900000	1	0
2016-08-31	3850000	2	0
2016-09-01	5450000	3	0
2016-09-02	3300000	4	0
2016-09-03	4950000	5	1
2016-09-04	3500000	6	1
2016-09-05	2300000	0	0
2016-09-06	1850000	1	0
2016-09-07	2900000	2	0
2016-09-08	2400000	3	0
2016-09-09	3100000	4	0
2016-09-10	2550000	5	1

```

# выполняем разбиение на обучающую и тестовую выборки,
# учитывающее временную структуру
X_flat_tr, X_flat_tst, y_flat_tr, y_flat_tst = train_test_split(
    flat_data.drop('Value_abs', axis=1),
    flat_data['Value_abs'],
    test_size=6,
    shuffle=False)

```

# Смотрим обучающую выборку

X\_flat\_tr

	dayofweek	weekend
<b>Date_Create</b>		
2016-08-22	0	0
2016-08-23	1	0
2016-08-24	2	0
2016-08-25	3	0
2016-08-26	4	0
2016-08-27	5	1
2016-08-28	6	1
2016-08-29	0	0
2016-08-30	1	0
2016-08-31	2	0
2016-09-01	3	0
2016-09-02	4	0
2016-09-03	5	1
2016-09-04	6	1

# Смотрим тестовую выборку

X\_flat\_tst

	dayofweek	weekend
<b>Date_Create</b>		
2016-09-05	0	0
2016-09-06	1	0
2016-09-07	2	0
2016-09-08	3	0
2016-09-09	4	0
2016-09-10	5	1

Видим, что все наблюдения в тестовой выборке относятся к более позднему периоду.

Давайте загрузим набор данных о ежемесячных продажах вина в Австралии с января 1980 года по август 1994 года включительно. Мы хотим сразу получить датафрейм с индексом, состоящим из правильно спарсенных дат. С помощью параметра `index_col` указываем столбец с датами в качестве индекса, а для параметра `parse_dates` указываем столбец с датами, чтобы выполнить парсинг дат сразу при создании датафрейма. Нередко, когда формат дат является сложным, парсинг может быть выполнен неверно. Тогда с помощью парамет-

ра `date_parser` и функции `pd.to_datetime()` мы можем задать точный формат даты для парсинга.

```
# загружаем набор о ежемесячных продажах вина в Австралии
wine_df = pd.read_csv('Data/monthly_australian_wine_sales.csv',
                     index_col='month',
                     parse_dates=['month'],
                     date_parser=lambda col: pd.to_datetime(
                         col, format='%Y-%m-%d'))

wine_df.head()
```

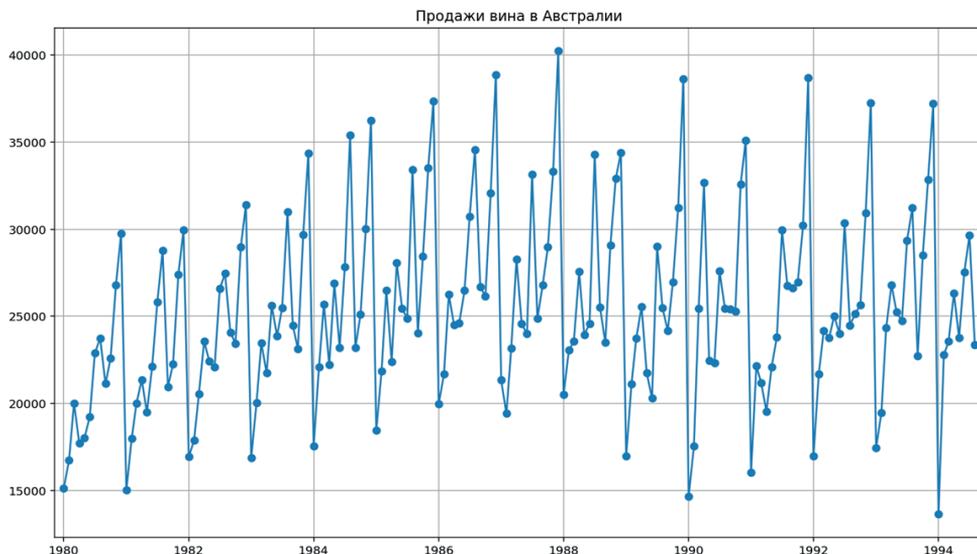
sales	
month	
1980-01-01	15136
1980-02-01	16733
1980-03-01	20016
1980-04-01	17708
1980-05-01	18019

Задача заключается в том, чтобы прогнозировать продажи вина на 6 месяцев вперед. Таким образом, горизонт прогнозирования составляет у нас 6 месяцев. Метрика качества – RMSE. Данные уже отсортированы по дате.

Визуализируем временной ряд.

```
# визуализируем временной ряд

# задаем размер графика
fig, ax = plt.subplots(figsize=(14, 8))
# строим график
ax.plot(wine_df['sales'],
        marker='o')
# задаем заголовок графика
ax.set_title("Продажи вина в Австралии")
# задаем начало оси x с отступом
ax.margins(x=0.01)
# задаем координатную сетку
ax.grid();
```



Помимо традиционных признаков на основе даты и времени (порядковый номер дня недели, порядковый номер месяца и т.д.) для прогнозирования временных рядов используют лаги, разности и скользящие статистики.

Лаговая переменная – это переменная, значение которой мы берем не за текущий, а за отстоящий от него на определенное расстояние предыдущий момент времени. Проще говоря, это переменные, взятые с запаздыванием во времени. Величина интервала запаздывания как раз и называется *лагом*. Такие переменные мы часто будем записывать так: `lag 10` или `lag = 10`, что будет эквивалентно записи `t = 10`.

Лаги можно создать с помощью метода `.shift()`, указав с помощью параметра `periods` порядок лага или количество периодов – моментов, на которые значение должно отстоять от текущего.

Параметр `periods` метода `.shift()` часто опускают.

```
data['Lag3'] = data['sales'].shift(periods=3)
```

или

```
data['Lag3'] = data['sales'].shift(3)
```

Допустим, у нас есть данные продаж за 12 дней и мы с помощью метода `.shift()` создали лаги с запаздыванием на 3, 5, 8, 12 и 13 дней.

```
# создаем набор продаж
```

```
example = pd.DataFrame(
```

```
    {'sales': [2400, 2800, 2500, 2890, 2610, 2500,
               2750, 2700, 2250, 2350, 2550, 3000]},
```

```
    index=pd.date_range(start='2018-01-09', periods=12)
```

```
)
```

```
# лаг с запаздыванием на 3 дня
example['Lag3'] = example['sales'].shift(periods=3)
# лаг с запаздыванием на 4 дня
example['Lag4'] = example['sales'].shift(periods=4)
# лаг с запаздыванием на 5 дней
example['Lag5'] = example['sales'].shift(periods=5)
# лаг с запаздыванием на 6 дней
example['Lag6'] = example['sales'].shift(periods=6)
# лаг равен длине набора
example['Lag12'] = example['sales'].shift(periods=12)
# лаг превышает длину набора
example['Lag13'] = example['sales'].shift(periods=13)
# смотрим результаты
example
```

	sales	<i>t</i> Lag3	<i>t</i> Lag4	<i>t</i> Lag5	<i>t</i> Lag6	<i>t</i> Lag12	<i>t</i> Lag13
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN	NaN	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0	NaN	NaN
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0	NaN	NaN
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0	NaN	NaN
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0	NaN	NaN
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0	NaN	NaN
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0	NaN	NaN

Порядок лага, настраиваемый с помощью параметра `periods`, не должен быть равен или превышать длину набора данных, на основе которого он создается, в противном случае он будет состоять из одних пропусков, что мы и видим для `Lag12` и `Lag13`. Чем больше порядок лагов, тем меньше наблюдений используется при его вычислении.

Отметим, что лаги будут полезными признаками при коротких горизонтах прогнозирования (до 30 дней включительно). Чем короче горизонт прогнозирования, тем эффективнее могут быть лаги в качестве признаков.

Лаги у нас были созданы до разбиения на обучающую и тестовую выборки. Разобьем наш набор данных так, чтобы в тестовую выборку попало 4 последних наблюдения. Допустим, нам надо будет прогнозировать продажи на 4 дня вперед, таким образом, горизонт прогнозирования – 4 дня.

```
# удалим переменные Lag12 и Lag13
example.drop(['Lag12', 'Lag13'], axis=1, inplace=True)
# разбиваем набор на обучающую и тестовую выборки
train, test = example[0:len(example)-4], example[len(example)-4:]

# смотрим обучающую выборку
train
```

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0

```
# смотрим тестовую выборку
test
```

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0

Обратите внимание: чем выше порядок лага, тем более ранние наблюдения обучающей выборки попадают в тестовую выборку.

Однако не все лаги в тесте используют наблюдения обучающего набора, лаг 3 «залез» в тест, что является подсматриванием в будущее (обведено красным овалом), которое нам неизвестно.

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	2400.0
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0

	sales	Lag3	Lag4	Lag5	Lag6
2018-01-17	2250	2500.0	2610.0	2890.0	2500.0
2018-01-18	2350	2750.0	2500.0	2610.0	2890.0
2018-01-19	2550	2700.0	2750.0	2500.0	2610.0
2018-01-20	3000	2250.0	2700.0	2750.0	2500.0

Рис. 60 При вычислении лага 3 произошла «протечка»

Одна из самых частых ошибок, связанных с подготовкой лаговой переменной, связана с тем, что лаговая переменная для тестового набора берется из самого тестового набора. Таким образом, необходимо создавать лаговые переменные так, чтобы они не проникали в тестовый набор. Проиллюстрируем на нашем примере.

	sales	Lag3	Lag4	Lag5	Lag6	
обучающая выборка	2018-01-09	2400	NaN	NaN	NaN	NaN
	2018-01-10	2800	NaN	NaN	NaN	NaN
	2018-01-11	2500	NaN	NaN	NaN	NaN
	2018-01-12	2890	2400.0	NaN	NaN	NaN
	2018-01-13	2610	2800.0	2400.0	NaN	NaN
	2018-01-14	2500	2500.0	2800.0	2400.0	NaN
	2018-01-15	2750	2890.0	2500.0	2800.0	2400.0
2018-01-16	2700	2610.0	2890.0	2500.0	2800.0	
тестовая выборка	2018-01-17	2250	2500.0	2610.0	2890.0	2500.0
	2018-01-18	2350	2750.0	2500.0	2610.0	2890.0
	2018-01-19	2550	2700.0	2750.0	2500.0	2610.0
	2018-01-20	3000	2250.0	2700.0	2750.0	2500.0

Lag 3 «проникает» в тест, мы берем информацию из тестовой выборки

Lag 4 и выше не «проникают» в тест, мы берем информацию из обучающей выборки

**Рис. 61** Лаги, у которых порядок меньше горизонта прогнозирования, «залазят» в тест

Здесь легко увидеть, что Lag 3 «проникает» в тест, мы берем информацию из тестового набора. Поэтому лаги вида  $L_{t-k}$  лучше создавать так, чтобы  $k$  был равен или превышал горизонт прогнозирования. Впрочем, в рамках соревновательного (непромышленного) подхода допускается создание лагов, у которых порядок будет меньше горизонта прогнозирования, однако тогда значения зависимой переменной в тестовой выборке нужно заменить на значения NaN. Если лаг залезет в тест, в тесте появятся значения NaN. В таком случае чем больше горизонт прогнозирования будет превышать порядок лага, тем больше пропусков будет в тесте. Очевидный недостаток лагов меньше горизонта прогнозирования заключается в том, что появляется дополнительная задача импутации пропусков в тесте.

На практике для избежания протечек при вычислении лагов (а также скользящих и расширяющихся статистик) поступают двумя способами:

- значения зависимой переменной в наблюдениях исходного набора, которые будут соответствовать будущей тестовой выборке (набору новых данных), заменяют значениями NaN;
- берем обучающую выборку и удлиняем ее на длину горизонта прогнозирования, зависимая переменная в наблюдениях, соответствующих новым временным меткам (т.е. в тестовой выборке / наборе новых данных), получает значения NaN.

В обоих случаях мы формируем защиту от протечек при вычислении лагов в тестовой выборке / наборе новых данных.

Давайте продемонстрируем первый способ.

Заново создадим данные продаж. Значения в наблюдениях, которые будут приходиться на тестовую выборку (последние 4 наблюдения исходного набора), заменяем на значения NaN. Затем формируем лаги порядка 1, 2, 3, 4, 5.

```
# создаем набор продаж
example = pd.DataFrame(
    {'sales': [2400, 2800, 2500, 2890, 2610, 2500,
              2750, 2700, 2250, 2350, 2550, 3000]},
    index=pd.date_range(start='2018-01-09', periods=12)
)

# значения в наблюдениях, которые будут приходиться на тест
# (последние 4 наблюдения), заменяем на значения NaN
HORIZON = 4
example['sales'].iloc[-HORIZON:] = np.NaN

# создаем лаги, лаги меньше горизонта прогнозирования
# получают пропуски в наблюдениях, приходящихся на тест,
# при этом чем меньше порядок лага горизонта
# прогнозирования, тем больше пропусков в тесте
# лаг с запаздыванием на 1 день
example['Lag1'] = example['sales'].shift(1)
# лаг с запаздыванием на 2 дня
example['Lag2'] = example['sales'].shift(2)
# лаг с запаздыванием на 3 дня
example['Lag3'] = example['sales'].shift(3)
# лаг с запаздыванием на 4 дня
example['Lag4'] = example['sales'].shift(4)
# лаг с запаздыванием на 5 дней
example['Lag5'] = example['sales'].shift(5)
example
```

	sales	Lag1	Lag2	Lag3	Lag4	Lag5
2018-01-09	2400.0	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800.0	2400.0	NaN	NaN	NaN	NaN
2018-01-11	2500.0	2800.0	2400.0	NaN	NaN	NaN
2018-01-12	2890.0	2500.0	2800.0	2400.0	NaN	NaN
2018-01-13	2610.0	2890.0	2500.0	2800.0	2400.0	NaN
2018-01-14	2500.0	2610.0	2890.0	2500.0	2800.0	2400.0
2018-01-15	2750.0	2500.0	2610.0	2890.0	2500.0	2800.0
2018-01-16	2700.0	2750.0	2500.0	2610.0	2890.0	2500.0
2018-01-17	NaN	2700.0	2750.0	2500.0	2610.0	2890.0
2018-01-18	NaN	NaN	2700.0	2750.0	2500.0	2610.0
2018-01-19	NaN	NaN	NaN	2700.0	2750.0	2500.0
2018-01-20	NaN	NaN	NaN	NaN	2700.0	2750.0

Наша  
тестовая выборка

Видим, что лаги, у которых порядок меньше горизонта прогнозирования (и они «залезают» в тестовую выборку – последние 4 наблюдения исходного набора), получают пропуски в наблюдениях, приходящихся на тестовую выборку, при этом чем больше горизонт прогнозирования превышает порядок лага, тем больше пропусков будет в тестовой выборке. Так срабатывает защита от протечек.

Теперь проиллюстрируем второй способ. Удалим ранее созданные лаги и сформируем обучающую выборку – первые 8 наблюдений исходного набора.

```
# удалим лаги по паттерну 'Lag'
example.drop(example.filter(regex='Lag').columns, axis=1,
             inplace=True)
# создаем обучающую выборку - первые
# 8 наблюдений исходного набора
train = example.iloc[:HORIZON]
train
```

	sales
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

Пишем функцию, которая берет нашу обучающую выборку и удлиняет ее на длину горизонта прогнозирования, зависящая переменная в наблюдениях, соответствующих новым временным меткам (т. е. в тестовой выборке / наборе новых данных), получает значения NaN.

```
# пишем функцию для создания лагов
# в обучающей и тестовой выборках
def calculate_lags(train, target, horizon, lags_range,
                  freq='D', aggregate=False):
    """
    Создает лаги в обучающей и тестовой выборках.

    Параметры
    -----
    train:
        Обучающий набор.
    target:
        Название зависимой переменной.
    horizon:
        Горизонт прогнозирования.
    lags_range:
        Диапазон значений порядка лагов.
    freq: str, значение по умолчанию 'D'
        Частота временного ряда.
    aggregate: bool, значение по умолчанию False
        Вычисляет агрегированный лаг.
    """
    if min(lags_range) < horizon:
        warnings.warn(f"\nКоличество периодов для лагов нужно задавать\n"
                    f"равным или больше горизонта прогнозирования")

    if pd.__version__ >= '1.4':
        # создаем метки времени для горизонта
        future_dates = pd.date_range(start=train.index[-1],
                                    periods=horizon + 1,
                                    freq=freq,
                                    inclusive='right')
    else:
        # создаем метки времени для горизонта
        future_dates = pd.date_range(start=train.index[-1],
                                    periods=horizon + 1,
                                    freq=freq,
                                    closed='right')

    # формируем новый удлиненный индекс
    new_index = train.index.append(future_dates)
    # выполняем переиндексацию
    new_df = train.reindex(new_index)
    # создаем лаги
    for i in lags_range:
        new_df[f"Lag_{i}"] = new_df[target].shift(i)

    if aggregate and min(lags_range) >= horizon:
        # вычисляем агрегированный лаг
        new_df['Agg_Lag'] = new_df[new_df.filter(
            regex='Lag').columns].mean(axis=1)
```

```

train = new_df.iloc[:-horizon]
test = new_df.iloc[-horizon:]

return train, test

```

Применяем нашу функцию, создаем лаги порядка 3, 4 и 5.

```

# создаем лаги для обучающей и тестовой выборки
train, test = calculate_lags(
    train, target='sales', horizon=4,
    lags_range=range(3, 6), freq='D')

```

Смотрим лаги в обучающей и тестовой выборках. Видим, что Lag\_3 получает пропуск в тестовой выборке, когда пытается использовать информацию тестовой выборки.

```

# смотрим лаги в обучающей выборке
train

```

	sales	Lag_3	Lag_4	Lag_5
2018-01-09	2400.0	NaN	NaN	NaN
2018-01-10	2800.0	NaN	NaN	NaN
2018-01-11	2500.0	NaN	NaN	NaN
2018-01-12	2890.0	2400.0	NaN	NaN
2018-01-13	2610.0	2800.0	2400.0	NaN
2018-01-14	2500.0	2500.0	2800.0	2400.0
2018-01-15	2750.0	2890.0	2500.0	2800.0
2018-01-16	2700.0	2610.0	2890.0	2500.0

```

# смотрим лаги в тестовой выборке
test

```

	sales	Lag_3	Lag_4	Lag_5
2018-01-17	NaN	2500.0	2610.0	2890.0
2018-01-18	NaN	2750.0	2500.0	2610.0
2018-01-19	NaN	2700.0	2750.0	2500.0
2018-01-20	NaN	NaN	2700.0	2750.0

Полезно добавлять усредненные лаги (их еще называют агрегированными). Например,  $(L_{t-7} + L_{t-14} + L_{t-21})/3$  или  $(L_{t-1} + L_{t-2} + \dots + L_{t-7})/7$ . Такие переменные используются в библиотеке Greykite. Однако по-прежнему помните, что в агрегации могут участвовать лаги, не использующие информацию теста (т. е. лаги, у которых порядок равен горизонту или превышает его).

Кроме того, можно создать разности между значениями соответствующего лага. Например, речь может идти о разнице между продажами на прошлой не-

деле и продажами на позапрошлой неделе или о разнице между продажами на прошлой неделе и продажами четырьмя неделями ранее.

Заново создадим данные продаж, сгенерируем лаги 3, 4, 5, а на их основе – разности в 1 и 2 периода.

```
# создаем набор продаж
example = pd.DataFrame(
    {'sales': [2400, 2800, 2500, 2890, 2610, 2500,
              2750, 2700, 2250, 2350, 2550, 3000]},
    index=pd.date_range(start='2018-01-09', periods=12)
)
```

```
# создаем лаги
data['Lag3'] = data['sales'].shift(3)
data['Lag4'] = data['sales'].shift(4)
data['Lag5'] = data['sales'].shift(5)
```

```
# создаем разности на основе лагов
data['Diff_on_Lag3'] = data['Lag3'].diff()
data['Diff_on_Lag4'] = data['Lag4'].diff()
data['Diff_on_Lag5'] = data['Lag5'].diff()
data['Diff2_on_Lag4'] = data['Lag4'].diff(2)
data['Diff2_on_Lag5'] = data['Lag5'].diff(2)
data
```

	sales	Lag3	Lag4	Lag5	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610	2800.0	2400.0	NaN	400.0	NaN	NaN	NaN	NaN
2018-01-14	2500	2500.0	2800.0	2400.0	-300.0	400.0	NaN	NaN	NaN
2018-01-15	2750	2890.0	2500.0	2800.0	390.0	-300.0	400.0	100.0	NaN
2018-01-16	2700	2610.0	2890.0	2500.0	-280.0	390.0	-300.0	90.0	100.0
2018-01-17	2250	2500.0	2610.0	2890.0	-110.0	-280.0	390.0	110.0	90.0
2018-01-18	2350	2750.0	2500.0	2610.0	250.0	-110.0	-280.0	-390.0	110.0
2018-01-19	2550	2700.0	2750.0	2500.0	-50.0	250.0	-110.0	140.0	-390.0
2018-01-20	3000	2250.0	2700.0	2750.0	-450.0	-50.0	250.0	200.0	140.0

Разобьем наш набор данных так, чтобы в тестовую выборку попало 4 последних наблюдения. Таким образом, горизонт прогнозирования – 4 дня.

```
# разбиваем набор на обучающую и тестовую выборки
train, test = example[0:len(example)-4], example[len(example)-4:]
```

```
# смотрим обучающую выборку
train
```

	sales	Lag3	Lag4	Lag5	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
2018-01-09	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890.0	2400.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-13	2610.0	2800.0	2400.0	NaN	400.0	NaN	NaN	NaN	NaN
2018-01-14	2500.0	2500.0	2800.0	2400.0	-300.0	400.0	NaN	NaN	NaN
2018-01-15	2750.0	2890.0	2500.0	2800.0	390.0	-300.0	400.0	100.0	NaN
2018-01-16	2700.0	2610.0	2890.0	2500.0	-280.0	390.0	-300.0	90.0	100.0

# Смотрим тестовую выборку

test

	sales	Lag3	Lag4	Lag5	Diff_on_Lag3	Diff_on_Lag4	Diff_on_Lag5	Diff2_on_Lag4	Diff2_on_Lag5
2018-01-17	2250	2500.0	2610.0	2890.0	-110.0	-280.0	390.0	110.0	90.0
2018-01-18	2350	2750.0	2500.0	2610.0	250.0	-110.0	-280.0	-390.0	110.0
2018-01-19	2550	2700.0	2750.0	2500.0	-50.0	250.0	-110.0	140.0	-390.0
2018-01-20	3000	2250.0	2700.0	2750.0	-450.0	-50.0	250.0	200.0	140.0

Обратите внимание, что разности, созданные на основе Lag3 (меньше горизонта прогнозирования), который использует информацию тестовой выборки, также будут использовать информацию тестовой выборки (выделено красной рамкой). Давайте посмотрим, как образуется «протечка» в разности, вычисленной на основе Lag3.

## Diff\_on\_Lag3

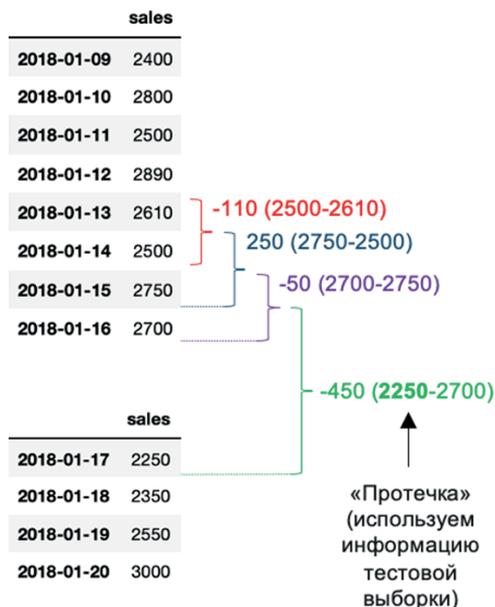


Рис. 62 Разность, созданная на основе лага с порядком меньше горизонта прогнозирования, тоже дает «протечку»

А в остальных разностях «протечки» не происходит.

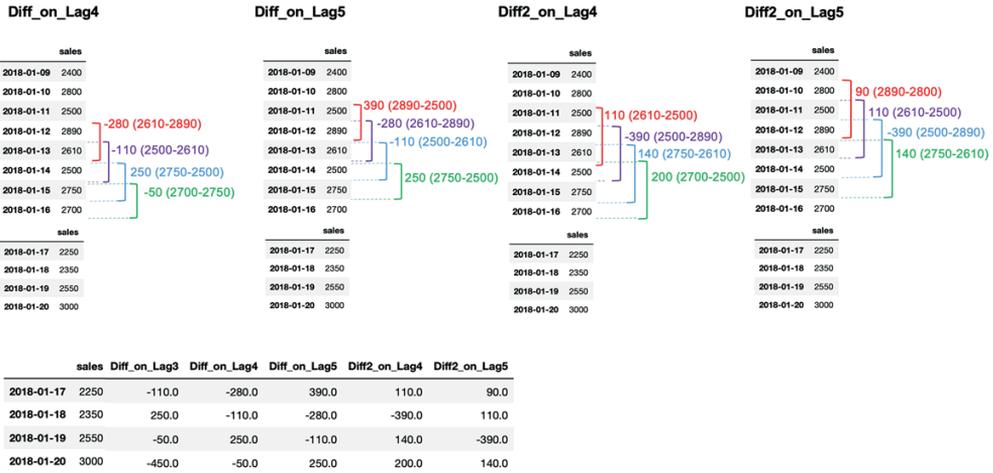


Рис. 63 Разности, созданные на основе лагов с порядком, равным горизонту прогнозирования и выше, не дают «протечек»

Кроме лагов и разностей, в качестве признаков можно использовать статистики, меняющиеся со временем, – скользящие статистики. В рамках подхода «скользящее окно» библиотека pandas вычисляет статистику по «окну» данных, представляющему определенный период времени. Затем окно смещается на определенный интервал времени, и статистика постоянно вычисляется для каждого нового окна до тех пор, пока окно охватывает даты временного ряда. Библиотека pandas непосредственно поддерживает скользящие оконные функции, предлагая метод `.rolling()` объекта Series и объекта DataFrame. В таблице ниже приведен ряд таких функций.

Таблица 11 Скользящие оконные функции в библиотеке pandas

Функция	Описание
<code>.rolling().mean()</code>	Среднее значение в окне
<code>.rolling().std()</code>	Стандартное отклонение в окне
<code>.rolling().var()</code>	Дисперсия в окне
<code>.rolling().min()</code>	Минимальное значение в окне
<code>.rolling().max()</code>	Максимальное значение в окне
<code>.rolling().cov()</code>	Коэффициент ковариации в окне
<code>.rolling().quantile()</code>	Оценка для заданного процентиля / выборочного квантиля в окне
<code>.rolling().corr()</code>	Коэффициент корреляции в окне

Функция	Описание
<code>.rolling().median()</code>	Медиана в окне
<code>.rolling().sum()</code>	Сумма в окне
<code>.rolling().apply()</code>	Применение пользовательской функции к значениям окна
<code>.rolling().count()</code>	Количество непропущенных значений в окне
<code>.rolling().skew()</code>	Коэффициент асимметрии в окне
<code>.rolling().kurt()</code>	Коэффициент эксцесса в окне

Самая часто используемая скользящая статистика – простое скользящее среднее (simple moving average):

$$SMA = \frac{A_1 + A_2 + \dots + A_n}{n}$$

Заметим, что скользящее среднее используется не только для конструирования признаков, но и в качестве прогнозной модели (когда прогноз – скользящее среднее  $n$  последних наблюдений), а также для сглаживания выбросов, краткосрочных колебаний и более четкого выделения долгосрочных тенденций в ряде данных. На практике ширину окна скользящего среднего устанавливают равной горизонту прогнозирования или больше его. При этом ширина окна скользящей статистики не должна быть равна или превышать длину набора данных, на основе которого она создается, в противном случае она будет состоять из одного значения и пропусков.

Заново создадим данные продаж и вычислим скользящие средние с шириной окна 3, шириной окна 12 (когда ширина окна равна длине набора), шириной окна 13 (когда ширина окна больше длины набора). Для этого воспользуемся параметром `window` метода `.rolling()`.

```
# создаем набор продаж
example = pd.DataFrame(
    {'sales': [2400, 2800, 2500, 2890, 2610, 2500,
              2750, 2700, 2250, 2350, 2550, 3000]},
    index=pd.date_range(start='2018-01-09', periods=12)
)

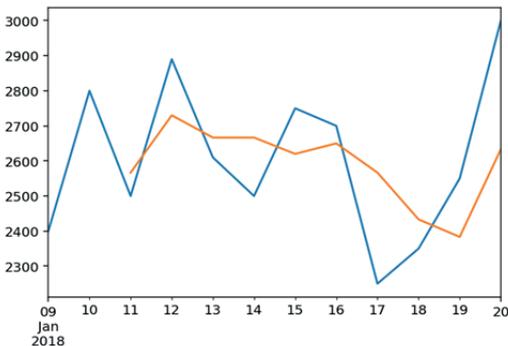
# создаем скользящие средние с шириной окна 3,
# шириной окна 12 и шириной окна 13
example['rolling_mean3'] = example['sales'].rolling(window=3).mean()
example['rolling_mean12'] = example['sales'].rolling(window=12).mean()
example['rolling_mean13'] = example['sales'].rolling(window=13).mean()
example
```

sales		rolling_mean3	rolling_mean12	rolling_mean13
2018-01-09	2400	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN
2018-01-11	2500	2566.66667	NaN	NaN
2018-01-12	2890	2730.00000	NaN	NaN
2018-01-13	2610	2666.66667	NaN	NaN
2018-01-14	2500	2666.66667	NaN	NaN
2018-01-15	2750	2620.00000	NaN	NaN
2018-01-16	2700	2650.00000	NaN	NaN
2018-01-17	2250	2566.66667	NaN	NaN
2018-01-18	2350	2433.33333	NaN	NaN
2018-01-19	2550	2383.33333	NaN	NaN
2018-01-20	3000	2633.33333	2608.33333	NaN

Видим, что скользящее среднее с шириной окна, равной длине набора, состоит только из одного непропущенного значения, а скользящее среднее с шириной окна, превышающей длину набора, состоит из одних пропусков.

Давайте визуализируем скользящее среднее с шириной окна 3.

```
# визуализируем продажи и скользящее
# среднее продаж с шириной окна 3
example['sales'].plot()
example['rolling_mean3'].plot();
```



С помощью параметра `min_periods` метода `.rolling()` можно задать минимальное количество наблюдений в окне, требуемое для вычисления значения (в противном случае результатом будет значение NaN). По умолчанию значение параметра `min_periods` равно значению параметра `window`.

Давайте создадим скользящее среднее с шириной окна 3 и минимальным количеством наблюдений в окне, равным 2. Для этого воспользуемся параметром `window` метода `.rolling()`.

```
# создаем скользящие средние, окно с шириной 3 и минимальным
# количеством наблюдений в окне, равным 2
example['rolling_mean3_min_periods_2'] = example['sales'].rolling(
    window=3, min_periods=2).mean()
example
```

	sales	rolling_mean3	rolling_mean12	rolling_mean13	rolling_mean3_min_periods_2
2018-01-09	2400	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	2600.000000
2018-01-11	2500	2566.666667	NaN	NaN	2566.666667
2018-01-12	2890	2730.000000	NaN	NaN	2730.000000
2018-01-13	2610	2666.666667	NaN	NaN	2666.666667
2018-01-14	2500	2666.666667	NaN	NaN	2666.666667
2018-01-15	2750	2620.000000	NaN	NaN	2620.000000
2018-01-16	2700	2650.000000	NaN	NaN	2650.000000
2018-01-17	2250	2566.666667	NaN	NaN	2566.666667
2018-01-18	2350	2433.333333	NaN	NaN	2433.333333
2018-01-19	2550	2383.333333	NaN	NaN	2383.333333
2018-01-20	3000	2633.333333	2608.333333	NaN	2633.333333

Самый простой способ получить скользящие статистики, не используя информацию тестовой выборки, – вычислить их с лагом, равным горизонту прогнозирования (или выше). Если горизонт прогнозирования равен 4 дням, скользящее среднее нужно вычислять с лагом 4. Давайте вычислим скользящие средние с шириной окна 3 и шириной окна 4 с лагом 3, 4 и 5, предварительно удалив ранее созданные переменные.

```
# удалим переменные по паттерну 'rolling_mean'
example.drop(example.filter(regex='rolling_mean').columns,
              axis=1, inplace=True)
# вычисляем скользящие средние с шириной 3 и 4
# и с лагом 3, 4, 5
example['rolling_mean3_lag3'] = example['sales'].shift(periods=3).rolling(
    min_periods=1, window=3).mean()
example['rolling_mean4_lag3'] = example['sales'].shift(periods=3).rolling(
    min_periods=1, window=4).mean()
example['rolling_mean3_lag4'] = example['sales'].shift(periods=4).rolling(
    min_periods=1, window=3).mean()
example['rolling_mean4_lag4'] = example['sales'].shift(periods=4).rolling(
    min_periods=1, window=4).mean()
example['rolling_mean3_lag5'] = example['sales'].shift(periods=5).rolling(
    min_periods=1, window=3).mean()
example['rolling_mean4_lag5'] = example['sales'].shift(periods=5).rolling(
    min_periods=1, window=4).mean()
example
```

	sales	rolling_mean3_lag3	rolling_mean4_lag3	rolling_mean3_lag4	rolling_mean4_lag4	rolling_mean3_lag5	rolling_mean4_lag5
2018-01-09	2400	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-10	2800	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-11	2500	NaN	NaN	NaN	NaN	NaN	NaN
2018-01-12	2890	2400.000000	2400.000000	NaN	NaN	NaN	NaN
2018-01-13	2610	2600.000000	2600.000000	2400.000000	2400.000000	NaN	NaN
2018-01-14	2500	2566.666667	2566.666667	2600.000000	2600.000000	2400.000000	2400.000000
2018-01-15	2750	2730.000000	2647.500000	2566.666667	2566.666667	2600.000000	2600.000000
2018-01-16	2700	2666.666667	2700.000000	2730.000000	2647.500000	2566.666667	2566.666667
2018-01-17	2250	2666.666667	2625.000000	2666.666667	2700.000000	2730.000000	2647.500000
2018-01-18	2350	2620.000000	2687.500000	2666.666667	2625.000000	2666.666667	2700.000000
2018-01-19	2550	2650.000000	2640.000000	2620.000000	2687.500000	2666.666667	2625.000000
2018-01-20	3000	2566.666667	2550.000000	2650.000000	2640.000000	2620.000000	2687.500000

Наша  
тестовая выборка

Теперь внимательно посмотрим, как была вычислена каждая скользящая статистика. Наблюдения обучающей выборки пометим синим цветом, наблюдения тестовой выборки – красным цветом.

Давайте посмотрим, как вычисляется скользящее среднее с шириной окна 3 и лагом 3 (*rolling\_mean3\_lag3*).

```
# вычисление rolling_mean3_lag3
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2800 + 2500 + 2890) / 3)
print((2500 + 2890 + 2610) / 3)
print((2890 + 2610 + 2500) / 3)
print((2610 + 2500 + 2750) / 3)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700 + 2250) / 3)
```

	rolling_mean3_lag3
nan	NaN
nan	NaN
nan	NaN
2400.0	NaN
2600.0	2400.000000
2566.6666666666665	2600.000000
2730.0	2566.666667
2666.6666666666665	2730.000000
2666.6666666666665	2666.666667
2620.0	2666.666667
2650.0	2620.000000
2566.6666666666665	2650.000000
	2566.666667

Видим, что скользящее среднее с шириной окна 3 и лагом 3 в последнем вычислении использует наблюдение тестовой выборки.

Смотрим, как вычисляется скользящее среднее с шириной окна 4 и лагом 3 (*rolling\_mean4\_lag3*).

```
# вычисление rolling_mean4_lag3
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700 + 2250) / 4)
```

	rolling_mean4_lag3
nan	NaN
nan	NaN
nan	NaN
2400.0	2400.000000
2600.0	2600.000000
2566.6666666666665	2566.666667
2647.5	2647.500000
2700.0	2700.000000
2625.0	2625.000000
2687.5	2687.500000
2640.0	2640.000000
2550.0	2550.000000

Видим, что скользящее среднее с шириной окна 4 и лагом 3 в последнем вычислении использует наблюдение тестовой выборки.

Теперь смотрим, как вычисляется скользящее среднее с шириной окна 3 и лагом 4 (*rolling\_mean3\_lag4*).

```
# вычисление rolling_mean3_lag4
print(np.nan)
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2800 + 2500 + 2890) / 3)
print((2500 + 2890 + 2610) / 3)
print((2890 + 2610 + 2500) / 3)
print((2610 + 2500 + 2750) / 3)
print((2500 + 2750 + 2700) / 3)
```

	rolling_mean3_lag4
nan	
nan	NaN
nan	NaN
nan	NaN
2400	NaN
2600.0	2400.000000
2566.6666666666665	2600.000000
2730.0	2566.666667
2666.6666666666665	2730.000000
2666.6666666666665	2666.666667
2620.0	2666.666667
2650.0	2620.000000
	2650.000000

Видим, что скользящее среднее с шириной окна 3 и лагом 4 ни в одном из вычислений не использует информацию тестовой выборки.

Теперь посмотрим, как вычисляется скользящее среднее с шириной окна 4 и лагом 4 (*rolling\_mean4\_lag4*).

```
# вычисление rolling_mean4_lag4
print(np.nan)
print(np.nan)
print(np.nan)
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
```

	rolling_mean4_lag4
nan	
nan	NaN
nan	NaN
nan	NaN
2400.0	NaN
2600.0	2400.000000
2566.6666666666665	2600.000000
2647.5	2566.666667
2700.0	2647.500000
2625.0	2700.000000
2687.5	2625.000000
2640.0	2687.500000
	2640.000000

Видим, что скользящее среднее с шириной окна 4 и лагом 4 ни в одном из вычислений не использует информацию тестовой выборки.

Легко понять, что скользящее среднее с шириной окна 3 и лагом 5 и скользящее среднее с шириной окна 4 и лагом 5 тоже не используют информацию тестовой выборки.

Теперь попробуем второй способ получить скользящие статистики, не используя информацию тестовой выборки.

Предварительно удалим ранее созданные переменные, а значения в наблюдениях, которые будут приходиться на тестовую выборку (последние 4 наблюдения исходного набора), заменим на значения NaN.

```
# удалим переменные по паттерну 'rolling_mean'
example.drop(example.filter(regex='rolling_mean').columns,
              axis=1, inplace=True)
# значения в наблюдениях, которые будут приходиться на тест
# (последние 4 наблюдения), заменяем на значения NaN
example['sales'].iloc[-HORIZON:] = np.NaN
```

Теперь создаем столбец *rolling\_mean4* – скользящее среднее с шириной окна 4, при этом скользящее среднее считаем с лагом 1. Позже поясним, зачем нам нужно вычислять скользящее среднее с лагом 1. Из-за лага скользящие средние будут записываться в столбец *rolling\_mean4*, начиная с 5-й строки (а не с 4-й). Для уменьшения количества значений NaN можно регулировать значение параметра *min\_periods*, в нашем случае установим его равным 1.

```
# вычисляем скользящее среднее с шириной окна 4 и лагом 1
example['rolling_mean4_lag1'] = example['sales'].shift(
    periods=1).rolling(window=4, min_periods=1).mean()
example
```

	sales	rolling_mean4_lag1
2018-01-09	2400.0	NaN
2018-01-10	2800.0	2400.000000
2018-01-11	2500.0	2600.000000
2018-01-12	2890.0	2566.666667
2018-01-13	2610.0	2647.500000
2018-01-14	2500.0	2700.000000
2018-01-15	2750.0	2625.000000
2018-01-16	2700.0	2687.500000
2018-01-17	NaN	2640.000000
2018-01-18	NaN	2650.000000
2018-01-19	NaN	2725.000000
2018-01-20	NaN	2700.000000

Наша  
тестовая выборка

Посмотрим, как было вычислено скользящее среднее с шириной окна 4 и лагом 1.

```
# вычисление rolling_mean4_lag1
print(np.nan)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
```

```

print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700) / 2)
print(2700 / 1)

```

Взглянем на первое скользящее среднее в тестовой выборке. Это будет среднее значение, взятое по последним 4 наблюдениям обучающей выборки.

sales	
2018-01-09	2400.0
2018-01-10	2800.0
2018-01-11	2500.0
2018-01-12	2890.0
2018-01-13	2610.0
2018-01-14	2500.0
2018-01-15	2750.0
2018-01-16	2700.0

} 2640

Второе, третье и четвертое скользящее среднее вычисляем как среднее, взятое по последним трем, двум, одному наблюдению в обучающей выборке соответственно.

sales	sales	sales			
2018-01-09	2400.0	2018-01-09	2400.0	2018-01-09	2400.0
2018-01-10	2800.0	2018-01-10	2800.0	2018-01-10	2800.0
2018-01-11	2500.0	2018-01-11	2500.0	2018-01-11	2500.0
2018-01-12	2890.0	2018-01-12	2890.0	2018-01-12	2890.0
2018-01-13	2610.0	2018-01-13	2610.0	2018-01-13	2610.0
2018-01-14	2500.0	2018-01-14	2500.0	2018-01-14	2500.0
2018-01-15	2750.0	2018-01-15	2750.0	2018-01-15	2750.0
2018-01-16	2700.0	2018-01-16	2700.0	2018-01-16	2700.0

} 2650

} 2725

Таким образом, при вычислении скользящих средних для более поздних моментов времени в тестовой выборке мы используем средние, взятые по более поздним наблюдениям в обучающей выборке, с уменьшением ширины окна. Уменьшение ширины окна позволяет сильнее учитывать более поздние наблюдения в обучающей выборке при вычислении скользящих средних для более поздних моментов времени в тестовой выборке. При вычислении скользящих средних для тестовой выборки *мы ни разу не использовали значения продаж в тестовой выборке.*

Легко понять, если бы мы вычисляли скользящее среднее с шириной окна 4 без лага 1, то для последнего наблюдения мы не смогли бы выполнить расчет.

*# вычисляем скользящее среднее с шириной окна 4 без лага*

```
example['rolling_mean4'] = example['sales'].rolling(
    window=4, min_periods=1).mean()
```

example

	sales	rolling_mean4_lag1	rolling_mean4
2018-01-09	2400.0	NaN	2400.000000
2018-01-10	2800.0	2400.000000	2600.000000
2018-01-11	2500.0	2600.000000	2566.666667
2018-01-12	2890.0	2566.666667	2647.500000
2018-01-13	2610.0	2647.500000	2700.000000
2018-01-14	2500.0	2700.000000	2625.000000
2018-01-15	2750.0	2625.000000	2687.500000
2018-01-16	2700.0	2687.500000	2640.000000
2018-01-17	NaN	2640.000000	2650.000000
2018-01-18	NaN	2650.000000	2725.000000
2018-01-19	NaN	2725.000000	2700.000000
2018-01-20	NaN	2700.000000	NaN

Для скользящего среднего с шириной окна больше горизонта прогнозирования лаг 1 не требуется. Давайте вычислим скользящие средние с шириной окна 5 без лага и с лагом.

*# вычисляем скользящие средние с шириной окна*

*# 5 без лага и с лагом*

```
example['rolling_mean5'] = example['sales'].rolling(
    window=5, min_periods=1).mean()
```

```
example['rolling_mean5_lag1'] = example['sales'].shift(1).rolling(
    window=5, min_periods=1).mean()
```

example

	sales	rolling_mean4_lag1	rolling_mean4	rolling_mean5	rolling_mean5_lag1
2018-01-09	2400.0	NaN	2400.000000	2400.000000	NaN
2018-01-10	2800.0	2400.000000	2600.000000	2600.000000	2400.000000
2018-01-11	2500.0	2600.000000	2566.666667	2566.666667	2600.000000
2018-01-12	2890.0	2566.666667	2647.500000	2647.500000	2566.666667
2018-01-13	2610.0	2647.500000	2700.000000	2640.000000	2647.500000
2018-01-14	2500.0	2700.000000	2625.000000	2660.000000	2640.000000
2018-01-15	2750.0	2625.000000	2687.500000	2650.000000	2660.000000
2018-01-16	2700.0	2687.500000	2640.000000	2690.000000	2650.000000
2018-01-17	NaN	2640.000000	2650.000000	2640.000000	2690.000000
2018-01-18	NaN	2650.000000	2725.000000	2650.000000	2640.000000
2018-01-19	NaN	2725.000000	2700.000000	2725.000000	2650.000000
2018-01-20	NaN	2700.000000	NaN	2700.000000	2725.000000

Посмотрим, как было вычислено скользящее среднее с шириной окна 5 без лага.

```
# вычисление rolling_means
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2400 + 2800 + 2500 + 2890 + 2610) / 5)
print((2800 + 2500 + 2890 + 2610 + 2500) / 5)
print((2500 + 2890 + 2610 + 2500 + 2750) / 5)
print((2890 + 2610 + 2500 + 2750 + 2700) / 5)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700) / 2)
print(2700 / 1)

2400.0
2600.0
2566.6666666666665
2647.5
2640.0
2660.0
2650.0
2690.0
2640.0
2650.0
2725.0
2700.0
```

Рекомендации для этого способа – используйте лаг 1, если берете ширину окна скользящего среднего по горизонту прогнозирования, можно отказаться от лага, если ширина окна превышает горизонт прогнозирования, не берите ширину окна, равную или превышающую длину обучающей выборки, всегда используйте защиту от протечек в виде значений NaN для тестовой части набора.

Разумеется, не стоит ограничиваться вычислением скользящих средних, вы можете попробовать применить скользящие медианы, суммы, минимумы, максимумы, стандартные отклонения, скользящие средние абсолютные отклонения, скользящие медианные абсолютные отклонения, скользящие разности (разность между первым и последним элементами окна, разность между максимальным и минимальным значениями окна). Кроме того, можно создавать скользящие статистики с регуляризацией, когда мы случайно пропускаем наблюдения для подсчета статистики по окну.

Полезно добавлять усредненные скользящие статистики (их еще называют агрегированными). Такие переменные используются в библиотеке Greykite. Однако по-прежнему помните, что в агрегации могут участвовать скользящие статистики, не использующие информацию теста.

Мы можем написать удобную функцию `moving_stats()` для вычисления различных скользящих статистик разными способами. Здесь мы можем дополнительно выбрать веса, коэффициенты сезонности для вычисления скользящих статистик.

```

# функция для создания скользящих статистик
def moving_stats(series, alpha=1, seasonality=1,
                 periods=1, min_periods=1, window=4,
                 aggfunc='mean', fillna=None):
    """
    Создает скользящие статистики.

    Параметры
    -----
    alpha: int, по умолчанию 1
        Коэффициент авторегрессии.
    seasonality: int, по умолчанию 1
        Коэффициент сезонности.
    periods: int, по умолчанию 1
        Порядок лага, с которым вычисляем скользящие
        статистики.
    min_periods: int, по умолчанию 1
        Минимальное количество наблюдений в окне для
        вычисления скользящих статистик.
    window: int, по умолчанию 4
        Ширина окна. Не должна быть меньше
        горизонта прогнозирования.
    aggfunc: string, по умолчанию 'mean'
        Агрегирующая функция.
    fillna: int, по умолчанию None
        Стратегия импутации пропусков.
    """
    # задаем размер окна для определения
    # диапазона коэффициентов
    size = window if window != -1 else len(series) - 1
    # задаем диапазон значений коэффициентов в виде списка
    alpha_range = [alpha ** i for i in range(0, size)]
    # задаем минимально требуемое количество наблюдений
    # в окне для вычисления скользящего среднего
    # с поправкой на сезонность
    min_required_len = max(
        min_periods - 1, 0) * seasonality + 1

    # функция для получения лагов в соответствии
    # с заданной сезонностью
    def get_required_lags(series):
        # возвращает вычисленные лаги в соответствии
        # с заданной сезонностью
        return pd.Series(series.values[::-1]
                        [:-: seasonality])

    # функция для вычисления скользящей статистики
    def aggregate_window(series):
        tmp_series = get_required_lags(series)
        size = len(tmp_series)
        tmp = tmp_series * alpha_range[-size:]

        if aggfunc == 'mdad':
            return tmp.to_frame().agg(
                lambda x: np.nanmedian(
                    np.abs(x - np.nanmedian(x))))

```

```

else:
    return tmp.agg(aggfunc)

# собственно вычисление скользящих статистик
features = series.shift(periods=periods).rolling(
    window=seasonality * window
    if window != -1 else len(series) - 1,
    min_periods=min_required_len).aggregate(
    aggregate_window)

# импутируем пропуски
if fillna is not None:
    features.fillna(fillna, inplace=True)
return features

```

Давайте вычислим скользящее среднее с шириной окна 4 обоими способами.

```

# удалим переменные по паттерну 'rolling_mean'
example.drop(example.filter(regex='rolling_mean').columns,
              axis=1, inplace=True)

# вычислим скользящее среднее с шириной окна 4 обоими способами
example['rolling_mean4_frst_method'] = moving_stats(
    example['sales'], window=4, periods=4)
example['rolling_mean4_scnd_method'] = moving_stats(
    example['sales'], window=4, periods=1, min_periods=1)
example

```

	sales	rolling_mean4_frst_method	rolling_mean4_scnd_method
2018-01-09	2400.0	NaN	NaN
2018-01-10	2800.0	NaN	2400.000000
2018-01-11	2500.0	NaN	2600.000000
2018-01-12	2890.0	NaN	2566.666667
2018-01-13	2610.0	2400.000000	2647.500000
2018-01-14	2500.0	2600.000000	2700.000000
2018-01-15	2750.0	2566.666667	2625.000000
2018-01-16	2700.0	2647.500000	2687.500000
2018-01-17	NaN	2700.000000	2640.000000
2018-01-18	NaN	2625.000000	2650.000000
2018-01-19	NaN	2687.500000	2725.000000
2018-01-20	NaN	2640.000000	2700.000000

Начинающего пользователя может сбить с толку такое большое количество параметров, поэтому можно обойтись более простым вариантом функции. Ниже приведен пример более простой функции для вычисления скользящих статистик. В ней реализован обычный режим вычислений и быстрый режим вычислений (на основе функций библиотеки `bottleneck`).

```

# более простая (и быстрая) функция
# для создания скользящих статистик
def fast_moving_stats(series, periods=1, min_count=1,
                      window=4, fast=True, fillna=None,
                      aggfunc='mean'):
    """
    Создает скользящие статистики.

    Параметры
    -----
    series:
        pandas.Series
    periods: int, по умолчанию 1
        Порядок лага, с которым вычисляем скользящие
        статистики.
    min_count: int, по умолчанию 1
        Минимальное количество наблюдений в окне для
        вычисления скользящих статистик.
    window: int, по умолчанию 4
        Ширина окна. Не должна быть меньше
        горизонта прогнозирования.
    fast: bool, по умолчанию True
        Режим вычислений скользящих статистик.
    fillna: int, по умолчанию None
        Стратегия импутации пропусков.
    aggfunc: string, по умолчанию 'mean'
        Стратегия импутации пропусков.
    """
    if fast:
        def shift(xs, n):
            return np.concatenate((np.full(n, np.nan),
                                   xs[:-n]))

        arr = series.values
        arr = shift(xs=arr, n=periods)
        if aggfunc == 'mean':
            arr = bn.move_mean(
                arr, window=window, min_count=min_count)
        if aggfunc == 'std':
            arr = bn.move_std(
                arr, window=window, min_count=min_count)
        if aggfunc == 'sum':
            arr = bn.move_sum(
                arr, window=window, min_count=min_count)
        if aggfunc == 'median':
            arr = bn.move_median(
                arr, window=window, min_count=min_count)

        features = pd.Series(arr)
        features.index = series.index
    else:
        if aggfunc == 'mean':
            features = series.shift(
                periods=periods).rolling(
                window=window,
                min_periods=min_count).mean()

```

```

if aggfunc == 'std':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).std()
if aggfunc == 'sum':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).sum()
if aggfunc == 'median':
    features = series.shift(
        periods=periods).rolling(
            window=window,
            min_periods=min_count).median()

# импутируем пропуски
if fillna is not None:
    features.fillna(fillna, inplace=True)

return features

```

Давайте вновь вычислим скользящее среднее с шириной окна 4 обоими способами.

```

# применяем нашу более простую и быструю функцию,
# вычислим скользящее среднее с шириной окна 4 обоими способами
example['roll_mean4_frst_method'] = fast_moving_stats(
    example['sales'], window=4, periods=4,
    fast=True, fillna=None, aggfunc='mean')
example['roll_mean4_scnd_method'] = fast_moving_stats(
    example['sales'], window=4, min_count=1,
    fast=True, fillna=None, aggfunc='mean')
example

```

	sales	rolling_mean4_frst_method	rolling_mean4_scnd_method	roll_mean4_frst_method	roll_mean4_scnd_method
2018-01-09	2400.0	NaN	NaN	NaN	NaN
2018-01-10	2800.0	NaN	2400.000000	NaN	2400.000000
2018-01-11	2500.0	NaN	2600.000000	NaN	2600.000000
2018-01-12	2890.0	NaN	2566.666667	NaN	2566.666667
2018-01-13	2610.0	2400.000000	2647.500000	2400.000000	2647.500000
2018-01-14	2500.0	2600.000000	2700.000000	2600.000000	2700.000000
2018-01-15	2750.0	2566.666667	2625.000000	2566.666667	2625.000000
2018-01-16	2700.0	2647.500000	2687.500000	2647.500000	2687.500000
2018-01-17	NaN	2700.000000	2640.000000	2700.000000	2640.000000
2018-01-18	NaN	2625.000000	2650.000000	2625.000000	2650.000000
2018-01-19	NaN	2687.500000	2725.000000	2687.500000	2725.000000
2018-01-20	NaN	2640.000000	2700.000000	2640.000000	2700.000000

Вернемся к продажам австралийского вина.

Создадим некоторые календарные признаки – месяц, квартал и сезон. Целесообразно написать функцию, которую можно будет применять в аналогичных задачах.

```

# пишем функцию конструирования календарных признаков
def create_calendar_vars(df):
    # выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА КВАРТАЛОВ
    df['quarter'] = df.index.quarter
    # выделяем из индекса ПОРЯДКОВЫЕ НОМЕРА МЕСЯЦЕВ,
    # где 1 - январь, 12 - декабрь
    df['month'] = df.index.month
    # пишем функцию для создания переменной - сезон
    def get_season(month):
        if (month > 11 or month < 3):
            return 0
        elif (month == 3 or month <= 5):
            return 1
        elif (month >= 6 and month < 9):
            return 2
        else:
            return 3
    # создаем переменную - сезон
    df['season'] = df['month'].apply(
        lambda x: get_season(x))

    return df

```

Применяем функцию и смотрим на созданные нами признаки.

```

# создаем календарные признаки
wine_df = create_calendar_vars(wine_df)
wine_df.head()

```

	sales	quarter	month	season
month				
1980-01-01	15136	1	1	0
1980-02-01	16733	1	2	0
1980-03-01	20016	1	3	1
1980-04-01	17708	2	4	1
1980-05-01	18019	2	5	1

Выведем последние 12 наблюдений набора. Почему именно 12? Нам нужно разбить набор на обучающую, проверочную и тестовую выборки. На проверочной выборке мы подбираем ширину окна скользящих статистик, количество периодов для лагов, гиперпараметры модели машинного обучения. На тестовой выборке однократно проверяем модель, давшую наилучшие результаты. Размер проверочной и тестовой выборок определяется горизонтом прогнозирования, который у нас равен 6 месяцам, нам нужно взглянуть на последние 12 дат – 12 месяцев – и на их основе сформировать выборки.

```

# задаем горизонт
HORIZON = 6

# выведем последние HORIZON * 2 наблюдений
wine_df.tail(HORIZON * 2)

```

	sales	quarter	month	season
1993-09-01	22724	3	9	3
1993-10-01	28496	4	10	3
1993-11-01	32857	4	11	3
1993-12-01	37198	4	12	0
1994-01-01	13652	1	1	0
1994-02-01	22784	1	2	0
1994-03-01	23565	1	3	1
1994-04-01	26323	2	4	1
1994-05-01	23779	2	5	1
1994-06-01	27549	2	6	2
1994-07-01	29660	3	7	2
1994-08-01	23356	3	8	2

Теперь формируем обучающую, проверочную и тестовую выборки. Проверочная выборка нам нужна для настройки гиперпараметров модели прогнозирования временного ряда. Кроме того, диапазон порядков лагов и инкремент (шаг, с которым увеличивается порядок последующего лага), диапазон размеров ширины окна скользящей статистики и инкремент (шаг, с которым увеличивается ширина окна последующей скользящей статистики) – это тоже гиперпараметры, гиперпараметры модели, конструирующей признаки. Помним, что мы всегда работаем с конвейером моделей. Проверочная выборка должна иметь тот же размер, что и тестовая выборка. На тестовой выборке мы получаем итоговую оценку качества прогнозов нашего конвейера.

В нашем случае наблюдения уже были отсортированы по возрастанию дат, однако всегда перед формированием выборок следует проверить, отсортированы ли наблюдения по возрастанию дат. Создав выборки, мы должны проверить временные рамки и размеры выборок.

```
# формируем обучающую, валидационную
# (проверочную) и тестовую выборки
train = wine_df[wine_df.index <= '1993-08-01']
valid = wine_df[(wine_df.index > '1993-08-01') &
                (wine_df.index <= '1994-02-01')]
test = wine_df[wine_df.index > '1994-02-01']

# посмотрим временные рамки выборок
print("временные рамки обучающей выборки:",
      train.index[0].strftime('%Y-%m-%d'),
      train.index[-1].strftime('%Y-%m-%d'))
print("временные рамки проверочной выборки:",
      valid.index[0].strftime('%Y-%m-%d'),
      valid.index[-1].strftime('%Y-%m-%d'))
print("временные рамки тестовой выборки:",
      test.index[0].strftime('%Y-%m-%d'),
      test.index[-1].strftime('%Y-%m-%d'))
```

```
# посмотрим размеры выборок
print("")
print(f"размер обучающей выборки: {len(train)}")
print(f"размер проверочной выборки: {len(valid)}")
print(f"размер тестовой выборки: {len(test)}")

временные рамки обучающей выборки: 1980-01-01 1993-08-01
временные рамки проверочной выборки: 1993-09-01 1994-02-01
временные рамки тестовой выборки: 1994-03-01 1994-08-01

размер обучающей выборки: 164
размер проверочной выборки: 6
размер тестовой выборки: 6
```

Теперь мы выполним конкатенацию обучающей и проверочной выборок и получим обучающе-проверочную выборку. Затем создаем копию зависимой переменной из этой обучающе-проверочной выборки и заменяем ее значения в наблюдениях, приходящихся на горизонт прогнозирования (проверочную часть), на значения NaN.

```
# создаем обучающе-проверочный набор
train_valid = pd.concat([train, valid], axis=0)
# создаем копию зависимой переменной и заменяем значения
# в наблюдениях, приходящихся на горизонт прогнозирования
# (проверочную часть), пропусками
y_train_valid_with_nan = train_valid['sales'].copy()
y_train_valid_with_nan.iloc[-HORIZON:] = np.NaN
y_train_valid_with_nan.tail(HORIZON + 2)
```

```
month
1993-07-01    29356.0
1993-08-01    31234.0
1993-09-01         NaN
1993-10-01         NaN
1993-11-01         NaN
1993-12-01         NaN
1994-01-01         NaN
1994-02-01         NaN
Name: sales, dtype: float64
```

Сейчас на основе зависимой переменной со значениями NaN в последних 6 наблюдениях (в проверочной части) мы создадим лаги и скользящие средние для обучающе-проверочной выборки. Наличие значений NaN в зависимой переменной для наблюдений, приходящихся на проверочную часть, является защитой от протечек, т.е. гарантирует, что лаги и скользящие средние не будут использовать информацию проверочной выборки. Мы могли бы сразу вычислить лаги и скользящие средние на всем наборе, не создавая защиту, однако всегда есть риск, что мы можем задать для лага порядок меньше горизонта прогнозирования или вычислить скользящее среднее с помощью первого способа, задав при этом лаг меньше горизонта прогнозирования. Особенно это актуально для специалистов с небольшим опытом прогнозирования временных рядов.

Мы создадим лаги порядка от 6 до 10 включительно с инкрементом 2 (6, 8, 10), т.е. лаги, равные горизонту прогнозирования и выше его, но при этом меньше

длины обучающей выборки. Обучающе-проверочный набор `train_valid` включает 170 наблюдений, 6 наблюдений уйдут в проверочную часть, поэтому максимальный порядок лага, при котором у нас будет хотя бы одно непропущенное наблюдение в обучающей выборке, должен быть равен  $170 - 6 - 1 = 163$ . Обычно диапазон порядков лагов и инкремент (шаг, с которым увеличивается количество периодов или порядок лага в следующем лаге) определяется подбором.

Помимо лагов, мы создадим скользящее среднее с шириной окна 6 и скользящее среднее с шириной окна 8, используя в обоих случаях второй способ.

Поскольку лаги и скользящие средние мы будем создавать еще не раз, а в последующих проектах, возможно, нам потребуется создавать не только скользящие средние, но и скользящие суммы, медианы, давайте напишем функцию `calculate_lags_and_stats()` для вычисления лагов и скользящих статистик. Обратите внимание, что эта функция под капотом будет использовать ранее написанную функцию `moving_stats()`.

```
# пишем функцию для вычисления лагов и скользящих статистик
def calculate_lags_and_stats(df, target, lags_range=None,
                             moving_stats_range=None,
                             periods=1, min_periods=1,
                             aggfunc='mean', seasonality=1):
    """
    Вычисляет лаги и скользящие статистики.

    Параметры
    -----
    df: pandas.DataFrame
        Набор данных
    target: pandas.Series
        Массив меток. Наблюдения, приходящиеся на горизонт
        прогнозирования (для валидации - проверочная/тестовая
        выборка, для применения - выборка новых данных),
        должны содержать значения NaN.
    lags_range:
        Диапазон значений, значение - количество
        периодов для лагов.
    moving_stats_range:
        Диапазон значений, значение - ширина окна
        для скользящих статистик.
    seasonality: int, по умолчанию 1
        Коэффициент сезонности для скользящих статистик.
    periods: int, по умолчанию 1
        Порядок лага, с которым вычисляются
        скользящие статистики.
    min_periods: int, по умолчанию 1
        Минимальное количество наблюдений, требуемое
        для вычисления скользящей статистики.
    aggfunc: string, значение по умолчанию 'mean'
        Агрегирующая функция.
    """
    if lags_range is not None:
        # создаем лагу
        for i in lags_range:
            df[f"Lag_{i}"] = target.shift(i)
```

```

if moving_stats_range is not None:
    # создаем скользящие статистики
    for i in moving_stats_range:
        df[f"Moving_{aggfunc}_{i}"] = moving_stats(
            target, window=i, periods=periods,
            min_periods=min_periods, aggfunc=aggfunc,
            seasonality=seasonality)
return df

```

Итак, применяем нашу функцию для генерации лагов и скользящих средних.

```

# создаем лаги и скользящие средние
train_valid = calculate_lags_and_stats(
    train_valid, y_train_valid_with_nan,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    periods=1, min_periods=1,
    aggfunc='mean', seasonality=1)

```

Взглянем на результаты.

```

# выведем первые 5 наблюдений
train_valid.head()

```

	sales	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>month</b>									
<b>1980-01-01</b>	15136	1	1	0	NaN	NaN	NaN	NaN	NaN
<b>1980-02-01</b>	16733	1	2	0	NaN	NaN	NaN	15136.00	15136.00
<b>1980-03-01</b>	20016	1	3	1	NaN	NaN	NaN	15934.50	15934.50
<b>1980-04-01</b>	17708	2	4	1	NaN	NaN	NaN	17295.00	17295.00
<b>1980-05-01</b>	18019	2	5	1	NaN	NaN	NaN	17398.25	17398.25

Разбиваем обучающе-проверочную выборку на обучающую и проверочную выборки (напомним: длина проверочной выборки определяется горизонтом прогнозирования).

```

# создаем обучающий массив признаков, проверочный
# массив признаков, обучающий массив меток,
# проверочный массив меток
train, valid, y_train, y_valid = train_test_split(
    train_valid.drop('sales', axis=1),
    train_valid['sales'],
    test_size=HORIZON,
    shuffle=False)

```

Взглянем на первые 5 наблюдений обучающей и проверочной выборок, убедимся в правильных временных рамках выборок.

```

# временные рамки обучающей выборки
print("временные рамки обучающей выборки:",
      train.index[0].strftime('%Y-%m-%d'),
      train.index[-1].strftime('%Y-%m-%d'))

```

```
# смотрим первые 5 наблюдений обучающей выборки
```

```
train.head()
```

```
временные рамки обучающей выборки: 1980-01-01 1993-08-01
```

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>month</b>								
1980-01-01	1	1	0	NaN	NaN	NaN	NaN	NaN
1980-02-01	1	2	0	NaN	NaN	NaN	15136.00	15136.00
1980-03-01	1	3	1	NaN	NaN	NaN	15934.50	15934.50
1980-04-01	2	4	1	NaN	NaN	NaN	17295.00	17295.00
1980-05-01	2	5	1	NaN	NaN	NaN	17398.25	17398.25

```
# временные рамки проверочной выборки
```

```
print("временные рамки проверочной выборки:",
```

```
      valid.index[0].strftime('%Y-%m-%d'),
```

```
      valid.index[-1].strftime('%Y-%m-%d'))
```

```
# смотрим первые 5 наблюдений проверочной выборки
```

```
valid.head()
```

```
временные рамки проверочной выборки: 1993-09-01 1994-02-01
```

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>month</b>								
1993-09-01	3	9	3	24352.0	17466.0	30923.0	26953.000000	24830.875
1993-10-01	4	10	3	26805.0	19463.0	37240.0	27473.200000	25883.000
1993-11-01	4	11	3	25236.0	24352.0	17466.0	27640.250000	26953.000
1993-12-01	4	12	0	24735.0	26805.0	19463.0	28441.666667	27473.200
1994-01-01	1	1	0	29356.0	25236.0	24352.0	30295.000000	27640.250

Полученные пропуски в лагах и скользящих средних в обучающей выборке можно оставить как есть (CatBoost сам заменит пропуски минимальными значениями соответствующих признаков), заменить нулями, можно просто удалить наблюдения с пропусками (очень расточительно), можно заменить пропуски средним значением лага, вычисленным на обучающей выборке, минимальным или средним значением, вычисленным по первым – порядок лага / ширина окна скользящего среднего. Не забывайте что способ обработки пропусков в лагах будет особенно важен, если взят большой горизонт прогнозирования (например, выше 30 дней) и, соответственно, у лагов будет более высокий порядок. Однако помните: чем больше горизонт прогнозирования, тем меньше эффективность лага как признака.

Заменяем пропуски в лагах и скользящих средних в обучающей выборке с помощью среднего значения, вычисленного по первым – порядок лага / ширина окна скользящего среднего.

```

# импутируем пропуски в лагах и скользящих средних
# средними значениями по первым n непропущенным
# наблюдениям, где n - порядок лага / ширина окна
pattern = train.columns.str.contains('Lag_|Moving_')
features = train.columns[pattern].tolist()
for i in features:
    train[i].fillna(
        train[i][train[i].notnull()].head(
            int(re.findall(r'\d+', i)[0])).mean(),
        axis=0, inplace=True)
train.head()

```

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
month								
1980-01-01	1	1	0	17806.5	19183.875	19719.5	16848.775	17351.208482
1980-02-01	1	2	0	17806.5	19183.875	19719.5	15136.000	15136.000000
1980-03-01	1	3	1	17806.5	19183.875	19719.5	15934.500	15934.500000
1980-04-01	2	4	1	17806.5	19183.875	19719.5	17295.000	17295.000000
1980-05-01	2	5	1	17806.5	19183.875	19719.5	17398.250	17398.250000

Теперь обучим модель градиентного бустинга CatBoost. Градиентный бустинг сейчас стал популярен для прогнозирования временных рядов, и отдадим дань этой популярности. На практике вы должны начинать с простых моделей и постепенно переходить к более сложным. В розничной торговле хорошо работают модели наивного прогноза, сезонного наивного прогноза, модели расширяющегося и скользящего среднего, затем можно попробовать модель тройного экспоненциального сглаживания (ETS), SARIMAX, BATS/TBATS, линейную регрессию с регуляризацией по методу эластичной сети и только потом градиентный бустинг. В противном случае можно потерять много времени с настройкой градиентного бустинга, а потом может оказаться, что более простая модель дает более адекватные прогнозы. К тому же переход к более сложной модели без тестирования более простых для работодателя является показателем некоторой беспомощности специалиста.

Если в качестве метода прогнозирования временных рядов выбран градиентный бустинг, то можно дать следующие советы.

Сначала постройте базовые модели из вышеприведенного списка, прогнозы которых будете сравнивать с прогнозами модели градиентного бустинга. Уже на этапе построения простых моделей вы, возможно, найдете приемлемую модель.

Если зависимая переменная (наш временной ряд) содержит тренд, можно попробовать детрендинг. Мы удаляем из зависимой переменной тренд, предварительно спрогнозированный линейной моделью. На полученных остатках обучаем градиентный бустинг и получаем прогнозы, к ним добавляем тренд. Если процедура детрендинга позволила повысить качество модели, можно попробовать в качестве признака добавить тренд.

Если зависимая переменная (наш временной ряд) содержит выбросы объективной природы, применяем логарифмирование и обучаем модель. В таком случае не забудьте применить к прогнозам обученной модели экспоненцирование. Если зависимая переменная (наш временной ряд) содержит выбросы

необъективной природы (ошибки регистрации и прочее), необходимо попробовать сглаживание зависимой переменной, начинаем с простого скользящего среднего и заканчиваем экспоненциально взвешенным скользящим средним. Следует помнить: чем мощнее метод, тем выше риск переобучения, сглаживание необходимо делать только для обучающей выборки, на тестовой выборке оцениваем качество модели, построенной на сглаженном временном ряде. Если используется перекрестная проверка расширяющимся/скользящим окном, в каждой итерации на обучающей выборке выполняем сглаживание заново, на тестовой выборке смотрим качество модели. Хорошие инструменты по обработке выбросов предлагает библиотека ETNA (речь идет о классах `PredictionIntervalOutliersTransform`, `MedianOutliersTransform` и `DensityOutliersTransform`, функциях `get_anomalies_median()`, `get_anomalies_density()`, `get_anomalies_prediction_interval()`, `get_anomalies_hist()`).

На основе (прологарифмированной) зависимой переменной с удаленным трендом создаем лаги, скользящие статистики. Можно вычислить скользящие средние, медианы, минимумы, максимумы, суммы, стандартные отклонения, средние/медианные абсолютные отклонения, разности. Однако наряду со скользящими средними скользящие статистики, основанные на разностях, могут быть более эффективными, чем остальные статистики. В качестве разностей вы можете брать разницу между минимальным и максимальным значениями в окне, разницу между первым и последним значениями в окне. Если набор является большим, лучше использовать функции модуля `move` пакета `bottleneck` вместо `pd.rolling().agg` агрегирующая функция().

Чем больше лагов и скользящих статистик, тем больше вероятность, что хорошо сработает случайный отбор признаков, для `CatBoost` нужно попробовать `rsm`, для `LightGBM` – `feature_fraction`, для `XGBoost` – `colsample_by_tree`, `colsample_by_node` и `colsample_by_level`. При коротких горизонтах (до 30 дней) лаги могут быть довольно эффективными горизонтами, на длинных горизонтах (от 30 дней и выше) гораздо важнее, как вы моделируете тренд и сезонность с помощью точек изменения тренда и точек изменения сезонности (необходимо настраивать количество точек смены тренда/сезонности, расстояния между ними и расстояние между последней точкой смены тренда/сезонности и последним моментом времени обучающей выборки).

Помните, что лаги и скользящие средние можно агрегировать, главное, чтобы лаги и скользящие средние, участвующие в агрегации, не использовали информацию тестовой выборки.

Затем нужно добавить календарные признаки, но не все разом, а в зависимости от сезонности. Мы можем проанализировать сезонности с помощью метода `.plot_components()` библиотеки `Prophet` или функции `plot_quantiles_and_overlays()` библиотеки `Greykite`. Смотрим, какие сезонности есть во временном ряде, и по ним уже добавляем календарные признаки. Например, можно добавить порядковый номер квартала в году, порядковый номер дня в неделе/месяце/квартале/году, порядковый номер недели в месяце/квартале/году, порядковый номер времени года (сезона). Можно попробовать день недели, месяц, квартал, сезон ввести в модель градиентного бустинга как количественную переменную, а также как категориальную (на примере сезона: 1, 2, 3, 4, а можно «зима», «весна», «лето», «осень»). Иногда помогает перевод в дробный формат: например, порядковый номер дня в неделе можно запи-

сать как 1, а можно как  $1/7 = 0,143$ . Для переменных циклической природы стоит попробовать двойное косинусно-синусное преобразование, чтобы передать модели информацию о цикличности признака. Кроме того, полезно добавить флаги – обычный/выходной день, начало/конец месяца, начало/конец квартала. Если есть информация о выплатах зарплаты, например каждого 15-го и 31-го, добавить ее. Кроме того, необходимо добавить праздники и особые события. При работе с особыми событиями следует выяснить, будут ли они доступны в будущем. Допустим, мы прогнозируем продажи. Власти города придумали ежегодно проводить с 1 по 3 июля 3-дневный городской фестиваль, такое событие будет у нас в будущем, и мы можем создать переменную на основе этого события, поскольку оно наверняка повлияет на продажи. Мы можем создать индикатор данного события. Возьмем другой пример. Произошло землетрясение, которое безусловно повлияет на продажи. Однако такое событие является редким, «черным лебедем» и вряд ли повторится в будущем в ту же самую дату.

Для моделирования сезонностей можно и нужно применять члены ряда Фурье с периодами, соответствующими годовой, недельной и квартальной сезонностям. Здесь опять можно выделить библиотеку ETNA и ее класс `FourierTransform` для генерации членов ряда Фурье. Полезно создавать взаимодействия членов ряда Фурье и календарных признаков.

Потом можно добавить групповые скользящие статистики, расширяющиеся статистики, групповые средние, обычные и с лагом.

Хорошие результаты показало добавление в качестве признаков для градиентного бустинга результатов тройного экспоненциального сглаживания. Для обучающей выборки мы берем сглаженные значения, для тестовой выборки берем прогнозы согласно таксономии, приведенной ниже.

**Таблица 10** Таксономия моделей тройного экспоненциального сглаживания

Тренд	Сезонность	
	Аддитивная	Мультипликативная
<b>Без тренда</b>	значение уровня $\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)\ell_{t-1}$ значение тренда $I_t = \gamma(y_t - \ell_{t-1}) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = \ell_{t-1} + I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t + I_{t-L+1+(m-1)}$ прогноз	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)\ell_{t-1}$ значение тренда $I_t = \gamma(y_t/\ell_{t-1}) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = \ell_{t-1} \cdot I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t \cdot I_{t-L+1+(m-1)}$ прогноз
<b>Аддитивный</b>	значение уровня $\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$ значение тренда $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$ значение сезонности $I_t = \gamma(y_t - \ell_{t-1} - \phi b_{t-1}) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = \ell_{t-1} + b_{t-1} + I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t + mb_t + I_{t-L+1+(m-1)}$ прогноз	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$ значение тренда $b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$ значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} + b_{t-1})) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = (\ell_{t-1} + b_{t-1}) \cdot I_{t-L}$ сглаженное значение $F_{t+m} = (\ell_t + mb_t) \cdot I_{t-L+1+(m-1)}$ прогноз
<b>Аддитивный с затуханием</b>	значение уровня $\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$ значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)\phi b_{t-1}$ значение сезонности $I_t = \gamma(y_t - \ell_{t-1} - \phi b_{t-1}) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = \ell_{t-1} + \phi b_{t-1} + I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t + I_{t-L+1+(m-1)}$ прогноз	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$ значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)\phi b_{t-1}$ значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} + \phi b_{t-1})) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = (\ell_{t-1} + \phi b_{t-1}) \cdot I_{t-L}$ сглаженное значение $F_{t+m} = (\ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t) \cdot I_{t-L+1+(m-1)}$ прогноз
<b>Мультипликативный</b>	значение уровня $\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$ значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$ значение сезонности $I_t = \gamma(y_t - \ell_{t-1} \cdot b_{t-1}) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = (\ell_{t-1} \cdot b_{t-1}) + I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t \cdot b_t^m + I_{t-L+1+(m-1)}$ прогноз	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$ значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$ значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1})) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = (\ell_{t-1} \cdot b_{t-1}) \cdot I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t \cdot b_t^m \cdot I_{t-L+1+(m-1)}$ прогноз
<b>Мультипликативный с затуханием</b>	значение уровня $\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$ значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}^\phi$ значение сезонности $I_t = \gamma(y_t - \ell_{t-1} \cdot b_{t-1}^\phi) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) + I_{t-L}$ сглаженное значение $F_{t+m} = \ell_t \cdot (b_t \cdot (\phi + \phi^2 + \dots + \phi^m)) + I_{t-L+1+(m-1)}$ прогноз	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$ значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}^\phi$ значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1}^\phi)) + (1 - \gamma)I_{t-L}$ значение сезонности $S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) \cdot I_{t-L}$ сглаженное значение $F_{t+m} = (\ell_t \cdot (b_t \cdot (\phi + \phi^2 + \dots + \phi^m))) \cdot I_{t-L+1+(m-1)}$ прогноз

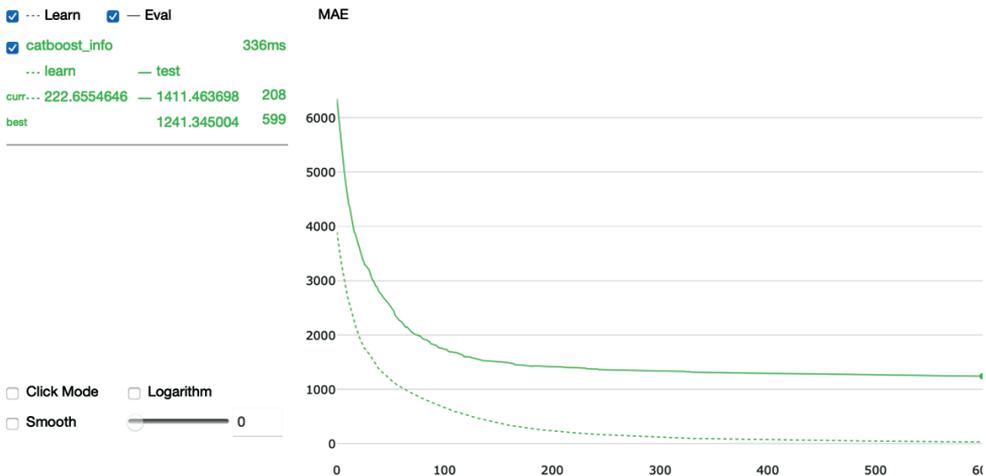
где:  $\ell_t$  – значение уровня для момента времени  $t$ ;  $b_t$  – значение тренда для момента времени  $t$ ;  $\alpha$  – сглаживающая константа для уровня;  $\beta$  – сглаживающая константа для тренда;  $\gamma$  – сглаживающая константа для сезонности;  $\phi$  – коэффициент затухания;  $I_t$  – значение индекса сезонности для момента времени  $t$ ;  $L$  – количество периодов;  $m$  – количество периодов, на которое делается прогноз.

Помните, что в CatBoost используется механизм зашумления оценок качества расщеплений с силой `random_strength`, который приводит к тому, что результаты модели будут зависеть от порядка обхода признаков.

Вернемся к нашему примеру. Мы можем использовать не только функцию потерь по умолчанию (RMSE), полезно попробовать и альтернативные функции потерь (MAE, Lq, Tweedie, Huber и Quantile). Сначала мы воспользуемся функцией потерь MAE.

```
# создаем модель CatBoost
ctbst_model = CatBoostRegressor(
    random_seed=42,
    n_estimators=600,
    learning_rate=0.05,
    loss_function='MAE',
    depth=9,
    logging_level='Silent')
```

```
# обучаем модель CatBoost
ctbst_model.fit(
    train, y_train,
    eval_set=(valid, y_valid),
    plot=True);
```



Оценим качество прогнозов для проверочной выборки.

```
# получаем прогнозы
ctbst_predictions = ctbst_model.predict(valid)
# вычисляем RMSE на проверочной выборке
ctbst_rmse = mean_squared_error(
    y_valid, ctbst_predictions, squared=False)
ctbst_rmse
```

1801.9610268193965

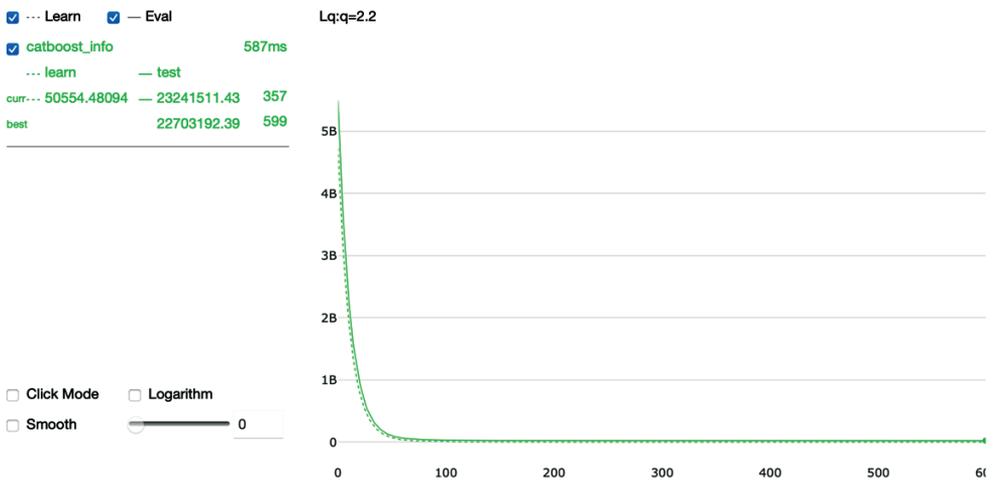
А теперь построим модель CatBoost с функцией потерь Lq.

*# создаем модель CatBoost*

```
ctbst_model2 = CatBoostRegressor(
    random_seed=42,
    n_estimators=600,
    learning_rate=0.05,
    loss_function='Lq:q=2.2',
    depth=9,
    logging_level='Silent')
```

*# обучаем модель CatBoost*

```
ctbst_model2.fit(
    train, y_train,
    eval_set=(valid, y_valid),
    plot=True);
```



Оценим качество прогнозов для проверочной выборки.

*# получаем прогнозы*

```
ctbst_predictions2 = ctbst_model2.predict(valid)
```

*# вычисляем RMSE на проверочной выборке*

```
ctbst_rmse2 = mean_squared_error(
    y_valid, ctbst_predictions2, squared=False)
ctbst_rmse2
```

2127.299403403843

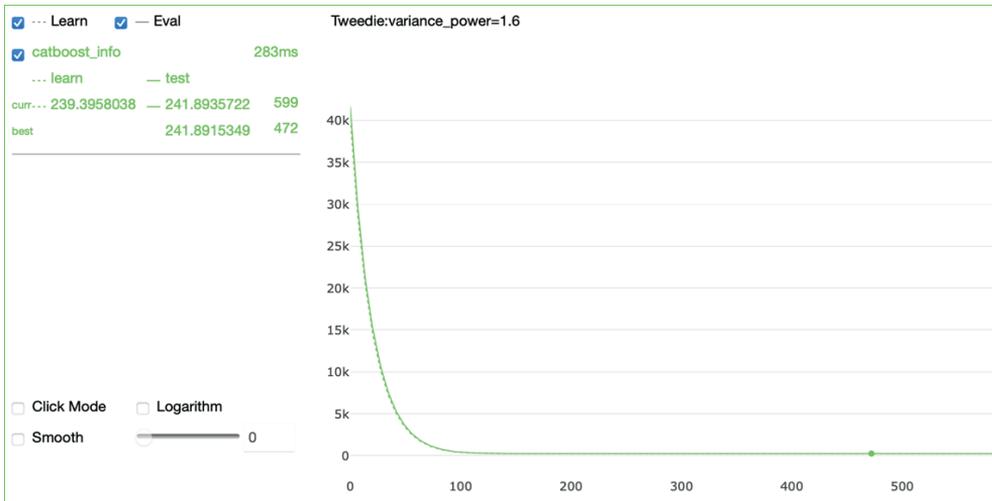
Наконец, построим модель CatBoost с функцией потерь Tweedie.

*# создаем модель CatBoost*

```
ctbst_model3 = CatBoostRegressor(
    random_seed=42,
    n_estimators=600,
    learning_rate=0.05,
    loss_function='Tweedie:variance_power=1.6',
    depth=6,
    logging_level='Silent')
```

*# обучаем модель CatBoost*

```
ctbst_model3.fit(
    train, y_train,
    eval_set=(valid, y_valid),
    plot=True);
```



Оценим качество прогнозов для проверочной выборки.

*# получаем прогнозы*

```
ctbst_predictions3 = ctbst_model3.predict(valid)
# вычисляем RMSE на проверочной выборке
ctbst_rmse3 = mean_squared_error(
    y_valid, ctbst_predictions3, squared=False)
ctbst_rmse3
```

2681.0465384633194

Остановим свой выбор на первой модели. Давайте визуализируем ее прогнозы.

*# визуализируем временной ряд*

```
# присваиваем прогнозам индексные метки
# проверочной выборки
ctbst_predictions = pd.Series(ctbst_predictions)
ctbst_predictions.index = valid.index
```

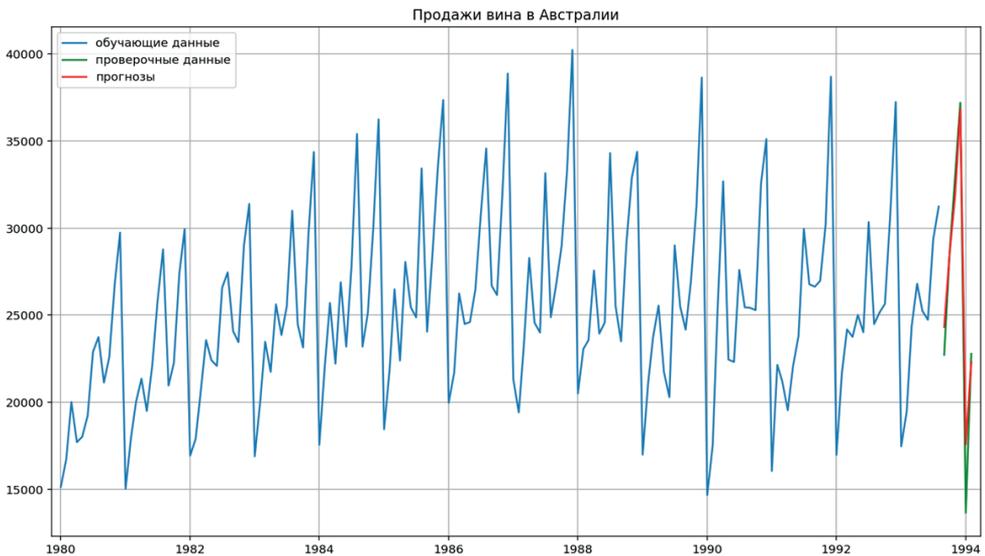
*# задаем размер графика*

```
fig, ax = plt.subplots(figsize=(14, 8))
# строим графики для обучающих данных,
# проверочных данных и прогнозов
ax.plot(y_train,
        label='обучающие данные')
ax.plot(y_valid,
        color='green',
        label='проверочные данные')
```

```

ax.plot(ctbst_predictions,
        color='red',
        label='прогнозы')
# задаем заголовок графика
ax.set_title("Продажи вина в Австралии")
# задаем начало оси x с отступом
ax.margins(x=0.01)
# задаем координатную сетку
ax.grid()
# задаем легенду
ax.legend();

```



Как вариант можно попробовать линейную модель, например гребневую регрессию. В таком случае мы должны выполнить стандартизацию и, если есть категориальные признаки (в нашей задаче их нет, хотя квартал, месяц и сезон можно превратить в категориальные признаки), применить дамми-кодирование.

```

# создаем экземпляр класса StandardScaler
scaler = StandardScaler()

# создаем список признаков для стандартизации
num_cols = train.select_dtypes(
    exclude='object').columns.tolist()

# создадим копии обучающего
# и проверочного массивов признаков
tr = train.copy()
val = valid.copy()

# выполняем стандартизацию
scaler.fit(tr[num_cols])
tr[num_cols] = scaler.transform(tr[num_cols])
val[num_cols] = scaler.transform(val[num_cols])

```

```
# создаем модель гребневой регрессии
ridge_model = Ridge(alpha=0.01)
# обучаем модель гребневой регрессии
ridge_model.fit(tr, y_train)

# получаем прогнозы
ridge_predictions = ridge_model.predict(val)
# вычисляем RMSE на проверочной выборке
ridge_rmse = mean_squared_error(
    y_valid, ridge_predictions, squared=False)
ridge_rmse
```

2538.4452183413346

Модель гребневой регрессии уступает по качеству обоим моделям CatBoost.

Давайте удалим лаги и скользящие средние в обучающе-проверочной выборке.

```
# удаляем лаги и скользящие средние
# в обучающе-проверочном наборе
train_valid = train_valid[train_valid.columns.drop(
    list(train_valid.filter(regex='Lag_|Moving_mean_')))]
```

Допустим, мы в качестве наилучшей модели выбрали модель CatBoost с функцией потерь MAE. Нам надо проверить ее качество на тестовой выборке. Помним, что проверочная выборка используется только для настройки гиперпараметров, при этом гиперпараметром может быть не только количество деревьев и темп обучения, но и диапазон порядков лагов, ширина окна скользящих статистик.

Сейчас мы выполним конкатенацию обучающе-проверочной и тестовой выборок и получим обучающе-тестовую выборку. Затем создаем копию зависимой переменной из этой обучающе-тестовой выборки и заменяем ее значения в наблюдениях, приходящихся на горизонт прогнозирования (тестовую часть), пропусками.

```
# создаем обучающе-тестовый набор
train_test = pd.concat([train_valid, test], axis=0)
# создаем копию зависимой переменной и заменяем значения
# в наблюдениях, приходящихся на горизонт прогнозирования
# (тестовую часть), пропусками
y_train_test_with_nan = train_test['sales'].copy()
y_train_test_with_nan.iloc[-HORIZON:] = np.NaN
y_train_test_with_nan.tail(HORIZON + 2)
```

```
month
1994-01-01    13652.0
1994-02-01    22784.0
1994-03-01         NaN
1994-04-01         NaN
1994-05-01         NaN
1994-06-01         NaN
1994-07-01         NaN
1994-08-01         NaN
Name: sales, dtype: float64
```

Сейчас на основе зависимой переменной со значениями NaN в последних 6 наблюдениях (в тестовой части) мы создадим лаги и скользящие средние для обучающе-тестовой выборки. Наличие значений NaN в зависимой переменной для наблюдений, приходящихся на тестовую часть, является защитой от протечек, т.е. гарантирует, что лаги и скользящие средние не будут использовать информацию тестовой выборки.

```
# создаем лаги и скользящие средние
train_test = calculate_lags_and_stats(
    train_test, y_train_test_with_nan,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    periods=1, min_periods=1,
    aggfunc='mean', seasonality=1)
```

Взглянем на результаты.

```
# выведем первые 5 наблюдений
train_test.head()
```

	sales	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>month</b>									
1980-01-01	15136	1	1	0	NaN	NaN	NaN	NaN	NaN
1980-02-01	16733	1	2	0	NaN	NaN	NaN	15136.00	15136.00
1980-03-01	20016	1	3	1	NaN	NaN	NaN	15934.50	15934.50
1980-04-01	17708	2	4	1	NaN	NaN	NaN	17295.00	17295.00
1980-05-01	18019	2	5	1	NaN	NaN	NaN	17398.25	17398.25

Разбиваем обучающе-тестовую выборку на обучающую и тестовую выборки.

```
# создаем обучающий массив признаков,
# тестовый массив признаков, обучающий
# массив меток, тестовый массив меток
train, test, y_train, y_test = train_test_split(
    train_test.drop('sales', axis=1),
    train_test['sales'],
    test_size=HORIZON,
    shuffle=False)
```

Взглянем на первые 5 наблюдений обучающей и тестовой выборки, убедимся в правильных временных рамках выборок.

```
# временные рамки обучающей выборки
print("временные рамки обучающей выборки:",
      train.index[0].strftime('%Y-%m-%d'),
      train.index[-1].strftime('%Y-%m-%d'))
# смотрим первые 5 наблюдений обучающей выборки
train.head()
```

временные рамки обучающей выборки: 1980-01-01 1994-02-01

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
month								
1980-01-01	1	1	0	NaN	NaN	NaN	NaN	NaN
1980-02-01	1	2	0	NaN	NaN	NaN	15136.00	15136.00
1980-03-01	1	3	1	NaN	NaN	NaN	15934.50	15934.50
1980-04-01	2	4	1	NaN	NaN	NaN	17295.00	17295.00
1980-05-01	2	5	1	NaN	NaN	NaN	17398.25	17398.25

```
# временные рамки тестовой выборки
print("временные рамки тестовой выборки:",
      test.index[0].strftime('%Y-%m-%d'),
      test.index[-1].strftime('%Y-%m-%d'))
# смотрим первые 5 наблюдений тестовой выборки
test.head()
```

временные рамки тестовой выборки: 1994-03-01 1994-08-01

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
month								
1994-03-01	1	3	1	22724.0	29356.0	25236.0	26285.166667	27287.625000
1994-04-01	2	4	1	28496.0	31234.0	24735.0	26997.400000	26992.142857
1994-05-01	2	5	1	32857.0	22724.0	29356.0	26622.750000	26285.166667
1994-06-01	2	6	2	37198.0	28496.0	31234.0	24544.666667	26997.400000
1994-07-01	3	7	2	13652.0	32857.0	22724.0	18218.000000	26622.750000

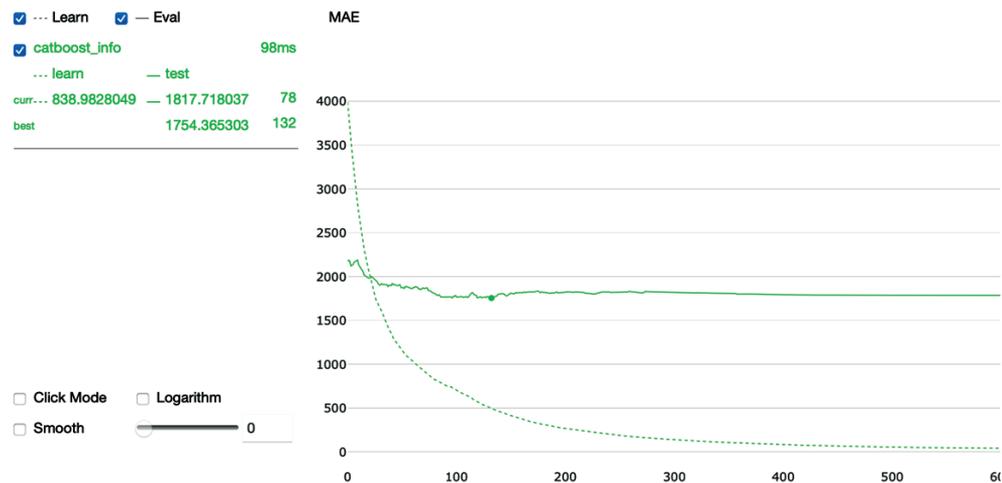
Заменяем пропуски в лагах и скользящих средних в обучающей выборке с помощью среднего значения, вычисленного по первым  $n$  – порядок лага или ширина скользящего среднего.

```
# импутируем пропуски в лагах и скользящих средних
# средними значениями по первым n непропущенным
# наблюдениям, где n - порядок лага / ширина окна
pattern = train.columns.str.contains('Lag_|Moving_')
features = train.columns[pattern].tolist()
for i in features:
    train[i].fillna(
        train[i][train[i].notnull()].head(
            int(re.findall(r'\d+', i)[0])).mean(),
        axis=0, inplace=True)
train.head()
```

quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8	
1980-01-01	1	1	0	17806.5	19183.875	19719.5	16848.775	17351.208482
1980-02-01	1	2	0	17806.5	19183.875	19719.5	15136.000	15136.000000
1980-03-01	1	3	1	17806.5	19183.875	19719.5	15934.500	15934.500000
1980-04-01	2	4	1	17806.5	19183.875	19719.5	17295.000	17295.000000
1980-05-01	2	5	1	17806.5	19183.875	19719.5	17398.250	17398.250000

Обучим модель градиентного бустинга CatBoost.

```
# обучаем модель CatBoost
ctbst_model.fit(
    train, y_train,
    eval_set=(test, y_test),
    plot=True);
```



Видим, что процесс обучения вначале является шумным. Улучшение качества является незначительным. Убеждаемся в том, что гиперпараметры и признаки, которые давали хорошее качество при обучении на одном временном интервале, при обучении на увеличенном временном интервале работают не так прекрасно. Здесь нам важно оценить качество прогнозов для тестовой выборки.

