

МНЕНИЕ ЭКСПЕРТОВ

---

# Python и машинное обучение

Машинное и глубокое обучение с использованием  
Python, scikit-learn и TensorFlow 2

**3-е издание — охватывает TensorFlow 2,  
порождающие состязательные сети и  
обучение с подкреплением**

---

Себастьян Рашка  
Вахид Мирджалили



**Packt**

# **Python и машинное обучение**

*3-е издание*



# **Python Machine Learning**

## ***Third Edition***

Machine Learning and Deep Learning  
with Python, scikit-learn, and  
TensorFlow 2

**Sebastian Raschka**  
**Vahid Mirjalili**

**Packt>**

BIRMINGHAM ♦ MUMBAI

# **Python и машинное обучение**

***3-е издание***

Машинное и глубокое обучение  
с использованием Python, scikit-learn  
и TensorFlow 2

**Себастьян Рашка  
Вахид Мирджалили**



Москва ♦ Санкт-Петербург  
2020

ББК 32.973.26-018.2.75

P28

УДК 681.3.07

ООО “Диалектика”

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция Ю.Н. Артеменко

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:

info.dialektika@gmail.com, <http://www.dialektika.com>

**Рашка, Себастьян, Мирджалили, Вахид.**

**P28 Python и машинное обучение: машинное и глубокое обучение с использованием Python, scikit-learn и TensorFlow 2, 3-е изд.: Пер. с англ. — СПб. : ООО “Диалектика”, 2020. — 848 с. : ил. — Парал. тит. англ.**

ISBN 978-5-907203-57-0 (рус.)

**ББК 32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Packt Publishing Ltd.

Authorized Russian translation of the English edition of *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*, 3rd Edition (ISBN 978-1-78995-575-0) © 2019 Packt Publishing. All rights reserved.

This translation is published and sold by permission of Packt Publishing Ltd., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

*Научно-популярное издание*

**Себастьян Рашка, Вахид Мирджалили**

**Python и машинное обучение:  
машинное и глубокое обучение с использованием  
Python, scikit-learn и TensorFlow 2  
3-е издание**

Подписано в печать 10.09.2020. Формат 70х100/16

Гарнитура Times

Усл. печ. л. 68,37. Уч.-изд. л. 38,9

Тираж 500 экз. Заказ № 6013

Отпечатано в АО “Первая Образцовая типография”

Филиал “Чеховский Печатный Двор”

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: [www.chpd.ru](http://www.chpd.ru), E-mail: [sales@chpd.ru](mailto:sales@chpd.ru), тел. 8 (499) 270-73-59

ООО “Диалектика”, 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907203-57-0 (рус.)

© ООО “Диалектика”, 2020

ISBN 978-1-78995-575-0 (англ.)

© Packt Publishing, 2019

# Оглавление

<b>Предисловие</b>	<b>20</b>
<b>Глава 1. Наделение компьютеров способностью обучения на данных</b>	<b>29</b>
<b>Глава 2. Обучение простых алгоритмов МО для классификации</b>	<b>49</b>
<b>Глава 3. Обзор классификаторов на основе машинного обучения с использованием scikit-learn</b>	<b>85</b>
<b>Глава 4. Построение хороших обучающих наборов — предварительная обработка данных</b>	<b>145</b>
<b>Глава 5. Сжатие данных с помощью понижения размерности</b>	<b>185</b>
<b>Глава 6. Освоение практического опыта оценки моделей и настройки гиперпараметров</b>	<b>233</b>
<b>Глава 7. Объединение разных моделей для ансамблевого обучения</b>	<b>273</b>
<b>Глава 8. Применение машинного обучения для смыслового анализа</b>	<b>313</b>
<b>Глава 9. Встраивание модели машинного обучения в веб-приложение</b>	<b>343</b>
<b>Глава 10. Прогнозирование значений непрерывных целевых переменных с помощью регрессионного анализа</b>	<b>377</b>
<b>Глава 11. Работа с непомеченными данными — кластерный анализ</b>	<b>419</b>
<b>Глава 12. Реализация многослойной искусственной нейронной сети с нуля</b>	<b>455</b>
<b>Глава 13. Распараллеливание процесса обучения нейронных сетей с помощью TensorFlow</b>	<b>501</b>
<b>Глава 14. Погружаемся глубже — механика TensorFlow</b>	<b>555</b>
<b>Глава 15. Классификация изображений с помощью глубоких сверточных нейронных сетей</b>	<b>609</b>
<b>Глава 16. Моделирование последовательных данных с использованием рекуррентных нейронных сетей</b>	<b>665</b>
<b>Глава 17. Порождающие состязательные сети для синтеза новых данных</b>	<b>723</b>
<b>Глава 18. Обучение с подкреплением для принятия решений в сложных средах</b>	<b>781</b>
<b>Предметный указатель</b>	<b>835</b>



# Содержание

Об авторах	17
<b>Предисловие</b>	20
Начало работы с машинным обучением	20
Практика и теория	20
Почему был выбран язык Python?	21
Исследование области машинного обучения	21
Для кого предназначена эта книга?	22
Что рассматривается в этой книге?	22
Что необходимо при работе с этой книгой?	26
Соглашения	26
Загрузка кода примеров	28
Ждем ваших отзывов!	28
<b>Глава 1. Наделение компьютеров способностью обучения на данных</b>	29
Построение интеллектуальных машин для трансформирования данных в знания	30
Три типа машинного обучения	30
Выработка прогнозов о будущем с помощью обучения с учителем	31
Решение интерактивных задач с помощью обучения с подкреплением	34
Обнаружение скрытых структур с помощью обучения без учителя	36
Введение в основную терминологию и обозначения	38
Обозначения и соглашения, используемые в книге	39
Терминология, связанная с машинным обучением	41
Дорожная карта для построения систем машинного обучения	42
Предварительная обработка — приведение данных в приемлемую форму	43
Обучение и выбор прогнозирующей модели	44
Оценка моделей и прогнозирование на не встречавшихся ранее образцах данных	45
Использование Python для машинного обучения	45
Установка Python и необходимых пакетов	46
Использование дистрибутива Anaconda и диспетчера пакетов	46
Пакеты для научных расчетов, науки о данных и машинного обучения	47
Резюме	48
<b>Глава 2. Обучение простых алгоритмов МО для классификации</b>	49
Искусственные нейроны — беглое знакомство с ранней историей машинного обучения	50
Формальное определение искусственного нейрона	51
Правило обучения персептрона	53

Реализация алгоритма обучения персептрона на Python	56
Объектно-ориентированный API-интерфейс персептрона	56
Обучение модели персептрона на наборе данных Iris	60
Адаптивные линейные нейроны и сходимость обучения	67
Минимизация функций издержек с помощью градиентного спуска	68
Реализация Adaline на Python	71
Улучшение градиентного спуска посредством масштабирования признаков	75
Крупномасштабное машинное обучение и стохастический градиентный спуск	78
Резюме	84
<b>Глава 3. Обзор классификаторов на основе машинного обучения с использованием scikit-learn</b>	85
Выбор алгоритма классификации	86
Первые шаги в освоении scikit-learn — обучение персептрона	87
Моделирование вероятностей классов посредством логистической регрессии	93
Понятие логистической регрессии и условные вероятности	94
Выяснение весов функции издержек для логистической регрессии	98
Преобразование реализации Adaline в алгоритм для логистической регрессии	101
Обучение логистической регрессионной модели с помощью scikit-learn	106
Решение проблемы переобучения с помощью регуляризации	109
Классификация с максимальным зазором с помощью методов опорных векторов	113
Понятие максимального зазора	113
Обработка нелинейно сепарабельного случая с использованием фиктивных переменных	115
Альтернативные реализации в scikit-learn	117
Решение нелинейных задач с применением ядерного метода опорных векторов	118
Ядерные методы для линейно сепарабельных данных	118
Использование ядерного трюка для нахождения разделяющих гиперплоскостей в пространстве высокой размерности	121
Обучение моделей на основе деревьев принятия решений	124
Доведение до максимума прироста информации — получение наибольшей отдачи	126
Построение дерева принятия решений	131
Объединение множества деревьев принятия решений с помощью случайных лесов	135
Метод $k$ ближайших соседей — алгоритм ленивого обучения	139
Резюме	143
<b>Глава 4. Построение хороших обучающих наборов — предварительная обработка данных</b>	145
Решение проблемы с недостающими данными	146
Идентификация недостающих значений в табличных данных	146