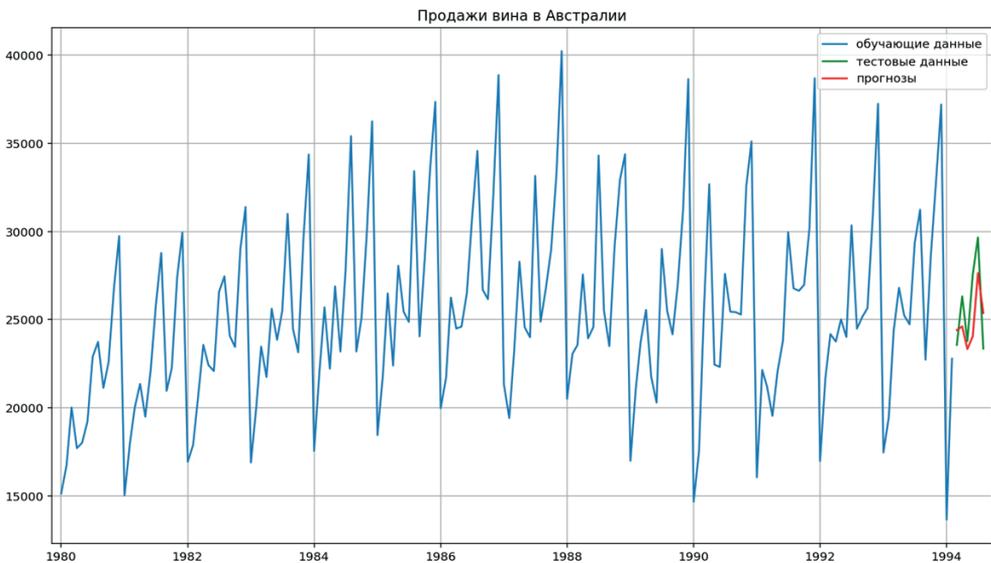
```
# получаем прогнозы
ctbst_predictions = ctbst_model.predict(test)
# вычисляем RMSE на тестовой выборке
ctbst_rmse = mean_squared_error(
    y_test, ctbst_predictions, squared=False)
ctbst_rmse
```

2007.9969479625

Давайте визуализируем прогнозы модели.

```
# визуализируем временной ряд
# присваиваем прогнозам индексные метки
# тестовой выборки
ctbst_predictions = pd.Series(ctbst_predictions)
ctbst_predictions.index = test.index

# задаем размер графика
fig, ax = plt.subplots(figsize=(14, 8))
# строим графики для обучающих данных,
# тестовых данных и прогнозов
ax.plot(y_train,
        label='обучающие данные')
ax.plot(y_test,
        color='green',
        label='тестовые данные')
ax.plot(ctbst_predictions,
        color='red',
        label='прогнозы')
# задаем заголовок графика
ax.set_title("Продажи вина в Австралии")
# задаем начало оси x с отступом
ax.margins(x=0.01)
# задаем координатную сетку
ax.grid()
# задаем легенду
ax.legend();
```



Если качество выбранной модели на тестовой выборке нас устраивает, нам нужно обучить ее на всей исторической выборке и получить прогнозы для новых данных, т.е. получить прогнозы на 6 месяцев вперед с сентября 1994 года по февраль 1995 года включительно.

Для начала удаляем лаги и скользящие средние в обучающе-тестовой выборке и присваиваем ей более нативное имя.

```
# удаляем лаги и скользящие средние
# в обучающе-тестовом наборе и присваиваем
# ему более нативное имя
hist_data = train_test[train_test.columns.drop(
    list(train_test.filter(regex='Lag_|Moving_mean_')))].copy()
```

Формируем исторический массив признаков и исторический массив меток.

```
# формируем исторический массив меток
# и исторический массив признаков
y_hist_data = hist_data.pop('sales')
```

Сейчас на основе зависимой переменной мы создадим лаги и скользящие средние для всей исторической выборки. Здесь нам защита от протечек в виде значений NaN не требуется.

```
# создаем лаги и скользящие средние
hist_data = calculate_lags_and_stats(
    hist_data, y_hist_data,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    periods=1, min_periods=1,
    aggfunc='mean', seasonality=1)
```

Взглянем на результаты.

```
# выведем первые 5 наблюдений
hist_data.head()
```

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>month</b>								
<b>1980-01-01</b>	1	1	0	NaN	NaN	NaN	NaN	NaN
<b>1980-02-01</b>	1	2	0	NaN	NaN	NaN	15136.00	15136.00
<b>1980-03-01</b>	1	3	1	NaN	NaN	NaN	15934.50	15934.50
<b>1980-04-01</b>	2	4	1	NaN	NaN	NaN	17295.00	17295.00
<b>1980-05-01</b>	2	5	1	NaN	NaN	NaN	17398.25	17398.25

Заменяем пропуски в лагах и скользящих средних в исторической выборке с помощью среднего значения, вычисленного по первым  $n$  – порядок лага или ширина скользящего среднего.

```
# импутируем пропуски в лагах и скользящих средних
# средними значениями по первым n непропущенным
# наблюдениям, где n - порядок лага / ширина окна
pattern = hist_data.columns.str.contains('Lag_|Moving_')
features = hist_data.columns[pattern].tolist()
```

```

for i in features:
    hist_data[i].fillna(
        hist_data[i][hist_data[i].notnull()].head(
            int(re.findall(r'\d+', i)[0])).mean(),
        axis=0, inplace=True)
hist_data.head()

```

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>month</b>								
<b>1980-01-01</b>	1	1	0	17806.5	19183.875	19719.5	16848.775	17351.208482
<b>1980-02-01</b>	1	2	0	17806.5	19183.875	19719.5	15136.000	15136.000000
<b>1980-03-01</b>	1	3	1	17806.5	19183.875	19719.5	15934.500	15934.500000
<b>1980-04-01</b>	2	4	1	17806.5	19183.875	19719.5	17295.000	17295.000000
<b>1980-05-01</b>	2	5	1	17806.5	19183.875	19719.5	17398.250	17398.250000

Создаем и обучаем модель CatBoost на всей исторической выборке. Используем те же гиперпараметры, что использовали при валидации и тестировании.

```

# создаем модель CatBoost для обучения
# на всей исторической выборке
hist_ctbst_model = CatBoostRegressor(
    random_seed=42,
    n_estimators=600,
    learning_rate=0.05,
    loss_function='MAE',
    depth=9,
    logging_level='Silent')

```

```

# обучаем модель CatBoost
# на всей исторической выборке
hist_ctbst_model.fit(
    hist_data, y_hist_data);

```

Создаем набор новых данных, длина которого равна горизонту прогнозирования, т.е. составляет 8 месяцев и охватывает период с сентября 1994 года по февраль 1995 года включительно.

```

# создаем датафрейм новых данных, длина которого равна
# горизонту прогнозирования, т.е. составляет 6 месяцев
new_data = pd.DataFrame(index=pd.date_range(
    start='1994-09-01', periods=HORIZON, freq='MS'))
new_data

```

**1994-09-01**

**1994-10-01**

**1994-11-01**

**1994-12-01**

**1995-01-01**

**1995-02-01**

С помощью ранее написанной функции `create_calendar_vars()` создаем календарные признаки для набора новых данных.

	quarter	month	season
<b>1994-09-01</b>	3	9	3
<b>1994-10-01</b>	4	10	3
<b>1994-11-01</b>	4	11	3
<b>1994-12-01</b>	4	12	0
<b>1995-01-01</b>	1	1	0
<b>1995-02-01</b>	1	2	0

Теперь формируем массив меток для новых данных, состоящих из значений NaN. По длине он равен горизонту прогнозирования.

```
# формируем массив меток для новых данных,
# состоящих из значений NaN, по длине
# равный горизонту прогнозирования
dates = pd.date_range(start='1994-09-01',
                      periods=HORIZON,
                      freq='MS')
y_new_data = pd.Series(np.NaN, index=dates)
y_new_data
```

```
1994-09-01    NaN
1994-10-01    NaN
1994-11-01    NaN
1994-12-01    NaN
1995-01-01    NaN
1995-02-01    NaN
Freq: MS, dtype: float64
```

Формируем объединенный массив меток, который будет состоять из исторического массива меток и массива меток для новых данных, заполненного значениями NaN.

```
# формируем объединенный массив меток из исторического
# массива меток и массива меток для новых данных
y_hist_data_new_data_with_nan = pd.concat(
    [y_hist_data, y_new_data], axis=0)
y_hist_data_new_data_with_nan.tail(HORIZON + 2)
```

```
1994-07-01    29660.0
1994-08-01    23356.0
1994-09-01         NaN
1994-10-01         NaN
1994-11-01         NaN
1994-12-01         NaN
1995-01-01         NaN
1995-02-01         NaN
dtype: float64
```

Сейчас на основе зависимой переменной со значениями NaN в последних 6 наблюдениях (в части, соответствующей новым данным) мы создадим лаги и скользящие средние для новых данных.

```
# создаем лаги и скользящие средние
new_data = calculate_lags_and_stats(
    new_data, y_hist_data_new_data_with_nan,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    periods=1, min_periods=1,
    aggfunc='mean', seasonality=1)
```

Взглянем на результаты.

```
# выведем первые 5 наблюдений
new_data.head()
```

	quarter	month	season	Lag_6	Lag_8	Lag_10	Moving_mean_6	Moving_mean_8
<b>1994-09-01</b>	3	9	3	23565.0	13652.0	32857.0	25705.333333	23833.500000
<b>1994-10-01</b>	4	10	3	26323.0	22784.0	37198.0	26133.400000	25288.000000
<b>1994-11-01</b>	4	11	3	23779.0	23565.0	13652.0	26086.000000	25705.333333
<b>1994-12-01</b>	4	12	0	27549.0	26323.0	22784.0	26855.000000	26133.400000
<b>1995-01-01</b>	1	1	0	29660.0	23779.0	23565.0	26508.000000	26086.000000

Теперь получаем прогнозы для набора новых данных с помощью модели CatBoost, обученной на всей исторической выборке.

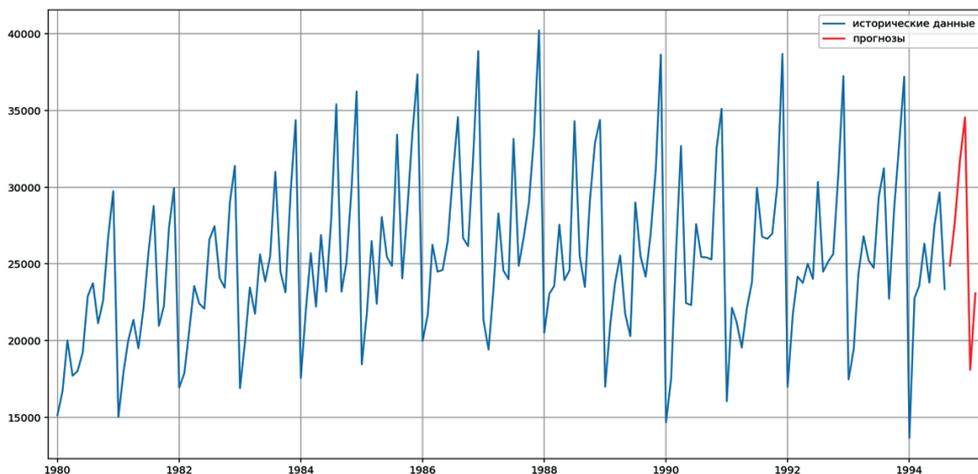
```
# получаем прогнозы для новых данных
new_data_predictions = hist_ctbst_model.predict(new_data)
new_data_pred = pd.Series(new_data_predictions)
new_data_pred.index = new_data.index
new_data_pred
```

```
1994-09-01    24879.071417
1994-10-01    27742.364082
1994-11-01    31794.016839
1994-12-01    34552.389683
1995-01-01    18096.807786
1995-02-01    23084.023440
Freq: MS, dtype: float64
```

Давайте визуализируем прогнозы модели для набора новых данных.

```
# задаем размер графика
fig, ax = plt.subplots(figsize=(16, 8))
# строим графики для исторических данных,
# новых данных, трендов и прогнозов
ax.plot(y_hist_data, label='исторические данные')
ax.plot(new_data_pred, color='red', label='прогнозы')
# задаем начало оси x с отступом
ax.margins(x=0.01)
```

```
# задаем координатную сетку
ax.grid()
# задаем легенду
ax.legend();
```



Проблема валидации с помощью разбиения на обучающую, проверочную и тестовую выборки заключается в том, что мы смотрим качество модели на каком-то ограниченном периоде и нам просто может «повезти» с проверочной и тестовой выборками, особенно если горизонт прогнозирования является небольшим и соответственно размеры проверочной и тестовой выборок будут небольшими. В нашем случае горизонт – 6 месяцев и мы смотрим, как модель прогнозирует продажи с сентября 1993 года по февраль 1994 года и с марта по август 1994 года. А если бы у нас горизонт был бы 2 месяца, мы бы смотрели, как модель прогнозирует продажи с мая по июнь 1994 года и с июля по август 1994 года. Получилось бы, что мы тестируем модель на очень ограниченном временном периоде. Нам важно посмотреть, какие прогнозы модель будет выдавать для *разных* временных периодов. В этом нам в сила помочь перекрестная проверка для временных рядов.

## 7.22. ВИДЫ ПЕРЕКРЕСТНОЙ ПРОВЕРКИ ДЛЯ ДАННЫХ ФОРМАТА «ОДИН ОБЪЕКТ – ОДНО НАБЛЮДЕНИЕ» (ПРИСУТСТВУЕТ ОСЬ ВРЕМЕНИ)

В этом разделе разберем виды перекрестных проверок, предназначенных для данных формата «один объект – одно наблюдение», когда присутствует ось времени.

Сначала ответим на вопрос: а можем ли мы применить обычную  $k$ -блочную перекрестную проверку к временным рядам?

Если мы применим традиционную  $k$ -блочную перекрестную проверку к данным с временной структурой, то на каждой итерации в нашу обучающую и тестовую выборки будут попадать наблюдения, относящиеся к разным мо-



<b>Date_Create</b>	<b>Value_abs</b>
2016-08-22	1450000
2016-08-23	1650000
2016-08-24	2250000
2016-08-25	1960000
2016-08-26	1950000
2016-08-27	1700000
2016-08-28	1550000
2016-08-29	2330000
2016-08-30	1900000
2016-08-31	3850000
2016-09-01	5450000
2016-09-02	3300000
2016-09-03	4950000
2016-09-04	3500000
2016-09-05	2300000
2016-09-06	1850000
2016-09-07	2900000
2016-09-08	2400000
2016-09-09	3100000
2016-09-10	2550000

Схематично изобразим загруженные данные.

0	2016-08-22	2016-08-23	2016-08-24	2016-08-25	2016-08-26	2016-08-27	2016-08-28	2016-08-29	2016-08-30	2016-08-31	2016-09-01	2016-09-02	2016-09-03	2016-09-04	2016-09-05	2016-09-06	2016-09-07	2016-09-08	2016-09-09	2016-09-10	
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					
9																					
10																					
11																					
12																					
13																					
14																					
15																					
16																					
17																					
18																					
19																					

Создаем массив признаков и массив меток и применяем обычную  $k$ -блочную перекрестную проверку.

*# создаем массив меток и массив признаков*

```
y_flat_data = flat_data.pop('Value_abs')
```

*# создаем экземпляр класса KFold*

```
kfoldcv = KFold(n_splits=3)
```

*# взглянем на индексы наблюдений, попавших в обучающую*

*# и тестовую выборки, по каждой из трех итераций*

```
for train_index, test_index in kfoldcv.split(flat_data):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = (flat_data.iloc[train_index],
                      flat_data.iloc[test_index])
    y_train, y_test = (y_flat_data.iloc[train_index],
                      y_flat_data.iloc[test_index])
    print("Общее кол-во наблюдений: %d" % (len(flat_data)))
    print("Обучающая выборка: %d" % (len(X_train)))
    print("Тестовая выборка: %d" % (len(X_test)))
    print("")
```

```

2016-08-29
2016-08-30
2016-08-31
2016-09-01
2016-09-02
2016-09-03
2016-09-04
2016-09-05
2016-09-06
2016-09-07
2016-09-08
2016-09-09
2016-09-10
2016-08-22
2016-08-23
2016-08-24
2016-08-25
2016-08-26
2016-08-27
2016-08-28
TRAIN: [ 7 8 9 10 11 12 13 14 15 16 17 18 19] TEST: [0 1 2 3 4 5 6]
```

Общее кол-во наблюдений: 20

Обучающая выборка: 13

Тестовая выборка: 7

**В тест попали самые ранние наблюдения**

```

2016-08-22
2016-08-23
2016-08-24
2016-08-25
2016-08-26
2016-08-27
2016-08-28
2016-09-05
2016-09-06
2016-09-07
2016-09-08
2016-09-09
2016-09-10
2016-08-29
2016-08-30
2016-08-31
2016-09-01
2016-09-02
2016-09-03
2016-09-04
TRAIN: [ 0 1 2 3 4 5 6 14 15 16 17 18 19] TEST: [7 8 9 10 11 12 13]
```

Общее кол-во наблюдений: 20

Обучающая выборка: 13

Тестовая выборка: 7

**В тест попали наблюдения, находящиеся между самыми ранними и самыми поздними наблюдениями**

```

2016-08-22
2016-08-23
2016-08-24
2016-08-25
2016-08-26
2016-08-27
2016-08-28
2016-08-29
2016-08-30
2016-09-31
2016-09-01
2016-09-02
2016-09-03
2016-09-04
2016-09-05
2016-09-06
2016-09-07
2016-09-08
2016-09-09
2016-09-10
TRAIN: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13] TEST: [14 15 16 17 18 19]
```

Общее кол-во наблюдений: 20

Обучающая выборка: 14

Тестовая выборка: 6

**И только здесь в тест попали самые поздние наблюдения**

Видим, что действительно при обучении наша модель получит «подсказки из будущего», а сама тестовая выборка будет неправдоподобной, потому что в ней присутствуют данные из более ранних и более поздних моментов времени (здесь наиболее показательна тестовая выборка на второй итерации), тогда как в реальности новые данные всегда будут относиться к более позднему периоду времени.

В перекрестной проверке, учитывающей временную структуру данных, не используется перемешивание данных и переставление обучающих блоков

и тестового блока местами, вместо этого каждый раз в хронологической последовательности формируется обучающая и тестовая выборки. Проще говоря, на каждой итерации мы разбиваем набор на обучающую и тестовую выборки так, чтобы в обучающую выборку попали более ранние наблюдения, а в тестовую выборку – более поздние. Здесь сразу договоримся о терминологии. Когда речь идет о проверке базовой модели, не предполагающей настройки гиперпараметров, мы в ходе перекрестной проверки формируем обучающие и тестовые выборки. Если предполагается настройка гиперпараметров, то в ходе перекрестной проверки мы формируем обучающие и проверочные выборки, предполагая, что для итоговой оценки модели у нас есть отложенная тестовая выборка, содержащая самые поздние данные. При реализации перекрестной проверки для временных рядов выполняется ряд правил.

Размер проверочной и тестовой выборок определяется, как правило, горизонтом прогнозирования, тот, в свою очередь, определяется бизнес-требованиями. Если вы предсказываете на 14 дней вперед, то и проверочная или тестовая выборка должна включать 14 более поздних наблюдений.

Размер тестовой выборки остается постоянным. Это значит, что метрики качества, полученные в результате вычислений прогнозов каждой обученной модели по тестовому набору, будут последовательны, и их можно объединять и сравнивать.

Размер обучающей выборки не может быть меньше проверочной/тестовой выборки.

Если данные содержат сезонность, обучающая выборка должна содержать не менее двух полных сезонных циклов (правило  $2L$ , где  $L$  – количество периодов в полном сезонном цикле, необходимое для инициализации параметров некоторых моделей, например для вычисления исходного значения тренда в модели тройного экспоненциального сглаживания), учитывая уменьшение длины ряда при выполнении процедур обычного и сезонного дифференцирования.

Если применяются признаки – лаги и скользящие статистики, то на каждой итерации лаги и скользящие статистики создаются заново и в проверочной/тестовой выборке они используют только данные обучающей выборки.

### 7.22.1. Перекрестная проверка расширяющимся окном

В перекрестной проверке расширяющимся окном (expanding window) количество наблюдений, использованных для обучения в каждой итерации, растет с числом итераций, предоставляя все больший объем данных для обучения.

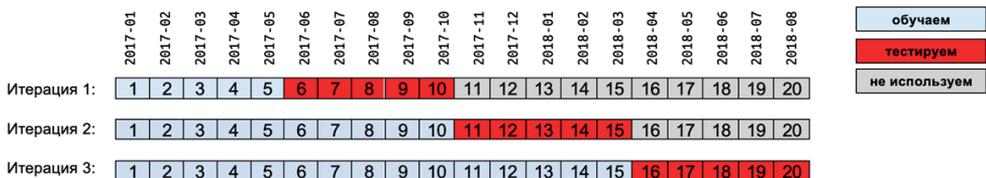


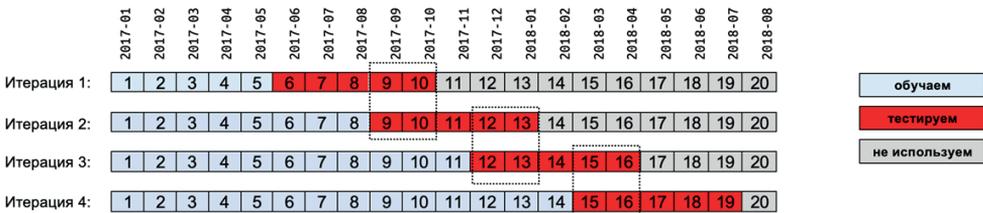
Рис. 64 Классическая перекрестная проверка расширяющимся окном

Видим, что с каждой итерацией объем обучающей выборки растет, при этом для тестирования мы каждый раз берем совершенно новые более поздние наблюдения. Обучающая выборка прирастает на количество наблюдений, равное горизонту прогнозирования.

Поскольку в рамках перекрестной проверки расширяющимся окном мы на каждой итерации обучаем модель на выборке все большего объема, при использовании моделей на основе градиентного бустинга это может потребовать коррекции темпа обучения, количества деревьев и максимальной глубины. При больших горизонтах прогнозирования, когда обучающая выборка увеличивается значительными темпами, целесообразна динамическая корректировка гиперпараметров, сама схема динамической корректировки гиперпараметров выступает как отдельный гиперпараметр.

Перекрестную проверку расширяющимся окном можно модифицировать так, чтобы обучающая выборка прирастала на количество наблюдений меньше горизонта прогнозирования, и тогда в тестовую выборку попадут наблюдения, уже попадавшие в тестовую выборку на предыдущей итерации. Это позволяет управлять скоростью обновления модели, лучше выявлять аномальные, нетипичные наблюдения, которые плохо предсказываются, точнее определять момент ухудшения качества модели.

В библиотеке `scikit-learn` с помощью класса `TimeSeriesSplit` ее нельзя корректно реализовать (для реализации используйте параметр `step_length` класса `ExpandingWindowSplitter` библиотеки `sktime`).



**Рис. 65** Модифицированная перекрестная проверка расширяющимся окном (в тестовую выборку попадают наблюдения, уже попадавшие в тестовую выборку на предыдущей итерации)

В Python перекрестную проверку расширяющимся окном можно реализовать с помощью класса `TimeSeriesSplit`.

```

from sklearn.model_selection import TimeSeriesSplit(n_splits='warn',
    ← Количество итераций (по умолчанию 5, должно быть не меньше 2)
    max_train_size=None, ← Максимальное количество наблюдений в обучающей выборке
    test_size=None,
    gap=0)
    ← Размер гэпа: количество наблюдений, которое необходимо удалить в конце обучающей выборки перед тестовой выборкой
    
```

Мы должны указать количество разбиений, после чего `TimeSeriesSplit` вернет индексы обучающих и тестовых наблюдений для каждого разбиения.

Общее количество обучающих и тестовых наблюдений вычисляется на каждой итерации () следующим образом:

```

training_size = i * n_samples / (n_splits + 1)
test_size = n_samples / (n_splits + 1),
    
```

где `n_samples` – общее количество наблюдений, а `n_splits` – количество разбиений.

Для наглядности рассмотрим наш пример с продажами квартир. У нас есть 20 наблюдений, и мы хотим сделать три итерации.

В первой итерации  $\text{training\_size} = 1 * 20 / (3+1) = 5$  и  $\text{test\_size} = 20 / (3+1) = 5$ .

Во второй итерации  $\text{training\_size} = 2 * 20 / (3+1) = 10$  и  $\text{test\_size} = 20 / (3+1) = 5$ .

В третьей итерации  $\text{training\_size} = 3 * 20 / (3+1) = 15$  и  $\text{test\_size} = 20 / (3+1) = 5$ .

Такая стратегия подойдет, если наш горизонт прогнозирования равен 5 – допустим, прогнозируем продажи на 5 дней вперед.

Давайте реализуем перекрестную проверку расширяющимся окном. Взглянем на индексы наблюдений, попавших в обучающую и тестовую выборки на каждой итерации.

```
# создаем экземпляр класса TimeSeriesSplit
expandcv = TimeSeriesSplit(n_splits=3)

# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train_index, test_index in expandcv.split(flat_data):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = (flat_data.iloc[train_index],
                      flat_data.iloc[test_index])
    y_train, y_test = (y_flat_data.iloc[train_index],
                      y_flat_data.iloc[test_index])

    print("Общее кол-во наблюдений: %d" % len(flat_data))
    print("Обучающая выборка: %d" % len(X_train))
    print("Тестовая выборка: %d" % len(X_test))
    print("")
```

```
TRAIN: [0 1 2 3 4] TEST: [5 6 7 8 9]
```

```
Общее кол-во наблюдений: 20
```

```
Обучающая выборка: 5
```

```
Тестовая выборка: 5
```

```
TRAIN: [0 1 2 3 4 5 6 7 8 9] TEST: [10 11 12 13 14]
```

```
Общее кол-во наблюдений: 20
```

```
Обучающая выборка: 10
```

```
Тестовая выборка: 5
```

```
TRAIN: [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14] TEST: [15 16 17 18 19]
```

```
Общее кол-во наблюдений: 20
```

```
Обучающая выборка: 15
```

```
Тестовая выборка: 5
```

Перед нами – перекрестная проверка расширяющимся окном. На каждой итерации мы разбиваем набор на обучающую и тестовую выборки так, чтобы в обучающую выборку попали более ранние наблюдения, а в тестовую выборку – более поздние. Размер тестовой выборки остается постоянным и равен 5 (допустим, мы предсказываем на 5 шагов вперед). Количество наблюдений, использованных для обучения в каждой итерации, растет с числом итераций, предоставляя все больший объем данных для обучения.

Можно зафиксировать не только количество разбиений, но и размер тестовой выборки, например нам нужно прогнозировать на 4 шага вперед, поэтому размер тестовой выборки каждый раз будет равен 4.

```
# создаем экземпляр класса TimeSeriesSplit
expandcv = TimeSeriesSplit(test_size=4, n_splits=3)

# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train_index, test_index in expandcv.split(flat_data):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = (flat_data.iloc[train_index],
                      flat_data.iloc[test_index])
    y_train, y_test = (y_flat_data.iloc[train_index],
                      y_flat_data.iloc[test_index])
    print("Общее кол-во наблюдений: %d" % len(flat_data))
    print("Обучающая выборка: %d" % len(X_train))
    print("Тестовая выборка: %d" % len(X_test))
    print("")
```

```
TRAIN: [0 1 2 3 4 5 6 7] TEST: [ 8  9 10 11]
Общее кол-во наблюдений: 20
Обучающая выборка: 8
Тестовая выборка: 4
```

```
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11] TEST: [12 13 14 15]
Общее кол-во наблюдений: 20
Обучающая выборка: 12
Тестовая выборка: 4
```

```
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15] TEST: [16 17 18 19]
Общее кол-во наблюдений: 20
Обучающая выборка: 16
Тестовая выборка: 4
```

Давайте загрузим данные о продажах вина в Австралии, создадим календарные признаки, сформируем массив признаков и массив меток, создадим модель CatBoost.

```
# загружаем набор о ежемесячных продажах вина в Австралии
wine_data = pd.read_csv('Data/monthly_australian_wine_sales.csv',
                       index_col=['month'],
                       parse_dates=['month'],
                       date_parser=lambda col: pd.to_datetime(
                           col, format='%Y-%m-%d'))

# создаем календарные признаки
wine_data = create_calendar_vars(wine_data)

# создаем массив меток и массив признаков
y_wine_data = wine_data.pop('sales')
```

```
# создаем модель CatBoost
ctbst_model = CatBoostRegressor(
    random_seed=42,
    n_estimators=600,
    learning_rate=0.05,
    loss_function='MAE',
    depth=9,
    logging_level='Silent')
```

Теперь нам нужно применить перекрестную проверку расширяющимся окном. На практике предпочитают не просто пользоваться классом `TimeSeriesSplit`, а пишут класс или функцию на основе класса `TimeSeriesSplit`, позволяющий(ую) вывести подробную информацию о том, что происходит под капотом перекрестной проверки, потому что очень часто что-то может пойти не так и нужно отслеживать возникающие проблемы. Обычно такой класс или функция выводит индексы наблюдений, попавших в обучающую и тестовую выборки на каждой итерации, вычисляет метрику качества на каждой итерации и усредненную метрику качества, визуализирует прогнозы модели для тестовой выборки на каждой итерации. Давайте напишем такую функцию.

```
# пишем функцию перекрестной проверки для рядов, которая выводит
# подробную информацию о происходящем внутри цикла перекрестной
# проверки
def timeseries_cv(data, y_data, model, max_train_size=None, test_size=None,
                 n_splits=3, gap=0, visualize=True, last_n_train=30):
    """
    Выполняет перекрестную проверку расширяющимся/
    скользящим окном без гэпа/с гэпом.

    Параметры
    -----
    X: pandas.DataFrame
        Массив признаков.
    y: pandas.Series
        Массив меток.
    model:
        Модель-регрессор: либо CatBoostRegressor, либо
        класс-регрессор библиотеки sklearn.
    max_train_size: int, по умолчанию None
        Максимальный размер обучающей выборки.
    test_size: int, по умолчанию None
        Максимальный размер тестовой выборки
        (определяется горизонтом прогнозирования).
    n_splits: int, по умолчанию 3
        Количество разбиений на обучающую
        и тестовую выборки.
    gap: int, по умолчанию 0
        Размер гэпа.
    visualize: bool, по умолчанию True
        Визуализация прогнозов.
    last_n_train: int, по умолчанию 30
        Вывод n наблюдений обучающей выборки
        при визуализации прогнозов.
    """
```

```

# создаем экземпляр класса TimeSeriesSplit
tscv = TimeSeriesSplit(max_train_size=max_train_size,
                      test_size=test_size,
                      n_splits=n_splits,
                      gap=gap)

# создаем пустой список для хранения значений метрики
rmse_lst = []

# взглянем на индексы наблюдений, попавших в обучающий
# и тестовый наборы, по каждой из трех итераций
for cnt, (train_index, test_index) in enumerate(tscv.split(data), 1):
    X_train, X_test = (data.iloc[train_index],
                      data.iloc[test_index])
    y_train, y_test = (y_data.iloc[train_index],
                      y_data.iloc[test_index])
    print("TRAIN:", [X_train.index[0].strftime('%Y-%m-%d'),
                    X_train.index[-1].strftime('%Y-%m-%d')],
          "TEST:", [X_test.index[0].strftime('%Y-%m-%d'),
                   X_test.index[-1].strftime('%Y-%m-%d')])
    print("Общее кол-во наблюдений: %d" % (len(data)))
    print("Обучающая выборка: %d" % (len(X_train)))
    print("Тестовая выборка: %d" % (len(X_test)))

# если модель - CatBoostRegressor
if model.__class__.__name__ == 'CatBoostRegressor':
    # создаем массив индексов категориальных
    # признаков для CatBoost
    categorical_features_indices = np.where(
        X_train.dtypes == object)[0]
    # формируем обучающий пул
    train_pool = Pool(
        X_train, y_train,
        cat_features=categorical_features_indices)
    # обучаем модель
    model.fit(train_pool)

# в противном случае
else:
    model.fit(X_train, y_train)

# обучаем модель
ctbst_model.fit(X_train, y_train)

# получаем прогнозы
predictions = model.predict(X_test)
predictions = pd.Series(predictions)
predictions.index = X_test.index

# визуализируем прогнозы
if visualize:
    # задаем размер графика
    plt.figure(figsize=(8, 4))
    # настраиваем ориентацию меток оси x
    plt.xticks(rotation=90)
    # строим графики для обучающих данных,
    # тестовых данных, прогнозов модели

```

```

plt.plot(y_train[-last_n_train:],
        label='обучающая выборка')
plt.plot(predictions, color='red',
        label='прогнозы')
plt.plot(y_test, color='green',
        label='тестовая выборка')
# задаем координатную сетку
plt.grid()
# задаем легенду
plt.legend()
plt.show()

# вычисляем RMSE на тестовой выборке в текущей итерации
rmse = mean_squared_error(y_test, predictions, squared=False)
# добавим найденное в данной итерации значение RMSE в список
rmse_lst.append(rmse)
print(f"RMSE={rmse:.3f} на {cnt}-й итерации\n")

# вычисляем усредненное значение метрики
mean_rmse = np.mean(rmse_lst)
print(f"Усредненное значение RMSE={mean_rmse:.3f}")

```

Применяем нашу функцию `timeseries_cv()` – запускаем перекрестную проверку расширяющимся окном. Горизонт прогнозирования зададим равным 6 месяцам.

```

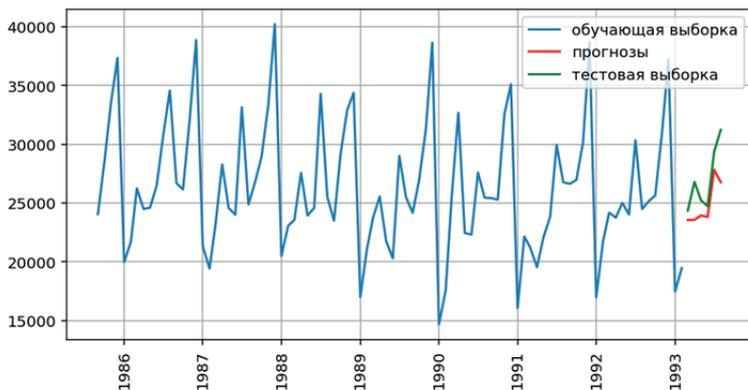
# применяем перекрестную проверку расширяющимся окном
timeseries_cv(wine_data, y_wine_data, ctbst_model,
              last_n_train=90, test_size=6,
              n_splits=3, gap=0)

```

```

TRAIN: ['1980-01-01', '1993-02-01'] TEST: ['1993-03-01', '1993-08-01']
Общее кол-во наблюдений: 176
Обучающая выборка: 158
Тестовая выборка: 6

```



```

RMSE=2444.249 на 1-й итерации

```

```

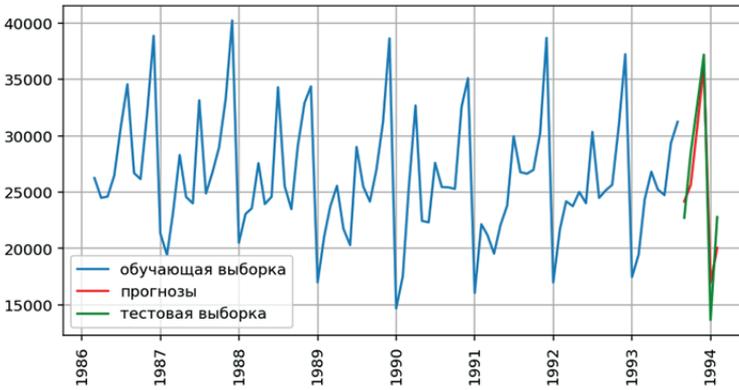
TRAIN: ['1980-01-01', '1993-08-01'] TEST: ['1993-09-01', '1994-02-01']

```

Общее кол-во наблюдений: 176

Обучающая выборка: 164

Тестовая выборка: 6



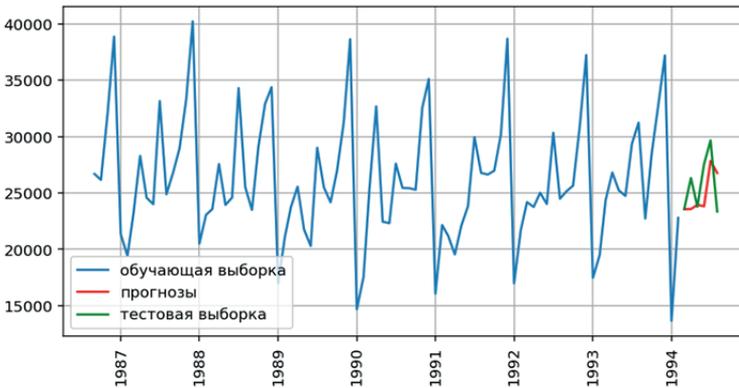
RMSE=2358.360 на 2-й итерации

TRAIN: ['1980-01-01', '1994-02-01'] TEST: ['1994-03-01', '1994-08-01']

Общее кол-во наблюдений: 176

Обучающая выборка: 170

Тестовая выборка: 6



RMSE=2468.700 на 3-й итерации

Усредненное значение RMSE=2423.770

Здесь мы видим довольно стабильное поведение модели, мы получаем примерно одинаковое качество на тестовой выборке для разных временных срезов.

Если мы хотим добавить лаги и скользящие статистики в качестве признаков, необходимо помнить, что мы не можем создать лаги и скользящие статистики, а потом запустить перекрестную проверку. Лаги и скользящие статистики должны заново создаваться на каждой итерации перекрестной проверки. Поэтому в этом случае тоже пишут собственный класс или собственную функцию перекрестной проверки на основе класса `TimeSeriesSplit`.

```

# пишем функцию перекрестной проверки расширяющимся/
# скользящим окном без гэпа / с гэпом
# с формированием лагов и скользящих
# статистик без протечек
def timeseries_cv_with_lags_and_moving_stats(
    data, y_data, model, lags_range=None, moving_stats_range=None,
    aggfunc='mean', seasonality=1, print_cv_scheme=True,
    print_features=True, visualize=True, last_n_train=5,
    max_train_size=None, test_size=None, n_splits=3,
    gap=0, fillna=None):
    """

```

Выполняет перекрестную проверку расширяющимся/  
скользящим окном без гэпа / с гэпом с  
формированием лагов и скользящих статистик.

Параметры

-----

data: pandas.DataFrame

Массив признаков.

y\_data: pandas.Series

Массив меток.

model:

Модель-регрессор: либо CatBoostRegressor, либо  
класс-регрессор библиотеки sklearn.

lags\_range:

Диапазон значений, значение – количество  
периодов для лагов.

moving\_stats\_range:

Диапазон значений, значение – ширина окна  
для скользящей статистики.

aggfunc: string, по умолчанию 'mean'

Агрегирующая функция для вычисления  
скользящей статистики.

seasonality: int, по умолчанию 1

Коэффициент сезонности для скользящей статистики.

print\_cv\_scheme: bool, по умолчанию True

Печать схемы перекрестной проверки.

print\_features: bool, по умолчанию True

Печать признаков.

visualize: bool, по умолчанию True

Визуализация прогнозов.

last\_n\_train: int, 5

Вывод n наблюдений обучающей выборки  
при визуализации прогнозов.

n\_splits: int, по умолчанию 3

Количество разбиений на обучающую и тестовую  
выборки.

max\_train\_size: int, по умолчанию None

Максимальный размер обучающей выборки.

test\_size: int, по умолчанию None

Максимальный размер тестовой выборки  
(определяется горизонтом прогнозирования).

gap: int, по умолчанию 0

Размер гэпа.

fillna: string, по умолчанию None

Стратегия импутации пропусков  
в обучающей выборке.

"""

```

if min(lags_range) < test_size:
    warnings.warn(
        "Количество периодов для лагов задавайте\n" +
        "равным или больше горизонта прогнозирования.")

if min(moving_stats_range) < test_size:
    warnings.warn(
        "Ширину окна для скользящих статистик задавайте\n" +
        "равной или больше горизонта прогнозирования.")

# создаем экземпляр класса TimeSeriesSplit
tscv = TimeSeriesSplit(
    max_train_size=max_train_size,
    test_size=test_size,
    n_splits=n_splits,
    gap=gap)

# создаем пустой список, куда записываем значения RMSE
rmse_lst = []

# запускаем цикл перекрестной проверки
for cnt, (train_index, test_index) in enumerate(
    tscv.split(data), 1):
    X_train, X_test = (data.iloc[train_index],
                      data.iloc[test_index])
    y_train, y_test = (y_data.iloc[train_index],
                      y_data.iloc[test_index])

    y_test_N = y_test.copy()

    # значения в тестовом массиве меток заменяем NaN
    y_test_N[:] = np.NaN
    # конкатенируем обучающий и тестовый массивы меток
    tmp_target = pd.concat([y_train, y_test_N])

    # конкатенируем обучающий и тестовый
    # массивы признаков
    concat_data = pd.concat([X_train, X_test])

    # печатаем схему валидации и сконкатенированный
    # массив меток
    if print_cv_scheme:
        print("-----")
        print("TRAIN:",
              [X_train.index[0].strftime('%Y-%m-%d'),
               X_train.index[-1].strftime('%Y-%m-%d')],
              "TEST:",
              [X_test.index[0].strftime('%Y-%m-%d'),
               X_test.index[-1].strftime('%Y-%m-%d')])
        print("\nОбщее кол-во наблюдений: %d" % (
            len(X_train) + len(X_test)))
        print("Обучающий набор: %d" % (len(X_train)))
        print("Тестовый набор: %d" % (len(X_test)))

    if print_features:
        print(f"\nЗащита:\n\n{tmp_target}\n")

```

```

# формируем лаги, количество периодов должно быть
# равно или превышать значение горизонта
# прогнозирования (если меньше, получим
# значения NaN)
if lags_range is not None:
    for i in lags_range:
        concat_data[f"Lag_{i}"] = tmp_target.shift(i)

# формируем скользящие статистики, ширина окна
# должна быть равна или превышать значение
# горизонта прогнозирования (если меньше,
# получим значения NaN)
if moving_stats_range is not None:
    for i in moving_stats_range:
        concat_data[f"Moving_{aggfunc}_{i}"] = \
            moving_stats(tmp_target, window=i, aggfunc=aggfunc,
                seasonality=seasonality)

# печатаем сконкатенированный массив признаков
# с новыми переменными - лагами и скользящими
# статистиками
if print_features:
    pattern = concat_data.columns.str.contains(
        'Lag|Moving_')
    feat = concat_data.columns[pattern]
    print(f"Доб. признаки:\n{concat_data[feat]}\n")

# сортируем столбцы для воспроизводимости
# (для CatBoost порядок генерации признаков
# влияет на результат)
concat_data = concat_data.sort_index(axis=1)

# снова выделяем обучающий и тестовый
# массивы признаков
X_train = concat_data[:-test_size]
X_test = concat_data[-test_size:]

# заполняем пропуски в обучающей выборке
if fillna == 'zero':
    X_train = X_train.fillna(0, axis=0)
if fillna == 'mean':
    X_train = X_train.fillna(X_train.mean(), axis=0)

# если модель - CatBoostRegressor
if model.__class__.__name__ == 'CatBoostRegressor':
    # создаем массив индексов категориальных
    # признаков для CatBoost
    categorical_features_indices = np.where(
        X_train.dtypes == object)[0]
    # формируем обучающий пул
    train_pool = Pool(
        X_train, y_train,
        cat_features=categorical_features_indices)
    # обучаем модель
    model.fit(train_pool)

```

```

# в противном случае
else:
    model.fit(X_train, y_train)

# получаем прогнозы
predictions = model.predict(X_test)
predictions = pd.Series(predictions)
predictions.index = X_test.index

# вычисляем RMSE на тестовой выборке
# в текущей итерации
rmse = mean_squared_error(
    y_test, predictions, squared=False)
# добавим найденное в данной итерации
# значение RMSE в список
rmse_lst.append(rmse)

print(f"\nRMSE={rmse:.3f} на {cnt}-й итерации\n")

# визуализируем прогнозы
if visualize:
    # задаем размер графика
    plt.figure(figsize=(8, 4))
    # настраиваем ориентацию меток оси x
    plt.xticks(rotation=90)
    # строим графики для обучающих данных,
    # тестовых данных, прогнозов модели
    plt.plot(y_train.iloc[-last_n_train:],
             label='обучающая выборка')
    plt.plot(predictions,
             color='red',
             label='прогнозы')
    plt.plot(y_test,
             color='green',
             label='тестовая выборка')
    # задаем координатную сетку
    plt.grid()
    # задаем легенду
    plt.legend()
    plt.show()

# расчет среднего значения RMSE
rmse_mean = np.mean(rmse_lst)
# выведем среднее значение RMSE
print(f"Среднее значение RMSE={rmse_mean:.3f}")

if print_features:
    # печатаем список признаков (порядок генерации
    # признаков может повлиять на результат CatBoost)
    feat_lst = concat_data.columns.tolist()
    print(f"\nСписок признаков:\n{feat_lst}")

```

Теперь мы применим нашу функцию `timeseries_cv_with_lags_and_moving_stats()` к данным о сделках с недвижимостью. Горизонт прогнозирования за-

дадим равным 4 дням. При этом подробно посмотрим, как формируются лаги и скользящие средние внутри каждой итерации перекрестной проверки.

*# применяем функцию перекрестной проверки расширяющимся  
# окном с формированием скользящих статистик и лагов  
# на каждой итерации*

```
timeseries_cv_with_lags_and_moving_stats(  
    flat_data, y_flat_data, model=ctbst_model,  
    lags_range=range(4, 6),  
    moving_stats_range=range(4, 6),  
    aggfunc='mean', test_size=4, n_splits=3)
```

**1-я итерация перекрестной проверки**

-----  
TRAIN: ['2016-08-22', '2016-08-29'] TEST: ['2016-08-30', '2016-09-02']

Общее кол-во наблюдений: 12  
Обучающий набор: 8  
Тестовый набор: 4

Защита:

Date_Create								
2016-08-22	1450000.00000	2016-08-26	1950000.00000	} /4 = 1882500		2016-08-25	1960000.00000	} /5 = 1898000
2016-08-23	1650000.00000	2016-08-27	1700000.00000			2016-08-26	1950000.00000	
2016-08-24	2250000.00000	2016-08-28	1550000.00000			2016-08-27	1700000.00000	
2016-08-25	1960000.00000	2016-08-29	2330000.00000			2016-08-28	1550000.00000	
2016-08-26	1950000.00000	2016-08-27	1700000.00000	} /3 = 1860000		2016-08-29	2330000.00000	} /4 = 1882500
2016-08-27	1700000.00000	2016-08-28	1550000.00000			2016-08-26	1950000.00000	
2016-08-28	1550000.00000	2016-08-29	2330000.00000	} /2 = 1940000		2016-08-27	1700000.00000	} /3 = 1860000
2016-08-29	2330000.00000	2016-08-28	1550000.00000			2016-08-28	1550000.00000	
2016-08-30	NaN	2016-08-29	2330000.00000	} /1 = 2330000		2016-08-29	2330000.00000	} /2 = 1940000
2016-08-31	NaN					2016-08-29	2330000.00000	
2016-09-01	NaN							
2016-09-02	NaN							

Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-22	NaN	NaN	NaN	NaN
2016-08-23	NaN	NaN	1450000.00000	1450000.00000
2016-08-24	NaN	NaN	1550000.00000	1550000.00000
2016-08-25	NaN	NaN	1783333.33333	1783333.33333
2016-08-26	1450000.00000	NaN	1827500.00000	1827500.00000
2016-08-27	1650000.00000	1450000.00000	1952500.00000	1852000.00000
2016-08-28	2250000.00000	1650000.00000	1965000.00000	1902000.00000
2016-08-29	1960000.00000	2250000.00000	1790000.00000	1882000.00000
2016-08-30	1950000.00000	1960000.00000	1882500.00000	1898000.00000
2016-08-31	1700000.00000	1950000.00000	1860000.00000	1882500.00000
2016-09-01	1550000.00000	1700000.00000	1940000.00000	1860000.00000
2016-09-02	2330000.00000	1550000.00000	2330000.00000	1940000.00000

Тестовая выборка

## 2-я итерация перекрестной проверки

-----  
 TRAIN: ['2016-08-22', '2016-09-02'] TEST: ['2016-09-03', '2016-09-06']

Общее кол-во наблюдений: 16  
 Обучающий набор: 12  
 Тестовый набор: 4

Защита:

Date_Create	2016-08-30	1900000.00000	} /4 = 3625000	2016-08-29	2330000.00000	} /5 = 3366000	
2016-08-22	1450000.00000	2016-08-31		3850000.00000	2016-08-30		1900000.00000
2016-08-23	1650000.00000	2016-09-01		5450000.00000	2016-08-31		3850000.00000
2016-08-24	2250000.00000	2016-09-02		3300000.00000	2016-09-01		5450000.00000
2016-08-25	1960000.00000	2016-08-31	3850000.00000	2016-08-30	1900000.00000	} /4 = 3625000	
2016-08-26	1950000.00000	2016-09-01	5450000.00000	2016-08-31	3850000.00000		
2016-08-27	1700000.00000	2016-09-02	3300000.00000	2016-09-01	5450000.00000		
2016-08-28	1550000.00000	2016-09-01	5450000.00000	2016-09-02	3300000.00000		
2016-08-29	2330000.00000	2016-09-02	3300000.00000	2016-08-31	3850000.00000	} /3 = 4200000	
2016-08-30	1900000.00000	2016-09-02	3300000.00000	2016-09-01	5450000.00000		
2016-08-31	3850000.00000			2016-09-02	3300000.00000		
2016-09-01	5450000.00000			2016-09-01	5450000.00000		
2016-09-02	3300000.00000			2016-09-02	3300000.00000	} /3 = 4200000	
2016-09-03	NaN			2016-09-01	5450000.00000		
2016-09-04	NaN			2016-09-02	3300000.00000		
2016-09-05	NaN						
2016-09-06	NaN					} /2 = 4375000	
				2016-09-01	5450000.00000		
				2016-09-02	3300000.00000		

Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-22	NaN	NaN	NaN	NaN
2016-08-23	NaN	NaN	1450000.00000	1450000.00000
2016-08-24	NaN	NaN	1550000.00000	1550000.00000
2016-08-25	NaN	NaN	1783333.33333	1783333.33333
2016-08-26	1450000.00000	NaN	1827500.00000	1827500.00000
2016-08-27	1650000.00000	1450000.00000	1952500.00000	1852000.00000
2016-08-28	2250000.00000	1650000.00000	1965000.00000	1902000.00000
2016-08-29	1960000.00000	2250000.00000	1790000.00000	1882000.00000
2016-08-30	1950000.00000	1960000.00000	1882500.00000	1898000.00000
2016-08-31	1700000.00000	1950000.00000	1870000.00000	1886000.00000
2016-09-01	1550000.00000	1700000.00000	2407500.00000	2266000.00000
2016-09-02	2330000.00000	1550000.00000	3382500.00000	3016000.00000
2016-09-03	1900000.00000	2330000.00000	3625000.00000	3366000.00000
2016-09-04	3850000.00000	1900000.00000	4200000.00000	3625000.00000
2016-09-05	5450000.00000	3850000.00000	4375000.00000	4200000.00000
2016-09-06	3300000.00000	5450000.00000	3300000.00000	4375000.00000

Тестовая выборка

**3-я итерация перекрестной проверки**

-----  
 TRAIN: ['2016-08-22', '2016-09-06'] TEST: ['2016-09-07', '2016-09-10']

Общее кол-во наблюдений: 20  
 Обучающий набор: 16  
 Тестовый набор: 4

Защита:

Date_Create							
2016-08-22	1450000.00000	2016-09-03	4950000.00000	} /4 = 3150000	2016-09-02	3300000.00000	} /5 = 3180000
2016-08-23	1650000.00000	2016-09-04	3500000.00000		2016-09-03	4950000.00000	
2016-08-24	2250000.00000	2016-09-05	2300000.00000		2016-09-04	3500000.00000	
2016-08-25	1960000.00000	2016-09-06	1850000.00000		2016-09-05	2300000.00000	
2016-08-26	1950000.00000				2016-09-06	1850000.00000	
2016-08-27	1700000.00000	2016-09-04	3500000.00000	} /3 = 2550000	2016-09-03	4950000.00000	} /4 = 3150000
2016-08-28	1550000.00000	2016-09-05	2300000.00000		2016-09-04	3500000.00000	
2016-08-29	2330000.00000	2016-09-06	1850000.00000		2016-09-05	2300000.00000	
2016-08-30	1900000.00000				2016-09-06	1850000.00000	
2016-08-31	3850000.00000	2016-09-05	2300000.00000	} /2 = 2075000	2016-09-04	3500000.00000	} /3 = 2550000
2016-09-01	5450000.00000	2016-09-06	1850000.00000		2016-09-05	2300000.00000	
2016-09-02	3300000.00000				2016-09-06	1850000.00000	} /2 = 2075000
2016-09-03	4950000.00000						
2016-09-04	3500000.00000						
2016-09-05	2300000.00000						
2016-09-06	1850000.00000						
2016-09-07	NaN						
2016-09-08	NaN						
2016-09-09	NaN						
2016-09-10	NaN						

Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-22	NaN	NaN	NaN	NaN
2016-08-23	NaN	NaN	1450000.00000	1450000.00000
2016-08-24	NaN	NaN	1550000.00000	1550000.00000
2016-08-25	NaN	NaN	1783333.33333	1783333.33333
2016-08-26	1450000.00000	NaN	1827500.00000	1827500.00000
2016-08-27	1650000.00000	1450000.00000	1952500.00000	1852000.00000
2016-08-28	2250000.00000	1650000.00000	1965000.00000	1902000.00000
2016-08-29	1960000.00000	2250000.00000	1790000.00000	1882000.00000
2016-08-30	1950000.00000	1960000.00000	1882500.00000	1898000.00000
2016-08-31	1700000.00000	1950000.00000	1870000.00000	1886000.00000
2016-09-01	1550000.00000	1700000.00000	2407500.00000	2266000.00000
2016-09-02	2330000.00000	1550000.00000	3382500.00000	3016000.00000
2016-09-03	1900000.00000	2330000.00000	3625000.00000	3366000.00000
2016-09-04	3850000.00000	1900000.00000	4387500.00000	3890000.00000
2016-09-05	5450000.00000	3850000.00000	4300000.00000	4210000.00000
2016-09-06	3300000.00000	5450000.00000	3512500.00000	3900000.00000
2016-09-07	4950000.00000	3300000.00000	3150000.00000	3180000.00000
2016-09-08	3500000.00000	4950000.00000	2550000.00000	3150000.00000
2016-09-09	2300000.00000	3500000.00000	2075000.00000	2550000.00000
2016-09-10	1850000.00000	2300000.00000	1850000.00000	2075000.00000

Тестовая выборка

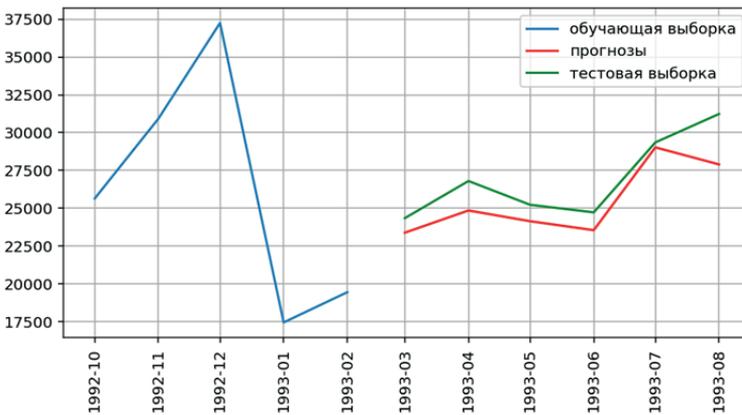
Теперь применим перекрестную проверку расширяющимся окном с формированием лагов и скользящих статистик для продаж вина в Австралии.

```
# применяем функцию перекрестной проверки расширяющимся
# окном с формированием скользящих статистик и лагов
# на каждой итерации
timeseries_cv_with_lags_and_moving_stats(
    wine_data, y_wine_data, model=ctbst_model,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    print_features=False,
    aggfunc='mean', test_size=6, n_splits=3)
```

-----  
 TRAIN: ['1980-01-01', '1993-02-01'] TEST: ['1993-03-01', '1993-08-01']

Общее кол-во наблюдений: 164  
 Обучающий набор: 158  
 Тестовый набор: 6

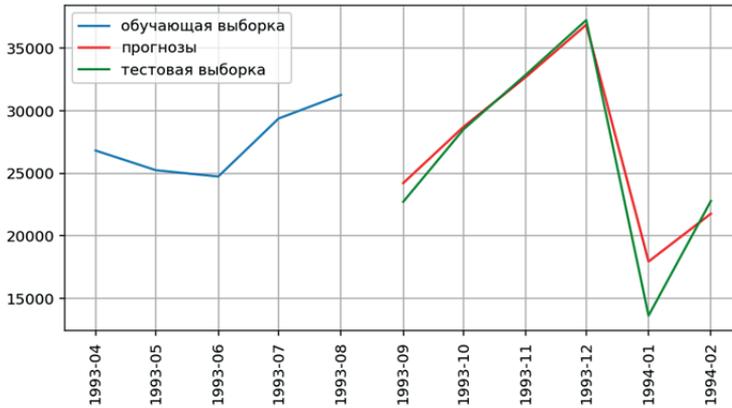
RMSE=1755.159 на 1-й итерации



-----  
 TRAIN: ['1980-01-01', '1993-08-01'] TEST: ['1993-09-01', '1994-02-01']

Общее кол-во наблюдений: 170  
 Обучающий набор: 164  
 Тестовый набор: 6

RMSE=1915.940 на 2-й итерации



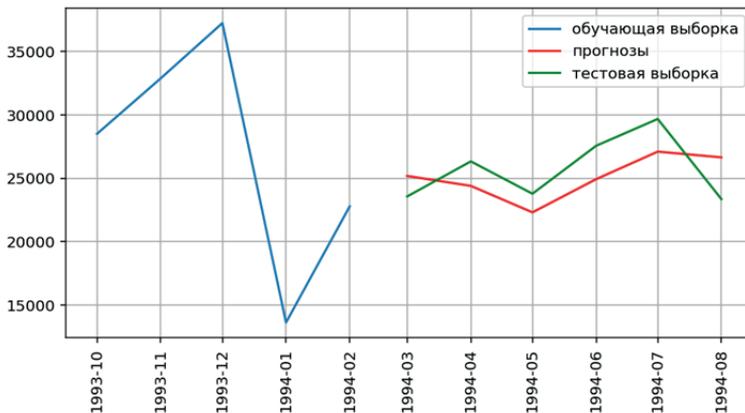
TRAIN: ['1980-01-01', '1994-02-01'] TEST: ['1994-03-01', '1994-08-01']

Общее кол-во наблюдений: 176

Обучающий набор: 170

Тестовый набор: 6

RMSE=2335.542 на 3-й итерации



Среднее значение RMSE=2002.214

Усредненное значение метрики говорит о том, что добавление лагов и скользящих средних улучшило качество прогнозов, однако значение метрики для третьего временного периода сильно отличается от значений метрик для первых двух временных периодов. Мы должны попробовать разное количество разбиений, нам нужно посмотреть, как модель прогнозирует на большем количестве временных отрезков. Если мы видим, что значения метрик сильно варьируют по периодам, можно сделать следующее. В функцию перекрестной проверки добавляем вычисление важностей признаков и отбираем признаки,

которые дают наибольшие усредненные важности (на основе информационного выигрыша/веса или частоты использования). Поскольку отбор признаков можно считать настройкой гиперпараметров, вам нужно разбить данные на обучающую и тестовую выборки с учетом временной структуры и запускать перекрестную проверку на обучающей выборке. Проверочные выборки перекрестной проверки вы используете для настройки гиперпараметров, в данном случае для отбора определенных комбинаций лагов и скользящих статистик, и затем проверяете качество модели с отобранной наилучшей комбинацией лагов и скользящих статистик на тестовой выборке. Нам нужно отобрать такие лаги и скользящие статистики, которые будут давать стабильно высокое качество прогнозов для *разных* временных периодов. В нашем же примере разброс значений метрик, вероятно, обусловлен тем, что выбранные лаги и скользящие средние хорошо работают на одном временном периоде и гораздо хуже в остальных, поскольку выбраны они были на основе валидационной выборки (см. предыдущий раздел), которая охватывала один конкретный временной период. Таким образом, при использовании разбиения на обучающую, проверочную и тестовую выборки существует риск получить лаги и скользящие статистики, настроенные на один конкретный временной период. Этот риск увеличивается при коротких горизонтах прогнозирования.

В сложных проектах практически всегда вам придется писать гораздо более сложные собственные функции/классы перекрестной проверки, под капотом которых будет реализовано вычисление лагов, скользящих статистик, вычисление важностей признаков, логарифмирование зависимой переменной и детрендинг (удаление из зависимой переменной – временного ряда тренда, – спрогнозированной отдельной, как правило, линейной моделью) с последующими обратными преобразованиями (экспоненцированием и добавлением тренда к прогнозам).

## 7.22.2. Перекрестная проверка скользящим окном

При необходимости обучение модели в каждом разбиении можно сделать последовательным, используя в каждой итерации для обучения фиксированное количество наиболее свежих (поздних) наблюдений, предшествующих точке разбиения. Таким образом, в каждой новой итерации мы будем обучаться на более свежих данных, обучающая выборка каждый раз сдвигается вперед по временной оси (обычно на горизонт прогнозирования), и такой способ проверки называют перекрестной проверкой скользящим окном (sliding/rolling window).

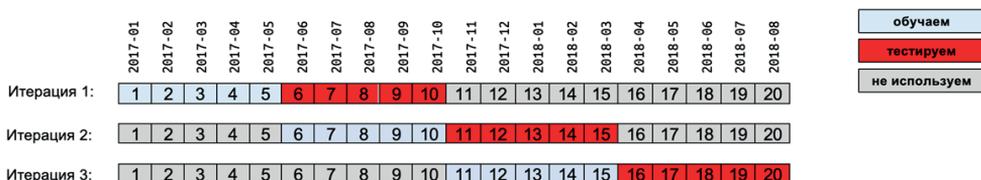
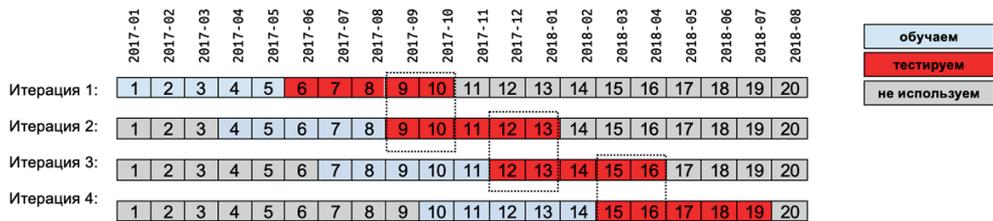


Рис. 66 Классическая перекрестная проверка скользящим окном

Видим, что с каждой итерацией обучающая выборка использует все более свежие наблюдения, при этом для тестирования мы каждый раз берем совершенно новые более поздние наблюдения.

Однако перекрестную проверку скользящим окном можно модифицировать так, чтобы обучающая выборка сдвигалась вперед не на весь горизонт прогнозирования, а на половину или на треть, и тогда в тестовую выборку попадут наблюдения, уже попадавшие в тестовую выборку на предыдущей итерации. Это позволяет управлять скоростью обновления модели, лучше выявлять аномальные, нетипичные наблюдения, которые плохо предсказываются, точнее определять момент ухудшения качества модели.

В библиотеке `scikit-learn` с помощью класса `TimeSeriesSplit` ее нельзя корректно реализовать (для реализации используйте параметр `step_length` класса `SlidingWindowSplitter` библиотеки `sktime`).



**Рис. 67** Модифицированная перекрестная проверка скользящим окном (в тестовую выборку попадают наблюдения, уже попадавшие в тестовую выборку на предыдущей итерации)

С помощью параметра `max_train_size` класса `TimeSeriesSplit` можно настроить перекрестную проверку скользящим окном. Мы зафиксируем максимальный размер обучающей выборки.

```
# создаем экземпляр класса TimeSeriesSplit,
# фиксируя максимальный размер обучающей выборки
slidingcv = TimeSeriesSplit(n_splits=3, max_train_size=5, test_size=4)

# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train_index, test_index in slidingcv.split(flat_data):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = (flat_data.iloc[train_index],
                      flat_data.iloc[test_index])
    y_train, y_test = (y_flat_data.iloc[train_index],
                      y_flat_data.iloc[test_index])
    print("Общее кол-во наблюдений: %d" % len(flat_data))
    print("Обучающая выборка: %d" % len(X_train))
    print("Тестовая выборка: %d" % len(X_test))
    print("")

TRAIN: [3 4 5 6 7] TEST: [ 8  9 10 11]
Общее кол-во наблюдений: 20
Обучающая выборка: 5
Тестовая выборка: 4
```

TRAIN: [ 7 8 9 10 11] TEST: [12 13 14 15]

Общее кол-во наблюдений: 20

Обучающая выборка: 5

Тестовая выборка: 4

TRAIN: [11 12 13 14 15] TEST: [16 17 18 19]

Общее кол-во наблюдений: 20

Обучающая выборка: 5

Тестовая выборка: 4

Перед нами – перекрестная проверка скользящим окном. На каждой итерации мы разбиваем набор на обучающую и тестовую выборки так, чтобы в обучающую выборку попали более ранние наблюдения, а в тестовую выборку – более поздние. Размеры обучающей и тестовой выборок остаются постоянными. В каждой новой итерации мы будем обучаться на более свежих данных.

Давайте применим перекрестную проверку скользящим окном к данным о продажах вина в Австралии. Размер обучающей выборки зададим равным 72 месяцам (6 годам), т.е. каждый раз мы будем обучаться на выборке длиной 72 месяца. Горизонт прогнозирования будет прежним – 6 месяцев.

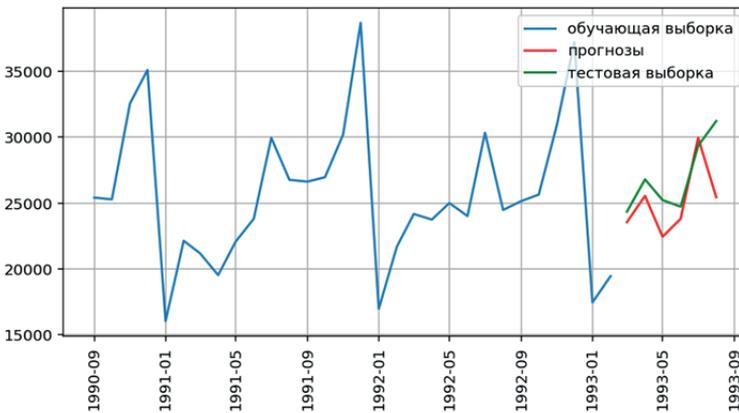
```
# применяем перекрестную проверку скользящим окном
timeseries_cv(wine_data, y_wine_data, ctbst_model,
              last_n_train=30, max_train_size=72,
              test_size=6, n_splits=3, gap=0)
```

TRAIN: ['1987-03-01', '1993-02-01'] TEST: ['1993-03-01', '1993-08-01']

Общее кол-во наблюдений: 176

Обучающая выборка: 72

Тестовая выборка: 6



RMSE=2725.442 на 1-й итерации

TRAIN: ['1987-09-01', '1993-08-01'] TEST: ['1993-09-01', '1994-02-01']

Общее кол-во наблюдений: 176

Обучающая выборка: 72

Тестовая выборка: 6



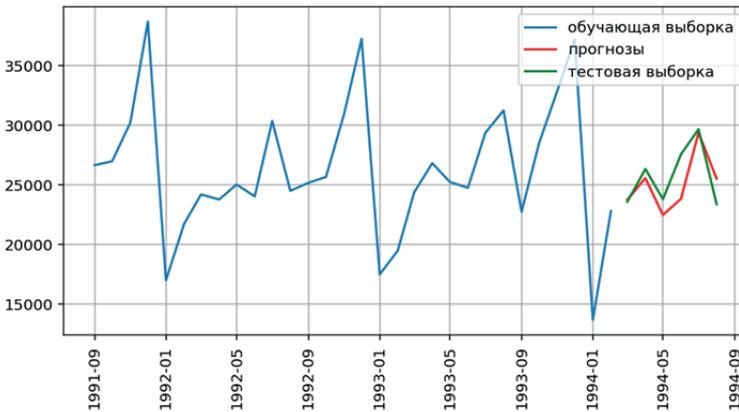
RMSE=2032.659 на 2-й итерации

TRAIN: ['1988-03-01', '1994-02-01'] TEST: ['1994-03-01', '1994-08-01']

Общее кол-во наблюдений: 176

Обучающая выборка: 72

Тестовая выборка: 6



RMSE=1871.080 на 3-й итерации

Усредненное значение RMSE=2209.727

Если мы хотим добавить лаги и скользящие статистики, напомним, они должны заново формироваться на каждой итерации перекрестной проверки. Функция `timeseries_cv_with_lags_and_moving_stats()` позволяет выполнить перекрестную проверку скользящим окном с формированием лагов и скользящих статистик внутри каждой итерации.

Давайте применим эту функцию к данным о сделках с недвижимостью. Максимальный размер обучающей выборки зададим равным 6 дням. Горизонт прогнозирования задим равным 4 дням.

```
# применяем функцию перекрестной проверки скользящим
# окном с формированием скользящих статистик и лагов
# на каждой итерации
```

```
timeseries_cv_with_lags_and_moving_stats(
    flat_data, y_flat_data, model=ctbst_model,
    lags_range=range(4, 6),
    moving_stats_range=range(4, 6),
    aggfunc='mean', max_train_size=6,
    test_size=4, n_splits=3)
```

```
-----
TRAIN: ['2016-08-24', '2016-08-29'] TEST: ['2016-08-30', '2016-09-02']
```

```
Общее кол-во наблюдений: 10
Обучающий набор: 6
Тестовый набор: 4
```

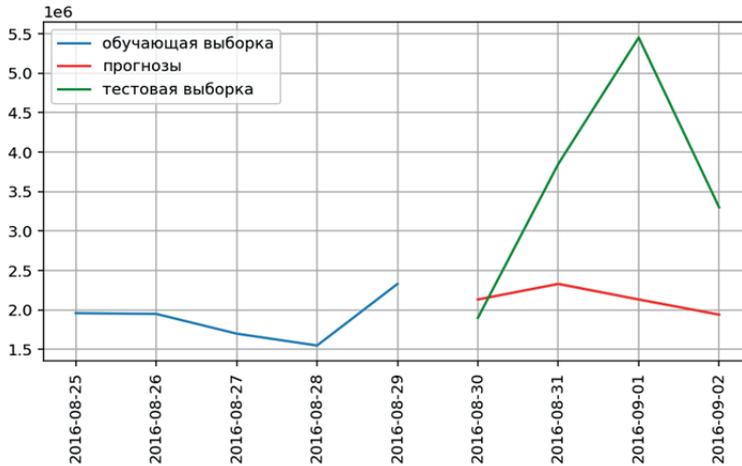
Защита:

```
Date_Create
2016-08-24  2250000.00000
2016-08-25  1960000.00000
2016-08-26  1950000.00000
2016-08-27  1700000.00000
2016-08-28  1550000.00000
2016-08-29  2330000.00000
2016-08-30           NaN
2016-08-31           NaN
2016-09-01           NaN
2016-09-02           NaN
Name: Value_abs, dtype: float64
```

Доб. признаки:

	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
Date_Create				
2016-08-24	NaN	NaN	NaN	NaN
2016-08-25	NaN	NaN	2250000.00000	2250000.00000
2016-08-26	NaN	NaN	2105000.00000	2105000.00000
2016-08-27	NaN	NaN	2053333.33333	2053333.33333
2016-08-28	2250000.00000	NaN	1965000.00000	1965000.00000
2016-08-29	1960000.00000	2250000.00000	1790000.00000	1882000.00000
2016-08-30	1950000.00000	1960000.00000	1882500.00000	1898000.00000
2016-08-31	1700000.00000	1950000.00000	1860000.00000	1882500.00000
2016-09-01	1550000.00000	1700000.00000	1940000.00000	1860000.00000
2016-09-02	2330000.00000	1550000.00000	2330000.00000	1940000.00000

RMSE=1950412.781 на 1-й итерации



TRAIN: ['2016-08-28', '2016-09-02'] TEST: ['2016-09-03', '2016-09-06']

Общее кол-во наблюдений: 10

Обучающий набор: 6

Тестовый набор: 4

Защита:

Date\_Create

2016-08-28 1550000.00000

2016-08-29 2330000.00000

2016-08-30 1900000.00000

2016-08-31 3850000.00000

2016-09-01 5450000.00000

2016-09-02 3300000.00000

2016-09-03 NaN

2016-09-04 NaN

2016-09-05 NaN

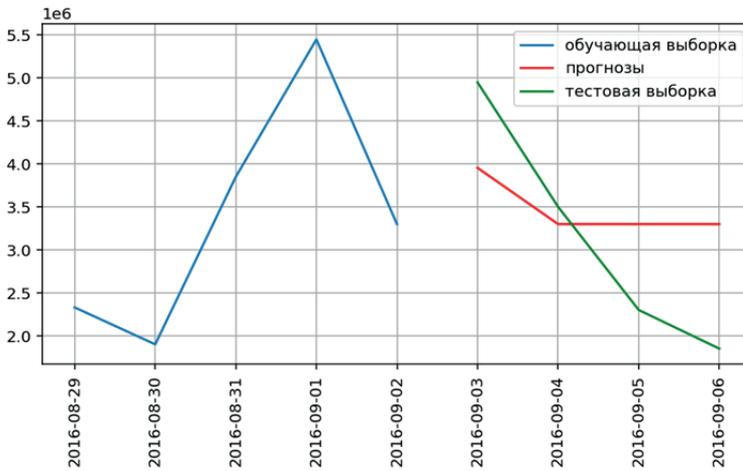
2016-09-06 NaN

Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-28	NaN	NaN	NaN	NaN
2016-08-29	NaN	NaN	1550000.00000	1550000.00000
2016-08-30	NaN	NaN	1940000.00000	1940000.00000
2016-08-31	NaN	NaN	1926666.66667	1926666.66667
2016-09-01	1550000.00000	NaN	2407500.00000	2407500.00000
2016-09-02	2330000.00000	1550000.00000	3382500.00000	3016000.00000
2016-09-03	1900000.00000	2330000.00000	3625000.00000	3366000.00000
2016-09-04	3850000.00000	1900000.00000	4200000.00000	3625000.00000
2016-09-05	5450000.00000	3850000.00000	4375000.00000	4200000.00000
2016-09-06	3300000.00000	5450000.00000	3300000.00000	4375000.00000

RMSE=1016567.260 на 2-й итерации



-----  
 TRAIN: ['2016-09-01', '2016-09-06'] TEST: ['2016-09-07', '2016-09-10']

Общее кол-во наблюдений: 10

Обучающий набор: 6

Тестовый набор: 4

Защита:

Date\_Create

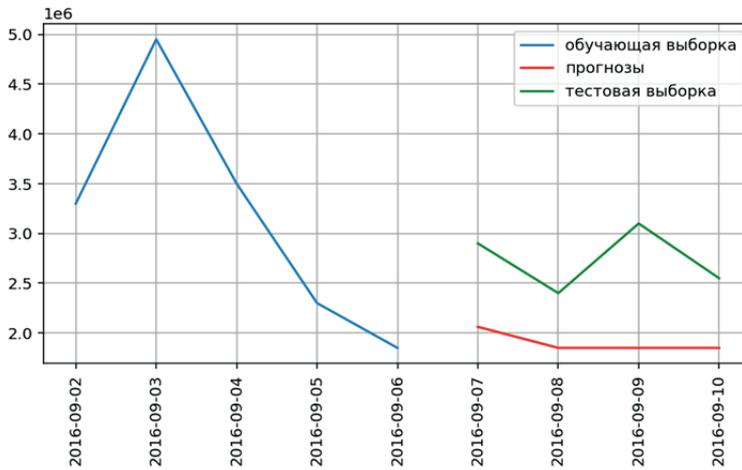
```

2016-09-01  5450000.00000
2016-09-02  3300000.00000
2016-09-03  4950000.00000
2016-09-04  3500000.00000
2016-09-05  2300000.00000
2016-09-06  1850000.00000
2016-09-07      NaN
2016-09-08      NaN
2016-09-09      NaN
2016-09-10      NaN
Name: Value_abs, dtype: float64
    
```

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-09-01	NaN	NaN	NaN	NaN
2016-09-02	NaN	NaN	5450000.00000	5450000.00000
2016-09-03	NaN	NaN	4375000.00000	4375000.00000
2016-09-04	NaN	NaN	4566666.66667	4566666.66667
2016-09-05	5450000.00000	NaN	4300000.00000	4300000.00000
2016-09-06	3300000.00000	5450000.00000	3512500.00000	3900000.00000
2016-09-07	4950000.00000	3300000.00000	3150000.00000	3180000.00000
2016-09-08	3500000.00000	4950000.00000	2550000.00000	3150000.00000
2016-09-09	2300000.00000	3500000.00000	2075000.00000	2550000.00000
2016-09-10	1850000.00000	2300000.00000	1850000.00000	2075000.00000

RMSE=874300.093 на 3-й итерации



Среднее значение RMSE=1280426.711

Список признаков:

```
['Lag_4', 'Lag_5', 'Moving_mean_4', 'Moving_mean_5']
```

Наконец, давайте применим функцию `timeseries_cv_with_lags_and_moving_stats()` к данным о продажах вина в Австралии.

```
# применяем функцию перекрестной проверки скользящим
# окном с формированием скользящих статистик и лагов
# на каждой итерации
```

```
timeseries_cv_with_lags_and_moving_stats(
    wine_data, y_wine_data, model=ctbst_model,
    lags_range=list(range(6, 11, 2)),
    moving_stats_range=[6, 8],
    print_features=False, aggfunc='mean',
    max_train_size=72, test_size=6, n_splits=3)
```

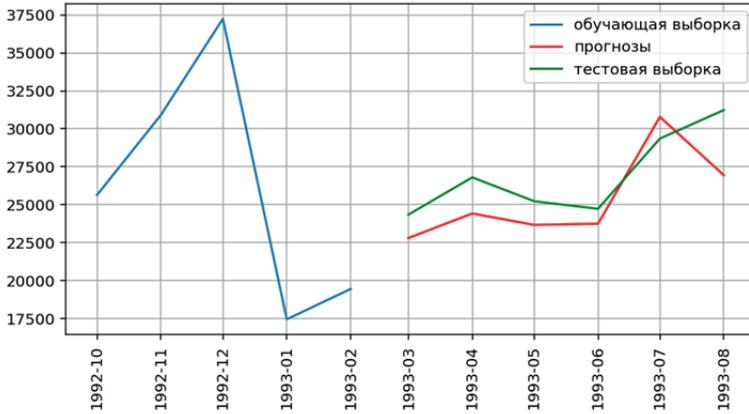
```
-----
TRAIN: ['1987-03-01', '1993-02-01'] TEST: ['1993-03-01', '1993-08-01']
```

Общее кол-во наблюдений: 78

Обучающий набор: 72

Тестовый набор: 6

RMSE=2302.505 на 1-й итерации



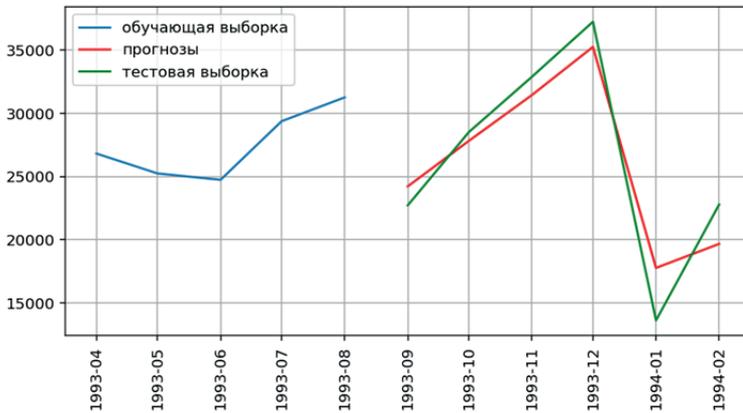
TRAIN: ['1987-09-01', '1993-08-01'] TEST: ['1993-09-01', '1994-02-01']

Общее кол-во наблюдений: 78

Обучающий набор: 72

Тестовый набор: 6

RMSE=2431.464 на 2-й итерации



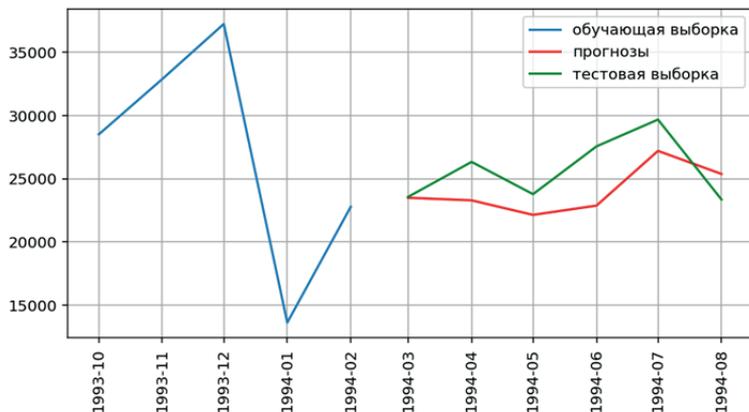
TRAIN: ['1988-03-01', '1994-02-01'] TEST: ['1994-03-01', '1994-08-01']

Общее кол-во наблюдений: 78

Обучающий набор: 72

Тестовый набор: 6

RMSE=2703.726 на 3-й итерации



Среднее значение  $RMSE=2479.232$

Усредненное значение метрики говорит о том, что добавление лагов и скользящих средних при таком сценарии проверки не улучшило качество модели. Здесь мы тоже должны попробовать разное количество разбиений, нам нужно посмотреть, как модель прогнозирует на большем количестве временных отрезков. Затем можно попробовать отбор признаков.

Часто возникает вопрос: в каких случаях использовать перекрестную проверку расширяющимся окном, а в каких случаях использовать перекрестную проверку скользящим окном? При использовании перекрестной проверки расширяющимся окном модель в большей степени нацелена на обнаружение глобальных паттернов и менее склонна к изменениям, т. е. более консервативна. При использовании перекрестной проверки скользящим окном используется меньше данных, модель быстрее меняет поведение, т. е. менее консервативна. В ситуации, когда вы уверены, что процесс, генерирующий данные, изменился или неоднократно менялся в течение периода, охватывающего исторические данные, используйте перекрестную проверку скользящим окном. Причиной изменения процесса, генерирующего данные, могут быть как внутренние факторы (изменение кредитной политики, изменение маркетинговой политики), так и внешние (изменение макроэкономической ситуации, повышение курса доллара, увеличение безработицы, инфляция). Для высококонкурентных рынков (например, рынок мобильных устройств), где выход нового устройства, новой модели устройств может сильно поменять потребительское поведение, обязательно необходимо попробовать перекрестную проверку скользящим окном. Для рынков товаров с низкой вовлеченностью (например, для строительно-отделочных материалов) в ситуации, когда вы уверены или у вас есть доказательства, что процесс, генерирующий данные, остается неизменным или претерпевает несущественные изменения, перекрестная проверка расширяющимся окном может быть более полезной. Здесь можно привести хрестоматийный пример с прогнозированием цен на недвижимость. У нас есть данные с 2008 по 2019 г. Нужно прогнозировать цены на 30 дней вперед в 2020 году. Берем перекрестную проверку расширяющимся окном с 12 разбиениями и перекрестную проверку скользящим окном с 12 разби-

ениями. Перекрестная проверка скользящим окном дает гораздо лучшее качество. Это говорит о том, что процесс, генерирующий данные, менялся. Например, в 2014 году после присоединения Крыма были введены санкции, которые отразились на социально-экономическом состоянии страны, темпах роста. Это событие так или иначе повлияло на поведение потребителей. Поэтому информация до 2014 года уже не актуальна, те паттерны поведения, которые были с 2008 по 2014 г., вряд ли будут воспроизводиться в будущем (а именно на это нацелена перекрестная проверка расширяющимся окном, она в силу накопления наблюдений «помнит» паттерны прошлого). Перекрестная проверка скользящим окном позволяет определить временную точку, после которой процесс, генерирующий данные, поменялся. Например, мы можем попробовать перекрестную проверку расширяющимся окном, но при этом брать данные не с 2008 года, а с 2014 года, вновь применить перекрестную проверку скользящим окном, но первая обучающая выборка будет начинаться не с 2008 года, а с 2014 года.

Во временных рядах перекрестная проверка, которую вы применяете, является прообразом вашей производственной системы. Если вы применяли для валидации перекрестную проверку расширяющимся окном, то и в производстве вы каждый раз должны обучать модель на обучающей выборке возрастающего объема и обновлять в том же темпе, что обновляли в ходе перекрестной проверки (на весь горизонт прогнозирования, на половину горизонта прогнозирования, на четверть горизонта прогнозирования и т. д.). Если вы применяли для валидации перекрестную проверку скользящим окном, то и в производстве вы должны каждый раз обучать модель на обучающей выборке все более свежих данных одинакового объема и обновлять в том же темпе, что обновляли в ходе перекрестной проверки (на весь горизонт прогнозирования, на половину горизонта прогнозирования, на четверть горизонта прогнозирования и т. д.).

### 7.22.3. Перекрестная проверка расширяющимся/скользящим окном с гэпом

Ранее рассмотренные схемы валидации не учитывают наличие гэпа. Гэп – это количество пропущенных интервалов времени между концом обучающей выборки и началом тестовой выборки (выборки выстроены в хронологическом порядке). Гэп означает недоступность самых свежих данных для обучения – достаточно распространенное явление на практике. Допустим, у нас есть процесс сбора данных, при этом данные за последнюю неделю для нас еще недоступны, но нам нужно делать прогноз.

Приведем пример. Предположим, у нас есть обучающие данные о ежедневных продажах за 1/1/2020, 2/1/2020, 3/1/2020, 4/1/2020. Всего для обучения доступны 4 дня. Кроме того, тестовые данные начинаются с 6/1/2020. У нас лишь 1 день в тестовых данных. Предыдущий день (5/1/2020) не относится к обучающим данным. Это день, который нельзя использовать для обучения (потому что информация для этого дня может быть недоступна во время скоринга). Этот день также нельзя использовать для получения информации (например, исторических лагов) для тестовых данных. Здесь интервал времени (или единица времени) составляет 1 день. Это временной интервал, который разделяет разные наблюдения/строки в данных. Таким образом, есть 4 временных интервала/единицы для обучающих данных и 1 временной интервал/единица

для тестовых данных плюс гэп. Чтобы оценить гэп между концом обучающих данных и началом тестовых данных, применяем следующую формулу:

$$Gap = \min(\text{номер интервала в тестовом наборе}) - \max(\text{номер интервала в обучающем наборе}) - 1.$$

В нашем случае  $\min(\text{номер интервала в тестовом наборе})$  равен 6 (или 6/1/2020). Это самый ранний (и единственный) день в тестовых данных.  $\max(\text{номер интервала в обучающем наборе})$  равен 4 (или 4/1/2020). Это самый последний (самый недавний) день в обучающих данных. Поэтому гэп равен 1 временному интервалу (или 1 дню в нашем случае), поскольку  $Gap = 6 - 4 - 1 = 1$ .

Мы можем выполнить перекрестную проверку расширяющимся окном с гэпом.



**Рис. 68** Сравнение перекрестной проверки расширяющимся окном без гэпа и перекрестной проверки расширяющимся окном с гэпом

Кроме того, мы можем выполнить перекрестную проверку скользящим окном с гэпом.



**Рис. 69** Сравнение перекрестной проверки скользящим окном без гэпа и перекрестной проверки скользящим окном с гэпом

Перекрестную проверку расширяющимся/скользящим окном с гэпом можно реализовать с помощью параметра `gap`.

Давайте проиллюстрируем перекрестную проверку расширяющимся окном с гэпом.

```
# фиксируем размер гэта
expand_gap_cv = TimeSeriesSplit(n_splits=3, test_size=4, gap=2)
```

```
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train_index, test_index in expand_gap_cv.split(flat_data):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = (flat_data.iloc[train_index],
                      flat_data.iloc[test_index])
    y_train, y_test = (y_flat_data.iloc[train_index],
                      y_flat_data.iloc[test_index])
    print("Общее кол-во наблюдений: %d" % len(flat_data))
    print("Обучающая выборка: %d" % len(X_train))
    print("Тестовая выборка: %d" % len(X_test))
    print("")
```

```
TRAIN: [0 1 2 3 4 5] TEST: [ 8  9 10 11]
Общее кол-во наблюдений: 20
Обучающая выборка: 6
Тестовая выборка: 4
```

```
TRAIN: [0 1 2 3 4 5 6 7 8 9] TEST: [12 13 14 15]
Общее кол-во наблюдений: 20
Обучающая выборка: 10
Тестовая выборка: 4
```

```
TRAIN: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13] TEST: [16 17 18 19]
Общее кол-во наблюдений: 20
Обучающая выборка: 14
Тестовая выборка: 4
```

Перед нами – перекрестная проверка расширяющимся окном с гэпом. На каждой итерации мы разбиваем набор на обучающую и тестовую выборки так, чтобы в обучающую выборку попали более ранние наблюдения, а в тестовую выборку – более поздние. Размер тестовой выборки остается постоянным. Количество наблюдений, использованных для обучения в каждой итерации, растет с числом итераций, предоставляя все больший объем данных для обучения. При этом у нас есть гэп, мы каждый раз исключаем по 2 наблюдения (`gap=2`) в конце обучающей выборки.

Теперь проиллюстрируем перекрестную проверку скользящим окном с гэпом.

```
# фиксируем размер гэта
sliding_gap_cv = TimeSeriesSplit(n_splits=3, max_train_size=5,
                                test_size=4, gap=2)
```

```
# взглянем на индексы наблюдений, попавших в обучающую
# и тестовую выборки, по каждой из трех итераций
for train_index, test_index in sliding_gap_cv.split(flat_data):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
X_train, X_test = (flat_data.iloc[train_index],
                  flat_data.iloc[test_index])
y_train, y_test = (y_flat_data.iloc[train_index],
                  y_flat_data.iloc[test_index])
print("Общее кол-во наблюдений: %d" % len(flat_data))
print("Обучающая выборка: %d" % len(X_train))
print("Тестовая выборка: %d" % len(X_test))
print("")
```

```
TRAIN: [1 2 3 4 5] TEST: [ 8  9 10 11]
Общее кол-во наблюдений: 20
Обучающая выборка: 5
Тестовая выборка: 4
```

```
TRAIN: [5 6 7 8 9] TEST: [12 13 14 15]
Общее кол-во наблюдений: 20
Обучающая выборка: 5
Тестовая выборка: 4
```

```
TRAIN: [ 9 10 11 12 13] TEST: [16 17 18 19]
Общее кол-во наблюдений: 20
Обучающая выборка: 5
Тестовая выборка: 4
```

Перед нами – перекрестная проверка скользящим окном с гэпом. На каждой итерации мы разбиваем набор на обучающую и тестовую выборки так, чтобы в обучающую выборку попали более ранние наблюдения, а в тестовую выборку – более поздние. Размеры обучающей и тестовой выборок остаются постоянными (может быть незначительное варьирование размера обучающей выборки). В каждой новой итерации мы будем обучаться на более свежих данных. При этом у нас есть гэп, мы каждый раз исключаем по два наблюдения ( $gap=2$ ) в конце обучающей выборки.

Обе функции, `timeseries_cv()` и `timeseries_cv_with_lags_and_moving_stats()`, поддерживают перекрестную проверку расширяющимся/скользящим окном с гэпом.