Исключение обучающих образцов или признаков с недостающими значениями	148
Условный расчет недостающих значений	149
Понятие API-интерфейса оценщиков <code>scikit-learn</code>	150
Обработка категориальных данных	151
Кодирование категориальных данных с помощью <code>pandas</code>	152
Отображение порядковых признаков	153
Кодирование меток классов	154
Выполнение унитарного кодирования на именных признаках	155
Разбиение набора данных на отдельные обучающий и испытательный наборы	159
Приведение признаков к тому же самому масштабу	162
Выбор значимых признаков	165
Регуляризация L1 и L2 как штрафы за сложность модели	166
Геометрическая интерпретация регуляризации L2	166
Разреженные решения с регуляризацией L1	169
Алгоритмы последовательного выбора признаков	173
Оценка важности признаков с помощью случайных лесов	180
Резюме	183
<b>Глава 5. Сжатие данных с помощью понижения размерности</b>	<b>185</b>
Понижение размерности без учителя с помощью анализа главных компонент	186
Основные шаги при анализе главных компонент	186
Выделение главных компонент шаг за шагом	188
Полная и объясненная дисперсия	191
Трансформация признаков	193
Анализ главных компонент в <code>scikit-learn</code>	197
Сжатие данных с учителем посредством линейного дискриминантного анализа	200
Анализ главных компонент в сравнении с линейным дискриминантным анализом	200
Внутреннее устройство линейного дискриминантного анализа	202
Вычисление матриц рассеяния	203
Выбор линейных дискриминантов для нового подпространства признаков	205
Проецирование образцов в новое подпространство признаков	208
Реализация LDA в <code>scikit-learn</code>	209
Использование ядерного анализа главных компонент для нелинейных отображающих функций	211
Ядерные функции и ядерный трюк	212
Реализация ядерного анализа главных компонент на языке Python	217
Проецирование новых точек данных	225
Ядерный анализ главных компонент в <code>scikit-learn</code>	229
Резюме	231

<b>Глава 6. Освоение практического опыта оценки моделей и настройки гиперпараметров</b>	<b>233</b>
Модернизация рабочих потоков с помощью конвейеров	234
Загрузка набора данных Breast Cancer Wisconsin	234
Объединение преобразователей и оценщиков в конвейер	236
Использование перекрестной проверки по $k$ блокам для оценки эффективности модели	238
Метод перекрестной проверки с удержанием	239
Перекрестная проверка по $k$ блокам	240
Отладка алгоритмов с помощью кривых обучения и проверки	245
Диагностирование проблем со смещением и дисперсией с помощью кривых обучения	246
Решение проблем недообучения и переобучения с помощью кривых проверки	250
Точная настройка моделей машинного обучения с помощью решетчатого поиска	252
Настройка гиперпараметров с помощью решетчатого поиска	253
Отбор алгоритма с помощью вложенной перекрестной проверки	255
Использование других метрик оценки эффективности	257
Чтение матрицы неточностей	258
Оптимизация точности и полноты классификационной модели	260
Построение кривой рабочей характеристики приемника	263
Метрики подсчета для многоклассовой классификации	266
Решение проблемы с дисбалансом классов	267
Резюме	271
<b>Глава 7. Объединение разных моделей для ансамблевого обучения</b>	<b>273</b>
Обучение с помощью ансамблей	273
Объединение классификаторов с помощью мажоритарного голосования	278
Реализация простого классификатора с мажоритарным голосованием	279
Использование принципа мажоритарного голосования для выработки прогнозов	285
Оценка и настройка ансамблевого классификатора	289
Бэггинг — построение ансамбля классификаторов из бутстрэп-образцов	296
Коротко о бэггинге	297
Применение бэггинга для классификации образцов в наборе данных Wine	298
Использование в своих интересах слабых учеников посредством адаптивного бустинга	302
Как работает бустинг	303
Применение алгоритма AdaBoost с помощью scikit-learn	308
Резюме	312



<b>Глава 8. Применение машинного обучения для смыслового анализа</b>	313
Подготовка данных с рецензиями на фильмы IMDb для обработки текста	314
Получение набора данных с рецензиями на фильмы	314
Предварительная обработка набора данных с целью приведения в более удобный формат	315
Модель суммирования слов	317
Трансформирование слов в векторы признаков	318
Оценка важности слов с помощью приема tf-idf	320
Очистка текстовых данных	323
Переработка документов в лексемы	325
Обучение логистической регрессионной модели для классификации документов	328
Работа с более крупными данными — динамические алгоритмы и внешнее обучение	331
Тематическое моделирование с помощью латентного размещения Дирихле	335
Разбиение текстовых документов с помощью LDA	336
Реализация LDA в библиотеке scikit-learn	337
Резюме	341
<b>Глава 9. Встраивание модели машинного обучения в веб-приложение</b>	343
Сериализация подогнанных оценщиков scikit-learn	344
Настройка базы данных SQLite для хранилища данных	348
Разработка веб-приложения с помощью Flask	350
Первое веб-приложение Flask	351
Проверка достоверности и визуализация форм	353
Превращение классификатора рецензий на фильмы в веб-приложение	360
Файлы и подкаталоги — дерево каталогов	362
Реализация главного приложения как app.py	363
Настройка формы для рецензии	366
Создание шаблона страницы результатов	367
Развертывание веб-приложения на публичном сервере	370
Создание учетной записи PythonAnywhere	370
Загрузка файлов для приложения классификации рецензий на фильмы	370
Обновление классификатора рецензий на фильмы	372
Резюме	375
<b>Глава 10. Прогнозирование значений непрерывных целевых переменных с помощью регрессионного анализа</b>	377
Ведение в линейную регрессию	378
Простая линейная регрессия	378
Множественная линейная регрессия	379
Исследование набора данных Housing	381
Загрузка набора данных Housing в объект DataFrame	381

Визуализация важных характеристик набора данных	383
Просмотр взаимосвязей с использованием корреляционной матрицы	385
Реализация линейной регрессионной модели с использованием обычного метода наименьших квадратов	388
Использование градиентного спуска для выяснения параметров регрессии	389
Оценка коэффициентов регрессионной модели с помощью scikit-learn	393
Подгонка надежной регрессионной модели с использованием RANSAC	396
Оценка эффективности линейных регрессионных моделей	399
Использование регуляризированных методов для регрессии	403
Превращение линейной регрессионной модели в криволинейную — полиномиальная регрессия	405
Добавление полиномиальных членов с использованием scikit-learn	405
Моделирование нелинейных связей в наборе данных Housing	407
Обработка нелинейных связей с использованием случайных лесов	410
Регрессия на основе дерева принятия решений	411
Регрессия на основе случайного леса	413
Резюме	417
<b>Глава 11. Работа с непомеченными данными — кластерный анализ</b>	419
Группирование объектов по подобию с применением алгоритма K-Means	420
Кластеризация K-Means с использованием scikit-learn	420
Более интеллектуальный способ размещения начальных центроидов кластеров с использованием алгоритма K-Means++	426
Жесткая или мягкая кластеризация	427
Использование метода локтя для нахождения оптимального количества кластеров	430
Количественная оценка качества кластеризации через графики силуэтов	431
Организация кластеров в виде иерархического дерева	436
Группирование кластеров в восходящей манере	437
Выполнение иерархической кластеризации на матрице расстояний	439
Прикрепление дендрограмм к тепловой карте	443
Применение агломеративной иерархической кластеризации с помощью scikit-learn	445
Нахождение областей высокой плотности с помощью DBSCAN	446
Резюме	452
<b>Глава 12. Реализация многослойной искусственной нейронной сети с нуля</b>	455
Моделирование сложных функций с помощью искусственных нейронных сетей	456
Краткое повторение однослойных нейронных сетей	458
Введение в архитектуру многослойных нейронных сетей	461
Активация нейронной сети посредством прямого распространения	464
Классификация рукописных цифр	467