На примере данных о сделках с недвижимостью проиллюстрируем, как выполняется перекрестная проверка расширяющимся окном с гэпом, когда используются лаги и скользящие средние. При этом подробно посмотрим, как формируются лаги и скользящие средние в первой итерации перекрестной проверки.

```
# применяем функцию перекрестной проверки расширяющимся
# окном с гэпом, формируем скользящие статистики и лаги
# на каждой итерации
timeseries_cv_with_lags_and_moving_stats(
    flat_data, y_flat_data, model=ctbst_model,
    lags_range=range(4, 6),
    moving_stats_range=range(4, 6),
    aggfunc='mean', test_size=4, n_splits=3, gap=2)
```

1-я итерация перекрестной проверки

TRAIN: ['2016-08-22', '2016-08-27'] TEST: ['2016-08-30', '2016-09-02']

Общее кол-во наблюдений: 10  
 Обучающий набор: 6  
 Тестовый набор: 4

Защита:

Date_Create	Value	Group	Date_Create	Value	Group			
2016-08-22	1450000.00000	/4 = 1965000	2016-08-24	2250000.00000	/5 = 1902000			
2016-08-23	1650000.00000		2016-08-25	1960000.00000				
2016-08-24	2250000.00000		2016-08-26	1950000.00000				
2016-08-25	1960000.00000		2016-08-27	1700000.00000				
2016-08-26	1950000.00000		2016-08-28	1960000.00000		/4 = 1965000		
2016-08-27	1700000.00000	2016-08-29	1950000.00000					
2016-08-30	NaN	2016-08-31	1700000.00000	2016-08-30	1950000.00000		/4 = 1825000	
2016-08-31	NaN	2016-09-01	1960000.00000	2016-08-31	1950000.00000			/3 = 1870000
2016-09-01	NaN	2016-09-02	1700000.00000	2016-08-27	1700000.00000			
2016-09-02	NaN		2016-08-27	1700000.00000	2016-08-26	1950000.00000	/2 = 1825000	
					2016-08-27	1700000.00000		

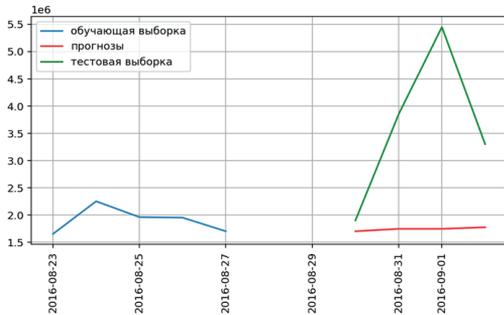
Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-22	NaN	NaN	NaN	NaN
2016-08-23	NaN	NaN	1450000.00000	1450000.00000
2016-08-24	NaN	NaN	1550000.00000	1550000.00000
2016-08-25	NaN	NaN	1783333.33333	1783333.33333
2016-08-26	1450000.00000	NaN	1827500.00000	1827500.00000
2016-08-27	1650000.00000	1450000.00000	1952500.00000	1852000.00000
2016-08-30	2250000.00000	1650000.00000	1965000.00000	1902000.00000
2016-08-31	1960000.00000	2250000.00000	1870000.00000	1965000.00000
2016-09-01	1950000.00000	1960000.00000	1825000.00000	1870000.00000
2016-09-02	1700000.00000	1950000.00000	1700000.00000	1825000.00000

RMSE=2265927.521 на 1-й итерации

Тестовая выборка



## 2-я итерация перекрестной проверки

-----  
 TRAIN: ['2016-08-22', '2016-08-31'] TEST: ['2016-09-03', '2016-09-06']

Общее кол-во наблюдений: 14

Обучающий набор: 10

Тестовый набор: 4

Защита:

Date\_Create

2016-08-22 1450000.00000

2016-08-23 1650000.00000

2016-08-24 2250000.00000

2016-08-25 1960000.00000

2016-08-26 1950000.00000

2016-08-27 1700000.00000

2016-08-28 1550000.00000

2016-08-29 2330000.00000

2016-08-30 1900000.00000

2016-08-31 3850000.00000

Гэп → 2016-09-03 NaN

2016-09-04 NaN

2016-09-05 NaN

2016-09-06 NaN

Name: Value\_abs, dtype: float64

Доб. признаки:

Date\_Create Lag\_4 Lag\_5 Moving\_mean\_4 Moving\_mean\_5

2016-08-22 NaN NaN NaN NaN

2016-08-23 NaN NaN 1450000.00000 1450000.00000

2016-08-24 NaN NaN 1550000.00000 1550000.00000

2016-08-25 NaN NaN 1783333.33333 1783333.33333

2016-08-26 1450000.00000 NaN 1827500.00000 1827500.00000

2016-08-27 1650000.00000 1450000.00000 1952500.00000 1852000.00000

2016-08-28 2250000.00000 1650000.00000 1965000.00000 1902000.00000

2016-08-29 1960000.00000 2250000.00000 1790000.00000 1882000.00000

2016-08-30 1950000.00000 1960000.00000 1882500.00000 1898000.00000

2016-08-31 1700000.00000 1950000.00000 1870000.00000 1886000.00000

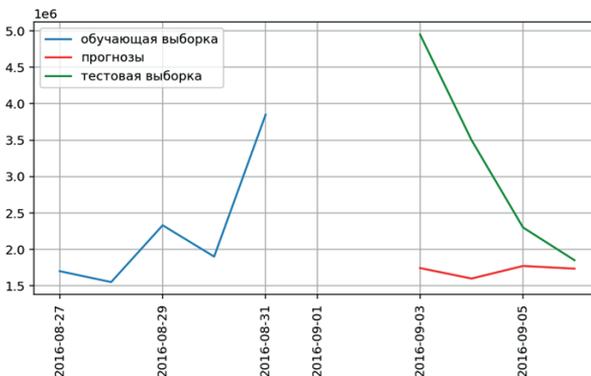
Гэп → 2016-09-03 1550000.00000 1700000.00000 2407500.00000 2266000.00000

2016-09-04 2330000.00000 1550000.00000 2693333.33333 2407500.00000

2016-09-05 1900000.00000 2330000.00000 2875000.00000 2693333.33333

2016-09-06 3850000.00000 1900000.00000 3850000.00000 2875000.00000

RMSE=1883973.065 на 2-й итерации



## 3-я итерация перекрестной проверки

```
-----
TRAIN: ['2016-08-22', '2016-09-04'] TEST: ['2016-09-07', '2016-09-10']
```

Общее кол-во наблюдений: 18

Обучающий набор: 14

Тестовый набор: 4

Защита:

Date\_Create

2016-08-22 1450000.00000

2016-08-23 1650000.00000

2016-08-24 2250000.00000

2016-08-25 1960000.00000

2016-08-26 1950000.00000

2016-08-27 1700000.00000

2016-08-28 1550000.00000

2016-08-29 2330000.00000

2016-08-30 1900000.00000

2016-08-31 3850000.00000

2016-09-01 5450000.00000

2016-09-02 3300000.00000

2016-09-03 4950000.00000

2016-09-04 3500000.00000

Гэп → 2016-09-07 NaN

2016-09-08 NaN

2016-09-09 NaN

2016-09-10 NaN

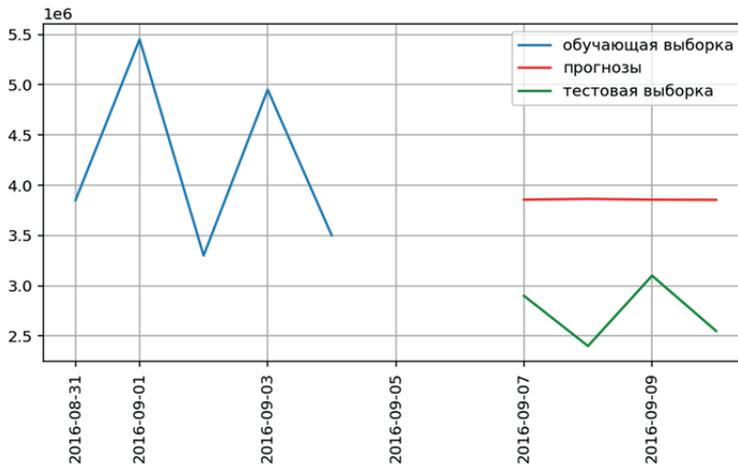
Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-22	NaN	NaN	NaN	NaN
2016-08-23	NaN	NaN	1450000.00000	1450000.00000
2016-08-24	NaN	NaN	1550000.00000	1550000.00000
2016-08-25	NaN	NaN	1783333.33333	1783333.33333
2016-08-26	1450000.00000	NaN	1827500.00000	1827500.00000
2016-08-27	1650000.00000	1450000.00000	1952500.00000	1852000.00000
2016-08-28	2250000.00000	1650000.00000	1965000.00000	1902000.00000
2016-08-29	1960000.00000	2250000.00000	1790000.00000	1882000.00000
2016-08-30	1950000.00000	1960000.00000	1882500.00000	1898000.00000
2016-08-31	1700000.00000	1950000.00000	1870000.00000	1886000.00000
2016-09-01	1550000.00000	1700000.00000	2407500.00000	2266000.00000
2016-09-02	2330000.00000	1550000.00000	3382500.00000	3016000.00000
2016-09-03	1900000.00000	2330000.00000	3625000.00000	3366000.00000
2016-09-04	3850000.00000	1900000.00000	4387500.00000	3890000.00000
2016-09-07	5450000.00000	3850000.00000	4300000.00000	4210000.00000
2016-09-08	3300000.00000	5450000.00000	3916666.66667	4300000.00000
2016-09-09	4950000.00000	3300000.00000	4225000.00000	3916666.66667
2016-09-10	3500000.00000	4950000.00000	3500000.00000	4225000.00000

Гэп → 2016-09-07 5450000.00000 3850000.00000 4300000.00000 4210000.00000

RMSE=1153976.754 на 3-й итерации



Среднее значение RMSE=1767959.113

Список признаков:

```
['Lag_4', 'Lag_5', 'Moving_mean_4', 'Moving_mean_5']
```

На тех же данных проиллюстрируем, как выполняется перекрестная проверка скользящим окном с гэпом, когда используются лаги и скользящие средние. При этом подробно посмотрим, как формируются лаги и скользящие средние в первой итерации перекрестной проверки.

```
# применяем функцию перекрестной проверки скользящим
# окном с гэпом, формируем скользящие статистики и лаги
# на каждой итерации
timeseries_cv_with_lags_and_moving_stats(
    flat_data, y_flat_data, model=ctbst_model,
    lags_range=range(4, 6),
    moving_stats_range=range(4, 6), aggfunc='mean',
    max_train_size=6, test_size=4, n_splits=3, gap=2)
```

1-я итерация перекрестной проверки

TRAIN: ['2016-08-22', '2016-08-27'] TEST: ['2016-08-30', '2016-09-02']

Общее кол-во наблюдений: 10  
 Обучающий набор: 6  
 Тестовый набор: 4

Защита:

Date_Create	Value_abs	Group
2016-08-22	1450000.00000	/4 = 1965000
2016-08-23	1650000.00000	
2016-08-24	2250000.00000	
2016-08-25	1960000.00000	
2016-08-26	1950000.00000	
2016-08-27	1700000.00000	/3 = 1870000
2016-08-28	1960000.00000	
2016-08-29	1950000.00000	/2 = 1825000
2016-08-30	NaN	
2016-08-31	NaN	/1 = 1700000
2016-09-01	NaN	
2016-09-02	NaN	

Date_Create	Value_abs	Group
2016-08-23	1650000.00000	/5 = 1902000
2016-08-24	2250000.00000	
2016-08-25	1960000.00000	
2016-08-26	1950000.00000	
2016-08-27	1700000.00000	/4 = 1965000
2016-08-28	1960000.00000	
2016-08-29	1950000.00000	
2016-08-30	NaN	/3 = 1870000
2016-08-31	NaN	
2016-09-01	NaN	
2016-09-02	NaN	/2 = 1825000
2016-09-03	NaN	

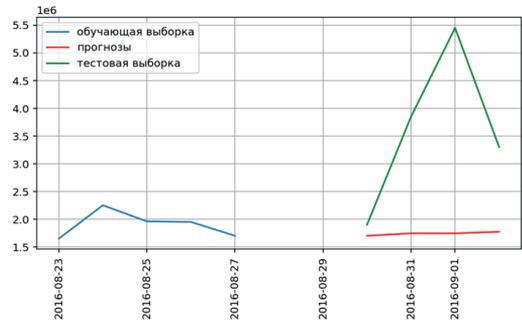
Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-22	NaN	NaN	NaN	NaN
2016-08-23	NaN	NaN	1450000.00000	1450000.00000
2016-08-24	NaN	NaN	1550000.00000	1550000.00000
2016-08-25	NaN	NaN	1783333.33333	1783333.33333
2016-08-26	1450000.00000	NaN	1827500.00000	1827500.00000
2016-08-27	1650000.00000	1450000.00000	1952500.00000	1852000.00000
2016-08-30	2250000.00000	1650000.00000	1965000.00000	1902000.00000
2016-08-31	1960000.00000	2250000.00000	1870000.00000	1965000.00000
2016-09-01	1950000.00000	1960000.00000	1825000.00000	1870000.00000
2016-09-02	1700000.00000	1950000.00000	1700000.00000	1825000.00000

RMSE=2265927.521 на 1-й итерации

Тестовая выборка



**2-я итерация перекрестной проверки**

-----  
 TRAIN: ['2016-08-26', '2016-08-31'] TEST: ['2016-09-03', '2016-09-06']

Общее кол-во наблюдений: 10

Обучающий набор: 6

Тестовый набор: 4

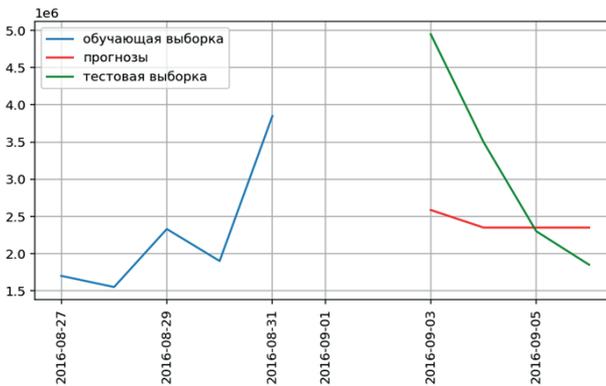
Защита:

Date\_Create  
 2016-08-26 1950000.00000  
 2016-08-27 1700000.00000  
 2016-08-28 1550000.00000  
 2016-08-29 2330000.00000  
 2016-08-30 1900000.00000  
 2016-08-31 3850000.00000  
 Гэп → 2016-09-03 NaN  
 2016-09-04 NaN  
 2016-09-05 NaN  
 2016-09-06 NaN  
 Name: Value\_abs, dtype: float64

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-26	NaN	NaN	NaN	NaN
2016-08-27	NaN	NaN	1950000.00000	1950000.00000
2016-08-28	NaN	NaN	1825000.00000	1825000.00000
2016-08-29	NaN	NaN	1733333.33333	1733333.33333
2016-08-30	1950000.00000	NaN	1882500.00000	1882500.00000
2016-08-31	1700000.00000	1950000.00000	1870000.00000	1886000.00000
Гэп → 2016-09-03	1550000.00000	1700000.00000	2407500.00000	2266000.00000
2016-09-04	2330000.00000	1550000.00000	2693333.33333	2407500.00000
2016-09-05	1900000.00000	2330000.00000	2875000.00000	2693333.33333
2016-09-06	3850000.00000	1900000.00000	3850000.00000	2875000.00000

RMSE=1338265.222 на 2-й итерации



### 3-я итерация перекрестной проверки

-----  
 TRAIN: ['2016-08-30', '2016-09-04'] TEST: ['2016-09-07', '2016-09-10']

Общее кол-во наблюдений: 10  
 Обучающий набор: 6  
 Тестовый набор: 4

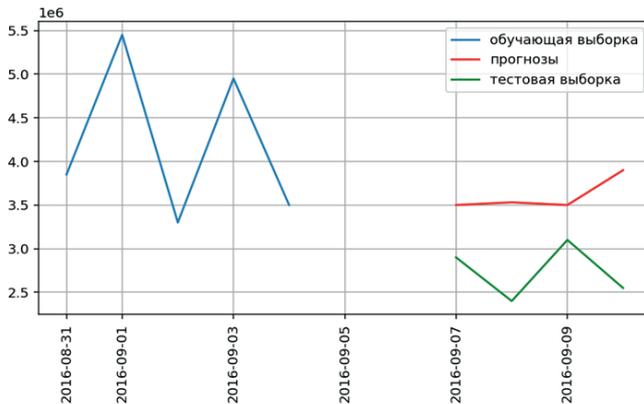
Защита:

```
Date_Create
2016-08-30 1900000.00000
2016-08-31 3850000.00000
2016-09-01 5450000.00000
2016-09-02 3300000.00000
2016-09-03 4950000.00000
2016-09-04 3500000.00000
Гэп → 2016-09-07 NaN
2016-09-08 NaN
2016-09-09 NaN
2016-09-10 NaN
Name: Value_abs, dtype: float64
```

Доб. признаки:

Date_Create	Lag_4	Lag_5	Moving_mean_4	Moving_mean_5
2016-08-30	NaN	NaN	NaN	NaN
2016-08-31	NaN	NaN	1900000.00000	1900000.00000
2016-09-01	NaN	NaN	2875000.00000	2875000.00000
2016-09-02	NaN	NaN	3733333.33333	3733333.33333
2016-09-03	1900000.00000	NaN	3625000.00000	3625000.00000
2016-09-04	3850000.00000	1900000.00000	4387500.00000	3890000.00000
Гэп → 2016-09-07	5450000.00000	3850000.00000	4300000.00000	4210000.00000
2016-09-08	3300000.00000	5450000.00000	3916666.66667	4300000.00000
2016-09-09	4950000.00000	3300000.00000	4225000.00000	3916666.66667
2016-09-10	3500000.00000	4950000.00000	3500000.00000	4225000.00000

RMSE=951258.299 на 3-й итерации



Среднее значение RMSE=1518483.681

Список признаков:

['Lag\_4', 'Lag\_5', 'Moving\_mean\_4', 'Moving\_mean\_5']

## 7.23. ПЕРЕКРЕСТНАЯ ПРОВЕРКА ДЛЯ ДАННЫХ ФОРМАТА «ОДИН ОБЪЕКТ – НЕСКОЛЬКО НАБЛЮДЕНИЙ» (ПРИСУТСТВУЕТ ОСЬ ВРЕМЕНИ)

До этого момента мы разбирали ситуации, когда у нас был временной ряд и каждому клиенту соответствовало одно наблюдение. Теперь рассмотрим случай, когда у нас по каждому клиенту есть несколько наблюдений и временной ряд.

Client	Date	Transaction	Status
Petrov	01.01.2011	12000	0
Petrov	01.01.2013	24000	0
Petrov	01.01.2012	34000	0
Ivanov	01.01.2013	10000	1
Ivanov	01.01.2012	45000	1
Ivanov	01.01.2011	15000	1
Sidorov	01.01.2012	18000	1
Sidorov	01.01.2011	69000	1
Sidorov	01.01.2013	80000	1

В данном случае нам потребуются внешний и внутренний циклы перекрестной проверки. Во внешнем цикле мы будем использовать перекрестную проверку, учитывающую группы связанных наблюдений (класс `GroupKFold`), поскольку у нас каждый клиент представлен группой наблюдений, а во внутреннем цикле мы воспользуемся перекрестной проверкой расширяющимся окном (класс `TimeSeriesSplit`), поскольку у нас по каждому клиенту есть временной ряд.

Применительно к нашему набору эта смешанная стратегия будет работать следующим образом.

Сидоров попадает в тестовую выборку, Петров и Иванов попадают в обучающую выборку. Сначала обучаем на 2011 году, смотрим качество на 2012, потом учим на 2011–2012, смотрим на 2013.

Петров попадает в тестовую выборку, Сидоров и Иванов попадают в обучающую выборку. Сначала обучаем на 2011 году, смотрим качество на 2012, потом учим на 2011–2012, смотрим на 2013.

Иванов попадает в тестовую выборку, Сидоров и Петров попадают в обучающую выборку. Сначала обучаем на 2011 году, смотрим качество на 2012, потом учим на 2011–2012, смотрим на 2013.

Давайте реализуем эту стратегию на вышеприведенном примере. Сначала загрузим и подготовим наши данные.

*# импортируем необходимые библиотеки и классы*

```
import numpy as np
import pandas as pd
from sklearn.model_selection import (GroupKFold,
                                     TimeSeriesSplit)
```

*# записываем CSV-файл в объект DataFrame*

```
toy_data = pd.read_csv('Data/toy_example.csv', sep=';')
toy_data
```

	Client	Date	Transaction	Status
0	Petrov	01.01.2011	12000	0
1	Petrov	01.01.2013	24000	0
2	Petrov	01.01.2012	34000	0
3	Ivanov	01.01.2013	10000	1
4	Ivanov	01.01.2012	45000	1
5	Ivanov	01.01.2011	15000	1
6	Sidorov	01.01.2012	18000	1
7	Sidorov	01.01.2011	69000	1
8	Sidorov	01.01.2013	80000	1

*# преобразовываем в формат даты*

```
toy_data['Date'] = pd.to_datetime(toy_data['Date'],
                                 format='%d.%m.%Y')
```

*# сортируем по дате и перезадаем индекс без вставки*

*# индекса в качестве отдельного столбца*

```
toy_data = toy_data.sort_values(by=['Date']).reset_index(drop=True)
toy_data
```

	Client	Date	Transaction	Status
0	Petrov	2011-01-01	12000	0
1	Ivanov	2011-01-01	15000	1
2	Sidorov	2011-01-01	69000	1
3	Petrov	2012-01-01	34000	0
4	Ivanov	2012-01-01	45000	1
5	Sidorov	2012-01-01	18000	1
6	Petrov	2013-01-01	24000	0
7	Ivanov	2013-01-01	10000	1
8	Sidorov	2013-01-01	80000	1

Теперь напишем класс `GroupKFoldPlusTimeSeriesSplit`, в котором реализуем внешний цикл перекрестной проверки на основе класса `GroupKFold` и внутренний цикл перекрестной проверки на основе класса `TimeSeriesSplit`.

```

# пишем класс GroupKFoldPlusTimeSeriesSplit, выполняющий
# проверку GroupKFold + TimeSeriesSplit
class GroupKFoldPlusTimeSeriesSplit():
    """
    Выполняет перекрестную проверку на основе классов
    GroupKFold и TimeSeriesSplit

    Параметры
    -----
    group_n_splits:
        Количество разбиений для GroupKFold.
    time_n_splits:
        Количество разбиений для TimeSeriesSplit.
    """

    def __init__(self, group_n_splits, time_n_splits):
        self.group_n_splits = group_n_splits
        self.time_n_splits = time_n_splits

    def get_n_splits(self, X, y, groups):
        return self.group_n_splits

    def split(self, X, y=None, groups=None):
        group_kfold = GroupKFold(n_splits=self.group_n_splits)
        for train_index, test_index in group_kfold.split(
            X, y, groups=groups):
            tscv = TimeSeriesSplit(n_splits=self.time_n_splits)
            for train_index_t, test_index_t in tscv.split(X, y):
                yield (np.intersect1d(train_index, train_index_t),
                       np.intersect1d(test_index, test_index_t))

```

Давайте воспользуемся нашим классом.

```

# создаем экземпляр класса GroupKFoldPlusTimeSeriesSplit
groupkfold_time_cv = GroupKFoldPlusTimeSeriesSplit(
    group_n_splits=3, time_n_splits=2)

# создаем массив признаков и массив меток
X_toy = toy_data.drop(columns=['Status'])
y_toy = toy_data['Status']
# задаем идентификатор меток
groups = toy_data['Client']
# взглянем на индексы наблюдений, попавших
# в обучающий и тестовый наборы
for train_index, test_index in groupkfold_time_cv.split(
    X_toy, y_toy, groups=groups):
    print("-----")
    print("TRAIN:", train_index)
    print("TEST:", test_index)
    X_train = X_toy.iloc[train_index]
    X_test = X_toy.iloc[test_index]
    print("---")
    print("X_train\n", X_train)
    print("X_test\n", X_test)

```

```

-----
TRAIN: [0 1]
TEST: [5]
---
X_train
  Client      Date      Transaction
0  Petrov 2011-01-01      12000
1  Ivanov 2011-01-01      15000
X_test
  Client      Date      Transaction
5  Sidorov 2012-01-01      18000
-----
TRAIN: [0 1 3 4]
TEST: [8]
---
X_train
  Client      Date      Transaction
0  Petrov 2011-01-01      12000
1  Ivanov 2011-01-01      15000
3  Petrov 2012-01-01      34000
4  Ivanov 2012-01-01      45000
X_test
  Client      Date      Transaction
8  Sidorov 2013-01-01      80000
-----
TRAIN: [1 2]
TEST: [3]
---
X_train
  Client      Date      Transaction
1  Ivanov 2011-01-01      15000
2  Sidorov 2011-01-01      69000
X_test
  Client      Date      Transaction
3  Petrov 2012-01-01      34000
-----
TRAIN: [1 2 4 5]
TEST: [6]
---
X_train
  Client      Date      Transaction
1  Ivanov 2011-01-01      15000
2  Sidorov 2011-01-01      69000
4  Ivanov 2012-01-01      45000
5  Sidorov 2012-01-01      18000
X_test
  Client      Date      Transaction
6  Petrov 2013-01-01      24000
-----
TRAIN: [0 2]
TEST: [4]
---
X_train
  Client      Date      Transaction
0  Petrov 2011-01-01      12000
2  Sidorov 2011-01-01      69000
X_test
  Client      Date      Transaction
4  Ivanov 2012-01-01      45000
-----
TRAIN: [0 2 3 5]
TEST: [7]
---
X_train
  Client      Date      Transaction
0  Petrov 2011-01-01      12000
2  Sidorov 2011-01-01      69000
3  Petrov 2012-01-01      34000
5  Sidorov 2012-01-01      18000
X_test
  Client      Date      Transaction
7  Ivanov 2013-01-01      10000

```

Сидоров попадает в тестовую выборку, Петров и Иванов попадают в обучающую выборку. Сначала обучаем на 2011 году, смотрим качество на 2012, потом учим на 2011–2012, смотрим на 2013.

Петров попадает в тестовую выборку, Сидоров и Иванов попадают в обучающую выборку. Сначала обучаем на 2011 году, смотрим качество на 2012, потом учим на 2011–2012, смотрим на 2013.

Иванов попадает в тестовую выборку, Сидоров и Петров попадают в обучающую выборку. Сначала обучаем на 2011 году, смотрим качество на 2012, потом учим на 2011–2012, смотрим на 2013.

## 7.24. Многоклассовая классификация: ПОДХОДЫ «ОДИН ПРОТИВ ВСЕХ», «ОДИН ПРОТИВ ОДНОГО» И «КОДЫ, ИСПРАВЛЯЮЩИЕ ОШИБКИ»

До этого момента мы решали только задачи регрессии и бинарной классификации. Однако нередко бывают задачи, когда наша зависимая переменная имеет больше двух классов. Предположим, что ваша задача состоит в том, чтобы классифицировать фрукты на три различных, взаимоисключающих класса: яблоки, груши и апельсины. Когда у нас зависимая переменная имеет больше двух классов, а наблюдение принадлежит строго одному классу (наблюдение не может быть одновременно грушей и яблоком), мы имеем дело с многоклассовой классификацией.

Давайте импортируем необходимые библиотеки, классы, функции и приведем пример такой трехклассовой классификации в Python.

```
# импортируем библиотеки, классы и функции
import pandas as pd
import numpy as np
from sklearn.multiclass import (OneVsRestClassifier,
                                OneVsOneClassifier,
                                OutputCodeClassifier)
from sklearn.preprocessing import (StandardScaler,
                                   LabelEncoder)
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.base import (clone,
                           MetaEstimatorMixin,
                           ClassifierMixin,
                           BaseEstimator)
from sklearn.model_selection import train_test_split
from sklearn.utils.metaestimators import _safe_split
from sklearn.metrics import (roc_auc_score,
                              pairwise_distances)
from sklearn.utils.fixes import delayed
from joblib import Parallel
import warnings

# отключаем экспоненциальное представление
np.set_printoptions(precision=None, suppress=True)

# пример трех классов
y = np.array(['apple', 'pear', 'apple',
              'orange', 'pear', 'apple'])
```

Для преобразования строковых меток классов в целочисленные можно воспользоваться классом `LabelEncoder`.

```
# строковые метки преобразовываем в целочисленные
le = LabelEncoder()
y_trn = le.fit_transform(y_trn)
y_trn

array([0, 2, 0, 1, 2, 0])
```

Большинство классификаторов scikit-learn поддерживают многоклассовую классификацию естественным образом:

- `naive_bayes.BernoulliNB`;
- `tree.DecisionTreeClassifier`;
- `tree.ExtraTreeClassifier`;
- `ensemble.ExtraTreesClassifier`;
- `naive_bayes.GaussianNB`;
- `neighbors.KNeighborsClassifier`;
- `semi_supervised.LabelPropagation`;
- `semi_supervised.LabelSpreading`;
- `discriminant_analysis.LinearDiscriminantAnalysis`;
- `svm.LinearSVC` (задаем `multi_class='crammer_singer'`);
- `linear_model.LogisticRegression` (задаем `multi_class='multinomial'`);
- `linear_model.LogisticRegressionCV` (задаем `multi_class='multinomial'`);
- `neural_network.MLPClassifier`;
- `neighbors.NearestCentroid`;
- `discriminant_analysis.QuadraticDiscriminantAnalysis`;
- `neighbors.RadiusNeighborsClassifier`;
- `ensemble.RandomForestClassifier`;
- `linear_model.RidgeClassifier`;
- `linear_model.RidgeClassifierCV`.

Ряд классификаторов scikit-learn поддерживают многоклассовую классификацию в рамках подхода «один против остальных»:

- `ensemble.GradientBoostingClassifier`;
- `gaussian_process.GaussianProcessClassifier` (задаем `multi_class='one_vs_rest'`);
- `svm.LinearSVC` (задаем `multi_class='ovr'`);
- `linear_model.LogisticRegression` (задаем `multi_class='ovr'`);
- `linear_model.LogisticRegressionCV` (задаем `multi_class='ovr'`);
- `linear_model.SGDClassifier`;
- `linear_model.Perceptron`;
- `linear_model.PassiveAggressiveClassifier`.

Несколько классификаторов scikit-learn поддерживают многоклассовую классификацию в рамках подхода «один против одного»:

- `svm.NuSVC`;
- `svm.SVC`;
- `gaussian_process.GaussianProcessClassifier` (задаем `multi_class='one_vs_one'`).

### 7.24.1. Подход «один против остальных» или «один против всех» («one versus rest», «one versus all»)

Подход «один против остальных» («one versus rest» или `ovr`), его еще называют «один против всех» («one versus all»), является общераспространенным подходом, позволяющим распространить алгоритм бинарной классификации на случай многоклассовой классификации.

У нас `n_classes` классов, решаем `n_classes` задач бинарной классификации, в  $i$ -й задаче отделяем  $i$ -й класс от остальных (объекты  $i$ -го класса получают метку 1, а остальные – метку 0). Новое наблюдение подаем `n_classes` обученным классификаторам, относим к тому классу, в котором соответствующий классификатор максимально уверен.

Разберем на нашем примере с фруктами. У нас – три класса 'apple', 'pear' и 'orange'.

Обучаем три классификатора:

- бинарный классификатор 1: 'apple' против ['pear', 'orange'], наблюдения с классом 'apple' считаем положительным классом, а наблюдения с классами 'pear', 'orange' считаем отрицательным классом;
- бинарный классификатор 2: 'pear' против ['apple', 'orange'], наблюдения с классом 'pear' считаем положительным классом, а наблюдения с классами 'apple', 'orange' считаем отрицательным классом;
- бинарный классификатор 3: 'orange' против ['pear', 'apple'], наблюдения с классом 'orange' считаем положительным классом, а наблюдения с классами 'pear', 'apple' считаем отрицательным классом.

Взяли новое наблюдение и подали трем классификаторам, получили оценки уверенности либо в виде вероятностей положительного класса, либо в виде решающей функции (если у класса нет метода `.predict_proba()`).

#### Значения вероятности

0 apple	1 pear	2 orange
0,12	0,6	0,28

#### Значения решающей функции

0 apple	1 pear	2 orange
1,15	5,1	2,3

Просто берем индекс 1 – индекс класса с максимальным значением вероятности положительного класса или решающей функции.

На примере задачи Otto Group Product Classification Challenge с Kaggle <https://www.kaggle.com/c/otto-group-product-classification-challenge/overview> проиллюстрируем, как можно выполнить многоклассовую классификацию естественным образом и с помощью подхода «один против остальных». Исторический набор данных состоит из 95 переменных (94 анонимизированных признака и зависимая переменная *target*). Зависимая переменная включает 9 категорий продукта. Задача заключается в том, чтобы предсказать вероятность выбора каждой из 9 категорий продукта.

Давайте загрузим и подготовим данные.

```

# загружаем наборы
otto = pd.read_csv('Data/ottogroup_train.csv')

# удаляем id из набора
otto.drop('id', axis=1, inplace=True)

# формируем массив меток и массив признаков,
# преобразуем в массивы NumPy
X_otto = otto.drop('target', axis=1).values
y_otto = otto['target'].values

# смотрим метки классов
np.unique(y_otto)

array(['Class_1', 'Class_2', 'Class_3', 'Class_4', 'Class_5', 'Class_6',
       'Class_7', 'Class_8', 'Class_9'], dtype=object)

# создание экземпляра класса LabelEncoder
le = LabelEncoder()
# строковые метки преобразовываем в целочисленные
y_otto = le.fit_transform(y_otto)
np.unique(y_otto)

array([0, 1, 2, 3, 4, 5, 6, 7, 8])

# разбиваем набор на обучающую и тестовую выборки
X_otto_train, X_otto_test, y_otto_train, y_otto_test = train_test_split(
    X_otto, y_otto, random_state=42)

```

Сейчас на примере задачи Otto Group Product Classification Challenge мы выполним многоклассовую классификацию естественным образом с помощью логистической регрессии. Нам нужно создать модель – экземпляр класса `LogisticRegression`, задав `multi_class='multinomial'` (используем логистическую функцию потерь для многоклассового случая) и обучить.

```

# создаем конвейер – экземпляр класса Pipeline
inh_lr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('logreg', LogisticRegression(solver='newton-cg',
                                  multi_class='multinomial'))
])

# применяем LogisticRegression для
# многоклассовой классификации
# естественным образом
inh_lr_pipe.fit(X_otto_train, y_otto_train);

```

Теперь выведем вероятности классов и значения решающей функции для тестовой выборки (максимальные значения в каждой строке выделены желтым фоном). Вывод модифицирован – добавлены индексы классов.

```

# вычисляем вероятности классов зависимой
# переменной для тестовой выборки
inh_lr_proba = np.round(
    inh_lr_pipe.predict_proba(X_otto_test), 3)
inh_lr_proba[:5]

      0      1      2      3      4      5      6      7      8
array([[0.025, 0.514, 0.135, 0.06 , 0.09 , 0.03 , 0.084, 0.044, 0.019],
       [0.013, 0.059, 0.067, 0.002, 0.   , 0.098, 0.748, 0.001, 0.011],
       [0.004, 0.007, 0.002, 0.   , 0.   , 0.934, 0.037, 0.009, 0.006],
       [0.003, 0.001, 0.   , 0.748, 0.   , 0.219, 0.015, 0.   , 0.013],
       [0.   , 0.   , 0.   , 0.   , 0.   , 1.   , 0.   , 0.   , 0.   ]])

```

```

# вычисляем значения решающей функции
# для тестовой выборки
inh_lr_dec_func = np.round(
    inh_lr_pipe.decision_function(X_otto_test), 3)
inh_lr_dec_func[:5]

```

```

      0      1      2      3      4      5      6      7      8
array([[ -0.942,  2.076,  0.74 , -0.076,  0.337, -0.776,  0.268, -0.387, -1.24 ],
       [ 1.088,  2.591,  2.717, -0.648, -13.425,  3.091,  5.123, -1.447,  0.91 ],
       [-0.326,  0.404, -0.835, -2.458, -4.933,  5.258,  2.042,  0.653,  0.195],
       [ 0.403, -1.058, -2.34 ,  5.903, -9.782,  4.674,  2.017, -1.661,  1.845],
       [ 0.204, -4.149,  0.794, -7.618, -15.39 , 17.632,  1.869,  7.978, -1.32 ]])

```

Теперь выведем прогнозы для тестовой выборки.

```

# вычисляем прогнозы для тестовой выборки
inh_lr_pred = inh_lr_pipe.predict(X_otto_test)
inh_lr_pred[:5]

```

```
array([1, 6, 5, 3, 5])
```

Теперь мы выполним многоклассовую классификацию по схеме «один против всех» с помощью логистической регрессии. Нам нужно создать модель – экземпляр класса `LogisticRegression`, задав `multi_class='ovr'`, и обучить.

```

# создаем конвейер – экземпляр класса Pipeline
ovr_lr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('logreg', LogisticRegression(solver='newton-cg',
                                 multi_class='ovr'))
])

```

```

# применяем LogisticRegression для
# многоклассовой классификации
# по схеме one-vs-rest
ovr_lr_pipe.fit(X_otto_train, y_otto_train);

```

Снова выведем вероятности классов и значения решающей функции для тестовой выборки (максимальные значения в каждой строке выделены желтым фоном). Вывод модифицирован – добавлены индексы классов.

```
# вычисляем вероятности классов зависимой
# переменной для тестовой выборки
ovr_lr_proba = np.round(
    ovr_lr_pipe.predict_proba(X_otto_test), 3)
ovr_lr_proba[:5]
```

	0	1	2	3	4	5	6	7	8
array([[	0.066,	0.42	0.137,	0.039,	0.111,	0.057,	0.083,	0.066,	0.021],
	[0.019,	0.103,	0.13,	0.008,	0.	0.045,	0.655,	0.001,	0.038],
	[0.009,	0.061,	0.02,	0.002,	0.	0.844,	0.046,	0.009,	0.008],
	[0.008,	0.008,	0.006,	0.447,	0.	0.456,	0.033,	0.	0.04 ],
	[0.	0.	0.008,	0.	0.	0.99,	0.	0.002,	0. ]])

```
# вычисляем значения решающей функции
# для тестовой выборки
ovr_lr_dec_func = np.round(
    ovr_lr_pipe.decision_function(X_otto_test), 3)
ovr_lr_dec_func[:5]
```

	0	1	2	3	4	5	6	7	8
array([[	-3.023,	-0.858,	-2.231,	-3.555,	-2.466,	-3.165,	-2.768,	-3.019,	-4.201],
	[ -3.913,	-2.134,	-1.868,	-4.742,	-17.881,	-3.017,	0.717,	-7.213,	-3.2 ],
	[ -4.639,	-2.605,	-3.758,	-5.887,	-8.113,	2.842,	-2.907,	-4.638,	-4.732],
	[ -4.709,	-4.74,	-4.968,	-0.086,	-14.566,	-0.049,	-3.299,	-8.045,	-3.097],
	[-10.845,	-9.92,	-4.786,	-14.167,	-22.513,	10.257,	-9.421,	-6.303,	-13.133]])

Теперь выведем прогнозы для тестовой выборки.

```
# вычисляем прогнозы для тестовой выборки
ovr_lr_pred = ovr_lr_pipe.predict(X_otto_test)
ovr_lr_pred[:5]
```

```
array([1, 6, 5, 5, 5])
```

Сейчас мы выполним многоклассовую классификацию по схеме «один против всех» с помощью логистической регрессии, но только теперь не будем задавать `multi_class='ovr'`, а воспользуемся специальным классом `OneVsRestClassifier`. Класс `OneVsRestClassifier` позволяет использовать любой классификатор для решения задач многоклассовой классификации в рамках подхода «один против всех». Класс `LogisticRegression` сам поддерживает подход «один против всех», однако есть классы, для которых этот подход не реализован, и можно воспользоваться классом `OneVsRestClassifier`. Под капотом все остается прежним: у нас происходит обучение одного классификатора для каждого класса, при котором мы считаем наблюдения с нужным классом положительными наблюдениями, а все остальные наблюдения – отрицательными.

```
# создаем конвейер - экземпляр класса Pipeline
lr_pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('logreg', LogisticRegression(solver='newton-cg'))
])
```

```
# создаем экземпляр класса OneVsRestClassifier
lr_ovr_classifier = OneVsRestClassifier(lr_pipe)
# применяем LogisticRegression для многоклассовой
# классификации по схеме one-vs-rest через
# класс OneVsRestClassifier
lr_ovr_classifier.fit(X_otto_train, y_otto_train);
```

Традиционно выведем вероятности классов и значения решающей функции для тестовой выборки (максимальные значения в каждой строке выделены желтым фоном). Вывод модифицирован – добавлены индексы классов.

```
# вычисляем вероятности классов зависимой
# переменной для тестовой выборки
lr_ovr_classifier_proba = np.round(
    lr_ovr_classifier.predict_proba(X_otto_test), 3)
lr_ovr_classifier_proba[:5]
```

	0	1	2	3	4	5	6	7	8
array([[	0.066,	0.42,	0.137,	0.039,	0.111,	0.057,	0.083,	0.066,	0.021],
	[0.019,	0.103,	0.13,	0.008,	0.	0.045,	0.655,	0.001,	0.038],
	[0.009,	0.061,	0.02,	0.002,	0.	0.844,	0.046,	0.009,	0.008],
	[0.008,	0.008,	0.006,	0.447,	0.	0.456,	0.033,	0.	0.04 ],
	[0.	0.	0.008,	0.	0.	0.99,	0.	0.002,	0. ]])

```
# вычисляем значения решающей функции
# для тестовой выборки
lr_ovr_classifier_dec_func = np.round(
    lr_ovr_classifier.decision_function(X_otto_test), 3)
lr_ovr_classifier_dec_func[:5]
```

	0	1	2	3	4	5	6	7	8
array([[	-3.023,	-0.858,	-2.231,	-3.555,	-2.466,	-3.165,	-2.768,	-3.019,	-4.201],
	[-3.913,	-2.134,	-1.868,	-4.742,	-17.881,	-3.017,	0.717,	-7.213,	-3.2 ],
	[-4.639,	-2.605,	-3.758,	-5.887,	-8.113,	2.842,	-2.907,	-4.638,	-4.732],
	[-4.709,	-4.74,	-4.968,	-0.086,	-14.566,	-0.049,	-3.299,	-8.045,	-3.097],
	[-10.845,	-9.92,	-4.786,	-14.167,	-22.513,	10.257,	-9.421,	-6.303,	-13.133]])

Теперь выведем прогнозы для тестовой выборки.

```
# вычисляем прогнозы для тестовой выборки
lr_ovr_classifier_pred = lr_ovr_classifier.predict(X_otto_test)
lr_ovr_classifier_pred[:5]
```

```
array([1, 6, 5, 5, 5])
```

Мы получили вероятности, значения решающей функции, прогнозы, аналогичные тем, которые были вычислены при использовании настройки `multi_class='ovr'` для класса `LogisticRegression`.

## 7.24.2. Подход «один против одного» («one versus one»)

В рамках подхода «один против одного» («one versus one», или ovo) мы решаем  $n\_classes*(n\_classes-1)/2$  задач, в каждой отделяем  $i$ -й класс от  $j$ -го.

Разберем на нашем примере с фруктами. У нас – три класса 'apple', 'pear' и 'orange'.

Обучаем три классификатора:

- бинарный классификатор 1: 'apple' против 'orange';
- бинарный классификатор 2: 'apple' против 'pear';
- бинарный классификатор 3: 'orange' против 'pear'.

Оценку принадлежности к  $i$ -му классу определяют как сумму степеней уверенности в принадлежности к  $i$ -му классу всех классификаторов, которые отделяют  $i$ -й класс от какого-то другого (их ровно  $n\_classes-1$ ). Относим новое наблюдение к классу с максимальной оценкой.

Поскольку требуется обучить  $n\_classes*(n\_classes-1)/2$  классификаторов, этот подход обычно является более медленным, чем «один против остальных», из-за его сложности  $O(n\_classes^2)$ . Однако этот метод может быть выгоден для ядерных алгоритмов, которые плохо масштабируются по мере роста количества наблюдений. Это связано с тем, что каждая отдельная задача обучения включает в себя только небольшое подмножество данных, тогда как в случае «один против остальных» полный набор данных используется  $n\_classes$  раз.

В библиотеке `scikit-learn` есть класс `OneVsOneClassifier`, который позволяет использовать любой классификатор для решения задач многоклассовой классификации в рамках подхода «один против один». Обратите внимание, что в отличие от класса `OneVsRestClassifier` у класса `OneVsOneClassifier` нет метода `.predict_proba()`.

На примере задачи `Otto Group Product Classification Challenge` мы выполним многоклассовую классификацию по схеме «один против всех» с помощью логистической регрессии, воспользовавшись специальным классом `OneVsOneClassifier`. Обратите внимание: с помощью параметра `multi_class` нельзя выбрать схему «один против одного».

```
# создаем экземпляр класса OneVsOneClassifier
lr_ovo_classifier = OneVsOneClassifier(lr_pipe)
# применяем LogisticRegression для многоклассовой
# классификации по схеме one-vs-one через
# класс OneVsOneClassifier
lr_ovo_classifier.fit(X_otto_train, y_otto_train);
```

Выведем значения решающей функции для тестовой выборки (максимальные значения в каждой строке выделены желтым фоном). Вероятности мы вывести не сможем, поскольку метод `.predict_proba()` у класса `OneVsOneClassifier` отсутствует. Вывод модифицирован, добавлены индексы классов.

```
# вычисляем значения решающей функции
# для тестовой выборки
lr_ovo_classifier_dec_func = np.round(
    lr_ovo_classifier.decision_function(X_otto_test), 3)
lr_ovo_classifier_dec_func[:5]
```

```

      0      1      2      3      4      5      6      7      8
array([[ 0.724,  8.312,  7.295,  5.299,  4.233,  0.695,  6.283,  2.717,  0.684],
       [ 4.279,  5.325,  6.322,  0.704, -0.33 ,  7.316,  8.325,  1.687,  2.734],
       [ 0.688,  6.318,  5.313,  1.693, -0.328,  8.326,  7.321,  3.777,  2.701],
       [ 3.289,  4.052,  2.698,  7.328, -0.33 ,  8.326,  3.774,  0.68 ,  6.322],
       [ 1.672,  5.324,  6.33 ,  0.673,  1.669,  8.331,  4.321,  6.32 ,  1.673]])

```

Теперь выведем прогнозы для тестовой выборки.

```
# вычисляем прогнозы для тестовой выборки
```

```
lr_ovo_classifier_pred = lr_ovo_classifier.predict(X_otto_test)
lr_ovo_classifier_pred[:5]
```

```
array([1, 6, 5, 5, 5])
```

Давайте подробнее разберем, что происходит под капотом подхода «один против одного», на примере игрушечных данных.

Давайте создадим игрушечные данные – обучающий массив признаков, обучающий массив меток и тестовый массив признаков.

```
# создаем обучающий массив признаков
```

```
X_trn = np.array([[4.2, 1.5],
                  [1.4, 2.1],
                  [3.1, 0.5],
                  [1.3, 2.2],
                  [6.9, 4.5],
                  [7.9, 7.1]])
```

```
# создаем обучающий массив меток
```

```
y_trn = np.array([0, 2, 0, 1, 2, 0])
```

```
# создаем тестовый массив признаков
```

```
X_tst = np.array([[2.8, 3.5],
                  [1.1, 1.8],
                  [8.9, 8.4]])
```

Итак, берем обучающий массив признаков, состоящий из двух столбцов, и обучающий массив меток.

ind	X_trn		y_trn
0	4.2	1.5	0
1	1.4	2.1	2
2	3.1	0.5	0
3	1.3	2.2	1
4	6.9	4.5	2
5	7.9	7.1	0

Нам нужно выполнить три сравнения, будут три модели.

ind	X_trn		y_trn	0 vs 1		0 vs 2		1 vs 2	
0	4.2	1.5	0	0	0	0	0	0	0
1	1.4	2.1	2	2	2	2	2	2	2
2	3.1	0.5	0	0	0	0	0	0	0
3	1.3	2.2	1	1	1	1	1	1	1
4	6.9	4.5	2	2	2	2	2	2	2
5	7.9	7.1	0	0	0	0	0	0	0

Метки классов для всех трех моделей переведем в бинарный формат (информацию о сравниваемых классах сохраним, выделив серым фоном).

ind	X_trn		y_trn	0 vs 1 0 vs 1		0 vs 2 0 vs 1		1 vs 2 0 vs 1	
0	4.2	1.5	0	0	0	0			
1	1.4	2.1	2		1	1	1	1	1
2	3.1	0.5	0	0	0	0			
3	1.3	2.2	1	1			0	0	0
4	6.9	4.5	2		1	1	1	1	1
5	7.9	7.1	0	0	0	0			

Обучаем три модели для трех наборов.

ind	X_trn		0 vs 1 0 vs 1
0	4.2	1.5	0
2	3.1	0.5	0
3	1.3	2.2	1
5	7.9	7.1	0

ind	X_trn		0 vs 2 0 vs 1
0	4.2	1.5	0
1	1.4	2.1	1
2	3.1	0.5	0
4	6.9	4.5	1
5	7.9	7.1	0

ind	X_trn		1 vs 2 0 vs 1
1	1.4	2.1	1
3	1.3	2.2	0
4	6.9	4.5	1

Получаем прогнозы с помощью трех обученных моделей.

ind	X_trn	
0	4.2	1.5
1	1.4	2.1
2	3.1	0.5
3	1.3	2.2
4	6.9	4.5
5	7.9	7.1

0 vs 1 0 vs 1	
0	0
1	1
2	0
3	1
4	0
5	0

0 vs 2 0 vs 1	
0	0
1	1
2	0
3	1
4	0
5	0

1 vs 2 0 vs 1	
0	1
1	1
2	1
3	1
4	1
5	1

Формируем матрицу прогнозов.

predictions

0 vs 1 0 vs 1	0 vs 2 0 vs 1	1 vs 2 0 vs 1
0	0	1
1	1	1
0	0	1
1	1	1
0	0	1
0	0	1

На основе матрицы прогнозов формируем матрицу голосов размером  $n_{\text{observations}} \times n_{\text{classes}}$ , инициализованную нулями.

votes

класс 0	класс 1	класс 2
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

В ней на основе матрицы прогнозов подсчитываем голоса, поданные за соответствующий класс.

**votes**

класс 0	класс 1	класс 2
2	0	1
0	1	2
2	0	1
0	1	2
2	0	1
2	0	1

Объясним, как были получены голоса для первого наблюдения – первой строки. Первое значение строки – количество голосов для класса 0. Нас интересуют сравнения, в которых участвует класс 0, – 0 vs 1 и 0 vs 2. В обоих сравнениях модель проголосовала за класс 0. Класс 0 получает 2 голоса.

**predictions**

0 vs 1 0 vs 1	0 vs 2 0 vs 1
0	0

Второе значение строки – количество голосов для класса 1. Нас интересуют сравнения, в которых участвует класс 1, – 0 vs 1 и 1 vs 2. В первом сравнении модель проголосовала за класс 0, а во втором сравнении модель проголосовала за класс 2. Класс 1 получает 0 голосов.

**predictions**

0 vs 1 0 vs 1	1 vs 2 0 vs 1
0	1

Третье значение строки – количество голосов для класса 2. Нас интересуют сравнения, в которых участвует класс 2, – 0 vs 2 и 1 vs 2. В первом сравнении модель проголосовала за класс 0, а во втором сравнении модель проголосовала за класс 2. Класс 2 получает 1 голос.

**predictions**

0 vs 2 0 vs 1	1 vs 2 0 vs 1
0	1

Нетрудно догадаться, что мы можем получить ситуацию, когда несколько классов получат одинаковое количество голосов. Поэтому мы будем еще учитывать оценки уверенности.

Получаем для обучающего набора оценки уверенности – значения решающей функции (если реализована решающая функция) или вероятности положительного класса (если не реализована решающая функция) с помощью трех обученных моделей и формируем матрицу уверенностей.

**confidences**

<b>0 vs 1 0 vs 1</b>	<b>0 vs 2 0 vs 1</b>	<b>1 vs 2 0 vs 1</b>
-2.04743668	-0.75518106	1.23741829
0.54659607	0.47450729	0.14616936
-1.38801103	-0.66553044	0.68037941
0.66210856	0.54344122	0.11535615
-3.50727061	-0.80156436	2.66134319
-3.61460391	-0.34469341	3.34334993

На основе матрицы уверенностей формируем матрицу сумм уверенностей размером  $n_{\text{observations}} \times n_{\text{classes}}$ , инициализованную нулями.

**sum\_of\_confidences**

<b>класс 0</b>	<b>класс 1</b>	<b>класс 2</b>
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

В ней мы на основе матрицы уверенностей подсчитываем суммарные уверенности для соответствующего класса.

**sum\_of\_confidences**

<b>класс 0</b>	<b>класс 1</b>	<b>класс 2</b>
2.80261774	-3.28485496	0.48223722
-1.02110336	0.40042671	0.62067665
2.05354147	-2.06839044	0.01484897
-1.20554979	0.54675242	0.65879737
4.30883496	-6.1686138	1.85977883
3.95929733	-6.95795384	2.99865652

Объясним, как были получены суммы уверенностей.

Сравниваем класс 0 с классом 1.

Из значений столбца с индексом 0 матрицы сумм уверенностей вычитаем значения столбца с индексом 0 матрицы уверенностей. К значениям столбца с индексом 1 матрицы сумм уверенностей прибавляем значения столбца с индексом 0 (далее для краткости – столбца 0, индексация с 0) матрицы уверенностей.

```
sum_of_confidences[:, 0] - confidences[:, 0]
sum_of_confidences[:, 1] + confidences[:, 0]
```

```
sum_of_confidences:                                confidences:
[[ 0.  0.  0.]                                     [[ -2.04743668 -0.75518106  1.23741829]
 [ 0.  0.  0.]                                     [  0.54659607  0.47450729  0.14616936]
 [ 0.  0.  0.]                                     [-1.38801103 -0.66553044  0.68037941]
 [ 0.  0.  0.]                                     [  0.66210856  0.54344122  0.11535615]
 [ 0.  0.  0.]                                     [-3.50727061 -0.80156436  2.66134319]
 [ 0.  0.  0.]                                     [-3.61460391 -0.34469341  3.34334993]]
```

Получаем обновленную матрицу сумм уверенностей.

```
sum_of_confidences:
[[ 2.04743668 -2.04743668  0.          ]
 [-0.54659607  0.54659607  0.          ]
 [ 1.38801103 -1.38801103  0.          ]
 [-0.66210856  0.66210856  0.          ]
 [ 3.50727061 -3.50727061  0.          ]
 [ 3.61460391 -3.61460391  0.          ]]
```

Сравниваем класс 0 с классом 2.

Из значений столбца 0 матрицы сумм уверенностей вычитаем значения столбца 1 матрицы уверенностей. К значениям столбца 2 матрицы сумм уверенностей прибавляем значения столбца 1 матрицы уверенностей.

```
sum_of_confidences[:, 0] - confidences[:, 1]
sum_of_confidences[:, 2] + confidences[:, 1]
```

```
sum_of_confidences:                                confidences:
[[ 2.04743668 -2.04743668  0.          ]          [[ -2.04743668 -0.75518106  1.23741829]
 [-0.54659607  0.54659607  0.          ]          [  0.54659607  0.47450729  0.14616936]
 [ 1.38801103 -1.38801103  0.          ]          [-1.38801103 -0.66553044  0.68037941]
 [-0.66210856  0.66210856  0.          ]          [  0.66210856  0.54344122  0.11535615]
 [ 3.50727061 -3.50727061  0.          ]          [-3.50727061 -0.80156436  2.66134319]
 [ 3.61460391 -3.61460391  0.          ]          [-3.61460391 -0.34469341  3.34334993]]
```

Получаем обновленную матрицу сумм уверенностей.

```
sum_of_confidences:
[[ 2.80261774 -2.04743668 -0.75518106]
 [-1.02110336  0.54659607  0.47450729]
 [ 2.05354147 -1.38801103 -0.66553044]
 [-1.20554979  0.66210856  0.54344122]
 [ 4.30883496 -3.50727061 -0.80156436]
 [ 3.95929733 -3.61460391 -0.34469341]]
```

Сравниваем класс 1 с классом 2.

Из значений столбца 1 матрицы сумм уверенностей вычитаем значения столбца 2 матрицы уверенностей. К значениям столбца 2 матрицы сумм уверенностей прибавляем значения столбца 2 матрицы уверенностей.

```
sum_of_confidences[:, 1] - confidences[:, 2]
sum_of_confidences[:, 2] + confidences[:, 2]
```

```
sum_of_confidences:                                confidences:
[[ 2.80261774 -2.04743668 -0.75518106]           [[-2.04743668 -0.75518106  1.23741829]
 [-1.02110336  0.54659607  0.47450729]           [ 0.54659607  0.47450729  0.14616936]
 [ 2.05354147 -1.38801103 -0.66553044]           [-1.38801103 -0.66553044  0.68037941]
 [-1.20554979  0.66210856  0.54344122]           [ 0.66210856  0.54344122  0.11535615]
 [ 4.30883496 -3.50727061 -0.80156436]           [-3.50727061 -0.80156436  2.66134319]
 [ 3.95929733 -3.61460391 -0.34469341]]           [-3.61460391 -0.34469341  3.34334993]]
```

Получаем итоговую матрицу сумм уверенностей.

```
sum_of_confidences:
[[ 2.80261774 -3.28485496  0.48223722]
 [-1.02110336  0.40042671  0.62067665]
 [ 2.05354147 -2.06839044  0.01484897]
 [-1.20554979  0.54675242  0.65879737]
 [ 4.30883496 -6.1686138  1.85977883]
 [ 3.95929733 -6.95795384  2.99865652]]
```

Матрицу сумм уверенностей подвергаем преобразованию  $x/(3*(|x| + 1))$ , чтобы положить их в диапазон  $(-1/3, 1/3)$ , и получаем матрицу преобразованных сумм уверенностей.

```
transformed_confidences = sum_of_confidences /
(3 * (np.abs(sum_of_confidences) + 1))
```

**transformed\_confidences**

класс 0	класс 1	класс 2
0.24567442	-0.25553996	0.10844805
-0.16840692	0.09531064	0.12765792
0.22417047	-0.22469874	0.00487723
-0.18219944	0.11782804	0.13238454
0.27054492	-0.28683434	0.21677420
0.26611951	-0.29144652	0.24997200

Складываем матрицу преобразованных сумм уверенностей и матрицу голо-сов, получаем матрицу итоговых сумм уверенностей.

```
final_confidences = votes + transformed_confidences
```

votes:

```
[[2. 0. 1.]
 [0. 1. 2.]
 [2. 0. 1.]
 [0. 1. 2.]
 [2. 0. 1.]
 [2. 0. 1.]]
```

transformed\_confidences

```
[[ 0.24567442 -0.25553996  0.10844805]
 [-0.16840692  0.09531064  0.12765792]
 [ 0.22417047 -0.22469874  0.00487723]
 [-0.18219944  0.11782804  0.13238454]
 [ 0.27054492 -0.28683434  0.21677420]
 [ 0.26611951 -0.29144652  0.24997200]]
```

final\_confidences

класс 0	класс 1	класс 2
2.24567442	-0.25553996	1.10844805
-0.16840692	1.09531064	2.12765792
2.22417047	-0.22469874	1.00487723
-0.18219944	1.11782804	2.13238454
2.27054492	-0.28683434	1.21677420
2.26611951	-0.29144652	1.24997200

Теперь в качестве прогноза по каждой строке берем индекс с максимальным значением итоговой суммы уверенностей.

final\_confidences.argmax(axis=1)

класс 0	класс 1	класс 2	predictions
2.24567442	-0.25553996	1.10844805	0
-0.16840692	1.09531064	2.12765792	2
2.22417047	-0.22469874	1.00487723	0
-0.18219944	1.11782804	2.13238454	2
2.27054492	-0.28683434	1.21677420	0
2.26611951	-0.29144652	1.24997200	0

Давайте реализуем собственный класс, выполняющий многоклассовую классификацию согласно подходу «один против одного». Сначала нам потребуется написать вспомогательный класс и ряд вспомогательных функций.

```
# создаем класс, выдающий константные прогнозы
class ConstantPredictor():
    def fit(self, X, y):
        self.y_ = y
        return self

    def predict(self, X):
        return np.repeat(self.y_, X.shape[0])

    def decision_function(self, X):
        return np.repeat(self.y_, X.shape[0])
```

```

def predict_proba(self, X):
    y_ = self.y_.astype(np.float64)
    return np.repeat([np.hstack([1 - y_, y_])],
                    X.shape[0], axis=0)

# пишем функцию, которая обучает отдельный
# бинарный классификатор
def _fit_binary(estimator, X, y, classes=None):
    """
    Обучает отдельный бинарный классификатор.
    """
    unique_y = np.unique(y)
    if len(unique_y) == 1:
        estimator = _ConstantPredictor().fit(X, unique_y)
    else:
        estimator = clone(estimator)
        estimator.fit(X, y)
    return estimator

# пишем функцию, которая обучает отдельный
# бинарный классификатор по схеме one-vs-one
def _fit_ovo_binary(estimator, X, y, i, j, verbose):
    """
    Обучает отдельный бинарный
    классификатор (one-vs-one).
    """
    cond = np.logical_or(y == i, y == j)
    y = y[cond]
    y_binary = np.empty(y.shape, int)
    y_binary[y == i] = 0
    y_binary[y == j] = 1
    indcond = np.arange(X.shape[0])[cond]

    if verbose:
        print(f"сравниваем класс {i} с классом {j}")
        print(f"индексы наблюдений, участвующих в сравнении:\n{indcond}\n")

    return (
        _fit_binary(
            estimator,
            _safe_split(estimator, X, None, indices=indcond)[0],
            y_binary,
            classes=[i, j],
        ),
        indcond,
    )

# пишем функцию, которая выдает прогнозы
# с помощью отдельного бинарного классификатора
def _predict_binary(estimator, X):
    """
    Выдает прогнозы с помощью отдельного
    бинарного классификатора.
    """
    try:
        # значения решающей функции
        score = np.ravel(estimator.decision_function(X))

```

```

except (AttributeError, NotImplementedError):
    # вероятности положительного класса
    score = estimator.predict_proba(X)[: , 1]
return score

# пишем функцию, которая задает порог для
# прогнозов бинарного классификатора
def _threshold_for_binary_predict(estimator):
    """
    Задает порог для прогнозов
    бинарного классификатора.
    """
    # если есть метод .decision_function()
    if hasattr(estimator, "decision_function"):
        return 0.0
    # в противном случае, т.е. если есть метод
    # .predict_proba()
    else:
        return 0.5

# пишем функцию, которая вычисляет итоговые уверенности на основе
# результатов многоклассовой классификации по схеме one-vs-one
def _ovr_decision_function(predictions, confidences, n_classes, verbose):
    """
    Вычисляет итоговые уверенности, исходя из результатов OvO.

    Параметры
    -----
    predictions : массив формы (n_samples, n_classifiers)
        Классы, спрогнозированные каждым бинарным классификатором.
    confidences : массив формы (n_samples, n_classifiers)
        Значения решающей функции или спрогнозированные вероятности
        положительного класса, полученные с помощью каждого
        бинарного классификатора.
    n_classes : int
        Количество классов. n_classifiers должно быть
        ``n_classes * (n_classes - 1) / 2``.
    """
    n_samples = predictions.shape[0]
    votes = np.zeros((n_samples, n_classes))
    sum_of_confidences = np.zeros((n_samples, n_classes))

    if verbose:
        print(f"инициализируем матрицу голосов:\n{votes}\n")
        print(f"инициализируем матрицу сумм "
              f"уверенностей:\n{sum_of_confidences}\n")

    k = 0
    for i in range(n_classes):
        for j in range(i + 1, n_classes):

            if verbose:
                print(f"сравниваем класс {i} с классом {j}\n")

            sum_of_confidences[:, i] -= confidences[:, k]
            sum_of_confidences[:, j] += confidences[:, k]

```

```

votes[predictions[:, k] == 0, i] += 1
votes[predictions[:, k] == 1, j] += 1
k += 1

if verbose:
    print(f"матрица голосов:\n{votes}\n")
    print(f"матрица сумм уверенностей:\n{sum_of_confidences}\n")

# Выполняем монотонное преобразование сумм уверенностей в (-1/3, 1/3)
# и потом добавим к голосам. Монотонное преобразование выглядит так:
# f: x -> x / (3 * (|x| + 1)), используем 1/3 вместо 1/2, чтобы
# не изменить порядок голосования.
# Мотивация состоит в том, чтобы использовать степени уверенности
# как способ разорвать связи в голосовании (ситуации, когда у
# нескольких классов - одинаковое количество голосов), не изменяя
# какое-либо решение на противоположное, исходя из разницы в
# в 1 голос.
transformed_confidences = sum_of_confidences / (
    3 * (np.abs(sum_of_confidences) + 1)
)

# получаем итоговые суммы уверенностей
final_confidences = votes + transformed_confidences

if verbose:
    print("подвергаем матрицу сумм уверенностей монотонному\n"
          "преобразованию x / (3 * (|x| + 1))\n")
    print(f"матрица преобразованных сумм уверенностей:\n"
          f"{transformed_confidences}\n")
    print(f"матрица итоговых сумм уверенностей:\n"
          f"{final_confidences}\n")

return final_confidences

```

Наконец, реализуем собственный класс CustomOneVsOneClassifier, выполняющий многоклассовую классификацию по схеме «один против одного».

```

# реализуем собственный класс, выполняющий многоклассовую
# классификацию по схеме one-vs-one
class CustomOneVsOneClassifier(MetaEstimatorMixin,
                               ClassifierMixin,
                               BaseEstimator):

    def __init__(self, estimator, n_jobs=None, verbose=False):
        self.estimator = estimator
        self.n_jobs = n_jobs
        self.verbose = verbose

    def fit(self, X, y):
        """
        Обучает соответствующие классификаторы.

        Параметры
        -----
        X : массив формы (n_samples, n_features)
            Массив признаков.

```

```

у : массив формы (n_samples,)
    Массив меток.

Возвращает
-----
self : объект
    Обученная модель.
"""
# записываем уникальные значения у
self.classes_ = np.unique(y)

if len(self.classes_) == 1:
    raise ValueError(
        "CustomOneVsOneClassifier нельзя обучить, "
        "если нет ни одного класса."
    )
# записываем количество классов
self.n_classes = self.classes_.shape[0]

if self.verbose:
    print(f"количество классов: {self.n_classes}\n")

# записываем список из двух кортежей, количество элементов
# в кортежах определяется n_classes * (n_classes - 1) / 2
# бинарных классификаторов, в первый кортеж записаны
# экземпляры моделей - бинарных классификаторов, во второй
# кортеж записаны массивы наблюдений, которые использовались
# для обучения соответствующего бинарного классификатора
estimators_indices = list(
    zip(
        *(
            Parallel(n_jobs=self.n_jobs)(
                delayed(_fit_ovo_binary)(
                    self.estimate, X, y, self.classes_[i],
                    self.classes_[j], self.verbose
                )
            )
            for i in range(self.n_classes)
            for j in range(i + 1, self.n_classes)
        )
    )
)

# записываем кортеж, состоящий из моделей
# - бинарных классификаторов
self.estimate_ = estimators_indices[0]

return self

def predict(self, X):
    """
    Выдает метку класса с наибольшей уверенностью
    для каждого наблюдения в массиве признаков X.
    Прогнозом будет ``argmax(decision_function(X), axis=1)``.

```

```

Параметры
-----
X : массив формы (n_samples, n_features)
    Массив признаков.

Returns
-----
y : массив формы [n_samples]
    Массив спрогнозированных меток классов.
"""
# получаем значения решающей функции
Y = self.decision_function(X)
# если два класса
if self.n_classes == 2:
    # задаем порог
    thresh = _threshold_for_binary_predict(self.estimated_[0])
    # получаем прогнозы на основе порога
    pred = self.classes_[(Y > thresh).astype(int)]
    return pred
pred = self.classes_[Y.argmax(axis=1)]
if self.verbose:
    print(f"Прогнозом будет индекс с максимальной\n"
          f"итоговой суммой уверенностей:\n{pred}")
return pred

def decision_function(self, X):
    """
    Решающая функция OneVsOneClassifier.
    Значения решающей функции для наблюдений вычисляются путем
    добавления к голосам нормализованной суммы уверенностей,
    вычисленных в результате попарных сравнений классов,
    чтобы устранить неоднозначность, когда классы получают
    одинаковое количество голосов, образуя связь.

    Параметры
    -----
    X : массив формы (n_samples, n_features)
        Массив признаков.

    Возвращает
    -----
    Y : массив формы (n_samples, n_classes) или (n_samples,)
        Результат вызова .decision_function() итоговой модели.
    """

    # получаем массив, состыкованный из len(self.estimated_)
    # массивов для получения прогнозов
    Xs = [X] * len(self.estimated_)

    # получаем прогнозы с помощью бинарных классификаторов
    predictions = np.vstack(
        [est.predict(Xi) for est, Xi in zip(self.estimated_, Xs)]
    ).T

    if self.verbose:
        print(f"матрица прогнозов бинарных "
              f"классификаторов:\n{predictions}\n")

```

```

# получаем уверенности с помощью бинарных классификаторов
confidences = np.vstack(
    [_predict_binary(est, Xi)
     for est, Xi in zip(self.estimated_, Xs)]
).T

if self.verbose:
    print(f"матрица уверенностей бинарных "
          f"классификаторов:\n{confidences}\n")

# получаем итоговые уверенности
Y = _ovr_decision_function(predictions, confidences,
                           len(self.classes_), self.verbose)

# если два класса
if self.n_classes == 2:
    return Y[:, 1]
return Y

```

Давайте проверим наш класс `CustomOneVsOneClassifier` в работе. Обучим с его помощью модель логистической регрессии по схеме «один против одного» и получим прогнозы для нашего игрушечного обучающего массива признаков. В классе есть возможность самокомментирования (включается с помощью параметра `verbose`), поэтому мы увидим, что происходит в процессе обучения и прогнозирования.

```

# строим логистическую регрессию по схеме one-vs-one
lr_ovo_cust_classifier = CustomOneVsOneClassifier(
    LogisticRegression(), verbose=True).fit(X_trn, y_trn)
# получим прогнозы для игрушечного
# обучающего массива признаков
lr_ovo_cust_classifier_tr_pred = lr_ovo_cust_classifier.predict(X_trn)
lr_ovo_cust_classifier_tr_pred

```

количество классов: 3

сравниваем класс 0 с классом 1  
индексы наблюдений, участвующих в сравнении:  
[0 2 3 5]

сравниваем класс 0 с классом 2  
индексы наблюдений, участвующих в сравнении:  
[0 1 2 4 5]

сравниваем класс 1 с классом 2  
индексы наблюдений, участвующих в сравнении:  
[1 3 4]

матрица прогнозов бинарных классификаторов:

```

[[0 0 1]
 [1 1 1]
 [0 0 1]
 [1 1 1]
 [0 0 1]
 [0 0 1]]

```

матрица уверенностей бинарных классификаторов:

```
[[-2.04743668 -0.75518106 1.23741829]
 [ 0.54659607 0.47450729 0.14616936]
 [-1.38801103 -0.66553044 0.68037941]
 [ 0.66210856 0.54344122 0.11535615]
 [-3.50727061 -0.80156436 2.66134319]
 [-3.61460391 -0.34469341 3.34334993]]
```

инициализируем матрицу голосов:

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

инициализируем матрицу сумм уверенностей:

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

сравниваем класс 0 с классом 1

матрица голосов:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [1. 0. 0.]
 [0. 1. 0.]
 [1. 0. 0.]
 [1. 0. 0.]]
```

матрица сумм уверенностей:

```
[ [ 2.04743668 -2.04743668 0. ]
 [-0.54659607 0.54659607 0. ]
 [ 1.38801103 -1.38801103 0. ]
 [-0.66210856 0.66210856 0. ]
 [ 3.50727061 -3.50727061 0. ]
 [ 3.61460391 -3.61460391 0. ]]
```

сравниваем класс 0 с классом 2

матрица голосов:

```
[[2. 0. 0.]
 [0. 1. 1.]
 [2. 0. 0.]
 [0. 1. 1.]
 [2. 0. 0.]
 [2. 0. 0.]]
```

матрица сумм уверенностей:

```
[ [ 2.80261774 -2.04743668 -0.75518106]
 [-1.02110336 0.54659607 0.47450729]
 [ 2.05354147 -1.38801103 -0.66553044]
```

```
[-1.20554979  0.66210856  0.54344122]
[ 4.30883496 -3.50727061 -0.80156436]
[ 3.95929733 -3.61460391 -0.34469341]]
```

сравниваем класс 1 с классом 2

матрица голосов:

```
[[2. 0. 1.]
 [0. 1. 2.]
 [2. 0. 1.]
 [0. 1. 2.]
 [2. 0. 1.]
 [2. 0. 1.]]
```

матрица сумм уверенностей:

```
[[ 2.80261774 -3.28485496  0.48223722]
 [-1.02110336  0.40042671  0.62067665]
 [ 2.05354147 -2.06839044  0.01484897]
 [-1.20554979  0.54675242  0.65879737]
 [ 4.30883496 -6.1686138   1.85977883]
 [ 3.95929733 -6.95795384  2.99865652]]
```

подвергаем матрицу сумм уверенностей монотонному преобразованию  $x / (3 * (|x| + 1))$

матрица преобразованных сумм уверенностей:

```
[[ 0.24567442 -0.25553996  0.10844805]
 [-0.16840692  0.09531064  0.12765792]
 [ 0.22417047 -0.22469874  0.00487723]
 [-0.18219944  0.11782804  0.13238454]
 [ 0.27054492 -0.28683434  0.2167742 ]
 [ 0.26611951 -0.29144652  0.249972  ]]
```

матрица итоговых сумм уверенностей:

```
[[ 2.24567442 -0.25553996  1.10844805]
 [-0.16840692  1.09531064  2.12765792]
 [ 2.22417047 -0.22469874  1.00487723]
 [-0.18219944  1.11782804  2.13238454]
 [ 2.27054492 -0.28683434  1.2167742 ]
 [ 2.26611951 -0.29144652  1.249972  ]]
```

Прогнозом будет индекс с максимальной итоговой суммой уверенностей:

```
[0 2 0 2 0 0]
```

```
argmax([0, 2, 0, 2, 0, 0])
```

Теперь получим прогнозы для нашего игрушечного тестового массива признаков.

```
# получим прогнозы для игрушечного
# тестового массива признаков
lr_ovo_cust_classifier_tst_pred = lr_ovo_cust_classifier.predict(X_tst)
lr_ovo_cust_classifier_tst_pred
```

матрица прогнозов бинарных классификаторов:

```
[[0 1 1]
 [1 1 0]
 [0 0 1]]
```

матрица уверенностей бинарных классификаторов:

```
[[-0.25562981  0.40092909  0.86834572]
 [ 0.71850161  0.49027405 -0.00858272]
 [-4.10030643 -0.30173175  3.8903464  ]]
```

инициализируем матрицу голосов:

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

инициализируем матрицу сумм уверенностей:

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

сравниваем класс 0 с классом 1

матрица голосов:

```
[[1. 0. 0.]
 [0. 1. 0.]
 [1. 0. 0.]]
```

матрица сумм уверенностей:

```
[ [ 0.25562981 -0.25562981  0.      ]
 [-0.71850161  0.71850161  0.      ]
 [ 4.10030643 -4.10030643  0.      ]]
```

сравниваем класс 0 с классом 2

матрица голосов:

```
[[1. 0. 1.]
 [0. 1. 1.]
 [2. 0. 0.]]
```

матрица сумм уверенностей:

```
[[-0.14529928 -0.25562981  0.40092909]
 [-1.20877566  0.71850161  0.49027405]
 [ 4.40203817 -4.10030643 -0.30173175]]
```

сравниваем класс 1 с классом 2

матрица голосов:

```
[[1. 0. 2.]
 [0. 2. 1.]
 [2. 0. 1.]]
```

матрица сумм уверенностей:

```
[[-0.14529928 -1.12397552  1.2692748  ]
 [-1.20877566  0.72708433  0.48169133]
 [ 4.40203817 -7.99065283  3.58861465]]
```

подвергаем матрицу сумм уверенностей монотонному преобразованию  $x / (3 * (|x| + 1))$

матрица преобразованных сумм уверенностей:

```
[[ -0.04228859 -0.17639493  0.18644353 ]
 [ -0.18242016  0.14032983  0.1083652  ]
 [  0.27162823 -0.29625779  0.26068977 ]]
```

матрица итоговых сумм уверенностей:

```
[[  0.95771141 -0.17639493  2.18644353 ]
 [ -0.18242016  2.14032983  1.1083652  ]
 [  2.27162823 -0.29625779  1.26068977 ]]
```

Прогнозом будет индекс с максимальной итоговой суммой уверенностей:

```
[2 1 0]
```

```
argmax([2, 1, 0])
```

### 7.24.3. Подход «коды, исправляющие ошибки» («error-correcting output codes»)

Теперь разберем третий подход «коды, исправляющие ошибки» (error-correcting output codes – ECOC). В рамках подхода «коды, исправляющие ошибки» каждый класс будет представлен в евклидовом пространстве, где каждое измерение может быть только 0 или 1. Другими словами, каждый класс представлен бинарным кодом (массивом нулей и единиц). Матрица, которая отслеживает местоположение/код каждого класса, называется **кодовой книгой (code book)**. **Размер кода (code size)** – это размерность вышеупомянутого пространства. Интуитивно понятно, что каждый класс должен быть представлен как можно более уникальным кодом, а для оптимизации точности классификации требуется хорошая кодовая книга. В реализации scikit-learn используется случайно сгенерированная кодовая книга, хотя могут применяться более сложные методы.

На этапе обучения для каждого бита в кодовой книге мы строим один бинарный классификатор. На этапе прогнозирования классификаторы используются для проецирования новых точек в пространстве классов, и выбирается класс, ближайший к точкам.

В библиотеке scikit-learn подход «коды, исправляющие ошибки» реализован в виде класса `OutputCodeClassifier`. В `OutputCodeClassifier` параметр `code_size` позволяет пользователю контролировать количество используемых классификаторов. Он представляет собой процент от общего количества классов. Число от 0 до 1 потребует меньше классификаторов, чем подход «один против остальных». Теоретически  $\log_2(n\_classes)/n\_classes$  достаточно для однозначного представления каждого класса. Однако на практике это может привести к не очень хорошему качеству прогнозов, поскольку  $\log_2(n\_classes)$  намного меньше, чем  $n\_classes$ . Число больше 1 потребует больше классификаторов, чем подход «один против остальных». В этом случае некоторые классификаторы теоретически будут исправлять ошибки, допущенные другими классификаторами, отсюда и название «исправление ошибок». Однако на практике этого может не произойти, поскольку ошибки классификаторов обычно коррелируют. Подход ECOC демонстрирует эффект, аналогичный бэггингу.

Обратите внимание, что у класса `OutputCodeClassifier` нет методов `.predict_proba()` и `.decision_function()`.

На примере задачи Otto Group Product Classification Challenge мы выполним многоклассовую классификацию по схеме ECOC с помощью логистической регрессии, воспользовавшись специальным классом `OutputCodeClassifier`.

```
# создаем экземпляр класса OutputCodeClassifier
lr_ecoc_classifier = OutputCodeClassifier(lr_pipe)
# применяем LogisticRegression для многоклассовой
# классификации по схеме ECOC через
# класс OutputCodeClassifier
lr_ecoc_classifier.fit(X_otto_train, y_otto_train);

# вычисляем прогнозы для тестовой выборки
lr_ecoc_classifier_pred = lr_ecoc_classifier.predict(X_otto_test)
lr_ecoc_classifier_pred[:5]

array([1, 6, 5, 5, 5])
```

Давайте подробнее разберем, что происходит под капотом подхода «коды, исправляющие ошибки» на примере уже знакомых нам игрушечных данных.

Обучение начинается с того, что корректируется значение параметра `code_size`. Параметр `code_size` задан равным 1,5, мы умножаем это значение на количество классов и берем целое число от результата.

Кодовая книга имеет размер `n_classes × code_size`:  $3 \times 4 = 12$ . Инициализируем ее случайным образом. Допустим, получили кодовую книгу следующего вида:

```
исходная кодовая книга:
[[0.37454012 0.95071431 0.73199394 0.59865848]
 [0.15601864 0.15599452 0.05808361 0.86617615]
 [0.60111501 0.70807258 0.02058449 0.96990985]]
```

Выполняем первую корректировку кодовой книги: все значения больше 0.5 приравниваем к 1.

```
кодовая книга
после первой корректировки
[[0.37454012 1.      1.      1.      ]
 [0.15601864 0.15599452 0.05808361 1.      ]
 [1.      1.      0.02058449 1.      ]]
```

Затем выполняем вторую корректировку кодовой книги: если есть метод `.decision_function()`, все значения, не являющиеся 1, приравниваем к -1, если нет метода `.decision_function()`, все значения, не являющиеся 1, приравниваем к 0. Допустим, у нас есть метод `.decision_function()`.

```
кодовая книга
после второй корректировки
[[-1.  1.  1.  1.]
 [-1. -1. -1.  1.]
 [ 1.  1. -1.  1.]]
```

Итоговую кодовую книгу используем для кодировки классов. Первая строка  $-1 \ 1 \ 1 \ 1$  будет использована для кодировки класса 0. Вторая строка  $-1 \ -1 \ -1 \ 1$  будет использована для класса 1. Третья строка будет использована  $1 \ 1 \ -1 \ 1$  для класса 2. В итоге наш обучающий массив меток

y_trn
0
2
0
1
2
0

будет преобразован в массив меток следующего вида:

```

новый массив меток
[[-1  1  1  1]
 [ 1  1 -1  1]
 [-1  1  1  1]
 [-1 -1 -1  1]
 [ 1  1 -1  1]
 [-1  1  1  1]]

```

Массив состоит из 4 столбцов, один столбец является константным. Мы обучим три модели машинного обучения – бинарные классификаторы, а для константного столбца обучим модель, всегда выдающую константные прогнозы. Затем получим прогнозы с помощью всех 4 моделей. Допустим, получили следующую матрицу прогнозов.

```

predictions:
[[-0.95401785  2.54618834  0.63156985  1.          ]
 [-0.57325232  0.51581038 -1.16309398  1.          ]
 [-1.00934633  1.96254221  0.44723263  1.          ]
 [-0.54662428  0.42968979 -1.2602978   1.          ]
 [-0.72776042  3.88431188  0.85418853  1.          ]
 [-0.39706322  4.12206578  0.30924707  1.          ]]

```

Теперь вычисляем евклидовы расстояния между прогнозами и значениями кодовой книги. Получаем матрицу евклидовых расстояний.

```

code book                predictions:
[[-1.  1.  1.  1.]        [[-0.95401785  2.54618834  0.63156985  1.          ]
 [-1. -1. -1.  1.]        [-0.57325232  0.51581038 -1.16309398  1.          ]
 [ 1.  1. -1.  1.]]        [-1.00934633  1.96254221  0.44723263  1.          ]
                               [-0.54662428  0.42968979 -1.2602978   1.          ]
                               [-0.72776042  3.88431188  0.85418853  1.          ]
                               [-0.39706322  4.12206578  0.30924707  1.          ]]

distances:

```

```
[[1.59014261 3.90379127 2.97840634]
 [2.25732778 1.5831596 1.65413485]
 [1.11001199 3.29715422 2.65677294]
 [2.37481566 1.52227376 1.6688485 ]
 [2.90079823 5.2315038 3.83958672]
 [3.25391565 5.32101668 3.66240471]]
```

В качестве прогноза по каждой строке матрицы расстояний берем индекс с минимальным расстоянием.

```
distances.argmax(axis=1)
```

distances		
1.59014261	3.90379127	2.97840634
2.25732778	1.5831596	1.65413485
1.11001199	3.29715422	2.65677294
2.37481566	1.52227376	1.6688485
2.90079823	5.2315038	3.83958672
3.25391565	5.32101668	3.66240471

predictions
0
1
0
1
0
0

Давайте реализуем собственный класс `CustomOutputCodeClassifier`, выполняющий многоклассовую классификацию согласно подходу «коды, исправляющие ошибки».

```
# реализуем собственный класс, выполняющий
# многоклассовую классификацию по схеме ECOC
class CustomOutputCodeClassifier(MetaEstimatorMixin,
                                ClassifierMixin,
                                BaseEstimator):
    """
    Класс, реализующий многоклассовую классификацию
    согласно подходу Error-Correcting Output Code

    Параметры
    -----
    estimator : объект - экземпляр класса
        Экземпляр класса, в котором реализована модель
        машинного обучения. У него должен быть либо метод
        .decision_function(), либо метод .predict_proba().
    code_size : float, значение по умолчанию 1.5
        Процент от количества классов для создания кодовой книги.
        Значение от 0 до 1 потребует меньшее количество
        классификаторов, чем подход "один против остальных".
        Значение выше 1 потребует большее число классификаторов,
        чем подход "один против остальных".
    metric: str, значение по умолчанию 'euclidean'
        Метрика расстояния.
    random_state : int, значение по умолчанию None
        Стартовое значение генератора псевдослучайных чисел.
    n_jobs : int, значение по умолчанию None
        Количество ядер процессора для распараллеливания.
```

`verbose`: bool, значение по умолчанию False  
Печатает процесс обучения.

Атрибуты

```
-----
estimators_ : список `int(n_classes * code_size)` классификаторов
              Классификаторы, используемые для предсказания.
classes_ : ndarray of shape (n_classes,)
           Массив с метками.
code_book_ : ndarray формы (n_classes, code_size)
           Массив бинарных значений, содержащий код
           каждого класса.
"""
```

```
def __init__(self, estimator, code_size=1.5, metric='euclidean',
             random_state=None, n_jobs=None, verbose=False):
    self.estimator = estimator
    self.code_size = code_size
    self.metric = metric
    self.random_state = random_state
    self.n_jobs = n_jobs
    self.verbose = verbose
```

```
def fit(self, X, y):
```

```
"""
Обучает классификаторы.
```

Параметры

```
-----
X : массив формы (n_samples, n_features)
   Массив признаков.
y : массив формы (n_samples,)
   Массив меток.
```

Возвращает

```
-----
self : object
      Обученная модель.
"""
```

```
self.classes_ = np.unique(y)
n_classes = self.classes_.shape[0]
if n_classes == 0:
    raise ValueError(
        "CustomOutputCodeClassifier нельзя обучить, "
        "если нет ни одного класса.")
```

```
code_size_ = int(n_classes * self.code_size)
```

```
if self.verbose:
    print(f"количество классов: {n_classes}")
    print(f"code_size с поправкой на n_classes: {code_size}")
    print(f"размер кодовой книги n_classes x "
          f"code_size: {n_classes} x {code_size}")
    print("")
```

```

rng = np.random.RandomState(self.random_state)
self.code_book_ = rng.uniform(size=(n_classes, code_size_))

if self.verbose:
    print(f"исходная кодовая книга:\n{self.code_book_}")
    print("")

self.code_book_[self.code_book_ > 0.5] = 1

if self.verbose:
    print(f"первая корректировка кодовой книги:\n"
          f"все значения больше 0.5 приравниваем к 1:\n"
          f"{self.code_book_}\n")

if hasattr(self.estimator, "decision_function"):
    self.code_book_[self.code_book_ != 1] = -1
else:
    self.code_book_[self.code_book_ != 1] = 0

if self.verbose:
    print(f"проверка наличия метода .decision_function(): "
          f"{hasattr(self.estimator, 'decision_function')}\n")
    print(f"вторая корректировка кодовой книги:\n"
          f"если есть метод .decision_function(), все значения,\n"
          f"не являющиеся 1, приравниваем к -1\n"
          f"если нет метода .decision_function(), все значения,\n"
          f"не являющиеся 1, приравниваем к 0\n"
          f"{self.code_book_}\n")

classes_index = {c: i for i, c in enumerate(self.classes_)}

if self.verbose:
    print(f"индексы классов:\n{classes_index}")
    print("")

Y = np.array(
    [self.code_book_[classes_index[y[i]]]
     for i in range(y.shape[0])],
    dtype=int,
)

self.estimated_ = Parallel(n_jobs=self.n_jobs)(
    delayed(_fit_binary)(self.estimator, X, Y[:, i])
    for i in range(Y.shape[1])
)

if self.verbose:
    print(f"массив меток:\n{Y}\n")
    print(f"_ConstantPredictor() нужен для ситуаций, когда "
          f"столбец является константным")
    print(f"список обученных моделей:\n", self.estimated_)
    print("")

return self

```

```

def predict(self, X):
    """
    Получаем прогнозы с помощью соответствующих классификаторов.

    Параметры
    -----
    X : массив формы (n_samples, n_features)
        Data.
    Массив признаков
    -----
    y : массив формы (n_samples,)
        Массив спрогнозированных меток классов.
    """

    Y = np.array([_predict_binary(e, X)
                  for e in self.estimators_]).T
    dist = pairwise_distances(Y, self.code_book_,
                              metric=self.metric)
    pred = dist.argmin(axis=1)

    if self.verbose:
        print(f"прогнозы бинарных классификаторов:\n{Y}\n")
        print(f"расстояния:\n{dist}\n")
        print(f"итоговые прогнозы:\n{pred}")

    return self.classes_[pred]

```

Проверим наш класс `CustomOneVsOneClassifier` в работе. Обучим с его помощью модель логистической регрессии по схеме «коды, исправляющие ошибки» и получим прогнозы для нашего игрушечного обучающего массива признаков. В классе есть возможность самокомментирования (включается с помощью параметра `verbose`), поэтому мы увидим, что происходит в процессе обучения и прогнозирования.

```

# строим логистическую регрессию по схеме ECOC
lr_ecoc_cust_classifier = CustomOutputCodeClassifier(
    LogisticRegression(),
    random_state=42, verbose=True)
lr_ecoc_cust_classifier.fit(X_trn, y_trn)
# получим прогнозы для игрушечного
# обучающего массива признаков
lr_ecoc_cust_classifier_tr_pred = lr_ecoc_cust_classifier.predict(X_trn)
lr_ecoc_cust_classifier_tr_pred

```

```

количество классов: 3
code_size с поправкой на n_classes: 4
размер кодовой книги n_classes x code_size: 3 x 4

```

```

исходная кодовая книга:
[[0.37454012 0.95071431 0.73199394 0.59865848]
 [0.15601864 0.15599452 0.05808361 0.86617615]
 [0.60111501 0.70807258 0.02058449 0.96990985]]

```

```

первая корректировка кодовой книги:
все значения больше 0.5 приравниваем к 1:

```

```
[ [0.37454012 1. 1. 1. ]
  [0.15601864 0.15599452 0.05808361 1. ]
  [1. 1. 0.02058449 1. ] ]]
```

проверка наличия метода `.decision_function()`: True

вторая корректировка кодовой книги:  
если есть метод `.decision_function()`, все значения,  
не являющиеся 1, приравниваем к -1  
если нет метода `.decision_function()`, все значения,  
не являющиеся 1, приравниваем к 0

```
[ [-1. 1. 1. 1.]
  [-1. -1. -1. 1.]
  [ 1. 1. -1. 1.]]]
```

индексы классов:  
{0: 0, 1: 1, 2: 2}

массив меток:  
[ [-1 1 1 1]  
 [ 1 1 -1 1]  
 [-1 1 1 1]  
 [-1 -1 -1 1]  
 [ 1 1 -1 1]  
 [-1 1 1 1]]]

`_ConstantPredictor()` нужен для ситуаций, когда столбец является константным  
список обученных моделей:

```
[LogisticRegression(), LogisticRegression(), LogisticRegression(), <__main__._.  
ConstantPredictor object at 0x7fa9f28cfbe0>]
```

прогнозы бинарных классификаторов:  
[ [-0.95401785 2.54618834 0.63156985 1. ]  
 [-0.57325232 0.51581038 -1.16309398 1. ]  
 [-1.00934633 1.96254221 0.44723263 1. ]  
 [-0.54662428 0.42968979 -1.2602978 1. ]  
 [-0.72776042 3.88431188 0.85418853 1. ]  
 [-0.39706322 4.12206578 0.30924707 1. ]]

расстояния:  
[[1.59014261 3.90379127 2.97840634]  
 [2.25732778 1.5831596 1.65413485]  
 [1.11001199 3.29715422 2.65677294]  
 [2.37481566 1.52227376 1.6688485 ]  
 [2.90079823 5.2315038 3.83958672]  
 [3.25391565 5.32101668 3.66240471]]]

итоговые прогнозы:  
[0 1 0 1 0 0]

`aggay([0, 1, 0, 1, 0, 0])`

Можно подробнее рассмотреть, как вычисляются расстояния.

*# пишем функцию, вычисляющую евклидово расстояние*

```
def euclidean_distance(x1, x2):
    """
    Вычисляет евклидово расстояние
    между двумя векторами.
    """
    distance = 0
    for i in range(len(x1)):
        distance += pow((x1[i] - x2[i]), 2)
    return np.sqrt(distance)
```

*# кодовая книга*

```
code_book = np.array([[ -1.,  1.,  1.,  1.],
                      [ -1., -1., -1.,  1.],
                      [  1.,  1., -1.,  1.]])
```

*# прогнозы бинарных классификаторов*

```
binary_cl_pred = np.array(
    [[-0.95401785,  2.54618834,  0.63156985,  1.],
     [-0.57325232,  0.51581038, -1.16309398,  1.],
     [-1.00934633,  1.96254221,  0.44723263,  1.],
     [-0.54662428,  0.42968979, -1.2602978,  1.],
     [-0.72776042,  3.88431188,  0.85418853,  1.],
     [-0.39706322,  4.12206578,  0.30924707,  1.]
])
```

*# вычисляем расстояния*

```
print(euclidean_distance(code_book[0], binary_cl_pred[0]))
print(euclidean_distance(code_book[0], binary_cl_pred[1]))
print(euclidean_distance(code_book[0], binary_cl_pred[2]))
print(euclidean_distance(code_book[0], binary_cl_pred[3]))
print(euclidean_distance(code_book[0], binary_cl_pred[4]))
print(euclidean_distance(code_book[0], binary_cl_pred[5]))
print("")
print(euclidean_distance(code_book[1], binary_cl_pred[0]))
print(euclidean_distance(code_book[1], binary_cl_pred[1]))
print(euclidean_distance(code_book[1], binary_cl_pred[2]))
print(euclidean_distance(code_book[1], binary_cl_pred[3]))
print(euclidean_distance(code_book[1], binary_cl_pred[4]))
print(euclidean_distance(code_book[1], binary_cl_pred[5]))
print("")
print(euclidean_distance(code_book[2], binary_cl_pred[0]))
print(euclidean_distance(code_book[2], binary_cl_pred[1]))
print(euclidean_distance(code_book[2], binary_cl_pred[2]))
print(euclidean_distance(code_book[2], binary_cl_pred[3]))
print(euclidean_distance(code_book[2], binary_cl_pred[4]))
print(euclidean_distance(code_book[2], binary_cl_pred[5]))
```

```
1.5901426087931865
2.2573277867455066
1.1100119932923562
2.3748156610146824
2.9007982340747644
3.2539156574237387
```

```
3.903791269560861
1.5831596056031012
3.2971542252756194
1.5222737545529064
5.231503802427298
5.321016679698285
```

```
2.9784063383459958
1.6541348484369
2.6567729419829744
1.6688485083435831
3.839586721876623
3.662404710791444
```

Теперь получим прогнозы для нашего игрушечного тестового массива признаков.

```
# получим прогнозы для игрушечного
# тестового массива признаков
lr_ecoc_cust_classifier_tst_pred = lr_ecoc_cust_classifier.predict(X_tst)
lr_ecoc_cust_classifier_tst_pred
```

```
прогнозы бинарных классификаторов:
[[-0.48172898  1.2365915 -0.98211356  1.          ]
 [-0.59286446  0.36135728 -1.20187549  1.          ]
 [-0.28194127  4.5849551  0.31210383  1.          ]]
```

```
расстояния:
[[2.06236625  2.29592384  1.50060529]
 [2.32849298  1.43520267  1.72795701]
 [3.72031082  5.78177726  4.02702036]]
```

```
итоговые прогнозы:
[2 1 0]
```

```
aggau([2, 1, 0])
```

## Другие полезные библиотеки

### 1. БИБЛИОТЕКИ ВИЗУАЛИЗАЦИИ MATPLOTLIB, SEABORN И PLOTLY

#### 1.1. MATPLOTLIB

Matplotlib (произносится как матплотлиб) – это основная библиотека для построения научных графиков в Python. Она включает функции для создания визуализаций типа линейных диаграмм, круговых диаграмм, гистограмм, диаграмм разброса и т. д. Библиотека matplotlib имеет иерархическую структуру. Наиболее простыми для понимания являются высокоуровневые функции. Поэтому знакомство с библиотекой matplotlib обычно начинают с самого высокоуровневого интерфейса, предлагаемого модулем pyplot.

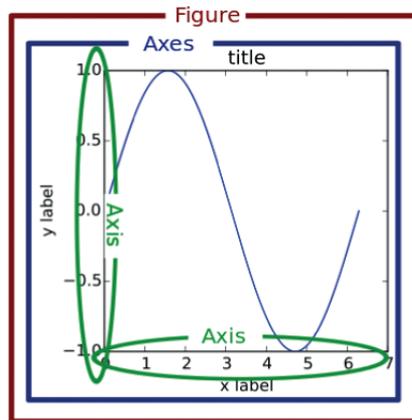


Рис. 1 Уровни рисунка в matplotlib

Построение рисунка в matplotlib подразумевает операции с разными уровнями: Figure (Изображение), Axes (Область рисования), Axis (Координатная ось), Artists (элементы рисунка). Таким образом, любой рисунок в matplotlib имеет иерархическую структуру.

Изображение (Figure) – высший уровень иерархии. Он содержит:

- область рисования (Axes);

- элементы рисунка Artists (заголовки, легенда и т. д.);
- основа-холст (Canvas).

На рисунке может быть несколько областей рисования Axes, но данная область рисования Axes может принадлежать только одной области Figure.

Область рисования (Axes) является объектом среднего уровня. Она представляет собой часть изображения с пространством данных. Каждая область рисования Axes содержит две (или три в случае трехмерных данных) координатные оси (Axis объектов), которые упорядочивают визуализацию данных.

Координатная ось (Axis) также является объектом среднего уровня, которая определяет область изменения данных. Она включает:

- деления (тики) ticks;
- метки делений ticklabels.

Расположение делений определяется объектом Locator (локатором), а формат делений обрабатывает объект Formatter (форматтер). Конфигурация координатных осей заключается в комбинировании различных свойств объектов Locator и Formatter.

Практически все, что отображается на рисунке, является элементом рисунка (Artist). Элементы рисунка Artists бывают двух типов: примитивы и контейнеры. Примитивы – это стандартные графические объекты, которые мы хотим нарисовать на холсте (Line2D, Rectangle, Circle, Ellipse, Polygon, Text, AxesImage). Примитивы, отвечающие за создание геометрических фигур (Rectangle, Circle, Ellipse, Polygon), так и называются – фигурами (patches). Например, мы создаем столбиковую диаграмму для переменной из 4 категорий. У нас будет четыре фигуры – прямоугольника. Контейнеры – это объекты, внутри которых размещаются примитивы.

Когда мы строим график, все элементы рисунка Artists наносятся на основу-холст (Canvas). Происходит связывание примитивов с контейнерами.

Визуализация данных и результатов анализа может дать вам важную информацию, и мы будем использовать matplotlib для построения графиков. При работе в Jupyter Notebook вы можете вывести рисунок прямо в браузере с помощью встроенных команд `%matplotlib notebook` и `%matplotlib inline`. Между командами есть небольшая разница. Команда `%matplotlib notebook` выводит в браузере интерактивный график, тогда как команда `%matplotlib inline` указывает, что график должен быть выведен в браузере как обычная картинка. Данный способ удобен тем, кто проводит очень много экспериментов в рамках одной веб-страницы. Команда `%matplotlib notebook` или команда `%matplotlib inline` задается в первой ячейке тетрадки. Блок кода, строящего график, будет, как правило, завершаться вызовом функции `matplotlib.pyplot.show()`. Функция делает массу операций под капотом и обычно позволяет избавиться от вывода лишней, технической информации о графике.

Давайте импортируем модуль `pyplot` библиотеки `matplotlib` как `plt` и соответственно функции будем называть с учетом этого алиаса: например, `plt.show()` вместо `matplotlib.pyplot.show()`. Затем импортируем библиотеки `pandas` и `NumPy`.

```
# импортируем модуль pyplot библиотеки matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
# импортируем pandas и numpy
import pandas as pd
import numpy as np
```

На экранах с высоким разрешением типа Retina графики в тетрадках по умолчанию выглядят размытыми, используйте настройку ниже для улучшения резкости.

```
# на экранах с высоким разрешением типа Retina графики в тетрадках
# по умолчанию выглядят размытыми, используем эту настройку
# для улучшения резкости
%config InlineBackend.figure_format = 'retina'
```

Кроме того, импортируем модули `dates` и `ticker` для настройки локаторов и форматтеров и модуль `markers` для настройки маркеров.

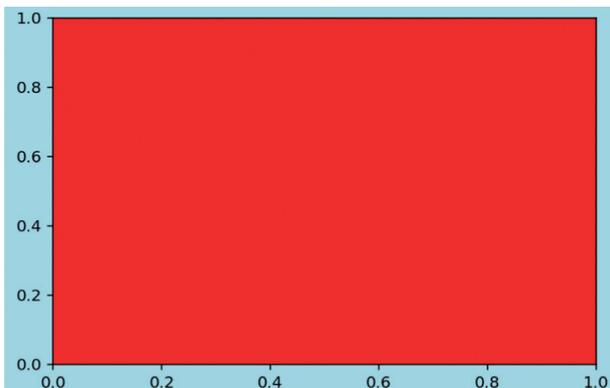
```
# импортируем модули dates и ticker для
# настройки локаторов и форматтеров
import matplotlib.dates as dates
import matplotlib.ticker as ticker
# импортируем модуль markers
from matplotlib import markers
```

Мы всегда начинаем с того, что с помощью функции `plt.figure()` создаем область `Figure`. Метод `.add_subplot()` разбивает область `Figure` на указанное количество строк и столбцов. Теперь область `Figure` можно представить как таблицу (или координатную сетку). Область `Axes` помещается в указанную ячейку. Для всего нужно лишь три числа, которые мы передаем в метод в качестве параметров:

- первое – количество строк;
- второе – количество столбцов;
- третье – индекс ячейки.

```
# создаем область Figure
fig = plt.figure()
# добавляем к области Figure область Axes
# размещаем в первой строке, первом столбце и первой
# (единственной) ячейке на сетке Figure
ax = fig.add_subplot(111)
# задаем цвет области Figure
fig.set(facecolor='lightblue')
# задаем цвет области Axes
ax.set(facecolor='red')

# выводим результат
plt.show()
```



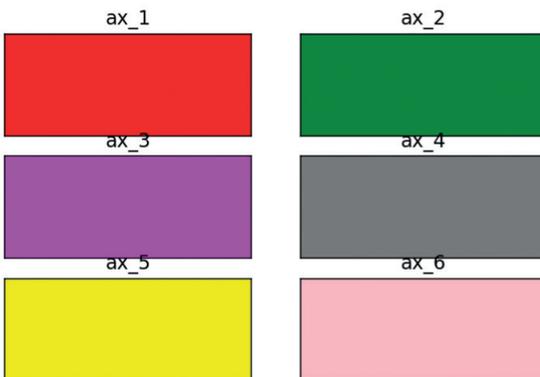
Выделение областей Figure и Axes разными цветами подтверждает, что Figure и Axes – разные области.

Индексирование полученных ячеек начинается с левого верхнего угла, выполняется построчно слева направо и заканчивается в правом нижнем углу. Давайте убедимся в этом.

```
# работаем с Figure как с сеткой
fig = plt.figure()

ax_1 = fig.add_subplot(3, 2, 1)
ax_1.set(facecolor='red', title='ax_1',
         xticks=[], yticks=[])
ax_2 = fig.add_subplot(3, 2, 2)
ax_2.set(facecolor='green', title='ax_2',
         xticks=[], yticks=[])
ax_3 = fig.add_subplot(3, 2, 3)
ax_3.set(facecolor='magenta', title='ax_3',
         xticks=[], yticks=[])
ax_4 = fig.add_subplot(3, 2, 4)
ax_4.set(facecolor='grey', title='ax_4',
         xticks=[], yticks=[])
ax_5 = fig.add_subplot(3, 2, 5)
ax_5.set(facecolor='yellow', title='ax_5',
         xticks=[], yticks=[])
ax_6 = fig.add_subplot(3, 2, 6)
ax_6.set(facecolor='pink', title='ax_6',
         xticks=[], yticks=[])

# выводим результат
plt.show()
```



На практике чаще всего используется функция `plt.subplots(nrows, ncols)`. Она также создает область Figure, сконфигурированную из сетки подграфиков (можно задавать количество строк и столбцов). В ее основе кортеж из двух элементов:

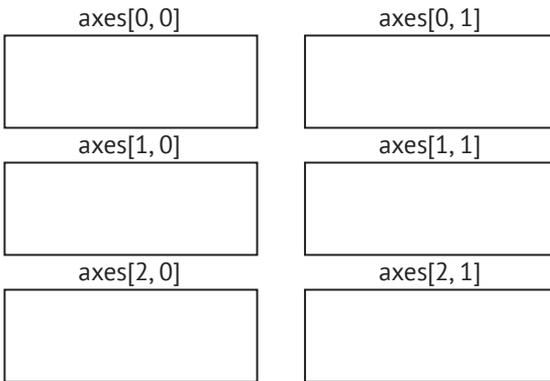
- область Figure;
- один или несколько объектов Axes.

Ниже мы с помощью функции `plt.subplots()` создаем сетку, образованную тремя строками и двумя столбцами.

```
# можно использовать функцию plt.subplots()
fig, axes = plt.subplots(nrows=3, ncols=2)
axes[0, 0].set(title='axes[0, 0]')
axes[0, 1].set(title='axes[0, 1]')
axes[1, 0].set(title='axes[1, 0]')
axes[1, 1].set(title='axes[1, 1]')
axes[2, 0].set(title='axes[2, 0]')
axes[2, 1].set(title='axes[2, 1]')

for ax in axes.flat:
    ax.set(xticks=[], yticks=[])

# выводим результат
plt.show()
```



По умолчанию количество строк и столбцов в методе равно 1, что позволяет быстро создать область `Figure` с одной областью `Axes`.

Давайте загрузим временной ряд, представляющий собой данные о продажах автомобилей.

```
# загружаем временной ряд - ежемесячные
# данные о продажах автомобилей
cars = pd.read_csv('Data/monthly_car_sales.csv',
                  header=0,
                  index_col=0,
                  parse_dates=True).squeeze('columns')

cars.head()

Month
1960-01-01    6550
1960-02-01    8728
1960-03-01   12026
1960-04-01   14395
1960-05-01   14587
Name: Sales, dtype: int64
```

Теперь визуализируем ряд, воспользовавшись функцией `plt.subplots()` и методом `.plot()` класса `Axes`, который построит обычный линейный график.

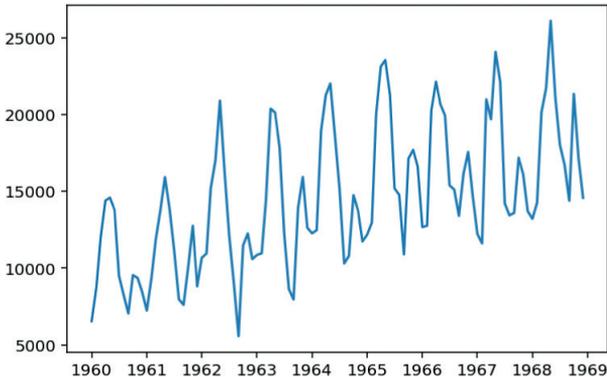
```
# задаем области Figure и Axes
```

```
fig, ax = plt.subplots()
```

```
ax.plot(cars)
```

```
# выводим график
```

```
plt.show()
```



С помощью параметра `figsize` функции `plt.subplots()` увеличим размер графика, а с помощью различных методов класса `Axes` задаем:

- элементы графика (тип маркера, тип линии) – метод `ax.plot()`;
- заголовок графика – метод `ax.set_title()`;
- заголовок оси  $x$  – метод `ax.set_xlabel()`;
- заголовок оси  $y$  – метод `ax.set_ylabel()`;
- цену основных делений оси  $x$  – `ax.xaxis.set_major_locator()`;
- цену промежуточных делений оси  $x$  – `ax.xaxis.set_minor_locator()`;
- цену основных делений оси  $y$  – `ax.yaxis.set_major_locator()`;
- цену промежуточных делений оси  $y$  – `ax.yaxis.set_minor_locator()`;
- ориентацию меток оси  $x$  – метод `ax.tick_params()`;
- ориентацию меток оси  $y$  – метод `ax.tick_params()`;
- формат основных делений оси  $y$  – метод `ax.yaxis.set_major_formatter()`;
- формат промежуточных делений оси  $y$  – `ax.yaxis.set_minor_formatter()`;
- координатную сетку – метод `ax.grid()`.

```
# задаем области Figure и Axes, настроив размер
```

```
fig, ax = plt.subplots(figsize=(14, 8))
```

```
# строим график, настроив тип маркера и тип линии
```

```
ax.plot(cars, marker='D', linestyle='--')
```

```
# задаем заголовок
```

```
ax.set_title('Количество проданных легковых автомобилей\n' +  
            '(1960-1968)')
```

```
# задаем начало оси x с отступом
```

```
ax.margins(x=0.01)
```

```
# задаем ориентацию меток по оси x
```

```
ax.tick_params(axis='x', labelrotation=90)
```

```
# задаем ориентацию меток по оси y
```

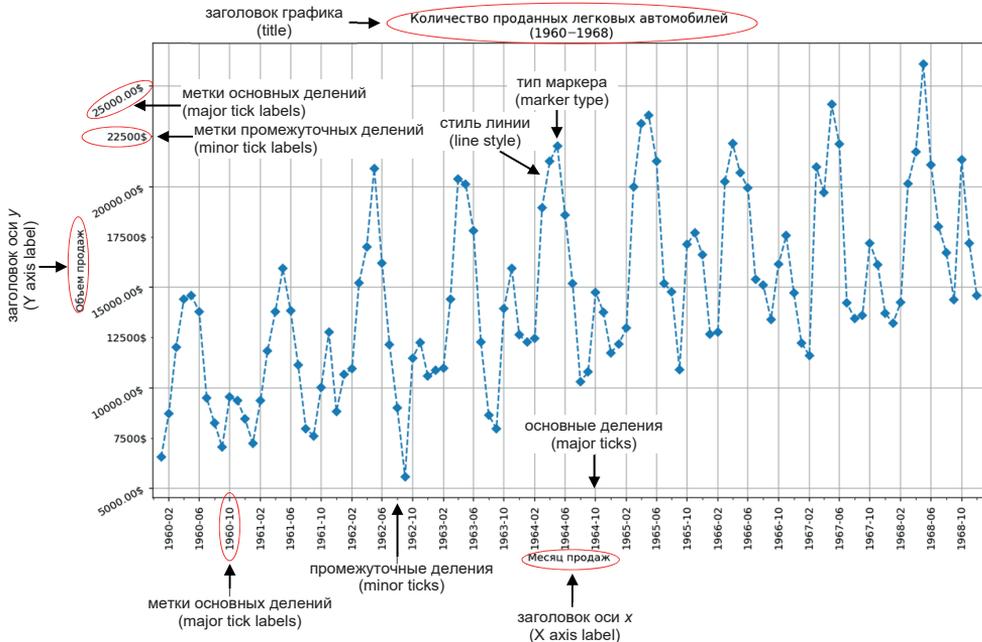
```
ax.tick_params(axis='y', labelrotation=30)
```

```

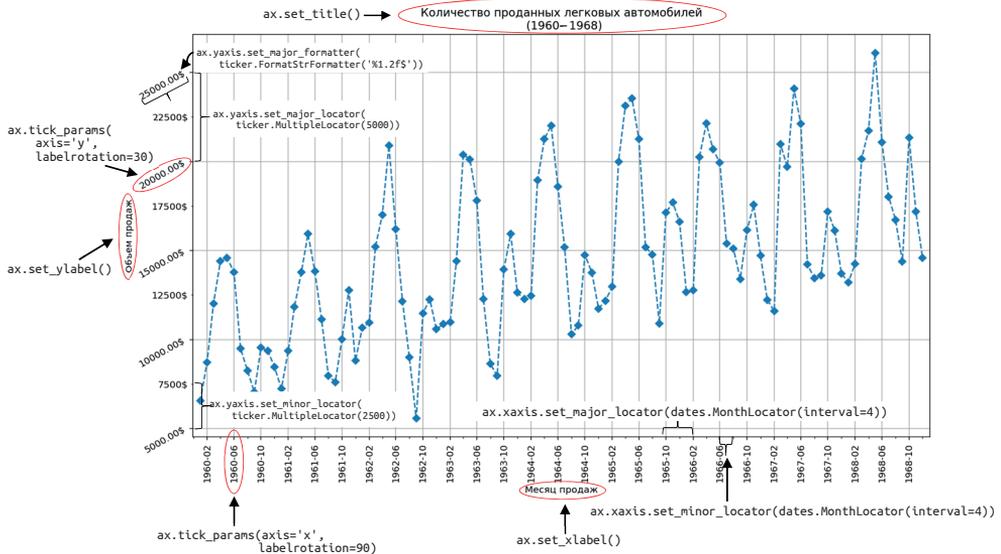
# задаем цену основных делений оси y - 5000
# с помощью локатора MultipleLocator
ax.yaxis.set_major_locator(ticker.MultipleLocator(5000))
# задаем цену промежуточных делений оси y - 2500
# с помощью локатора MonthLocator
ax.yaxis.set_minor_locator(ticker.MultipleLocator(2500))
# задаем цену основных делений оси x - 4 месяца
# с помощью локатора MonthLocator
ax.xaxis.set_major_locator(dates.MonthLocator(interval=4))
# задаем цену промежуточных делений оси x - 2 месяца
# с помощью локатора MonthLocator
ax.xaxis.set_minor_locator(dates.MonthLocator(interval=2))
# задаем формат промежуточных делений оси y -
# целочисленное значение с символом $
ax.yaxis.set_minor_formatter(
    ticker.FormatStrFormatter('%1.0f$'))
# задаем формат основных делений оси y -
# значение с двумя десятичными знаками и
# символом $
ax.yaxis.set_major_formatter(
    ticker.FormatStrFormatter('%1.2f$'))
# задаем координатную сетку
ax.grid()
# задаем заголовок оси x
ax.set_xlabel('Месяц продаж')
# задаем заголовок оси y
ax.set_ylabel('Объем продаж')

# выводим график
plt.show()

```



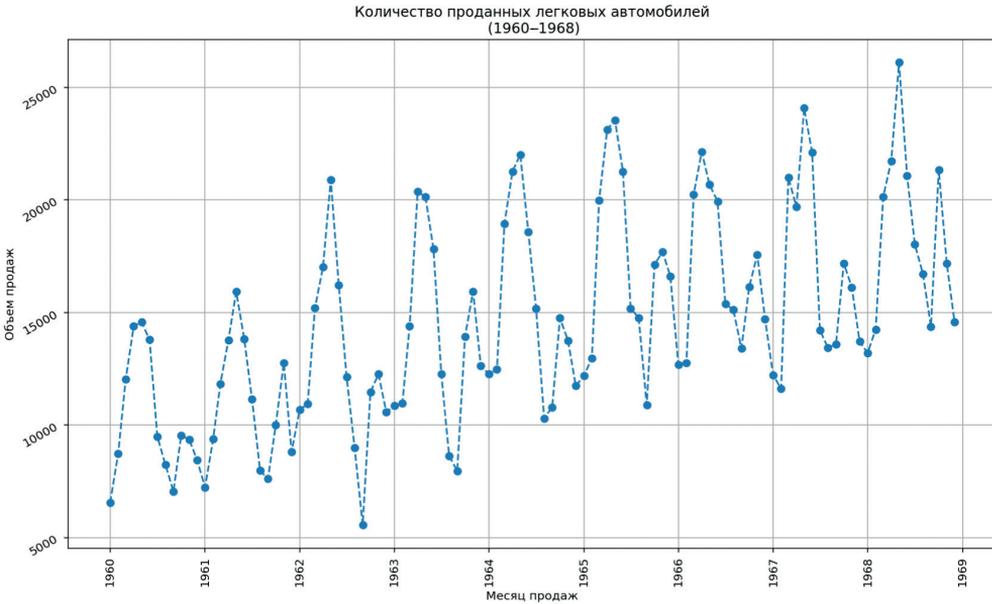
Ниже показано, какой элемент графика настраивается с помощью соответствующего метода класса Axes.



Большинство методов класса Axes есть в импортированном модуле `matplotlib.pyplot`. Например, модуль `matplotlib.pyplot` при вызове функции `plt.title()` вызывает метод `ax.set_title()`, при вызове функции `plt.grid()` – метод `ax.grid()`. Функции модуля `matplotlib.pyplot` удобны для быстрого построения несложных графиков, однако для построения сложных графиков применение методов класса Axes будет более приоритетным. Давайте визуализируем наш временной ряд с помощью функций модуля `matplotlib.pyplot`.

```
# задаем размер графика
plt.figure(figsize=(14, 8))
# задаем график, настроив тип маркера и стиль линии
plt.plot(cars, marker='o', linestyle='--')
# задаем заголовок графика
plt.title('Количество проданных легковых автомобилей\n' +
          '(1960-1968)')
# задаем заголовок оси x
plt.xlabel('Месяц продаж')
# задаем ориентацию меток по оси y
plt.yticks(rotation=30)
# задаем ориентацию меток по оси x
plt.xticks(rotation=90)
# задаем заголовок оси y
plt.ylabel('Объем продаж')
# задаем координатную сетку
plt.grid()

# выводим график
plt.show()
```



Давайте на основе произвольных данных построим круговую (секторную) диаграмму. Сначала мы создаем список меток для секторов круговой диаграммы.

```
# задаем список меток для секторов круговой диаграммы
labels = ['Python', 'C++', 'Ruby', 'Java']
```

Потом создаем список размеров секторов.

```
# задаем список размеров секторов
sizes = [215, 130, 245, 210]
```

Затем создаем список цветов секторов.

```
# задаем список цветов секторов
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']
```

Отделяем первый сектор.

```
# отделяем первый сектор
explode = (0.1, 0, 0, 0)
```

Теперь нам понадобится функция `plt.pie()`. Она имеет вид:

<code>plt.pie(x,</code>	←	Задаёт размеры секторов (объект, подобный массиву)
<code>explode=None,</code>	←	Определяет, какой сектор нужно отделить от остальных (объект, подобный массиву)
<code>labels=None,</code>	←	Задаёт последовательность строковых значений - меток для секторов (список)

<code>colors=None,</code>	←	Задаёт последовательность строковых значений – названий цветов для секторов (объект, подобный массиву)
<code>autopct=None,</code>	←	Позволяет задать вывод значений в процентном виде
<code>shadow=False,</code>		Задаёт отбрасывание тени (булево значение)
<code>startangle=None)</code>	←	Задаёт начальный угол в градусах

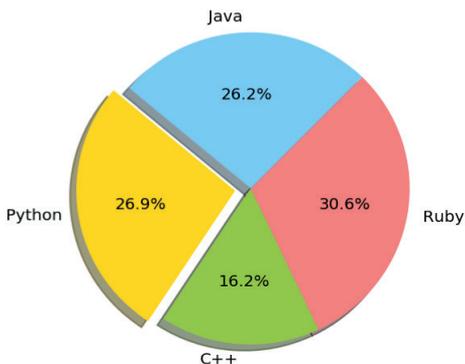
```
# строим круговую диаграмму
plt.pie(x=sizes, # передаем список размеров секторов
        explode=explode, # передаем информацию об отделении сектора
        labels=labels, # передаем список меток секторов
        colors=colors, # передаем список размеров секторов
        autopct='%1.1f%%', # выводим значения в виде процентов
        shadow=True, # используем тень
        startangle=140 # задаем начальный угол в градусах
    )
```

Мы передаем в функцию `plt.pie()` список размеров секторов, информацию об отделении первого сектора, список меток секторов, список размеров секторов, задаем вывод значений в виде процентов, включаем отбрасывание тени и задаем начальный угол в градусах.

Затем нам нужно воспользоваться функциями `plt.axis()` и `plt.show()`. Функция `plt.axis()` задает масштабирование осей, а функция `plt.show()` выводит график.

```
# задаем одинаковое масштабирование осей
# (делаем круги кругами)
plt.axis('equal')
```

```
# выводим круговую диаграмму
plt.show()
```

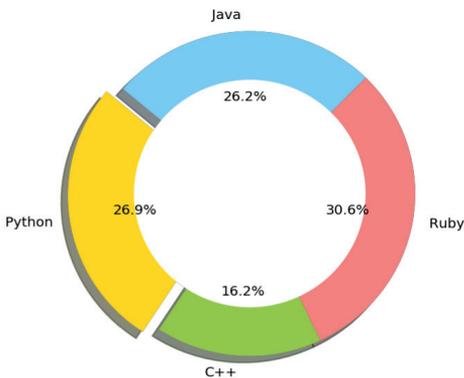


Теперь немного модифицируем круговую диаграмму, превратив ее в кольцевую. Для этого поместим внутри нашей круговой диаграммы белый круг. Нам понадобится метод `.gcf()`, возвращающий текущее изображение (`gcf` – это сокращение от `get current figure`), и метод `.gca()`, возвращающий текущие области рисования для текущего изображения (`gca` – это сокращение от `get current axes`), метод `.add_artist()`, добавляющий элемент рисунка (`Artist`) в рисунок. Функция `plt.tight_layout()` позволяет выполнить укладку, т.е. оптимально по-

позиционировать элементы графика на рисунке (например, чтобы метки оси не наезжали друг на друга и пр.).

```
# строим круговую диаграмму
plt.pie(x=sizes, # передаем список размеров секторов
        explode=explode, # передаем информацию об отделении сектора
        labels=labels, # передаем список меток секторов
        colors=colors, # передаем список размеров секторов
        autopct='%1.1f%%', # выводим значения в виде процентов
        shadow=True, # используем тень
        startangle=140 # задаем начальный угол в градусах
        )
# создаем белый круг
centre_circle = plt.Circle((0, 0), 0.70, fc='white')
# получаем текущий рисунок
fig = plt.gcf()
# получаем текущую область рисования для текущего рисунка
# и добавляем элемент - белый круг в наш текущий рисунок
# - круговую диаграмму
fig.gca().add_artist(centre_circle)
# задаем одинаковое масштабирование осей
# (делаем круги кругами)
plt.axis('equal')
# выполняем укладку
plt.tight_layout()

# выводим нашу кольцевую диаграмму
plt.show()
```



Давайте загрузим данные о кредитоспособности заемщиков и построим ряд диаграмм.

```
# прочитываем данные
data = pd.read_csv('Data/Visualizations.csv',
                  encoding='cp1251', sep=';') data.head()
```

	возраст	образование	стаж	проживание	доход	статус	дефолт
0	41	незак. высшее	17	12	176	постоянный	да
1	27	неполное среднее	10	6	31	постоянный	нет
2	40	среднее	15	14	55	постоянный	нет
3	41	среднее	15	14	120	постоянный	нет
4	24	среднее специальное	2	0	28	постоянный	да

Давайте построим гистограмму распределения клиентов по возрасту с помощью функции `plt.hist()`. Она имеет вид:

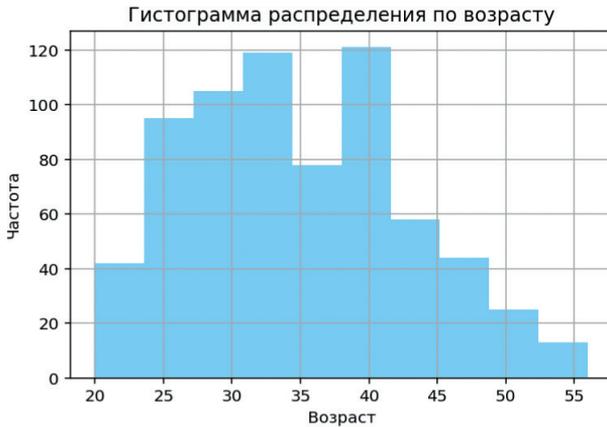
<code>plt.hist(x,</code>	←	Задаёт массив или последовательность массивов значений
<code>    bins=None,</code>	←	Задаёт количество интервалов (бинов). Увеличение количества интервалов детализирует гистограмму
<code>    orientation='vertical',</code>	←	Задаёт ориентацию гистограммы
<code>    color=None)</code>	←	Задаёт цвет гистограммы

Параметр `x` задаёт массив или последовательность массивов значений. Параметр `bins` задаёт количество интервалов (бинов). Увеличение количества интервалов детализирует гистограмму. Параметр `orientation` задаёт ориентацию гистограммы. Параметр `color` задаёт цвет гистограммы.

С помощью функции `plt.grid()` задаём координатную сетку. С помощью функции `plt.xlabel()` задаём заголовок оси *X*. С помощью функции `plt.ylabel()` задаём заголовок оси *Y*.

```
# строим гистограмму распределения по возрасту
plt.hist(x=data['возраст'], color='lightskyblue')
# задаем заголовок гистограммы
plt.title('Гистограмма распределения по возрасту')
# задаем координатную сетку
plt.grid()
# подписываем ось X
plt.xlabel('Возраст')
# подписываем ось Y
plt.ylabel('Частота')

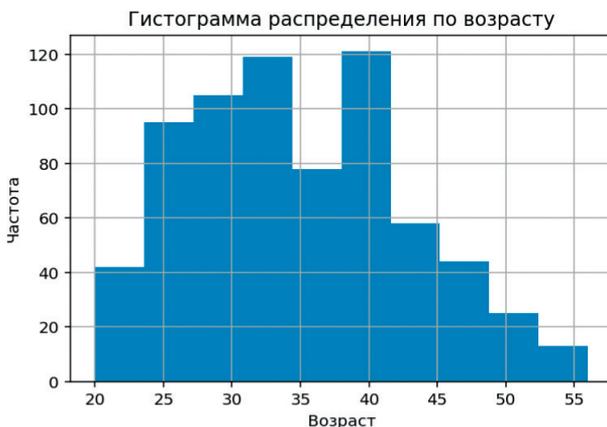
# выводим гистограмму
plt.show()
```



Функцию `plt.hist()` можно также вызвать с помощью метода `.hist()` библиотеки `pandas`.

```
# строим гистограмму распределения по возрасту
# с помощью метода .hist() библиотеки pandas
data['возраст'].hist()
# задаем заголовок гистограммы
plt.title('Гистограмма распределения по возрасту')
# подписываем ось X
plt.xlabel('Возраст')
# подписываем ось Y
plt.ylabel('Частота')

# выводим гистограмму
plt.show()
```



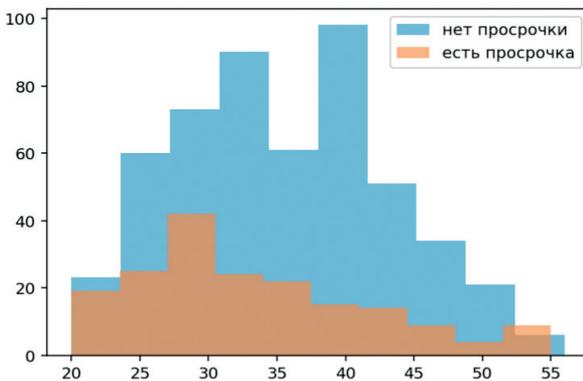
Чтобы наложить несколько гистограмм друг на друга в рамках одного и того же рисунка (и тем самым визуализировать разницу распределений), нужно несколько раз вызвать функцию `matplotlib.pyplot.hist()`. Сейчас мы сравним рас-

пределение клиентов без просрочки и распределение клиентов с просрочкой по возрасту. Параметр `alpha` регулирует прозрачность цвета гистограмм.

```
# получаем возраст для клиентов без просрочки
age_by_good = data[(data['дефолт'] == 'нет')]['возраст']
# получаем возраст для клиентов с просрочкой
age_by_bad = data[(data['дефолт'] == 'да')]['возраст']

# строим график с помощью plt.hist, 50 % прозрачности
plt.hist(age_by_good, alpha=0.5, label='нет просрочки')
# строим график с помощью plt.hist, 50 % прозрачности
plt.hist(age_by_bad, alpha=0.5, label='есть просрочка')
# размещаем легенду в верхнем правом углу
plt.legend(loc='upper right')

# выводим гистограммы
plt.show()
```



Обратите внимание, что местоположение легенды можно задать с помощью параметра `loc` функции `plt.legend()`. По умолчанию функция задает местоположение `'best'`, которое дает инструкции библиотеке `matplotlib` исследовать график и определить наилучшее местоположение легенды. Однако вы можете указать любой вариант местоположения из приведенных ниже, чтобы более точно разместить легенду (можно использовать либо строковое значение, либо числовой код).

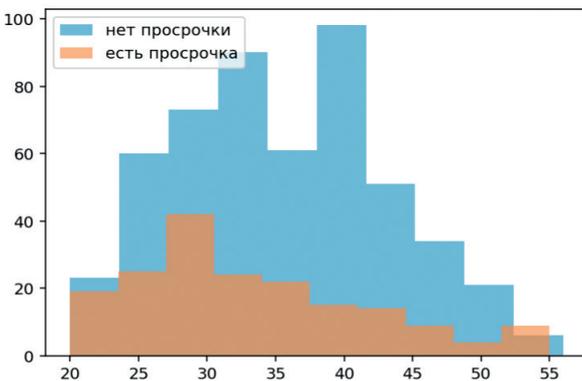
**Таблица 1** Значения для расположения легенды

Строковое значение	Код	Описание
<code>'best'</code>	0	Наилучшее расположение с учетом исследования графика
<code>'upper right'</code>	1	Расположение в верхнем правом углу
<code>'upper left'</code>	2	Расположение в верхнем левом углу
<code>'lower left'</code>	3	Расположение в нижнем левом углу
<code>'lower right'</code>	4	Расположение в нижнем правом углу

Строковое значение	Код	Описание
'right'	5	Горизонтальное выравнивание справа с вертикальным выравниванием по центру
'center left'	6	Вертикальное выравнивание по центру с горизонтальным выравниванием слева
'center right'	7	Вертикальное выравнивание по центру с горизонтальным выравниванием справа
'lower center'	8	Вертикальное выравнивание снизу с горизонтальным выравниванием по центру
'upper center'	9	Вертикальное выравнивание сверху с горизонтальным выравниванием по центру
'center'	10	Вертикальное выравнивание по центру и горизонтальное выравнивание по центру

```
# строим график с помощью plt.hist, 50 % прозрачности
plt.hist(age_by_good, alpha=0.5, label='нет просрочки')
# строим график с помощью plt.hist, 50 % прозрачности
plt.hist(age_by_bad, alpha=0.5, label='есть просрочка')
# разместили легенду в верхнем левом углу
plt.legend(loc='upper left')
```

```
# выводим гистограммы
plt.show()
```



Теперь вычислим частоты категорий переменной Образование и полученный результат запишем в словарь. Затем с помощью методов словаря `.keys()` и `.values()` сформируем список меток категорий и список частот категорий.

```
# вычислим частоты категорий переменной Образование
# и результат запишем в словарь
d = dict(data['образование'].value_counts())
d
```

```
{'среднее': 283,
'среднее специальное': 198,
```

```
'высшее': 163,
'неполное среднее': 34,
'незак. высшее': 22}
```

```
# формируем список меток категорий
```

```
ed_labels = [k for k in d.keys()]
ed_labels
```

```
dict_keys(['среднее', 'среднее специальное', 'высшее', 'неполное среднее', 'незак. высшее'])
```

```
# формируем список частот категорий
```

```
ed_counts = [v for v in d.values()]
ed_counts
```

```
dict_values([283, 198, 163, 34, 22])
```

Сейчас мы построим столбиковую диаграмму для частот категорий переменной Образование с помощью функции `plt.bar()`. Она имеет вид:

```
plt.bar(x,          ←   Задает столбик (скаляр или список)
        height,    ←   Задает высоту столбика (скаляр или список)
        width=0.8) ←   Задает ширину столбика (скаляр или список)
```

Параметр `x` задает последовательность меток (список). Параметр `height` задает последовательность скалярных значений (список). Параметр `width` задает ширину столбика (скаляр или объект, подобный списку).

```
# строим столбиковую диаграмму
```

```
plt.bar(x=ed_labels, height=ed_counts)
```

```
# подписываем ось X
```

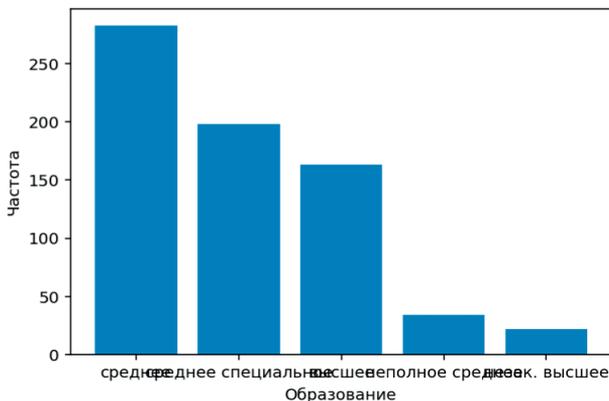
```
plt.xlabel('Образование')
```

```
# подписываем ось Y
```

```
plt.ylabel('Частота')
```

```
# выводим столбиковую диаграмму
```

```
plt.show()
```

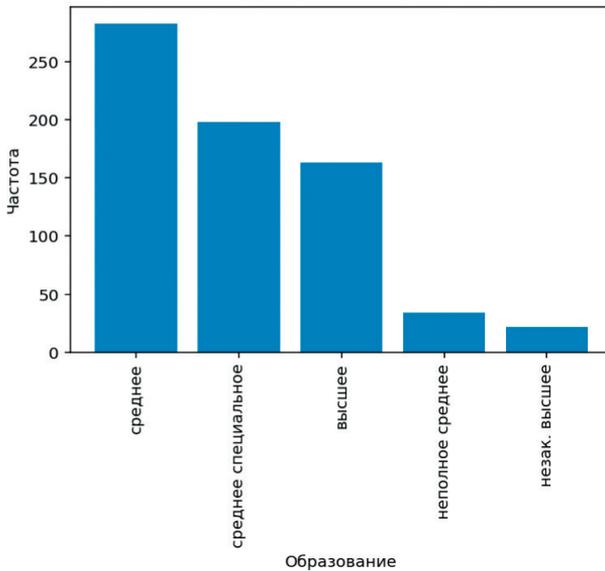


Недостаток построенной диаграммы очевиден: метки категорий наезжают друг на друга.

Изменим ориентацию меток оси  $X$  с помощью параметра `rotation` функции `plt.xticks()`.

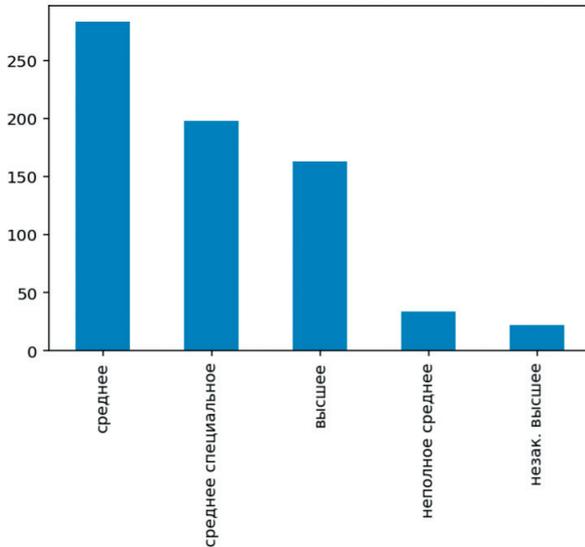
```
# строим столбиковую диаграмму
plt.bar(x=ed_labels, height=ed_counts)
# подписываем ось X
plt.xlabel('Образование')
# подписываем ось Y
plt.ylabel('Частота')
# изменяем ориентацию меток оси X
plt.xticks(rotation=90)

# выводим столбиковую диаграмму
plt.show()
```



Функцию `plt.hist()` можно также вызвать с помощью значения параметра `kind='bar'` метода `.plot()` библиотеки `pandas`.

```
# строим столбиковую переменную по переменной Образование
# с помощью значения параметра kind='bar'
# метода .plot() библиотеки pandas
ed_counts = data['образование'].value_counts()
my_bar_plot = ed_counts.plot(kind='bar')
```



А теперь поменяем абсолютные значения шкалы Y на проценты. Это можно сделать с помощью форматтеров, которые задают формат меток, выводимых под делениями оси (нужно воспользоваться модулем `ticker` библиотеки `matplotlib`). Чтобы нанести проценты над каждым столбиком, нам нужно извлечь фигуры с помощью `patches` (нетрудно догадаться, что в нашем случае это будут четыре прямоугольника). С помощью метода `.text()` мы задаем формат и расположение значений, выводимых над столбиками. Благодаря методу `.get_x()` мы можем сдвигать эти значения влево или вправо, а при помощи метода `.get_height()`, вычисляющего высоту столбика, мы можем располагать значения над столбиками (прибавляя какое-то число к результату метода `.get_height()`) или внутри столбиков (вычитая какое-то число из результата метода `.get_height()`). Параметр `ha` задает тип горизонтального выравнивания значений. Параметр `va` задает тип вертикального выравнивания значений.

```
# все то же самое, только используем процентную шкалу

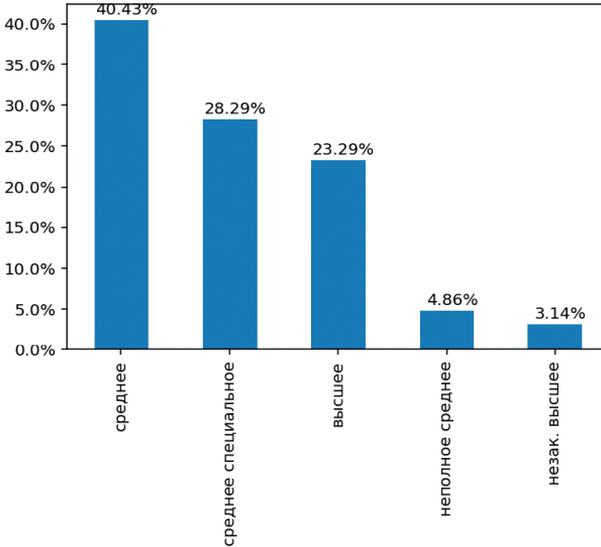
# получаем из частот проценты
ed_percents = (data['образование'].value_counts() / len(data)) * 100

# строим столбиковую диаграмму
my_bar_plot = ed_percents.plot(kind='bar')
# задаем процентную шкалу по оси Y
my_bar_plot.yaxis.set_major_formatter(ticker.PercentFormatter())

# над каждым столбиком размещаем процент

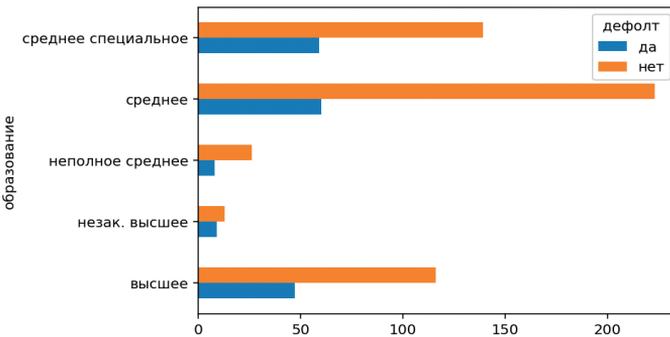
# извлекаем фигуры (patches) - прямоугольники (столбики)
rects = my_bar_plot.patches
# по каждому прямоугольнику (столбику)
for rect in rects:
```

```
# размещаем значение (get_x сдвигает значение влево или вправо,
# get_height сдвигает вверх или вниз)
my_bar_plot.text(rect.get_x() + 0.3,
rect.get_height() + 0.3,
str(round(rect.get_height(), 2)) + '%',
ha='center',
va='bottom')
```

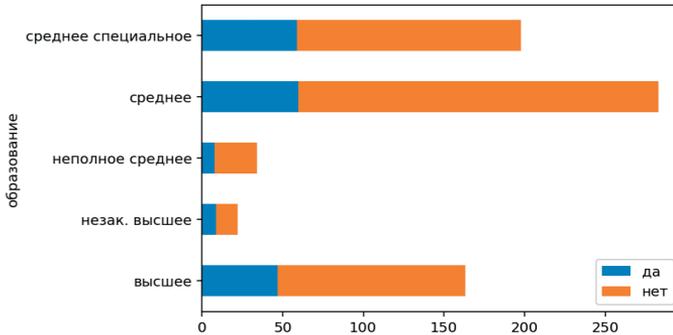


С помощью сочетания методов `.crosstab()` и `plot.barh()` библиотеки `pandas` можно построить кластеризованную и состыкованную столбиковые диаграммы.

```
# строим кластеризованную столбиковую диаграмму
pd.crosstab(data['образование'], data['дефолт']).plot.barh(stacked=False);
```



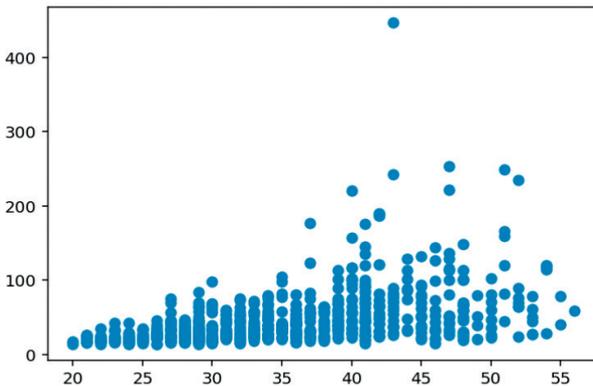
```
# строим состыкованную столбиковую диаграмму
pd.crosstab(data['образование'], data['дефолт']).plot.barh(stacked=True)
plt.legend(loc='lower right');
```



С помощью диаграммы рассеяния проанализируем взаимосвязь между возрастом и доходом. Диаграмму рассеяния можно построить с помощью функции `plt.scatter()`.

*# строим диаграмму рассеяния*

```
plot = plt.scatter(x=data['возраст'], y=data['доход'])
```



Убедились, что определенная взаимосвязь между возрастом и доходом существует.

## 1.2. SEABORN

Seaborn (<http://seaborn.pydata.org/introduction.html>, произносится как сиборн) – это библиотека для создания привлекательных и информативных статистических графиков в среде Python. Она надстраивается поверх библиотеки `matplotlib` и обеспечивает дополнительную функциональность, выходящую за пределы библиотеки `matplotlib`, а также по умолчанию демонстрирует более богатые и более современные способы визуализации, чем `matplotlib`. Установить библиотеку можно с помощью команды `pip install seaborn`.

Давайте импортируем библиотеку `seaborn`.

```
# импортируем библиотеку seaborn,
# предварительно установив библиотеку с помощью
# команды pip install seaborn
import seaborn as sns
```

Теперь запишем частоты и названия категорий переменной *образование*.

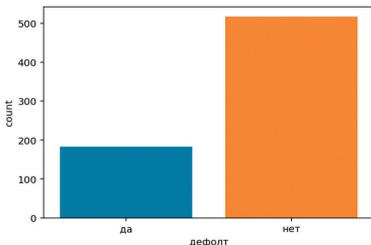
```
# вычислим частоты категорий переменной образование
ed_counts = data['образование'].value_counts()
# результат запишем в словарь
d = dict(ed_counts)
# формируем список меток категорий
ed_labels = [k for k in d.keys()]
```

Давайте построим обычную столбиковую диаграмму для переменной *дефолт*. Это можно сделать с помощью функции `seaborn.countplot()`. Она имеет вид:

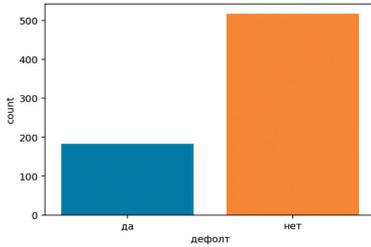
```
seaborn.countplot(x=None,
                  y=None,
                  hue=None,
                  data=None)
```

Параметры *x* и *y* задают переменную, для которой нужно вывести столбиковую диаграмму. Если задан один параметр *x*, будет построена вертикальная столбиковая диаграмма, если задан один параметр *y*, будет построена горизонтальная столбиковая диаграмма. Параметр *hue* позволяет проанализировать переменную, заданную с помощью *x* или *y*, по другой переменной (получаем в итоге кластеризованную столбиковую диаграмму). Параметр *data* задает имя набора данных (датафрейм, массив, список массивов). Имя набора данных для параметра *data* можно не задавать, если для параметров *x* и *y* имя переменной задано в формате `имя_набора[ 'имя_переменной' ]`. Если для параметров *x* и *y* имя переменной задано в формате `'имя_переменной'`, для параметра *data* необходимо задать имя набора.

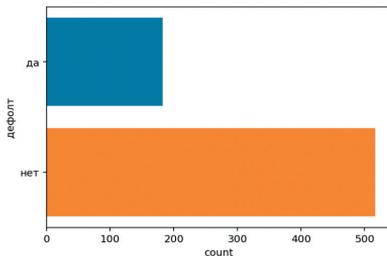
```
# строим обычную вертикальную столбиковую
# диаграмму для переменной дефолт
plot = sns.countplot(x=data[ 'дефолт' ])
```



```
# а можно еще так
plot = sns.countplot(x='дефолт', data=data)
```

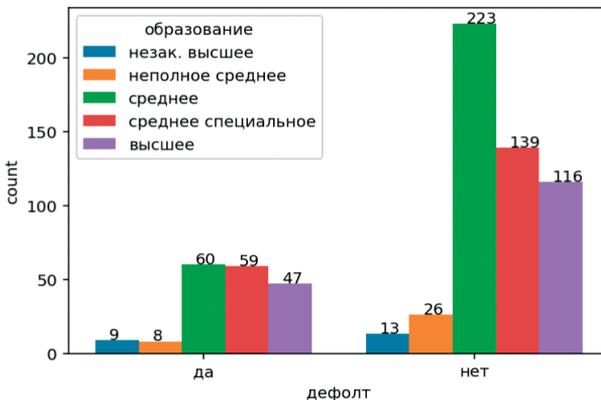


```
# строим обычную горизонтальную столбиковую
# диаграмму для переменной дефолт
plot = sns.countplot(y=data['дефолт'])
```



Можно построить кластеризованную столбиковую диаграмму: посмотреть уровень образования среди «хороших» и «плохих» заемщиков. Для вывода значений над столбиками применим метод `.annotate()`. Параметр `text` задает формат значения, обозначающего высоту столбика (ее получаем с помощью метода `.get_height()`), параметр `xy` задает расположение значения.

```
# строим кластеризованную столбиковую диаграмму
plot = sns.countplot(x=data['дефолт'],
                    hue=data['образование'])
for p in plot.patches:
    plot.annotate(text='{:.0f}'.format(p.get_height()),
                 xy=(p.get_x() + 0.05, p.get_height() + 0.5))
```



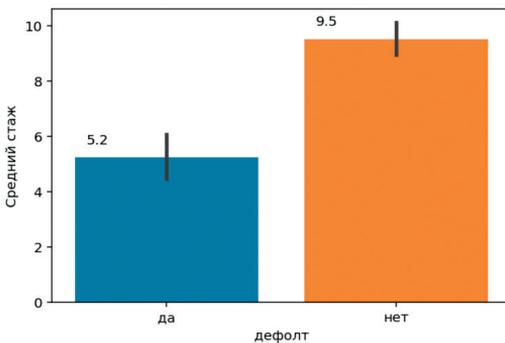
Столбиковую диаграмму еще можно построить с помощью функции `seaborn.barplot()`. Она имеет вид:

```
seaborn.countplot(x=None,
                  y=None,
                  hue=None,
                  data=None,
                  estimator=<function mean>)
```

Она интересна своим параметром `estimator`. Параметр `estimator` задает статистическую функцию для вычисления значения для каждого бина (по умолчанию используется функция, вычисляющая среднее значение). В отличие от функции `seaborn.countplot()`, должны быть заданы оба параметра, `x` и `y`.

Сейчас мы построим столбиковую диаграмму, где столбики будут показывать средний стаж работы для клиентов без просрочки и клиентов с просрочкой.

```
# строим столбиковую диаграмму, смотрим средний
# стаж у клиентов без/с просрочкой
plot = sns.barplot(x=data['дефолт'],
                  y=data['стаж'])
# подписываем ось Y
plot.set(ylabel='Средний стаж')
# выводим значения над столбиками
for p in plot.patches:
    plot.annotate(text='{:.1f}'.format(p.get_height()),
                 xy=(p.get_x() + 0.05, p.get_height() + 0.5))
```

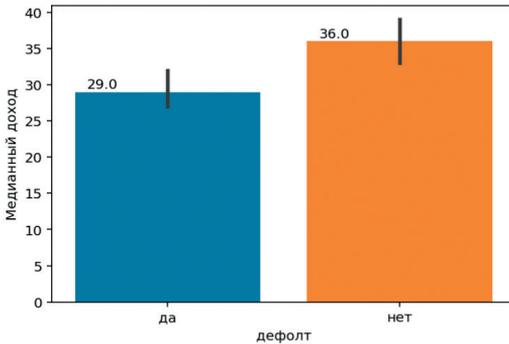


Видим, что у клиентов с просрочкой средний стаж работы ниже.

Сейчас мы построим столбиковую диаграмму, где столбики будут показывать медианный доход для клиентов без просрочки и клиентов с просрочкой.

```
# строим столбиковую диаграмму, смотрим медианный
# доход у клиентов без/с просрочкой
plot = sns.barplot(x=data['дефолт'],
                  y=data['доход'],
                  estimator=lambda x: np.median(x))
# подписываем ось Y
plot.set(ylabel='Медианный доход')
```

```
# выводим значения над столбиками
for p in plot.patches:
    plot.annotate(text='{:.1f}'.format(p.get_height()),
                  xy=(p.get_x() + 0.05, p.get_height() + 0.5))
```



Видим, что у клиентов с просрочкой медианный доход ниже.

С помощи функции `seaborn.heatmap()` можно построить теплокарту. Теплокарта часто используется для вывода корреляционного анализа или результатов поиска по сетке. Функция `seaborn.heatmap()` имеет вид:

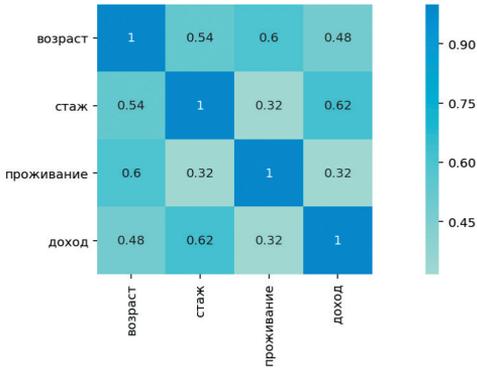
```
seaborn.heatmap(data,
                 vmin=None,
                 vmax=None,
                 cmap=None,
                 center=None,
                 annot=None,
                 fmt='%.2g')
```

Параметр `data` задает двумерный массив данных (датафрейм `pandas` или массив `NumPy`). Значения `vmin` и `vmax` определяют диапазон данных, который будет покрыт цветовой картой (`colormap`). По умолчанию цветовая карта охватывает весь диапазон значений отображаемых данных. Параметр `cmap` задает цветовую карту. Параметр `center` задает значение, по которому происходит центрирование цветовой карты при визуализации отклоняющихся значений. Параметр `annot` задает вывод значений в каждой ячейке. Параметр `fmt` задает формат выводимых значений.

Давайте построим корреляционную матрицу в виде теплокарты для нашего примера.

```
# выведем корреляционную матрицу
plot = sns.heatmap(data=data.corr(), annot=True, center=2)
plt.xticks(rotation=90)
plt.yticks(rotation=0)
plt.axis('equal')

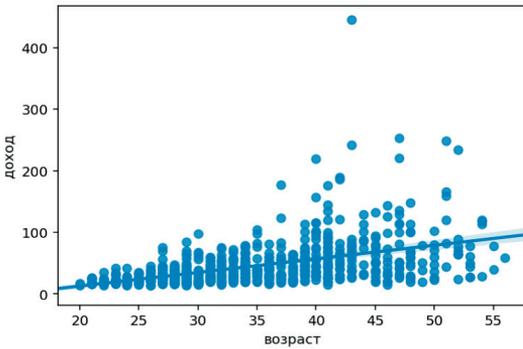
plt.show()
```



С помощи функции `seaborn.regplot()` можно построить диаграмму рассеяния.

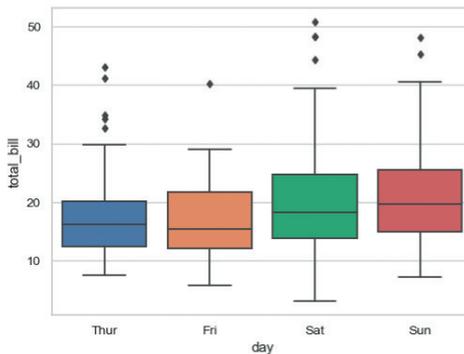
*# строим диаграмму рассеяния*

```
plot = sns.regplot(x=data['возраст'], y=data['доход'])
```



Видим определенную связь между доходом и возрастом.

Один из видов полезных диаграмм – ящичковые (ящичные) диаграммы. Ящичковую диаграмму еще называют ящиком с усами.



**Рис. 2** Ящичковые (ящичные) диаграммы

Ящичковые диаграммы полезны как раз тем, что позволяют визуализировать и сравнивать распределение и основную тенденцию числовых значений посредством их квартилей. Квартиль – это статистика, с помощью которой мы разбиваем упорядоченный числовой ряд на 4 равнонаполненные части таким образом, что 25 % единиц совокупности будут меньше по величине  $Q_1$  (первого квартиля); 25 % будут заключены между  $Q_1$  и  $Q_2$  (между первым и вторым квартилями); 25 % – между  $Q_2$  и  $Q_3$  (между вторым и третьим квартилями); остальные 25 % превосходят  $Q_3$  (третий квартиль). Первый квартиль – это 0,25-квантиль, второй квартиль – это 0,50-квантиль, третий квартиль – это 0,75-квантиль.

Границами ящика служат первый и третий квартили, линия в середине ящика – медиана (средний квартиль). Концы усов – края статистически значимой выборки (без выбросов), и они могут определяться несколькими способами. Наиболее распространенные значения, определяющие длину «усов»:

- минимальное и максимальное наблюдаемые значения данных по выборке (в этом случае выбросы отсутствуют);
- разность первого квартиля и полутора межквартильных расстояний; сумма третьего квартиля и полутора межквартильных расстояний. В общем виде эта формула имеет вид  $X_1 = Q_1 - k(Q_3 - Q_1)$  и  $X_2 = Q_3 + k(Q_3 - Q_1)$ , где:  $X_1$  – нижняя граница уса,  $X_2$  – верхняя граница уса,  $Q_1$  – первый квартиль,  $Q_3$  – третий квартиль,  $k$  – коэффициент, наиболее часто употребляемое значение которого равно 1,5.

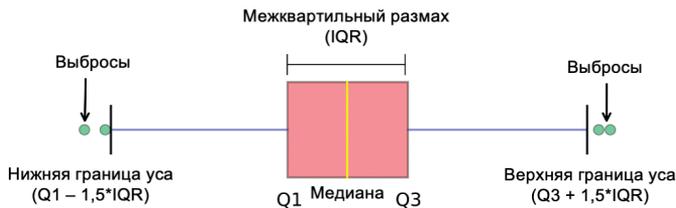
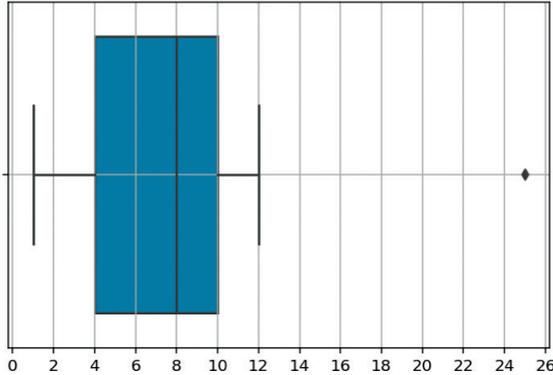


Рис. 3 Структура ящичковой диаграммы

Допустим, у нас есть уже упорядоченный числовой ряд: 1, 1, 4, 4, 5, 8, 8, 9, 10, 10, 12, 25. Построим для него ящичковую диаграмму. Предварительно зададим координатную сетку, изменим цену основных делений оси  $X$  с помощью локатора `MultipleLocator` и метода `ax.xaxis.set_major_locator()`.

```
# создаем серию из числового ряда
s = pd.Series([1, 1, 4, 4, 5, 8, 8, 9, 10, 10, 12, 25])
# строим ящичковую диаграмму
plot = sns.boxplot(x=s)
# накладываем координатную сетку
plt.grid()
# задаем цену деления оси X
plot.xaxis.set_major_locator(ticker.MultipleLocator(2))
```



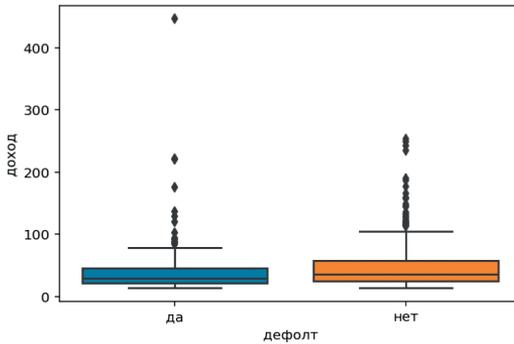
Вычисляем первый квантиль, третий квантиль и межквантильный размах. Первый квантиль равен 4, третий квантиль равен 10, межквантильный размах равен  $10 - 4 = 6$ , что мы и видим, анализируя границы ящика. Теперь разбираемся с усами. Верхняя граница уса равна  $10 + 1,5 * 6 = 19$ , однако максимальное значение 25 больше этого значения, поэтому оно считается выбросом. Наибольшее значение, которое не превышает значение 25, будет 12, значит, верхняя граница уса будет простирается до 12. Нижняя граница уса равна  $4 - 1,5 * 6 = -5$ , минимальное значение 0 меньше этого значения, поэтому слева у нас выбросов нет. Наименьшим значением перед значением  $-5$  будет 1, значит, нижняя граница уса будет простирается до 1.

```
# вычисляем первый и третий квантили
q1 = s.quantile(0.25)
q3 = s.quantile(0.75)
# печатаем результаты
print("первый квантиль", q1)
print("третий квантиль", q3)
print("межквантильный размах", q3 - q1)
print("нижнее значение", q1 - 1.5 * (q3 - q1))
print("верхнее значение", q3 + 1.5 * (q3 - q1))
print("минимум", s.min())
print("максимум", s.max())
```

```
первый квантиль 4.0
третий квантиль 10.0
межквантильный размах 6.0
нижнее значение -5.0
верхнее значение 19.0
минимум 1
максимум 25
```

Давайте построим ящичковые диаграммы для возраста заемщиков с просрочкой и возраста заемщиков без просрочки.

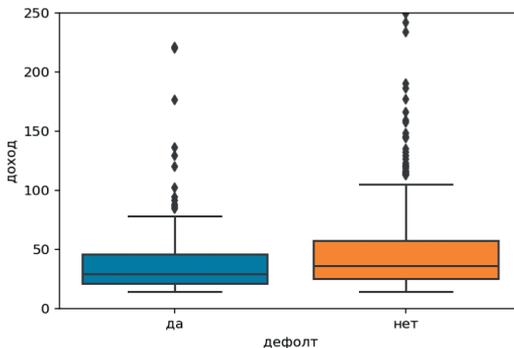
```
# строим ящичковую диаграмму
plot = sns.boxplot(x='дефолт', y='доход', data=data)
```



С помощью функций `plt.xlim()` и `plt.ylim()` можно изменить нижнее и верхнее значения (пределы) соответствующей оси.

```
# строим ящичковую диаграмму
plot = sns.boxplot(x='дефолт', y='доход', data=data)
# задаем пределы значений оси Y
plt.ylim([0, 250])

plt.show()
```



## 1.3. PLOTLY

Plotly (произносится как плотли) – это интерактивная библиотека для Python, которая поддерживает 40 типов диаграмм. Построенная поверх JavaScript-библиотеки Plotly, `plotly` позволяет пользователям Python создавать красивые интерактивные веб-визуализации, которые можно размещать в тетрадках Jupyter, сохранять в автономных HTML-файлах или интегрировать в питоновские веб-приложения с помощью Dash.

Благодаря глубокой интеграции с утилитой экспорта изображений `orca` `plotly` также поддерживает не только веб-приложения, но и интегрированные среды разработки и интерфейсы командной строки (например, QtConsole, Spyder, VSCode, PyCharm), а еще и публикацию статических документов (например, экспорт тетрадков в PDF с поддержкой высококачественных вектор-

ных изображений). Установить библиотеку можно с помощью команды `pip install plotly`.

Для построения графиков нам потребуются два основных модуля:

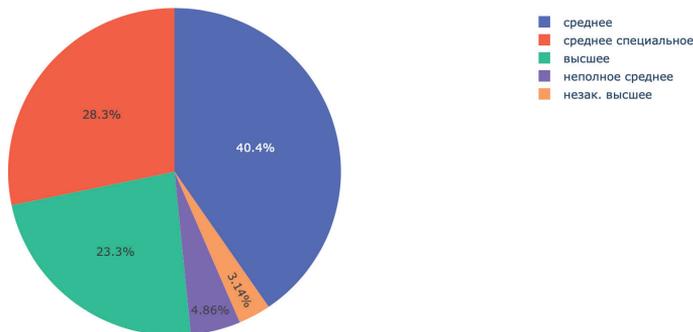
- `plotly.graph_objs` – содержит функции, которые будут строить графические объекты;
- `plotly.express` – модуль, задача которого – упростить и ускорить создание графиков на основе подготовленных данных с помощью однострочных команд на Python (`plotly.express` по отношению к `plotly` является тем же, чем `seaborn` по отношению к `matplotlib`).

Для создания графика в `plotly` используются такие понятия, как `Trace`, `Layout` и `Figure`. `Trace` – это тип визуализации какого-то одного распределения данных (круговая диаграмма, диаграмма рассеяния) и собственно сами данные. `Layout` задает элементы визуализации, которые не связаны с данными. Например, мы можем изменить заголовок, заголовки осей, диапазон, шрифт. `Figure` – итоговый рисунок, результат связывания `Trace` и `Layout` воедино.

Давайте построим круговую диаграмму для переменной Образование.

```
# импортируем модуль graph_objects библиотеки plotly,
# предварительно установив библиотеку с помощью
# команды pip install plotly
import plotly.graph_objects as go
# задаем данные для круговой диаграммы
trace = go.Pie(labels=ed_labels, values=ed_counts)
# задаем заголовок
ttl = 'Распределение заемщиков по уровню образования'
layout = go.Layout(title=ttl)
# строим диаграмму
fig = go.Figure(data=[trace], layout=layout)
fig
```

Распределение клиентов по уровню образования

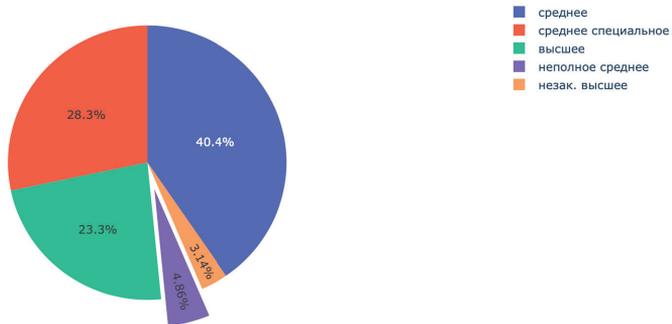


Теперь отсоединим один из сегментов, продемонстрировав более лаконичный способ построения круговой диаграммы. С помощью параметра `pull` мы как бы «вытягиваем» один из сегментов.

*# отсоединяем один из сегментов*

```
fig = go.Figure(data=[go.Pie(
    labels=ed_labels,
    values=ed_counts,
    pull=[0, 0, 0, 0.2, 0]),
    layout_title_text=ttl)
fig.show()
```

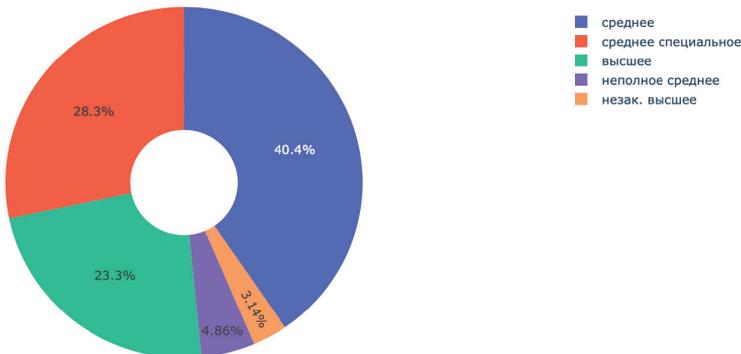
Распределение заемщиков по уровню образования



Теперь круговую диаграмму превратим в кольцевую (пончиковобразную). С помощью параметра `hole` задаем размер «дырки» нашего «пончика».

*# превращаем нашу круговую диаграмму в кольцевую*

```
fig = go.Figure(data=[go.Pie(labels=ed_labels,
    values=ed_counts,
    hole=.3)])
fig.show()
```



Сейчас мы проанализируем распределение по образованию среди клиентов без просрочки и клиентов с просрочкой, воспользовавшись функцией `make_subplots()` библиотеки `plotly`.

Функция `make_subplots()` создает изображение, предварительно сконфигурированное из сетки подграфиков (можно задавать количество строк и столбцов), для которых можно задавать объекты `Trace` (данные для каждого подграфика).

Параметр `specs` функции `make_subplots()` используется для настройки спецификаций каждого подграфика. Каждая спецификация объекта `Trace` имеет специальный ключ `type`, который задает тип визуализации (например, `'bar'`, `'scatter'`, `'counter'`, для круговых диаграмм нужно использовать значение `'domain'`). Значением `specs` должен быть двумерный список с размерностями, соответствующими тем, которые заданы в качестве `row` и `cols`. Элементами `specs` может быть либо значение `None`, которое указывает на то, что не следует инициализировать подграфик, начиная с этой ячейки сетки, либо словарь, содержащий спецификации подграфика. Объекты `Trace` могут добавляться на рисунок с помощью метода `.add_trace()`. Метод `.update_layout()` позволяет изменить элементы визуализации, заданные по умолчанию.

```
# записываем словарь с распределением частот категорий
# переменной Образование по заемщикам с просрочкой
ed_bads = dict(data.loc[data['дефолт'] == 'да'].groupby(
    ['образование']).size())
# записываем словарь с распределением частот категорий
# переменной Образование по заемщикам без просрочки
ed_goods = dict(data.loc[data['дефолт'] == 'нет'].groupby(
    ['образование']).size())
# печатаем словари
print(ed_bads)
print(ed_goods)

{'высшее': 47, 'незак. высшее': 9, 'неполное среднее': 8, 'среднее': 60, 'среднее
специальное': 59}
{'высшее': 116, 'незак. высшее': 13, 'неполное среднее': 26, 'среднее': 223, 'среднее
специальное': 139}

# извлекаем список меток
labels = [k for k in ed_bads.keys()]
labels

['высшее',
 'незак. высшее',
 'неполное среднее',
 'среднее',
 'среднее специальное']

# извлекаем список значений для заемщиков с просрочкой
ed_bads_values = [k for k in ed_bads.values()]
ed_bads_values

[47, 9, 8, 60, 59]

# извлекаем список значений для заемщиков без просрочки
ed_goods_values = [k for k in ed_goods.values()]
ed_goods_values

[116, 13, 26, 223, 139]

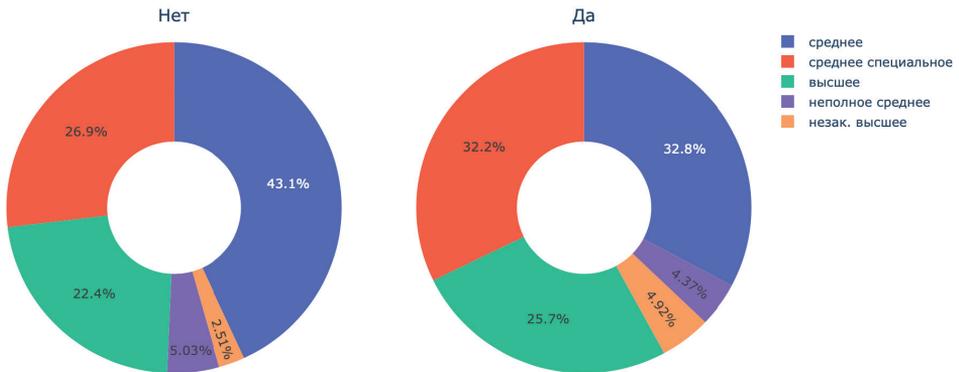
# импортируем функцию make_subplots()
from plotly.subplots import make_subplots
```

```

# определяем формат подграфиков
fig = make_subplots(
    rows=1, cols=2,
    specs=[[{'type': 'domain'}, {'type': 'domain'}]],
    subplot_titles=['Нет', 'Да'])
# добавляем первый подграфик
fig.add_trace(go.Pie(labels=labels,
                    values=ed_goods_values,
                    name='Нет'),
             row=1, col=1)
# добавляем второй подграфик
fig.add_trace(go.Pie(labels=labels,
                    values=ed_bads_values,
                    name='Да'),
             row=1, col=2)
# добавляем отверстие
fig.update_traces(hole=.4)
# добавляем заголовок
fig.update_layout(title_text='дефолт')

```

дефолт



## Задачи с собеседования (теория вероятности)

1. Светофор на пешеходном переходе одну минуту разрешает переходить улицу, а две минуты – запрещает. Найдите среднее время ожидания зеленого света пешеходом, который подошел к перекрестку.

2. Из 15 билетов выигрышными являются 4. Какова вероятность того, что среди взятых наудачу 6 билетов будут 2 выигрышных?

3. Номера машин в России имеют следующий код: вначале идет буква, затем трехзначный номер, потом еще две буквы и далее двузначное число – код региона. Сколько всего можно составить различных номеров машин?

# 2. БИБЛИОТЕКА ПРОГНОЗИРОВАНИЯ ВРЕМЕННЫХ РЯДОВ ETNA

## 2.1. ОБЩЕЕ ЗНАКОМСТВО

ETNA – сравнительно молодая библиотека для прогнозирования временных рядов от команды Тинькофф Банк.

Мы будем работать с etna 1.12 (команда для установки `pip install etna==1.12`). Программный код, приведенный в книге, вы найдете в тетрадке *Часть 3 2.1.-2.22. Общее знакомство с библиотекой ETNA (версия 1.12).ipynb* и скрипте *etna\_utils.py*. Если вы захотите воспользоваться версией 1.14 (последней на момент написания книги), смотрите тетрадку *Часть 3 2.1.-2.22. Общее знакомство с библиотекой ETNA (версия 1.14).ipynb*. Давайте импортируем необходимые библиотеки, классы и функции, настроим визуализацию.

Первым делом импортируем библиотеки pandas, NumPy, torch модули re, datetime и random.

```
# импортируем библиотеки pandas, numpy, torch,  
# модули re, datetime и random  
import pandas as pd  
import numpy as np  
import torch  
import datetime  
import random
```

Теперь по порядку импортируем функции и классы библиотеки ETNA. Функция `generate_const_df()` генерирует искусственные датафреймы, на которых часто бывает удобно проиллюстрировать функционал библиотеки. Класс `TSDataset` необходим для преобразования датафреймов в формат etna.

```
# импортируем функцию generate_const_df() для  
# генерирования искусственных датафреймов  
from etna.datasets import generate_const_df  
# импортируем класс TSDataset для преобразования  
# датафреймов в формат ETNA  
from etna.datasets.tsdataset import TSDataset
```

Теперь импортируем классы библиотеки ETNA, в которых реализованы модели прогнозирования временных рядов. Как правило, это будут классы, являющиеся обертками над уже существующими классами библиотек scikit-learn, statsmodels, prophet, pytorch-forecasting, catboost и др. Например, с помощью класса `ProphetModel` мы можем построить модель `Prophet`, с помощью классов `ElasticPerSegmentModel` и `ElasticMultiSegmentModel` – модель линейной регрессии с регуляризацией «эластичная сеть», реализованную в библиотеке scikit-learn в виде класса `ElasticNet`, с помощью классов `CatBoostModelPerSegment` и `CatBoostModelMultiSegment` – модель градиентного бустинга `CatBoost`, решающую задачу регрессии и реализованную в библиотеке `catboost` в виде класса `CatBoostRegressor`. `PerSegment` обозначает, что для каждого сегмента – временного ряда будет построена отдельная модель. `MultiSegment` обозначает, что для всех сегментов – временных рядов будет построена одна модель.

Класс DeepAR строит модель авторегрессионной рекуррентной нейронной сети с LSTM- или GRU-ячейками из библиотеки `pytorch-forecasting`. Кроме того, для построения моделей нейронных сетей можно использовать класс TFT – трансформер из библиотеки `pytorch-forecasting` и класс `RNNModel` – рекуррентную нейронную сеть с LSTM-ячейками от разработчиков.

```
# импортируем модели наивного прогноза, скользящего
# среднего, сезонного скользящего среднего, SARIMAX,
# тройного экспоненциального сглаживания (Холта-Винтерса)
from etna.models import (NaiveModel,
                          MovingAverageModel,
                          SeasonalMovingAverageModel,
                          SARIMAXModel,
                          HoltWintersModel)

# импортируем класс ProphetModel для
# построения модели ProphetModel
from etna.models import ProphetModel
# импортируем классы ElasticPerSegmentModel
# и ElasticMultiSegmentModel для построения
# модели ElasticNet
from etna.models import (ElasticPerSegmentModel,
                          ElasticMultiSegmentModel)

# импортируем классы CatBoostModelPerSegment
# и CatBoostModelMultiSegment
from etna.models import (CatBoostModelPerSegment,
                          CatBoostModelMultiSegment)

# импортируем модель нейронной сети
from etna.models.nn import DeepARModel
```

Классы библиотеки ETNA `VotingEnsemble` и `StackingEnsemble` позволяют создавать ансамбли моделей и стекинг моделей.

```
# импортируем класс VotingEnsemble для построения ансамблей
from etna.ensembles import VotingEnsemble
# импортируем класс StackingEnsemble для построения стекинга
from etna.ensembles import StackingEnsemble
```

Базовые классы библиотеки ETNA `BaseAdapter`, `PerSegmentModel` и `MultiSegmentModel` позволяют использовать для прогнозирования временных рядов модели, решающие задачу регрессии, которые не относятся к библиотеке `scikit-learn` и еще не реализованы в библиотеке ETNA. Например, с помощью этих классов вы можете, помимо модели `CatBoost`, реализовать модель `LightGBM` для прогнозирования тысяч временных рядов.

```
# импортируем базовые классы BaseAdapter, PerSegmentModel и
# MultiSegmentModel для построения не sklearn-моделей в ETNA
from etna.models.base import (BaseAdapter,
                              PerSegmentModel,
                              MultiSegmentModel)
```

Базовые классы библиотеки ETNA `SklearnPerSegmentModel` и `SklearnMultiSegmentModel` позволяют использовать для прогнозирования временных рядов модели `scikit-learn`, поддерживающие интерфейс `scikit-learn`, которые решают

задачу регрессии и еще не реализованы в библиотеке ETNA. Например, с помощью этих классов вы можете, помимо модели линейной регрессии с регуляризацией «эластичной сети» (класс `ElasticNet`), реализовать модель случайного леса (класс `RandomForestRegressor`) или модель многослойного перцептрона (класс `MLPRegressor`) для прогнозирования тысяч временных рядов.

```
# импортируем базовые классы SklearnPerSegmentModel,
# SklearnMultiSegmentModel для построения sklearn-
# моделей в ETNA
from etna.models.sklearn import (SklearnPerSegmentModel,
                                 SklearnMultiSegmentModel)
```

Класс `Pipeline` предназначен для последовательного выполнения цепочки преобразований, конструирования признаков и получения прогнозов для каждого сегмента.

```
# импортируем класс Pipeline для последовательного выполнения
# цепочки преобразований и/или конструирования признаков
from etna.pipeline import Pipeline
```

Метрики качества в ETNA реализованы в виде классов.

```
# импортируем классы для вычисления метрик
from etna.metrics import MAE, MSE, SMAPE, MAPE
```

Библиотека ETNA предлагает ряд функций для выполнения различных визуализаций.

<code>plot_forecast()</code>	визуализирует прогнозы
<code>plot_backtest()</code>	визуализирует результаты перекрестной проверки
<code>plot_periodogram()</code>	строит периодограмму для подбора периода компонент ряда Фурье с целью моделирования сезонностей
<code>plot_anomalies()</code>	визуализирует выбросы
<code>plot_anomalies_interactive()</code>	выполняет интерактивную визуализацию выбросов
<code>plot_imputation()</code>	визуализирует пропуски
<code>plot_trend()</code>	визуализирует тренд
<code>plot_feature_relevance()</code> работает в паре с классом <code>ModelRelevanceTable</code>	визуализирует важности признаков

```
# импортируем функцию plot_forecast() для визуализации прогнозов
from etna.analysis import plot_forecast
# импортируем функцию plot_backtest() для визуализации
# результатов перекрестной проверки
```

```

from etna.analysis.plotters import plot_backtest
# импортируем функцию plot_periodogram()
# для построения периодограммы
from etna.analysis import plot_periodogram
# импортируем функцию визуализации выбросов
from etna.analysis import plot_anomalies
# импортируем функцию import plot_anomalies_interactive()
# для интерактивной визуализации выбросов
from etna.analysis import plot_anomalies_interactive
# импортируем функцию plot_imputation()
# для визуализации импутации пропусков
from etna.analysis import plot_imputation
# импортируем функцию визуализации тренда
from etna.analysis import plot_trend
# для визуализации графика важностей признаков
# импортируем функцию plot_feature_relevance()
from etna.analysis import plot_feature_relevance
# импортируем класс ModelRelevanceTable
# для вычисления важностей признаков
from etna.analysis import ModelRelevanceTable

```

Теперь импортируем классы-трансформеры для генерации признаков и преобразований:

- PytorchForecastingTransform – для выполнения преобразований перед применением моделей из библиотеки PytorchForecasting;
- MeanTransform – для генерации скользящих средних;
- LagTransform – для генерации лагов;
- MeanSegmentEncoderTransform – для кодирования сегментов расширяющимся средним значением зависимой переменной;
- SegmentEncoderTransform – для кодирования меток сегментов целочисленными значениями в лексикографическом порядке (LabelEncoding): сегменты a, b, c, d получают значения 0, 1, 2, 3;
- HolidayTransform – для выделения национальных праздников;
- DateFlagsTransform – для создания календарных признаков;
- TrendTransform – для добавления тренда в качестве признака;
- FourierTransform – для генерации членов Фурье;
- LogTransform – для логарифмирования и экспоненцирования зависимой переменной;
- LinearTrendTransform – для предсказания, удаления и восстановления линейного тренда;
- BinsegTrendTransform – для предсказания, удаления и восстановления кусочно-линейного тренда;
- TheilSenTrendTransform – для предсказания, удаления и восстановления тренда, полученного с помощью оценочной функции Тейла–Сена;
- MedianOutliersTransform – для замены выбросов, обнаруженных в соответствии с медианным методом, на значения NaN;
- DensityOutliersTransform – для замены выбросов, обнаруженных в соответствии с методом на основе плотности, на значения NaN;
- TimeSeriesImputerTransform – для импутации значений NaN в соответствии с выбранной стратегией;

- `StandardScalerTransform`, `MinMaxScalerTransform`, `RobustScalerTransform` и `MaxAbsScalerTransform` – для стандартизации признаков (нужно для линейных моделей).

```
# импортируем классы для генерации признаков и преобразований:
# PytorchForecastingTransform - для выполнения преобразований
# перед применением моделей из библиотеки PytorchForecasting,
# MeanTransform - для генерации скользящих средних,
# LagTransform - для генерации лагов,
# MeanSegmentEncoderTransform - для кодирования сегментов
# расширяющимся средним значением зависимой переменной,
# SegmentEncoderTransform - для кодирования меток сегментов
# целочисленными значениями в лексикографическом порядке,
# (LabelEncoding): сегменты a, b, c, d получают значения 0, 1, 2, 3,
# HolidayTransform - для выделения национальных праздников,
# DateFlagsTransform - для создания календарных признаков
# TrendTransform - для добавления тренда в качестве признака,
# FourierTransform - для генерации членов Фурье,
# LogTransform - для логарифмирования и экспоненцирования зависимой переменной,
# LinearTrendTransform - для предсказания,
# удаления и восстановления линейного тренда,
# BinsegTrendTransform - для предсказания,
# удаления и восстановления кусочно-линейного тренда,
# TheilSenTrendTransform - для предсказания, удаления
# и восстановления тренда, полученного с помощью
# оценочной функции Тейла-Сена,
# MedianOutliersTransform - для замены выбросов, обнаруженных
# в соответствии с медианным методом, на значения NaN,
# DensityOutliersTransform - для замены выбросов, обнаруженных
# в соответствии с методом на основе плотности, на значения NaN,
# TimeSeriesImputerTransform - для импутации значений NaN
# в соответствии с выбранной стратегией,
# StandardScalerTransform, MinMaxScalerTransform, RobustScalerTransform
# и MaxAbsScalerTransform - для стандартизации признаков
# (нужно для линейных моделей)
from etna.transforms import (
    PytorchForecastingTransform, MeanTransform,
    LagTransform, MeanSegmentEncoderTransform,
    SegmentEncoderTransform, HolidayTransform,
    DateFlagsTransform, TrendTransform, FourierTransform,
    LogTransform, LinearTrendTransform, BinsegTrendTransform,
    TheilSenTrendTransform, MedianOutliersTransform,
    DensityOutliersTransform, TimeSeriesImputerTransform,
    StandardScalerTransform, MinMaxScalerTransform,
    RobustScalerTransform, MaxAbsScalerTransform)
```

Дополнительно импортируем класс `OldMeanTransform`, в котором реализован старый способ вычисления скользящих средних.

```
# импортируем класс OldMeanTransform
from etna_utils import OldMeanTransform
```

С помощью параметра `in_column` класса-трансформера мы задаем переменную, которую нужно преобразовать или на основе которой нужно создать признаки (как правило, это название зависимой переменной). С помощью параметра `out_column` можно задать имена генерируемых признаков (этот пара-

метр есть у всех классов-трансформеров, создающих признаки), в противном случае это имя будет создано автоматически. Параметр `out_column` будет только у классов-трансформеров, генерирующих признаки.

Библиотека ETNA предлагает ряд полезных функций обнаружения выбросов на основе определенного метода.

```
# импортируем функции обнаружения выбросов
# на основе определенного метода
from etna.analysis.outliers import (
    get_anomalies_median,
    get_anomalies_density,
    get_anomalies_hist)
```

Нам понадобится класс `RandomForestRegressor` для построения модели, на основе которой будут вычислены важности признаков.

```
# импортируем класс RandomForestRegressor для
# построения модели, на основе которой
# будут вычислены важности признаков
from sklearn.ensemble import RandomForestRegressor
```

Теперь импортируем классы с различными реализациями градиентного бустинга (`LightGBM`, `XGBoost`, реализации градиентного бустинга `scikit-learn`) и класс `BaggingRegressor`, которые мы реализуем в ETNA для прогнозирования временных рядов. Класс `BaggingClassifier` примечателен тем, что в нем реализована метамодель, которая обучает базовые алгоритмы на случайном подмножестве исходного набора, а затем объединяет их индивидуальные прогнозы путем голосования или усреднения с целью получения итогового ответа. Когда случайные подмножества набора представляют собой случайные выборки наблюдений без возвращения (подвыборки наблюдений), речь идет о пэстинге. Когда случайные подмножества набора представляют собой случайные выборки наблюдений с возвращением (бутстреп-выборки наблюдений), речь идет о бэггинге. Когда случайные подмножества набора представляют собой случайные выборки признаков, речь идет о случайных подпространствах. Когда случайные подмножества набора представляют собой подвыборки наблюдений и признаков, речь идет о случайных патчах.

```
# импортируем класс LGBMRegressor
from lightgbm import LGBMRegressor
# импортируем класс XGBRegressor
from xgboost import XGBRegressor
# импортируем классы HistGradientBoostingRegressor,
# GradientBoostingRegressor
from sklearn.ensemble import (HistGradientBoostingRegressor,
                               GradientBoostingRegressor,
                               BaggingRegressor)
```

Затем импортируем классы `CatBoostRegressor`, `Pool` библиотеки `catboost`. Они пригодятся нам для иллюстрации того, что происходит под капотом классов `CatBoostPerSegmentModel` и `CatBoostMultiSegmentModel`.

```
# импортируем классы CatBoostRegressor, Pool
from catboost import CatBoostRegressor, Pool
```

Импортируем класс `ExponentialSmoothing` библиотеки `statsmodels`.

```
# импортируем класс ExponentialSmoothing
from statsmodels.tsa.api import ExponentialSmoothing
```

Импортируем функции `plot_acf()`, `plot_pacf()` для построения графиков автокорреляции и частной автокорреляции, функцию `adfuller()` для проведения расширенного теста Дикки-Фуллера. Эти функции потребуются для предварительного анализа перед построением модели SARIMAX.

```
# импортируем функции plot_acf(), plot_pacf() для построения
# графиков автокорреляции и частной автокорреляции,
# функцию adfuller() для проведения расширенного теста
# Дикки-Фуллера
from statsmodels.graphics.tsaplots import (plot_acf,
                                           plot_pacf)
from statsmodels.tsa.stattools import adfuller
```

Импортируем классы `LinearRegression`, `Pipeline` и `PolynomialFeatures` библиотеки `scikit-learn` для иллюстрации детрендинга. Обратите внимание: класс `Pipeline` библиотеки `scikit-learn` импортируем как `Pipeline_sklearn`, чтобы избежать конфликта с классом `Pipeline` библиотеки `ETNA`.

```
# импортируем классы LinearRegression, Pipeline (как Pipeline_sklearn),
# PolynomialFeatures для иллюстрации детрендинга
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline as Pipeline_sklearn
from sklearn.preprocessing import PolynomialFeatures
```

Импортируем типы `Optional` и `List` для аннотирования.

```
# импортируем типы Optional и List для аннотирования
from typing import Optional, List
```

Импортируем класс `GroupNormalizer` для выполнения групповой нормализации, которая необходима для моделей нейронных сетей.

```
# импортируем класс GroupNormalizer для
# выполнения групповой нормализации
from pytorch_forecasting.data import GroupNormalizer
```

Наконец, отключаем предупреждения, увеличиваем количество отображаемых столбцов и настраиваем визуализацию.

```
# отключаем предупреждения
import warnings
warnings.filterwarnings('ignore')

# увеличиваем количество отображаемых столбцов
pd.set_option('display.max_columns', 200)
# настроим визуализацию
%config InlineBackend.figure_format = 'retina'
```

Загрузим данные о ежемесячных продажах вина в Австралии.

```
# загружаем набор о ежемесячных продажах вина в Австралии
original_df = pd.read_csv('Data/monthly_australian_wine_sales.csv')
original_df.head()
```

	month	sales
0	1980-01-01	15136
1	1980-02-01	16733
2	1980-03-01	20016
3	1980-04-01	17708
4	1980-05-01	18019

ETNA требует определенного формата данных: столбец, который мы хотим спрогнозировать, должен называться `target`, столбец с временными метками должен называться `timestamp`. Поскольку библиотека может работать с несколькими временными рядами, столбец `segment` также является обязательным. Если мы работаем с одним временным рядом, т. е. с одним сегментом, для столбца `segment` по соглашению используется значение `'main'` (хотя это не является принципиальным).