Получение и подготовка набора данных MNIST	468
Реализация многослойного персептрона	476
Обучение искусственной нейронной сети	488
Вычисление логистической функции издержек	488
Выработка общего понимания обратного распространения	491
Обучение нейронных сетей с помощью обратного распространения	493
О сходимости в нейронных сетях	498
Несколько слов о реализации нейронных сетей	499
Резюме	500
<b>Глава 13. Распараллеливание процесса обучения нейронных сетей с помощью TensorFlow</b>	<b>501</b>
TensorFlow и производительность обучения	502
Проблемы, связанные с производительностью	502
Что такое TensorFlow?	504
Как мы будем изучать TensorFlow	506
Первые шаги при работе с библиотекой TensorFlow	506
Установка TensorFlow	506
Создание тензоров в TensorFlow	507
Манипулирование типом данных и формой тензора	508
Применение математических операций к тензорам	509
Расщепление, укладывание стопкой и объединение тензоров	511
Построение входных конвейеров с использованием <code>tf.data</code> — API-интерфейса Dataset библиотеки TensorFlow	513
Создание объекта Dataset из существующих тензоров	514
Объединение двух тензоров в общий набор данных	515
Тасование, создание пакетов и повторение	516
Создание набора данных из файлов на локальном диске	520
Извлечение доступных наборов данных из библиотеки <code>tensorflow_datasets</code>	524
Построение нейросетевой модели в TensorFlow	530
API-интерфейс Keras в TensorFlow ( <code>tf.keras</code> )	530
Построение линейной регрессионной модели	531
Обучение модели с помощью методов <code>.compile()</code> и <code>.fit()</code>	536
Построение многослойного персептрона для классификации цветков в наборе данных Iris	538
Оценка обученной модели на испытательном наборе данных	542
Сохранение и повторная загрузка обученной модели	543
Выбор функций активации для многослойных нейронных сетей	544
Краткое повторение логистической функции	545
Оценка вероятностей классов в многоклассовой классификации через многопеременную логистическую функцию	547
Расширение выходного спектра с использованием гиперболического тангенса	548

Активация на основе выпрямленного линейного элемента	551
Резюме	553
<b>Глава 14. Погружаемся глубже — механика TensorFlow</b>	555
Ключевые средства TensorFlow	556
Вычислительные графы TensorFlow: переход на TensorFlow v2	558
Понятие вычислительных графов	558
Создание графа в TensorFlow v1.x	559
Перенос графа в TensorFlow v2	560
Загрузка входных данных в модель: стиль TensorFlow v1.x	561
Загрузка входных данных в модель: стиль TensorFlow v2	561
Увеличение вычислительной мощности с помощью декораторов функций	562
Объекты Variable библиотеки TensorFlow для хранения и обновления параметров модели	565
Расчет градиентов посредством автоматического дифференцирования и GradientTape	569
Расчет градиентов потери по отношению к обучаемым переменным	570
Расчет градиентов по отношению к необучаемым тензорам	571
Сохранение ресурсов для множества вычислений градиентов	572
Упрощение реализаций распространенных архитектур посредством API-интерфейса Keras	573
Решение задачи классификации XOR	577
Увеличение гибкости построения моделей с помощью функционального API-интерфейса Keras	583
Реализация моделей на основе класса Model библиотеки Keras	584
Реализация специальных слоев Keras	586
Оценщики TensorFlow	590
Работа со столбцами признаков	591
Машинное обучение с использованием готовых оценщиков	596
Использование оценщиков для классификации рукописных цифр MNIST	601
Создание специального оценщика из существующей модели Keras	604
Резюме	606
<b>Глава 15. Классификация изображений с помощью глубоких сверточных нейронных сетей</b>	609
Строительные блоки сверточных нейронных сетей	610
Понятие сетей CNN и иерархий признаков	611
Выполнение дискретных сверток	613
Слои подвыборки	624
Группирование всего вместе — реализация сверточной нейронной сети	626
Работа с множественными входными или цветовыми каналами	627
Регуляризация нейронной сети с помощью отключения	631



Функции потерь для классификации	635
Реализация глубокой сверточной нейронной сети с использованием TensorFlow	638
Архитектура многослойной сверточной нейронной сети	638
Загрузка и предварительная обработка данных	639
Реализация сверточной нейронной сети с использованием API-интерфейса Keras библиотеки TensorFlow	641
Классификация полов по изображениям лиц с использованием сверточной нейронной сети	648
Загрузка набора данных CelebA	648
Трансформация изображений и дополнение данных	649
Обучение классификатора полов, основанного на сверточной нейронной сети	656
Резюме	662
<b>Глава 16. Моделирование последовательных данных с использованием рекуррентных нейронных сетей</b>	665
Понятие последовательных данных	666
Моделирование последовательных данных — вопросы порядка	666
Представление последовательностей	667
Категории моделирования последовательностей	668
Рекуррентные нейронные сети для моделирования последовательностей	670
Механизм организации циклов рекуррентной нейронной сети	670
Вычисление активаций в сети RNN	673
Рекуррентность скрытого слоя или рекуррентность выходного слоя	676
Сложности изучения долгосрочных взаимодействий	680
Ячейки долгой краткосрочной памяти	681
Реализация многослойных рекуррентных нейронных сетей для моделирования последовательностей в TensorFlow	684
Проект номер один — прогнозирование отношения в рецензиях на фильмы IMDb	685
Подготовка данных с рецензиями на фильмы	685
Слои вложений для кодирования предложений	691
Построение модели на основе рекуррентной нейронной сети	694
Проект номер два — моделирование языка на уровне символов в TensorFlow	702
Понимание языка с помощью модели “Преобразователь”	716
Механизм самовнимания	717
Многоголовое внимание и блок “Преобразователь”	720
Резюме	722
<b>Глава 17. Порождающие состязательные сети для синтеза новых данных</b>	723
Понятие порождающих состязательных сетей	724
Начало работы с автокодировщиками	725
Порождающие модели для синтеза новых данных	727
Генерирование новых образцов с помощью порождающих состязательных сетей	729

Функции потерь сетей генератора и дискриминатора в модели GAN	731
Реализация порождающей состязательной сети с нуля	733
Обучение моделей GAN в среде Google Colab	734
Реализация сетей генератора и дискриминатора	737
Определение обучающего набора данных	742
Обучение модели GAN	744
Повышение качества синтезированных изображений с использованием сверточной сети GAN и сети GAN Вассерштейна	753
Транспонированная свертка	753
Пакетная нормализация	756
Реализация генератора и дискриминатора	759
Меры несходства между двумя распределениями	766
Практическое использование расстояния Вассерштейна для порождающих состязательных сетей	770
Штраф градиента	771
Реализация порождающей состязательной сети Вассерштейна со штрафом градиента для обучения модели DCGAN	772
Коллапс мод	777
Другие приложения порождающих состязательных сетей	778
Резюме	780
<b>Глава 18. Обучение с подкреплением для принятия решений в сложных средах</b>	781
Введение — обучение на опыте	782
Понятие обучения с подкреплением	782
Определение интерфейса “агент–среда” системы обучения с подкреплением	785
Теоретические основы обучения с подкреплением	786
Марковские процессы принятия решений	787
Математическая формулировка марковских процессов принятия решений	787
Терминология обучения с подкреплением: отдача, политика и функция ценности	791
Динамическое программирование с использованием уравнения Беллмана	796
Алгоритмы обучения с подкреплением	797
Динамическое программирование	798
Обучение с подкреплением с помощью метода Монте-Карло	801
Обучение методом временных разностей	804
Реализация первого алгоритма обучения с подкреплением	808
Введение в комплект инструментов OpenAI Gym	808
Решение задачи с миром сетки с помощью Q-обучения	818
Обзор глубокого Q-обучения	823
Резюме по главе и по книге	832
<b>Предметный указатель</b>	835

## Об авторах

**Себастьян Рашка** получил докторскую степень в Университете штата Мичиган, где занимался разработкой методов на стыке вычислительной биологии и машинного обучения. Летом 2018 года он получил должность старшего преподавателя статистики в Висконсинском университете в Мэдисоне. Его исследовательская деятельность включает разработку новых архитектур глубокого обучения для решения задач в области биометрии.