```
# задаем обязательные столбцы target, timestamp и segment
original_df['timestamp'] = pd.to_datetime(original_df['month'])
original_df['target'] = original_df['sales']
original_df.drop(columns=['month', 'sales'], inplace=True)
original_df['segment'] = 'main'
original_df.head()
```

	timestamp	target	segment
0	1980-01-01	15136	main
1	1980-02-01	16733	main
2	1980-03-01	20016	main
3	1980-04-01	17708	main
4	1980-05-01	18019	main

## 2.2. СОЗДАНИЕ ОБЪЕКТА TSDATASET

Библиотека работает со специальной структурой данных `TSDataset`. Класс `TSDataset` – основной класс библиотеки ETNA для работы с временными рядами. Его главные методы:

- `.to_dataset()` – превращает датафрейм с плоским индексом в датафрейм с мультииндексом;
- `.to_pandas()` – превращает объект `TSDataset` в датафрейм с плоским индексом (`flatten=True`) или датафрейм с мультииндексом (`flatten=False`);
- `.fit_transform()` – вычисляет и применяет преобразования и конструирование признаков (например, логарифмирует зависимую переменную, прогнозирует тренд и удаляет тренд из зависимой переменной, вычисляет скользящее среднее с шириной окна 20 дней и создает столбец с ним);
- `.transform()` – применяет преобразования и конструирование признаков (например, логарифмирует зависимую переменную, удаляет тренд из зависимой переменной и создает столбец со скользящим средним);

- `.inverse_transform()` – применяет обратные преобразования (например, выполняет экспоненцирование прологарифмированной зависимой переменной, добавляет тренд к зависимой переменной с удаленным трендом);
- `.make_future()` – создает тестовый набор / набор новых данных, увеличив обучающий/исторический набор данных на длину горизонта прогнозирования и применив преобразования и конструирование признаков;
- `.train_test_split()` – разбивает на обучающую и тестовую выборки с учетом временной структуры;
- `.describe()` – вывод описательных статистик;
- `.plot()` – визуализирует временной ряд в переменной *target* (в зависимости от количества сегментов может быть визуализирован один или несколько временных рядов).

Давайте подробнее разберем работу методов `.to_dataset()` и `.to_pandas()`. С помощью функции `generate_const_df()` сгенерируем датафрейм `pandas` с искусственными данными, у него будет два сегмента и плоский индекс.

```
# сгенерируем датафрейм с двумя сегментами
# и плоским индексом
expl = generate_const_df(
    periods=30, start_time='2021-06-01',
    n_segments=2, scale=1)
print(type(expl))
expl.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

	timestamp	segment	target
0	2021-06-01	segment_0	1.0
1	2021-06-02	segment_0	1.0
2	2021-06-03	segment_0	1.0
3	2021-06-04	segment_0	1.0
4	2021-06-05	segment_0	1.0

Теперь с помощью метода `.to_dataset()` класса `TSdataset` превращаем наш датафрейм с плоским индексом в датафрейм с мультииндексом.

```
# превращаем наш датафрейм с плоским индексом
# в датафрейм с мультииндексом
expl_ts_format = TSdataset.to_dataset(expl)
print(type(expl_ts_format))
expl_ts_format.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

segment	segment_0	segment_1
feature	target	target
timestamp		
2021-06-01	1.0	1.0
2021-06-02	1.0	1.0
2021-06-03	1.0	1.0
2021-06-04	1.0	1.0
2021-06-05	1.0	1.0

А сейчас с помощью класса TSDataset превращаем наш датафрейм в объект TSDataset, указав частоту временного ряда.

```
# превращаем наш датафрейм с мультииндексом
# в объект TSDataset
expl_ts_format = TSDataset(expl_ts_format, 'D')
print(type(expl_ts_format))
expl_ts_format.head()
```

```
<class 'etna.datasets.tsdataset.TSDataset'>
```

segment	segment_0	segment_1
feature	target	target
timestamp		
2021-06-01	1.0	1.0
2021-06-02	1.0	1.0
2021-06-03	1.0	1.0
2021-06-04	1.0	1.0
2021-06-05	1.0	1.0

Теперь с помощью значения параметра `flatten=True` метода `.to_pandas()` превращаем объект TSDataset в датафрейм с плоским индексом.

```
# превращаем объект TSDataset в датафрейм с плоским индексом
ftidx_expl = expl_ts_format.to_pandas(flatten=True)
print(type(ftidx_expl))
ftidx_expl.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

	timestamp	target	segment
0	2021-06-01	1.0	segment_0
1	2021-06-02	1.0	segment_0
2	2021-06-03	1.0	segment_0
3	2021-06-04	1.0	segment_0
4	2021-06-05	1.0	segment_0

Теперь с помощью значения параметра `flatten=False` метода `.to_pandas()` превращаем объект TSDataset в датафрейм с мультииндексом.

```
# превращаем объект TSDataset в датафрейм с мультииндексом
mltidx_expl = expl_ts_format.to_pandas(flatten=False)
print(type(mltidx_expl))
mltidx_expl.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

segment	segment_0	segment_1
feature	target	target
timestamp		
2021-06-01	1.0	1.0
2021-06-02	1.0	1.0
2021-06-03	1.0	1.0
2021-06-04	1.0	1.0
2021-06-05	1.0	1.0

Давайте вернемся к нашим данным о продажах вина и превратим датафрейм pandas в удобный мультииндексный датафрейм.

```
# вернемся к данным о продажах вина и превратим датафрейм
# с плоским индексом в датафрейм с мультииндексом
df = TSDataset.to_dataset(original_df)
df.head()
```

segment	main
feature	target
timestamp	
1980-01-01	15136
1980-02-01	16733
1980-03-01	20016
1980-04-01	17708
1980-05-01	18019

Видим, что сегмент представляет собой верхний уровень мультииндекса столбцов.

Превращаем датафрейм в объект TSDataset, задав частоту временного ряда.

```
# превращаем датафрейм в объект TSDataset,
# задав частоту временного ряда
ts = TSDataset(df, freq='MS')
```

Как вариант можно написать функцию, которая потом позволит вам в одну строку преобразовывать набор данных с временными рядами в объект TSDataset.

```
# напишем функцию преобразования
# набора данных в объект TSDataset
def read_ts(df, date_column, freq='D', melt_df=False):
    """
    Преобразовывает набор данных с временными
    рядами в объект TSDataset.
```

```

Параметры
-----
df: pandas.DataFrame
    Набор данных с временным рядом.
date_column: string
    Столбец с датами.
freq: string, по умолчанию 'D'
    Частота временного ряда.
melt_df: bool, по умолчанию False
    Имеет ли набор данных длинный формат
    (long format).
"""
df = df.rename(columns={date_column: 'timestamp'})
if not melt_df:
    df = df.melt(id_vars='timestamp',
                 var_name='segment',
                 value_name='target')
df = TSDataset.to_dataset(df)
ts = TSDataset(df, freq=freq)
return ts

# загружаем набор о ежемесячных продажах вина в Австралии
df = pd.read_csv('Data/monthly_australian_wine_sales.csv')
# с помощью нашей функции получаем объект TSDataset
timeseries = read_ts(df, date_column='month', freq='MS', melt_df=False)
timeseries.head()

```

segment	sales
feature	target
timestamp	
1980-01-01	15136
1980-02-01	16733
1980-03-01	20016
1980-04-01	17708
1980-05-01	18019

Получаем тот же объект TSDataset с той только разницей, что для столбца segment будет использоваться значение – имя зависимой переменной 'sales' (это не повлияет на результаты вычислений).

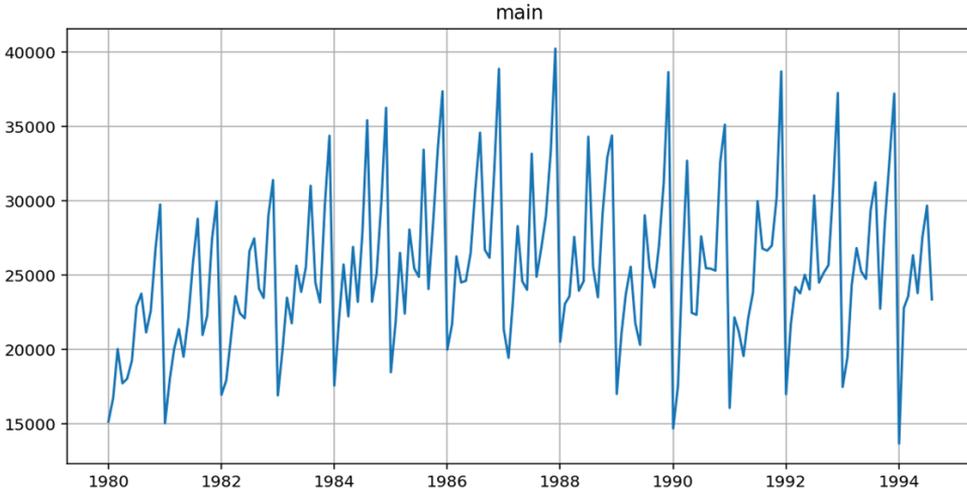
## 2.3. ВИЗУАЛИЗАЦИЯ РЯДОВ ОБЪЕКТА TSDATASET

Давайте визуализируем временной ряд.

```

# визуализируем временной ряд
ts.plot()

```



Временной ряд является ежемесячным. Мы видим тренд, который является не линейным, а скорее затухающим. Он сначала растет, выходит на среднее, и примерно в 1989 году происходит изменение структуры тренда. Кроме того, этот ряд обладает ярко выраженной годовой сезонностью: максимум продаж за год приходится на декабрь, а затем, в январе, происходит существенное падение.

## 2.4. ПОЛУЧЕНИЕ СВОДКИ ХАРАКТЕРИСТИК ПО ОБЪЕКТУ `TSDataset`

С помощью метода `.describe()` можем посмотреть базовые характеристики объекта `TSDataset`:

- `start_timestamp` – стартовая дата временного ряда;
- `end_timestamp` – конечная дата временного ряда;
- `length` – длина временного ряда;
- `num_missing` – количество пропусков;
- `num_segments` – количество сегментов (рядов);
- `num_exogs` – количество экзогенных переменных;
- `num_regressors` – количество регрессоров;
- `num_known_future` – количество регрессоров, добавленных из датафрейма `df_exog`;
- `freq` – частота временного ряда.

Все переменные в ETNA делятся на целевые переменные (собственно прогнозируемые временные ряды, которые получают названия `target`) и экзогенные переменные – все переменные, кроме целевых. Упрощенно говоря, экзогенные переменные будут нашими признаками, помогающими спрогнозировать целевую переменную. Экзогенные переменные делятся на переменные, которые мы создаем с помощью трансформеров ETNA, и переменные, которые мы создаем самостоятельно и добавляем к признакам, созданным с помощью ETNA, с помощью параметра `df_exog` класса `TSDataset`. Экзогенные переменные вне зависимости от того, как они были созданы (в ETNA или самостоятельно), делятся на два типа –

экзогенные переменные, значения которых будут известны в будущем (в ETNA их называют регрессорами), и экзогенные переменные, значения которых будут не известны в будущем. Например, лаг является регрессором, последние значения лага станут значениями в тестовой выборке/выборке новых данных. А скользящая статистика не является регрессором, чтобы значения скользящей статистики можно было использовать в тестовой выборке/выборке новых данных, ее нужно взять с лагом и тогда скользящая статистика, взятая с лагом, становится регрессором.

```
# посмотрим базовые характеристики
ts.describe()
```

	start_timestamp	end_timestamp	length	num_missing	num_segments	num_exogs	num_regressors	num_known_future	freq
<b>segments</b>									
main	1980-01-01	1994-08-01	176	0	1	0	0	0	MS

## 2.5. Модель наивного прогноза

Допустим, наш горизонт прогнозирования равен 8 месяцам. Разбиваем набор (наш объект `TSDataset`) на обучающую и тестовую выборки с учетом временной структуры, так чтобы последние 8 месяцев ушли в тестовую выборку.

```
# разбиваем набор на обучающую и тестовую выборки с учетом временной структуры
train_ts, test_ts = ts.train_test_split(train_start='1980-01-01',
                                       train_end='1993-12-01',
                                       test_start='1994-01-01',
                                       test_end='1994-08-01')
```

Теперь явно задаем горизонт прогнозирования – 8 меток времени (8 месяцев).

```
# задаем горизонт прогнозирования -
# 8 меток времени (8 месяцев)
HORIZON = 8
```

Прогнозирование временных рядов нужно начинать с самых простых моделей. Мы начнем с модели наивного прогноза, т. е. в качестве прогноза берем последнее известное значение. Для этого воспользуемся классом `NaiveModel` – моделью наивного прогноза и обучим ее. По сути, она представляет собой обертку вокруг класса `SeasonalMovingAverageModel` – модели сезонного скользящего среднего. Прогноз для точки  $y_t$  вычисляется как среднее  $y_{t-s}, y_{t-2s}, \dots, y_{t-ns}$ , где  $s$  – это сезонность (лаг между значениями, используемыми для прогноза), а  $n$  – размер окна (количество значений в исторической выборке, которое мы берем для прогноза). По умолчанию для класса `SeasonalMovingAverageModel` заданы значения параметров `seasonality=7` и `window=5`, для класса `NaiveModel` задано `window=1` без возможности изменения извне (поскольку в наивном прогнозе используем последнее известное значение), а параметр `seasonality` настраивается через параметр `lag`, по умолчанию значение для параметра `lag` равно 1.

```
# создаем экземпляр класса NaiveModel, в котором
# реализована модель наивного прогноза
model = NaiveModel()
# обучаем модель наивного прогноза
model.fit(train_ts)
```

Формируем набор, для которого нужно получить прогнозы (по сути, тестовую выборку), длина набора определяется горизонтом прогнозирования.

```
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
future_ts
```

segment	main
feature	target
timestamp	
1994-01-01	NaN
1994-02-01	NaN
1994-03-01	NaN
1994-04-01	NaN
1994-05-01	NaN
1994-06-01	NaN
1994-07-01	NaN
1994-08-01	NaN

← здесь будут  
наши прогнозы

С помощью метода `.forecast()` получаем наши прогнозы.

```
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main
feature	target
timestamp	
1994-01-01	37198.0
1994-02-01	37198.0
1994-03-01	37198.0
1994-04-01	37198.0
1994-05-01	37198.0
1994-06-01	37198.0
1994-07-01	37198.0
1994-08-01	37198.0

Видим, что прогнозы – это последнее известное значение, т. е. последнее значение обучающей выборки. Давайте убедимся в этом, для этого выведем последние 12 наблюдений обучающей выборки.

```
# выведем последние 12 наблюдений обучающей выборки
train_ts.tail(12)
```

segment	main
feature	target
timestamp	
1993-01-01	17466
1993-02-01	19463
1993-03-01	24352
1993-04-01	26805
1993-05-01	25236
1993-06-01	24735
1993-07-01	29356
1993-08-01	31234

Теперь создаем модель наивного прогноза, задав 3 лага. Прогнозами будут последние 3 известных значения.

```
# создаем экземпляр класса NaiveModel, в котором
# реализована модель наивного прогноза,
# задав 3 лага
model = NaiveModel(lag=3)
# обучаем модель наивного прогноза
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main
feature	target
timestamp	
1994-01-01	28496.0
1994-02-01	32857.0
1994-03-01	37198.0
1994-04-01	28496.0
1994-05-01	32857.0
1994-06-01	37198.0
1994-07-01	28496.0
1994-08-01	32857.0

segment	main
feature	target
timestamp	
1993-01-01	17466
1993-02-01	19463
1993-03-01	24352
1993-04-01	26805
1993-05-01	25236
1993-06-01	24735
1993-07-01	29356
1993-08-01	31234
1993-09-01	22724
1993-10-01	28496
1993-11-01	32857
1993-12-01	37198

Здесь мы видим, что для прогнозов используем уже последние 3 значения обучающей выборки. Логично, что количество лагов лучше задавать равным горизонту прогнозирования и выше его, чтобы избежать повторения одних и тех же прогнозов.

Итак, создаем модель наивного прогноза, задав 8 лагов, т. е. по длине горизонта. Теперь прогнозами будут последние 8 известных значений.

```
# создаем экземпляр класса NaiveModel, в котором
# реализована модель наивного прогноза,
# задав 8 лагов
model = NaiveModel(lag=8)
# обучаем модель наивного прогноза
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main	segment	main
feature	target	feature	target
timestamp		timestamp	
		1993-01-01	17466
		1993-02-01	19463
		1993-03-01	24352
		1993-04-01	26805
1994-01-01	25236.0	1993-05-01	25236
1994-02-01	24735.0	1993-06-01	24735
1994-03-01	29356.0	1993-07-01	29356
1994-04-01	31234.0	1993-08-01	31234
1994-05-01	22724.0	1993-09-01	22724
1994-06-01	28496.0	1993-10-01	28496
1994-07-01	32857.0	1993-11-01	32857
1994-08-01	37198.0	1993-12-01	37198

Наконец, создадим модель наивного прогноза, задав 12 лагов.

```
# создаем экземпляр класса NaiveModel, в котором
# реализована модель наивного прогноза,
# задав 12 лагов
model = NaiveModel(lag=12)
# обучаем модель наивного прогноза
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)

# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main	segment	main
feature	target	feature	target
timestamp		timestamp	
1994-01-01	17466.0	1993-01-01	17466
1994-02-01	19463.0	1993-02-01	19463
1994-03-01	24352.0	1993-03-01	24352
1994-04-01	26805.0	1993-04-01	26805
1994-05-01	25236.0	1993-05-01	25236
1994-06-01	24735.0	1993-06-01	24735
1994-07-01	29356.0	1993-07-01	29356
1994-08-01	31234.0	1993-08-01	31234
		1993-09-01	22724
		1993-10-01	28496
		1993-11-01	32857
		1993-12-01	37198

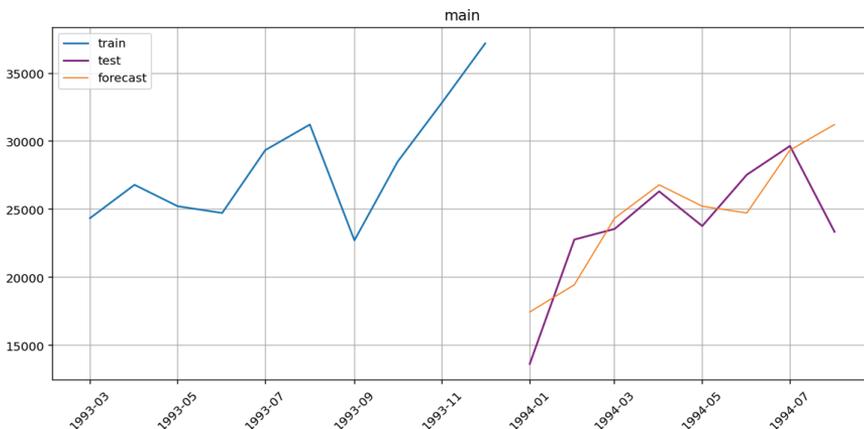
Теперь прогнозами будут уже не последние 8 наблюдений, мы делаем 4 шага назад (поскольку разница между горизонтом прогнозирования 8 и количеством лагов 12 составляет 4) с конца обучающей выборки и берем последние 8 наблюдений. Таким образом, прогнозами становятся уже более ранние наблюдения.

Оценим качество наших прогнозов с помощью метрики SMAPE (реализована в соответствующем классе SMAPE).

```
# вычисляем метрику SMAPE
smape = SMAPE()
smape(y_true=test_ts, y_pred=forecast_ts)
{'main': 11.492045838249387}
```

С помощью функции `plot_forecast()` визуализируем наши прогнозы. Параметр `n_train_samples` – `n` последних наблюдений в обучающей выборке.

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_ts, test_ts,
              train_ts, n_train_samples=10)
```



Удобство библиотеки ETNA заключается в том, что мы можем применить нашу модель сразу к нескольким временным рядам и получить прогнозы для каждого из них.

Давайте загрузим набор данных, содержащий несколько временных рядов. Речь идет о ежедневном потреблении электроэнергии по 4 сегментам.

```
# теперь загружаем набор, в котором каждому
# сегменту соответствует свой временной ряд
original_df2 = pd.read_csv('Data/example_dataset.csv')
original_df2.head()
```

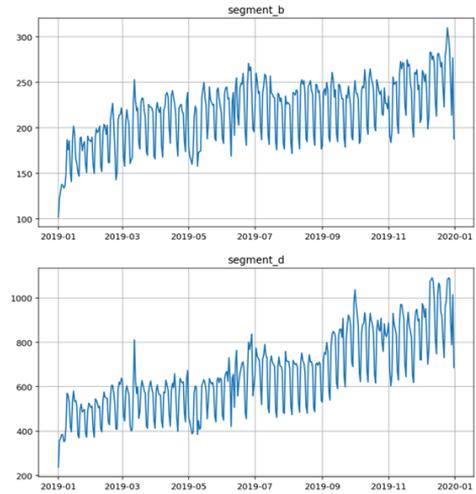
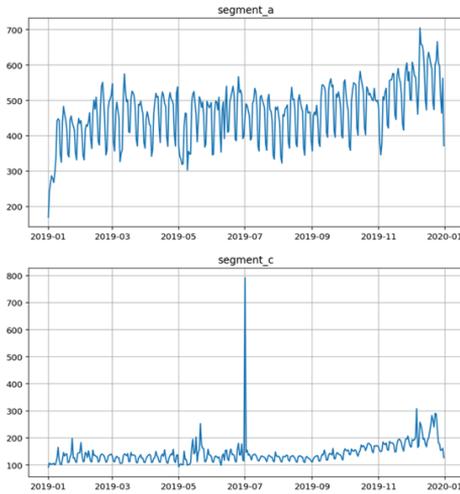
	timestamp	segment	target
0	2019-01-01	segment_a	170
1	2019-01-02	segment_a	243
2	2019-01-03	segment_a	267
3	2019-01-04	segment_a	287
4	2019-01-05	segment_a	279

Теперь превратим наш датафрейм в мультииндексный датафрейм, затем в объект TSDataset и визуализируем ряды.

```
# конвертируем датафрейм в формат ETNA
df2 = TSDataset.to_dataset(original_df2)
df2.head()
```

	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-01-01	170	102	92	238
2019-01-02	243	123	107	358
2019-01-03	267	130	103	366
2019-01-04	287	138	103	385
2019-01-05	279	137	104	384

```
# превращаем датафрейм в объект TSDataset,
# задав частоту временного ряда
mult_ts = TSDataset(df2, freq='D')
# визуализируем ряды
mult_ts.plot()
```



Горизонт прогнозирования зададим равным 7 дням.

```
# задаем горизонт прогнозирования -
# 7 меток времени (7 дней)
mult_HORIZON = 7
```

Разбиваем набор (наш объект TSDataset) на обучающую и тестовую выборки с учетом временной структуры, так чтобы последние 7 дней ушли в тестовую выборку.

```
# разбиваем набор на обучающую и тестовую
# выборки с учетом временной структуры
train_mult_ts, test_mult_ts = mult_ts.train_test_split(
    train_start='2019-01-01',
    train_end='2019-12-24',
    test_start='2019-12-25',
    test_end='2019-12-31')
```

Теперь нам осталось обучить модель наивного прогноза, получить прогнозы, оценить их качество и визуализировать.

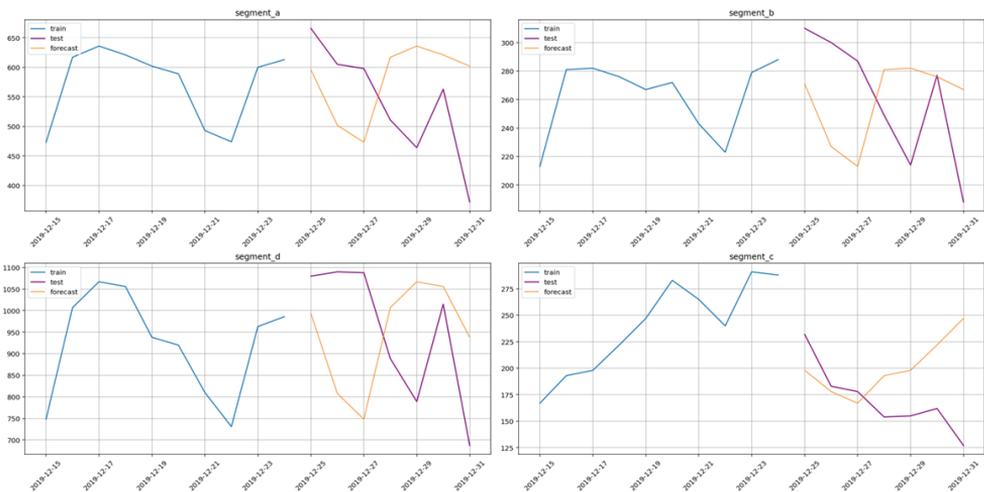
```
# обучаем модель наивного прогноза
model.fit(train_mult_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_mult_ts = train_mult_ts.make_future(mult_HORIZON)
# получаем прогнозы
forecast_mult_ts = model.forecast(future_mult_ts)
forecast_mult_ts
```

segment	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-12-25	596.0	271.0	198.0	994.0
2019-12-26	502.0	227.0	178.0	808.0
2019-12-27	473.0	213.0	167.0	748.0
2019-12-28	617.0	281.0	193.0	1007.0
2019-12-29	636.0	282.0	198.0	1067.0
2019-12-30	621.0	276.0	222.0	1056.0
2019-12-31	602.0	267.0	247.0	938.0

```
# вычисляем метрику SMAPE
smape(y_true=test_mult_ts, y_pred=forecast_mult_ts)
```

```
{'segment_a': 22.8767574890079,
 'segment_b': 20.75869697463065,
 'segment_d': 21.757368837930883,
 'segment_c': 23.888994797023148}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_mult_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```



## 2.6. Модель скользящего среднего

Теперь для нашего ряда продаж австралийского вина построим модель скользящего среднего. Для этого воспользуемся классом `MovingAverageModel` – моделью скользящего среднего и обучим ее. По сути, он тоже представляет собой

обертку вокруг уже известного нам класса `SeasonalMovingAverageModel` – модели сезонного скользящего среднего. Для класса `MovingAverageModel` задано `seasonality=1` без возможности изменения извне, а параметр `window` настраивается через параметр `window`, по умолчанию значение для параметра `window` равно 5.

```
# создаем экземпляр класса MovingAverageModel, в котором
# реализована модель скользящего среднего
model = MovingAverageModel(window=5)
# обучаем модель скользящего среднего
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main
feature	target
timestamp	
1994-01-01	30501.800000
1994-02-01	30355.360000
1994-03-01	31881.632000
1994-04-01	32558.758400
1994-05-01	32499.110080
1994-06-01	31559.332096
1994-07-01	31770.838515
1994-08-01	32053.934218

Давайте выясним, как были вычислены скользящие средние.

Для этого задаем ширину окна (`window`), сезонность (`seasonality`), при этом сезонность всегда равна 1, помним, что ее нельзя поменять извне, и смещение (`shift`), вычисляемое как произведение ширины окна и сезонности. Смещение (`shift`) будет всегда равно ширине окна, поскольку сезонность всегда равна 1. Затем выделяем зависимую переменную в обучающей выборке и записываем в отдельную переменную. Из полученного массива берем `shift` последних наблюдений в обучающей выборке.

```
# задаем ширину окна
window = 5
# задаем сезонность (константа, нельзя изменить извне)
seasonality = 1
# задаем смещение
shift = window * seasonality
# выделяем зависимую переменную в обучающей выборке
targets = original_df[0:len(original_df) - HORIZON]['target']
# берем shift последних наблюдений в обучающей выборке
series = targets[-shift:].values
series

array([31234, 22724, 28496, 32857, 37198])
```

Создаем массив, состоящий из `shift` последних наблюдений обучающей выборки и `HORIZON` нулевых значений, и вычисляем скользящие средние. Для этого формируем окно из первых `window` наблюдений массива `res`, наблюдения берем с шагом, равным `seasonality` (т. е. с шагом 1, первые пять наблюдений), по ним вычисляем скользящее среднее, передвигаем окно на одно значение вперед, добавив в конец окна вычисленное скользящее среднее, снова вычисляем скользящее среднее – и так до тех пор, пока не получим `HORIZON` скользящих средних.

```
# создаем массив, состоящий из shift последних наблюдений
# обучающей выборки и HORIZON нулевых значений
res = np.append(series, np.zeros(HORIZON))
print(res)
print('')
# вычисляем скользящие средние
for i in range(shift, len(res)):
    print(res[i - shift : i : seasonality])
    res[i] = res[i - shift : i : seasonality].mean()
    print(res[i])
    y_pred = res[-HORIZON:]
```

```
[31234. 22724. 28496. 32857. 37198.    0.    0.    0.    0.    0.
  0.    0.    0.]
```

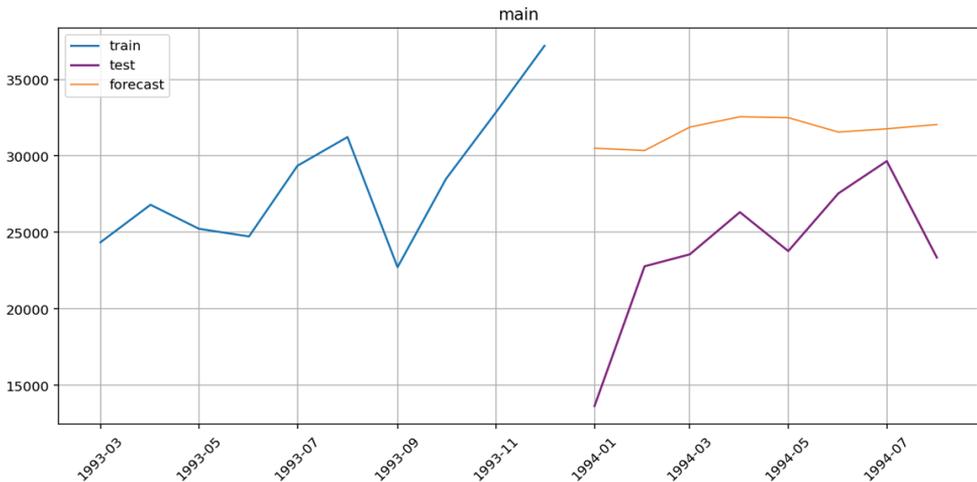
```
[31234. 22724. 28496. 32857. 37198.]
30501.8
[22724. 28496. 32857. 37198. 30501.8]
30355.359999999997
[28496. 32857. 37198. 30501.8 30355.36]
31881.632
[32857. 37198. 30501.8 30355.36 31881.632]
32558.758400000002
[37198. 30501.8 30355.36 31881.632 32558.7584]
32499.110080000002
[30501.8 30355.36 31881.632 32558.7584 32499.11008]
31559.332096000006
[30355.36 31881.632 32558.7584 32499.11008 31559.332096]
31770.8385152
[31881.632 32558.7584 32499.11008 31559.332096 31770.8385152]
32053.934218240007
```

Здесь мы видим, что первый наш прогноз – это среднее, которое вычислено на основе массива, состоящего из последних `HORIZON` наблюдений обучающей выборки. Затем мы каждый раз перемещаем окно и добавляем в конец этого массива ранее вычисленное среднее, которое само по себе является прогнозом, и в какой-то момент мы начнем вычислять скользящее среднее только на основе прогнозов. Таким образом, происходит накопление ошибки прогнозов. Давайте оценим качество прогнозов и визуализируем их, хотя нетрудно догадаться, что качество прогнозов будет невысоким.

```
# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)
```

```
{'main': 29.85307984438601}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_ts, test_ts, train_ts, n_train_samples=10)
```



Теперь строим модель скользящего среднего для потребления электроэнергии по 4 сегментам.

```
# обучаем модель скользящего среднего
model.fit(train_mult_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_mult_ts = train_mult_ts.make_future(mult_HORIZON)
# получаем прогнозы
forecast_mult_ts = model.forecast(future_mult_ts)
forecast_mult_ts
```

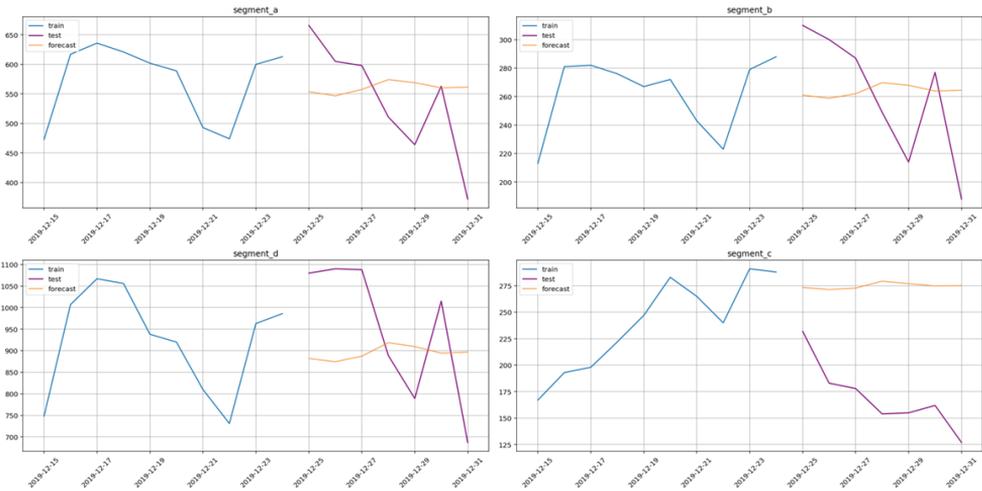
segment	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-12-25	553.800000	261.000000	273.400000	882.000000
2019-12-26	546.760000	258.800000	271.480000	874.400000
2019-12-27	557.512000	261.960000	272.776000	887.280000
2019-12-28	574.214400	269.752000	279.331200	918.536000
2019-12-29	569.057280	267.902400	276.997440	909.643200
2019-12-30	560.268736	263.882880	274.796928	894.371840
2019-12-31	561.562483	264.459456	275.076314	896.846208

```
# вычисляем метрику SMAPE
smape(y_true=test_mult_ts, y_pred=forecast_mult_ts)
```

```
{'segment_a': 15.514795643434814,
 'segment_b': 15.72148537309564,
```

```
'segment_d': 17.009119473438595,
'segment_c': 48.142748849658304}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_mult_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```



## 2.7. Модель сезонного скользящего среднего

А сейчас для продаж австралийского вина построим модель сезонного скользящего среднего. Для этого воспользуемся классом `SeasonalMovingAverageModel` – моделью сезонного скользящего среднего и обучим ее. Мы зададим `window=5` и `seasonality=12`.

```
# создаем экземпляр класса SeasonalMovingAverageModel, в котором
# реализована модель сезонного скользящего среднего
model = SeasonalMovingAverageModel(window=5, seasonality=12)
# обучаем модель скользящего среднего
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main
feature	target
timestamp	
1994-01-01	16431.0
1994-02-01	20391.6
1994-03-01	23784.4
1994-04-01	25668.0
1994-05-01	23306.8
1994-06-01	23036.0
1994-07-01	29253.2
1994-08-01	26689.2

Давайте выясним, как были вычислены сезонные скользящие средние.

Для этого задаем ширину окна (*window*), сезонность (*seasonality*), ее теперь можно менять, и смещение (*shift*), вычисляемое как произведение ширины окна и сезонности. Затем выделяем зависимую переменную в обучающей выборке и записываем в отдельную переменную. Из полученного массива берем *shift* последних наблюдений в обучающей выборке.

```
# задаем ширину окна
window = 5
# задаем сезонность (можно менять)
seasonality = 12
# задаем смещение
shift = window * seasonality
# выделяем зависимую переменную в обучающей выборке
targets = original_df[0:len(original_df) - HORIZON]['target']
# берем shift последних наблюдений в обучающей выборке
series = targets[-shift:].values
series

array([16991, 21109, 23740, 25552, 21752, 20294, 29009, 25500, 24166,
       26960, 31222, 38641, 14672, 17543, 25453, 32683, 22449, 22316,
       27595, 25451, 25421, 25288, 32568, 35110, 16052, 22146, 21198,
       19543, 22084, 23816, 29961, 26773, 26635, 26972, 30207, 38687,
       16974, 21697, 24179, 23757, 25013, 24019, 30345, 24488, 25156,
       25650, 30923, 37240, 17466, 19463, 24352, 26805, 25236, 24735,
       29356, 31234, 22724, 28496, 32857, 37198])
```

Создаем массив, состоящий из *shift* последних наблюдений обучающей выборки и *HORIZON* нулевых значений, и вычисляем сезонные скользящие средние. Для этого формируем окно из первых *window* наблюдений массива *res*, наблюдения берем с шагом, равным *seasonality* (т. е. с шагом 12, 1-е наблюдение, 13-е наблюдение, 25-е наблюдение, 37-е наблюдение, 49-е наблюдение), по ним вычисляем скользящее среднее, передвигаем окно на одно значение вперед, снова берем наблюдения с шагом, равным *seasonality*, снова вычисляем скользящее среднее – и так до тех пор, пока не получим *HORIZON* скользящих средних.

```

# создаем массив, состоящий из shift последних наблюдений
# обучающей выборки и HORIZON нулевых значений
res = np.append(series, np.zeros(HORIZON))
print(res)
print('')
# вычисляем скользящие средние
for i in range(shift, len(res)):
    print(res[i - shift : i : seasonality])
    res[i] = res[i - shift : i : seasonality].mean()
    print(res[i])
    y_pred = res[-HORIZON:]

[16991. 21109. 23740. 25552. 21752. 20294. 29009. 25500. 24166. 26960.
 31222. 38641. 14672. 17543. 25453. 32683. 22449. 22316. 27595. 25451.
 25421. 25288. 32568. 35110. 16052. 22146. 21198. 19543. 22084. 23816.
 29961. 26773. 26635. 26972. 30207. 38687. 16974. 21697. 24179. 23757.
 25013. 24019. 30345. 24488. 25156. 25650. 30923. 37240. 17466. 19463.
 24352. 26805. 25236. 24735. 29356. 31234. 22724. 28496. 32857. 37198.
   0.    0.    0.    0.    0.    0.    0.    0.]

[16991. 14672. 16052. 16974. 17466.]
16431.0
[21109. 17543. 22146. 21697. 19463.]
20391.6
[23740. 25453. 21198. 24179. 24352.]
23784.4
[25552. 32683. 19543. 23757. 26805.]
25668.0
[21752. 22449. 22084. 25013. 25236.]
23306.8
[20294. 22316. 23816. 24019. 24735.]
23036.0
[29009. 27595. 29961. 30345. 29356.]
29253.2
[25500. 25451. 26773. 24488. 31234.]
26689.2

```

Давайте оценим качество прогнозов и визуализируем их.

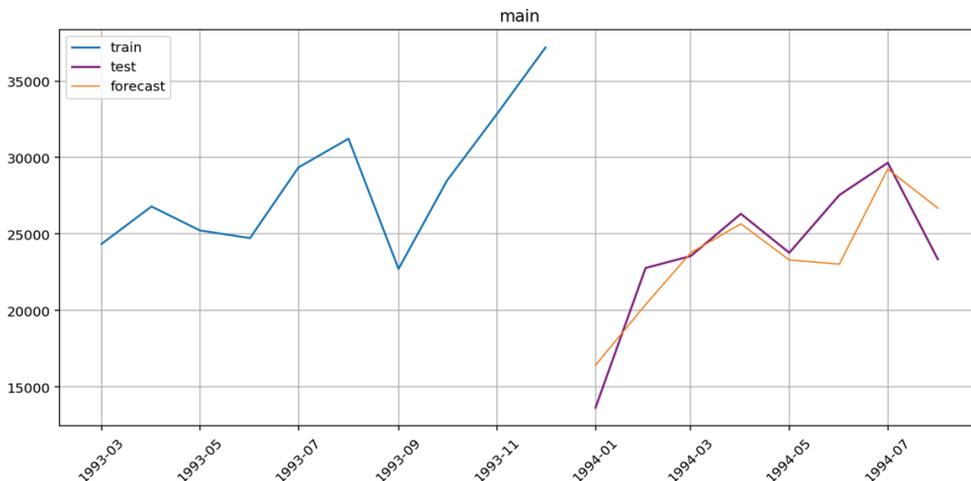
```

# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)

{'main': 8.444354755762415}

# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_ts, test_ts, train_ts, n_train_samples=10)

```



А сейчас создадим и построим модель сезонного скользящего среднего, прогнозирующую потребление электроэнергии по 4 сегментам.

```
# создаем экземпляр класса SeasonalMovingAverageModel, в котором
# реализована модель сезонного скользящего среднего
model = SeasonalMovingAverageModel(window=5, seasonality=7)
# обучаем модель сезонного скользящего среднего
model.fit(train_mult_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_mult_ts = train_mult_ts.make_future(mult_HORIZON)
# получаем прогнозы
forecast_mult_ts = model.forecast(future_mult_ts)
forecast_mult_ts
```

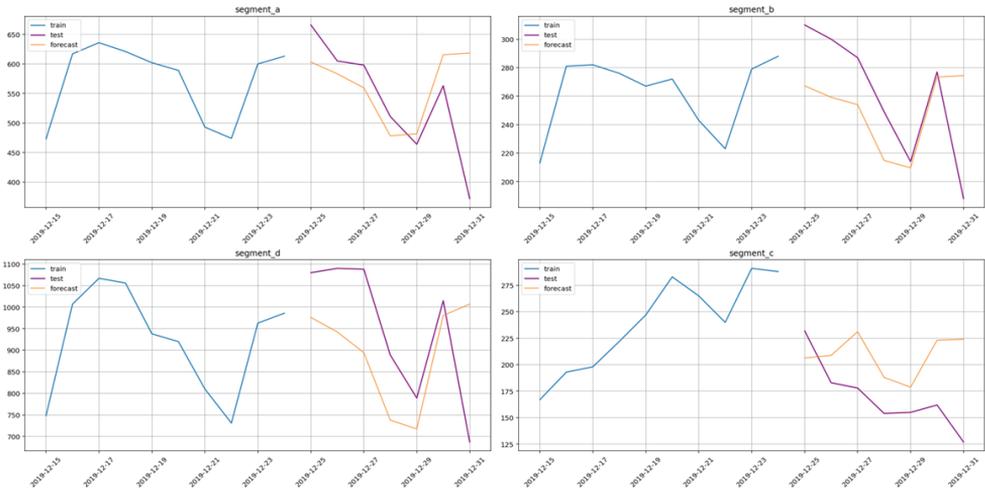
segment	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-12-25	603.2	267.2	206.4	976.2
2019-12-26	583.2	259.2	208.8	942.4
2019-12-27	559.6	254.0	231.2	894.4
2019-12-28	478.2	214.8	188.0	737.8
2019-12-29	481.8	209.6	178.8	717.4
2019-12-30	615.6	273.4	223.2	980.6
2019-12-31	618.4	274.4	224.0	1007.2

```
# вычисляем метрику SMAPE
smape(y_true=test_mult_ts, y_pred=forecast_mult_ts)
```

```
{'segment_a': 12.75413973441905,
 'segment_b': 13.875095600615348,
```

```
'segment_d': 16.2135368848414,
'segment_c': 24.577172132520488}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_mult_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```



Итак, мы строили простые модели, для которых требуется только сам временной ряд, нам не нужно создавать признаки, выполнять какие-то преобразования. Теперь перейдем к более сложным моделям, для которых требуются определенные преобразования и конструирование признаков.

## 2.8. Модель SARIMAX

SARIMAX (аббревиатура от **S**easonal **A**uto**R**egressive **I**ntegrated **M**oving **A**verage with **e**Xogenous factors) – это сезонная авторегрессионная интегрированная модель скользящего среднего (SARIMAX-модель) с экзогенными переменными. Напомним, что по соглашению параметры несезонной части обозначаются буквами нижнего регистра, а параметры сезонной части обозначаются буквами верхнего регистра.

$$(p, d, q)$$

несезонная  
часть модели

$$(P, D, Q)_s$$

сезонная  
часть модели

- $p$  – порядок (несезонной) авторегрессии;
- $d$  – порядок (несезонного) дифференцирования;
- $q$  – порядок (несезонного) скользящего среднего;
- $P$  – порядок сезонной авторегрессии;
- $D$  – порядок сезонного дифференцирования;
- $Q$  – порядок сезонного скользящего среднего;
- $s$  – периодичность (количество периодов в полном сезонном цикле).

SARIMAX требует стационарного временного ряда. Для этого потребуются некоторые преобразования.

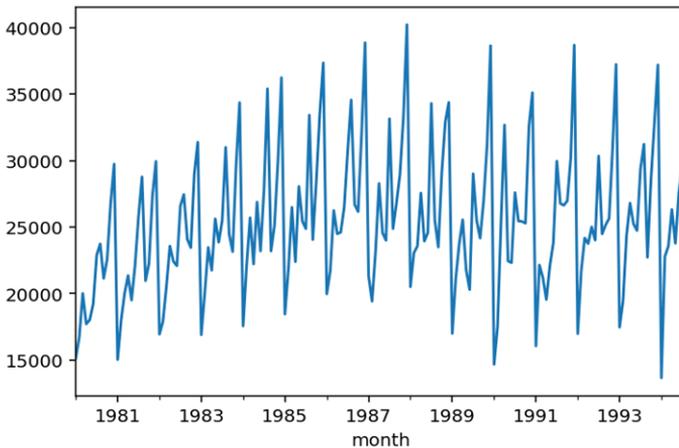
Давайте загрузим наш ряд с продажами вина в виде обычного объекта Series и визуализируем его.

```
# загружаем данные в виде серии
wine_data = pd.read_csv('Data/monthly_australian_wine_sales.csv',
                        header=0,
                        index_col=0,
                        parse_dates=True).squeeze('columns')

wine_data.head()

month
1980-01-01    15136
1980-02-01    16733
1980-03-01    20016
1980-04-01    17708
1980-05-01    18019
Name: sales, dtype: int64

# визуализируем ряд
wine_data.plot();
```



Ряд имеет сложную структуру тренда и сезонность и, разумеется, не является стационарным. Для проверки временного ряда на стационарность также применяются тесты на единичные корни, в частности расширенный тест Дикки-Фуллера. Нулевая гипотеза теста звучит так: рассматриваемый ряд является нестационарным. Давайте воспользуемся этим тестом.

```
# применяем расширенный тест Дикки-Фуллера
result = adfuller(wine_data)
# печатаем результаты теста
print("Статистика ADF: %f" % result[0])
print("p-значение: %f" % result[1])
print("Критические значения:")
for key, value in result[4].items():
    print("\t%s: %.3f" % (key, value))
```

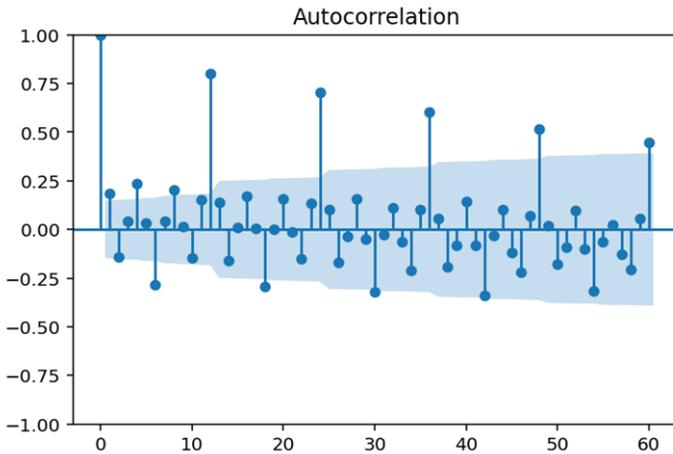
Статистика ADF: -2.852468  
 p-значение: 0.051161  
 Критические значения:  
 1%: -3.471  
 5%: -2.879  
 10%: -2.576

Видим, что  $p$ -значение (вероятность ложного отклонения нулевой гипотезы) выше 0,05. Мы не можем отклонить нулевую гипотезу о том, что временной ряд является нестационарным. Делаем вывод, что тестируемый ряд является нестационарным.

Итак, тренд, сезонность и результаты теста говорят о нестационарности этого временного ряда, поэтому сначала нужно применить сезонное дифференцирование. Однако для этого нужно определить количество периодов для сезонного дифференцирования.

Для этого строим график ACF. Лаги, соответствующие длине полного сезонного цикла ( $s$ ) или кратные ему ( $2s$ ,  $3s$  и т. д.), будут иметь высокие значения коэффициентов.

```
# строим график автокорреляционной функции
plot_acf(wine_data, lags=60);
```



Мы видим высокие значения коэффициентов в лагах 12, 24 и т. д. На поведение коэффициентов именно в этих лагах (при условии значимости коэффициентов) нужно будет смотреть, чтобы определить количество сезонных AR-членов или сезонных MA-членов. Количество периодов для сезонного дифференцирования и соответственно параметр  $s$  – количество периодов в полном сезонном цикле для SARIMAX-модели (последнее значение кортежа (0,0,0,0) для параметра `seasonal_order`) будут равны 12.

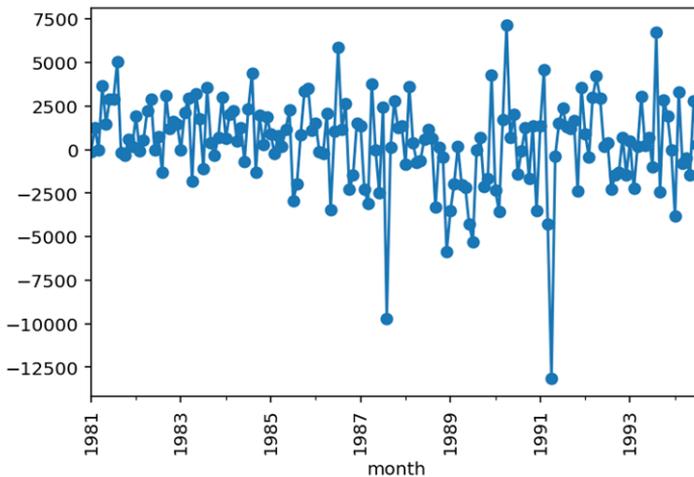
Давайте напишем функцию дифференцирования, с помощью которой можно будет выполнить сезонное дифференцирование для превращения нестационарного временного ряда в стационарный. Мы будем выполнять преобразование по формуле `series[i] - series[i - periods]` и тем самым получим новый ряд.

```
# пишем функцию для выполнения дифференцирования
def difference(series, periods=1):
    diff = list()
    for i in range(periods, len(series)):
        value = series[i] - series[i - periods]
        diff.append(value)
    return pd.Series(diff)
```

Итак, выполняем сезонное дифференцирование с количеством периодов 12 и визуализируем получившийся ряд.

```
# выполняем сезонное дифференцирование
wine_seasonal_diff = difference(wine_data, periods=12)
wine_seasonal_diff.index = wine_data.index[12:]
wine_seasonal_diff = wine_seasonal_diff.rename_axis('month')

# визуализируем временной ряд
wine_seasonal_diff.plot(rot=90, marker='o');
```



Снова воспользуемся расширенным тестом Дикки-Фуллера.

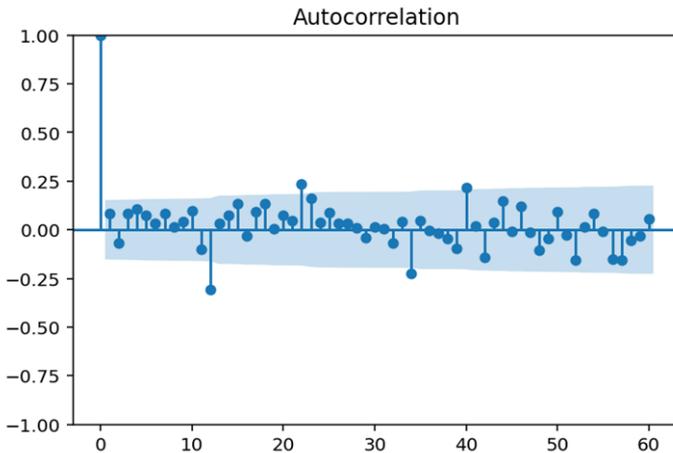
```
# применяем расширенный тест Дикки-Фуллера
result = adfuller(wine_seasonal_diff)
# печатаем результаты теста
print("Статистика ADF: %f" % result[0])
print("p-значение: %f" % result[1])
print("Критические значения:")
for key, value in result[4].items():
    print("\t%s: %.3f" % (key, value))
```

```
Статистика ADF: -2.622085
p-значение: 0.088518
Критические значения:
1%: -3.475
5%: -2.881
10%: -2.577
```

Вновь видим, что  $p$ -значение (вероятность ложного отклонения нулевой гипотезы) выше 0,05. Мы не можем отклонить нулевую гипотезу о том, что временной ряд является нестационарным. Делаем вывод, что тестируемый ряд является нестационарным.

Взглянем на график ACF.

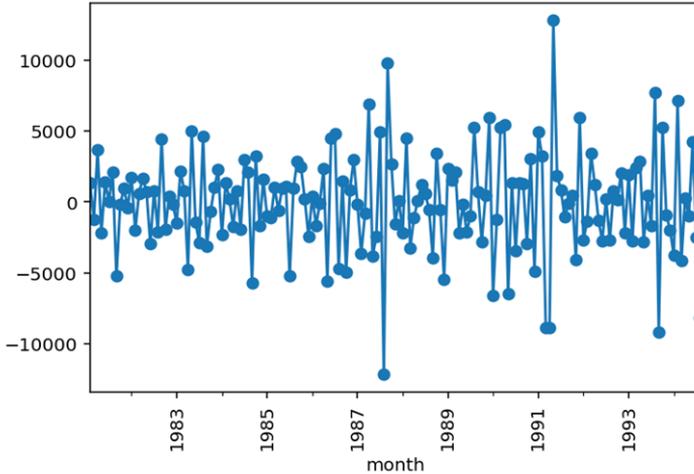
```
# строим график автокорреляционной функции
plot_acf(wine_seasonal_diff, lags=60);
```



Применяем к нашему временному ряду, который уже был преобразован с помощью сезонного дифференцирования, дифференцирование первого порядка и визуализируем получившийся ряд.

```
# применяем дифференцирование первого порядка к ряду,
# к которому было применено сезонное дифференцирование
wine_seasonal_diff_frst_diff = difference(wine_seasonal_diff, 1)
wine_seasonal_diff_frst_diff.index = wine_seasonal_diff.index[1:]
wine_seasonal_diff_frst_diff = wine_seasonal_diff_frst_diff.rename_axis('month')
```

```
# визуализируем временной ряд
wine_seasonal_diff_frst_diff.plot(rot=90, marker='o');
```



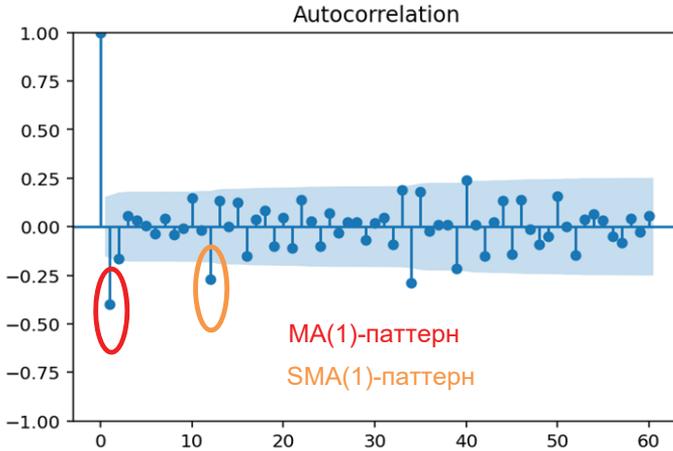
Опять воспользуемся расширенным тестом Дикки-Фуллера.

```
# применяем расширенный тест Дикки-Фуллера
result = adfuller(wine_seasonal_diff)
# печатаем результаты теста
print("Статистика ADF: %f" % result[0])
print("p-значение: %f" % result[1])
print("Критические значения:")
for key, value in result[4].items():
    print("\t%s: %.3f" % (key, value))
```

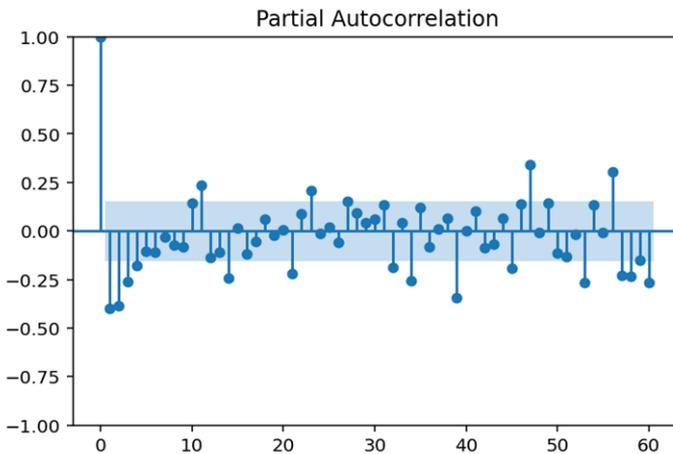
```
Статистика ADF: -5.522619
p-значение: 0.000002
Критические значения:
1%: -3.475
5%: -2.881
10%: -2.577
```

Теперь тест позволяет нам отвергнуть нулевую гипотезу о нестационарности рассматриваемого ряда. Мы можем заняться подбором параметров  $p$ ,  $d$ ,  $q$ ,  $P$ ,  $D$ ,  $Q$  и  $s$ . В этом нам помогут графики ACF и PACF. Параметры  $d$ ,  $D$  и  $s$  у нас уже подобраны:  $d = 1$ ,  $D = 1$  и  $s = 12$ .

```
# строим график автокорреляционной функции
plot_acf(wine_seasonal_diff_frst_diff, lags=60);
```



# строим график частной автокорреляционной функции  
`plot_pacf(wine_seasonal_diff_frst_diff, lags=60);`



Если график ACF демонстрирует резкое падение коэффициентов и/или имеет отрицательную автокорреляцию в лаге 1, т. е. если ряд слегка свёрхдифференцирован (при этом на графике PACF обычно происходит затухание коэффициентов экспоненциально или по синусоиде), то нужно рассмотреть добавление MA-члена в модель. Номер лага, в котором начинается резкое падение на графике ACF, – это количество MA-членов. Здесь мы можем рассматривать MA(1)-паттерн. Таким образом, несезонная часть SARIMAX (параметр `order`) будет выглядеть как (0, 1, 1).

Осталось подобрать параметры P и Q.

Если автокорреляция в сезонный период положительна, рассмотрите возможность добавления SAR-члена в модель. Если автокорреляция в сезонный период отрицательна, рассмотрите возможность добавления SMA-члена в модель. В нашем случае автокорреляция в сезонный период отрицательна. Здесь

мы можем рассматривать SMA(1)-паттерн. Таким образом, сезонная часть SARIMAX (параметр `seasonal_order`) будет выглядеть как (0, 1, 1, 12). По сути мы будем строить  $SARIMA(0,1,12)(0,1,1)_{12}$ -модель. Давайте займемся этим. Обратите внимание, по умолчанию строится  $SARIMA(2,1,0)(1,1,0)_{12}$ -модель.

```
# создаем экземпляр класса SARIMAXModel
model = SARIMAXModel(order=(0, 1, 1),
                      seasonal_order=(0, 1, 1, 12))
# обучаем модель SARIMAX
model.fit(train_ts);
```

Получаем прогнозы, оцениваем качество модели и визуализируем прогнозы.

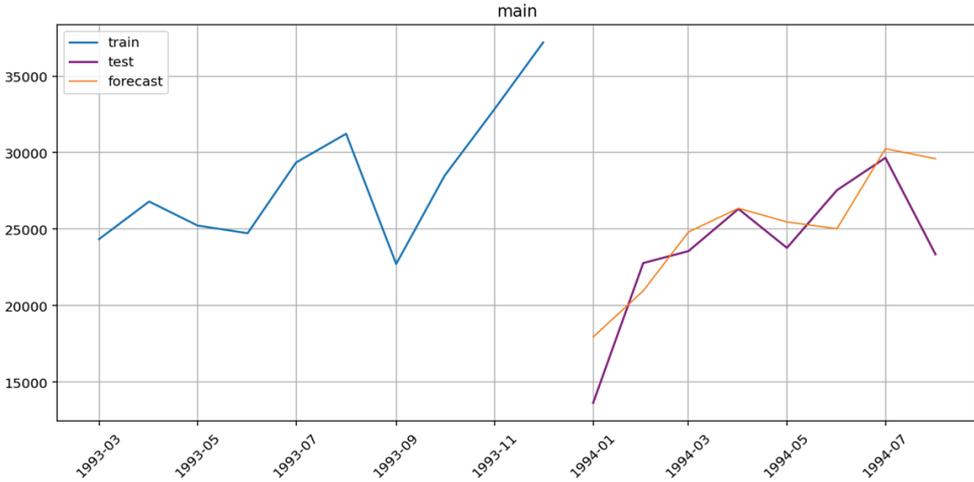
```
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main
feature	target
timestamp	
1994-01-01	17954.929734
1994-02-01	20970.647076
1994-03-01	24813.077323
1994-04-01	26365.565701
1994-05-01	25472.486561
1994-06-01	25031.715503
1994-07-01	30258.338614
1994-08-01	29606.651278

```
# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)
```

```
{'main': 10.361344664891181}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_ts, test_ts, train_ts, n_train_samples=10)
```



А сейчас мы создадим и построим модель SARIMAX для прогнозирования потребления электроэнергии по 4 сегментам. Мы воспользуемся моделью с настройками по умолчанию SARIMA(2,1,0)(1,1,0)<sub>12</sub>-модель), изменив лишь значение параметра  $s$  на 7, поскольку графики явно содержат недельную сезонность.

```
# создаем экземпляр класса SARIMAXModel
model = SARIMAXModel(seasonal_order=(1, 1, 0, 7))
# обучаем модель SARIMAX
model.fit(train_mult_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_mult_ts = train_mult_ts.make_future(mult_HORIZON)

# получаем прогнозы
forecast_mult_ts = model.forecast(future_mult_ts)
forecast_mult_ts
```

segment	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-12-25	605.739431	281.981635	292.143161	997.929058
2019-12-26	581.913113	275.746166	289.836735	927.628256
2019-12-27	552.676557	275.678545	310.320533	875.393132
2019-12-28	458.171390	240.908868	291.822971	728.062459
2019-12-29	433.459400	222.957168	273.361123	657.923746
2019-12-30	566.995394	283.581159	312.616350	902.064155
2019-12-31	582.168120	289.239564	313.606301	942.475802

```
# вычисляем метрику SMAPE
```

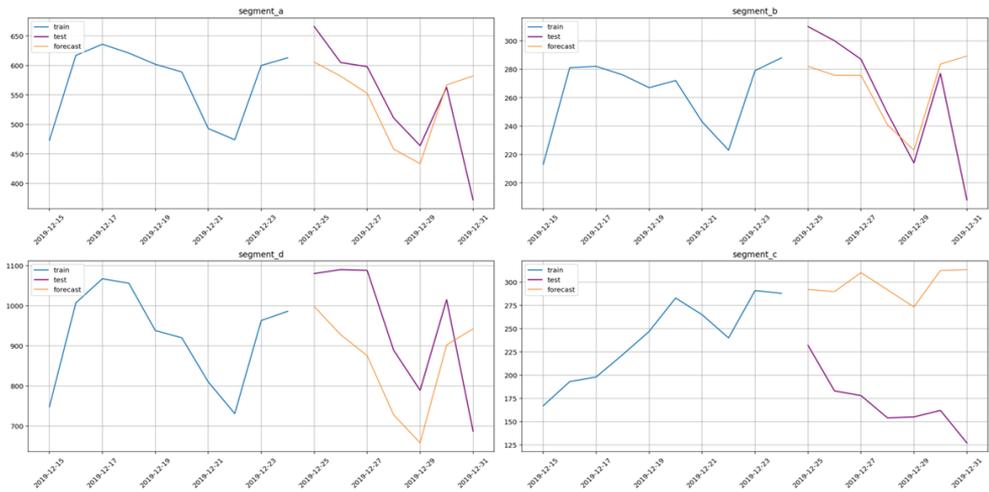
```
smape(y_true=test_mult_ts, y_pred=forecast_mult_ts)
```

```
{'segment_a': 11.958918711818542,
 'segment_b': 10.58475745100541,
 'segment_d': 18.11621681182218,
 'segment_c': 55.37097600019527}
```

```
# визуализируем прогнозы, n_train_samples -
```

```
# n последних наблюдений в обучающей выборке
```

```
plot_forecast(forecast_mult_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```



## 2.9. Модель Хольта–Винтерса (модель тройного экспоненциального сглаживания, модель ETS)

Математический аппарат моделей тройного экспоненциального сглаживания можно представить в виде таксономии, приведенной ниже.

Тренд		Сезонность	
Без тренда		Аддитивная	Мультипликативная
значение уровня		$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)\ell_{t-1}$	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)\ell_{t-1}$
значение тренда			значение тренда
значение сезонности		$I_t = \gamma(y_t - \ell_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности $I_t = \gamma(y_t/\ell_{t-1}) + (1 - \gamma)I_{t-L}$
сглаженное значение		$S_t = \ell_{t-1} + I_{t-L}$	сглаженное значение $S_t = \ell_{t-1} \cdot I_{t-L}$
прогноз		$F_{t+m} = \ell_t + I_{t-L+1+(m-1)}$	прогноз $F_{t+m} = \ell_t \cdot I_{t-L+1+(m-1)}$
<b>Аддитивный</b>			
значение уровня		$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$
значение тренда		$b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$	значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$
значение сезонности		$I_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} + b_{t-1})) + (1 - \gamma)I_{t-L}$
сглаженное значение		$S_t = \ell_{t-1} + b_{t-1} + I_{t-L}$	сглаженное значение $S_t = (\ell_{t-1} + b_{t-1}) \cdot I_{t-L}$
прогноз		$F_{t+m} = \ell_t + mb_t + I_{t-L+1+(m-1)}$	прогноз $F_{t+m} = (\ell_t + mb_t) \cdot I_{t-L+1+(m-1)}$
<b>Аддитивный с затуханием</b>			
значение уровня		$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$
значение тренда		$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)\phi b_{t-1}$	значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)\phi b_{t-1}$
значение сезонности		$I_t = \gamma(y_t - \ell_{t-1} - \phi b_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} + \phi b_{t-1})) + (1 - \gamma)I_{t-L}$
сглаженное значение		$S_t = \ell_{t-1} + \phi b_{t-1} + I_{t-L}$	сглаженное значение $S_t = (\ell_{t-1} + \phi b_{t-1}) \cdot I_{t-L}$
прогноз		$F_{t+m} = \ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t + I_{t-L+1+(m-1)}$	прогноз $F_{t+m} = (\ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t) \cdot I_{t-L+1+(m-1)}$
<b>Мультипликативный</b>			
значение уровня		$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$
значение тренда		$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$	значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$
значение сезонности		$I_t = \gamma(y_t - \ell_{t-1} \cdot b_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1})) + (1 - \gamma)I_{t-L}$
сглаженное значение		$S_t = (\ell_{t-1} \cdot b_{t-1}) + I_{t-L}$	сглаженное значение $S_t = (\ell_{t-1} \cdot b_{t-1}) \cdot I_{t-L}$
прогноз		$F_{t+m} = \ell_t \cdot b_t^m + I_{t-L+1+(m-1)}$	прогноз $F_{t+m} = \ell_t \cdot b_t^m \cdot I_{t-L+1+(m-1)}$
<b>Мультипликативный с затуханием</b>			
значение уровня		$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$	значение уровня $\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$
значение тренда		$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}^\phi$	значение тренда $b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}^\phi$
значение сезонности		$I_t = \gamma(y_t - \ell_{t-1} \cdot b_{t-1}^\phi) + (1 - \gamma)I_{t-L}$	значение сезонности $I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1}^\phi)) + (1 - \gamma)I_{t-L}$
сглаженное значение		$S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) + I_{t-L}$	сглаженное значение $S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) \cdot I_{t-L}$
прогноз		$F_{t+m} = (\ell_t \cdot (b_t^{\phi + \phi^2 + \dots + \phi^m})) + I_{t-L+1+(m-1)}$	прогноз $F_{t+m} = (\ell_t \cdot (b_t^{\phi + \phi^2 + \dots + \phi^m})) \cdot I_{t-L+1+(m-1)}$

где:

- $\ell_t$  – значение уровня для момента времени  $t$ ;
- $b_t$  – значение тренда для момента времени  $t$ ;
- $\alpha$  – сглаживающая константа для уровня;
- $\beta$  – сглаживающая константа для тренда;
- $\gamma$  – сглаживающая константа для сезонности;
- $\phi$  – коэффициент затухания;
- $I_t$  – значение индекса сезонности для момента времени  $t$ ;
- $L$  – количество периодов;
- $m$  – количество периодов, на которое делается прогноз.

Применим модель тройного экспоненциального сглаживания к продажам австралийского вина.

# создаем модель ETS

```
model = HoltWintersModel(seasonal='additive',
                        trend='additive',
                        seasonal_periods=12,
                        smoothing_level=0.05,
                        smoothing_trend=0.01,
                        smoothing_seasonal=0.35)
```

# обучаем модель ETS

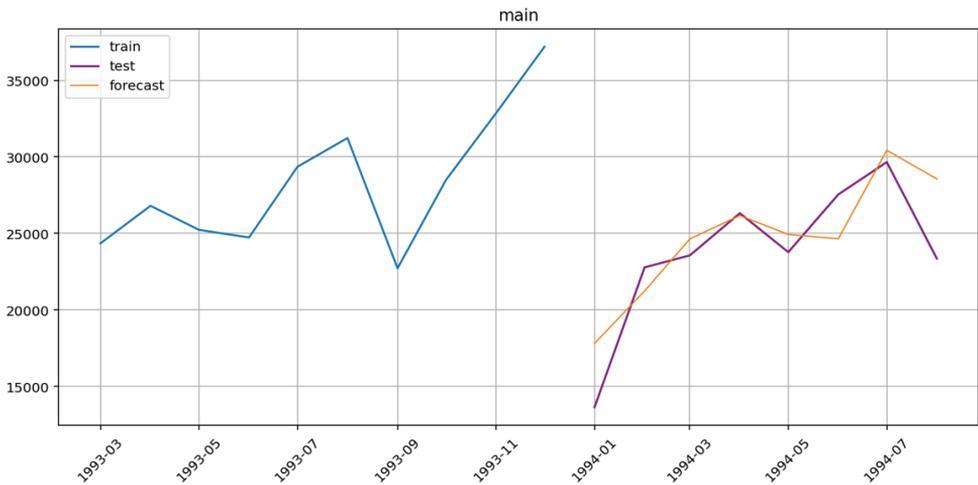
```
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	main
feature	target
timestamp	
1994-01-01	17816.728640
1994-02-01	21239.004422
1994-03-01	24640.815327
1994-04-01	26170.157860
1994-05-01	24933.606383
1994-06-01	24651.221358
1994-07-01	30434.767469
1994-08-01	28558.822451

```
# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)
```

```
{'main': 9.624875288710275}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_ts, test_ts, train_ts, n_train_samples=10)
```



Давайте выясним, как были получены прогнозы. Для этого превратим обучающий набор, записанный в виде объекта `TSDataset`, в обычный плоский датафрейм, построим модель тройного экспоненциального сглаживания с теми же значениями параметров с помощью класса `ExponentialSmoothing`, получим значения уровня, значения тренда, сезонные значения, сглаженные значения, которые объединим в одном датафрейме, и, наконец, получим прогнозы.

```
# обучающий набор запишем в виде плоского датафрейма
tr = train_ts.to_pandas(flatten=True)['target']
```

```
# выполняем тройное экспоненциальное сглаживание
# для аддитивного тренда и аддитивной сезонности
# с помощью класса ExponentialSmoothing
triple = ExponentialSmoothing(train_ts.to_pandas(flatten=True)['target'],
                              trend='additive', seasonal='additive',
                              seasonal_periods=12,
                              initialization_method='estimated').fit(
    smoothing_level=0.05, smoothing_trend=0.01, smoothing_seasonal=0.35)
```

```
# получаем фактические значения
actual_values = tr
# получаем значения уровня
level_values = triple.level
# получаем значения тренда
trend_values = triple.trend
# получаем сезонные значения
season_values = triple.season
# получаем сглаженные значения
smooth_values = triple.fittedvalues
# создаем датафрейм с результатами
res = pd.DataFrame({'Y': actual_values,
                   'L': level_values,
                   'B': trend_values,
                   'S': season_values,
                   'F': smooth_values})
```

```
# получаем прогнозы
triple_preds = triple.forecast(HORIZON)
print(res, '\n=====')
print(triple_preds)
```

	Y	L	B	S	F
0	15136	21340.005301	38.019269	-6500.426153	14641.965248
1	16733	21370.450694	37.943530	-4546.564179	16884.477525
2	20016	21416.120605	38.020794	-1492.837177	19861.472380
3	17708	21371.603087	37.195411	-2673.143342	19358.766242
4	18019	21337.637708	36.483803	-2464.708221	19442.215811
...	...	...	...	...	...
163	31234	27180.708403	28.904293	1129.246849	26360.592088
164	22724	27040.211178	27.210278	-2283.392961	26112.030363
165	28496	27144.718914	27.983253	423.711590	26950.050840
166	32857	27217.893573	28.435167	5096.809557	31953.171883
167	37198	27205.370897	28.025588	10484.123211	38017.156847

```
[168 rows x 5 columns]
```

```
=====
```

```
168 17816.728640
169 21239.004422
170 24640.815327
171 26170.157860
172 24933.606383
173 24651.221358
174 30434.767469
175 28558.822451
dtype: float64
```

Теперь обращаемся к таксономии, нас интересует формула прогнозов для сочетания аддитивной сезонности и аддитивного тренда:

Тренд	Аддитивная		Сезонность	
Без тренда	значение уровня	$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)\ell_{t-1}$	значение уровня	$\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)\ell_{t-1}$
	значение тренда	-	значение тренда	-
	значение сезонности	$I_t = \gamma(y_t - \ell_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности	$I_t = \gamma(y_t/\ell_{t-1}) + (1 - \gamma)I_{t-L}$
	сглаженное значение	$S_t = \ell_{t-1} + I_{t-L}$	сглаженное значение	$S_t = \ell_{t-1} \cdot I_{t-L}$
прогноз	$F_{t+m} = \ell_t + I_{t-L+1+(m-1)}$	прогноз	$F_{t+m} = \ell_t \cdot I_{t-L+1+(m-1)}$	
Аддитивный	значение уровня	$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$	значение уровня	$\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} + b_{t-1})$
	значение тренда	$b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)b_{t-1}$	значение тренда	$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$
	значение сезонности	$I_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности	$I_t = \gamma(y_t/(\ell_{t-1} + b_{t-1})) + (1 - \gamma)I_{t-L}$
	сглаженное значение	$S_t = \ell_{t-1} + b_{t-1} + I_{t-L}$	сглаженное значение	$S_t = (\ell_{t-1} + b_{t-1}) \cdot I_{t-L}$
прогноз	$F_{t+m} = \ell_t + mb_t + I_{t-L+1+(m-1)}$	прогноз	$F_{t+m} = (\ell_t + mb_t) \cdot I_{t-L+1+(m-1)}$	
Аддитивный с затуханием	значение уровня	$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$	значение уровня	$\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} + \phi b_{t-1})$
	значение тренда	$b_t = \beta(\ell_t - \ell_{t-1}) + (1 - \beta)\phi b_{t-1}$	значение тренда	$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)\phi b_{t-1}$
	значение сезонности	$I_t = \gamma(y_t - \ell_{t-1} - \phi b_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности	$I_t = \gamma(y_t/(\ell_{t-1} + \phi b_{t-1})) + (1 - \gamma)I_{t-L}$
	сглаженное значение	$S_t = \ell_{t-1} + \phi b_{t-1} + I_{t-L}$	сглаженное значение	$S_t = (\ell_{t-1} + \phi b_{t-1}) \cdot I_{t-L}$
прогноз	$F_{t+m} = \ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t + I_{t-L+1+(m-1)}$	прогноз	$F_{t+m} = (\ell_t + (\phi + \phi^2 + \dots + \phi^m)b_t) \cdot I_{t-L+1+(m-1)}$	
Мультипликативный	значение уровня	$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$	значение уровня	$\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1})$
	значение тренда	$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$	значение тренда	$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}$
	значение сезонности	$I_t = \gamma(y_t \cdot \ell_{t-1} \cdot b_{t-1}) + (1 - \gamma)I_{t-L}$	значение сезонности	$I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1})) + (1 - \gamma)I_{t-L}$
	сглаженное значение	$S_t = (\ell_{t-1} \cdot b_{t-1}) + I_{t-L}$	сглаженное значение	$S_t = (\ell_{t-1} \cdot b_{t-1}) \cdot I_{t-L}$
прогноз	$F_{t+m} = \ell_t \cdot b_t^m + I_{t-L+1+(m-1)}$	прогноз	$F_{t+m} = \ell_t \cdot b_t^m \cdot I_{t-L+1+(m-1)}$	
Мультипликативный с затуханием	значение уровня	$\ell_t = \alpha(y_t - I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$	значение уровня	$\ell_t = \alpha(y_t/I_{t-L}) + (1 - \alpha)(\ell_{t-1} \cdot b_{t-1}^\phi)$
	значение тренда	$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}^\phi$	значение тренда	$b_t = \beta(\ell_t/\ell_{t-1}) + (1 - \beta)b_{t-1}^\phi$
	значение сезонности	$I_t = \gamma(y_t \cdot \ell_{t-1} \cdot b_{t-1}^\phi) + (1 - \gamma)I_{t-L}$	значение сезонности	$I_t = \gamma(y_t/(\ell_{t-1} \cdot b_{t-1}^\phi)) + (1 - \gamma)I_{t-L}$
	сглаженное значение	$S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) + I_{t-L}$	сглаженное значение	$S_t = (\ell_{t-1} \cdot b_{t-1}^\phi) \cdot I_{t-L}$
прогноз	$F_{t+m} = (\ell_t \cdot (b_t^{(\phi+\phi^2+\dots+\phi^m)})) + I_{t-L+1+(m-1)}$	прогноз	$F_{t+m} = (\ell_t \cdot (b_t^{(\phi+\phi^2+\dots+\phi^m)})) \cdot I_{t-L+1+(m-1)}$	

где:

- $\ell_t$  – значение уровня для момента времени  $t$ ;
- $b_t$  – значение тренда для момента времени  $t$ ;
- $\alpha$  – сглаживающая константа для уровня;
- $\beta$  – сглаживающая константа для тренда;
- $\gamma$  – сглаживающая константа для сезонности;
- $\phi$  – коэффициент затухания;
- $I_t$  – значение индекса сезонности для момента времени  $t$ ;
- $L$  – количество периодов;
- $m$  – количество периодов, на которое делается прогноз.

Прогнозная формула для тройного экспоненциального сглаживания с аддитивным трендом и аддитивной сезонностью выглядит так:  $F_{t+m} = I_t + mb_t + I_{t-L+1+(m-1)}$ , где  $m$  – количество периодов, на которое делается прогноз,  $L$  – количество периодов в полном сезонном цикле.

Зададим  $L$  (в нашем случае количество периодов в полном сезонном цикле равно 12) и выясним, как были получены первый и второй прогнозы.

*# задаем количество сезонных периодов*

`L = 12`

*# выясним, как был получен первый прогноз, прогноз для  $t + 1$ :*

*# это значение уровня для  $t$  плюс значение тренда для  $t$ , умноженное на  $m$ , плюс*

*# значение сезонности для  $t - L + 1 + (m - 1)$ , где  $L$  – количество сезонных периодов,*

*#  $m$  – количество периодов, на которое нужно сделать прогноз, в нашем случае 1*

`m = 1`

`forecast_frst = res.L[167] + m * res.B[167] + res.S[167 - L + 1 + (m - 1)]`

`forecast_frst`

17816.728640340873

*# выясним, как был получен второй прогноз, прогноз для  $t + 2$ :*

*# это значение уровня для  $t$  плюс значение тренда для  $t$ , умноженное на  $m$ , плюс*

*# значение сезонности для  $t - L + 1 + (m - 1)$ , где  $L$  – количество сезонных периодов,*

*#  $m$  – количество периодов, на которое нужно сделать прогноз, в нашем случае 2*

`m = 2`

`forecast_scnd = res.L[167] + m * res.B[167] + res.S[167 - L + 1 + (m - 1)]`

`forecast_scnd`

21239.00442238099

Применим модель тройного экспоненциального сглаживания для прогнозирования потребления электроэнергии по 4 сегментам.

```
# создаем модель ETS
model = HoltWintersModel(seasonal='additive',
                        trend='additive',
                        seasonal_periods=7,
                        smoothing_level=0.1,
                        smoothing_trend=0.1,
                        smoothing_seasonal=0.1)

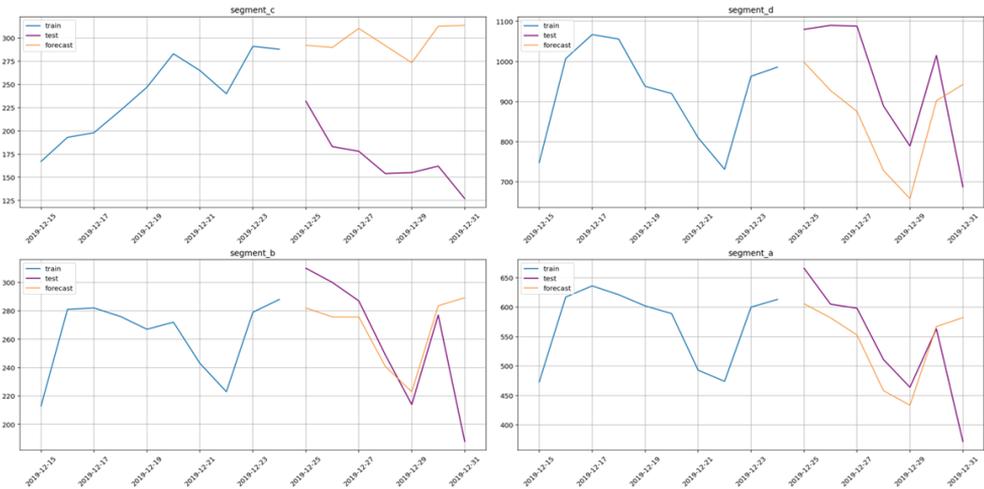
# обучаем модель ETS
model.fit(train_mult_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_mult_ts.make_future(mult_HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
forecast_ts
```

segment	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-12-25	627.306273	280.992010	281.099469	1049.895884
2019-12-26	618.062155	278.590260	286.591345	1029.405752
2019-12-27	602.363126	275.623818	302.409143	1001.550110
2019-12-28	520.314354	240.518589	221.140743	811.031610
2019-12-29	514.584333	235.681252	208.890741	775.068063
2019-12-30	628.016217	287.063425	305.859954	1030.056532
2019-12-31	640.778274	290.907030	337.914037	1091.318101

```
# вычисляем метрику SMAPE
smape(y_true=test_mult_ts, y_pred=forecast_mult_ts)
```

```
{'segment_c': 55.37097600019527,
 'segment_d': 18.11621681182218,
 'segment_b': 10.58475745100541,
 'segment_a': 11.958918711818542}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_mult_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```



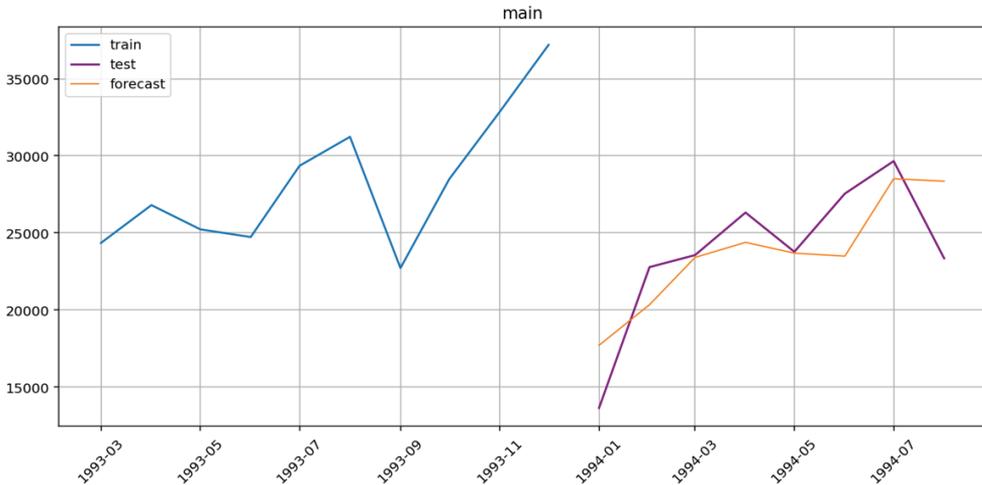
## 2.10. Модель PROPHET

Теперь получим прогнозы продаж австралийского вина с помощью популярной модели Prophet, для этого надо воспользоваться классом-оберткой `ProphetModel`. Строим базовую модель без преобразований и конструирования признаков.

```
# создаем экземпляр класса ProphetModel
model = ProphetModel()
# обучаем модель Prophet
model.fit(train_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_ts.make_future(HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
# оцениваем качество прогнозов
smape(y_true=test_ts, y_pred=forecast_ts)

{'main': 10.626418322451338}

# визуализируем прогнозы
plot_forecast(forecast_ts, test_ts,
              train_ts, n_train_samples=10)
```



Для более надежной оценки качества модели Prophet воспользуемся перекрестной проверкой расширяющимся окном. Ее можно реализовать с помощью класса `Pipeline`.

Сначала создадим список преобразований. В данном случае он будет пустым, мы не будем выполнять каких-либо преобразований или конструировать какие-либо признаки внутри цикла перекрестной проверки.

```
# не будем делать никаких преобразований
transforms = []
```

Теперь создаем конвейер для выполнения перекрестной проверки расширяющимся окном, передав в него модель, пустой список преобразований и горизонт прогнозирования.

```
# создаем конвейер
pipeline = Pipeline(model=model, transforms=transforms, horizon=HORIZON)
```

Запустить перекрестную проверку можно с помощью метода `.backtest()` класса `Pipeline`. Передаем в метод набор данных и указываем следующие параметры:

- `metrics` – список метрик;
- `n_folds` – количество тестовых выборок перекрестной проверки (по умолчанию 5);
- `mode` – режим перекрестной проверки (по умолчанию `'expand'`, можно задать `'expand'` – перекрестную проверку расширяющимся окном или `'constant'` – перекрестную проверку скользящим окном);
- `aggregate_metrics` – агрегирование метрик по тестовым выборкам перекрестной проверки (по умолчанию `False`);
- `n_jobs` – количество ядер процессора для распараллеливания (по умолчанию 1).

Метод `.backtest()` возвращает три датафрейма – датафрейм с метриками по каждой тестовой выборке перекрестной проверки, датафрейм с прогнозами, датафрейм с временными метками обучающей и тестовой выборок перекрестной проверки.

Итак, выполняем перекрестную проверку расширяющимся окном, возвращаем все три датафрейма.

```
# выполняем перекрестную проверку расширяющимся окном,
# возвращаем metrics_df - датафрейм с метриками
# по каждой тестовой выборке перекрестной проверки,
# forecast_df - датафрейм с прогнозами, fold_info_df -
# датафрейм с временными метками обучающей и тестовой
# выборки перекрестной проверки
metrics_df, forecast_df, fold_info_df = pipeline.backtest(
    mode='expand', ts=ts, metrics=[smape])
```

Смотрим датафрейм с метриками по каждой тестовой выборке перекрестной проверки.

```
# смотрим датафрейм с метриками по каждой
# тестовой выборке перекрестной проверки
metrics_df
```

	<b>segment</b>	<b>SMAPE</b>	<b>fold_number</b>
<b>0</b>	main	6.720895	0
<b>0</b>	main	6.394211	1
<b>0</b>	main	4.915642	2
<b>0</b>	main	7.222128	3
<b>0</b>	main	10.626418	4

Смотрим первые 5 строк датафрейма с прогнозами.

```
# смотрим первые 5 строк
# датафрейма с прогнозами
forecast_df.head()
```

<b>segment</b>	<b>main</b>		
<b>feature</b>	<b>target</b>	<b>fold_number</b>	
<b>timestamp</b>			
<b>1991-05-01</b>	23229.092865	0	
<b>1991-06-01</b>	22837.450406	0	
<b>1991-07-01</b>	27569.150843	0	
<b>1991-08-01</b>	27753.810518	0	
<b>1991-09-01</b>	23975.536490	0	

Смотрим датафрейм с временными метками обучающей и тестовой выборки для каждой итерации перекрестной проверки.

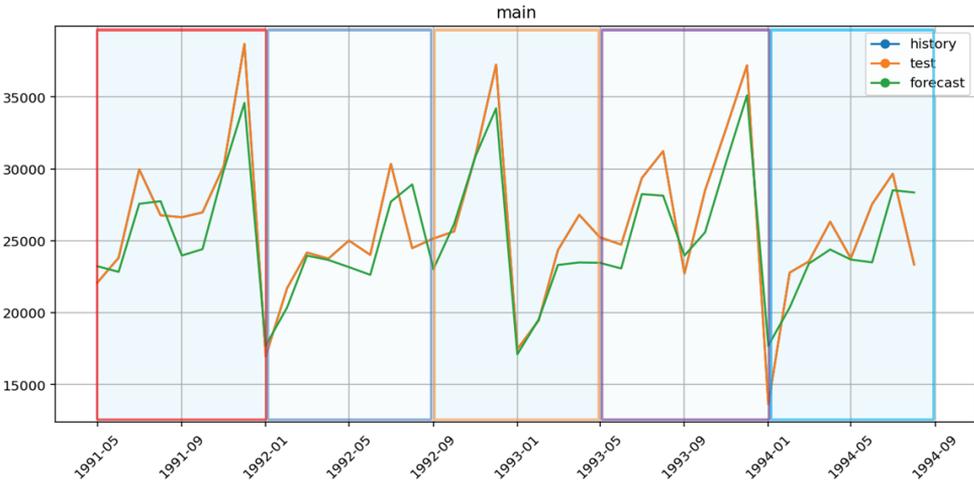
```
# смотрим датафрейм с временными метками
# обучающей и тестовой выборки
fold_info_df
```

	train_start_time	train_end_time	test_start_time	test_end_time	fold_number
0	1980-01-01	1991-04-01	1991-05-01	1991-12-01	0
0	1980-01-01	1991-12-01	1992-01-01	1992-08-01	1
0	1980-01-01	1992-08-01	1992-09-01	1993-04-01	2
0	1980-01-01	1993-04-01	1993-05-01	1993-12-01	3
0	1980-01-01	1993-12-01	1994-01-01	1994-08-01	4

Видим, что с каждой итерацией размер обучающей выборки увеличивается (отсюда и название режима 'expand').

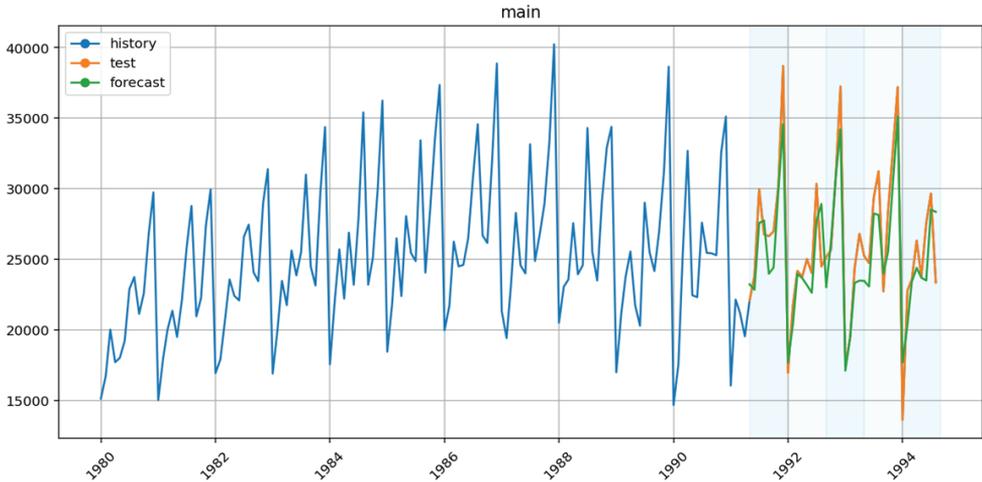
Визуализируем результаты перекрестной проверки.

```
# визуализируем результаты перекрестной проверки
plot_backtest(forecast_df, ts)
```



Вновь визуализируем результаты перекрестной проверки, но рассмотрим подробнее первую обучающую выборку.

```
# визуализируем результаты перекрестной проверки,
# посмотрим подробнее первую обучающую выборку
plot_backtest(forecast_df, ts, history_len=137)
```



Теперь выполним перекрестную проверку скользящим окном, опять выведем датафрейм с метриками по каждой тестовой выборке перекрестной проверки, датафрейм с прогнозами, датафрейм с временными метками обучающей и тестовой выборки и вновь визуализируем результаты перекрестной проверки.

```
# выполняем перекрестную проверку расширяющимся окном,
# возвращаем metrics_df - датафрейм с метриками
# по каждой тестовой выборке перекрестной проверки,
# forecast_df - датафрейм с прогнозами, fold_info_df -
# датафрейм с временными метками обучающей и тестовой
# выборки перекрестной проверки
metrics_df, forecast_df, fold_info_df = pipeline.backtest(
    mode='constant', ts=ts, metrics=[smape])
```

Смотрим датафрейм с метриками по каждой тестовой выборке перекрестной проверки.

```
# смотрим датафрейм с метриками по каждой
# тестовой выборке перекрестной проверки
metrics_df
```

	segment	SMAPE	fold_number
0	main	6.720895	0
0	main	6.492677	1
0	main	5.387386	2
0	main	6.869401	3
0	main	10.251426	4

Смотрим первые 5 строк датафрейма с прогнозами.

```
# смотрим первые 5 строк
# датафрейма с прогнозами
forecast_df.head()
```

segment	main	feature	target	fold_number
timestamp				
1991-05-01	23229.092865			0
1991-06-01	22837.450406			0
1991-07-01	27569.150843			0
1991-08-01	27753.810518			0
1991-09-01	23975.536490			0

Смотрим датафрейм с временными метками обучающей и тестовой выборки для каждой итерации перекрестной проверки.

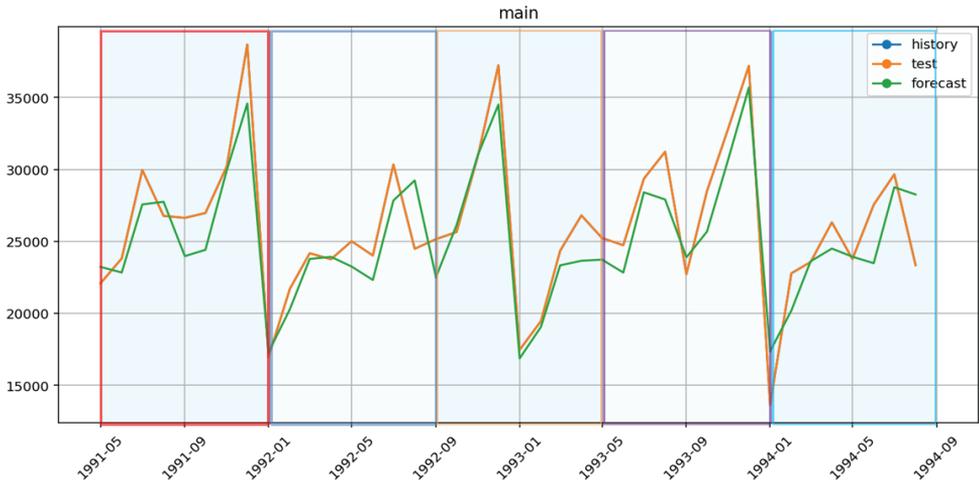
```
# смотрим датафрейм с временными метками
# обучающей и тестовой выборки
fold_info_df
```

	train_start_time	train_end_time	test_start_time	test_end_time	fold_number
0	1980-01-01	1991-04-01	1991-05-01	1991-12-01	0
0	1980-09-01	1991-12-01	1992-01-01	1992-08-01	1
0	1981-05-01	1992-08-01	1992-09-01	1993-04-01	2
0	1982-01-01	1993-04-01	1993-05-01	1993-12-01	3
0	1982-09-01	1993-12-01	1994-01-01	1994-08-01	4

Видим, что с каждой итерацией обучающая выборка охватывает более поздние наблюдения и имеет один и тот же размер (отсюда и название режима 'constant').

Визуализируем результаты перекрестной проверки.