Себастьян обладает многолетним опытом написания кода на языке Python и проводил многочисленные семинары по практическому применению науки о данных, машинного обучения и глубокого обучения, в числе которых ведущая конференция, посвященная научным расчетам с помощью Python.

Среди достижений Себастьяна — его книга “Python Machine Learning”, которая стала бестселлером в Packt и Amazon.com. Книга получила награду ACM Computing Reviews’ Best of 2016 и была переведена на многие языки, включая немецкий, корейский, китайский, японский, русский, польский и итальянский.

В свободное время Себастьян любит участвовать в проектах с открытым кодом, а методы, которые он реализовал, теперь успешно используются в состязаниях по машинному обучению, таких как Kaggle.

Я хотел бы воспользоваться этой возможностью, чтобы поблагодарить замечательное сообщество Python и разработчиков пакетов с открытым кодом, которые помогли мне создать идеальную среду для научных исследований и науки о данных. Также я хочу поблагодарить своих родителей, всегда поощряющих и поддерживающих меня в выборе пути и занятия, в которое я страстно влюблен.

Выражаю особую благодарность основным разработчикам scikit-learn и TensorFlow. Как участник этого проекта и пользователь, я получил удовольствие от работы с прекрасными людьми, которые не только очень хорошо осведомлены в том, что касается машинного обучения, но также являются великолепными программистами.

**Вахид Мирджалили** получил степень PhD по машиностроению, работая над новаторскими методами для крупномасштабных вычислительных эмуляций молекулярных структур в Университете штата Мичиган. Будучи увлеченным областью машинного обучения, он был принят в лабораторию iProVe Университета штата Мичиган, где занимался применением машинного обучения в областях компьютерного зрения и биометрии. После нескольких продуктивных лет, проведенных в лаборатории iProVe, и многих лет нахождения в академических кругах Вахид недавно стал научным сотрудником в 3M Company, где может использовать свой опыт и применять современные методики машинного и глубокого обучения для решения реальных задач в разнообразных приложениях, направленных на улучшение нашей жизни.

Я хотел бы поблагодарить свою жену Табан Эслами, оказавшую мне большую поддержку и поощрявшую мой карьерный путь. Кроме того, выражаю особую благодарность моим руководителям Николаю Приезжеву, Майклу Фейгу и Аруну Россу за поддержку меня во время обучения в аспирантуре, а также моим профессорам Вишну Бодети, Лесли Куну и Сяомину Лю, которые научили меня настолько многому и вдохновили двигаться своим путем.



## 0 технических рецензентах

**Рагхав Бали** — старший научный сотрудник в одной из крупнейших в мире организаций здравоохранения. Его работа связана с исследованиями и разработкой корпоративных решений на основе машинного обучения, глубокого обучения и обработки естественного языка для сценариев использования, касающихся здравоохранения и страхования. На своей предыдущей должности в компании Intel он принимал участие в реализации упреждающих и управляемых данными инициатив в области информационных технологий, которые задействовали обработку естественного языка, глубокое обучение и традиционные статистические методы. Он также трудился в сфере финансов в American Express, решая задачи цифрового привлечения и удержания клиентов.

Рагхав также является автором нескольких книг, опубликованных ведущими издательствами, самая недавняя из которых посвящена последним достижениям в исследовании обучения методом передачи знаний.

Рагхав получил степень магистра (с отличием) по информационным технологиям в Международном институте информационных технологий, Бангалор. Он любит читать и фотографировать, запечатлевая те моменты, когда он не занят решением задач.

**Мотаз Саад** получил степень PhD по компьютерным наукам в Университете Лотарингии. Он любит данные и ему нравится с ними играть. Обладает более чем десятилетним опытом в области обработки естественного языка, вычислительной лингвистики, науки о данных и машинного обучения. В настоящее время работает старшим преподавателем на факультете информационных технологий, IUG.

# ПРЕДИСЛОВИЕ

**Б**лагодаря новостям и социальным медиаресурсам вам наверняка хорошо известен тот факт, что машинное обучение стало одной из самых захватывающих технологий нашего времени. Крупные компании, такие как Google, Facebook, Apple, Amazon и IBM, вполне обоснованно делают значительные инвестиции в исследования и приложения машинного обучения. Хотя может показаться, что сейчас машинное обучение превратилось в надоедливое словечко, это определенно не пускание пыли в глаза. Сфера машинного обучения открывает путь к новым возможностям и становится незаменимой в повседневной жизни. Подумайте о разговоре с голосовым помощником в наших смартфонах, рекомендации правильного товара нашим заказчикам, предотвращении мошенничества с кредитными картами, фильтрации спама из почтовых ящиков, обнаружении и диагностировании медицинских заболеваний — список можно продолжать и продолжать.

## Начало работы с машинным обучением

Если вы хотите стать специалистом-практиком в области машинного обучения, лучше решать задачи или, может быть, даже подумываете о том, чтобы заняться исследованиями в этой сфере, тогда настоящая книга для вас! Для новичка теоретические концепции, лежащие в основе машинного обучения, могут оказаться непреодолимыми, но в последние годы вышло много книг, ориентированных на практику, которые помогут начать работу с машинным обучением через реализацию мощных алгоритмов обучения.

## Практика и теория

Ознакомление с практическими примерами кода и проработка образцов приложений — замечательный способ погрузиться в эту сферу. Вдобавок конкретные примеры помогают проиллюстрировать более широкие концеп-

ции, применяя изложенный материал сразу на практике. Тем не менее, помните о том, что большая мощь предполагает и большую ответственность!

Кроме предложения практического опыта работы с машинным обучением, используя язык программирования Python и библиотеки на Python для машинного обучения, эта книга ознакомит вас с математическими концепциями, лежащими в основе алгоритмов машинного обучения, которые жизненно важны для успешного применения машинного обучения. Следовательно, данная книга не является чисто практической; это книга, в которой обсуждаются необходимые детали, касающиеся концепций машинного обучения, а также предлагаются интуитивно понятные и вместе с тем информативные объяснения того, как работают алгоритмы машинного обучения, как их использовать, и что самое важное, как избежать распространенных ловушек.

## **Почему был выбран язык Python?**

Прежде чем углубляться в область машинного обучения, мы ответим на самый важный ваш вопрос: почему был выбран язык Python? Ответ прост: он мощный, но очень доступный. Python стал наиболее популярным языком программирования для науки о данных, потому что позволяет нам забыть о скучных частях программирования и предлагает среду, где мы можем быстро набросать пришедшие в голову идеи и воплотить их в жизнь.

## **Исследование области машинного обучения**

Если вы введете поисковый термин “machine learning” (“машинное обучение”) в системе Google Scholar, то она возвратит ошеломляюще огромное количество публикаций — 4 890 000 (по состоянию на июнь 2020 года — *Примеч.пер.*). Конечно, обсудить особенности и детали всех алгоритмов и приложений, появившихся за прошедшие 60 лет, попросту нереально. Однако в книге мы совершим захватывающее путешествие, которое охватит все жизненно важные темы и концепции, чтобы дать вам хороший старт в освоении данной области. На тот случай, если вы обнаружите, что ваша жажда знаний не удовлетворена, в книге приводятся многочисленные ссылки на полезные ресурсы, которыми можно воспользоваться для слежения за выдающимися достижениями в этой сфере.

Мы как авторы искренне заявляем, что исследование машинного обучения сделало нас лучшими учеными, мыслителями и решателями задач.

В этой книге мы хотим поделиться с вами этим знанием. Знание добывается изучением, ключом к которому служит наш энтузиазм, а подлинное мастерство владения навыками может быть достигнуто только практикой.

Дорога впереди временами может быть ухабистой, а некоторые темы более сложными, чем другие, но мы надеемся на то, что вы воспользуетесь возможностью и сконцентрируетесь на вознаграждении. Не забывайте, что мы путешествуем вместе, и повсюду в книге мы будем добавлять в ваш арсенал многие мощные приемы, которые помогут решить даже самые трудные задачи в управляемой данными манере.

## **Для кого предназначена эта книга?**

Если вы уже хорошо знаете теорию машинного обучения, тогда книга покажет, как применить имеющиеся знания на практике. Если вы использовали приемы машинного обучения ранее и хотите лучше понять, как фактически работает машинное обучение, то эта книга для вас.

Не беспокойтесь, если вы — полный новичок в области машинного обучения; у вас есть даже еще больше поводов для заинтересованности! Можно надеяться, что машинное обучение изменит ваше представление о задачах, подлежащих решению, и покажет, как взяться за них, высвободив всю мощь данных. Если вы хотите узнать, как применять язык Python, чтобы начать отвечать на критически важные вопросы об имеющихся данных, тогда приступайте к чтению этой книги. Начинаете вы с нуля или расширяете свои познания в области науки о данных, настоящая книга будет неотъемлемым и незаменимым ресурсом.

## **Что рассматривается в этой книге?**

В главе 1, “Наделение компьютеров способностью обучения на данных”, мы предложим введение в основные подобласти машинного обучения для решения разнообразных задач. Вдобавок в главе обсуждаются важные шаги по созданию типового конвейера построения моделей машинного обучения, который будет направлять нас через последующие главы.

В главе 2, “Обучение простых алгоритмов МО для классификации”, мы обратимся к истокам машинного обучения, представив классификаторы на основе двоичного персептрона и адаптивных линейных нейронов. Глава является кратким введением в фундаментальные основы классификации об-

разцов и сосредоточена на взаимодействии алгоритмов оптимизации и машинного обучения.

В главе 3, **“Обзор классификаторов на основе машинного обучения с использованием scikit-learn”**, описаны важные алгоритмы машинного обучения, предназначенные для классификации, и приведены практические примеры применения одной из самых популярных и всеобъемлющих библиотек машинного обучения с открытым кодом — scikit-learn.

В главе 4, **“Построение хороших обучающих наборов — предварительная обработка данных”**, показано, как иметь дело с распространенными проблемами в необработанных наборах данных, такими как недостающие данные. В ней обсуждаются подходы к идентификации наиболее информативных признаков в наборах данных, а также объясняется, каким образом подготовить переменные разных типов с целью использования в качестве надлежащего входа для алгоритмов машинного обучения.

В главе 5, **“Сжатие данных с помощью понижения размерности”**, описаны важные приемы сокращения количества признаков в наборе данных с целью получения меньших наборов, которые все же сохраняют большинство полезной и отличительной информации. Здесь также обсуждается стандартный подход понижения размерности посредством анализа главных компонент, а также проводится его сравнение с приемами линейной и нелинейной трансформации с учителем.

В главе 6, **“Освоение практического опыта оценки моделей и настройки гиперпараметров”**, обсуждаются правила для оценки эффективности прогнозирующих моделей. Кроме того, в ней описаны различные метрики для измерения эффективности моделей и методики для точной настройки алгоритмов машинного обучения.

В главе 7, **“Объединение разных моделей для ансамблевого обучения”**, представлены концепции рационального объединения нескольких алгоритмов обучения. В ней объясняется, как построить ансамбль экспертов, чтобы преодолеть слабость индивидуальных учеников, вырабатывая в результате более точные и надежные прогнозы.

В главе 8, **“Применение машинного обучения для смыслового анализа”**, обсуждаются важные шаги для преобразования текстовых данных в содержательные представления для алгоритмов машинного обучения, прогнозирующих мнения людей на основе их текстов.

В главе 9, **“Встраивание модели машинного обучения в веб-приложение”**, продолжается работа с прогнозирующей моделью из предыдущей главы, а также демонстрируются важные шаги разработки веб-приложений со встроеными моделями машинного обучения.

В главе 10, **“Прогнозирование значений непрерывных целевых переменных с помощью регрессионного анализа”**, описаны приемы моделирования линейных взаимосвязей между целевыми и объясняющими переменными для прогнозирования значений с непрерывным масштабом. После представления различных линейных моделей также обсуждаются подходы с полиномиальной регрессией и на основе деревьев.

В главе 11, **“Работа с непомеченными данными — кластерный анализ”**, внимание переключается на другую подобласть машинного обучения — обучение без учителя. Здесь раскрываются алгоритмы из трех фундаментальных семейств алгоритмов кластеризации, которые ищут группы объектов, обладающих определенной степенью подобия.

В главе 12, **“Реализация многослойной искусственной нейронной сети с нуля”**, расширяется представленная в главе 2 концепция оптимизации, основанная на градиентах. Мы будем с помощью Python строить мощные многослойные *нейронные сети* на базе популярного алгоритма обратного распространения.

В главе 13, **“Распараллеливание процесса обучения нейронных сетей с помощью TensorFlow”**, опираясь на материал предыдущей главы, предоставляется практическое руководство по более эффективному обучению нейронных сетей. Основное внимание в главе сосредоточено на TensorFlow 2.0 — библиотеке Python с открытым кодом, которая позволяет задействовать множество ядер современных графических процессоров и конструировать глубокие нейронные сети из распространенных строительных блоков посредством дружественного к пользователю API-интерфейса Keras.

В главе 14, **“Погружаемся глубже — механика TensorFlow”**, изложение продолжается с места, где оно было оставлено в предыдущей главе, и представляются более сложные концепции и функциональность TensorFlow 2.0. Библиотека TensorFlow чрезвычайно обширна и развита, и в главе рассматриваются такие концепции, как компилирование кода в статический граф для более быстрого выполнения и определение обучаемых параметров модели. Кроме того, в главе предлагается дополнительный практический опыт обу-

чения глубоких нейронных сетей с использованием API-интерфейса Keras библиотеки TensorFlow, а также готовые оценщики TensorFlow.

В главе 15, **“Классификация изображений с помощью глубоких сверточных нейронных сетей”**, вводятся *сверточные нейронные сети*. Сверточная нейронная сеть представляет собой отдельный тип архитектуры глубоких нейронных сетей, которая особенно хорошо подходит для наборов данных с изображениями. Благодаря превосходной эффективности в сравнении с традиционными подходами сверточные нейронные сети теперь широко применяются в компьютерном зрении, добиваясь самых современных результатов для задач распознавания изображений. В ходе чтения этой главы вы узнаете, каким образом сверточные слои можно использовать в качестве мощных средств выделения признаков при классификации изображений.

В главе 16, **“Моделирование последовательных данных с использованием рекуррентных нейронных сетей”**, вводится еще одна популярная архитектура нейронных сетей для глубокого обучения, которая особенно хорошо подходит, когда нужно работать с текстом и другими типами последовательных данных и данных временных рядов. В качестве разминки в главе представлены рекуррентные нейронные сети для прогнозирования отношений рецензентов на фильмы. Затем в главе раскрывается процесс обучения рекуррентных сетей для систематизации информации из книг с целью генерирования нового текста.

В главе 17, **“Порождающие состязательные сети для синтеза новых данных”**, представлен популярный режим состязательного обучения для нейронных сетей, который можно применять, чтобы генерировать новые реалистично выглядящие изображения. Глава начинается с краткого введения в автокодировщики — отдельный тип архитектуры нейронных сетей, которую можно использовать для сжатия данных. Затем в главе будет показано, как объединить часть декодировщика со второй нейронной сетью, которая способна проводить различие между настоящими и синтезированными изображениями. Позволив двум нейронным сетям соревноваться друг с другом при состязательном обучении, вы реализуете порождающую состязательную сеть, которая генерирует новые изображения с рукописными цифрами. Наконец, после представления базовых концепций порождающих состязательных сетей в главе рассматриваются улучшения, которые могут стабилизировать состязательное обучение, такие как применение метрики в виде расстояния Васерштейна.