```
# визуализируем результаты перекрестной проверки
plot_backtest(forecast_df, ts)
```



Теперь построим базовую модель Prophet без преобразований и конструирования признаков, прогнозирующую потребление электроэнергии по 4 сегментам.

```
# обучаем модель Prophet
model.fit(train_mult_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_mult_ts = train_mult_ts.make_future(mult_HORIZON)
```

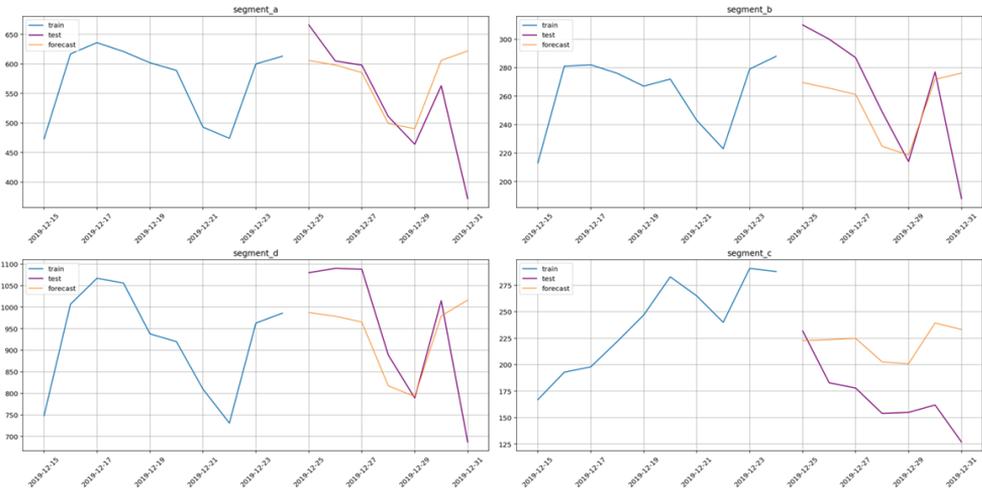
```
# получаем прогнозы
forecast_mult_ts = model.forecast(future_mult_ts)
forecast_mult_ts
```

segment	segment_a	segment_b	segment_c	segment_d
feature	target	target	target	target
timestamp				
2019-12-25	605.796691	269.520872	222.809160	987.488328
2019-12-26	598.216815	265.596568	223.666580	978.997418
2019-12-27	585.364256	261.201677	224.876972	965.388670
2019-12-28	499.184664	224.711458	202.787067	817.427426
2019-12-29	490.356447	218.613838	200.915058	792.657854
2019-12-30	606.022677	271.752397	239.416863	979.711150
2019-12-31	621.977284	276.181000	233.283487	1016.429698

```
# оцениваем качество прогнозов
smape(y_true=test_mult_ts, y_pred=forecast_mult_ts)
```

```
{'segment_a': 11.178856451812036,
 'segment_b': 12.548795664835707,
 'segment_d': 11.812767707492396,
 'segment_c': 28.290508131744424}
```

```
# визуализируем прогнозы, n_train_samples -
# n последних наблюдений в обучающей выборке
plot_forecast(forecast_mult_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```



Для более надежной оценки качества модели Prophet воспользуемся перекрестной проверкой расширяющимся окном.

Создадим список преобразований. В данном случае он будет пустым, мы не будем выполнять каких-либо преобразований или конструировать какие-либо признаки внутри цикла перекрестной проверки.

```
# не будем делать никаких преобразований
mult_transforms = []
```

Создаем конвейер для выполнения перекрестной проверки расширяющимся окном, передав в него модель, пустой список преобразований и горизонт прогнозирования.

```
# создаем конвейер
mult_pipeline = Pipeline(model=model,
                        transforms=mult_transforms,
                        horizon=mult_HORIZON)
```

Выполняем перекрестную проверку расширяющимся окном. С помощью `aggregate_metrics=True` вычислим для каждого ряда (сегмента) среднее значение метрики, отказавшись от вывода метрики по каждой итерации.

```
# выполняем перекрестную проверку расширяющимся окном,
# возвращаем metrics_mult_df - датафрейм с метриками
# по каждой тестовой выборке перекрестной проверки,
# forecast_mult_df - датафрейм с прогнозами, fold_info_mult_df -
# датафрейм с временными метками обучающей и тестовой
# выборки перекрестной проверки
metrics_mult_df, forecast_mult_df, fold_info_mult_df = mult_pipeline.backtest(
    n_folds=12, mode='expand', ts=mult_ts, metrics=[smape],
    aggregate_metrics=True)
```

Смотрим датафрейм с метриками по каждой тестовой выборке перекрестной проверки.

```
# смотрим датафрейм с метриками по каждой
# тестовой выборке перекрестной проверки
metrics_mult_df
```

	segment	SMAPE
0	segment_a	7.502665
1	segment_b	6.025290
2	segment_c	14.068599
3	segment_d	6.652540

Давайте взглянем на последние 20 строк датафрейма с прогнозами по каждому ряду для каждой итерации.

```
# смотрим последние 20 строк датафрейма
# с прогнозами для каждой итерации
forecast_mult_df.tail(20)
```

segment	segment_a		segment_b		segment_c		segment_d	
feature	fold_number	target	fold_number	target	fold_number	target	fold_number	target
timestamp								
2019-12-12	9	568.975554	9	252.833879	9	198.455254	9	940.714209
2019-12-13	9	556.598146	9	248.191683	9	198.704034	9	928.441921
2019-12-14	9	470.558422	9	211.573881	9	176.280301	9	780.367540
2019-12-15	9	462.141878	9	205.793204	9	174.850075	9	757.319321
2019-12-16	9	576.834518	9	258.439545	9	213.127396	9	941.895040
2019-12-17	9	592.529626	9	262.719851	9	206.477005	9	978.345249
2019-12-18	10	595.632369	10	263.307425	10	201.218234	10	974.275478
2019-12-19	10	588.264265	10	259.425007	10	201.370286	10	967.966219
2019-12-20	10	575.394455	10	254.780901	10	201.660141	10	954.431227
2019-12-21	10	489.384736	10	218.077306	10	179.268159	10	805.644875
2019-12-22	10	480.743244	10	212.197504	10	177.636498	10	781.944192
2019-12-23	10	596.191188	10	265.219074	10	215.661682	10	968.166568
2019-12-24	10	612.077640	10	269.505804	10	209.227126	10	1005.121455
2019-12-25	11	605.796691	11	269.520872	11	222.809160	11	987.488328
2019-12-26	11	598.216815	11	265.596568	11	223.666580	11	978.997418
2019-12-27	11	585.364256	11	261.201677	11	224.876972	11	965.388670
2019-12-28	11	499.184664	11	224.711458	11	202.787067	11	817.427426
2019-12-29	11	490.356447	11	218.613838	11	200.915058	11	792.657854
2019-12-30	11	606.022677	11	271.752397	11	239.416863	11	979.711150
2019-12-31	11	621.977284	11	276.181000	11	233.283487	11	1016.429698

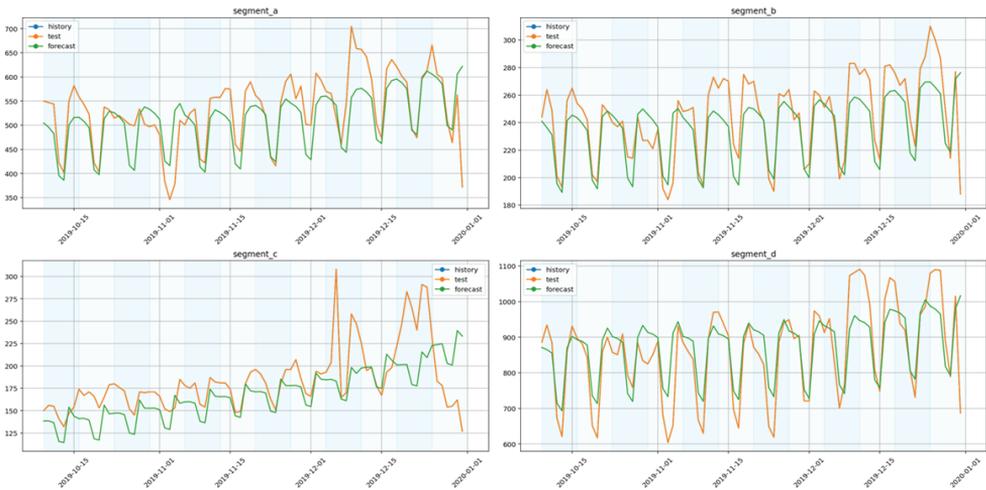
Теперь посмотрим датафрейм с временными метками обучающей и тестовой выборок.

```
# смотрим датафрейм с временными метками
# обучающей и тестовой выборки
fold_info_mult_df
```

	train_start_time	train_end_time	test_start_time	test_end_time	fold_number
0	2019-01-01	2019-10-08	2019-10-09	2019-10-15	0
0	2019-01-01	2019-10-15	2019-10-16	2019-10-22	1
0	2019-01-01	2019-10-22	2019-10-23	2019-10-29	2
0	2019-01-01	2019-10-29	2019-10-30	2019-11-05	3
0	2019-01-01	2019-11-05	2019-11-06	2019-11-12	4
0	2019-01-01	2019-11-12	2019-11-13	2019-11-19	5
0	2019-01-01	2019-11-19	2019-11-20	2019-11-26	6
0	2019-01-01	2019-11-26	2019-11-27	2019-12-03	7
0	2019-01-01	2019-12-03	2019-12-04	2019-12-10	8
0	2019-01-01	2019-12-10	2019-12-11	2019-12-17	9
0	2019-01-01	2019-12-17	2019-12-18	2019-12-24	10
0	2019-01-01	2019-12-24	2019-12-25	2019-12-31	11

Наконец, визуализируем результаты перекрестной проверки.

```
# визуализируем результаты перекрестной проверки
plot_backtest(forecast_mult_df, mult_ts)
```



Теперь выполним перекрестную проверку скользящим окном.

```

# выполняем перекрестную проверку скользящим окном,
# возвращаем metrics_mult_df - датафрейм с метриками
# по каждой тестовой выборке перекрестной проверки,
# forecast_mult_df - датафрейм с прогнозами, fold_info_mult_df -
# датафрейм с временными метками обучающей и тестовой
# выборки перекрестной проверки
metrics_mult_df, forecast_mult_df, fold_info_mult_df = mult_pipeline.backtest(
    n_folds=12, mode='constant', ts=mult_ts, metrics=[smape],
    aggregate_metrics=True)

```

Смотрим датафрейм с метриками по каждой тестовой выборке перекрестной проверки.

```

# смотрим датафрейм с метриками по каждой
# тестовой выборке перекрестной проверки
metrics_mult_df

```

	<b>segment</b>	<b>SMAPE</b>
<b>0</b>	segment_a	7.168168
<b>1</b>	segment_b	5.884772
<b>2</b>	segment_c	14.578508
<b>3</b>	segment_d	6.706074

Давайте взглянем на последние 20 строк датафрейма с прогнозами по каждому ряду для каждой итерации.

```

# смотрим последние 20 строк датафрейма
# с прогнозами для каждой итерации
forecast_mult_df.tail(20)

```

segment	segment_a		segment_b		segment_c		segment_d	
feature	fold_number	target	fold_number	target	fold_number	target	fold_number	target
timestamp								
2019-12-12	9	571.182287	9	254.527063	9	196.692131	9	943.326571
2019-12-13	9	559.080274	9	248.999004	9	200.062512	9	930.297014
2019-12-14	9	471.742384	9	210.945896	9	176.413557	9	771.981864
2019-12-15	9	464.535119	9	205.092952	9	174.560930	9	747.825495
2019-12-16	9	583.783998	9	260.589799	9	216.299778	9	945.305068
2019-12-17	9	602.047030	9	265.915585	9	208.058322	9	989.238990
2019-12-18	10	601.531915	10	265.227021	10	200.088799	10	977.521414
2019-12-19	10	592.621820	10	261.914384	10	199.547509	10	970.733675
2019-12-20	10	582.634653	10	257.326475	10	203.476592	10	959.166631
2019-12-21	10	492.595487	10	218.162897	10	179.301951	10	796.836922
2019-12-22	10	484.600395	10	211.974022	10	177.132681	10	771.026601
2019-12-23	10	603.969380	10	267.783960	10	218.872343	10	970.380160
2019-12-24	10	623.114260	10	273.304951	10	210.682413	10	1015.704355
2019-12-25	11	611.033347	11	272.104515	11	221.047076	11	992.046930
2019-12-26	11	602.320661	11	268.880557	11	221.449511	11	983.542843
2019-12-27	11	591.830567	11	264.406114	11	226.276659	11	970.686064
2019-12-28	11	501.641755	11	225.556271	11	202.538275	11	809.134410
2019-12-29	11	493.194522	11	219.079737	11	199.956893	11	780.513606
2019-12-30	11	613.408795	11	275.077909	11	242.363817	11	982.902800
2019-12-31	11	631.236523	11	280.323407	11	234.413817	11	1023.443525

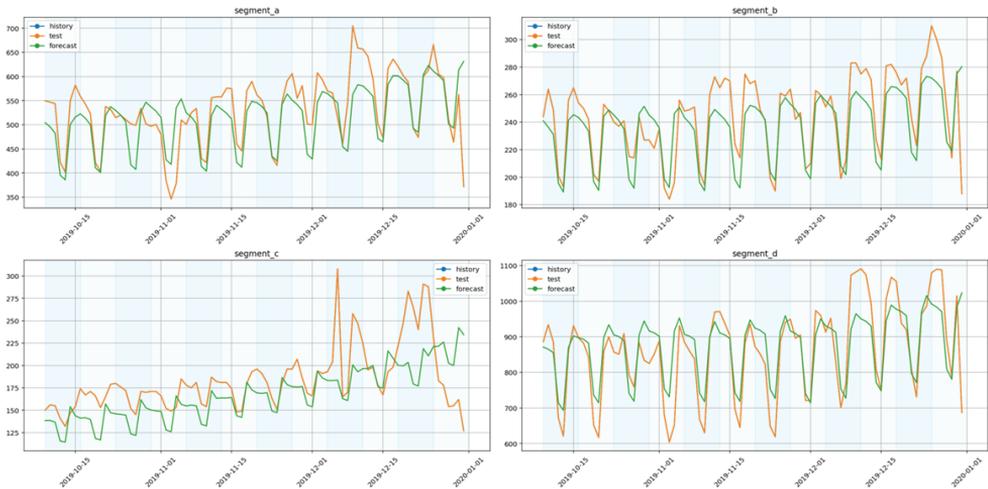
Теперь посмотрим датафрейм с временными метками обучающей и тестовой выборки.

```
# смотрим датафрейм с временными метками
# обучающей и тестовой выборки
fold_info_mult_df
```

	train_start_time	train_end_time	test_start_time	test_end_time	fold_number
0	2019-01-01	2019-10-08	2019-10-09	2019-10-15	0
0	2019-01-08	2019-10-15	2019-10-16	2019-10-22	1
0	2019-01-15	2019-10-22	2019-10-23	2019-10-29	2
0	2019-01-22	2019-10-29	2019-10-30	2019-11-05	3
0	2019-01-29	2019-11-05	2019-11-06	2019-11-12	4
0	2019-02-05	2019-11-12	2019-11-13	2019-11-19	5
0	2019-02-12	2019-11-19	2019-11-20	2019-11-26	6
0	2019-02-19	2019-11-26	2019-11-27	2019-12-03	7
0	2019-02-26	2019-12-03	2019-12-04	2019-12-10	8
0	2019-03-05	2019-12-10	2019-12-11	2019-12-17	9
0	2019-03-12	2019-12-17	2019-12-18	2019-12-24	10
0	2019-03-19	2019-12-24	2019-12-25	2019-12-31	11

Наконец, визуализируем результаты перекрестной проверки.

```
# визуализируем результаты перекрестной проверки
plot_backtest(forecast_mult_df, mult_ts)
```



## 2.11. Модель CatBoost

Давайте спрогнозируем продажи австралийского вина с помощью модели CatBoost. Для этого можно воспользоваться классами `CatBoostPerSegmentModel` и `CatBoostMultiSegmentModel`. Они являются обычными классами-обертками над классом `CatboostRegressor`. Разница заключается в том, что класс `CatBoostPerSegmentModel` обучает отдельную модель для каждого сегмента, а класс `CatBoostMultiSegmentModel` обучает одну модель для всех сегментов.

Модель является сложной, требует настройки гиперпараметров, поэтому нам потребуется обучающая выборка для обучения модели, валидационная выборка для настройки гиперпараметров и тестовая выборка для итоговой оценки качества модели.

Давайте выделим обучающую и валидационную выборки с учетом временной структуры. Валидационная выборка должна иметь ту же длину, что и тестовая выборка, обе должны быть равны длине горизонта прогнозирования, в нашем случае – 8 дням.

```
# разбиваем набор на обучающую и валидационную выборки
# с учетом временной структуры
train_ts, valid_ts = ts.train_test_split(train_start='1980-01-01',
                                         train_end='1993-04-01',
                                         test_start='1993-05-01',
                                         test_end='1993-12-01')
```

```
# смотрим обучающую выборку
train_ts
```

<b>segment</b>	<b>main</b>
<b>feature</b>	<b>target</b>
<b>timestamp</b>	
1980-01-01	15136
1980-02-01	16733
1980-03-01	20016
1980-04-01	17708
1980-05-01	18019
...	...
1992-12-01	37240
1993-01-01	17466
1993-02-01	19463
1993-03-01	24352
1993-04-01	26805

*# смотрим валидационную выборку*  
valid\_ts

<b>segment</b>	<b>main</b>
<b>feature</b>	<b>target</b>
<b>timestamp</b>	
1993-05-01	25236
1993-06-01	24735
1993-07-01	29356
1993-08-01	31234
1993-09-01	22724
1993-10-01	28496
1993-11-01	32857
1993-12-01	37198

На этот раз мы создадим новые признаки.

Нам понадобятся следующие классы-трансформеры:

- класс `MeanTransform` для вычисления скользящих средних по заданному окну;
- класс `LagTransform` для генерации лагов.

Класс `MeanTransform` позволяет создать скользящие средние. Ширину окна обычно берут не меньше горизонта прогнозирования. Если задать ширину окна равной горизонту, в последнем наблюдении валидационной/тестовой выборки появится пропуск, который будет заменен нулем. Если задать ширину окна меньше горизонта, в валидационной/тестовой выборке появятся пропуски, которые будут заменены нулями. Ранее скользящее среднее вычислялось с лагом 1, теперь лаг 1 не используется. Способ вычисления скользящих средних

был изменен в версии 1.6.3. Если необходим старый способ вычисления скользящих средних, можно воспользоваться классом `OldMeanTransform`. С помощью параметра `in_column` мы указываем зависимую переменную 'target', на основе которой создаем скользящие статистики. С помощью параметра `out_column` мы задаем имя создаваемой переменной.

Кроме скользящего среднего библиотека ETNA позволяет вычислить и другие скользящие статистики:

- скользящие минимальные значения (класс `MinTransform`);
- скользящие максимальные значения (класс `MaxTransform`);
- скользящие средние абсолютные отклонения (класс `MADTransform`);
- скользящие стандартные отклонения (класс `StdTransform`);
- скользящие медианные значения (класс `MedianTransform`);
- скользящие квантильные значения (класс `QuantileTransform`);
- скользящие суммарные значения (класс `SumTransform`);
- скользящие разности между максимальным и минимальным значениями окна (класс `MinMaxDifferenceTransform`).

Как правило, для градиентного бустинга лучше всего работают скользящие средние, скользящие стандартные отклонения, скользящие средние / медианные абсолютные отклонения.

Класс `LagTransform` позволяет создать лаги. С помощью параметра `in_column` мы указываем зависимую переменную 'target', на основе которой создаем лаги. Лаги вида  $L_{t-k}$  создаем так, чтобы  $k$  был не меньше горизонта прогнозирования. С помощью параметра `lags` можно просто указать список лагов. Если лаги будут меньше горизонта прогнозирования, в валидационной/тестовой выборке такие лаги будут содержать пропуски. У нас горизонт прогнозирования равен 8, поэтому минимальный порядок лага должен быть равен 8.

Сейчас мы продемонстрируем создание лагов с порядком от 6-го по 9-й и вычисление скользящих средних с шириной окна 8, 9, 10 и 11 обоими способами для обучающей и валидационной выборок.

```
# создаем экземпляры класса MeanTransform для вычисления
# скользящего среднего разной ширины,
# с помощью in_column задаем переменную, на основе
# которой вычисляем скользящие средние
mean7 = MeanTransform(in_column='target',
                      window=7,
                      out_column='mean_07')
mean8 = MeanTransform(in_column='target',
                      window=8,
                      out_column='mean_08')
mean9 = MeanTransform(in_column='target',
                      window=9,
                      out_column='mean_09')
mean10 = MeanTransform(in_column='target',
                       window=10,
                       out_column='mean_10')
mean11 = MeanTransform(in_column='target',
                       window=11,
                       out_column='mean_11')
```

```
# создаем экземпляры класса OldMeanTransform для вычисления
# скользящего среднего разной ширины,
# с помощью in_column задаем переменную, на основе
# которой вычисляем скользящие средние
```

```
mean7_old = OldMeanTransform(in_column='target',
                             window=7,
                             out_column='mean_07_old')
mean8_old = OldMeanTransform(in_column='target',
                             window=8,
                             out_column='mean_08_old')
mean9_old = OldMeanTransform(in_column='target',
                             window=9,
                             out_column='mean_09_old')
mean10_old = OldMeanTransform(in_column='target',
                              window=10,
                              out_column='mean_10_old')
mean11_old = OldMeanTransform(in_column='target',
                              window=11,
                              out_column='mean_11_old')
```

```
# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 6 по 9,
# порядок лагов не должен быть меньше длины горизонта
```

```
lags = LagTransform(in_column='target',
                   lags=list(range(6, 10, 1)),
                   out_column='lag')
```

```
# добавляем скользящие статистики в обучающую выборку
```

```
train_ts.fit_transform([mean7, mean8, mean9, mean10, mean11,
                       mean7_old, mean8_old, mean9_old,
                       mean10_old, mean11_old, lags])
```

```
train_ts.head()
```

segment	main														
feature	lag_6	lag_7	lag_8	lag_9	mean_07	mean_07_old	mean_08	mean_08_old	mean_09	mean_09_old	mean_10	mean_10_old	mean_11	mean_11_old	target
1980-01-01	NaN	NaN	NaN	NaN	15136.00	0.00	15136.00	0.00	15136.00	0.00	15136.00	0.00	15136.00	0.00	15136
1980-02-01	NaN	NaN	NaN	NaN	15934.50	15136.00	15934.50	15136.00	15934.50	15136.00	15934.50	15136.00	15934.50	15136.00	16733
1980-03-01	NaN	NaN	NaN	NaN	17295.00	15934.50	17295.00	15934.50	17295.00	15934.50	17295.00	15934.50	17295.00	15934.50	20016
1980-04-01	NaN	NaN	NaN	NaN	17398.25	17295.00	17398.25	17295.00	17398.25	17295.00	17398.25	17295.00	17398.25	17295.00	17708
1980-05-01	NaN	NaN	NaN	NaN	17522.40	17398.25	17522.40	17398.25	17522.40	17398.25	17522.40	17398.25	17522.40	17398.25	18019

```
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
```

```
future_ts = train_ts.make_future(HORIZON)
future_ts
```

segment	main														
feature	lag_6	lag_7	lag_8	lag_9	mean_07	mean_07_old	mean_08	mean_08_old	mean_09	mean_09_old	mean_10	mean_10_old	mean_11	mean_11_old	target
timestamp															
1993-05-01	30923.0	25650.0	25156.0	24488.0	26041.5	25985.571429	25985.571429	25881.875000	25881.875000	25727.000000	25727.000000	26188.800000	26188.800000	25991.545455	NaN
1993-06-01	37240.0	30923.0	25650.0	25156.0	25065.2	26041.500000	26041.500000	25985.571429	25985.571429	25881.875000	25881.875000	25727.000000	25727.000000	26188.800000	NaN
1993-07-01	17466.0	37240.0	30923.0	25650.0	22021.5	25065.200000	25065.200000	26041.500000	26041.500000	25985.571429	25985.571429	25881.875000	25881.875000	25727.000000	NaN
1993-08-01	19463.0	17466.0	37240.0	30923.0	23540.0	22021.500000	22021.500000	25065.200000	25065.200000	26041.500000	26041.500000	25985.571429	25985.571429	25881.875000	NaN
1993-09-01	24352.0	19463.0	17466.0	37240.0	25578.5	23540.000000	23540.000000	22021.500000	22021.500000	25065.200000	25065.200000	26041.500000	26041.500000	25985.571429	NaN
1993-10-01	26805.0	24352.0	19463.0	17466.0	26805.0	25578.500000	25578.500000	23540.000000	23540.000000	22021.500000	22021.500000	25065.200000	25065.200000	26041.500000	NaN
1993-11-01	NaN	26805.0	24352.0	19463.0	0.0	26805.000000	26805.000000	25578.500000	25578.500000	23540.000000	23540.000000	22021.500000	22021.500000	25065.200000	NaN
1993-12-01	NaN	NaN	26805.0	24352.0	0.0	0.000000	0.000000	26805.000000	26805.000000	25578.500000	25578.500000	23540.000000	23540.000000	22021.500000	NaN

Видим, что скользящие средние по новому способу вычисляются в обучающей выборке без лага. Новый способ дает для скользящего среднего с шириной окна, равной горизонту прогнозирования, пропуск в последнем значении валидационной выборки, который заменяется нулем (выделено красной рамкой). Скользящие средние с шириной окна меньше горизонта прогнозирования получают пропуски в валидационной выборке, которые заменяются нулями (выделено синими рамками). Лаги, у которых порядок меньше горизонта, в валидационной выборке получают пропуски (выделено фиолетовыми рамками).

Для особо любопытных читателей ниже на игрушечном примере подробно продемонстрировано вычисление скользящих средних в соответствии со старым и новым способами.

```
# загружаем игрушечные данные, сразу выполнив парсинг дат
toy_data = pd.read_csv('Data/example.csv', sep='\t',
                      parse_dates=['date'],
                      date_parser=lambda col: pd.to_datetime(
                          col, format='%d.%m.%Y'))

# задаем обязательные столбцы target, timestamp и segment
toy_data.rename(columns={'date': 'timestamp', 'sales': 'target'},
                inplace=True)
toy_data['segment'] = 'main'
toy_data
```

	timestamp	target	segment
0	2018-01-09	2400	main
1	2018-01-10	2800	main
2	2018-01-11	2500	main
3	2018-01-12	2890	main
4	2018-01-13	2610	main
5	2018-01-14	2500	main
6	2018-01-15	2750	main
7	2018-01-16	2700	main
8	2018-01-17	2250	main
9	2018-01-18	2350	main
10	2018-01-19	2550	main
11	2018-01-20	3000	main

```

# превратим датафрейм с плоским индексом
# в датафрейм с мультииндексом
toy_data = TSDataset.to_dataset(toy_data)
# превращаем датафрейм в объект TSDataset,
# задав частоту временного ряда
toy_ts = TSDataset(toy_data, freq='D')
# задаем горизонт
h = 4
# разбиваем на обучающую и тестовую выборки
toy_train_ts, toy_test_ts = toy_ts.train_test_split(test_size=h)
# создаем экземпляр класса MeanTransform
mean4 = MeanTransform(in_column='target',
                      window=4,
                      out_column='mean_04')
# создаем экземпляр класса OldMeanTransform
mean4_old = OldMeanTransform(in_column='target',
                             window=4,
                             out_column='mean_04_old')

# вычисляем скользящее среднее шириной 4 с помощью старого
# и нового способов в обучающей выборке
toy_train_ts.fit_transform([mean4, mean4_old])
toy_train_ts

```

segment	main		
feature	mean_04	mean_04_old	target
timestamp			
2018-01-09	2400.000000	0.000000	2400
2018-01-10	2600.000000	2400.000000	2800
2018-01-11	2566.666667	2600.000000	2500
2018-01-12	2647.500000	2566.666667	2890
2018-01-13	2700.000000	2647.500000	2610
2018-01-14	2625.000000	2700.000000	2500
2018-01-15	2687.500000	2625.000000	2750
2018-01-16	2640.000000	2687.500000	2700

```

# вычисляем скользящее среднее шириной 4 с помощью
# старого и нового способов в тестовой выборке
toy_future_ts = toy_train_ts.make_future(h)
toy_future_ts

```

segment	main		
feature	mean_04	mean_04_old	target
timestamp			
2018-01-17	2650.0	2640.0	NaN
2018-01-18	2725.0	2650.0	NaN
2018-01-19	2700.0	2725.0	NaN
2018-01-20	0.0	2700.0	NaN

```
# вычисление скользящих средних по новому способу
```

```
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700) / 2)
print(2700 / 1)
print(np.NaN)
```

```
2400.0
2600.0
2566.6666666666665
2647.5
2700.0
2625.0
2687.5
2640.0
2650.0
2725.0
2700.0
nan
```

```
# вычисление скользящих средних по старому способу
```

```
print(np.NaN)
print(2400 / 1)
print((2400 + 2800) / 2)
print((2400 + 2800 + 2500) / 3)
print((2400 + 2800 + 2500 + 2890) / 4)
print((2800 + 2500 + 2890 + 2610) / 4)
print((2500 + 2890 + 2610 + 2500) / 4)
print((2890 + 2610 + 2500 + 2750) / 4)
print((2610 + 2500 + 2750 + 2700) / 4)
print((2500 + 2750 + 2700) / 3)
print((2750 + 2700) / 2)
print(2700 / 1)
```

```
nan
2400.0
2600.0
2566.6666666666665
2647.5
2700.0
2625.0
2687.5
2640.0
2650.0
2725.0
2700.0
```

Идея проста: как только наше скользящее окно выходит за пределы обучающей выборки, происходит уменьшение ширины окна. Получается, что при

вычислении скользящих средних для более поздних моментов времени в тестовой выборке мы используем средние, взятые по более поздним наблюдениям в обучающей выборке. При вычислении скользящих средних для тестовой выборки обоими способами *мы ни разу не использовали* значения продаж в тестовой выборке.

Давайте вернемся к продажам австралийского вина и заново создадим обучающую и валидационную выборки. Для нашей модели мы вычислим скользящее среднее с шириной окна 8 и сгенерируем лаги порядка с 8-го по 23-й.

```
# разбиваем набор на обучающую и валидационную выборки
# с учетом временной структуры
train_ts, valid_ts = ts.train_test_split(train_start='1980-01-01',
                                       train_end='1993-04-01',
                                       test_start='1993-05-01',
                                       test_end='1993-12-01')

# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 8 до 23,
# порядок лагов не должен быть меньше длины горизонта
lags = LagTransform(in_column='target',
                   lags=list(range(8, 24, 1)),
                   out_column='lag')

# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего с шириной окна 8
mean8 = MeanTransform(in_column='target',
                     window=8,
                     out_column='mean8')
```

Для применения преобразований и создания признаков используем метод `.fit_transform()`.

```
# добавляем лаги и mean8 в обучающую выборку
train_ts.fit_transform([lags, mean8])
train_ts
```

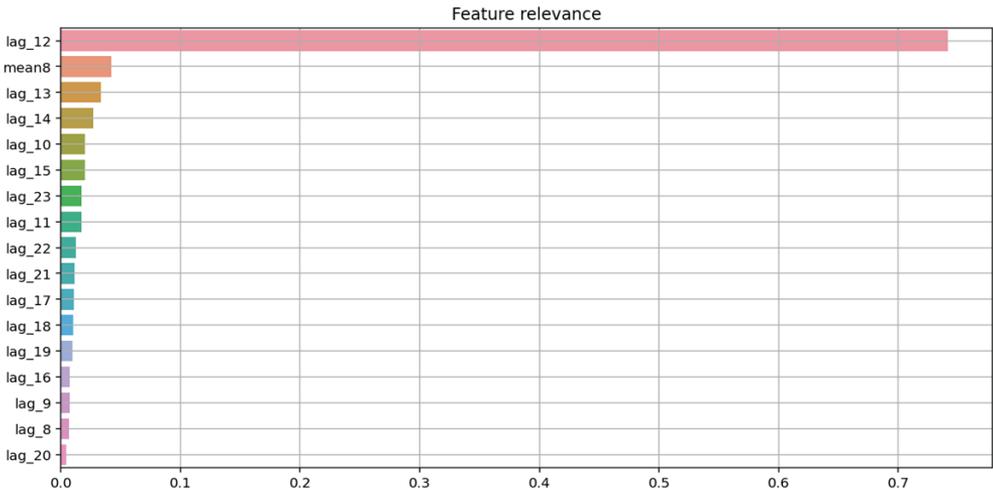
segment	main																target	
feature	lag_10	lag_11	lag_12	lag_13	lag_14	lag_15	lag_16	lag_17	lag_18	lag_19	lag_20	lag_21	lag_22	lag_23	lag_8	lag_9	mean8	target
1980-01-01	NaN	15136.000	15136															
1980-02-01	NaN	15934.500	16733															
1980-03-01	NaN	17295.000	20016															
1980-04-01	NaN	17398.250	17708															
1980-05-01	NaN	17522.400	18019															
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1992-12-01	21697.0	16974.0	38687.0	30207.0	26972.0	26635.0	26773.0	29961.0	23816.0	22084.0	19543.0	21198.0	22146.0	16052.0	23757.0	24179.0	27854.250	37240
1993-01-01	24179.0	21697.0	16974.0	38687.0	30207.0	26972.0	26635.0	26773.0	29961.0	23816.0	22084.0	19543.0	21198.0	22146.0	25013.0	23757.0	26910.875	17466
1993-02-01	23757.0	24179.0	21697.0	16974.0	38687.0	30207.0	26972.0	26635.0	26773.0	29961.0	23816.0	22084.0	19543.0	21198.0	24019.0	25013.0	26341.375	19463
1993-03-01	25013.0	23757.0	24179.0	21697.0	16974.0	38687.0	30207.0	26972.0	26635.0	26773.0	29961.0	23816.0	22084.0	19543.0	30345.0	24019.0	25592.250	24352
1993-04-01	24019.0	25013.0	23757.0	24179.0	21697.0	16974.0	38687.0	30207.0	26972.0	26635.0	26773.0	29961.0	23816.0	22084.0	24488.0	30345.0	25881.875	26805

С помощью функций `plot_feature_relevance()` и `ModelRelevanceTable()` выведем важности признаков на основе усредненного уменьшения меры Джини (мы воспользуемся моделью случайного леса, реализованной в классе `RandomForestRegressor`). Функция `plot_feature_relevance()` имеет следующие параметры:

- `ts` – обучающий временной ряд;
- `relevance_table` – функция, вычисляющая важности признаков и приводящая их к табличному виду;
- `normalized` – нормализация, приводящая значения важностей признаков в диапазон от 0 до 1;
- `relevance_aggregation_mode` – способ агрегации значений важностей;
- `top_k` – выводит топ  $k$  важных признаков;
- `columns_num` – количество колонок для графика;
- `segments` – задает список сегментов (список строковых значений).

Давайте выведем график топ-20 важных признаков.

```
# выведем график топ-20 важных признаков
plot_feature_relevance(
    ts=train_ts,
    relevance_table=ModelRelevanceTable(),
    normalized=True,
    top_k=20,
    relevance_params=dict(
        model=RandomForestRegressor(n_estimators=100)
    )
)
```



Обучаем модель CatBoost для всех сегментов и получаем прогнозы.

```
# создаем экземпляр класса CatBoostModelMultiSegment
model = CatBoostModelMultiSegment(loss_function='MAE',
    n_estimators=600,
    learning_rate=0.05,
    depth=9,
    random_seed=42)
```



Мы возьмем лаги с порядком от 8 до 12, создав новый экземпляр класса `LagTransform`.

```
# уменьшаем количество лагов до 12
lags2 = LagTransform(in_column='target',
                    lags=list(range(8, 13, 1)),
                    out_column='lag')
```

С помощью класса `Pipeline` создаем и обучаем конвейер с моделью и списком преобразований, в который передаем новый экземпляр класса `LagTransform` и экземпляр класса `MeanTransform`.

```
# создаем конвейер с моделью и списком преобразований
pipeline = Pipeline(model=model,
                   transforms=[lags2, mean8],
                   horizon=HORIZON)

# обучаем конвейер
pipeline.fit(train_ts)
```

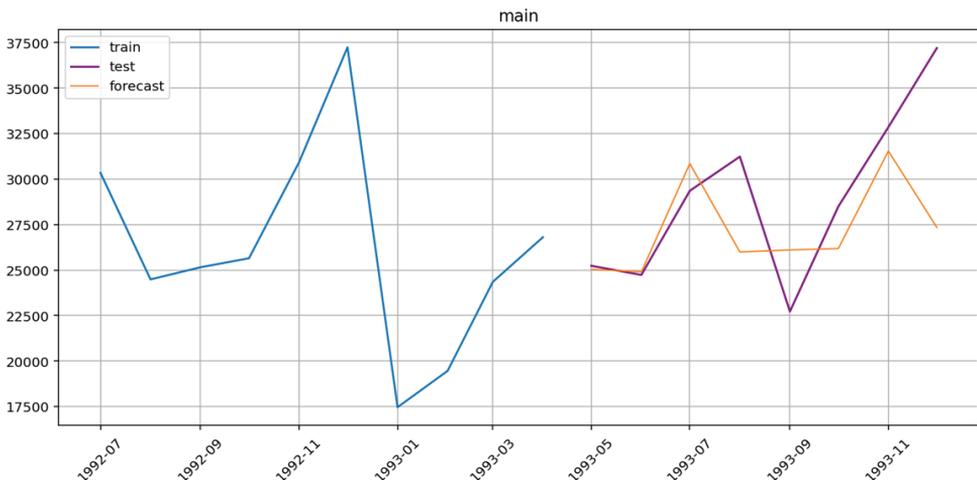
С помощью класса `Pipeline` также получаем прогнозы для валидационной выборки.

```
# получаем прогнозы
forecast_ts = pipeline.forecast()
```

Теперь оцениваем качество прогнозов на валидационной выборке и визуализируем их.

```
# оцениваем качество прогнозов на валидационной выборке
print(smape(y_true=valid_ts, y_pred=forecast_ts))
# визуализируем прогнозы
plot_forecast(forecast_ts, test_ts,
             train_ts, n_train_samples=10)
```

```
{'main': 10.204354075542287}
```



Модель CatBoost с меньшим количеством лагов имеет большую ошибку.

Удобство конструкции с использованием класса Pipeline заключается не только в краткости, но и в том, что нам не нужно выполнять обратные преобразования обучающей выборки для правильной визуализации прогнозов (например, если бы мы применили логарифмирование зависимой переменной, нам бы потребовалось экспоненцирование прогнозов, полученных с помощью модели, которая обучалась на прологарифмированной зависимой переменной). Недостаток заключается в невозможности взглянуть на преобразованный набор данных (все преобразования происходят под капотом метода `.fit()` экземпляра класса Pipeline).

Ниже приведен пример написания и использования собственной функции `etna_validation_optimize()` для настройки гиперпараметра – набора преобразований/признаков с помощью валидационной выборки.

```
# пишем функцию, выполняющую подбор гиперпараметра -
# набора преобразований/признаков
# с помощью валидационной выборки
def etna_validation_optimize(ts,
                             train_start,
                             train_end,
                             valid_start,
                             valid_end,
                             model,
                             horizon,
                             transfrms,
                             metrics):
    """
    Выполняет подбор гиперпараметра - набора преобразований/
    признаков с помощью валидационной выборке.

    Параметры
    -----
    ts: pandas.DataFrame
        Мультииндексный pandas.DataFrame в формате ETNA
        (объект TSDataset), содержащий один или
        несколько временных рядов.
    train_start, str
        Стартовая точка обучающей выборки
    train_end, str
        Конечная точка обучающей выборки
    valid_start, str
        Стартовая точка проверочной выборки
    valid_end, str
        Конечная точка проверочной выборки
    model: instance of class etna.models
        Модель прогнозирования.
    horizon: int
        Горизонт прогнозирования.
    transfrms: list
        Список преобразований/признаков.
    metrics: instance of class etna.metrics
        Метрика качества.
    """
```

```

# инициализируем наилучшее значение метрики
# положительной бесконечностью
best_score = np.inf

# с помощью цикла for
for trans in transforms:
    # разбиваем набор на обучающую и валидационную выборки
    # с учетом временной структуры
    train_ts, valid_ts = ts.train_test_split(train_start=train_start,
                                             train_end=train_end,
                                             test_start=valid_start,
                                             test_end=valid_end)

    # создаем конвейер с моделью и списком
    # преобразований/признаков
    pipe = Pipeline(
        model=model,
        transforms=trans,
        horizon=horizon)

    # обучаем конвейер
    pipe.fit(train_ts)
    # получаем прогнозы
    forecast_ts = pipe.forecast()
    # оцениваем качество прогнозов на валидационной выборке
    metrics_score = metrics(y_true=valid_ts,
                           y_pred=forecast_ts).get('main')
    print(f"trans:\n{trans}")
    print(f"{metrics.__class__.__name__}: {metrics_score}\n")

    # если получаем минимальное усредненное значение,
    # сохраняем его и наилучший набор преобразований/признаков
    if metrics_score < best_score:
        best_score = metrics_score
        best_parameters = {'trans': trans}

# печатаем наилучший набор преобразований/признаков
# и наилучшее значение метрики
print(f"Наилучший набор преобразований/признаков:\n{best_parameters}\n")
print(f"Лучшее значение {metrics.__class__.__name__}: {best_score:.4f}\n")
# выполняем подбор гиперпараметра – набора преобразований/
# признаков с помощью валидационной выборки
etna_validation_optimize(ts,
                        train_start='1980-01-01',
                        train_end='1993-04-01',
                        valid_start='1993-05-01',
                        valid_end='1993-12-01',
                        model=model,
                        horizon=HORIZON,
                        transforms=[[lags, mean8], [lags2, mean8]],
                        metrics=SMAPE())

trans:
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, season-
ality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', )]
SMAPE: 7.955966593676093

```

```

trans:
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12], out_column = 'lag', ), MeanT-
ransform(in_column = 'target', window = 8, seasonality = 1, alpha = 1, min_periods = 1, fillna
= 0, out_column = 'mean8', )]
SMAPE: 10.204354075542287

```

Наилучший набор преобразований/признаков:

```

{'trans': [LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window =
8, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', )]}

```

Лучшее значение SMAPE: 7.9560

С целью получения более надежных оценок при подборе гиперпараметров выполним перекрестную проверку обеих моделей (воспользуемся перекрестной проверкой расширяющимся окном).

Давайте выделим обучающую и тестовую выборки с учетом временной структуры. Тестовая выборка содержит последние 8 дней исходного набора. Сама перекрестная проверка запускается на обучающей выборке. Обратите внимание, что применение перекрестной проверки не отменяет откладывание тестовой выборки. Проверочные выборки перекрестной проверки используются для настройки гиперпараметров, а тестовая выборка – для итоговой оценки модели. Признаки создаются, а преобразования применяются заново на каждой итерации перекрестной проверки.

*# создаем список процедур*

```
transforms = [lags, mean8]
```

*# создаем конвейер*

```
pipeline = Pipeline(model=model,
                    transforms=transforms,
                    horizon=HORIZON)
```

*# разбиваем набор на обучающую и тестовую выборки*

*# с учетом временной структуры*

```
train_ts, test_ts = ts.train_test_split(train_start='1980-01-01',
                                       train_end='1993-12-01',
                                       test_start='1994-01-01',
                                       test_end='1994-08-01')
```

*# выполняем перекрестную проверку расширяющимся окном,*

*# возвращаем metrics\_df - датафрейм с метриками*

*# по каждой проверочной выборке перекрестной проверки,*

*# forecast\_df - датафрейм с прогнозами, fold\_info\_df -*

*# датафрейм с временными метками обучающей и проверочной*

*# выборки перекрестной проверки*

```
metrics_df, forecast_df, fold_info_df = pipeline.backtest(
    mode='expand', ts=train_ts, metrics=[smape])
```

*# смотрим датафрейм с метриками по каждой*

*# тестовой выборке перекрестной проверки*

```
metrics_df
```

	segment	SMAPE	fold_number
0	main	13.524289	0
0	main	8.387370	1
0	main	6.158660	2
0	main	7.704558	3
0	main	7.955967	4

```
# вычислим среднее значение и стандартное отклонение SMAPE
print(metrics_df['SMAPE'].mean())
print(metrics_df['SMAPE'].std())

8.746168790146212
2.8001672699916154

# создаем список процедур
transforms2 = [lags, mean8]

# создаем конвейер
pipeline2 = Pipeline(model=model,
                    transforms=transforms2,
                    horizon=HORIZON)

# разбиваем набор на обучающую и тестовую выборки
# с учетом временной структуры
train_ts, test_ts = ts.train_test_split(train_start='1980-01-01',
                                       train_end='1993-12-01',
                                       test_start='1994-01-01',
                                       test_end='1994-08-01')

# выполняем перекрестную проверку расширяющимся окном,
# возвращаем metrics_df2 - датафрейм с метриками
# по каждой проверочной выборке перекрестной проверки,
# forecast_df2 - датафрейм с прогнозами, fold_info_df2 -
# датафрейм с временными метками обучающей и проверочной
# выборки перекрестной проверки
metrics_df2, forecast_df2, fold_info_df2 = pipeline2.backtest(
    mode='expand', ts=train_ts, metrics=[smape])

# смотрим датафрейм с метриками по каждой
# тестовой выборке перекрестной проверки
metrics_df2
```

	segment	SMAPE	fold_number
0	main	16.109081	0
0	main	11.264537	1
0	main	6.455346	2
0	main	5.699620	3
0	main	10.204354	4

```
# вычислим среднее значение и стандартное отклонение SMAPE
print(metrics_df2['SMAPE'].mean())
print(metrics_df2['SMAPE'].std())

9.946587571595813
4.183457137940414
```

Результаты перекрестной проверки также показывают, что модель CatBoost с меньшим количеством лагов имеет бóльшую ошибку.

Ниже приведен пример написания и использования собственной функции `etna_cv_optimize()` для настройки гиперпараметра – набора преобразований/признаков с помощью перекрестной проверки. Функция `etna_cv_optimize()` находит набор преобразований/признаков, дающий наилучшее значение метрики по итогам перекрестной проверки, запущенной на обучающей выборке, и затем обучает конвейер с наилучшим набором на всей обучающей выборке и вычисляет метрику качества на тестовой выборке.

```
# пишем функцию, выполняющую подбор гиперпараметра – набора
# преобразований/признаков с помощью перекрестной проверки
def etna_cv_optimize(ts, model, horizon, transfrms, n_folds, mode, metrics,
                    refit=True, n_train_samples=10):
    """
    Выполняет подбор гиперпараметра – набора преобразований/
    признаков с помощью перекрестной проверки.

    Параметры
    -----
    ts: pandas.DataFrame
        Мультииндексный pandas.DataFrame в формате ETNA
        (объект TSDataset), содержащий один или
        несколько временных рядов.
    model: instance of class etna.models
        Модель прогнозирования.
    horizon: int
        Горизонт прогнозирования.
    transfrms: list
        Список преобразований/признаков.
    n_folds, int
        Количество тестовых выборок перекрестной проверки.
    mode: str
        Тип перекрестной проверки.
    metrics: instance of class etna.metrics
        Метрика качества.
    refit: bool
        Нужно ли строить наилучшую модель на всей обучающей выборке.
    n_train_sample: int
        n последних наблюдений обучающей выборки
        на графике прогнозов.
    """
    # разбиваем набор на обучающую и тестовую выборки
    # с учетом временной структуры, размер тестовой
    # выборки задаем равным горизонту
    train_ts, test_ts = ts.train_test_split(test_size=horizon)

    # инициализируем наилучшее значение метрики
    # положительной бесконечностью
    best_score = np.inf

    # с помощью цикла for
    for trans in transfrms:
        # создаем конвейер с моделью и списком преобразований
```

```

pipe = Pipeline(
    model=model,
    transforms=trans,
    horizon=horizon)

# находим метрики моделей по сегменту/сегментам
# по итогам перекрестной проверки
df_metrics, _, _ = pipe.backtest(mode=mode,
                                n_folds=n_folds,
                                ts=train_ts,
                                metrics=[metrics],
                                aggregate_metrics=False,
                                joblib_params=dict(verbose=0))

# вычисляем значение метрики, усредненное по тестовым выборкам
metrics_mean = df_metrics[metrics.__class__.__name__].mean()
# вычисляем стандартное отклонение метрики
metrics_std = df_metrics[metrics.__class__.__name__].std()
print(f"trans:\n{trans}")
print(f"{metrics.__class__.__name__}_mean: {metrics_mean}")
print(f"{metrics.__class__.__name__}_std: {metrics_std}\n")
# если получаем максимальное усредненное значение, сохраняем
# его и наилучший набор преобразований/признаков
if metrics_mean < best_score:
    best_score = metrics_mean
    best_parameters = {'trans': trans}

# печатаем наилучший набор преобразований/признаков
# и наилучшее значение метрики по итогам
# перекрестной проверки
print(f"Наилучший набор преобразований/признаков:\n{best_parameters}\n")
print(f"Лучшее значение {metrics.__class__.__name__} cv: {best_score:.4f}\n")

if refit:
    # создаем конвейер с наилучшим набором преобразований/признаков
    pipe = Pipeline(model=model,
                    transforms=best_parameters.get('trans'),
                    horizon=horizon)
    # обучаем конвейер на всей обучающей выборке
    pipe.fit(train_ts)
    # получаем прогнозы
    forecast_ts = pipe.forecast()
    # оцениваем качество прогнозов
    print(metrics(y_true=test_ts, y_pred=forecast_ts))
    # визуализируем прогнозы
    plot_forecast(forecast_ts, test_ts,
                  train_ts, n_train_samples=n_train_samples)

# выполняем подбор гиперпараметра - набора преобразований/
# признаков с помощью перекрестной проверки
etna_cv_optimize(ts=ts,
                 model=model,
                 horizon=HORIZON,
                 transfrms=[transforms, transforms2],
                 n_folds=5,
                 mode='expand',
                 metrics=SMAPE(),
                 refit=True)

```

trans:

```
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', )]
```

SMAPE\_mean: 8.746168790146212

SMAPE\_std: 2.8001672699916154

trans:

```
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', )]
```

SMAPE\_mean: 9.946587571595813

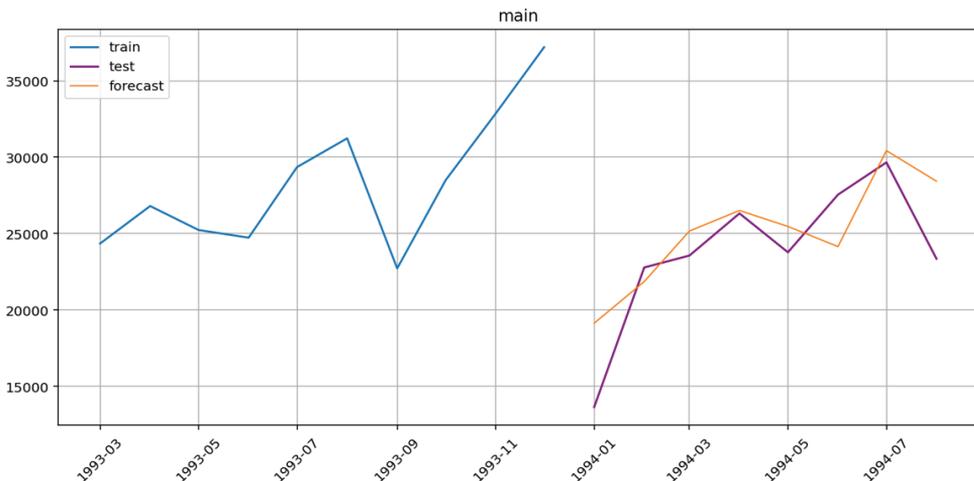
SMAPE\_std: 4.183457137940414

Наилучший набор преобразований/признаков:

```
{'trans': [LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', )]}
```

Лучшее значение SMAPE cv: 8.7462

```
{'main': 10.8808405270105}
```



Тестовая выборка используется один раз, но ради эксперимента сделаем исключение: посмотрим, какой результат на тестовой выборке дала бы модель с меньшим количеством лагов.

```
# разбиваем набор на обучающую и тестовую выборки
```

```
# с учетом временной структуры
```

```
train_ts, test_ts = ts.train_test_split(train_start='1980-01-01',
                                       train_end='1993-12-01',
                                       test_start='1994-01-01',
                                       test_end='1994-08-01')
```

```
# обучаем конвейер
```

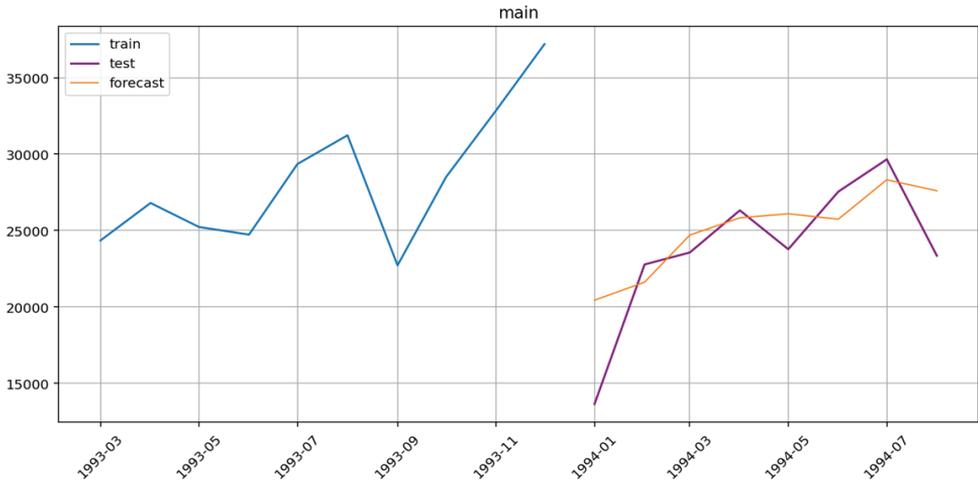
```
pipeline2.fit(train_ts)
```

```
# получаем прогнозы
```

```
forecast_ts2 = pipeline2.forecast()
```

```
# оцениваем качество прогнозов
print(smape(y_true=test_ts, y_pred=forecast_ts2))
# визуализируем прогнозы
plot_forecast(forecast_ts2, test_ts,
              train_ts, n_train_samples=10)
```

```
{'main': 11.123314225805467}
```



Модель с меньшим количеством лагов дает бóльшую ошибку на тестовой выборке, однако разница с первой моделью не является значительной.

Использование валидационной выборки при подборе гиперпараметров для нескольких временных рядов будет довольно громоздкой процедурой, поэтому здесь и далее будем применять перекрестную проверку расширяющимся окном.

Мы создадим лаги с порядком от 7 до 210 с шагом 7 и попробуем варьировать количество скользящих средних.

```
# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 7 до 210
# с шагом 7, порядок лагов не должен быть меньше длины горизонта
mult_lags = LagTransform(in_column='target',
                        lags=list(range(7, 211, 7)),
                        out_column='lag')
```

```
# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего с шириной окна 7
mult_mean7 = MeanTransform(in_column='target',
                          window=7,
                          out_column='mean7')
```

```
# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего с шириной окна 14
mult_mean14 = MeanTransform(in_column='target',
                           window=14,
                           out_column='mean14')
```

*# создаем экземпляр класса MeanTransform для вычисления*

*# скользящего среднего с шириной окна 30*

```
mult_mean30 = MeanTransform(in_column='target',
                           window=30,
                           out_column='mean30')
```

*# создаем списки процедур*

```
mult_transforms = [mult_lags, mult_mean7]
mult_transforms2 = [mult_lags, mult_mean7, mult_mean14]
mult_transforms3 = [mult_lags, mult_mean7, mult_mean14, mult_mean30]
```

*# создаем экземпляр класса CatBoostMultiSegmentModel*

```
model = CatBoostMultiSegmentModel()
```

*# выполняем подбор гиперпараметра - набора преобразований/*

*# признаков с помощью перекрестной проверки*

```
etna_cv_optimize(ts=mult_ts,
                model=model,
                horizon=mult_HORIZON,
                transforms=[mult_transforms, mult_transforms2, mult_transforms3],
                n_folds=12,
                mode='expand',
                metrics=SMAPE(),
                refit=True)
```

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84,
91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210],
out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha
= 1, min_periods = 1, fillna = 0, out_column = 'mean7', )]
SMAPE_mean: 8.803075891643422
SMAPE_std: 5.070471883638845
```

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84,
91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210],
out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha
= 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target',
window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', )]
SMAPE_mean: 8.091017304280527
SMAPE_std: 5.095530519939849
```

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84,
91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210],
out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha
= 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target',
window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14',
), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods
= 1, fillna = 0, out_column = 'mean30', )]
SMAPE_mean: 7.972182262782657
SMAPE_std: 5.0951576992863306
```

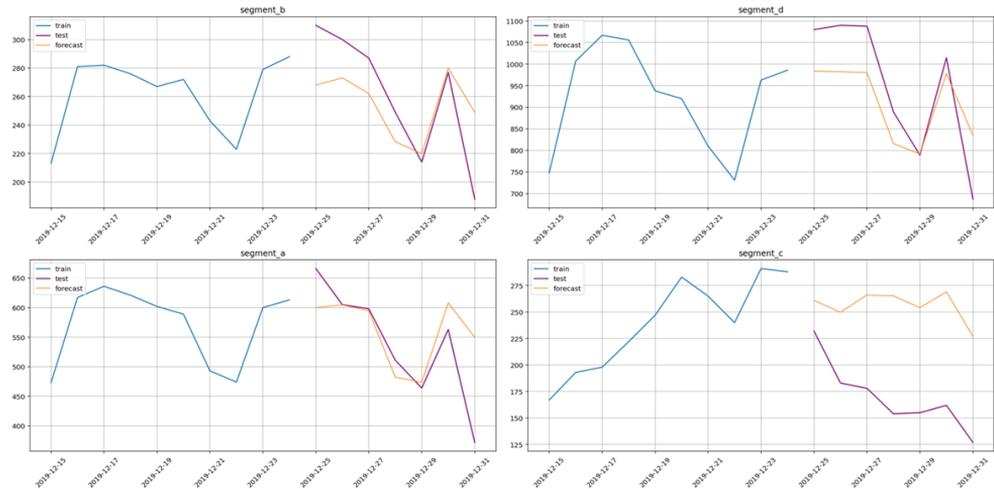
Наилучший набор преобразований/признаков:

```
{'trans': [LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70,
77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203,
210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality =
1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column
= 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column
```

```
= 'mean14', ), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1,
min_periods = 1, fillna = 0, out_column = 'mean30', )]]
```

Лучшее значение SMAPE cv: 7.9722

```
{'segment_b': 10.476681702546312, 'segment_d': 8.877571363079682, 'segment_a':
9.3133845436543, 'segment_c': 41.42742530510707}
```



## 2.12. Модель линейной регрессии с регуляризацией «ЭЛАСТИЧНАЯ СЕТЬ»

Теперь для прогнозирования продаж австралийского вина попробуем модель линейной регрессии с регуляризацией «эластичная сеть». Она реализована в классах `ElasticPerSegmentModel` и `ElasticMultiSegmentModel`, под капотом которых используется класс `ElasticNet` библиотеки `scikit-learn`. Класс `ElasticNet` универсален тем, что в зависимости от значения `l1_ratio` мы можем построить эластичную сеть ( $0 < l1\_ratio < 1$ ), лассо ( $l1\_ratio=1$ ) или гребневую регрессию ( $l1\_ratio=0$ ). Нам потребуются классы `StandardScalerTransform`, `MinMaxTransform`, `RobustScalerTransform` и `MaxAbsScalerTransform` для стандартизации признаков. Класс `StandardScalerTransform` стандартизирует признаки по формуле  $\frac{x_i - \text{mean}(x)}{\text{stdev}(x)}$ . Класс `MinMaxTransform` стандартизирует признаки по

формуле  $\frac{x_i - \min(x)}{\max(x) - \min(x)}$ . Класс `RobustScalerTransform` стандартизирует при-

знаки по формуле  $\frac{x_i - Q_1(x)}{Q_3(x) - Q_1(x)}$ . Класс `MaxAbsScalerTransform` стандартизирует

признаки по формуле  $\frac{x_i}{|\max(x)|}$ .

Создаем экземпляры классов `StandardScalerTransform`, `MinMaxScalerTransform`, `RobustScalerTransform` и `MaxAbsScalerTransform` для выполнения стандартизации всех переменных.

```
# создаем экземпляр класса StandardScalerTransform
# для выполнения стандартизации всех признаков
standardscaler = StandardScalerTransform(in_column=None)
```

```
# создаем экземпляр класса MaxAbsScalerTransform
# для выполнения стандартизации всех признаков
maxabsscaler = MaxAbsScalerTransform(in_column=None)
```

```
# создаем экземпляр класса MinMaxScalerTransform
# для выполнения стандартизации всех признаков
minmaxscaler = MinMaxScalerTransform(in_column=None)
```

```
# создаем экземпляр класса RobustScalerTransform
# для выполнения стандартизации всех признаков
robustscaler = RobustScalerTransform(in_column=None)
```

Создаем списки процедур. Попробуем лаги с порядком от 8 до 23, скользящее среднее с шириной окна 8 с разными вариантами стандартизации.

```
# создаем списки процедур
elast_transforms = [lags, mean8, standardscaler]
elast_transforms2 = [lags, mean8, maxabsscaler]
elast_transforms3 = [lags, mean8, minmaxscaler]
elast_transforms4 = [lags, mean8, robustscaler]
```

Конкретно в нашем случае мы построим модель гребневой регрессии.

```
# создаем экземпляр класса ElasticMultiSegmentModel
model = ElasticMultiSegmentModel(l1_ratio=0, alpha=0,
                                  random_state=42)
```

Давайте найдем наилучший набор преобразований и признаков с помощью перекрестной проверки расширяющимся окном.

```
# выполняем подбор гиперпараметра - набора преобразований/
# признаков с помощью перекрестной проверки
etna_cv_optimize(ts=ts,
                 model=model,
                 horizon=HORIZON,
                 transfrms=[elast_transforms,
                             elast_transforms2,
                             elast_transforms3,
                             elast_transforms4],
                 n_folds=5,
                 mode='expand',
                 metrics=SMAPE(),
                 refit=True)
```

```
trans:
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, season-
```

```
ality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', ), StandardScaler-
Transform(in_column = None, inplace = True, out_column = None, with_mean = True, with_std =
True, mode = 'per-segment', )]
SMAPE_mean: 8.88150866724813
SMAPE_std: 3.4845385529667037
```

trans:

```
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8,
seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', ), MaxAbsS-
calerTransform(in_column = None, inplace = True, out_column = None, mode = 'per-segment', )]
SMAPE_mean: 8.881508667248113
SMAPE_std: 3.484538552966679
```

trans:

```
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, sea-
sonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', ), MinMaxScaler-
Transform(in_column = None, inplace = True, out_column = None, feature_range = (0, 1), clip
= True, mode = 'per-segment', )]
SMAPE_mean: 8.918299760675247
SMAPE_std: 3.5643051281606786
```

trans:

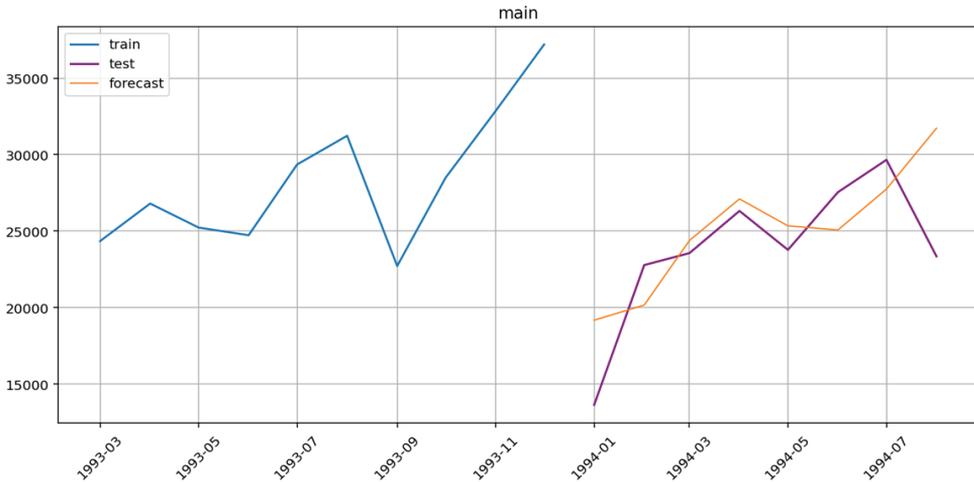
```
[LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 8, season-
ality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', ), RobustScalerTrans-
form(in_column = None, inplace = True, out_column = None, with_centering = True, with_scaling
= True, quantile_range = (25, 75), unit_variance = False, mode = 'per-segment', )]
SMAPE_mean: 8.88150866724813
SMAPE_std: 3.484538552966667
```

Наилучший набор преобразований/признаков:

```
{'trans': [LagTransform(in_column = 'target', lags = [8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, 20, 21, 22, 23], out_column = 'lag', ), MeanTransform(in_column = 'target', window =
8, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean8', ), MaxAbsS-
calerTransform(in_column = None, inplace = True, out_column = None, mode = 'per-segment', )]}
```

Лучшее значение SMAPE cv: 8.8815

```
{'main': 13.131923924791511}
```



Теперь применяем модель линейной регрессии с регуляризацией «эластичная сеть» для прогнозирования потребления электроэнергии по 4 сегментам.

Создаем списки процедур. Попробуем лаги с порядком от 7 до 210 с шагом 7, скользящее среднее с шириной окна 7 с разными вариантами стандартизации.

*# создаем списки процедур*

```
mult_elast_transforms = [mult_lags, mult_mean7, standardscaler]
mult_elast_transforms2 = [mult_lags, mult_mean7, maxabsscaler]
mult_elast_transforms3 = [mult_lags, mult_mean7, minmaxscaler]
mult_elast_transforms4 = [mult_lags, mult_mean7, robustscaler]
```

Опять находим наилучший набор преобразований и признаков с помощью перекрестной проверки расширяющимся окном.

*# выполняем подбор гиперпараметра - набора преобразований/  
# признаков с помощью перекрестной проверки*

```
etna_cv_optimize(ts=mult_ts,
                 model=model,
                 horizon=mult_HORIZON,
                 transfrms=[mult_elast_transforms,
                             mult_elast_transforms2,
                             mult_elast_transforms3,
                             mult_elast_transforms4],
                 n_folds=12,
                 mode='expand',
                 metrics=SMAPE(),
                 refit=True)
```

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), StandardScalerTransform(in_column = None, inplace = True, out_column = None, with_mean = True, with_std = True, mode = 'per-segment', )]
```

SMAPE\_mean: 8.063960799697611

SMAPE\_std: 4.532432762838726

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MaxAbsScalerTransform(in_column = ['lag_10', 'lag_11', 'lag_12', 'lag_13', 'lag_14', 'lag_15', 'lag_16', 'lag_17', 'lag_18', 'lag_19', 'lag_20', 'lag_21', 'lag_22', 'lag_23', 'lag_8', 'lag_9', 'mean8', 'target'], inplace = True, out_column = None, mode = 'per-segment', )]
```

SMAPE\_mean: 9.390634343540002

SMAPE\_std: 5.285904579070057

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MinMaxScalerTransform(in_column = None, inplace = True, out_column = None, feature_range = (0, 1), clip = True, mode = 'per-segment', )]
```

SMAPE\_mean: 9.973514207360305

SMAPE\_std: 7.2790997376453666

trans:

```
[LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), RobustScalerTransform(in_column = None, inplace = True, out_column = None, with_centering = True, with_scaling = True, quantile_range = (25, 75), unit_variance = False, mode = 'per-segment', )]
```

SMAPE\_mean: 7.4564054377020765

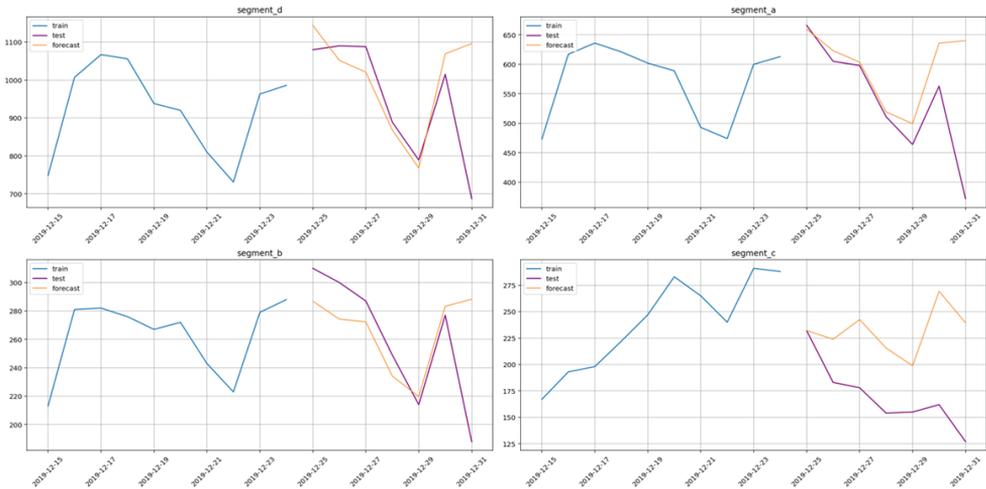
SMAPE\_std: 3.9494087381741894

Наилучший набор преобразований/признаков:

```
{'trans': [LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), RobustScalerTransform(in_column = None, inplace = True, out_column = None, with_centering = True, with_scaling = True, quantile_range = (25, 75), unit_variance = False, mode = 'per-segment', )]}
```

Лучшее значение SMAPE cv: 7.4564

```
{'segment_d': 10.268022701512683, 'segment_a': 11.267597403603467, 'segment_b': 10.729246765064282, 'segment_c': 31.426525384779108}
```



## 2.13. ОБЪЕДИНЕНИЕ ПРОЦЕДУРЫ ПОСТРОЕНИЯ МОДЕЛИ, ОЦЕНКИ КАЧЕСТВА И ВИЗУАЛИЗАЦИИ ПРОГНОЗОВ В ОДНОЙ ФУНКЦИИ

В этом разделе объединим процедуры разбиения набора на обучающую и тестовую выборки, построения модели, получения прогнозов, оценки качества и визуализации прогнозов в одной удобной функции `train_and_evaluate_model()`.

*# пишем функцию построения модели ETNA и оценки ее качества*

```
def train_and_evaluate_model(ts,
                             model,
                             transforms,
                             horizon,
                             metrics,
                             print_metrics=False,
                             print_plots=False,
                             n_train_samples=None):
```

"""

Обучает модель, вычисляет прогнозы для  
тестовой выборки и строит график прогнозов.

Параметры

-----

`ts`: `pandas.DataFrame`  
Временной ряд.  
`model`: instance of class `etna`  
Экземпляр класса библиотеки `etna`.  
`transforms`: list  
Список преобразований.  
`horizon`: int  
Горизонт прогнозирования.  
`metrics`: instance of class `etna.metrics.SMAPE/`  
`MAE/R2/MAPE/MedAE/MSLE`  
Метрика качества.

```

print_metrics: bool, по умолчанию False
    Печать метрик.
print_plots: bool, по умолчанию False
    Печать графиков прогнозов.
n_train_sample: int
    n последних наблюдений обучающей выборки
    на графике прогнозов.
"""

if not print_plots and n_train_samples is not None:
    raise ValueError(
        "Параметр n_train_samples задается при print_plots=True")

# разбиваем набор на обучающую и тестовую выборки
# с учетом временной структуры, размер тестовой
# выборки задаем равным горизонту
train_ts, test_ts = ts.train_test_split(test_size=horizon)
# создаем конвейер
pipe = Pipeline(model=model,
                transforms=transforms,
                horizon=horizon)
# обучаем конвейер
pipe.fit(train_ts)
# получаем прогнозы
forecast_ts = pipe.forecast()
# оцениваем качество прогнозов по сегментам
segment_metrics = metrics(test_ts, forecast_ts)
segment_metrics = pd.Series(segment_metrics)

if print_metrics:
    print(segment_metrics.to_string())
    print("")
    # оцениваем качество прогнозов в среднем
    print(f"Усредненная метрика:"
          f"{sum(segment_metrics) / len(segment_metrics)}")

if print_plots:
    # визуализируем прогнозы, здесь n_train_samples -
    # n последних наблюдений в обучающей выборке
    plot_forecast(forecast_ts, test_ts,
                  train_ts, n_train_samples=n_train_samples)

```

Давайте воспользуемся нашей функцией `train_and_evaluate_model()` для прогнозирования продаж австралийского вина. Мы создаем модель, список преобразований, передаем их в функцию `train_and_evaluate_model()`, также указываем горизонт, метрику, отрисовку графика прогнозов и количество последних наблюдений в обучающей выборке на графике прогнозов.

```

# создаем экземпляр класса CatBoostMultiSegmentModel
model = CatBoostMultiSegmentModel(loss_function='MAE',
                                   n_estimators=600,
                                   learning_rate=0.05,
                                   depth=9,
                                   random_seed=42)

# задаем список преобразований
preprocess = [lags, mean8]

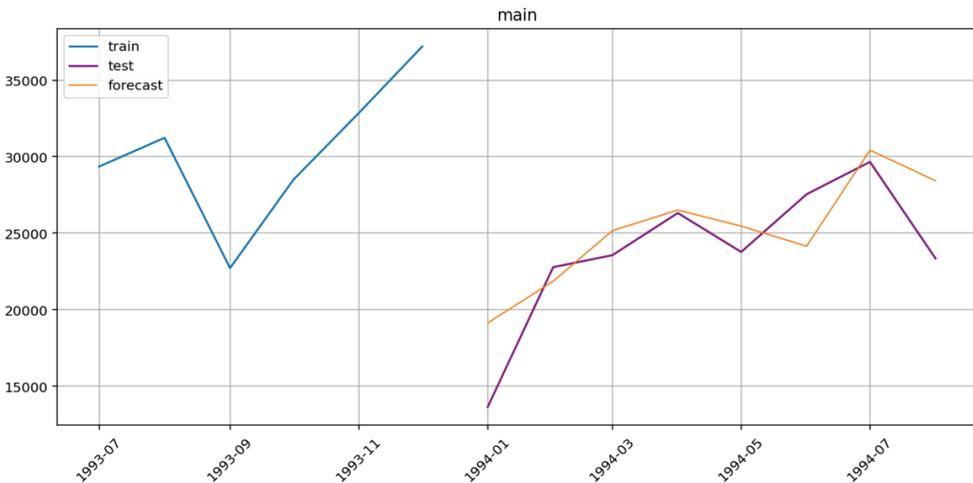
```

```
# строим и оцениваем модель CatBoostMultiSegmentModel
```

```
train_and_evaluate_model(
    ts=ts,
    model=model,
    transforms=preprocess,
    horizon=HORIZON,
    metrics=smape,
    print_metrics=True,
    print_plots=True,
    n_train_samples=6)
```

```
main    10.880841
```

```
Усредненная метрика:10.8808405270105
```



Теперь применим функцию `train_and_evaluate_model()` для прогнозирования потребления электроэнергии по 4 сегментам.

```
# создаем экземпляр класса CatBoostMultiSegmentModel
```

```
model = CatBoostMultiSegmentModel()
```

```
# задаем список преобразований
```

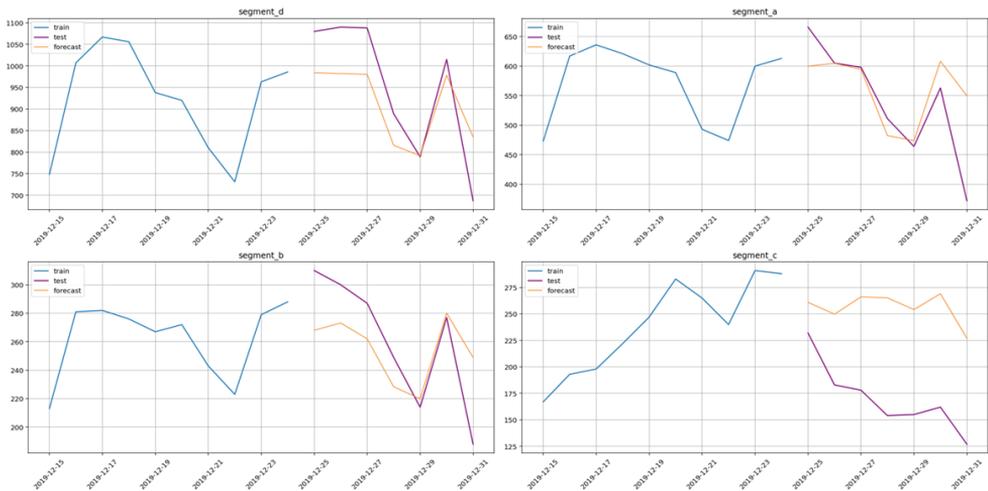
```
mult_preprocess = [mult_lags, mult_mean7,
                   mult_mean14, mult_mean30]
```

```
# строим и оцениваем модель CatBoostMultiSegmentModel
```

```
train_and_evaluate_model(
    ts=mult_ts,
    model=model,
    transforms=mult_preprocess,
    horizon=mult_HORIZON,
    metrics=smape,
    print_metrics=True,
    print_plots=True,
    n_train_samples=10)
```

```
segment_d    8.877571
segment_a    9.313385
segment_b    10.476682
segment_c    41.427425
```

Усредненная метрика: 17.523765728596842



## 2.14. ПЕРЕКРЕСТНАЯ ПРОВЕРКА НЕСКОЛЬКИХ МОДЕЛЕЙ

В ряде случаев бывает полезно сравнить несколько базовых моделей, не предполагая настройки гиперпараметров. Например, мы много сил вкладываем в модель CatBoost, но вполне возможно, что более простая модель даст лучшее качество прогнозов (ситуация, встречающаяся в прогнозировании рядов сплошь и рядом). Такую процедуру тоже можно выполнить с помощью класса Pipeline.

Сначала выполним перекрестную проверку нескольких моделей для продаж австалийского вина.

*# задаем конвейер преобразований для модели наивного прогноза*

```
naive_pipeline = Pipeline(
    model=NaiveModel(lag=12), transforms=[], horizon=HORIZON
)
```

*# задаем конвейер преобразований для Prophet*

```
prophet_pipeline = Pipeline(
    model=ProphetModel(), transforms=[], horizon=HORIZON
)
```

*# задаем конвейер преобразований для модели сезонного*

*# скользящего среднего с seasonality=12*

```
sma_seasonality_12_pipeline = Pipeline(
    model=SeasonalMovingAverageModel(window=5, seasonality=12),
    transforms=[], horizon=HORIZON
)
```

*# задаем конвейер преобразований для модели SARIMAX*

```
sarimax_pipeline = Pipeline(
    model=SARIMAXModel(order=(0, 1, 1),
                        seasonal_order=(0, 1, 1, 12)),
    transforms=[], horizon=HORIZON
)
```

```

# задаем конвейер преобразований для CatBoost
catboost_pipeline = Pipeline(
    model=CatBoostMultiSegmentModel(),
    transforms=preprocess,
    horizon=HORIZON
)

# задаем список имен конвейеров
pipeline_names = ['naive', 'prophet', 'sma_seasonality_12',
                  'sarimax', 'catboost']

# задаем список конвейеров
pipelines = [naive_pipeline, prophet_pipeline,
              sma_seasonality_12_pipeline,
              sarimax_pipeline, catboost_pipeline]

# задаем пустой список метрик
metrics = []

# записываем метрики в список
for pipeline in pipelines:
    metrics.append(
        pipeline.backtest(
            ts=ts, metrics=[MAE(), MSE(), SMAPE(), MAPE()],
            n_folds=3, aggregate_metrics=True,
            joblib_params=dict(verbose=0))[0].iloc[:, 1:])

# конкатенируем метрики
metrics = pd.concat(metrics)
# в качестве индекса используем список имен конвейеров
metrics.index = pipeline_names
metrics

```

	MAE	MSE	SMAPE	MAPE
<b>naive</b>	1987.333333	7.688508e+06	8.062820	8.242755
<b>prophet</b>	1899.264320	5.486450e+06	7.588063	7.599401
<b>sma_seasonality_12</b>	1580.191667	4.394046e+06	6.512327	6.471100
<b>sarimax</b>	1796.780630	5.979354e+06	7.466573	7.668463
<b>catboost</b>	2116.123614	8.040332e+06	8.368528	8.774443

Перекрестная проверка дает нам возможность убедиться в силе простых моделей: модель сезонного скользящего среднего работает лучше, чем CatBoost.

Теперь выполним перекрестную проверку нескольких моделей для прогнозирования потребления электроэнергии по 4 сегментам.

```

# задаем конвейер преобразований для модели наивного прогноза
m_naive_pipeline = Pipeline(
    model=NaiveModel(lag=12), transforms=[], horizon=mult_HORIZON
)

```

```

# задаем конвейер преобразований для Prophet
m_prophet_pipeline = Pipeline(
    model=ProphetModel(), transforms=[], horizon=mult_HORIZON
)

# задаем конвейер преобразований для модели сезонного
# скользящего среднего с seasonality=7
m_sma_seasonality_7_pipeline = Pipeline(
    model=SeasonalMovingAverageModel(window=5, seasonality=7),
    transforms=[], horizon=mult_HORIZON
)

# задаем конвейер преобразований для модели SARIMAX
m_sarimax_pipeline = Pipeline(
    model=SARIMAXModel(seasonal_order=(1, 1, 0, 7)),
    transforms=[], horizon=mult_HORIZON
)

# задаем конвейер преобразований для CatBoost
m_catboost_pipeline = Pipeline(
    model=CatBoostMultiSegmentModel(),
    transforms=mult_preprocess,
    horizon=mult_HORIZON
)

# задаем список имен конвейеров
m_pipeline_names = ['naive', 'prophet', 'sma_seasonality_7',
                    'sarimax', 'catboost']

# задаем список конвейеров
m_pipelines = [m_naive_pipeline, m_prophet_pipeline,
               m_sma_seasonality_7_pipeline,
               m_sarimax_pipeline, m_catboost_pipeline]

# задаем пустой список метрик
metrics = []

# записываем метрики в список
for pipeline in m_pipelines:
    metrics.append(
        pipeline.backtest(
            ts=mult_ts, metrics=[MAE(), MSE(), SMAPE(), MAPE()],
            n_folds=12, aggregate_metrics=True,
            joblib_params=dict(verbose=0))[0].iloc[:, 1:]
    )

```

Давайте сконкатенируем полученные метрики и получим итоговый да-тафрейм с метриками для каждого сегмента (ряда) по каждой модели.

```

# конкатенируем метрики
metrics = pd.concat(metrics)
metrics

```

	MAE	MSE	SMAPE	MAPE
0	90.892857	11046.702381	17.852963	18.035695
1	39.119048	2016.261905	16.626977	16.647956
2	27.178571	1519.440476	14.575549	14.260105
3	164.345238	37300.583333	19.803840	20.208163
0	38.791544	2927.305380	7.502665	7.556337
1	14.655235	379.815245	6.025290	6.014973
2	25.814599	1245.350978	14.068599	13.624464
3	56.135618	5971.894435	6.652540	6.927647
0	41.397619	3272.597619	7.991923	7.987848
1	15.342857	404.531429	6.378363	6.336805
2	24.226190	1228.716667	12.862943	12.294276
3	61.350000	7357.222381	7.170621	7.079425
0	45.313331	3510.820844	8.879059	8.977679
1	14.111440	391.952536	5.819930	5.926351
2	32.644883	2813.400721	15.540450	17.361461
3	63.857407	6563.728032	7.555191	7.521144
0	39.139109	2900.886853	7.683514	7.641113
1	14.849488	375.661535	6.153316	6.138244
2	28.373851	1769.186021	14.472334	14.699103
3	62.121861	7024.016952	7.247378	7.087250

Теперь создаем список имен конвейеров и список сегментов.

```
# задаем количество конвейеров
n_pipeline_names = len(m_pipeline_names)
# задаем количество сегментов (рядов)
n_segments = len(mult_ts.columns.get_level_values('segment'))

# создаем список имен конвейеров
pipeline_list = []
for i in range(n_pipeline_names):
    pipeline_list += [m_pipeline_names[i]] * n_segments
pipeline_list

['naive',
 'naive',
 'naive',
 'naive',
 'prophet',
 'prophet',
 'prophet',
 'prophet',
 'sma_seasonality_7',
 'sma_seasonality_7',
 'sma_seasonality_7',
 'sma_seasonality_7',
```

```
'sarimax',
'sarimax',
'sarimax',
'sarimax',
'catboost',
'catboost',
'catboost',
'catboost']
```

*# создаем список сегментов*

```
ts_list = sorted(set(
    mult_ts.columns.get_level_values('segment'))) * n_pipelines
ts_list
```

```
['segment_a',
'segment_b',
'segment_c',
'segment_d',
'segment_a',
'segment_b',
'segment_c',
'segment_d']
```

Теперь на основе списка имен конвейеров и списка сегментов с помощью функции `zip()` создаем список кортежей – комбинаций имен конвейеров и сегментов.

*# получаем список кортежей – комбинаций имен конвейеров и сегментов*

```
names = list(zip(pipeline_list, ts_list))
names
```

```
(('naive', 'segment_a'),
('naive', 'segment_b'),
('naive', 'segment_c'),
('naive', 'segment_d'),
('prophet', 'segment_a'),
('prophet', 'segment_b'),
('prophet', 'segment_c'),
('prophet', 'segment_d'),
('sma_seasonality_7', 'segment_a'),
('sma_seasonality_7', 'segment_b'),
('sma_seasonality_7', 'segment_c'),
('sma_seasonality_7', 'segment_d'),
('sarimax', 'segment_a'),
```

```
( 'sarimax', 'segment_b'),
( 'sarimax', 'segment_c'),
( 'sarimax', 'segment_d'),
( 'catboost', 'segment_a'),
( 'catboost', 'segment_b'),
( 'catboost', 'segment_c'),
( 'catboost', 'segment_d']
```

В качестве индекса датафрейма с метриками для каждого сегмента (ряда) по каждой модели используем комбинации имен конвейеров и сегментов.

*# в качестве индекса используем комбинации*

*# имен конвейеров и сегментов*

```
metrics.index = names
```

```
metrics
```

	MAE	MSE	SMAPE	MAPE
<b>(naive, segment_a)</b>	90.892857	11046.702381	17.852963	18.035695
<b>(naive, segment_b)</b>	39.119048	2016.261905	16.626977	16.647956
<b>(naive, segment_c)</b>	27.178571	1519.440476	14.575549	14.260105
<b>(naive, segment_d)</b>	164.345238	37300.583333	19.803840	20.208163
<b>(prophet, segment_a)</b>	38.791544	2927.305380	7.502665	7.556337
<b>(prophet, segment_b)</b>	14.655235	379.815245	6.025290	6.014973
<b>(prophet, segment_c)</b>	25.814599	1245.350978	14.068599	13.624464
<b>(prophet, segment_d)</b>	56.135618	5971.894435	6.652540	6.927647
<b>(sma_seasonality_7, segment_a)</b>	41.397619	3272.597619	7.991923	7.987848
<b>(sma_seasonality_7, segment_b)</b>	15.342857	404.531429	6.378363	6.336805
<b>(sma_seasonality_7, segment_c)</b>	24.226190	1228.716667	12.862943	12.294276
<b>(sma_seasonality_7, segment_d)</b>	61.350000	7357.222381	7.170621	7.079425
<b>(sarimax, segment_a)</b>	45.313331	3510.820844	8.879059	8.977679
<b>(sarimax, segment_b)</b>	14.111440	391.952536	5.819930	5.926351
<b>(sarimax, segment_c)</b>	32.644883	2813.400721	15.540450	17.361461
<b>(sarimax, segment_d)</b>	63.857407	6563.728032	7.555191	7.521144
<b>(catboost, segment_a)</b>	39.139109	2900.886853	7.683514	7.641113
<b>(catboost, segment_b)</b>	14.849488	375.661535	6.153316	6.138244
<b>(catboost, segment_c)</b>	28.373851	1769.186021	14.472334	14.699103
<b>(catboost, segment_d)</b>	62.121861	7024.016952	7.247378	7.087250

## 2.15. Ансамбли

С помощью класса `VotingEnsemble` можно выполнить обучение и перекрестную проверку ансамбля моделей. Веса моделей можно задавать с помощью параметра `weights`.

Сначала мы попробуем ансамбль моделей для продаж австралийского вина.

```
# выбираем наиболее сильные модели
pipes = [prophet_pipeline, sma_seasonality_12_pipeline]
# создаем экземпляр класса VotingEnsemble
vot_ens = VotingEnsemble(pipelines=pipes,
                          weights=[0.5, 9], n_jobs=2)
# получаем метрики
vot_ens_metrics, vot_ens_forecasts, _ = vot_ens.backtest(
    ts=ts, metrics=[MAE(), MSE(), SMAPE(), MAPE()], n_folds=3,
    aggregate_metrics=True, n_jobs=2)
vot_ens_metrics.index = ['voting ensemble']

# смотрим датафрейм с метриками
vot_ens_metrics
```

	MAE	MSE	SMAPE	MAPE
<b>voting ensemble</b>	1568.642377	4.345794e+06	6.474433	6.437749

```
# визуализируем результаты
plot_backtest(vot_ens_forecasts, ts)
```



Теперь мы попробуем ансамбль моделей для прогнозирования потребления электроэнергии по 4 сегментам.

```
# выбираем наиболее сильные модели
m_pipelines = [m_prophet_pipeline,
               m_sma_seasonality_7_pipeline,
               m_sarimax_pipeline,
               m_catboost_pipeline]
```

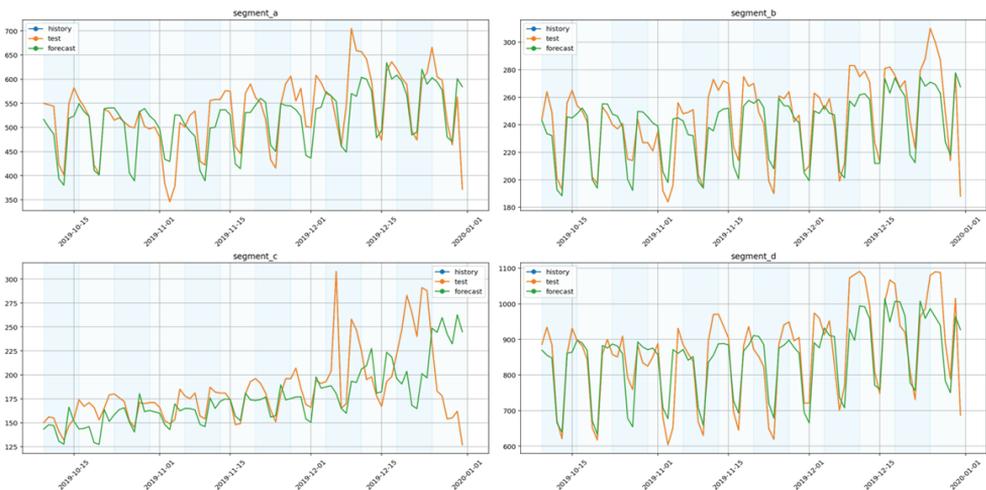
```
# создаем экземпляр класса VotingEnsemble
m_vot_ens = VotingEnsemble(pipelines=m_pipelines,
                           weights=[1, 1, 1, 2], n_jobs=1)

# получаем метрики
m_vot_ens_metrics, m_vot_ens_forecasts, _ = m_vot_ens.backtest(
    ts=mult_ts, metrics=[SMAPE()], n_folds=12,
    aggregate_metrics=True, n_jobs=1)

# смотрим датафрейм с метриками
m_vot_ens_metrics
```

	segment	SMAPE
0	segment_a	7.254052
1	segment_b	5.591747
2	segment_c	13.104487
3	segment_d	6.355952

```
# визуализируем результаты
plot_backtest(m_vot_ens_forecasts, mult_ts)
```



## 2.16. СТЕКИНГ

С помощью класса `StackingEnsemble` можно выполнить стекинг. Мы прогнозируем будущее, используя метамодель (линейную регрессию по умолчанию) для объединения прогнозов моделей в списке конвейеров. С помощью параметра `final_model` можно задать метамодель. С помощью `features_to_use` можно задавать признаки для метамодели:

- `None` (по умолчанию) – метамодель в качестве признаков может использовать прогнозы моделей конвейеров;
- `List` – прогнозы моделей конвейеров плюс признаки из списка (в виде строковых значений);
- `'all'` – все доступные признаки.

С помощью параметра `n_folds` задаем количество тестовых выборок перекрестной выборки (используем не для оценки моделей, а для получения прогнозов, которые станут у нас потом признаками).