В главе 18, “Обучение с подкреплением для принятия решений в сложных средах”, раскрывается подкатегория машинного обучения, которая обычно используется для обучения роботов и других автономных систем. Глава начинается с введения в основы обучения с подкреплением, чтобы ознакомить вас с взаимодействиями агента со средой, процессом награды систем обучения с подкреплением и концепцией обучения на опыте. В главе рассматриваются две главные категории обучения с подкреплением — на основе модели и без модели. После того, как вы узнаете о базовых алгоритмических подходах, таких как метод Монте-Карло и метод временных разностей, будет предложена реализация и обучение агента, который способен перемещаться в среде мира сетки с применением алгоритма Q-обучения. В заключение главы представлен алгоритм глубокого Q-обучения, который является разновидностью Q-обучения, использующей глубокие нейронные сети.

## Что необходимо при работе с этой книгой?

Выполнение примеров кода, приводимых в книге, требует установки Python 3.7.0 или более новой версии на машине с macOS, Linux или Microsoft Windows. В книге будут повсеместно применяться важные библиотеки Python для научных расчетов, такие как SciPy, NumPy, scikit-learn, Matplotlib и pandas.

В главе 1 будут представлены инструкции и полезные советы по настройке среды Python и указанных основных библиотек. К имеющейся совокупности мы добавим дополнительные библиотеки, предоставляя инструкции по установке в соответствующих главах, например, библиотеку NLTK для обработки естественного языка (глава 8), веб-фреймворк Flask (глава 9) и библиотеку TensorFlow для эффективного обучения нейронных сетей на графических процессорах (главы 13–18).

## Соглашения

Для представления различных видов информации в книге используется несколько стилей текста. Ниже приведен ряд примеров таких стилей с объяснениями, что они означают.



Фрагменты кода в тексте представляются в следующем виде: “Уже установленные пакеты можно обновить, указав флаг `--upgrade`”.

Вот как представляется блок кода:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)
>>> X = df.iloc[0:100, [0, 2]].values
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='x', label='setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='o', label='versicolor')
>>> plt.xlabel('sepal length')
>>> plt.ylabel('petal length')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Любой ввод или вывод в командной строке записывается так:

```
> dot -Tpng tree.dot -o tree.png
```

*Новые термины и важные слова* выделяются *курсивом*. Слова, которые отображаются на экране, например, в меню или диалоговых окнах, выделяются следующим образом: “Щелкните на пункте Change runtime type (Изменить тип исполняющей среды) в меню Runtime (Исполняющая среда) тетради и выберите в раскрывающемся списке Hardware accelerator (Аппаратный ускоритель) вариант GPU (ГП)”.



На  
заметку!

Здесь приводятся предостережения и важные примечания.



Совет

Здесь даются советы и трюки.

## Загрузка кода примеров

Исходный код всех примеров, рассмотренных в книге, доступен для загрузки по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition>. После загрузки распакуйте файл с помощью последней версии архиваторов:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

## Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: [info.dialektika@gmail.com](mailto:info.dialektika@gmail.com)

WWW: <http://www.dialektika.com>



Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/pythonml>



# НАДЕЛЕНИЕ КОМПЬЮТЕРОВ СПОСОБНОСТЬЮ ОБУЧЕНИЯ НА ДАННЫХ

**П**о нашему мнению, *машинное обучение* (МО) как практическое применение и наука об алгоритмах, которым понятен смысл данных, является самой захватывающей областью компьютерных наук! Мы живем в эпоху изобилия данных; используя самообучающиеся алгоритмы из области МО, мы можем превратить эти данные в знания. Благодаря многочисленным мощным библиотекам с открытым кодом, которые были разработаны в последние годы, пожалуй, никогда еще не было лучшего времени, чтобы заняться областью МО и научиться задействовать мощные алгоритмы для выявления шаблонов в данных и выработки прогнозов о будущих событиях.

В настоящей главе вы узнаете об основных концепциях и различных типах МО. Наряду с базовым введением в важную терминологию мы зложим фундамент для успешного применения приемов МО при решении практических задач.

В главе будут раскрыты следующие темы:

- общие понятия МО;
- три типа обучения и основная терминология;
- строительные блоки для успешного проектирования систем МО;
- установка и настройка Python для анализа данных и МО.

## Построение интеллектуальных машин для трансформирования данных в знания

При нынешней зрелости современных технологий есть один ресурс, которого у нас вдоволь: значительный объем структурированных и неструктурированных данных. Во второй половине двадцатого века МО развивалось как подобласть *искусственного интеллекта* (ИИ), вовлекающая самообучающиеся алгоритмы, которые выводили знания из данных с целью выработки прогнозов. Вместо того чтобы требовать от людей выводить правила вручную и строить модели путем анализа крупных объемов данных, МО предлагает более эффективную альтернативу для сбора знаний в данных, которая постепенно улучшает эффективность прогнозирующих моделей и принимает решения, управляемые данными.

Важность МО возрастает не только в исследованиях, имеющих отношение к компьютерным наукам; его роль в нашей повседневной жизни становится даже еще больше. МО позволяет нам пользоваться надежными фильтрами почтового спама, удобным ПО распознавания текста и речи, испытанными поисковыми механизмами и перспективными программами игры в шахматы. Надо надеяться, что скоро мы добавим к этому перечню безопасные и эффективные беспилотные автомобили. Кроме того, заметный прогресс был достигнут в медицинских приложениях; например, исследователи продемонстрировали, что модели глубокого обучения способны обнаруживать рак кожи с почти человеческой точностью (<https://www.nature.com/articles/nature21056>). Еще одна веха была недавно достигнута исследователями из DeepMind, которые использовали глубокое обучение для прогнозирования третичных структур белков, впервые превзойдя физические подходы (<https://deepmind.com/blog/alphafold/>).

## Три типа машинного обучения

В этом разделе мы взглянем на три типа МО: *обучение с учителем* (*supervised learning*), *обучение без учителя* (*unsupervised learning*) и *обучение с подкреплением* (*reinforcement learning*). Мы объясним фундаментальные отличия между указанными тремя типами обучения и с помощью концептуальных примеров разовьем интуитивное понимание реальных предметных областей, где они могут быть применены (рис. 1.1).

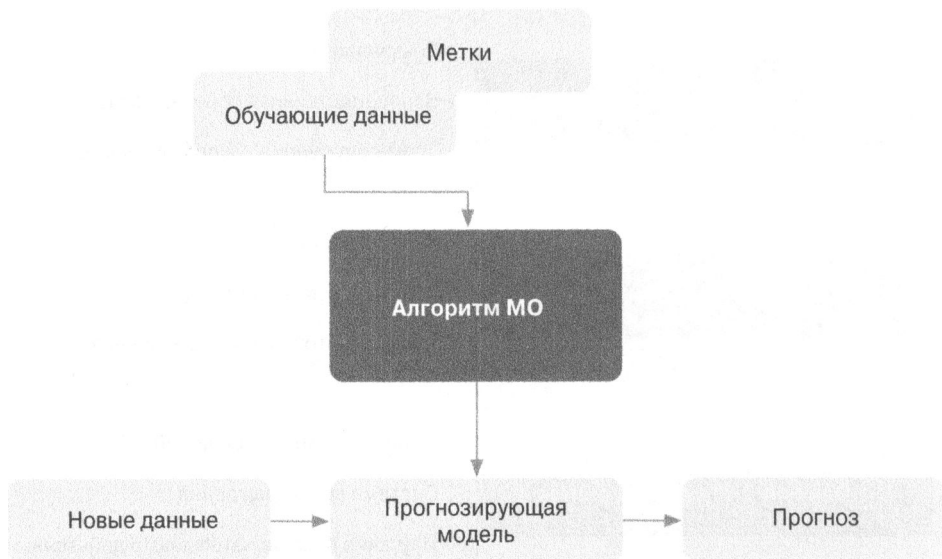


*Рис. 1.1. Типы обучения и предметные области, в которых они могут применяться*

## Выработка прогнозов о будущем с помощью обучения с учителем

Главная цель обучения с учителем — обучить модель на помеченных обучающих данных, что позволит вырабатывать прогнозы на не встречавшихся ранее или будущих данных. Здесь понятие “с учителем” относится к набору обучающих образцов (входных данных), где желаемые выходные сигналы (метки) уже известны. На рис. 1.2 представлена типичная последовательность действий при обучении с учителем, где помеченные обучающие данные передаются алгоритму МО для подгонки к прогнозирующей модели, которая может вырабатывать прогнозы на новых непомеченных входных данных.

Рассматривая в качестве примера фильтрацию почтового спама, мы можем обучить модель с использованием алгоритма машинного обучения с учителем на корпусе помеченных почтовых сообщений, т.е. сообщений, которые корректно маркированы как спам или не спам, и прогнозировать, к какой из этих двух категорий принадлежит новое сообщение. Задача обучения с учителем с метками дискретных классов, такими как в нашем примере фильтрации почтового спама, также называется *задачей классификации*. Другой подкатегорией обучения с учителем является *регрессия*, в которой результирующий сигнал представляет собой непрерывную величину.



*Рис. 1.2. Типичная последовательность действий при обучении с учителем*

### **Классификация для прогнозирования меток классов**

Классификация — это подкатегория обучения с учителем, где целью будет прогнозирование категориальных меток классов, к которым принадлежат новые образцы, на основе прошлых наблюдений. Такие метки классов представляют собой дискретные неупорядоченные значения, которые могут пониматься как принадлежность к группам образцов. Ранее упомянутый пример выявления почтового спама демонстрировал типичную задачу двоичной классификации, когда алгоритм МО изучал набор правил, чтобы проводить различие между двумя возможными классами: спамными и неспамными почтовыми сообщениями.

На рис. 1.3 иллюстрируется концепция задачи двоичной классификации с имеющимися 30 обучающими образцами; 15 обучающих образцов помечены как отрицательный класс (знаком “минус”) и 15 обучающих образцов помечены как положительный класс (знаком “плюс”). При таком сценарии наш набор данных оказывается двумерным, т.е. с каждым образцом ассоциированы два значения:  $x_1$  и  $x_2$ . Далее мы можем воспользоваться алгоритмом МО с учителем, чтобы узнать правило (границу решений, представленную в виде пунктирной линии), которое способно отделять эти два класса и относить новые данные к каждой из двух категорий, имея их значения  $x_1$  и  $x_2$ .

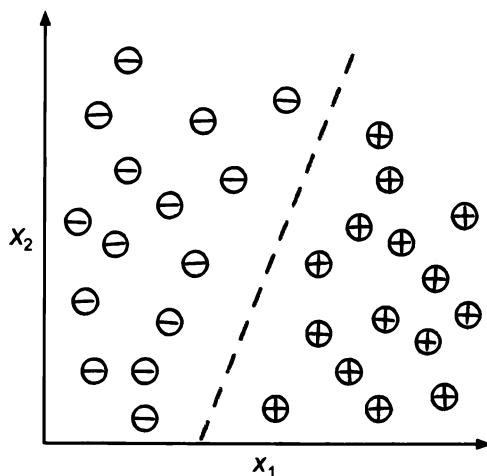


Рис. 1.3. Концепция задачи двоичной классификации

Однако набор меток классов вовсе не обязан иметь двоичную природу. Прогнозирующая модель, обученная алгоритмом обучения с учителем, способна назначать новому непомеченному образцу любую метку класса, которая была представлена в обучающем наборе данных.

Характерным примером задачи *многоклассовой классификации* считается распознавание рукописных символов. Мы можем подготовить обучающий набор данных, содержащий множество рукописных примеров для каждой буквы алфавита. Буквы (А, В, С и т.д.) будут представлять различные неупорядоченные категории или метки классов, которые мы хотим прогнозировать. Если теперь пользователь с помощью устройства ввода предоставит новый рукописный символ, тогда наша прогнозирующая модель будет в состоянии с определенной точностью предсказать корректную букву алфавита для введенного рукописного символа. Тем не менее, наша система МО не будет способна правильно распознавать, например, цифры между 0 и 9, если они не входили в состав обучающего набора данных.

### Регрессия для прогнозирования непрерывных результатов

В предыдущем разделе мы выяснили, что задача классификации сводится к назначению образцам категориальных неупорядоченных меток. Вторым типом обучения с учителем является прогнозирование непрерывных результатов, которое также называется *регрессионным анализом*. При регрессионном анализе мы имеем несколько (*поясняющих*) переменных прогнозаторов



и переменную непрерывного отклика (*исход* или *результат*) и пытаемся отыскать между указанными переменными взаимосвязь, которая позволила бы прогнозировать результат.

Обратите внимание, что в области МО переменные прогнозаторов обычно называют “признаками” (*feature*), а на переменные откликов в большинстве случаев ссылаются как на “целевые переменные” (*target variable*). Мы будем придерживаться таких соглашений на протяжении всей книги.

Скажем, пусть нас интересует прогнозирование оценок, получаемых студентами в результате прохождения теста SAT Math (<https://ru.wikipedia.org/wiki/SAT>). Если существует какая-то взаимосвязь между временем, потраченным на подготовку к прохождению теста, и финальными оценками, то мы могли бы применить ее в качестве обучающих данных с целью обучения модели, которая будет использовать время подготовки для прогнозирования будущих оценок студентов, планирующих пройти тест.



На заметку!

### Регрессия к среднему

Термин “регрессия” был введен Фрэнсисом Гальтоном в написанной им статье “Regression towards Mediocrity in Hereditary Stature” (Регрессия к заурядности при наследовании роста), опубликованной в 1886 году. Гальтон описал биологическое явление, которое заключалось в том, что изменчивость роста среди популяции совершенно не увеличивается с течением времени.

Он обнаружил, что рост родителей не передается их детям, а взамен рост детей движется назад к среднему по популяции.

На рис. 1.4 демонстрируется концепция линейной регрессии. Для переменной признака  $x$  и целевой переменной  $y$  мы подгоняем прямую линию к этим данным, чтобы свести к минимуму расстояние — обычно среднеквадратическое — между точками данных и подогнанной линией. Теперь мы можем применять свободный член и наклон, выясненные из имеющихся данных, для прогнозирования целевой переменной новых данных.

## Решение интерактивных задач с помощью обучения с подкреплением

Третьим типом МО, который мы рассмотрим здесь, будет обучение с подкреплением (рис. 1.5). Целью такого обучения является разработка системы (*агента*), которая улучшает свои характеристики на основе взаимодействий со средой.

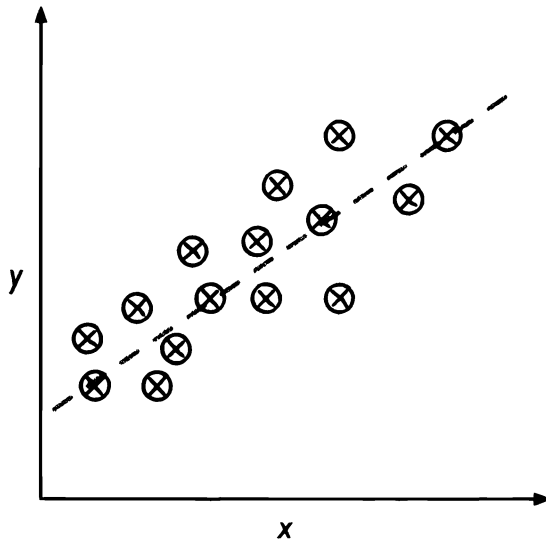


Рис. 1.4. Концепция линейной регрессии

Поскольку информация о текущем состоянии среды обычно включает так называемый *сигнал награды*, мы можем трактовать обучение с подкреплением как область, родственную обучению с учителем. Однако при обучении с подкреплением такая обратная связь не будет истинной меткой или значением, а мерой того, насколько хорошо действие было оценено функцией наград. При взаимодействии со средой агент может использовать обучение с подкреплением для выявления последовательности действий, которые доводят до максимума награду, применяя исследовательский метод проб и ошибок или совещательное планирование.