Давайте выполним стекинг, используя в качестве признаков прогнозы моделей конвейеров.

```
# создаем экземпляр класса StackingEnsemble,
# признаки - прогнозы конвейеров
stacking_ensemble_unfeatured = StackingEnsemble(
    features_to_use='None', pipelines=pipelines, n_folds=10, n_jobs=4)
# выполняем стекинг
stacking_ensemble_metrics = stacking_ensemble_unfeatured.backtest(
    ts=ts, metrics=[MAE(), MSE(), SMAPE(), MAPE()], n_folds=3,
    aggregate_metrics=True, n_jobs=2)[0].iloc[:, 1:]
stacking_ensemble_metrics.index = ['stacking ensemble']
stacking_ensemble_metrics
```

	MAE	MSE	SMAPE	MAPE
<b>stacking ensemble</b>	1797.744433	5.600547e+06	7.376548	7.311183

## 2.17. СОЗДАНИЕ СОБСТВЕННЫХ КЛАССОВ ДЛЯ ОБУЧЕНИЯ МОДЕЛЕЙ

Предположим, мы хотим использовать LightGBM вместо CatBoost. Нам понадобятся класс `LGBMRegressor` и базовые классы библиотеки ETNA `BaseAdapter`, `PerSegmentModel` и `MultiSegmentModel`. Сначала надо написать ядро – внутренний класс `_LGBMAdapter`, в котором используется `LGBMRegressor`. Символ нижнего подчеркивания указывает на то, что данный класс будет использоваться внутри других классов. У класса `_LGBMModel` будут три метода `.fit()`, `.predict()` и `.get_model()`.

```
# пишем ядро - внутренний класс _LGBMAdapter,
# внутри - класс LGBMRegressor
class _LGBMAdapter(BaseAdapter):
    def __init__(
        self,
        boosting_type='gbdt',
        num_leaves=31,
        max_depth=-1,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs,
    ):
        self.model = LGBMRegressor(
            boosting_type=boosting_type,
            num_leaves=num_leaves,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
        self._categorical = None
```

```

def fit(self, df: pd.DataFrame, regressors: List[str]):
    features = df.drop(columns=['timestamp', 'target'])
    self._categorical = features.select_dtypes(
        include=['category']).columns.to_list()
    target = df['target']
    self.model.fit(X=features, y=target,
                  categorical_feature=self._categorical)

    return self

def predict(self, df: pd.DataFrame):
    features = df.drop(columns=['timestamp', 'target'])
    pred = self.model.predict(features)
    return pred

def get_model(self) -> LGBMRegressor:
    return self.model

```

Вспомним, что мы можем строить отдельную модель для каждого сегмента и одну модель для всего набора (т. е. для всех сегментов). Значит, мы можем написать два класса. Начнем с класса, который будет строить отдельную модель LightGBM для каждого сегмента. Назовем его `LGBMPerSegmentModel`. Для этого воспользуемся наследованием, нам понадобится базовый класс `PerSegmentModel`.

*# пишем класс LGBMPerSegmentModel, который строит  
# отдельную модель для каждого сегмента*

```

class LGBMPerSegmentModel(PerSegmentModel):
    def __init__(
        self,
        boosting_type='gbdt',
        num_leaves=31,
        max_depth=-1,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.boosting_type = boosting_type
        self.num_leaves = num_leaves
        self.max_depth = max_depth
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.kwargs = kwargs
        model = _LGBMAdapter(
            boosting_type=boosting_type,
            num_leaves=num_leaves,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
        super().__init__(base_model=model)

```

Теперь напишем класс, который будет строить одну модель LightGBM для всех сегментов. Назовем его `LGBMMultiSegmentModel`. Для этого вновь воспользуемся наследованием, нам понадобится базовый класс `MultiSegmentModel`.

*# пишем класс LGBMMultiSegmentModel, который строит*

*# одну модель LightGBM для всех сегментов*

```
class LGBMMultiSegmentModel(MultiSegmentModel):
```

```
    def __init__(
        self,
        boosting_type='gbdt',
        num_leaves=31,
        max_depth=-1,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.boosting_type = boosting_type
        self.num_leaves = num_leaves
        self.max_depth = max_depth
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.kwargs = kwargs
        model = _LGBMAdapter(
            boosting_type=boosting_type,
            num_leaves=num_leaves,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
        super().__init__(base_model=model)
```

Аналогично можно реализовать XGBoost.

*# пишем ядро - внутренний класс \_XGBAdapter,*

*# внутри - класс XGBRegressor*

```
class _XGBAdapter(BaseAdapter):
```

```
    def __init__(
        self,
        booster='gbtree',
        max_depth=3,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.model = XGBRegressor(
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
    def fit(self, df: pd.DataFrame, regressors: List[str]):
        features = df.drop(columns=['timestamp', 'target'])
        for col in features.columns.tolist():
            features[col] = features[col].astype('category').cat.codes
        target = df['target']
        self.model.fit(X=features, y=target)
        return self
```

```

def predict(self, df: pd.DataFrame):
    features = df.drop(columns=['timestamp', 'target'])
    for col in features.columns.tolist():
        features[col] = features[col].astype('category').cat.codes
    pred = self.model.predict(features)
    return pred
def get_model(self) -> XGBRegressor:
    return self.model

```

*# пишем класс XGBPerSegmentModel, который строит  
# отдельную модель для каждого сегмента*

```

class XGBPerSegmentModel(PerSegmentModel):
    def __init__(
        self,
        booster='gbtree',
        max_depth=3,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs
    ):
        self.booster = booster
        self.max_depth = max_depth
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.kwargs = kwargs
        model = _XGBAdapter(
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
        super().__init__(base_model=model)

```

*# пишем класс XGBMultiSegmentModel, который строит  
# одну модель XGBoost для всех сегментов*

```

class XGBMultiSegmentModel(MultiSegmentModel):
    def __init__(
        self,
        booster='gbtree',
        max_depth=3,
        learning_rate=0.1,
        n_estimators=100,
        **kwargs,
    ):
        self.booster = booster
        self.max_depth = max_depth
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.kwargs = kwargs
        model = _XGBAdapter(
            booster=booster,
            max_depth=max_depth,
            learning_rate=learning_rate,
            n_estimators=n_estimators,
            **kwargs
        )
        super().__init__(base_model=model)

```

С помощью базовых классов библиотеки ETNA `SklearnPerSegmentModel` и `SklearnMultiSegmentModel` можно обучать модели-регрессоры библиотеки `scikit-learn` в ETNA. Давайте реализуем класс `GradientBoostingRegressor` в ETNA, у нас будет класс `GPerSegmentModel`, который строит отдельную модель градиентного бустинга для каждого сегмента, и класс `GBMultiSegmentModel`, который строит одну модель градиентного бустинга для всех сегментов.

*# пишем классы `GPerSegmentModel` и `GBMultiSegmentModel`  
# для применения класса `GradientBoostingRegressor` в ETNA*

```
class GPerSegmentModel(SklearnPerSegmentModel):
    """
    Класс :py:class:`sklearn.ensemble.GradientBoostingRegressor`
    для сегмента.
    """
    def __init__(self,
                 learning_rate: float = 0.01,
                 n_estimators: int = 100,
                 max_depth: int = 20,
                 **kwargs):
        """
        Создает экземпляр класса GradientBoostingRegressor
        с заданными параметрами.

        Параметры
        -----
        learning_rate:
            Темп обучения.
        n_estimators:
            Количество деревьев.
        max_depth:
            Максимальная глубина деревьев.
        """
        self.learning_rate = learning_rate
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.kwargs = kwargs
        super().__init__(regressor=GradientBoostingRegressor(
            learning_rate=self.learning_rate,
            n_estimators=self.n_estimators,
            max_depth=self.max_depth,
            **self.kwargs))

class GBMultiSegmentModel(SklearnMultiSegmentModel):
    """
    Класс :py:class:`sklearn.ensemble.GradientBoostingRegressor`
    для всех сегментов.
    """
    def __init__(self,
                 learning_rate: float = 0.01,
                 n_estimators: int = 100,
                 max_depth: int = 20,
                 **kwargs):
        """
        Создает экземпляр класса GradientBoostingRegressor
        с заданными параметрами.
        """
```

```

    Параметры
    -----
    learning_rate:
        Темп обучения.
    n_estimators:
        Количество деревьев.
    max_depth:
        Максимальная глубина деревьев.
    """
    self.learning_rate = learning_rate
    self.n_estimators = n_estimators
    self.max_depth = max_depth
    self.kwargs = kwargs
    super().__init__(regressor=GradientBoostingRegressor(
        learning_rate=self.learning_rate,
        n_estimators=self.n_estimators,
        max_depth=self.max_depth,
        **self.kwargs))

```

Теперь реализуем класс `HistGradientBoostingRegressor` в ETNA, у нас будет класс `HistGBPerSegmentModel`, который строит отдельную модель градиентного бустинга с гистограммированием для каждого сегмента, и класс `HistGBMultiSegmentModel`, который строит одну модель градиентного бустинга с гистограммированием для всех сегментов.

*# пишем классы HistGBPerSegmentModel и HistGBMultiSegmentModel  
# для применения класса HistGradientBoostingRegressor в ETNA*

```

class HistGBPerSegmentModel(SklearnPerSegmentModel):
    """
    Класс :py:class:`sklearn.ensemble.HistGradientBoostingRegressor`
    для сегмента.
    """

    def __init__(self,
                 learning_rate: float = 0.05,
                 max_iter: int = 100,
                 max_depth: int = 20,
                 **kwargs):
        """
        Создает экземпляр класса HistGradientBoostingRegressor
        с заданными параметрами.

        Параметры
        -----
        learning_rate:
            Темп обучения.
        max_iter:
            Максимальное количество деревьев.
        max_depth:
            Максимальная глубина деревьев.
        """
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.max_depth = max_depth
        self.kwargs = kwargs

```

```

super().__init__(regressor=HistGradientBoostingRegressor(
    learning_rate=self.learning_rate,
    max_iter=self.max_iter,
    max_depth=self.max_depth,
    **self.kwargs))

```

```

class HistGBMultiSegmentModel(SklearnMultiSegmentModel):

```

```

    """

```

```

    Класс :py:class:`sklearn.ensemble.HistGradientBoostingRegressor`
    для всех сегментов.
    """

```

```

def __init__(self,
              learning_rate: float = 0.05,
              max_iter: int = 100,
              max_depth: int = 20,
              **kwargs):

```

```

    """

```

```

    Создает экземпляр класса HistGradientBoostingRegressor
    с заданными параметрами.

```

```

    Параметры
    -----

```

```

    learning_rate:

```

```

        Темп обучения.

```

```

    max_iter:

```

```

        Максимальное количество деревьев.

```

```

    max_depth:

```

```

        Максимальная глубина деревьев.
    """

```

```

self.learning_rate = learning_rate
self.max_iter = max_iter
self.max_depth = max_depth
self.kwargs = kwargs
super().__init__(regressor=HistGradientBoostingRegressor(
    learning_rate=self.learning_rate,
    max_iter=self.max_iter,
    max_depth=self.max_depth,
    **self.kwargs))

```

Реализуем класс `RandomForestRegressor` в ETNA, у нас будет класс `RFPPerSegmentModel`, который строит отдельную модель случайного леса для каждого сегмента, и класс `RFMultiSegmentModel`, который строит одну модель случайного леса для всех сегментов.

```

# пишем классы RFPPerSegmentModel и RFMultiSegmentModel для
# применения класса RandomForestRegressor в ETNA

```

```

class RFPPerSegmentModel(SklearnPerSegmentModel):

```

```

    """

```

```

    Класс :py:class:`sklearn.ensemble.RandomForestRegressor`
    для сегмента.
    """

```

```

def __init__(self,
              n_estimators: int = 100,
              max_depth: int = 20,
              **kwargs):

```

```

"""
Создает экземпляр класса RandomForestRegressor
с заданными параметрами.

Параметры
-----
n_estimators:
    Количество деревьев случайного леса.
max_depth:
    Максимальная глубина деревьев
    случайного леса.
"""
self.n_estimators = n_estimators
self.max_depth = max_depth
self.kwargs = kwargs
super().__init__(regressor=RandomForestRegressor(
    n_estimators=self.n_estimators,
    max_depth=self.max_depth,
    **self.kwargs))

```

### Реализуем класс BaggingRegressor в ETNA

*# пишем классы BRPerSegmentModel и BRMultiSegmentModel  
# для применения класса BaggingRegressor в ETNA*

```

class RFMultiSegmentModel(SklearnMultiSegmentModel):
    """
    Класс :py:class:`sklearn.ensemble.BaggingRegressor`
    для сегмента.
    """

    def __init__(self,
                 n_estimators: int = 100,
                 max_samples: float = 1.0,
                 max_features: float = 1.0,
                 bootstrap: bool = True,
                 bootstrap_features: bool = False,
                 **kwargs):
        """
        Создает экземпляр класса BaggingRegressor
        с заданными параметрами.

        Параметры
        -----
        n_estimators:
            Количество базовых моделей.
        max_samples:
            Количество наблюдений, используемых
            для обучения каждой базовой модели.
        max_features:
            Количество признаков, используемых
            для обучения каждой базовой модели.
        bootstrap:
            Нужен ли отбор наблюдений с возвращением.
        bootstrap_features:
            Нужен ли отбор признаков с возвращением.
        """

```

```

self.n_estimators = n_estimators
self.max_samples = max_samples
self.max_features = max_features
self.bootstrap = bootstrap
self.bootstrap_features = bootstrap_features
self.kwags = kwags
super().__init__(regressor=BaggingRegressor(
    n_estimators=self.n_estimators,
    max_samples=self.max_samples,
    max_features=max_features,
    bootstrap=bootstrap,
    bootstrap_features=bootstrap_features,
    **self.kwags))

```

```
class BRMultiSegmentModel(SklearnMultiSegmentModel):
```

```
    """
```

```
    Класс :py:class:`sklearn.ensemble.BaggingRegressor`
    для всех сегментов.
```

```
    """
```

```

def __init__(self,
              n_estimators: int = 100,
              max_samples: float = 1.0,
              max_features: float = 1.0,
              bootstrap: bool = True,
              bootstrap_features: bool = False,
              **kwargs):

```

```
    """
```

```
    Создает экземпляр класса BaggingRegressor
    с заданными параметрами.
```

```
    Параметры
```

```
    -----
```

```
    n_estimators:
```

```
        Количество базовых моделей.
```

```
    max_samples:
```

```
        Количество наблюдений, используемых для обучения каждой базовой модели.
```

```
    max_features:
```

```
        Количество признаков, используемых
        для обучения каждой базовой модели.
```

```
    bootstrap:
```

```
        Нужен ли отбор наблюдений с возвращением.
```

```
    bootstrap_features:
```

```
        Нужен ли отбор признаков с возвращением.
```

```
    """
```

```

self.n_estimators = n_estimators
self.max_samples = max_samples
self.max_features = max_features
self.bootstrap = bootstrap
self.bootstrap_features = bootstrap_features
self.kwags = kwargs
super().__init__(regressor=BaggingRegressor(
    n_estimators=self.n_estimators,
    max_samples=self.max_samples,
    max_features=max_features,
    bootstrap=bootstrap,
    bootstrap_features=bootstrap_features,
    **self.kwags))

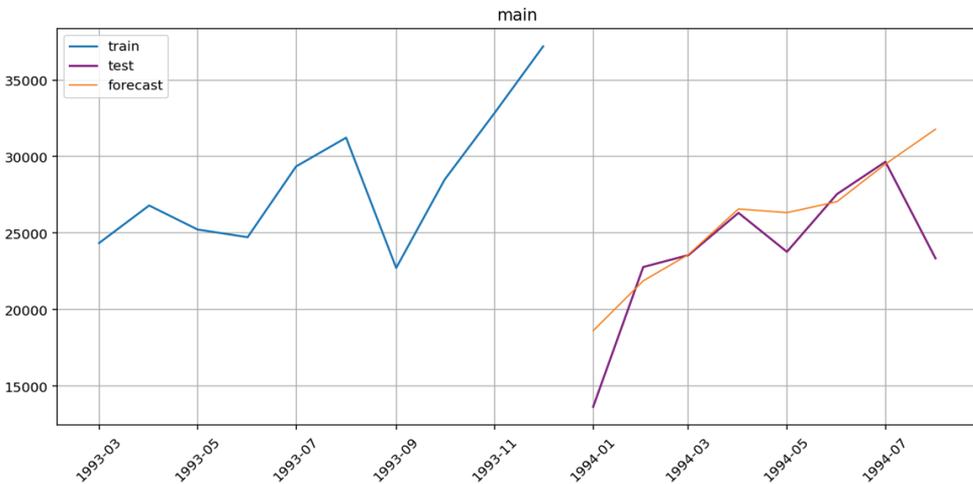
```

Пришло время попробовать наши классы в действии!

Для этого разбиваем набор с продажами австралийского вина на обучающую и тестовую выборки с учетом временной структуры и создаем экземпляр класса `Pipeline`, передав ему модель `LGBMMultiSegmentModel`, ранее созданный список преобразований `preprocess` и горизонт `HORIZON`. Обучаем модель, получаем прогнозы и визуализируем их. Мы можем воспользоваться как классом `LGBMMultiSegmentModel`, так и классом `LGBMPerSegmentModel`.

```
# разбиваем набор на обучающую и тестовую выборки
# с учетом временной структуры
train_ts, test_ts = ts.train_test_split(test_size=HORIZON)
# создаем экземпляр класса LGBMMultiSegmentModel
model = LGBMMultiSegmentModel()
# создаем конвейер с моделью и списком преобразований
pipeline = Pipeline(model=model,
                    transforms=preprocess,
                    horizon=HORIZON)
# обучаем конвейер
pipeline.fit(train_ts)
# получаем прогнозы
forecast_ts = pipeline.forecast()
# оцениваем качество прогнозов
print(smape(y_true=test_ts, y_pred=forecast_ts))
# визуализируем прогнозы
plot_forecast(forecast_ts, test_ts,
              train_ts, n_train_samples=10)

{'main': 9.885424521781644}
```



Для сокращения программного кода можно воспользоваться функцией `train_and_evaluate_model()`.

```
# строим и оцениваем модель LGBMMultiSegmentModel
# с помощью функции train_and_evaluate_model()
train_and_evaluate_model(
    ts=ts,
```

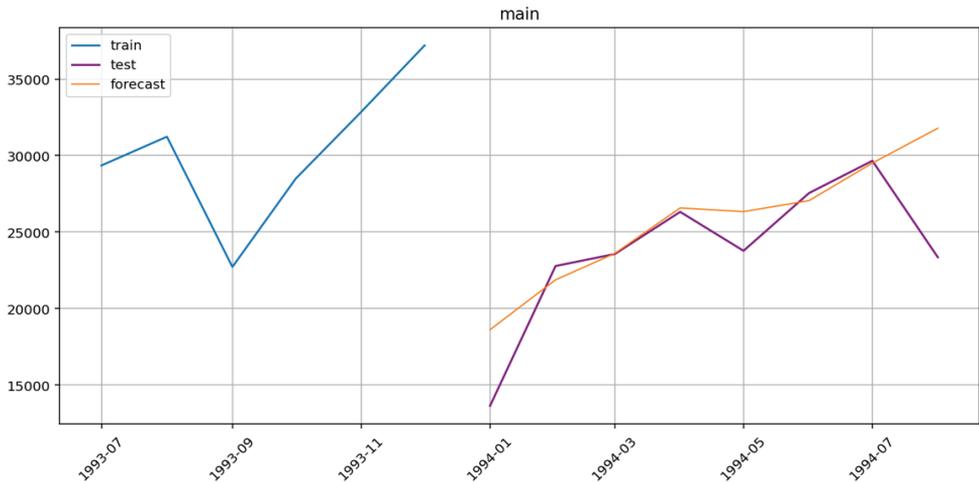
```

model=model,
transforms=preprocess,
horizon=HORIZON,
metrics=smape,
print_metrics=True,
print_plots=True,
n_train_samples=6)

```

```
main 9.885425
```

Усредненная метрика:9.885424521781644



Теперь попробуем XGBoost, реализации градиентного бустинга и случайного леса из библиотеки scikit-learn.

*# создаем список моделей*

```

model_lst = [XGBMultiSegmentModel(),
             GBMultiSegmentModel(random_state=42),
             HistGBMultiSegmentModel(random_state=42),
             RFMultiSegmentModel(random_state=42),
             BRMultiSegmentModel(random_state=42)]

```

*# строим модели для одного ряда*

```

for model in model_lst:
    print('')
    print(model)
    # строим и оцениваем модель из списка с помощью
    # функции train_and_evaluate_model()
    train_and_evaluate_model(
        ts=ts,
        model=model,
        transforms=preprocess,
        horizon=HORIZON,
        metrics=smape,
        print_metrics=False,
        print_plots=False,
        n_train_samples=None)

```

```

XGBMultiSegmentModel(booster = 'gbtree', max_depth = 3, learning_rate = 0.1, n_estimators = 100, )
main 24.088075

```

Усредненная метрика:24.08807498819172

```
GBMultiSegmentModel(learning_rate = 0.01, n_estimators = 100, max_depth = 20, random_state = 42, )
main 11.266909
```

Усредненная метрика:11.266909093657251

```
HistGBMultiSegmentModel(learning_rate = 0.05, max_iter = 100, max_depth = 20, random_state = 42, )
main 12.058172
```

Усредненная метрика:12.058172491644067

```
RFMultiSegmentModel(n_estimators = 100, max_depth = 20, random_state = 42, )
main 11.363349
```

Усредненная метрика:11.363348727514055

```
BRMultiSegmentModel(n_estimators = 100, max_samples = 1.0, max_features = 1.0, bootstrap = True,
bootstrap_features = False, random_state = 42, )
main 11.767457
```

Усредненная метрика:11.767457218444097

Теперь разбиваем набор с продажами по 4 сегментам на обучающую и тестовую выборки с учетом временной структуры, опять создаем экземпляр класса Pipeline, передав ему модель LGBMMultiSegmentModel, ранее созданный список преобразований mult\_preprocess и горизонт mult\_HORIZON. Обучаем модель, получаем прогнозы и визуализируем их.

```
# разбиваем набор на обучающую и тестовую выборки с учетом временной структуры
```

```
train_mult_ts, test_mult_ts = mult_ts.train_test_split(
    test_size=mult_HORIZON)
```

```
# создаем конвейер с моделью и списком преобразований
pipeline = Pipeline(model=model,
                    transforms=mult_preprocess,
                    horizon=mult_HORIZON)
```

```
# обучаем конвейер
```

```
pipeline.fit(train_mult_ts)
```

```
# получаем прогнозы
```

```
forecast_ts = pipeline.forecast()
```

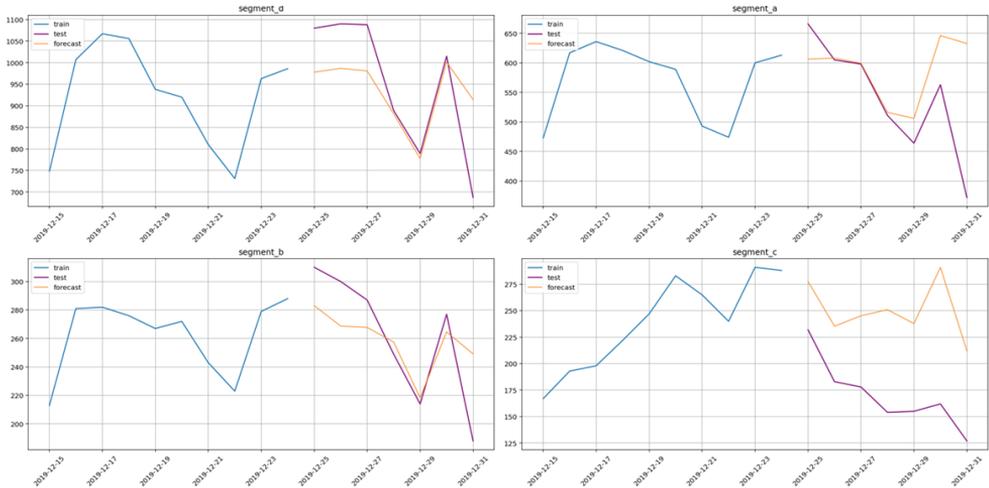
```
# оцениваем качество прогнозов
```

```
print(smape(y_true=test_mult_ts, y_pred=forecast_ts))
```

```
# визуализируем прогнозы
```

```
plot_forecast(forecast_ts, test_mult_ts,
              train_mult_ts, n_train_samples=10)
```

```
{'segment_d': 8.900654173819444, 'segment_a': 12.208450214006076,
 'segment_b': 9.276289390570494, 'segment_c': 38.83811042426918}
```



Опять для сокращения программного кода можно воспользоваться функцией `train_and_evaluate_model()`.

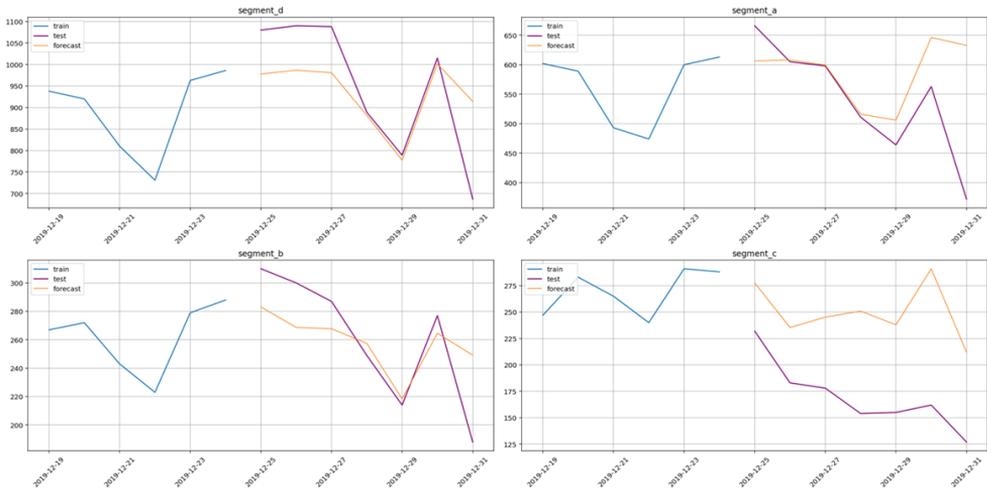
```
# строим и оцениваем модель LGBMMultiSegmentModel
```

```
# с помощью функции train_and_evaluate_model()
```

```
train_and_evaluate_model(
    ts=mult_ts,
    model=model,
    transforms=mult_preprocess,
    horizon=mult_HORIZON,
    metrics=smape,
    print_metrics=True,
    print_plots=True,
    n_train_samples=6)
```

```
segment_d    8.900654
segment_a   12.208450
segment_b    9.276289
segment_c   38.838110
```

Усредненная метрика: 17.3058760506663



Пробуем XGBoost, реализации градиентного бустинга и случайного леса из библиотеки scikit-learn уже для прогнозирования потребления электроэнергии по 4 сегментам.

*# строим модели для нескольких рядов*

```
for model in model_lst:
    print('')
    print(model)
    # строим и оцениваем модель из списка с помощью
    # функции train_and_evaluate_model()
    train_and_evaluate_model(
        ts=mult_ts,
        model=model,
        transforms=mult_preprocess,
        horizon=mult_HORIZON,
        metrics=smape,
        print_metrics=False,
        print_plots=False,
        n_train_samples=None)
```

```
XGBMultiSegmentModel(booster = 'gbtree', max_depth = 3, learning_rate = 0.1, n_estimators = 100, )
segment_d    148.806518
segment_a    121.669349
segment_b     72.658253
segment_c     34.216415
```

Усредненная метрика:94.33763387952567

```
GBMultiSegmentModel(learning_rate = 0.01, n_estimators = 100, max_depth = 20, random_state = 42, )
segment_d    23.105640
segment_a    16.337240
segment_b    17.993655
segment_c    57.698551
```

Усредненная метрика:28.783771421484886

```
HistGBMultiSegmentModel(learning_rate = 0.05, max_iter = 100, max_depth = 20, random_state = 42, )
segment_d    11.610009
segment_a    13.709616
```

```
segment_b 12.684326
segment_c 32.387613
```

Усредненная метрика:17.59789086837028

```
RFMultiSegmentModel(n_estimators = 100, max_depth = 20, random_state = 42, )
segment_d 11.308050
segment_a 13.100380
segment_b 12.799451
segment_c 32.826279
```

Усредненная метрика:17.508539969728798

```
BRMultiSegmentModel(n_estimators = 100, max_samples = 1.0, max_features = 1.0, bootstrap = True,
bootstrap_features = False, random_state = 42, )
segment_d 11.410641
segment_a 13.258087
segment_b 12.634842
segment_c 33.192655
```

Усредненная метрика:17.624056253211485

Разумеется, мы можем ансамблировать собственные классы.

Применим ансамбль бустингов (CatBoost и собственные на базе XGBoost и LightGBM) для прогнозирования продаж австралийского вина.

*# задаем конвейеры*

```
catboost_pipeline = Pipeline(
    model=CatBoostMultiSegmentModel(),
    transforms=preprocess,
    horizon=HORIZON)

lightgbm_pipeline = Pipeline(
    model=LGBMMultiSegmentModel(learning_rate=0.1),
    transforms=preprocess,
    horizon=HORIZON)

xgboost_pipeline = Pipeline(
    model=XGBMultiSegmentModel(learning_rate=0.2,
                                n_estimators=500,
                                max_depth=1),
    transforms=preprocess,
    horizon=HORIZON)
```

*# задаем список конвейеров*

```
pipelines = [catboost_pipeline, lightgbm_pipeline, xgboost_pipeline]
```

*# создаем экземпляр класса VotingEnsemble*

```
voting_ensemble = VotingEnsemble(pipelines=pipelines,
                                weights=[10, 1, 1], n_jobs=1)
```

*# получаем метрики*

```
voting_ensemble_metrics, voting_ensemble_forecasts, _ = voting_ensemble.backtest(
    ts=ts, metrics=[SMAPE()], n_folds=3, aggregate_metrics=True)
```

*# смотрим SMAPE*

```
voting_ensemble_metrics
```

segment	SMAPE
0	main 8.776227

```
# визуализируем результаты
plot_backtest(voting_ensemble_forecasts, ts)
```



Применим ансамбль бустингов (CatBoost и собственные на базе XGBoost и LightGBM) для прогнозирования потребления электроэнергии по 4 сегментам.

```
# задаем конвейеры
m_catboost_pipeline = Pipeline(
    model=CatBoostMultiSegmentModel(),
    transforms=mult_preprocess,
    horizon=mult_HORIZON)

m_lightgbm_pipeline = Pipeline(
    model=LGBMMultiSegmentModel(learning_rate=0.1),
    transforms=mult_preprocess,
    horizon=mult_HORIZON)

m_xgboost_pipeline = Pipeline(
    model=XGBMultiSegmentModel(learning_rate=0.2,
                                n_estimators=500,
                                max_depth=1),
    transforms=mult_preprocess,
    horizon=mult_HORIZON)

# задаем список конвейеров
m_pipelines = [m_catboost_pipeline,
               m_lightgbm_pipeline,
               m_xgboost_pipeline]
```

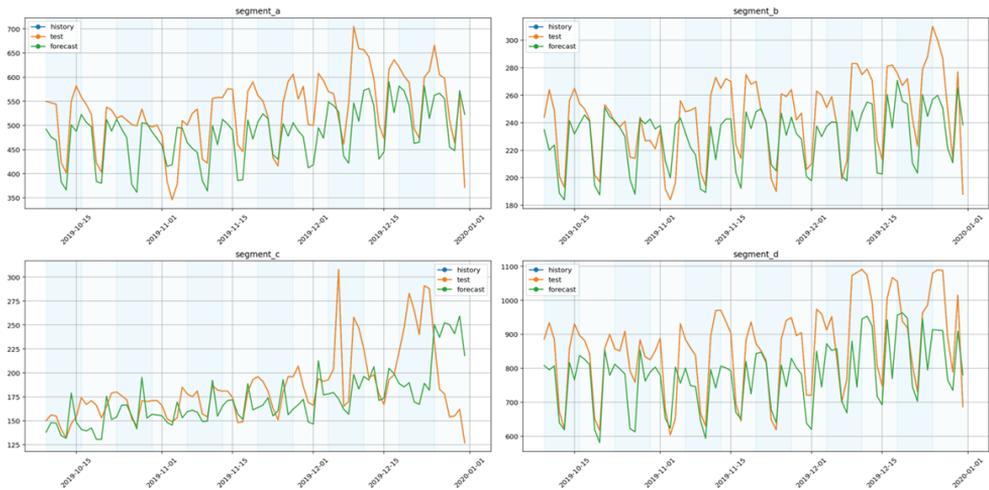
```
# создаем экземпляр класса VotingEnsemble
m_vot_ens = VotingEnsemble(pipelines=m_pipelines,
                           weights=[10, 1, 1], n_jobs=1)

# получаем метрики
m_vot_ens_metrics, m_vot_ens_forecasts, _ = m_vot_ens.backtest(
    ts=mult_ts, metrics=[SMAPE()], n_folds=12, aggregate_metrics=True)

# смотрим значения SMAPE по каждому сегменту
m_vot_ens_metrics
```

	segment	SMAPE
0	segment_a	11.480537
1	segment_b	8.113221
2	segment_c	14.749487
3	segment_d	10.830426

```
# визуализируем результаты
plot_backtest(m_vot_ens_forecasts, mult_ts)
```



## 2.18. ИМПУТАЦИЯ ПРОПУСКОВ

Библиотека ETNA позволяет выполнять импутацию пропусков. Давайте загрузим первый временной ряд продаж из 8 рядов задачи Райффайзен Банка.

```
# загружаем исторические данные о продажах
raif_df = pd.read_csv('Data/advert/sales_train.csv',
                    parse_dates=['date'],
                    usecols=['market_1', 'date'])

# переименовываем столбец date в timestamp
raif_df = raif_df.rename(columns={'date': 'timestamp'})
raif_df.head(10)
```

	timestamp	market_1
0	2008-01-02	1.000000
1	2008-01-03	NaN
2	2008-01-04	1.015188
3	2008-01-05	NaN
4	2008-01-06	0.995762
5	2008-01-07	0.992160
6	2008-01-08	1.004674
7	2008-01-09	1.014755
8	2008-01-10	NaN
9	2008-01-11	NaN

Видим, что временной ряд содержит пропуски.  
Вычислим справочно среднее значение ряда.

```
# вычислим среднее значение ряда
```

```
raif_df['market_1'].mean()
```

```
1.6799237050141351
```

«Расплавляем» исторические данные о продажах в длинный датафрейм.

```
# "расплавляем" исторические данные
```

```
# о продажах в длинный датафрейм
```

```
raif_df = raif_df.melt(id_vars='timestamp',
                      var_name='segment',
                      value_name='target')
```

```
raif_df.head()
```

	timestamp	segment	target
0	2008-01-02	market_1	1.000000
1	2008-01-03	market_1	NaN
2	2008-01-04	market_1	1.015188
3	2008-01-05	market_1	NaN
4	2008-01-06	market_1	0.995762

Конвертируем датафрейм в мультииндексный формат.

```
# конвертируем датафрейм в мультииндексный формат
```

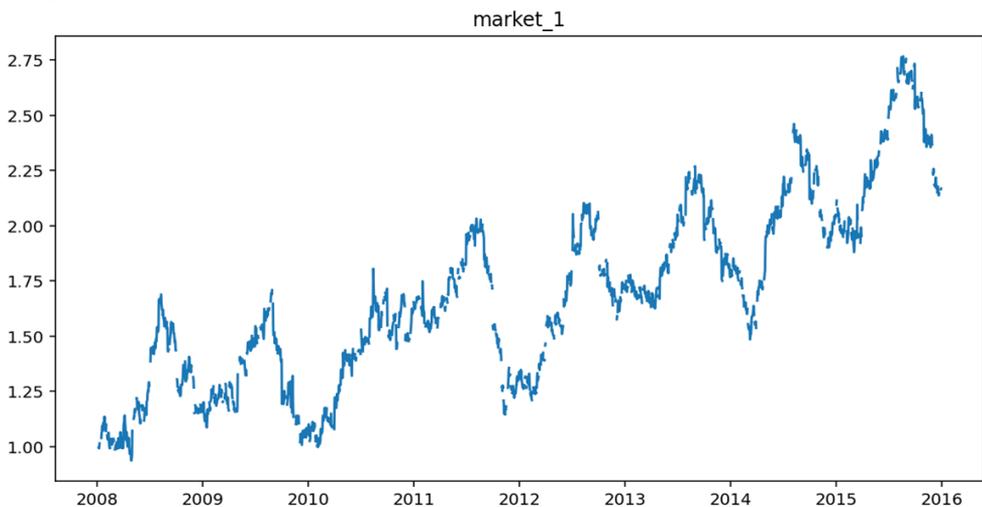
```
raif_df = TSDataset.to_dataset(raif_df)
```

```
raif_df.head()
```

segment	market_1
feature	target
timestamp	
2008-01-02	1.000000
2008-01-03	NaN
2008-01-04	1.015188
2008-01-05	NaN
2008-01-06	0.995762

Теперь превращаем датафрейм в объект `TSDataset`, задав частоту временного ряда, и визуализируем ряд.

```
# превращаем датафрейм в объект TSDataset,
# задав частоту временного ряда
raif_ts = TSDataset(raif_df, freq='D')
# визуализируем ряд
raif_ts.plot()
```



Класс `TimeSeriesImputerTransform` импутирует пропуски. С помощью параметра `in_column` класса-трансформера задаем переменную, пропуски в которой нужно импутировать. Параметр `strategy` задает стратегию импутации пропусков:

- 'zero' – заполняет пропуски нулем;
- 'constant' – заполняет пропуски константным значением;
- 'mean' – заполняет пропуски средним значением;
- 'running\_mean' – заполняет пропуски скользящим средним значением, вычисленным по окну заданной ширины (по умолчанию пропуски будут заполнены скользящим средним значением с шириной окна, определяемой количеством предыдущих значений);

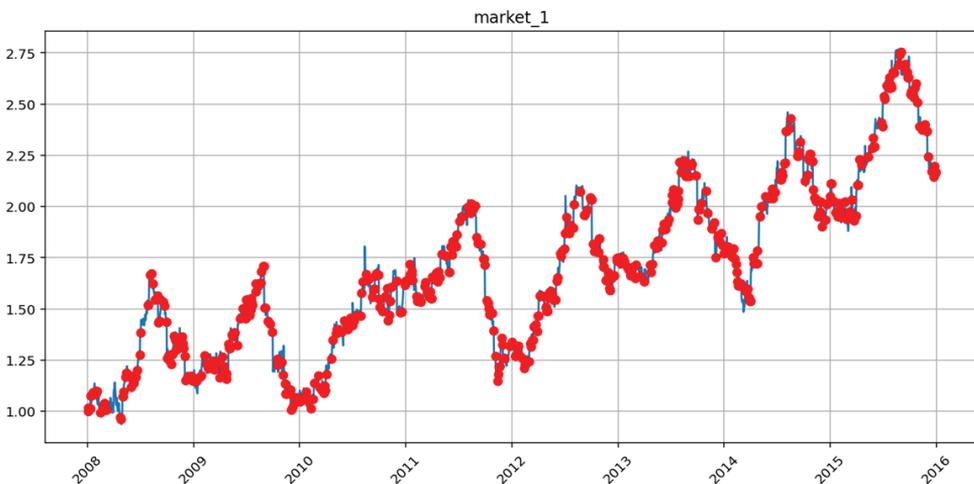
- 'seasonal' – заполняет пропуски сезонным скользящим средним значением;
- 'forward\_fill' – заполняет пропуски последним известным значением.

Параметр `window` задает ширину окна (по умолчанию -1). Если `strategy='running_mean'`, а `window=-1`, то все предыдущие данные принимаются во внимание, в случае любого положительного значения для `window` учитывается `window` предыдущих значений (например, если мы зададим `strategy='running_mean'` и `window=3`, то пропуски будут заполнены скользящим средним с шириной окна 3, т. е. средним, вычисленным по 3 предшествующим значениям).

Давайте попробуем импутацию 'forward\_fill', т. е. пропуски будем заполнять последним известным значением. Помимо класса `TimeSeriesImputerTransform`, воспользуемся функцией `plot_imputation()`, которая позволяет визуализировать результаты импутации (импутированные значения будут представлены в виде красных точек).

```
# пробуем импутацию forward_fill (заполняем последним
# известным значением)

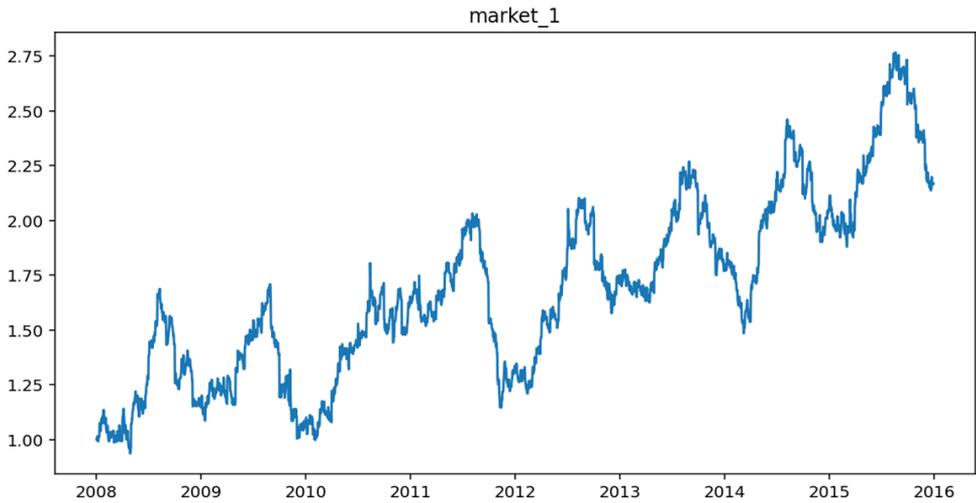
# создаем экземпляр класса TimeSeriesImputerTransform,
# задав strategy='forward_fill'
imputer = TimeSeriesImputerTransform(
    in_column='target',
    strategy='forward_fill')
# визуализируем результаты импутации
plot_imputation(ts=raif_ts, imputer=imputer)
```



Видим, что в данном случае импутация пропусков последним известным значением является довольно адекватным способом восстановления пропусков.

Убедившись в адекватности способа импутации, применяем импутацию к нашему ряду с помощью метода `.fit_transform()`, передав в него экземпляр класса `TimeSeriesImputerTransform`, и визуализируем ряд.

```
# выполняем импутацию
raif_ts.fit_transform([imputer])
# визуализируем ряд
raif_ts.plot()
```



Теперь взглянем на ряд в табличном виде. Здесь хорошо видно, что для импутации пропусков использовалось последнее известное значение.

```
# взглянем на ряд
raif_ts.head(10)
```

segment	market_1	
feature	target	
timestamp	market_1	
2008-01-02	1.000000	1.000000
2008-01-03	1.000000	NaN
2008-01-04	1.015188	1.015188
2008-01-05	1.015188	NaN
2008-01-06	0.995762	0.995762
2008-01-07	0.992160	0.992160
2008-01-08	1.004674	1.004674
2008-01-09	1.014755	1.014755
2008-01-10	1.014755	NaN
2008-01-11	1.014755	NaN

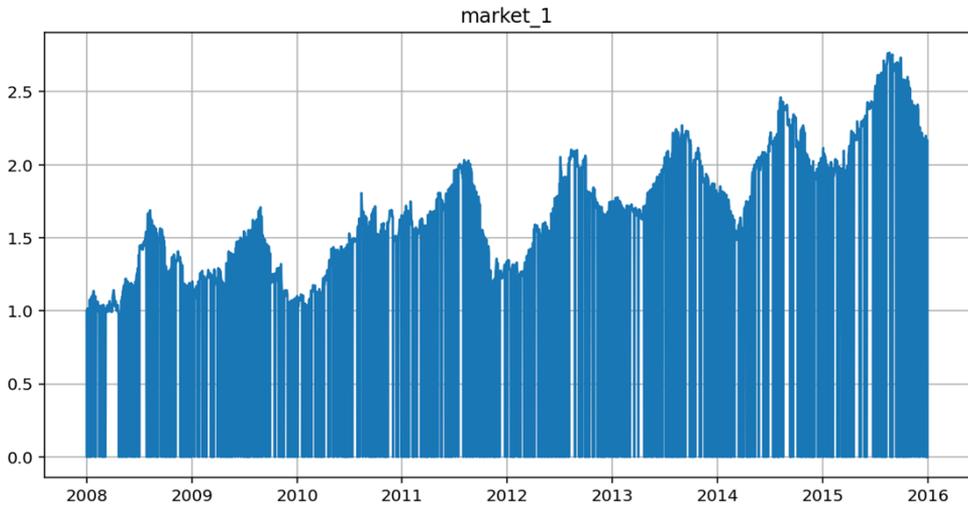
А сейчас напишем функцию импутации для быстрой визуализации результатов и попробуем разные способы импутации для нашего случая.

```
# пишем функцию импутации
def etna_impute(df, strategy='mean', window=-1, seasonality=1):
    # превращаем датафрейм в объект TSDataset,
    # задав частоту временного ряда
    ts = TSDataset(df, freq='D')
    # выполняем импутацию
    imputer = TimeSeriesImputerTransform(
        in_column='target',
        strategy=strategy,
        window=window,
        seasonality=seasonality)
    ts.fit_transform([imputer])
    # визуализируем ряд
    ts.plot()
    return ts
```

Попробуем выполнить импутацию пропусков нулями.

```
# импутируем пропуски нулями
raif_ts = etna_impute(raif_df, strategy='zero')
raif_ts.head(10)
```

segment	market_1	feature	target
timestamp			market_1
2008-01-02	1.000000		1.000000
2008-01-03	0.000000		NaN
2008-01-04	1.015188		1.015188
2008-01-05	0.000000		NaN
2008-01-06	0.995762		0.995762
2008-01-07	0.992160		0.992160
2008-01-08	1.004674		1.004674
2008-01-09	1.014755		1.014755
2008-01-10	0.000000		NaN
2008-01-11	0.000000		NaN



На графике видим, что для нашего случая импутация нулями совершенно не подходит.

Попробуем выполнить импутацию пропусков средним значением.

```
# импутируем пропуски средним значением
raif_ts = etna_impute(raif_df, strategy='mean')
raif_ts.head(10)
```

```
# импутируем пропуски средним значением
raif_ts = etna_impute(raif_df, strategy='mean')
raif_ts.head(10)
```

segment	market_1	
feature	target	
timestamp	market_1	
2008-01-02	1.000000	1.000000
2008-01-03	1.679924	NaN
2008-01-04	1.015188	1.015188
2008-01-05	1.679924	NaN
2008-01-06	0.995762	0.995762
2008-01-07	0.992160	0.992160
2008-01-08	1.004674	1.004674
2008-01-09	1.014755	1.014755
2008-01-10	1.679924	NaN
2008-01-11	1.679924	NaN

```
# вычислим среднее значение ряда
sales_df['market_1'].mean()
```

```
1.6799237050141351
```

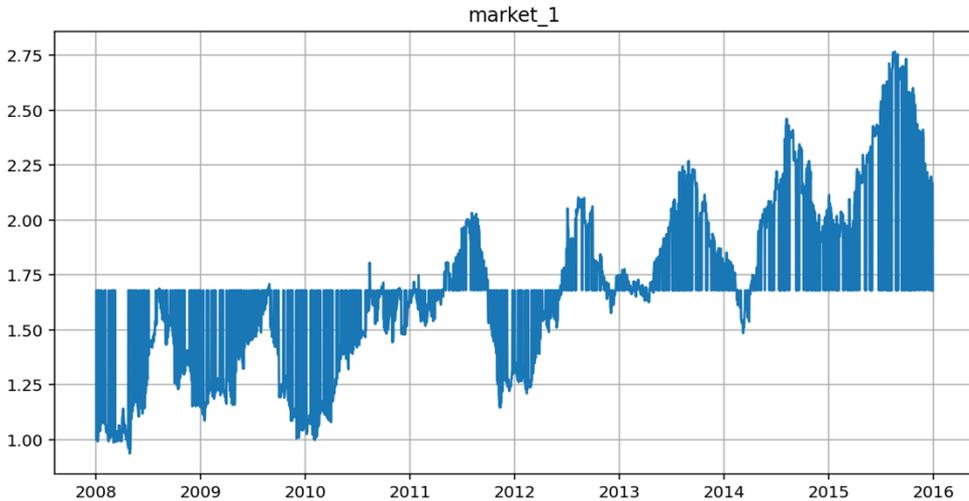
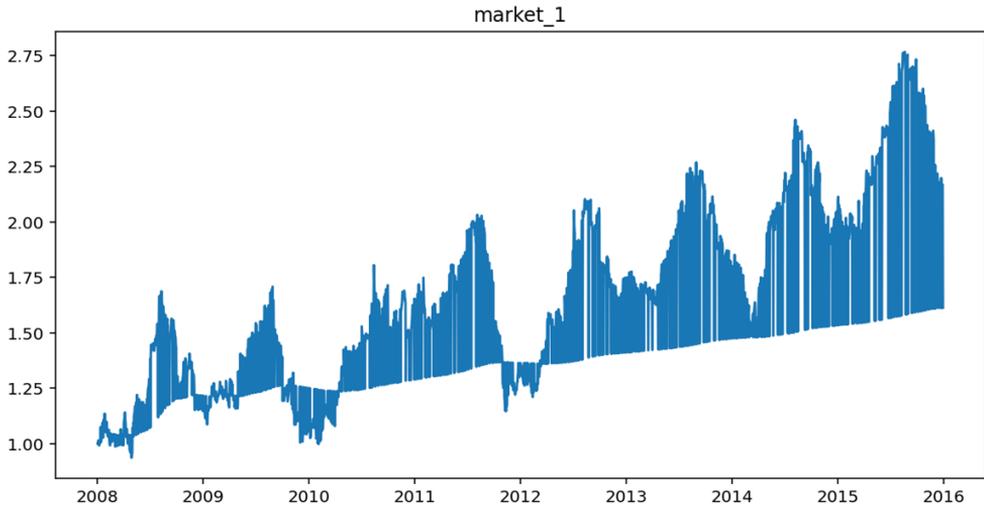


График показывает, что для нашего случая импутация средним тоже совершенно не подходит.

Попробуем выполнить импутацию пропусков скользящим средним.

```
# импутируем скользящим средним, ширина окна
# определяется количеством предыдущих значений
raif_ts = etna_impute(raif_df, strategy='running_mean')
raif_ts.head(10)
```

segment	market_1	
feature	target	
timestamp	market_1	
2008-01-02	1.000000	1.000000
2008-01-03	1.000000	NaN
2008-01-04	1.015188	1.015188
2008-01-05	1.005063	NaN
2008-01-06	0.995762	0.995762
2008-01-07	0.992160	0.992160
2008-01-08	1.004674	1.004674
2008-01-09	1.014755	1.014755
2008-01-10	1.003450	NaN
2008-01-11	1.003450	NaN



Выясним, как были вычислены скользящие средние.

```
# посмотрим, как были вычислены скользящие средние
print(1.000000 / 1)
print((1.000000 + 1.000000 + 1.015188) / 3)
print((1.000000 + 1.000000 + 1.015188 + 1.005063 +
       0.995762 + 0.992160 + 1.004674 + 1.014755) / 8)
print((1.000000 + 1.000000 + 1.015188 + 1.005063 + 0.995762 +
       0.992160 + 1.004674 + 1.014755 + 1.003450) / 9)

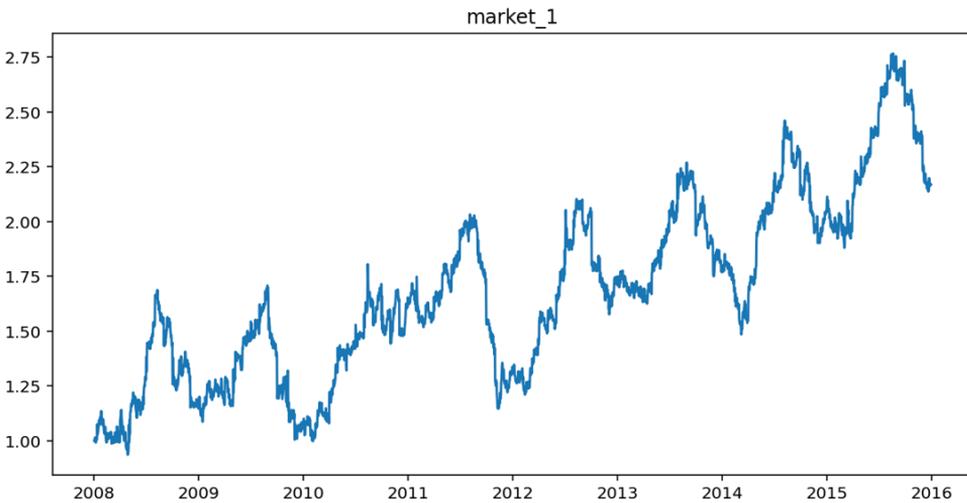
1.0
1.0050626666666667
1.0034502500000002
1.0034502222222226
```

График позволяет понять, что для нашего случая импутация скользящим средним работает чуть адекватнее, необходимо скорректировать ширину окна.

Попробуем выполнить импутацию пропусков скользящим средним с шириной окна 3.

```
# импутируем скользящим средним с шириной окна 3
raif_ts = etna_impute(raif_df, strategy='running_mean', window=3)
raif_ts.head(10)
```

segment	market_1	
feature	target	
timestamp	market_1	
2008-01-02	1.000000	1.000000
2008-01-03	1.000000	NaN
2008-01-04	1.015188	1.015188
2008-01-05	1.005063	NaN
2008-01-06	0.995762	0.995762
2008-01-07	0.992160	0.992160
2008-01-08	1.004674	1.004674
2008-01-09	1.014755	1.014755
2008-01-10	1.003863	NaN
2008-01-11	1.007764	NaN



Вновь выясним, как были вычислены скользящие средние.

```
# посмотрим, как были вычислены скользящие средние
print(1.000000 / 1)
print((1.000000 + 1.000000 + 1.015188) / 3)
print((0.992160 + 1.004674 + 1.014755) / 3)
print((1.004674 + 1.014755 + 1.003863) / 3)

1.0
1.0050626666666667
1.0038630000000002
1.007764
```

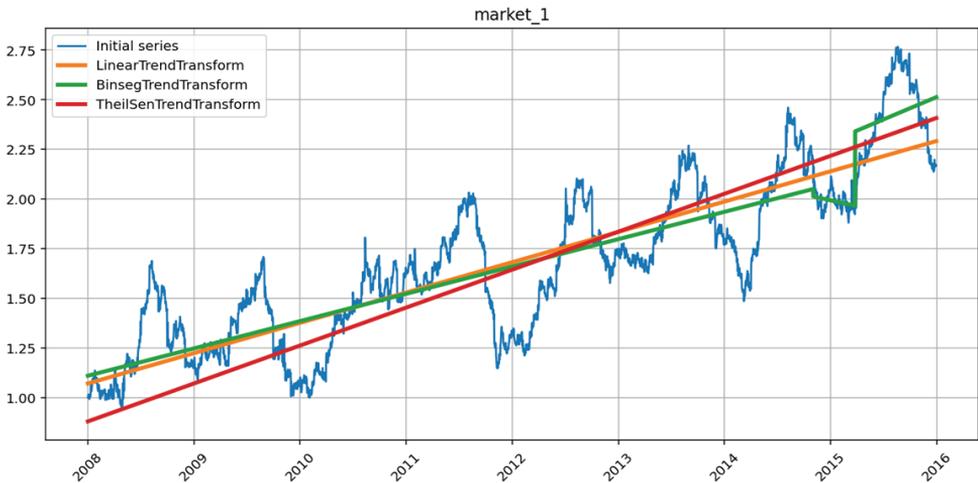
На графике видим, что для нашего случая импутация скользящим средним с шириной окна 3 сработала вполне адекватно.

## 2.19. РАБОТА С ТРЕНДОМ И СЕЗОННОСТЬЮ

Библиотека ETNA предлагает удобную функцию `plot_trend()`, позволяющую вычислить тренд разными способами с последующей визуализацией.

Давайте с помощью этой функции для нашего ряда продаж из задачи Райффайзенбанка вычислим и визуализируем три типа тренда – линейный, кусочно-линейный и линейный, полученный с помощью оценки Тейла-Сена.

```
# вычислим и визуализируем три типа тренда - линейный,
# кусочно-линейный, линейный, полученный
# с помощью оценки Тейла-Сена
plot_trend(
    ts=raif_ts,
    trend_transform=[
        LinearTrendTransform(in_column='target'),
        BinsegTrendTransform(in_column='target', n_bkps=2,
                             min_size=100),
        TheilSenTrendTransform(in_column='target')
    ]
)
```



Исследователям, применяющим градиентный бустинг для прогнозирования временных рядов, следует помнить, что одним из недостатков деревьев и их потомков – случайного леса и градиентного бустинга можно считать принципиальную неспособность деревьев выполнять экстраполяцию при решении задачи регрессии: предсказание для любой комбинации предикторов всегда будет средним значением в одном из листьев, т. е. за пределы диапазона значений зависимой переменной в обучающей выборке выйти невозможно. При работе с временными рядами, содержащими тренд, получится, что на тестовой выборке мы выйдем из диапазона значений, известных модели де-

рева на обучающей выборке, и модель дерева просто будет продолжать предсказывать последнюю известную точку (прогнозы будут представлять собой горизонтальную линию). В этом случае можно воспользоваться процедурой удаления тренда (детрендинг) с последующим восстановлением. Процедура состоит из следующих шагов:

1. На обучающем массиве признаков, в котором единственный признак – номер периода, и обучающем массиве меток (собственно временном ряде) обучаем линейную модель (например, модель линейной регрессии). Прогнозами у нас будут значения тренда для обучающего ряда.
2. Если у нас сезонность является мультипликативной, то значения обучающего массива меток делим на значения тренда для обучающей выборки и получаем обучающий массив меток без тренда. Если у нас сезонность является аддитивной, то из значений обучающего массива меток вычитаем значения тренда для обучающего ряда и получаем обучающий массив меток без тренда.
3. Вычисляем значения тренда для тестового ряда. Берем последнее значение тренда для обучающего ряда и к нему прибавляем второй коэффициент линейной регрессии, отвечающий за шаг, получаем стартовое значение. Создаем пустой список, в который будем записывать значения тренда для тестового ряда. Пишем цикл `for`, на каждой итерации добавляем в созданный список стартовое значение, прибавляем к стартовому значению шаг и обновляем его. Количество итераций определяется горизонтом прогнозирования (длиной тестового ряда). Обратите внимание, что здесь мы нигде не используем значения тестового ряда, мы помним, что тестовый ряд – это прообраз новых данных, о котором мы ничего не знаем.
4. На обучающем массиве признаков и обучающем массиве меток без тренда обучаем модель градиентного бустинга. Затем получаем прогнозы для тестового массива признаков.
5. Наконец, выполняем восстановление тренда. Если у нас была аддитивная сезонность, мы к прогнозам градиентного бустинга для тестового ряда прибавляем значения тренда для тестового ряда (потому что ранее вычитали тренд). Если у нас была мультипликативная сезонность, мы умножаем прогнозы градиентного бустинга для тестового ряда на значения тренда для тестового ряда (потому что ранее делили на тренд).

Данную процедуру можно выполнить с помощью классов `LinearTrendTransform`, `BinsegTrendTransform` и `TheilSenTrendTransform`. Различие между ними заключается в том, что `LinearTrendTransform` вычисляет линейный тренд с помощью линейной регрессии (используется класс `LinearRegression` библиотеки `scikit-learn`), `BinsegTrendTransform` находит точки изменения тренда и вычисляет кусочно-линейный тренд (используется класс `Binseg` библиотеки `ruptures`), а `TheilSenTrendTransform` вычисляет линейный тренд с помощью линейной регрессии, в которой выбирается медиана наклонов всех прямых, проходящих через пары точек выборки на плоскости, этот подход называется методом оценочной функции Тейла-Сена (используется класс `TheilSenRegressor` библиотеки `scikit-learn`).

Часто процедуру детрендинга сочетают с процедурой логарифмирования: сначала выполняют логарифмирование, а затем детрендинг. Логарифмирование позволяет стабилизировать дисперсию временного ряда, особенно это актуально, когда дисперсия возрастает со временем и это может ухудшить прогноз как линейной модели, так и модели на основе деревьев решений. Если выполняется логарифмирование, то прогнозы с восстановленным трендом необходимо экспоненцировать. Мы можем выполнить процедуру логарифмирования с помощью класса `LogTransform`.

Кроме того, библиотека `ETNA` предлагает инструменты для моделирования сезонности.

Напомним, что сезонные колебания или сезонная компонента – регулярные изменения уровня ряда с постоянным периодом. Этим периодом может быть час, день, неделя, месяц, квартал, год. Сезонность может быть аддитивной (амплитуда сезонных колебаний остается неизменной) и мультипликативной (амплитуда сезонных колебаний варьирует). Сезонность всегда фиксирована и известна.

Сезонность может быть привязана к определенному календарному временному интервалу.

Например, сезонность может быть:

- годовой (продажи мороженого или кондиционеров, пик продаж – в летние месяцы и максимальный спад продаж – в зимние месяцы);
- месячной (продажи в дни выдачи зарплат в моногородах, городах с 2-3 крупными предприятиями – каждого 1-го и каждого 15-го числа месяца);
- недельной (объем поездок на велосипедах – подъем в будние дни, снижение в выходные дни);
- дневной (пик пассажиропотока в утренние часы, спад в середине дня и подъем в вечерние часы).

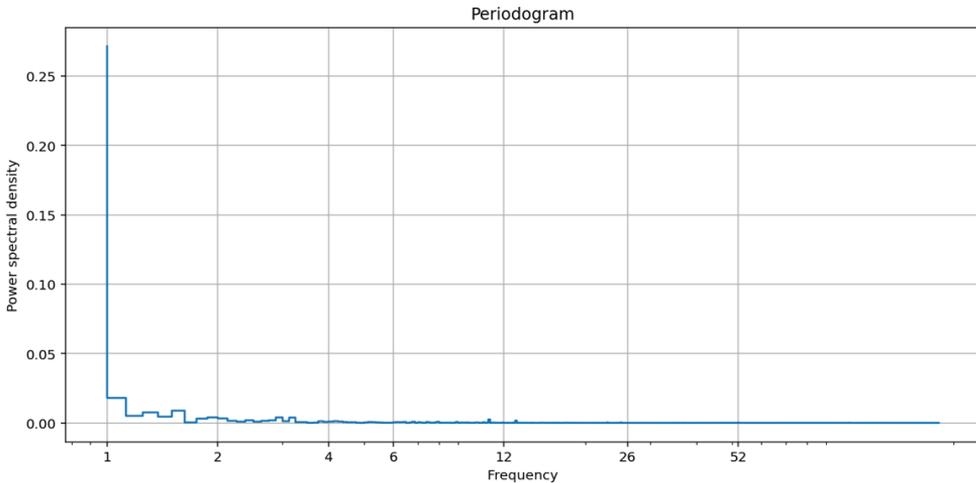
Данные могут содержать смесь сезонностей. Например, продажи пива характеризуются годовой (летом пиво покупают чаще) и недельной (в выходные пиво покупают чаще) сезонностями.

Сезонность может относиться к какому-либо событию, которое прямо не соотносится с конкретными календарными интервалами (продажа зонтов и плащей в сезон дождей, продажи в Черную пятницу и День города).

Сезонность можно моделировать с помощью компонент ряда Фурье.

Для этого с помощью функции `plot_periodogram()` нашей библиотеки строим периодограмму с амплитудами компонент ряда Фурье для временного периода.

```
# строим периодограмму
plot_periodogram(raif_ts,
                 period=365.24,
                 xticks=[1, 2, 4, 6, 12, 26, 52])
```



Видим пик в районе 1. У нас – годовая сезонность.

Сезонность моделируется через количество периодов – временных интервалов в сезонном цикле. Допустим, мы работаем с ежедневным рядом. Если у нас – недельная сезонность, мы зададим 7 периодов. Если у нас – годовая сезонность, мы зададим 365 периодов (более точно 365,2425). В помощь – таблица ниже.

Таблица 2 Определение количества периодов

Частота ряда	Сезонный цикл			
	Неделя	Месяц	Квартал	Год
Ежеквартальный				4
Ежемесячный			3	12
Еженедельный		4.348125	13.044375	52.1775
Ежедневный	7	30.436875	91.310625	365.2425

С помощью класса `FourierTransform` подбираем порядок (параметр `order`), т. е. количество компонентов (членов) ряда Фурье с периодом, соответствующим обнаруженной годовой сезонности, т. е. с периодом 365,24 (параметр `period`).

Сезонность можно моделировать не только с помощью компонент ряда Фурье, но и с помощью календарных признаков. Здесь для простоты под календарными признаками понимаются признаки времени в рамках 24-часового суточного цикла и признаки дат в рамках годового цикла. Для дневной сезонности можно взять час, время суток, для недельной сезонности – порядковый номер дня недели (в виде числа или строкового значения), для месячной сезонности – порядковый номер дня месяца. Для квартальной сезонности – порядковый номер дня квартала. Для годовой сезонности можно использовать порядковый номер дня года, порядковый номер недели, порядковый номер месяца.

Признаки времени в рамках 24-часового суточного цикла можно создать с помощью параметров класса `TimeFlagsTransform`:

- `minute_in_hour_number` – порядковый номер минуты в часе;
- `fifteen_minutes_in_hour_number` – порядковый номер 15-минутного интервала в часе;
- `hour_number` – порядковый номер часа в сутках;
- `half_hour_number` – порядковый номер половины часа: 0 – для первой половины часа, 1 – для второй половины часа;
- `half_day_number` – порядковый номер половины суток: 0 – для первой половины суток, 1 – для второй половины суток;
- `one_third_day_number` – порядковый номер 8-часового интервала (нумерация начинается с 0).

Признаки дат в рамках годового цикла можно создать с помощью параметров класса `DateFlagsTransform`:

- `day_number_in_week` – порядковый номер дня недели;
- `day_number_in_month` – порядковый номер дня месяца;
- `day_number_in_year` – порядковый номер дня года (для високосного года варьирует от 1 до 366);
- `week_number_in_month` – порядковый номер недели в месяце;
- `week_number_in_year` – порядковый номер недели в году;
- `month_number_in_year` – порядковый номер месяца в году;
- `season_number` – порядковый номер сезона;
- `year_number` – номер года;
- `is_weekend` – индикатор выходного дня;
- `special_days_in_week` – список специальных дней в неделе ([0, 6], например у нас самые высокие продажи наблюдаются по понедельникам и средам, задаем [0, 2]);
- `special_days_in_month` – список специальных дней в месяце ([1, 31], например у нас выплаты зарплаты происходят каждого 1-го и 15-го, задаем [1, 15]).

Кроме того, класс `HolidayTransform` позволяет учитывать национальные праздники (параметр `iso_code` задает ISO-код страны).

Вычислим временные рамки нашего временного ряда продаж.

```
# взглянем на временные рамки ряда
print('временные рамки ряда:',
      raif_ts.index[0].strftime('%Y-%m-%d'),
      raif_ts.index[-1].strftime('%Y-%m-%d'))
```

временные рамки ряда: 2008-01-02 2015-12-31

Допустим, наш горизонт прогнозирования равен 90 дням.

```
# задаем горизонт прогнозирования
raif_HORIZON = 90
```

Вычислим стартовую точку тестовой выборки и стартовую точку валидационной выборки.

```
# вычисляем стартовую точку тестовой выборки
start_test = datetime.date(2015, 12, 31) - datetime.timedelta(
    raif_HORIZON - 1)
start_test

datetime.date(2015, 10, 3)

# вычисляем стартовую точку валидационной выборки
start_valid = datetime.date(2015, 10, 2) - datetime.timedelta(
    raif_HORIZON - 1)
start_valid

datetime.date(2015, 7, 5)
```

Стартовые точки выборок еще можно вычислить так.

```
# вычисляем стартовую точку тестовой выборки
raif_df.iloc[-raif_HORIZON:].index[0]

Timestamp('2015-10-03 00:00:00')

# вычисляем стартовую точку валидационной выборки
raif_df.iloc[-(2 * raif_HORIZON)].index[0]

Timestamp('2015-07-05 00:00:00')
```

Теперь мы разбиваем набор на обучающую и валидационную выборки, учитывая временную структуру и информацию о стартовой точке валидационной выборки.

```
# разбиваем набор на обучающую и валидационную выборки
# с учетом временной структуры
train_raif_ts, valid_raif_ts = raif_ts.train_test_split(
    train_start='2008-01-02',
    train_end='2015-07-04',
    test_start='2015-07-05',
    test_end='2015-10-02')
```

На этот раз наш набор преобразований/признаков будет посложнее: мы выполним логарифмирование зависимой переменной с последующим детрендингом, добавим лаги, скользящие средние, признаки дат, праздники, компоненты ряда Фурье в обучающую выборку.

```
# создаем экземпляр класса LogTransform
# для логарифмирования зависимой переменной
log = LogTransform(in_column='target')

# создаем экземпляр класса LinearTrendTransform для детрендинга
detrend = LinearTrendTransform(in_column='target')

# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 90 до 270
# с шагом 30, порядок лагов не должен быть меньше длины горизонта
lags = LagTransform(in_column='target',
                    lags=list(range(90, 271, 30)),
                    out_column='lag')
```

```
# создаем экземпляры класса MeanTransform для вычисления  
# скользящих средних  
mean90 = MeanTransform(in_column='target',  
                        window=90,  
                        out_column='mean90')  
  
mean150 = MeanTransform(in_column='target',  
                        window=150,  
                        out_column='mean150')  
  
mean210 = MeanTransform(in_column='target',  
                        window=210,  
                        out_column='mean210')  
  
# создаем экземпляр класса DateFlagsTransform для генерации  
# признаков на основе дат - порядковый номер дня недели,  
# порядковый номер дня месяца, порядковый номер месяца в году  
d_flags = DateFlagsTransform(day_number_in_week=True,  
                             day_number_in_month=True,  
                             month_number_in_year=True,  
                             out_column='datetime')  
  
# создаем экземпляр класса HolidayTransform  
# для генерации признаков на основе дат  
holidays = HolidayTransform(iso_code='RUS',  
                             out_column='RUS_holidays')  
  
# создаем экземпляр класса FourierTransform  
# для вычисления членов Фурье  
fourier_year = FourierTransform(period=365.24,  
                                order=3,  
                                out_column='fourier_year')  
  
# задаем список преобразований/признаков  
trans = [log, detrend, lags, mean90, mean150, mean210,  
         d_flags, holidays, fourier_year]  
  
# выполняем логарифмирование, детрендинг, добавляем лаги,  
# скользящие средние, признаки даты, праздники,  
# компоненты Фурье в обучающую выборку  
train_raif_ts.fit_transform(trans)  
train_raif_ts
```

Результат применения  
класса HolidayTransform (6)

Результат применения  
класса DateFlagsTransform (5)

segment	market_1	feature	timestamp	datetime_day_number_in_month	datetime_day_number_in_week	datetime_is_weekend	datetime_month_number_in_year
		RUS_holidays					
			2008-01-02	2	2	False	1
			2008-01-03	3	3	False	1
			2008-01-04	4	4	False	1
			2008-01-05	5	5	True	1
			2008-01-06	6	6	True	1
		...	...	...	...	...	...
			2015-06-30	30	1	False	6
			2015-07-01	1	2	False	7
			2015-07-02	2	3	False	7
			2015-07-03	3	4	False	7
			2015-07-04	4	5	True	7

Результат применения  
класса FourierTransform (7)

fourier_year_1	fourier_year_2	fourier_year_3	fourier_year_4	fourier_year_5	fourier_year_6
0.000000	1.000000	0.000000	1.000000	0.000000	1.000000
0.017202	0.999852	0.034399	0.999408	0.051586	0.998669
0.034399	0.999408	0.068757	0.997633	0.103034	0.994678
0.051586	0.998669	0.103034	0.994678	0.154208	0.988038
0.068757	0.997633	0.137189	0.990545	0.204972	0.978768
...	...	...	...	...	...
0.056739	-0.998389	-0.113295	0.993561	0.169487	-0.985533
0.039556	-0.999217	-0.079051	0.996871	0.118421	-0.992963
0.022362	-0.999750	-0.044713	0.999000	0.067041	-0.997750
0.005161	-0.999987	-0.010322	0.999947	0.015482	-0.999880
-0.012042	-0.999927	0.024082	0.999710	-0.036118	-0.999348

Результат применения класса LagTransform							Результат применения класса MeanTransform			Результат применения класса LogTransform и затем класса LinearTrendTransform
lag_120	lag_150	lag_180	lag_210	lag_240	lag_270	lag_90	mean150	mean210	mean90	target
NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.027939	-0.027939	-0.027939	-0.027939
NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.027970	-0.027970	-0.027970	-0.028002
NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.026907	-0.026907	-0.026907	-0.024779
NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.026937	-0.026937	-0.026937	-0.027030
NaN	NaN	NaN	NaN	NaN	NaN	NaN	-0.027372	-0.027372	-0.027372	-0.029112
...	...	...	...	...	...	...	...	...	...	...
-0.028004	-0.016142	-0.006440	-0.025257	0.019324	0.015790	-0.008227	0.002031	-0.002369	0.017139	0.030671
-0.031865	-0.016988	0.000744	-0.020984	0.002681	0.016215	-0.007893	0.002421	-0.002072	0.017687	0.041429
-0.034333	-0.015123	0.003302	-0.024231	0.013835	0.012140	-0.002174	0.002813	-0.001748	0.018197	0.043754
-0.032513	-0.018520	-0.000904	-0.019133	-0.000417	0.010695	-0.000285	0.003254	-0.001430	0.018730	0.047661
-0.024315	-0.017001	0.000925	-0.019757	-0.002869	0.012897	-0.003477	0.003683	-0.001111	0.019295	0.047319

Теперь обучаем модель CatBoost, получаем прогнозы, оцениваем качество модели и визуализируем прогнозы.

```
# создаем экземпляр класса CatBoostMultiSegmentModel
model = CatBoostMultiSegmentModel()
# обучаем модель CatBoost
model.fit(train_raif_ts)
# формируем набор, для которого нужно получить прогнозы,
# длина набора определяется горизонтом прогнозирования
future_ts = train_raif_ts.make_future(raif_HORIZON)
# получаем прогнозы
forecast_ts = model.forecast(future_ts)
# оцениваем качество прогнозов на валидационной выборке
smape(y_true=valid_raif_ts, y_pred=forecast_ts)

{'market_1': 5.029572625200159}
```

Давайте выясним, как происходят логарифмирование и детрендинг зависимой переменной.

Вернемся к исходному обучающему массиву меток.

```
# вернемся к исходному обучающему массиву меток
train = raif_ts.to_pandas(flatten=True)[
    ['timestamp', 'target']].iloc[:-(2 * raif_HORIZON)]
train = train.set_index('timestamp', drop=True)
train
```

	target
timestamp	
2008-01-02	1.000000
2008-01-03	1.000000
2008-01-04	1.015188
2008-01-05	1.005063
2008-01-06	0.995762
...	...
2015-06-30	2.402702
2015-07-01	2.488553
2015-07-02	2.507788
2015-07-03	2.539997

Теперь выполняем логарифмирование зависимой переменной.

```
# выполняем логарифмирование зависимой переменной
train['target'] = np.log1p(train['target']) / np.log(10)
train
```

	target
timestamp	
2008-01-02	0.301030
2008-01-03	0.301030
2008-01-04	0.304316
2008-01-05	0.302128
2008-01-06	0.300109
...	...
2015-06-30	0.531824
2015-07-01	0.542645
2015-07-02	0.545033
2015-07-03	0.549003
2015-07-04	0.548724

2741 rows × 1 columns

Теперь выполняем детрендинг. Сначала создаем конвейер, состоящий из классов `PolynomialFeatures` и `LinearRegression`, по сути, мы строим линейную регрессию, используя в качестве обучающего массива признаков константный признак (столбец из единиц) и даты, переведенные в вещественные значения, а обучающим массивом меток будет прологарифмированная зависимая переменная. С помощью обученного конвейера прогнозируем тренд для обучающей выборки и вычитаем его из прологарифмированной зависимой переменной.

```

# задаем степень полинома
poly_degree = 1

# создаем конвейер
pipeline = Pipeline_sklern(
    [('polynomial', PolynomialFeatures(degree=1,
                                       include_bias=False)),
     ('regressor', LinearRegression())]
)

# пишем функцию, превращающую временные метки
# в вещественные значения - признак для
# линейной регрессии
def get_x(df) -> np.ndarray:
    series_len = len(df)
    x = df.index.to_series()
    # проверка типа
    if isinstance(type(x.dtype), pd.Timestamp):
        raise ValueError(f'Your timestamp column has wrong format.'
                        f'Need np.datetime64 or datetime.datetime')
    x = x.apply(lambda ts: ts.timestamp())
    # переводим в 2D-массив NumPy
    x = x.to_numpy().reshape(series_len, 1)
    return x

# получаем из временных меток вещественные значения,
# по сути, получаем обучающий массив признаков
x_train = get_x(train)
x_train

array([[1.1992320e+09],
       [1.1993184e+09],
       [1.1994048e+09],
       ...,
       [1.4357952e+09],
       [1.4358816e+09],
       [1.4359680e+09]])

# получаем обучающий массив меток
y_train = train['target'].values

# обучаем конвейер
pipeline.fit(x_train, y_train)
# прогнозируем тренд для обучающей выборки
train_trend = pipeline.predict(x_train)
train_trend

array([0.32896897, 0.3290319 , 0.32909484, ..., 0.50127907, 0.501342 ,
       0.50140493])

# удаляем тренд из зависимой переменной
train['target'] = y_train - train_trend
train

```

target	
timestamp	
2008-01-02	-0.027939
2008-01-03	-0.028002
2008-01-04	-0.024779
2008-01-05	-0.027030
2008-01-06	-0.029112
...	...
2015-06-30	0.030671
2015-07-01	0.041429
2015-07-02	0.043754
2015-07-03	0.047661
2015-07-04	0.047319

Взглянем на первые 5 прогнозов.

```
# взглянем на первые 5 прогнозов
forecast_ts.loc[:, pd.IndexSlice[:, 'target']].head()
```

segment	market_1
feature	target
timestamp	
2015-07-05	2.580451
2015-07-06	2.485676
2015-07-07	2.475557
2015-07-08	2.456239
2015-07-09	2.474086

Выясним, как были получены прогнозы. Для этого нам нужно подготовить обучающие массив признаков и массив меток, удалить наблюдения с пропусками и затем обучить модель CatBoost.

```
# получаем обучающий набор в плоском формате
train = train_raif_ts.to_pandas(flatten=True)
# переменную timestamp переводим в индекс
train = train.set_index('timestamp', drop=True)
# удаляем переменную segment
train.drop('segment', axis=1, inplace=True)
# по умолчанию удаляем наблюдения с пропусками
train = train.dropna()
# создаем обучающие массив меток и массив
# признаков для модели CatBoost
y_train = train.pop('target')

# создаем экземпляр класса CatBoostRegressor
ctbst = CatBoostRegressor()
```

```

# определяем категориальные признаки
# (для модели Catboost)
cat_feat = train.select_dtypes(
    include=['category']).columns.to_list()

# задаем набор для обучения
train_pool = Pool(train,
                  y_train,
                  cat_features=cat_feat)

# обучаем модель Catboost
ctbst.fit(train_pool, silent=True);

```

Теперь подготовим валидационный массив признаков, для которого нужно получить прогнозы с помощью модели CatBoost.

```

# получаем валидационный набор в плоском формате
valid = future_ts.to_pandas(flatten=True)
valid = valid.set_index('timestamp', drop=True)
# получаем валидационный массив признаков для модели CatBoost
valid.drop(['segment', 'target'], axis=1, inplace=True)

```

Получаем сырые прогнозы CatBoost для валидационного массива признаков.

```

# получаем сырые прогнозы для валидационного
# массива признаков
raw_pred = ctbst.predict(valid)
raw_pred

array([ 0.05246986,  0.04075621,  0.03943064,  0.0369471 ,  0.03912102,
        0.04143967,  0.04262432,  0.04746484,  0.03632904,  0.04255608,
        0.04111785,  0.03951881,  0.04411646,  0.04139433,  0.04531096,
        0.04242163,  0.04832866,  0.04347636,  0.03952975,  0.04860329,
        0.04018305,  0.04943478,  0.03850779,  0.04457708,  0.0410882 ,
        0.03568354,  0.04912936,  0.04937777,  0.04146886,  0.04149474,
        0.04990667,  0.05334586,  0.05361552,  0.04949207,  0.05646728,
        0.04673181,  0.04755111,  0.04642787,  0.05305291,  0.0532577 ,
        0.05698076,  0.05668822,  0.04821247,  0.04456592,  0.05309563,
        0.05419545,  0.04789977,  0.05987839,  0.05754141,  0.04975479,
        0.0474137 ,  0.0514482 ,  0.05017209,  0.05152579,  0.06024912,
        0.05413196,  0.05462657,  0.05059076,  0.04758791,  0.04059275,
        0.04899278,  0.04740089,  0.04458993,  0.04175516,  0.03959055,
        0.0368722 ,  0.04040923,  0.03707813,  0.03639016,  0.0349144 ,
        0.0321418 ,  0.03214884,  0.03151594,  0.03512916,  0.02949367,
        0.0341175 ,  0.0322319 ,  0.03220787,  0.03289555,  0.03092776,
        0.03639273,  0.03499965,  0.03375239,  0.0324428 ,  0.03552419,
        0.03163535,  0.03380777,  0.03684376,  0.01621757, -0.00988179])

```

Теперь вычислим тренд для валидационной выборки. Для этого нам требуются временные метки валидационного массива признаков, превращенные в вещественные значения.

```

# получаем из временных меток вещественные значения,
# по сути, получаем валидационный массив
# признаков для линейной регрессии
x_valid = get_x(valid)

```

```
# прогнозируем тренд для валидационной выборки
```

```
valid_trend = pipeline.predict(x_valid)
valid_trend
```

```
array([0.50146787, 0.5015308 , 0.50159373, 0.50165666, 0.5017196 ,
        0.50178253, 0.50184546, 0.5019084 , 0.50197133, 0.50203426,
        0.50209719, 0.50216013, 0.50222306, 0.50228599, 0.50234893,
        0.50241186, 0.50247479, 0.50253772, 0.50260066, 0.50266359,
        0.50272652, 0.50278946, 0.50285239, 0.50291532, 0.50297825,
        0.50304119, 0.50310412, 0.50316705, 0.50322998, 0.50329292,
        0.50335585, 0.50341878, 0.50348172, 0.50354465, 0.50360758,
        0.50367051, 0.50373345, 0.50379638, 0.50385931, 0.50392225,
        0.50398518, 0.50404811, 0.50411104, 0.50417398, 0.50423691,
        0.50429984, 0.50436278, 0.50442571, 0.50448864, 0.50455157,
        0.50461451, 0.50467744, 0.50474037, 0.50480331, 0.50486624,
        0.50492917, 0.5049921 , 0.50505504, 0.50511797, 0.5051809 ,
        0.50524384, 0.50530677, 0.5053697 , 0.50543263, 0.50549557,
        0.5055585 , 0.50562143, 0.50568437, 0.5057473 , 0.50581023,
        0.50587316, 0.5059361 , 0.50599903, 0.50606196, 0.5061249 ,
        0.50618783, 0.50625076, 0.50631369, 0.50637663, 0.50643956,
        0.50650249, 0.50656543, 0.50662836, 0.50669129, 0.50675422,
        0.50681716, 0.50688009, 0.50694302, 0.50700595, 0.50706889])
```

Добавляем к сырым прогнозам тренд и выполняем экспоненцирование.

```
# добавляем к сырым прогнозам тренд
```

```
# и выполняем экспоненцирование
```

```
prediction = np.expm1((raw_pred + valid_trend) * np.log(10))
prediction
```

```
array([2.58045094, 2.48567592, 2.47555664, 2.45623901, 2.47408641,
        2.49318991, 2.50323916, 2.54301697, 2.45382529, 2.50421171,
        2.49313241, 2.48079893, 2.51835387, 2.49687675, 2.52906685,
        2.5061742 , 2.55470409, 2.51571843, 2.48441924, 2.55849923,
        2.49067622, 2.5663522 , 2.4782451 , 2.5277061 , 2.49998721,
        2.45720174, 2.56642814, 2.56898583, 2.50508734, 2.50580421,
        2.57488863, 2.60383285, 2.60659385, 2.57303046, 2.63140648,
        2.55142234, 2.55864411, 2.54996648, 2.6050577 , 2.60728074,
        2.63886493, 2.63694162, 2.56716761, 2.53785396, 2.60854845,
        2.61822269, 2.56666684, 2.66694247, 2.64779184, 2.58349137,
        2.56474288, 2.59853426, 2.58849599, 2.60022049, 2.67379874,
        2.62293994, 2.627594 , 2.5945605 , 2.57030965, 2.5137726 ,
        2.58291595, 2.57032423, 2.5478041 , 2.52523276, 2.50821432,
        2.4868294 , 2.51585266, 2.48949429, 2.48447581, 2.47315876,
        2.45155628, 2.45211247, 2.44758491, 2.47689141, 2.43256338,
        2.46980704, 2.45527535, 2.45558487, 2.46156252, 2.44641299,
        2.4905609 , 2.47988648, 2.47040973, 2.46046207, 2.48560694,
        2.45503544, 2.47286466, 2.49773401, 2.33598064, 2.14186195])
```

Сейчас мы создадим еще дополнительные списки преобразований/ признаков и с помощью перекрестной проверки выберем наилучший набор преобразований/признаков.

*# задаем еще списки преобразований/признаков*

```
trans2 = [detrend, lags, mean90, mean150, mean210,
          d_flags, holidays, fourier_year]
```

```
trans3 = [log, detrend, lags, mean90, d_flags,
          holidays, fourier_year]
```

*# выполняем подбор гиперпараметра - набора преобразований/*

*# признаков с помощью перекрестной проверки*

```
etna_cv_optimize(ts=raif_ts,
                 model=model,
                 horizon=raif_HORIZON,
                 transfrms=[trans, trans2, trans3],
                 n_folds=10,
                 mode='expand',
                 metrics=SMAPE(),
                 refit=True)
```

trans:

```
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [90, 120, 150, 180, 210, 240, 270], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 90, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean90', ), MeanTransform(in_column = 'target', window = 150, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean150', ), MeanTransform(in_column = 'target', window = 210, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean210', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False, month_number_in_year = True, season_number = False, year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), FourierTransform(period = 365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', )]
```

SMAPE\_mean: 5.23530844863599

SMAPE\_std: 2.557166962690118

trans:

```
[LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [90, 120, 150, 180, 210, 240, 270], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 90, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean90', ), MeanTransform(in_column = 'target', window = 150, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean150', ), MeanTransform(in_column = 'target', window = 210, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean210', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month = False, week_number_in_month = False, week_number_in_year = False, month_number_in_year = True, season_number = False, year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), FourierTransform(period = 365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', )]
```

SMAPE\_mean: 6.087109478370556

SMAPE\_std: 2.5703484840407893

trans:

```
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [90, 120, 150, 180, 210, 240, 270], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 90, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0,
```

```

out_column = 'mean90', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month
= True, day_number_in_year = False, week_number_in_month = False, week_number_in_year =
False, month_number_in_year = True, season_number = False, year_number = False, is_weekend
= True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ),
HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), FourierTransform(period =
365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', ))
SMAPE_mean: 5.408665376852765
SMAPE_std: 2.28434588988295

```

Наилучший набор преобразований/признаков:

```

{'trans': [LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None,
), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column
= 'target', lags = [90, 120, 150, 180, 210, 240, 270], out_column = 'lag', ), MeanTrans-
form(in_column = 'target', window = 90, seasonality = 1, alpha = 1, min_periods = 1, fillna =
0, out_column = 'mean90', ), MeanTransform(in_column = 'target', window = 150, seasonality =
1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean150', ), MeanTransform(in_column
= 'target', window = 210, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_col-
umn = 'mean210', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month =
True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False,
month_number_in_year = True, season_number = False, year_number = False, is_weekend = True,
special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), Holi-
dayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), FourierTransform(period =
365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', )]}

```

Лучшее значение SMAPE cv: 5.2353

```

{'market_1': 4.880159484974012}

```



## 2.20. ОБРАБОТКА ВЫБРОСОВ

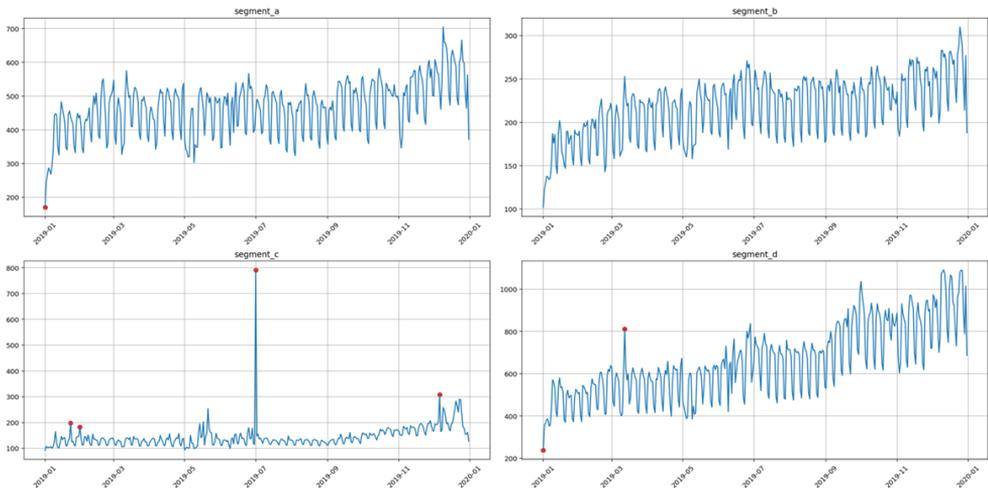
С помощью функций `get_anomalies_median()`, `get_anomalies_density()` и `get_anomalies_hist()` мы можем попробовать разные способы обработки выбросов. Они используются только для разведочного анализа. Мы продемонстрируем работу этих функций на примере потребления электроэнергии по 4 сегментам.

Сначала с помощью функции `get_anomalies_median()` проанализируем выбросы на основе медианного способа. Согласно этому способу выбросы – это все точки, отклоняющиеся от медианы более чем на  $\alpha * \text{std}$ . Поясним некоторые параметры:

- `alpha` – это порог (по умолчанию 3);
- `std` – выборочная дисперсия в окне размером `window_size`;
- `window_size` – размер окна или количество наблюдений (по умолчанию 10).

Дополнительно нам понадобится функция `plot_anomalies()`, которая позволяет визуализировать выбросы (выбросы будут представлены в виде красных точек).

```
# проанализируем выбросы с помощью медианного способа,
# согласно которому выбросы - это все точки, отклоняющиеся
# от медианы более чем на alpha * std, где alpha - порог
# (по умолчанию 3), std - выборочная дисперсия в window_size,
# window_size - размер окна (по умолчанию 10)
anomaly_dict = get_anomalies_median(mult_ts, window_size=100)
plot_anomalies(mult_ts, anomaly_dict)
```

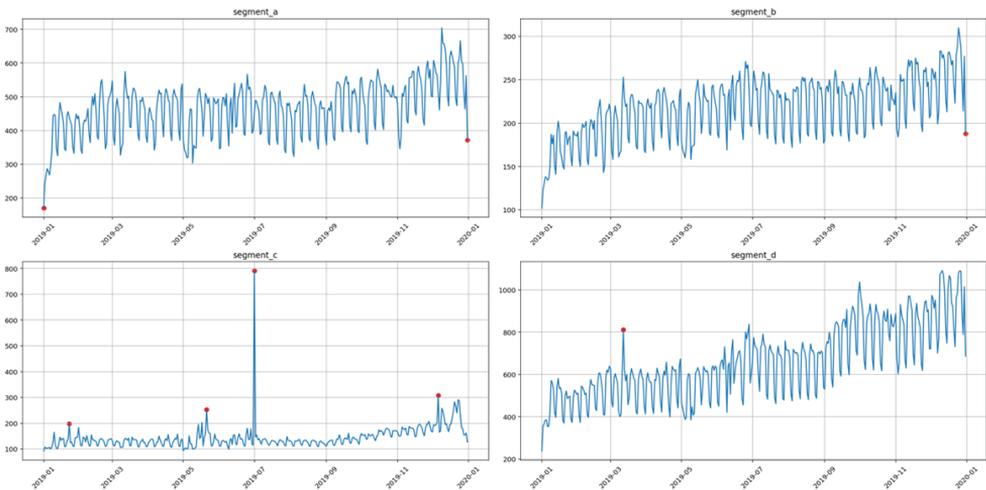


Теперь с помощью функции `get_anomalies_density()` проанализируем выбросы на основе плотности. Согласно этому способу для каждой точки (элемента) ряда мы строим все окна размера `window_size`, содержащие эту точку, и если какое-либо из окон содержит по крайней мере `n_neighbors`, которые находятся ближе, чем  $\text{distance\_coef} * \text{std}(\text{series})$  до интересующей точки в соответствии с `distance_func`, то интересующая нас точка не является выбросом. Поясним некоторые параметры:

- `window_size` – размер окна (по умолчанию 15);
- `n_neighbors` – минимальное количество соседей точки, необходимое для того, чтобы точка не была объявлена выбросом (по умолчанию 3);
- `distance_coef` – множитель для стандартного отклонения, который формирует пороговое значение расстояния, при котором мы считаем точки близко расположенными друг к другу (по умолчанию 3);
- `distance_func` – функция расстояния, которой по умолчанию является абсолютная разница между двумя значениями.

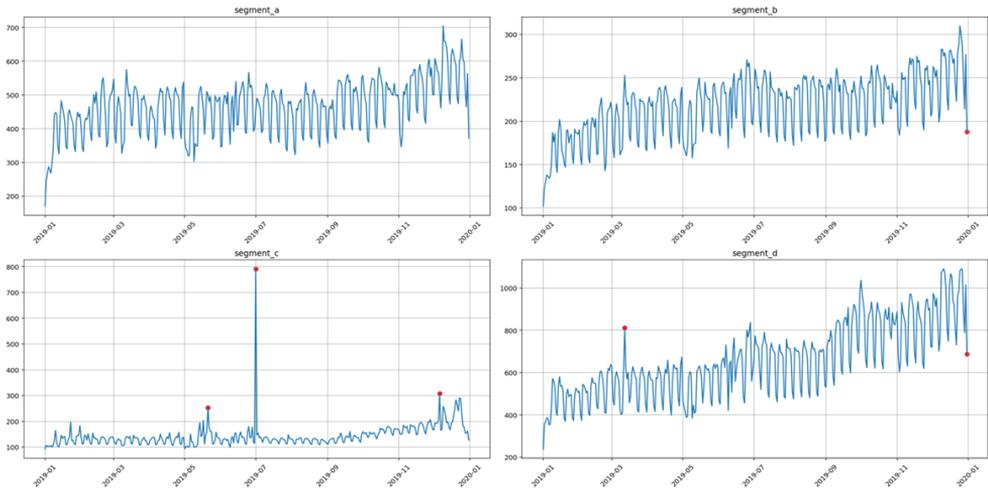
```
# проанализируем выбросы с помощью способа на основе
# плотности: для каждой точки ряда мы строим
# все окна размера window_size, содержащие эту точку, и
# если какое-либо из окон содержит по крайней мере
# n_neighbors, которые находятся ближе, чем
# distance_coef * std(series) до интересующей точки в
# соответствии с distance_func (абсолютной разницей
# между двумя значениями), то интересующая нас точка
# не является выбросом
```

```
anomaly_dict = get_anomalies_density(
    mult_ts,
    window_size=18,
    distance_coef=1,
    n_neighbors=4)
plot_anomalies(mult_ts, anomaly_dict)
```



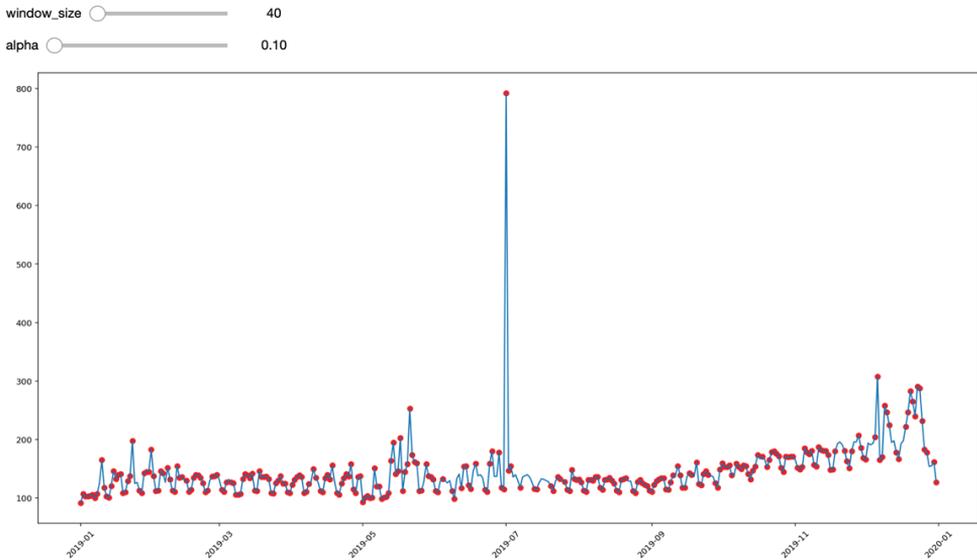
Теперь с помощью функции `get_anomalies_hist()` проанализируем выбросы на основе гистограммирования. Согласно этому способу выбросы – это все точки, удаление которых приводит к гистограмме с меньшей ошибкой аппроксимации, даже если количество бинов (`bins_number`) меньше, чем количество выбросов.

```
# проанализируем выбросы с помощью способа на основе
# гистограммирования: выбросы – это все точки, удаление
# которых приводит к гистограмме с меньшей ошибкой
# аппроксимации, даже если количество бинов меньше,
# чем количество выбросов, bins_number – количество бинов
anomaly_dict = get_anomalies_hist(mult_ts, bins_number=10)
plot_anomalies(mult_ts, anomaly_dict)
```



Для выбора наилучших значений гиперпараметров идентификаторов выбросов можно воспользоваться интерактивной визуализацией, которая реализована в функции `plot_anomalies_interactive()`. Нам нужно только указать сегмент, метод обнаружения выбросов и сетку в формате (минимум, максимум, шаг) для каждого настраиваемого гиперпараметра.

```
# задаем сегмент с сильным выбросом
segment = 'segment_c'
# задаем метод
method = get_anomalies_median
# задаем сетку гиперпараметров
params_bounds = {'window_size': (40, 70, 1),
                 'alpha': (0.1, 4, 0.25)}
# выведем интерактивную визуализацию
plot_anomalies_interactive(
    ts=mult_ts,
    segment=segment,
    method=method,
    params_bounds=params_bounds)
```

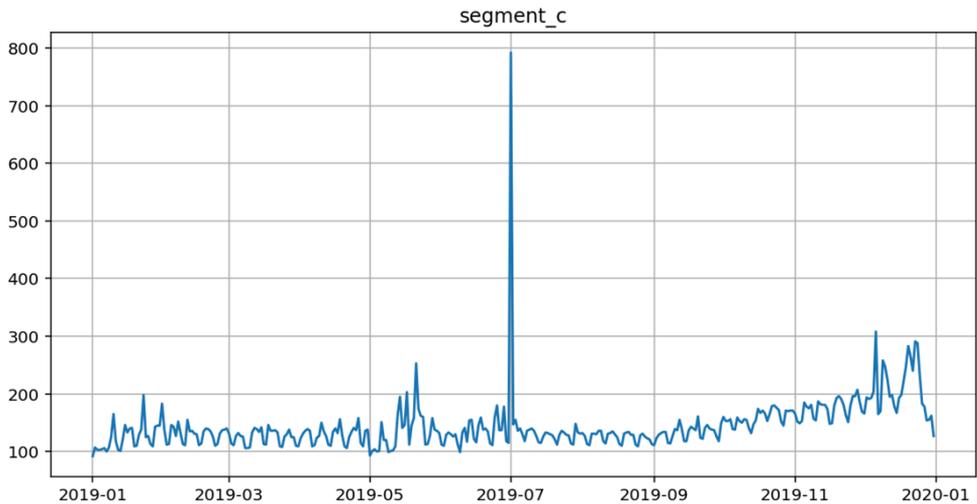


Давайте выберем наилучшие гиперпараметры и выделим сегмент с большими выбросами.

```

# выделим сегмент с большими выбросами
outlier_ts = mult_ts[:, segment, :]
outlier_ts = TSDataset(outlier_ts, freq='D')
outlier_ts.plot()

```



```

# задаем наилучшие гиперпараметры
# (здесь выбраны произвольно)
best_params = {'window_size':60,
               'alpha':2.35}

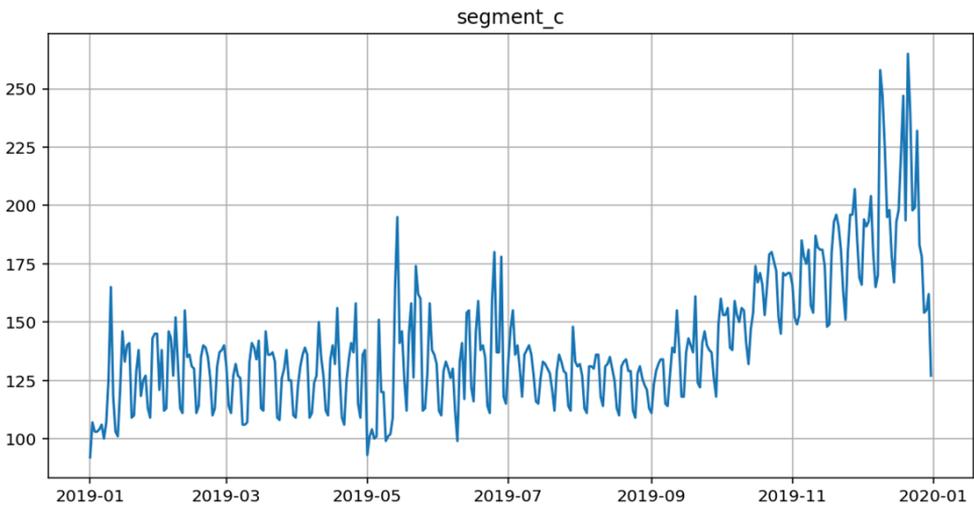
```

В библиотеке ETNA процесс обработки выбросов состоит из двух этапов:

- 1) заменяем выбросы, обнаруженные с помощью определенного метода, на значения NaN, используя экземпляр соответствующего класса `OutliersTransform`;
- 2) импутируем значения NaN с помощью класса `TimeSeriesImputerTransform`.

Давайте импутируем выбросы, обнаруженные с помощью медианного метода, используя стратегию `running_mean`.

```
# заменяем выбросы на NaN
outliers_removal = MedianOutliersTransform(
    in_column='target',
    **best_params)
# импутируем значения NaN с помощью скользящего среднего
outliers_imputer = TimeSeriesImputerTransform(
    in_column='target', strategy='running_mean', window=30)
outlier_ts.fit_transform([outliers_removal, outliers_imputer])
outlier_ts.plot()
```



Сейчас мы напишем функцию оценки качества прогнозов, функцию построения модели скользящего среднего и затем с их помощью сравним качество двух моделей – без обработки и с обработкой выбросов.

```
# пишем функцию, вычисляющую метрики качества прогнозов
def get_metrics(forecast, test):
    """
    Вычисляет метрики качества прогнозов
    """
    metrics = {'MAE':MAE(), 'MSE':MSE(), 'SMAPE':SMAPE()}
    results = dict()
    for name,metric in metrics.items():
        results[name] = metric(
            y_true=test, y_pred=forecast)['segment_c']
    return results
```

```

# пишем функцию, которая строит модель скользящего
# среднего и оценивает качество прогнозов
def test_transforms(transforms=[]):
    """
    Строит модель и оценивает качество прогнозов
    """
    classic_df = pd.read_csv('Data/example_dataset.csv')
    df = TSDataset.to_dataset(
        classic_df[classic_df['segment'] == segment])
    ts = TSDataset(df, freq='D')
    train, test = ts.train_test_split(
        train_start='2019-05-20',
        train_end='2019-07-10',
        test_start='2019-07-11',
        test_end='2019-08-09')
    model = Pipeline(model=MovingAverageModel(window=30),
                    transforms=transforms, horizon=30)
    model.fit(train)
    forecast = model.forecast()
    metrics = get_metrics(forecast, test)
    return metrics

```

Итак, сравниваем качество двух моделей – без обработки и с обработкой выбросов.

```

# строим модель скользящего среднего и оцениваем
# качество прогнозов, не применив обработку выбросов
test_transforms()

```

```

{'MAE': 40.08799715116714,
 'MSE': 1704.8554888537708,
 'SMAPE': 27.36913416466395}

```

```

# строим модель скользящего среднего и оцениваем
# качество прогнозов, применив обработку выбросов
transforms = [outliers_removal, outliers_imputer]
test_transforms(transforms)

```

```

{'MAE': 11.606826505006092,
 'MSE': 196.5736131226554,
 'SMAPE': 8.94919204071121}

```

Видим, что обработка выбросов позволила улучшить качество прогнозов.

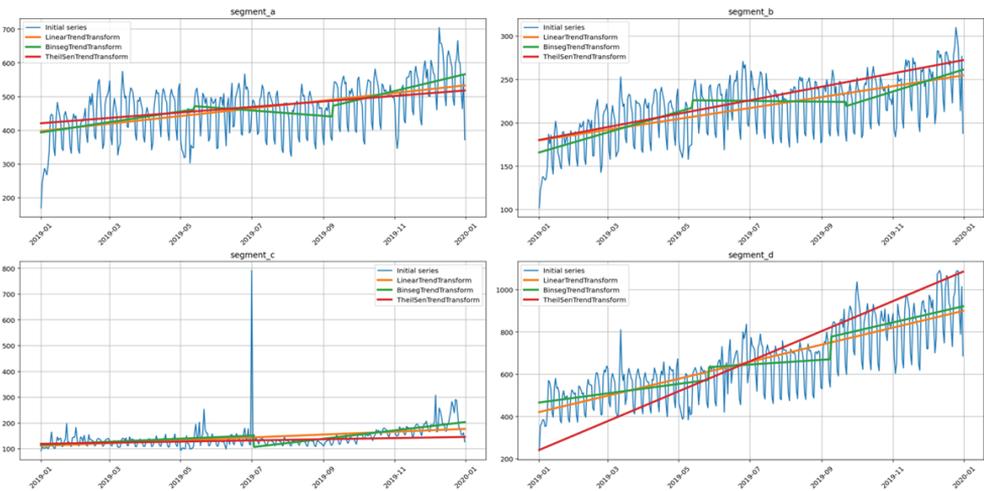
## 2.21. СОБИРАЕМ ВСЕ ВМЕСТЕ

Сейчас попробуем применить знания, полученные в предыдущих разделах, для прогнозирования потребления электроэнергии по 4 сегментам.

Проведем анализ трендов и сезонности с помощью функций `plot_trend()` и `plot_periodogram()`.

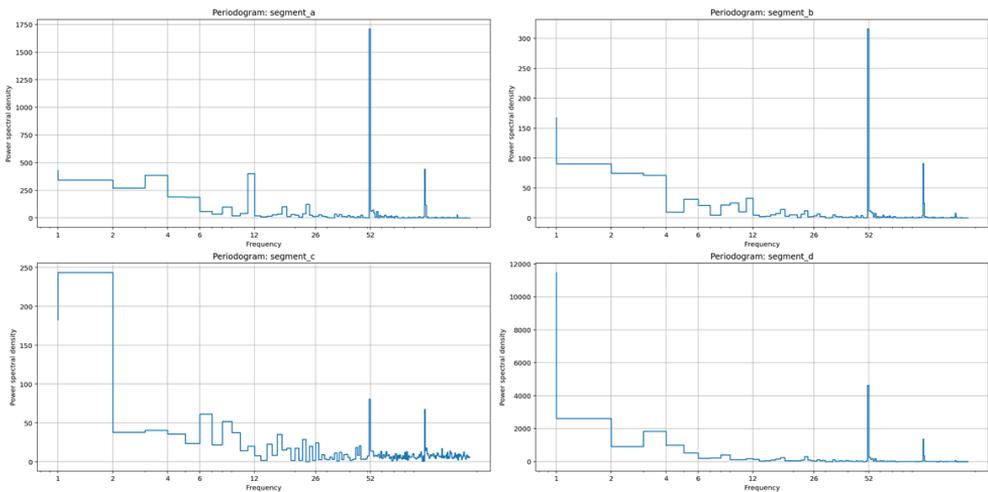
```
# вычислим и визуализируем три типа тренда - линейный,
# кусочно-линейный, линейный, полученный с
# помощью оценки Тейла-Сена
```

```
plot_trend(
    ts=mult_ts,
    trend_transform=[
        LinearTrendTransform(in_column='target'),
        BinsegTrendTransform(in_column='target', n_bkps=2,
                             min_size=100),
        TheilSenTrendTransform(in_column='target')
    ]
)
```



```
# строим периодограмму
```

```
plot_periodogram(mult_ts,
                 amplitude_aggregation_mode='per-segment',
                 period=365.24,
                 xticks=[1, 2, 4, 6, 12, 26, 52])
```



Мы применим детрендинг на основе линейного тренда (поскольку видим на графиках трендов для большинства рядов четкий восходящий тренд) и создадим компоненты ряда Фурье с периодами, равными годовой и недельной сезонности (поскольку видим на периодограммах пики в районе 1 и 52).

Сначала воспользуемся моделью CatBoost с помощью класса `CatBoostModel-MultiSegment`. Перед построением модели выполним некоторые преобразования и создадим новые признаки для наших рядов.

Нам понадобятся следующие классы-трансформеры:

- класс `MedianOutliersTransform` для замены выбросов, обнаруженных в соответствии с медианным методом, на значения `NaN`;
- класс `TimeSeriesImputerTransform` для импутации значений `NaN` в соответствии с выбранной стратегией;
- класс `LogTransform` для логарифмирования и экспоненцирования переменной (логарифмирование позволяет сгладить негативное влияние выбросов объективной природы, помогает выделить тренд);
- класс `LinearTrendTransform` для прогнозирования тренда, удаления тренда из данных и добавления тренда к прогнозам (это необходимо для деревьев решений и для ансамблей деревьев решений, не умеющих экстраполировать);
- класс `TrendTransform` для добавления тренда в качестве признака;
- класс `SegmentEncoderTransform` для кодирования меток сегментов целочисленными значениями в лексикографическом порядке (`LabelEncoding`);
- класс `LagTransform` для генерации лагов;
- класс `DateFlagsTransform` для генерации признаков на основе дат – порядковый номер дня недели, порядковый номер дня месяца, порядковый номер месяца в году;
- класс `HolidayTransform` для генерации праздников на основе дат;
- класс `MeanTransform` для вычисления скользящего среднего по заданному окну;
- класс `FourierTransform` для генерации компонент ряда Фурье.

```
# создаем экземпляр класса MedianOutliersTransform для
# замены выбросов, обнаруженных в соответствии
# с медианным методом, на значения NaN
mult_outliers_removal = MedianOutliersTransform(
    in_column='target', window_size=150, alpha=15)

# создаем экземпляр класса TimeSeriesImputerTransform для
# импутации значений NaN в соответствии с выбранной стратегией
mult_outliers_imputer = TimeSeriesImputerTransform(
    in_column='target', strategy='forward_fill')

# создаем экземпляр класса LogTransform для логарифмирования
# и экспоненцирования зависимой переменной
mult_log = LogTransform(in_column='target')

# создаем экземпляр класса LinearTrendTransform
# для прогнозирования линейного тренда, его
# удаления из данных и добавления к прогнозам
mult_linear_trend = LinearTrendTransform(in_column='target')
```

```

# создаем экземпляр класса TrendTransform для
# добавления тренда в качестве признака
mult_trend_feat = TrendTransform(in_column='target',
                                out_column='trend')

# создаем экземпляр класса SegmentEncoderTransform
# для кодирования меток сегментов целочисленными
# значениями в лексикографическом порядке (LabelEncoding):
# сегменты a, b, c, d получают значения 0, 1, 2, 3
mult_seg = SegmentEncoderTransform()

# создаем экземпляр класса LagTransform для генерации лагов,
# с помощью in_column задаем переменную, на основе которой
# генерируем лаги, мы будем генерировать лаги порядка от 7 до 210
# с шагом 7, порядок лагов не должен быть меньше длины горизонта
mult_lags = LagTransform(in_column='target',
                        lags=list(range(7, 211, 7)),
                        out_column='lag')

# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего с шириной окна 7
mult_mean7 = MeanTransform(in_column='target',
                           window=7,
                           out_column='mean7')

# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего с шириной окна 14
mult_mean14 = MeanTransform(in_column='target',
                             window=14,
                             out_column='mean14')

# создаем экземпляр класса MeanTransform для вычисления
# скользящего среднего с шириной окна 30
mult_mean30 = MeanTransform(in_column='target',
                             window=30,
                             out_column='mean30')

# создаем экземпляр класса DateFlagsTransform для генерации
# признаков на основе дат - порядковый номер дня недели,
# порядковый номер дня месяца, порядковый номер месяца в году,
# индикатор выходного дня
mult_d_flags = DateFlagsTransform(day_number_in_week=True,
                                  day_number_in_month=True,
                                  month_number_in_year=True,
                                  is_weekend=True,
                                  out_column='datetime')

# создаем экземпляр класса DateFlagsTransform для генерации
# признаков на основе дат - индикатор выходного дня
mult_d_flags2 = DateFlagsTransform(is_weekend=True,
                                   out_column='datetime2')

# создаем экземпляр класса HolidayTransform
# для генерации признаков на основе дат
mult_holidays = HolidayTransform(iso_code='RUS',
                                  out_column='RUS_holidays')

# создаем экземпляр класса FourierTransform

```

```

# для вычисления членов Фурье с периодом 365.24
# для моделирования годовой сезонности
mult_fourier_year = FourierTransform(period=365.24,
                                     order=3,
                                     out_column='fourier_year')

# создаем экземпляр класса FourierTransform
# для вычисления членов Фурье с периодом 7
# для моделирования недельной сезонности
mult_fourier_week = FourierTransform(period=7,
                                     order=2,
                                     out_column='fourier_week')

```

Полный конвейер будет выглядеть так. Сначала мы выполним импутацию выбросов. Потом применим логарифмирование зависимой переменной и вычтем из прологарифмированной переменной линейный тренд, спрогнозированный с помощью линейной регрессии. Затем добавим тренд в качестве признака. После этого выполним кодирование меток сегментов целочисленными значениями в лексикографическом порядке. Потом на основе прологарифмированной зависимой переменной с удаленным трендом мы создадим лаги и скользящие средние. Последними добавим календарные признаки, праздники и компоненты ряда Фурье с периодами 365,24 и 7.

Теперь создаем списки преобразований/признаков. Мы посмотрим, какое качество прогнозов даст полный конвейер, конвейер без детрендinга и добавления тренда в качестве признака, конвейер без лагов.

```

# создаем списки преобразований/признаков для модели CatBoost
mult_ctbst_preprocess = [mult_outliers_remove, mult_outliers_imputer,
                        mult_log, mult_linear_trend, mult_trend_feat,
                        mult_seg, mult_lags, mult_mean7, mult_mean14,
                        mult_mean30, mult_d_flags, mult_d_flags2,
                        mult_holidays, mult_fourier_year, mult_fourier_week]

# создаем списки преобразований/признаков для модели CatBoost
mult_ctbst_preprocess2 = [mult_outliers_remove, mult_outliers_imputer,
                        mult_log, mult_seg, mult_lags, mult_mean7,
                        mult_mean14, mult_mean30, mult_d_flags, mult_d_flags2,
                        mult_holidays, mult_fourier_year, mult_fourier_week]

# создаем списки преобразований/признаков для модели CatBoost
mult_ctbst_preprocess3 = [mult_outliers_remove, mult_outliers_imputer,
                        mult_log, mult_linear_trend, mult_trend_feat,
                        mult_seg, mult_mean7, mult_mean14, mult_mean30,
                        mult_d_flags, mult_d_flags2, mult_holidays,
                        mult_fourier_year, mult_fourier_week]

# создаем экземпляр класса CatBoostMultiSegmentModel
ctbst_model = CatBoostMultiSegmentModel()

# выполняем подбор гиперпараметра - набора преобразований/
# признаков с помощью перекрестной проверки модели CatBoost
etna_cv_optimize(ts=mult_ts,
                model=ctbst_model,
                horizon=mult_HORIZON,

```

```

transforms=[mult_ctbst_preprocess,
             mult_ctbst_preprocess2,
             mult_ctbst_preprocess3],
n_folds=12,
mode='expand',
metrics=SMAPE(),
refit=True)

```

trans:

```

[MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15, ), TimeSeries-
ImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1, seasonality =
1, default_value = None, constant_value = 0, ), LogTransform(in_column = 'target', base = 10,
inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree
= 1, ), TrendTransform(in_column = 'target', out_column = 'trend', detrend_model = Linear-
Regression(), model = 'ar', custom_cost = None, min_size = 2, jump = 1, n_bkps = 5, pen =
None, epsilon = None, ), SegmentEncoderTransform(), LagTransform(in_column = 'target', lags
= [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147,
154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column
= 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column
= 'mean7', ), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1,
min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTransform(in_column = 'target',
window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean30', ),
DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year
= False, week_number_in_month = False, week_number_in_year = False, month_number_in_year
= True, season_number = False, year_number = False, is_weekend = True, special_days_in_week =
(), special_days_in_month = (), out_column = 'datetime', ), DateFlagsTransform(day_number_
in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_mont
h = False, week_number_in_year = False, month_number_in_year = False, season_number = False,
year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month =
(), out_column = 'datetime2', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holi-
days', ), FourierTransform(period = 365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_col-
umn = 'fourier_year', ), FourierTransform(period = 7, order = None, mods = [1, 2, 3, 4],
out_column = 'fourier_week', )]
SMAPE_mean: 6.6259029846724005
SMAPE_std: 4.731481065519761

```

trans:

```

[MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15, ), TimeSeries-
ImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1, seasonality =
1, default_value = None, constant_value = 0, ), LogTransform(in_column = 'target', base = 10,
inplace = True, out_column = None, ), SegmentEncoderTransform(), LagTransform(in_col-
umn = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112,
119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag',
), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods
= 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14,
seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTran-
sform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna
= 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_in_
month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year
= False, month_number_in_year = True, season_number = False, year_number = False, is_weekend
= True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ),
DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year
= False, week_number_in_month = False, week_number_in_year = False, month_number_in_year
= False, season_number = False, year_number = False, is_weekend = True, special_days_in_week
= (), special_days_in_month = (), out_column = 'datetime2', ), HolidayTransform(iso_code =

```

```
'RUS', out_column = 'RUS_holidays', ), FourierTransform(period = 365.24, order = None, mods
= [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', ), FourierTransform(period = 7, order =
None, mods = [1, 2, 3, 4], out_column = 'fourier_week', )]
SMAPE_mean: 7.2259229652840515
SMAPE_std: 4.743618976290882
```

trans:

```
[MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15, ), TimeSeries-
ImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1, seasonality =
1, default_value = None, constant_value = 0, ), LogTransform(in_column = 'target', base = 10,
inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree
= 1, ), TrendTransform(in_column = 'target', out_column = 'trend', detrend_model = Linea
rRegression(), model = 'ar', custom_cost = None, min_size = 2, jump = 1, n_bkps = 5, pen
= None, epsilon = None, ), SegmentEncoderTransform(), MeanTransform(in_column = 'target',
window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7',
), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1, min_periods
= 1, fillna = 0, out_column = 'mean14', ), MeanTransform(in_column = 'target', window = 30,
seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean30', ), DateFl
agsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year =
False, week_number_in_month = False, week_number_in_year = False, month_number_in_year =
True, season_number = False, year_number = False, is_weekend = True, special_days_in_week =
(), special_days_in_month = (), out_column = 'datetime', ), DateFlagsTransform(day_number_
in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_month
= False, week_number_in_year = False, month_number_in_year = False, season_number = False,
year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month =
(), out_column = 'datetime2', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holi-
days', ), FourierTransform(period = 365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_col-
umn = 'fourier_year', ), FourierTransform(period = 7, order = None, mods = [1, 2, 3, 4],
out_column = 'fourier_week', )]
SMAPE_mean: 7.285592468074067
SMAPE_std: 4.166917379086768
```

Наилучший набор преобразований/признаков:

```
{'trans': [MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15,
), TimeSeriesImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1,
seasonality = 1, default_value = None, constant_value = 0, ), LogTransform(in_column =
'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column
= 'target', poly_degree = 1, ), TrendTransform(in_column = 'target', out_column = 'trend',
detrend_model = LinearRegression(), model = 'ar', custom_cost = None, min_size = 2, jump =
1, n_bkps = 5, pen = None, epsilon = None, ), SegmentEncoderTransform(), LagTransform(in_col-
umn = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112,
119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag',
), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods
= 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14,
seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTran-
sform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna
= 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_in_
month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year
= False, month_number_in_year = True, season_number = False, year_number = False, is_weekend
= True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ),
DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year
= False, week_number_in_month = False, week_number_in_year = False, month_number_in_year =
False, season_number = False, year_number = False, is_weekend = True, special_days_in_week
= (), special_days_in_month = (), out_column = 'datetime2', ), HolidayTransform(iso_code =
'RUS', out_column = 'RUS_holidays', ), FourierTransform(period = 365.24, order = None, mods
= [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', ), FourierTransform(period = 7, order =
None, mods = [1, 2, 3, 4], out_column = 'fourier_week', )]}
```



```
# выполняем подбор гиперпараметра - набора преобразований/
# признаков с помощью перекрестной проверки модели ElasticNet
etna_cv_optimize(ts=mult_ts,
                 model=elastnet_model,
                 horizon=mult_HORIZON,
                 transfrms=[mult_elastnet_preprocess,
                             mult_elastnet_preprocess2],
                 n_folds=12,
                 mode='expand',
                 metrics=SMAPE(),
                 refit=True)
```

trans:

```
[MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15, ), TimeSeries-
ImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1, seasonality =
1, default_value = None, constant_value = 0, ), LogTransform(in_column = 'target', base = 10,
inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree
= 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77,
84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210],
out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha
= 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target',
window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ),
MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1,
fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_
in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year
= False, month_number_in_year = True, season_number = False, year_number = False, is_weekend =
True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), Holi-
dayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), MinMaxScalerTransform(in_column
= None, inplace = True, out_column = None, feature_range = (0, 1), clip = True, mode = 'per-seg-
ment', ), FourierTransform(period = 365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_column
= 'fourier_year', ), FourierTransform(period = 7, order = None, mods = [1, 2, 3, 4], out_col-
umn = 'fourier_week', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month =
True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False,
month_number_in_year = False, season_number = False, year_number = False, is_weekend = True,
special_days_in_week = (), special_days_in_month = (), out_column = 'datetime2', )]
SMAPE_mean: 8.916830169651051
SMAPE_std: 4.659897872383401
```

trans:

```
[MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15, ), TimeSeries-
ImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1, seasonality =
1, default_value = None, constant_value = 0, ), LogTransform(in_column = 'target', base = 10,
inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree
= 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70,
77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203,
210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality =
1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column
= 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column
= 'mean14', ), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1,
min_periods = 1, fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week =
True, day_number_in_month = True, day_number_in_year = False, week_number_in_month = False,
week_number_in_year = False, month_number_in_year = True, season_number = False, year_number
= False, is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column
= 'datetime', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), Stan-
dardScalerTransform(in_column = None, inplace = True, out_column = None, with_mean = True,
with_std = True, mode = 'per-segment', ), FourierTransform(period = 365.24, order = None,
```

```

mods = [1, 2, 3, 4, 5, 6], out_column = 'fourier_year', ), FourierTransform(period = 7, order
= None, mods = [1, 2, 3, 4], out_column = 'fourier_week', ), DateFlagsTransform(day_number_
in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_month
= False, week_number_in_year = False, month_number_in_year = False, season_number = False,
year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month =
(), out_column = 'datetime2', )]
SMAPE_mean: 10.454824684897938
SMAPE_std: 8.610477742077938

```

Наилучший набор преобразований/признаков:

```

{'trans': [MedianOutliersTransform(in_column = 'target', window_size = 150, alpha = 15, ),
TimeSeriesImputerTransform(in_column = 'target', strategy = 'forward_fill', window = -1, season-
ality = 1, default_value = None, constant_value = 0, ), LogTransform(in_column = 'target', base =
10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree
= 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77,
84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210],
out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha
= 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target',
window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ),
MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1,
fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_
in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year
= False, month_number_in_year = True, season_number = False, year_number = False, is_weekend =
True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), Holiday
Transform(iso_code = 'RUS', out_column = 'RUS_holidays', ), MinMaxScalerTransform(in_column =
None, inplace = True, out_column = None, feature_range = (0, 1), clip = True, mode = 'per-seg-
ment', ), FourierTransform(period = 365.24, order = None, mods = [1, 2, 3, 4, 5, 6], out_column
= 'fourier_year', ), FourierTransform(period = 7, order = None, mods = [1, 2, 3, 4], out_col-
umn = 'fourier_week', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month =
True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False,
month_number_in_year = False, season_number = False, year_number = False, is_weekend = True,
special_days_in_week = (), special_days_in_month = (), out_column = 'datetime2', )]}

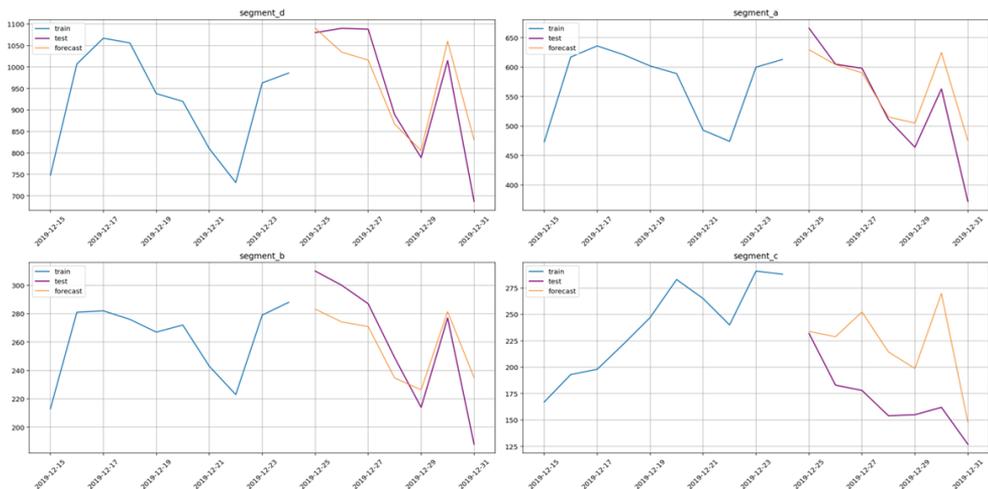
```

Лучшее значение SMAPE cv: 8.9168

```

{'segment_d': 5.814919385661757, 'segment_a': 7.288353991674514, 'segment_b':
8.442390485980827, 'segment_c': 25.749164122474948}

```



Дополнительно можно написать функцию, выполняющую подбор гиперпараметра – набора преобразований/признаков с помощью перекрестной проверки, при этом она делает пошаговый проход по конвейеру. Таким образом, мы можем определить, как включение того или иного этапа в конвейере повлияло на метрику.

```
# пишем функцию, выполняющую подбор гиперпараметра – набора
# преобразований/признаков с помощью перекрестной проверки
# (для пошагового прохода по конвейеру)
def etna_staged_cv_optimize(ts, model, horizon, init_transfrms,
                             transfrms, n_folds, mode, metrics,
                             refit=True, n_train_samples=10):
    """
    Выполняет подбор гиперпараметра – набора преобразований/
    признаков с помощью перекрестной проверки (пошаговый
    проход конвейера)

    Параметры
    -----
    ts: pandas.DataFrame
        Мультииндексный pandas.DataFrame в формате ETNA
        (объект TSDataset), содержащий один или
        несколько временных рядов.
    model: instance of class etna.models
        Модель прогнозирования.
    horizon: int
        Горизонт прогнозирования.
    init_transfrms: list
        Список исходных преобразований.
    transfrms: list
        Список преобразований/признаков.
    n_folds, int
        Количество тестовых выборок перекрестной проверки.
    mode: str
        Тип перекрестной проверки.
    metrics: instance of class etna.metrics
        Метрика качества.
    refit: bool
        Нужно ли строить наилучшую модель на всей обучающей выборке.
    n_train_sample: int
        n последних наблюдений обучающей выборки
        на графике прогнозов.
    """
    # разбиваем набор на обучающую и тестовую выборки
    # с учетом временной структуры, размер тестовой
    # выборки задаем равным горизонту
    train_ts, test_ts = ts.train_test_split(test_size=horizon)

    # инициализируем наилучшее значение метрики
    # положительной бесконечностью
    best_score = np.inf

    # задаем пустые списки
    el_lst = list()
    transfrms_lst = list()
```

```

# получаем список списков
for el in transforms:
    el_lst.append(el)
    transforms_lst.append(el_lst.copy())

# добавляем в начало каждого списка список
# со стартовыми преобразованиями
for i in range(len(transforms_lst)):
    transforms_lst[i] = init_transforms + transforms_lst[i]

# с помощью цикла for
for trans in transforms_lst:
    # создаем конвейер с моделью и списком преобразований
    pipe = Pipeline(
        model=model,
        transforms=trans,
        horizon=horizon)

    # находим метрики моделей по сегменту/сегментам
    # по итогам перекрестной проверки
    df_metrics, _, _ = pipe.backtest(mode=mode,
                                    n_folds=n_folds,
                                    ts=train_ts,
                                    metrics=[metrics],
                                    aggregate_metrics=False,
                                    joblib_params=dict(verbose=0))

    # вычисляем значение метрики, усредненное по тестовым выборкам
    metrics_mean = df_metrics[metrics.__class__.__name__].mean()
    # вычисляем стандартное отклонение метрики
    metrics_std = df_metrics[metrics.__class__.__name__].std()
    print(f"trans:\n{trans}")
    print(f"{metrics.__class__.__name__}_mean: {metrics_mean}")
    print(f"{metrics.__class__.__name__}_std: {metrics_std}\n")

    # если получаем максимальное усредненное значение, сохраняем
    # его и наилучший набор преобразований/признаков
    if metrics_mean < best_score:
        best_score = metrics_mean
        best_parameters = {'trans': trans}

# печатаем наилучший набор преобразований/признаков
# и наилучшее значение метрики по итогам
# перекрестной проверки
print(f"Наилучший набор преобразований/признаков:\n{best_parameters}\n")
print(f"Лучшее значение {metrics.__class__.__name__} cv: {best_score:.4f}\n")

if refit:
    # создаем конвейер с наилучшим набором преобразований/признаков
    pipe = Pipeline(model=model,
                    transforms=best_parameters.get('trans'),
                    horizon=horizon)

    # обучаем конвейер на всей обучающей выборке
    pipe.fit(train_ts)
    # получаем прогнозы
    forecast_ts = pipe.forecast()
    # оцениваем качество прогнозов
    print(metrics(y_true=test_ts, y_pred=forecast_ts))

```

```

# визуализируем прогнозы
plot_forecast(forecast_ts, test_ts,
              train_ts, n_train_samples=n_train_samples)
# выполняем подбор гиперпараметра - набора преобразований/
# признаков с помощью перекрестной проверки модели ElasticNet
# (пошаговый проход по конвейеру)
etna_staged_cv_optimize(ts=mult_ts,
                        model=elastnet_model,
                        horizon=mult_HORIZON,
                        init_transfrms=[mult_log, mult_linear_trend],
                        transfrms=[mult_lags, mult_mean7, mult_mean14,
                                   mult_mean30, mult_d_flags, mult_holidays,
                                   minmaxscaler],
                        n_folds=12,
                        mode='expand',
                        metrics=SMAPE(),
                        refit=True)

trans:
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', )]
SMAPE_mean: 7.591626468811179
SMAPE_std: 4.017484274515624

trans:
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', )]
SMAPE_mean: 7.591626475699847
SMAPE_std: 4.017484273089682

trans:
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', )]
SMAPE_mean: 7.591626475699847
SMAPE_std: 4.017484273089682

trans:
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean30', )]

```

SMAPE\_mean: 7.591626475699847

SMAPE\_std: 4.017484273089682

trans:

```
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False, month_number_in_year = True, season_number = False, year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', )]
```

SMAPE\_mean: 7.55810889228147

SMAPE\_std: 3.944214915284621

trans:

```
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False, month_number_in_year = True, season_number = False, year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', )]
```

SMAPE\_mean: 7.4306763980479325

SMAPE\_std: 3.948210492500154

trans:

```
[LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None, ), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_column = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112, 119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag', ), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanTransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_number_in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_in_year = False, month_number_in_year = True, season_number = False, year_number = False, is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column = 'datetime', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), MinMaxScalerTransform(in_column = ['RUS_holidays', 'datetime_day_number_in_month', 'datetime_day_number_in_week', 'datetime_is_weekend', 'datetime_month_number_in_year', 'lag_105', 'lag_112', 'lag_119', 'lag_126', 'lag_133', 'lag_14', 'lag_140', 'lag_147', 'lag_154', 'lag_161', 'lag_168', 'lag_175', 'lag_182', 'lag_189', 'lag_196', 'lag_203', 'lag_21', 'lag_210', 'lag_28', 'lag_35', 'lag_42', 'lag_49', 'lag_56', 'lag_63', 'lag_7', 'lag_70', 'lag_77',
```

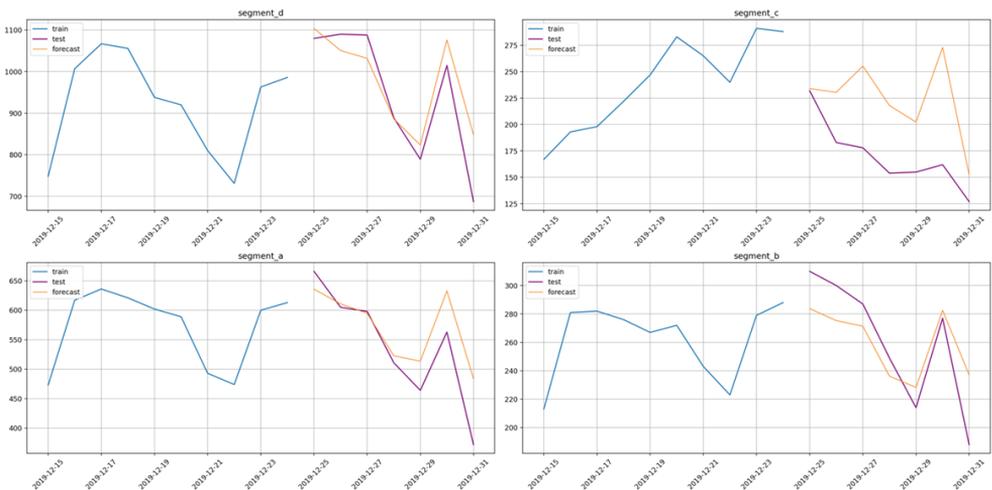
```
'lag_84', 'lag_91', 'lag_98', 'mean14', 'mean30', 'mean7', 'target'], inplace = True, out_
column = None, feature_range = (0, 1), clip = True, mode = 'per-segment', )]
SMAPE_mean: 7.283446331203172
SMAPE_std: 3.847961666710825
```

Наилучший набор преобразований/признаков:

```
{'trans': [LogTransform(in_column = 'target', base = 10, inplace = True, out_column = None,
), LinearTrendTransform(in_column = 'target', poly_degree = 1, ), LagTransform(in_col-
umn = 'target', lags = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98, 105, 112,
119, 126, 133, 140, 147, 154, 161, 168, 175, 182, 189, 196, 203, 210], out_column = 'lag',
), MeanTransform(in_column = 'target', window = 7, seasonality = 1, alpha = 1, min_peri-
ods = 1, fillna = 0, out_column = 'mean7', ), MeanTransform(in_column = 'target', window = 14,
seasonality = 1, alpha = 1, min_periods = 1, fillna = 0, out_column = 'mean14', ), MeanT-
ransform(in_column = 'target', window = 30, seasonality = 1, alpha = 1, min_periods = 1,
fillna = 0, out_column = 'mean30', ), DateFlagsTransform(day_number_in_week = True, day_num-
ber_in_month = True, day_number_in_year = False, week_number_in_month = False, week_number_
in_year = False, month_number_in_year = True, season_number = False, year_number = False,
is_weekend = True, special_days_in_week = (), special_days_in_month = (), out_column = 'da-
atetime', ), HolidayTransform(iso_code = 'RUS', out_column = 'RUS_holidays', ), MinMaxScaler-
Transform(in_column = ['RUS_holidays', 'datetime_day_number_in_month', 'datetime_day_num-
ber_in_week', 'datetime_is_weekend', 'datetime_month_number_in_year', 'lag_105', 'lag_112',
'lag_119', 'lag_126', 'lag_133', 'lag_14', 'lag_140', 'lag_147', 'lag_154', 'lag_161',
'lag_168', 'lag_175', 'lag_182', 'lag_189', 'lag_196', 'lag_203', 'lag_21', 'lag_210',
'lag_28', 'lag_35', 'lag_42', 'lag_49', 'lag_56', 'lag_63', 'lag_7', 'lag_70', 'lag_77',
'lag_84', 'lag_91', 'lag_98', 'mean14', 'mean30', 'mean7', 'target'], inplace = True, out_
column = None, feature_range = (0, 1), clip = True, mode = 'per-segment', )]}
```

Лучшее значение SMAPE cv: 7.2834

```
{'segment_d': 6.0767690873995885, 'segment_c': 27.09978850322134, 'segment_a':
8.046000789920978, 'segment_b': 8.548360948383472}
```



## 2.22. МОДЕЛИ НЕЙРОННЫХ СЕТЕЙ

Рассмотрим наиболее важные параметры класса `PytorchForecastingTransform`:

- `time_varying_known_reals` – известные вещественные значения, изменяющиеся во времени (вещественные регрессоры), теперь необходимо добавить в список переменную `time_idx`;
- `time_varying_unknown_reals` – вещественные значения зависимой переменной, задаем его равным `['target']`;
- `max_prediction_length` – наш горизонт прогнозирования;
- `max_encoder_length` – максимальная длина истории, исходя из которой модель будет строить признаки;
- `static_categoricals` – статические категориальные значения, например если мы используем несколько сегментов, это могут быть некоторые его характеристики, включая идентификатор `'segment'`;
- `time_varying_known_categoricals` – известные категориальные значения, которые изменяются во времени (категориальные регрессоры);
- `target_normalizer` – класс для нормализации зависимых переменных в разных сегментах.

Давайте построим DeepAR-модель для прогнозирования потребления электроэнергии по 4 сегментам.

Сначала настроим воспроизводимость результатов.

```
# настраиваем воспроизводимость результатов
torch.manual_seed(42)
random.seed(42)
np.random.seed(42)
```

В качестве признаков возьмем порядковый номер дня недели, лаги и компоненты ряда Фурье.

```
# используем календарный признак
transform_date = DateFlagsTransform(day_number_in_week=True,
                                   out_column='dateflag')
dateflag = 'dateflag_day_number_in_week'

# используем лаги
num_lags = 10
transform_lag = LagTransform(
    in_column='target',
    lags=[mult_HORIZON + i for i in range(num_lags)],
    out_column='target_lag',
)
lag_columns = [f"target_lag_{mult_HORIZON+i}" for i in range(num_lags)]
lag_columns

['target_lag_7',
'target_lag_8',
'target_lag_9',
'target_lag_10',
'target_lag_11',
'target_lag_12',
'target_lag_13',
```

```
'target_lag_14',
'target_lag_15',
'target_lag_16']
```

*# используем компоненты ряда Фурье*

```
num_fourier = 6
fourier_year = FourierTransform(period=365.24,
                                order=3,
                                out_column='fourier_year')
fourier_columns = [f"fourier_year_{i+1}" for i in range(num_fourier)]
fourier_columns
```

```
['fourier_year_1',
 'fourier_year_2',
 'fourier_year_3',
 'fourier_year_4',
 'fourier_year_5',
 'fourier_year_6']
```

Теперь создаем экземпляр класса `PytorchForecastingTransform`, задав параметры.

*# создаем экземпляр класса PytorchForecastingTransform*

```
transform_deepar = PytorchForecastingTransform(
    max_encoder_length=mult_HORIZON,
    max_prediction_length=mult_HORIZON,
    time_varying_known_reals=['time_idx'] + lag_columns + fourier_columns,
    time_varying_unknown_reals=['target'],
    time_varying_known_categoricals=[dateflag],
    target_normalizer=GroupNormalizer(groups=['segment']),
)
```

Теперь создаем экземпляр класса `DeepARModel`, список метрик, конвейер и выполняем перекрестную проверку модели.

*# создаем экземпляр класса DeepARModel*

```
model_deepar = DeepARModel(max_epochs=150,
                           learning_rate=[0.01],
                           gpus=0,
                           batch_size=64)
```

*# список метрик*

```
metrics = [SMAPE(), MAPE(), MAE()]
```

*# создаем конвейер*

```
pipeline_deepar = Pipeline(
    model=model_deepar,
    horizon=mult_HORIZON,
    transforms=[transform_lag, fourier_year,
                transform_date, transform_deepar],
)
```

*# выполняем перекрестную проверку модели*

```
metrics_deepar, forecast_deepar, fold_info_deepar = pipeline_deepar.backtest(
    mult_ts, metrics=metrics, aggregate_metrics=True, n_folds=3, n_jobs=1)
```

Давайте посмотрим метрику качества по сегментам и визуализируем прогнозы.

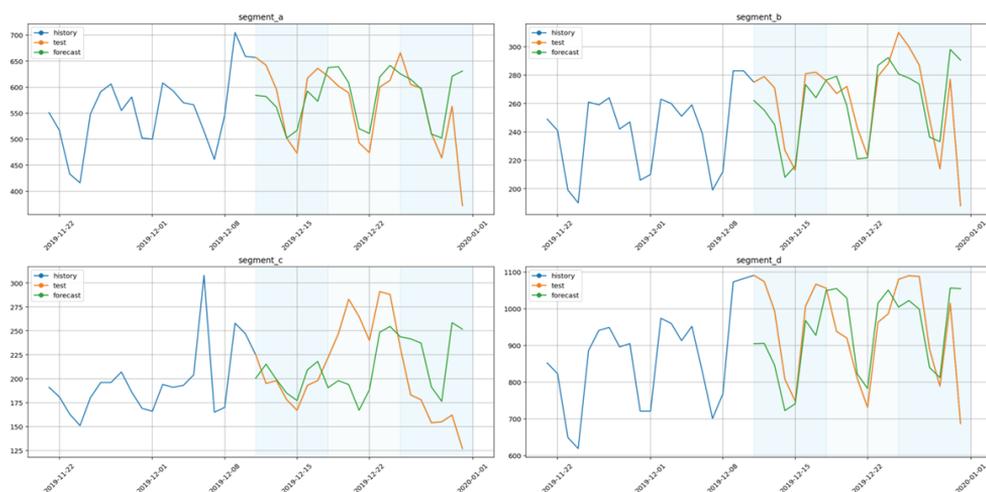
```
# смотрим метрики по сегментам
```

```
metrics_deepar
```

	segment	SMAPE	MAPE	MAE
0	segment_a	7.671311	8.529388	42.503334
1	segment_b	7.313505	7.760041	18.667973
2	segment_c	20.423617	22.224012	43.037298
3	segment_d	9.636106	9.956957	90.522856

```
# визуализируем прогнозы
```

```
plot_backtest(forecast_deepar, mult_ts, history_len=20)
```



## 2.23. ОПТИМИЗАЦИЯ ГИПЕРПАРАМЕТРОВ С ПОМОЩЬЮ ОРТУНА ОТ РАЗРАБОТЧИКОВ

В этом разделе рассмотрим оптимизацию гиперпараметров с помощью Optuna от разработчиков библиотеки ETNA.

Все необходимое находится в скрипте `optuna_example.py` и тетрадке *Часть 3\_2.23. Оптимизация гиперпараметров в ETNA от разработчиков.ipynb*.

Скрипт начинается с импорта необходимых библиотек, классов и функций. Обратите внимание: мы будем использовать библиотеку для создания CLI-приложений typer и MLOPs-платформу Weights & Biases для визуализации результатов оптимизации (необходимо завести аккаунт на <https://wandb.ai> и получить API-ключ).

```
# импортируем необходимые библиотеки, классы и функции
```

```
import random
from functools import partial
```

```

from pathlib import Path
from typing import Optional

import numpy as np
import optuna
import pandas as pd
import typer

from etna.datasets import TSDataset
from etna.loggers import WandbLogger
from etna.loggers import tslogger
from etna.metrics import MAE
from etna.metrics import MSE
from etna.metrics import SMAPE
from etna.metrics import Sign
from etna.models import CatBoostModelMultiSegment
from etna.pipeline import Pipeline
from etna.transforms import LagTransform
from etna.transforms import SegmentEncoderTransform
from etna.transforms import StandardScalerTransform

```

Теперь задаем переменную пути к файлу данных.

```

# задаем переменную пути к файлу данных
FILE_PATH = Path(__file__)

```

Теперь пишем функцию, задающую стартовое значение генератора псевдослучайных чисел для воспроизводимости.

```

# пишем функцию, задающую стартовое значение генератора
# псевдослучайных чисел для воспроизводимости
def set_seed(seed: int = 42):
    random.seed(seed)
    np.random.seed(seed)

```

Пишем функцию инициализации логгера. В ее основе – класс WandbLogger библиотеки PyTorch Lightning, которая предлагает обучение моделей PyTorch с помощью высокоуровневого интерфейса и их логирование с помощью Weights & Biases.

```

# пишем функцию инициализации логгера
def init_logger(config: dict, project: str = 'wandb-sweeps',
               tags: Optional[list] = ['test', 'sweeps']):
    tslogger.loggers = []
    wblogger = WandbLogger(project=project,
                          tags=tags,
                          config=config)
    tslogger.add(wblogger)

```

Пишем функцию, загружающую данные.

```

# пишем функцию, загружающую данные
def data_loader(file_path: Path, freq: str = 'D') -> TSDataset:
    df = pd.read_csv(file_path)

```

```
df = TSDataset.to_dataset(df)
ts = TSDataset(df=df, freq=freq)
return ts
```

Теперь пишем целевую функцию Optuna. В ее основе лежит перекрестная проверка конвейера преобразований / операций конструирования признаков расширяющимся или скользящим окном. Конкретно в данном случае мы выполняем стандартизацию зависимой переменной, кодирование сегментов целочисленными метками и генерирование лагов (с порядком, равным горизонту прогнозирования, и выше).

```
# пишем целевую функцию Optuna
def objective(trial: optuna.Trial,
             metric_name: str,
             ts: TSDataset,
             horizon: int,
             lags: int,
             seed: int):
    """
    Целевая функция Optuna.
    """

    # задаем стартовое значение генератора псевдослучайных
    # чисел для воспроизводимости
    set_seed(seed)

    # задаем модель и признаки
    pipeline = Pipeline(
        model=CatBoostModelMultiSegment(
            iterations=trial.suggest_int('iterations', 10, 100),
            depth=trial.suggest_int('depth', 1, 12),
        ),
        transforms=[
            StandardScalerTransform('target'),
            SegmentEncoderTransform(),
            LagTransform(
                in_column='target',
                lags=list(range(horizon,
                               horizon +
                               trial.suggest_int('lags', 1, lags)))),
        ],
        horizon=horizon,
    )

    # инициализируем логгер WandB
    init_logger(pipeline.to_dict())

    # запускаем перекрестную проверку
    metrics, _, _ = pipeline.backtest(
        ts=ts, metrics=[MAE(), SMAPE(), Sign(), MSE()])
    return metrics[metric_name].mean()
```

Затем создаем приложение турег. Турег и с помощью декоратора сообщаем Турег, что нижеприведенная функция запуска оптимизации будет «командой».

```

# используем декоратор, он сообщает Typer, что
# нижеприведенная функция является «командой»
@app.command()
# пишем функцию запуска оптимизации
def run_optuna(
    horizon: int = 14,
    metric_name: str = 'MAE',
    storage: str = 'sqlite:///optuna.db',
    study_name: Optional[str] = None,
    n_trials: int = 200,
    file_path: Path = 'Data/example_dataset.csv',
    direction: str = 'minimize',
    freq: str = 'D',
    lags: int = 24,
    seed: int = 11,
):
    """
    Запускает оптимизацию Optuna
    для CatBoostModelMultiSegment.
    """
    # загружаем данные
    ts = dataloader(file_path, freq=freq)

    # создаем сессию оптимизации Optuna
    study = optuna.create_study(
        storage=storage,
        study_name=study_name,
        sampler=optuna.samplers.TPESampler(multivariate=True,
                                           group=True),
        load_if_exists=True,
        direction=direction,
    )

    # запускаем оптимизацию Optuna
    study.optimize(
        partial(objective, metric_name=metric_name, ts=ts,
                horizon=horizon, lags=lags, seed=seed),
        n_trials=n_trials
    )

```

С помощью `typer.run()` мы будем запускать оптимизацию.

```

# запускаем
if __name__ == '__main__':
    typer.run(run_optuna)

```

Открываем терминал, переходим в папку, где лежит скрипт, и запускаем, задав 5 итераций поиска и метрику MAE для оптимизации.

```

(base) MacBook-Pro-Artem:~ artemgruzdev$ cd Documents/TimeSeries/Code
(base) MacBook-Pro-Artem:Code artemgruzdev$ python optuna_example.py --n-trials=5 --metric-name=MAE

```

Может потребоваться выбор варианта оптимизации – с визуализацией или без нее:



# Ответы на вопросы с собеседований

## Задача с собеседования (математика)

### Вопрос 1

Идут три охотника на охоту. У одного – 3 стакана крупы, у другого – 5 стаканов крупы, у третьего – 8 патронов. Сварили кашу, и все поели поровну. Третий охотник решил отблагодарить двух и отдать им все патроны. Как поделить патроны справедливо?

### Ответ на вопрос 1

$3 + 5 = 8$  стаканов пошло на кашу.

8: 3 = 2,66 стакана – кашу из такого количества крупы съел каждый охотник.

$3 - 2,66 = 0,33$  стакана – такой излишек остался у первого для третьего.

$5 - 2,66 = 2,33$  стакана – такой излишек остался у второго для третьего.

Вклады первого и второго охотника в порцию третьего относятся как 1 к 7. 8 патронов составила плата за кашу. 1 патрон нужно отдать первому охотнику, а 7 патронов – второму.

## Задача с собеседования (SQL)

### Вопрос 1

Создайте с помощью языка SQL таблицу employees.

id	name	age	department
1	David	22	B
2	Paul	33	B
3	Jeremy	26	B
4	Jack	21	C
5	John	36	A
6	David	45	A

### Ответ на вопрос 1

```
CREATE TABLE employees (id INTEGER, name TEXT, age INTEGER, department TEXT);
INSERT INTO employees (id, name, age, department)
VALUES (1, 'David', 22, 'B');
```

```

INSERT INTO employees (id, name, age, department)
VALUES (2, 'Paul', 33, 'B');
INSERT INTO employees (id, name, age, department)
VALUES (3, 'Jeremy', 26, 'B');
INSERT INTO employees (id, name, age, department)
VALUES (4, 'Jack', 21, 'C');
INSERT INTO employees (id, name, age, department)
VALUES (5, 'John', 36, 'A');
INSERT INTO employees (id, name, age, department)
VALUES (6, 'David', 45, 'A');
    
```

Таблица:

	id	name	age	department
	Фильтр	Фильтр	Фильтр	Фильтр
1	1	David	22	B
2	2	Paul	33	B
3	3	Jeremy	26	B
4	4	Jack	21	C
5	5	John	36	A
6	6	David	45	A

## Задача с собеседования (математика)

### Вопрос 1

Является ли число 3599 простым?

### Ответ на вопрос 1

Простое число – натуральное (целое положительное) число, имеющее ровно два различных натуральных делителя – единицу и самого себя. Другими словами, число  $x$  является простым, если оно больше 1 и при этом делится без остатка только на 1 и на  $x$ . К примеру, 5 – это простое число, а 6 не является простым числом, так как, помимо 1 и 6, также делится на 2 и на 3.

$3599 = 3600 - 1 = 60^2 - 1^2 = (60 - 1) * (60 + 1) = 59 * 61$ , число не является простым.

## Задача с собеседования (математика)

### Вопрос 1

Что больше,  $\log_2(3)$  или  $\log_3(5)$ ?

Это одна из известных «еврейских задачеч», которые любят задавать на собеседованиях в американских банках (Citibank, Capital One, Goldman Sachs). Речь идет о задачах, которые давали евреям и прочим попадающим под национальный ценз на устном экзамене по математике на мехмате МГУ (в 1970-х и 1980-х). Как правило, это были задачи с решением, которое выглядит просто, но

которое очень сложно найти. Список подобных задачеч с идеями для решений и собственно решениями вы найдете здесь: <https://arxiv.org/pdf/1110.1556.pdf>.

## Ответ на вопрос 1

Сравним оба числа с  $3/2$ .

$\log_2(3) > 3/2$ , потому что  $2^{\log_2(3)} > 2^{(3/2)}$ , потому что  $3 > \sqrt{2^3}$ , потому что  $9 > 8$ .

$\log_3(5) < 3/2$ , потому что  $3^{\log_3(5)} < 3^{(3/2)}$ , потому что  $5 < \sqrt{3^3}$ , потому что  $25 < 27$ .

Таким образом,  $\log_2(3) > 3/2 > \log_3(5)$ , поэтому  $\log_2(3) > \log_3(5)$ .

## Задачи с собеседований (Python)

### Вопрос 1

Расскажите об операторах принадлежности `in` и `not in`.

### Ответ на вопрос 1

Они используются для проверки того, найдено ли значение или переменная в последовательности (строка, список, кортеж, множество и словарь).

```
import numpy as np
a = np.array([1, 2, 3])
```

```
1 in a
```

```
True
```

```
5 not in a
```

```
True
```

### Вопрос 2

Расскажите про операторы тождественности `is` и `is not`.

### Ответ на вопрос 2

Они используются для проверки того, занимают ли два значения (или переменные) одну и ту же позицию памяти. Две равные переменные не означают, что они идентичны.

```
True is False
```

```
False
```

```
True is not False
```

```
True
```

### Вопрос 3

Дано `a = [1, 2, 3]` и `b = [1, 2, 3]`,

- какое значение получим, применив оператор `a == b`?
- какое значение получим, применив оператор `a is b`?

### Ответ на вопрос 3

В первом случае получим `True`, во втором случае – `False`. Списки равны, но не идентичны. Это потому, что интерпретатор размещает их в отдельных участках памяти, хотя они равны.

```
a = [1, 2, 3]
b = [1, 2, 3]
```

```
a == b
```

```
True
```

```
a is b
```

```
False
```

С помощью функции `id()` убедимся, что у `a` и `b` – разные идентификаторы.

```
print(id(a))
print(id(b))
```

```
140471747331776
140471206545792
```

### Вопрос 4

Какое значение получим, применив оператор `a is b`:

- если `a = 123` и `b = 123`;
- если `a = 'hello'` и `b = 'hello'`.

Необходимо пояснить причину получения данных значений.

### Ответ на вопрос 4

В обоих случаях получим значение `True`. Убедимся в этом.

```
a = 123
b = 123
a is b
```

```
True
```

```
a = 'hello'
b = 'hello'
a is b
```

```
True
```

Этот эффект обусловлен тем, что Python в целях производительности кеширует короткие строки и малые целые числа (от  $-5$  до  $256$ ). Если взять числа побольше, числа с плавающей точкой, строки подлиннее, то этот эффект пропадает. Убедимся в этом.

```
a = 'hello world'
b = 'hello world'
a is b
```

False

```
a = 257
b = 257
a is b
```

False

```
a = 256
b = 256
a is b
```

True

## Вопрос 5

Преобразуйте список ['one', 'two', 'three', 'four'] в строку.

Воспользуемся методом `.join()`, который возвращает строку, собранную из элементов указанного объекта, поддерживающего итерирование.

```
lst = ['one', 'two', 'three', 'four']
s = ''.join(lst)
s
```

'one two three four'

## Вопрос 6

У нас есть список менеджеров, и по каждому менеджеру известен результат его работы – объем продаж, нужно получить реестр вида:

```
Petrov 200000
Sidorov 100000
Ivanov 50000
```

## Ответ на вопрос 6

«Сшиваем» два списка – список с фамилиями менеджеров и список с суммами продаж – с помощью функции `zip()`.

```
members = ['Petrov', 'Sidorov', 'Ivanov']
sales = [200000, 100000, 50000]
for i, j in zip(members, sales):
    print(i, j)
```

Petrov 200000  
Sidorov 100000  
Ivanov 50000

## Вопрос 7

Как в Python узнать, в каком каталоге мы сейчас находимся?

## Ответ на вопрос 7

Необходимо воспользоваться методом `getcwd()` модуля `os`.

```
import os
os.getcwd()

'/Users/artemgruzdev/Documents/Курс/Course_ML/Модуль_1'
```

## Вопрос 8

Создайте список [1, 3, 5, 7, 9] с помощью генератора списков (нужно воспользоваться одной строкой программного кода).

## Ответ на вопрос 8

```
[i for i in range(1, 10, 2)]

[1, 3, 5, 7, 9]
```

## Вопрос 9

Обратите порядок элементов в списке [1, 3, 5, 7, 9].

## Ответ на вопрос 9

```
a = [1, 3, 5, 7, 9]
a.reverse()
a

[9, 7, 5, 3, 1]
```

## Вопрос 10

У нас есть список под названием `a` вида [9, 7, 5, 3, 1], какой элемент вернет программный код `a[-2]`?

## Ответ на вопрос 10

Он вернет 2-й с конца элемент.

```
a = [9, 7, 5, 3, 1]
a[-2]

3
```

## Вопрос 11

У нас есть два списка ['Petrov', 'Sidorov', 'Ivanov'] и [200000, 100000, 50000]. Нужно превратить их в словарь вида {'Petrov': 200000, 'Sidorov': 100000, 'Ivanov': 50000}.

## Ответ на вопрос 11

С помощью функций `dict()` и `zip()` создаем из двух списков словарь.

```
dictionary = dict(zip(members, sales))
dictionary
```

```
{'Petrov': 200000, 'Sidorov': 100000, 'Ivanov': 50000}
```

## Вопрос 12

Как убрать дубликаты из списка [1, 2, 1, 3, 4, 2]?

## Ответ на вопрос 12

Воспользуемся функцией `set()`, чтобы превратить список в множество, вспомним, что множество – это коллекция, содержащая только уникальные значения.

```
lst = [1, 2, 1, 3, 4, 2]
set(lst)
```

```
{1, 2, 3, 4}
```

## Вопрос 13

Как работает отрицательный индекс?

## Ответ на вопрос 13

В отличие от положительного индекса, отрицательный начинает поиск с конца.

```
mylist = [0, 1, 2, 3, 4, 5, 6, 7, 8]
mylist[-3]
```

```
6
```

## Вопрос 14

Необходимо получить индекс по каждому элементу списка ['Petrov', 'Ivanov', 'Sidorov'] в следующем виде:

```
(0, 'Petrov')
(1, 'Ivanov')
(2, 'Sidorov')
```

## Ответ на вопрос 14

Для решения воспользуемся функцией `enumerate()`.

```
lst = ['Petrov', 'Ivanov', 'Sidorov']
for i in enumerate(lst):
    print(i)
```

```
(0, 'Petrov')
(1, 'Ivanov')
(2, 'Sidorov')
```

## Вопрос 15

Дан список признаков `['age', 'credit', 'debt']` и список регрессионных коэффициентов `[0.4, -0.2, -0.13]`. Нужно получить список кортежей вида `[('age', 0.4), ('credit', -0.2), ('debt', -0.13)]`.

## Ответ на вопрос 15

Воспользуемся функциями `list()` и `zip()`.

```
features = ['age', 'credit', 'debt']
coeff = [0.4, -0.2, -0.13]
list_of_tuples = list(zip(features, coeff))
list_of_tuples
```

```
[('age', 0.4), ('credit', -0.2), ('debt', -0.13)]
```

## Вопрос 16

У нас есть набор данных, записанный в файле `alfa_python_test.csv`.

Для каждого уникального `id` оставьте только одну строку с самой поздней датой.

Сколько теперь в каждом городе уникальных `id`?

Сколько теперь `id` содержится в каждом множестве городов?

## Ответ на вопрос 16

Импортируем библиотеку `pandas` и загружаем данные.

```
# импортируем библиотеку pandas
import pandas as pd
# загружаем данные
df = pd.read_csv('Data/alfa_python_test.csv')
# смотрим первые 20 наблюдений
df.head(20)
```

	city	date	id
0	Kazan, St-Petersburg, Sochi	2014-10-06 03:16:18	49
1	Moscow, Kazan	2013-01-04 09:44:04	252
2	Kazan, Moscow	2013-12-04 11:39:03	208
3	St-Petersburg, Kazan	2014-09-04 10:09:53	81
4	St-Petersburg, Moscow, Sochi	2014-09-25 02:04:14	109
5	Moscow, Sochi, St-Petersburg, Kazan	2014-02-24 09:30:03	27
6	Moscow, St-Petersburg, Sochi, Kazan	2014-09-20 18:31:50	3
7	St-Petersburg, Moscow	2014-08-28 22:07:53	213
8	Moscow, St-Petersburg, Kazan, Sochi	2013-08-30 08:20:26	50
9	Kazan, Moscow, St-Petersburg	2014-07-24 00:58:48	143
10	Kazan, Sochi, Moscow, St-Petersburg	2014-04-01 23:04:58	196
11	NaN	2013-01-12 15:12:03	45
12	St-Petersburg, Moscow, Kazan, Sochi	2013-12-23 17:00:00	102
13	Sochi	2014-05-03 01:18:26	120
14	Kazan, St-Petersburg, Sochi, Moscow	2014-09-02 10:25:27	139
15	Moscow	2013-01-13 09:10:31	292
16	St-Petersburg, Moscow, Kazan	2014-02-19 16:34:49	256
17	Moscow, St-Petersburg	2014-05-30 23:58:56	287
18	Sochi	2014-07-13 12:51:25	240
19	Moscow, Kazan, Sochi, St-Petersburg	2014-01-05 01:13:11	98

Смотрим типы переменных и количество пропусков.

```
# смотрим типы переменных и наличие пропусков
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 1000 entries, 0 to 999
```

```
Data columns (total 3 columns):
```

```
# Column Non-Null Count Dtype
```

```
--- ---
```

```
0 city 805 non-null object
```

```
1 date 1000 non-null object
```

```
2 id 1000 non-null int64
```

```
dtypes: int64(1), object(2)
```

```
memory usage: 23.6+ KB
```

Приводим столбец *date* к типу *datetime*.

```
# приводим столбец date к типу datetime
```

```
df['date'] = pd.to_datetime(df['date'])
```

Отсортируем по *id* и выведем первые 10 наблюдений.

```
# отсортируем по id и выведем первые 10 наблюдений
```

```
df.sort_values('id').head(10)
```

	city	date	id
106	Kazan, Moscow, St-Petersburg	2014-10-06 20:58:34	0
157	Moscow, Sochi	2013-09-24 18:06:24	0
335	Moscow	2013-06-27 21:55:49	0
150	Sochi	2014-02-19 22:19:14	1
300	Sochi, St-Petersburg, Moscow, Kazan	2013-03-18 06:46:12	1
873	St-Petersburg, Moscow, Sochi, Kazan	2014-11-04 11:53:27	1
86	Moscow, Sochi	2014-09-14 12:37:29	1
46	Sochi, St-Petersburg, Kazan, Moscow	2014-03-20 21:25:05	2
224	Moscow, Kazan, St-Petersburg, Sochi	2014-07-02 02:04:42	2
484	NaN	2014-05-21 10:29:54	3

Итак, для каждого уникального *id* оставляем только одну строку с самой поздней датой.

```
# для каждого уникального id оставим только одну строку
# с самой поздней датой, idmax() выдает индексы
# максимальных значений, в нашем случае - в колонке 'date'
df = df.loc[df.groupby('id')['date'].idxmax()].sort_values('id')
df.head(10)
```

	city	date	id
106	Kazan, Moscow, St-Petersburg	2014-10-06 20:58:34	0
873	St-Petersburg, Moscow, Sochi, Kazan	2014-11-04 11:53:27	1
224	Moscow, Kazan, St-Petersburg, Sochi	2014-07-02 02:04:42	2
819	Moscow, Sochi	2014-10-31 13:06:28	3
983	Kazan, Moscow, Sochi, St-Petersburg	2013-06-10 22:25:47	4
470	Kazan, Sochi, Moscow	2013-03-23 11:45:54	5
76	St-Petersburg, Sochi, Moscow	2014-02-06 23:17:13	6
709	Sochi, St-Petersburg	2013-10-01 10:10:24	7
265	Kazan	2014-04-30 04:47:09	8
80	Moscow, Sochi, St-Petersburg	2013-11-07 11:38:20	9

Удаляем пропуски и смотрим, сколько теперь в каждом городе уникальных *id*.

```
# удаляем пропуски
df.dropna(inplace=True)

# создаем пустой список
cities = []
```



## Задача с собеседования (теория вероятности)

### Вопрос 1

В урне находится 15 белых, 5 красных и 10 черных шаров. Наугад извлекается 1 шар. Найти вероятность того, что он будет: а) белым; б) красным; в) черным.

### Ответ на вопрос 1

Пусть опытом будет извлечение шара из урны, а событие  $A$  состоит в том, что извлеченный из урны шар окажется белым. Тогда общее число элементарных событий  $n = 15 + 5 + 10 = 30$ , из которых  $m = 15$  элементарных событий благоприятствуют наступлению события  $A$ . Вспомним классическое определение вероятности случайного события  $A$   $P(A) = m/n$ , где  $n$  – число равновозможных исходов данного опыта, а  $m$  – число равновозможных исходов, приводящих к появлению события. В нашем случае  $P(A) = m/n = 15/30 = 1/2$ . Теперь пусть событие  $A$  состоит в том, что извлеченный из урны шар окажется красным, тогда  $P(A) = m/n = 5/30 = 1/6$ . Теперь пусть событие  $A$  состоит в том, что извлеченный из урны шар окажется черным, тогда  $P(A) = m/n = 10/30 = 1/3$ .

## Задача с собеседования (SQL)

### Вопрос 1

Из таблицы employees получите с помощью SQL-запросов:

- список сотрудников с именем 'David';
- сотрудника с минимальным возрастом;
- сотрудника с максимальным возрастом;
- список сотрудников старше 30 лет;
- список сотрудников, у которых в имени содержится буква 'j';
- средний возраст сотрудника в каждом отделе;
- количество сотрудников в каждом отделе;
- список сотрудников моложе 27 и работающих в отделе 'B'.

id	name	age	department
1	David	22	B
2	Paul	33	B
3	Jeremy	26	B
4	Jack	21	C
5	John	36	A
6	David	45	A

```

SELECT * FROM employees WHERE name='David';
SELECT MIN(age) FROM employees;
SELECT MAX(age) FROM employees;
SELECT * FROM employees WHERE age>30;
SELECT * FROM employees WHERE name LIKE '%j%'
SELECT AVG(age), department FROM employees GROUP BY department;
SELECT COUNT(age), department FROM employees GROUP BY department;
SELECT * FROM employees WHERE age<27 AND department='B'

```

## Задача с собеседования (теория вероятности)

### Вопрос 1

В урне имеется 3 белых и 4 черных шара. Из урны вытягиваются 3 шара. Найти вероятность, что хотя бы один из них окажется белым.

### Ответ на вопрос 1

Пусть событие  $A$  состоит в том, что вытянут хотя бы один белый шар. Тогда  $\bar{A}$  – ни один из вытянутых шаров не является белым.

Воспользуемся утверждением  $p(A) = 1 - p(\bar{A})$ .

По классическому определению вероятности определим  $p(\bar{A})$ :  $p(A) = \frac{m}{n}$ .

В данном случае мы имеем дело с множеством  $X = \{x_1, x_2, \dots, x_k\}$ , состоящим из  $k$  элементов.  $(k, l)$ -выборкой будет множество, состоящее из  $l$  элементов, взятых из множества  $X$ . В нашей выборке не важен порядок следования элементов, и поэтому она будет *неупорядоченной выборкой (сочетанием)*. Каждый отобранный элемент после выбора не возвращается в исходное множество и не может повторяться в данной выборке больше одного раза, поэтому у нас будет неупорядоченная  $(k, l)$ -выборка (сочетание) *без повторений*. Таким образом, для вычисления количества благоприятных исходов и количества всевозможных исходов нам нужно воспользоваться формулой для вычисления числа неупорядоченных  $(k, l)$ -выборок (сочетаний) без повторений  $C_k^l = \frac{k!}{l!(k-l)!}$ .

Рассчитаем количество благоприятных исходов (изъято 3 шара из черных шаров):

$$m = C_{7-3}^3 = C_4^3 = \frac{4!}{3!(4-3)!} = \frac{4!}{3!1!} = \frac{24}{6 \times 1} = 4.$$

Далее определим количество всевозможных исходов (общее количество вариантов для изъятия 3 шаров из 7 возможных):

$$n = C_7^3 = \frac{7!}{3!(7-3)!} = \frac{7!}{3!4!} = \frac{5040}{6 \times 24} = 35.$$

Отсюда получаем, что  $p(\bar{A}) = \frac{m}{n} = \frac{4}{35}$ .

А подставляя в  $p(A) = 1 - p(\bar{A})$ , получаем, что  $p(A) = 1 - \frac{4}{35} = \frac{31}{35}$ .

## Вопрос 2

Игральный кубик бросается 6 раз. Найти вероятность, что выпадет хотя бы одна шестерка.

## Ответ на вопрос 2

Здесь у нас речь идет о повторных независимых испытаниях. Такая последовательность испытаний называется *схемой Бернулли*, или *биномиальной схемой*, а сами испытания – *испытаниями Бернулли*.

Вероятность  $P_n(k)$  того, что в серии из  $n$  испытаний Бернулли окажется ровно  $k$  успешных, рассчитывается по *формуле Бернулли*:  $P_n(k) = C_n^k * p^k * (1-p)^{n-k}$ .

Перейдем к рассмотрению противоположного события «шестерка не выпала ни разу», т. е. в схеме Бернулли  $k = 0$  для  $n = 6$ ,  $p = 1/6$ ,  $q = 1 - p = 1 - 1/6 = 5/6$ .

Вероятность противоположного события равна

$$p(\bar{A}) = C_6^0 * \left(\frac{1}{6}\right)^0 * \left(\frac{5}{6}\right)^6 = \frac{6!}{0!(6-0)!} * \left(\frac{1}{6}\right)^0 * \left(\frac{5}{6}\right)^6 = 1 * 1 * \left(\frac{5}{6}\right)^6 = \left(\frac{5}{6}\right)^6.$$

Тогда искомая вероятность будет равна:

$$p(A) = 1 - p(\bar{A}) = 1 - \left(\frac{5}{6}\right)^6 = 1 - 0,335 = 0,665.$$

## Задачи с собеседований (математическая статистика)

### Вопрос 1

В ходе исследования были опрошены 1000 человек. 20 % из них заинтересовались новым продуктом.

Вычислите 95%-ный доверительный интервал для реальной доли заинтересованных в продукте.

### Ответ на вопрос 1

Необходимо воспользоваться формулой доверительного интервала для доли

$$\hat{p} \pm z_{\alpha/2} \sqrt{\frac{\hat{p}\hat{q}}{n}},$$

где:

$z_{\alpha/2}$  – z-значение или значение стандартного нормального распределения, определяемое в зависимости от выбранного доверительного уровня (доверительной вероятности);

$\hat{p}$  – оцениваемая доля с наличием исследуемого признака (интерес к новому продукту) в генеральной совокупности;

$\hat{q}$  – оцениваемая доля взрослых с отсутствием исследуемого признака (интерес к новому продукту) в генеральной совокупности;

$n$  – размер выборки.

Сначала вычисляем  $z_{\alpha/2}$  – z-значение для 95%-го доверительного интервала. Делим 0,95 на 2,  $0,95/2 = 0,475$ , находим по таблице z-значений ближайшее z-значение для 0,475 – 1,96.

Вычисляем  $\hat{p}$  – оцениваемую долю взрослых с наличием исследуемого признака (интерес к новому продукту) в генеральной совокупности.  $20\% = 0,2$ . Если доля неизвестна, принимаем за 50 %.

Вычисляем  $\hat{q}$  – оцениваемую долю взрослых с отсутствием исследуемого признака (интерес к новому продукту) в генеральной совокупности.

$$\hat{q} = 1 - \hat{p} = 1 - 0,2 = 0,8.$$

Размер выборки  $n = 1000$ .

Доверительный интервал для доли:

$$\hat{p} \pm z_{\alpha/2} \sqrt{\frac{\hat{p}\hat{q}}{n}} = 0,2 \pm 1,96 \sqrt{\frac{0,2 \times 0,8}{1000}} = 0,2 \pm 0,025 = 20\% \pm 2,5\%.$$

Можно сделать вывод, что наш интервал (17,5 % – 22,5 %) накрывает истинную долю заинтересованных в новом продукте (долю заинтересованных в новом продукте в генеральной совокупности) с доверительной вероятностью, приблизительно равной 95 %.

## Вопрос 2

Вычислите объем выборки, предельная ошибка которой составит 4 %. При этом мы принимаем 95%-ный доверительный уровень, генеральная совокупность значительно больше выборки, доля респондентов с наличием исследуемого признака равна 0,5. Принять, что выборка является простой случайной, объем выборки значительно меньше генеральной совокупности.

## Ответ на вопрос 2

Необходимо воспользоваться формулой  $n = \frac{z_{\gamma}^2 w(1-w)}{\Delta_w^2}$ ,

где:

$n$  – минимальный объем выборки;

$z_{\gamma}$  – z-значение или значение стандартного нормального распределения, определяемое в зависимости от выбранного доверительного уровня (доверительной вероятности)  $\gamma$ ;

$w$  – доля респондентов с наличием исследуемого признака;

$\Delta_w$  – максимально допустимая предельная ошибка оценки доли респондентов с наличием исследуемого признака (предельная ошибка выборки).

$$n = \frac{z_\gamma^2 w(1-w)}{\Delta_w^2} = \frac{1,96^2 \times 0,5 \times 0,5}{0,04^2} = 600,25 \approx 600 \text{ человек.}$$

## Задача с собеседования (теория вероятности)

### Вопрос 1

По данным ФБР, около 80 % всех преступлений против собственности остаются нераскрытыми. Предположим, что в вашем городе совершено 3 таких преступления, каждое из которых считается независимым друг от друга. Какова вероятность раскрытия *точно* одного из трех преступлений? Какова вероятность раскрытия *по крайней мере* одного преступления?

### Ответ на вопрос 1

Здесь речь идет о биномиальном распределении – это особый случай дискретного распределения, в котором есть два разных исхода: «успех» и «неудача».

Мы имеем дело с биномиальным экспериментом, если выполняются ВСЕ нижеприведенные четыре условия:

- эксперимент состоит из  $n$  идентичных испытаний;
- каждое испытание приводит к одному из двух результатов, называемых «успехом» и «неудачей»;
- вероятность «успеха», обозначаемая  $p$ , остается неизменной от испытания к испытанию;
- $n$  испытаний независимы, т. е. результат любого испытания не влияет на результаты остальных.

Тип вероятности может быть либо отдельным успехом («точно»), либо совокупностью успехов («меньше чем», «максимум», «больше чем», «по крайней мере»).

Сначала мы должны определить, удовлетворяет ли описанная в вопросе ситуация ВСЕМ четырем условиям биномиального эксперимента:

- Удовлетворяет ли она фиксированному количеству испытаний? ДА, количество испытаний равно 3 ( $n = 3$ ).
- У нее всего 2 исхода? ДА (раскрыто и не раскрыто).
- Все ли испытания имеют одинаковую вероятность успеха? ДА ( $p = 0,2$ ).
- Все ли преступления независимы? ДА (указано в описании).

Вычислим вероятность раскрытия *точно* одного из трех преступлений. Речь идет об отдельном успехе.

Чтобы определить вероятность раскрытия только одного из трех преступлений, мы сначала вычисляем вероятность раскрытия одного преступления. При трех таких событиях (преступлениях) есть три последовательности, в которых раскрывается только одно преступление:

- Первое раскрыто, второе не раскрыто, третье не раскрыто =  $(0,2) \times (0,8) \times (0,8) = 0,128$ .
- Первое не раскрыто, второе раскрыто, третье не раскрыто =  $(0,8) \times (0,2) \times (0,8) = 0,128$ .
- Первое не раскрыто, второе не раскрыто, третье раскрыто =  $(0,8) \times (0,8) \times (0,2) = 0,128$ .

Мы складываем эти вероятности и получаем  $0,128 + 0,128 + 0,128 = 0,384$ .

Для более элегантного решения мы можем воспользоваться формулой биномиальной вероятности:

$$P(X) = \binom{n}{X} \cdot p^X \cdot (1-p)^{n-X},$$

где:

$X$  – количество «успехов» (раскрытых преступлений), должно быть меньше или равно количеству испытаний;

$n$  – количество испытаний;

$p$  – вероятность «успеха» в отдельном испытании (от 0 до 1);

$\binom{n}{X}$  – биномиальный коэффициент, который вычисляется по формуле

$$\binom{n}{X} = \frac{n!}{X!(n-X)!}.$$

Восклицательный знак (!) используется в математике для обозначения факториала. Факториал числа означает, что нужно взять это число и умножить его на каждое число, стоящее перед ним, вплоть до единицы (исключая 0). Например,  $3! = 3 \times 2 \times 1 = 6$ . Помните, что  $1! = 1$  и  $0! = 1$ .

Таким образом, получаем

$$P(X) = \frac{n!}{X!(n-X)!} \cdot p^X \cdot (1-p)^{n-X};$$

$$P(1) = \frac{3!}{1!(3-1)!} \cdot 0,2^1 \cdot (1-0,2)^{3-1} = 3 \cdot 0,2 \cdot 0,8^2 = 0,384.$$

Теперь вычислим вероятность раскрытия *по крайней мере* одного из трех преступлений. Речь идет о совокупности успехов. Здесь нам нужно найти  $P(X \geq 1)$ . Формулировка «по крайней мере одно преступление» включает значения  $X = 1, 2, 3$ . Чтобы решить эту задачу, надо найти сумму биномиальных вероятностей для каждого из значений  $X$  или, если существует только одно значение  $X$ , найти вероятность  $P(X)$ . Нам нужно решить  $P(1) + P(2) + P(3)$ . Опять нужно воспользоваться формулой биномиальной вероятности для каждого значения  $X$ , а затем найти сумму индивидуальных вероятностей. На всякий случай напомним, что любое число в нулевой степени равно единице.

$$P(1) = \frac{3!}{1!(3-1)!} \cdot 0,2^1 \cdot (1-0,2)^{3-1} = 3 \cdot 0,2 \cdot 0,8^2 = 0,384,$$

$$P(2) = \frac{3!}{2!(3-2)!} \cdot 0,2^2 \cdot (1-0,2)^{3-2} = 3 \cdot 0,2^2 \cdot 0,8^1 = 0,096,$$

$$P(3) = \frac{3!}{3!(3-3)!} \cdot 0,2^3 \cdot (1-0,2)^{3-3} = 1 \cdot 0,2^3 \cdot 0,8^0 = 0,008,$$

$$P(1) + P(2) + P(3) = 0,384 + 0,096 + 0,008 = 0,48.$$

Для более короткого решения мы можем воспользоваться правилом дополнения. Здесь дополнение к  $P(X \geq 1)$  равно  $1 - P(X < 1)$ , что эквивалентно  $1 - P(X = 0)$ .

$$1 - P(0) = 1 - \frac{3!}{0!(3-0)!} \cdot 0,2^0 \cdot (1-0,2)^{3-0} = 1 - 1 \cdot 1 \cdot 0,8^3 = 1 - 0,512 = 0,488.$$

## Задача с собеседования (математическая статистика)

### Вопрос 1

1. Почему на практике мы, как правило, пользуемся бутстрепированными доверительными интервалами метрик качества (например, бутстрепированным доверительным интервалом AUC-ROC), вместо того чтобы вычислять доверительные интервалы по асимптотическому методу, пользуясь центральной предельной теоремой?

### Ответ на вопрос 1

Вычисление доверительного интервала метрики качества по асимптотическому методу сопряжено с проблемой – мы не знаем распределения метрики. Вовсе не обязательно, что метрика будет подчиняться нормальному распределению. В частности, AUC-ROC зависит от данных, с которыми мы работаем. Поэтому не существует общепринятого подхода к вычислению стандартной ошибки AUC-ROC.

## Задача с собеседования (теория вероятности)

### Вопрос 1

Светофор на пешеходном переходе одну минуту разрешает переходить улицу, а две минуты – запрещает. Найдите среднее время ожидания зеленого света пешеходом, который подошел к перекрестку.

**Ответ на вопрос 1**

С вероятностью  $1/3$  пешеход попадает в промежуток, когда ему горит зеленый свет (событие  $G$ ), и поэтому условное математическое ожидание времени ожидания  $T$  в этом случае равно 0:  $E(T|G) = 0$ . С вероятностью  $2/3$  пешеход попадает на красный сигнал и вынужден ждать (событие  $\bar{G}$ ). В этом случае время ожидания распределено равномерно на отрезке от 0 до 2 минут, поэтому условное математическое ожидание величины  $T$  при условии  $\bar{G}$  равно одной минуте:

$$E(T|\bar{G}) = \frac{0+2}{2} = 1 \text{ (мин)}.$$

Получаем:

$$E(T) = E(T|G) \cdot P(G) + E(T|\bar{G}) \cdot P(\bar{G}) = \frac{1}{3} \cdot 0 + \frac{2}{3} \cdot 1 = \frac{2}{3} \text{ мин, или } 40 \text{ с.}$$

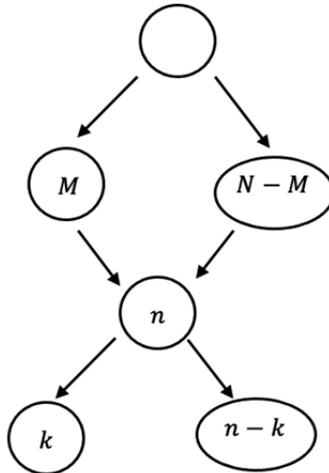
**Вопрос 2**

Из 15 билетов выигрышными являются 4. Какова вероятность того, что среди взятых наудачу 6 билетов будут 2 выигрышных?

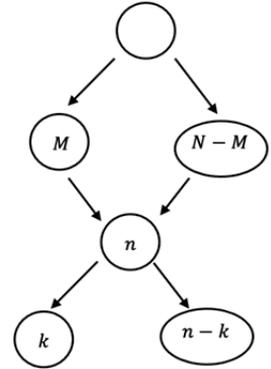
**Ответ на вопрос 2**

Перед нами – задача о выборке. Она обычно звучит следующим образом. В партии из  $N$  деталей имеется  $M$  стандартных. Наудачу отобраны  $n$  деталей. Найти вероятность того, что среди отобранных деталей ровно  $k$  стандартных.

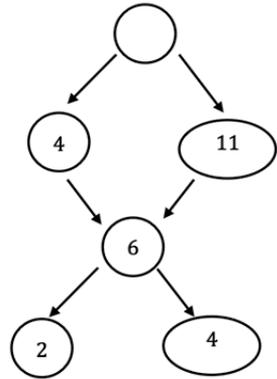
Условие задачи отображают в виде схемы:



Общее число возможных элементарных исходов испытания равно числу способов, которыми можно извлечь  $n$  деталей из  $N$ , т. е.  $C_N^n$  – числу испытаний из  $N$  элементов по  $n$ . Подсчитаем число исходов, благоприятствующих интересующему событию (среди  $n$  деталей ровно  $k$  стандартных).  $k$  стандартных деталей можно взять из  $M$  стандартных  $C_M^k$  способами, при этом остальные  $n - k$  деталей должны быть нестандартными. Взять  $n - k$  нестандартных деталей из  $N - M$  нестандартных деталей можно  $C_{N-M}^{n-k}$  способами. Следовательно, число благоприятствующих исходов по правилу произведения равно  $C_M^k \cdot C_{N-M}^{n-k}$ .



Тогда искомая вероятность  $P(A) = \frac{C_M^k \cdot C_{N-M}^{n-k}}{C_N^n}$ . Формула используется при выборочном контроле и называется гипергеометрической формулой.



Вернемся к нашему примеру. Здесь  $n = C_{15}^6$  – число способов, которыми можно извлечь 6 билетов из 15. Нас интересует событие  $A$ , состоящее в том, что 2 билета являются выигрышными (2 выигрышных билета выбираются из 4 выигрышных  $C_4^2$  способами, остальные 4 билета выбираются из 11 невыигрышных  $C_{11}^4$  способами). Поэтому  $m = C_4^2 \cdot C_{11}^4$  и

$$\begin{aligned}
 P(A) &= \frac{C_4^2 \cdot C_{11}^4}{C_{15}^6} = \frac{4!}{2!(4-2)!} \cdot \frac{11!}{4!(11-4)!} = \frac{4 \cdot 3 \cdot 2 \cdot 1}{2 \cdot 1 \cdot 2 \cdot 1} \cdot \frac{11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}{15 \cdot 14 \cdot 13 \cdot 12 \cdot 11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} = \\
 &= \frac{4 \cdot 3}{15 \cdot 14 \cdot 13 \cdot 12 \cdot 11 \cdot 10} \cdot \frac{11 \cdot 10 \cdot 9 \cdot 8}{1 \cdot 2 \cdot 3 \cdot 4} = \frac{36}{91} \approx 0,395.
 \end{aligned}$$

### Вопрос 3

Номера машин в России имеют следующий код: вначале идет буква, затем трехзначный номер, потом еще две буквы и затем двузначное число – код региона. Сколько всего можно составить различных номеров машин?

### Ответ на вопрос 3

Речь идет о числе комбинаций. Пусть нам надо выбрать по одному объекту из  $k$  групп объектов, причем в первой группе –  $n_1$  объектов, во второй –  $n_2$  объектов, в третьей –  $n_3$  объектов и т. д. Тогда первый объект можно выбрать  $n_1$  способами, независимо от него – второй,  $n_2$  способами, независимо от первых двух – третий,  $n_3$  способами и т. д., и, наконец,  $k$ -й объект выбирается независимо от всех предыдущих –  $n_k$  способами. Общее число способов выбрать  $k$  объектов, называемое *числом комбинаций*, равно  $K = n_1 \cdot n_2 \cdot n_3 \cdot \dots \cdot n_k$ .

В номерах машин используются только буквы, написание которых есть как в русском, так и в латинском алфавитах. Таких букв – 12 (А, В, Е, К, М, Н, О, Р, С, Т, У и Х). Таким образом, первую букву можно выбрать 12 способами. Ненулевых трехзначных чисел – 999. Каждую из следующих букв также можно выбрать 12 способами. Ненулевых двузначных чисел – 99. Итого всего номеров может быть  $12 \times 999 \times 12 \times 12 \times 99 = 170\,900\,928$ .

## Задачи с собеседований (Python)

### Вопрос 1

Напишите функцию, которая находит общие элементы, кратные 7 в двух списках ниже:

```
a = [2, 4, 7, 9, 14, 20, 21, 22]
b = [3, 5, 8, 10, 14, 20, 21, 30]
```

### Ответ на вопрос 1

```
def find(lst1, lst2):
    lst1 = [i for i in lst1 if i % 7 == 0]
    lst2 = [i for i in lst2 if i % 7 == 0]
    return set(lst1) & set(lst2)
```

```
find(a, b)
{14, 21}
```

### Вопрос 2

Напишите функцию, возвращающую список из элемента первого списка, отсортированный по элементам второго (порядок элементов по совпадающему «ключу» второго не важен):

```
a = ["a", "b", "c", "d", "e", "f"]
b = [1, 0, 9, 3, 2, 0]
```

## Ответ на вопрос 2

```
def get_sorted(lst1, lst2):
    lst_tmp = [x for x in zip(lst1, lst2)]
    lst_tmp.sort(key=lambda x: x[1])
    return [x[0] for x in lst_tmp]
```

```
c = get_sorted(a, b)
```

```
c
```

```
['b', 'f', 'a', 'e', 'd', 'c']
```

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «КТК Галактика» наложенным платежом, выслать открытку или письмо по почтовому адресу:

115487, г. Москва, пр. Андропова д. 38 оф. 10.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: [www.galaktika-dmk.com](http://www.galaktika-dmk.com).

Оптовые закупки: тел. (499) 782-38-89.

Электронный адрес: [books@aliens-kniga.ru](mailto:books@aliens-kniga.ru).

**Груздев Артём Владимирович**

## **Предварительная подготовка данных в Python**

Том 1. Инструменты и валидация

Главный редактор *Мовчан Д. А.*

[dmkpress@gmail.com](mailto:dmkpress@gmail.com)

Зам. главного редактора *Сенченкова Е. А.*

Корректор *Синяева Г. И.*

Верстка *Луценко С. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитура «PT Serif». Печать цифровая.

Усл. печ. л. 65,98. Тираж 100 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)