Рис. 1.5. Концепция обучения с подкреплением

Популярный пример обучения с подкреплением — шахматный движок. Агент выбирает последовательность ходов в зависимости от состояния доски (среды), а награда может быть определена как “выигрыш” или “проигрыш” в конце игры.

Различают много подтипов обучения с подкреплением. Тем не менее, общепринятая схема обучения с подкреплением заключается в том, что агент пытается довести до максимума награду через последовательность взаимодействий со средой. С каждым состоянием можно ассоциировать положительную или отрицательную награду, причем награда может быть определена как достижение общей цели, подобной выигрышу или проигрышу партии в шахматы. Так, в шахматной игре результат каждого хода можно представлять себе как отличающееся состояние среды.

Чтобы продолжить пример с шахматами, давайте будем считать, что посещение определенных позиций шахматной доски связано с состояниями, которые более вероятно приведут к выигрышу — скажем, взятие шахматной фигуры противника или угроза ферзю. Однако посещение других позиций связано с состояниями, которые более вероятно приведут к проигрышу, такими как сдача фигуры противнику на следующем ходе. Награда в шахматной игре (положительная при выигрыше и отрицательная при проигрыше) не будет предоставляться вплоть до конца партии. Вдобавок финальная награда также будет зависеть от того, как играет противник. Например, противник может принести в жертву ферзя, но все равно выиграть партию.

Обучение с подкреплением учится выбирать последовательность действий, доводящих до максимума итоговую награду, которая может быть получена либо немедленно, либо через отсроченную обратную связь.

## **Обнаружение скрытых структур с помощью обучения без учителя**

При обучении с учителем правильный ответ нам известен заранее, когда мы обучаем модель, а при обучении с подкреплением мы определяем меру награды для отдельных действий, предпринимаемых агентом. Тем не менее, при обучении без учителя мы имеем дело с непомеченными данными или данными с неизвестной структурой. Использование приемов обучения без учителя дает нам возможность исследовать структуру данных для извлечения значимой информации без управления со стороны известной целевой переменной или функции награды.

## Нахождение подгрупп с помощью кластеризации

*Кластеризация* — это исследовательская методика анализа данных, которая позволяет организовать нагромождение информации в виде содержательных подгрупп (*кластеров*), не обладая какими-то априорными знаниями о членстве в группах. Каждый кластер, появляющийся во время анализа, устанавливает группу объектов, которые обладают определенной степенью подобия, но менее похожи на объекты в других кластерах. По указанной причине кластеризацию иногда называют *классификацией без учителя*. Кластеризация является великолепным способом структурирования информации и выведения значимых взаимосвязей из данных. Например, она предоставляет специалистам по маркетингу возможность выявления групп на основе их интересов для разработки индивидуальных маркетинговых программ.

На рис. 1.6 показано, как можно применить кластеризацию с целью организации непомеченных данных в три отдельных группы, основываясь на сходстве их признаков  $x_1$  и  $x_2$ .

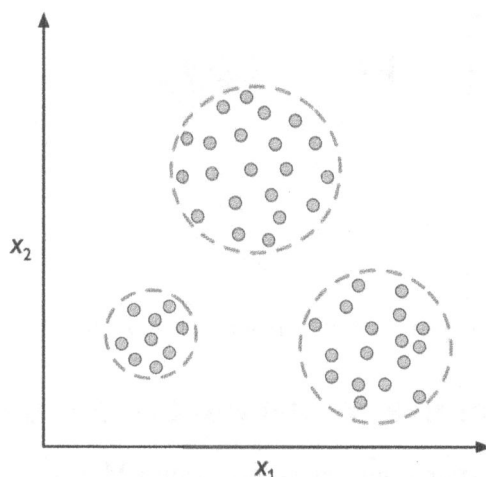


Рис. 1.6. Пример использования кластеризации

## Понижение размерности для сжатия данных

*Понижение размерности* (*dimensionality reduction*) — еще одна подобласть обучения без учителя. Зачастую мы работаем с данными высокой размерности (каждое наблюдение сопровождается большим количеством измерений), которые могут представлять проблему для ограниченного пространства хранения и вычислительной мощности алгоритмов МО. Понижение размер-

ности без учителя является распространенным подходом к предварительной обработке признаков для устранения из данных шума, который также может приводить к ухудшению прогнозирующей эффективности определенных алгоритмов, и сжатия данных в подпространство меньшей размерности с сохранением большинства существенной информации.

Иногда понижение размерности может оказаться полезным при визуализации данных; скажем, набор признаков высокой размерности может быть спроецирован в одно-, двух- или трехмерные пространства признаков с целью их визуализации через двумерные или трехмерные графики рассеяния либо гистограммы. На рис. 1.7 приведен пример применения нелинейного понижения размерности для сжатия трехмерного швейцарского рулета (3D Swiss Roll) в новое двумерное подпространство признаков.

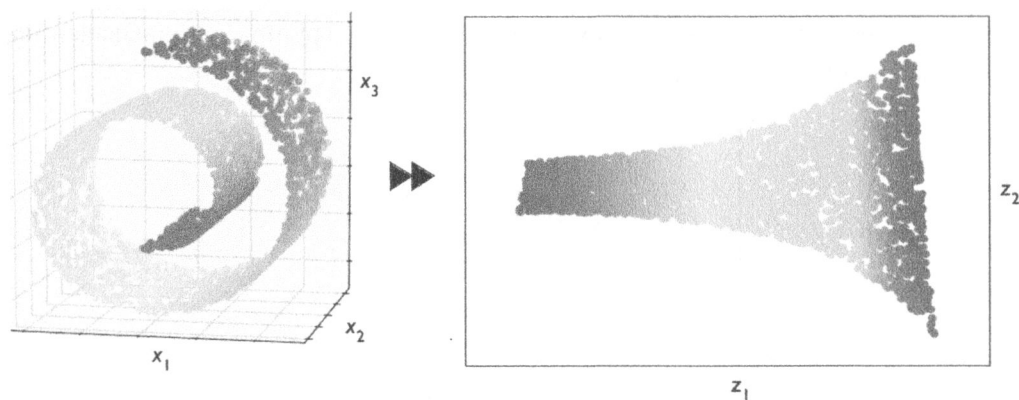


Рис. 1.7. Пример применения нелинейного понижения размерности

## Введение в основную терминологию и обозначения

После обсуждения трех обширных категорий МО — с учителем, без учителя и с подкреплением — давайте бегло взглянем на основную терминологию, которая будет использоваться повсеместно в книге. В следующем подразделе раскрываются распространенные термины, которые мы будем использовать при ссылке на различные аспекты набора данных, а также математические обозначения для более точного и эффективного описания.

Поскольку МО представляет собой обширную и междисциплинарную область, вы гарантированно столкнетесь со многими отличающимися тер-

минами, которые относятся к тем же самым концепциям, так что пусть это произойдет раньше, чем позже. Во втором подразделе собраны самые широко применяемые термины, встречающиеся в литературе по МО, и он может послужить справочником при чтении вами более несходной литературы по теме МО.

## Обозначения и соглашения, используемые в книге

На рис. 1.8 изображена таблица с выборками из набора данных об ирисах (Iris), который является классическим примером в области МО. Набор данных Iris содержит результаты измерений 150 цветков ириса трех видов — ирис щетинистый (*iris setosa*), ирис разноцветный (*iris versicolor*) и ирис виргинский (*iris virginica*). Каждый образец цветка представляет одну строку в нашем наборе данных, а измерения цветка в сантиметрах хранятся в виде столбцов, которые также называются *признаками* (*feature*) набора данных.

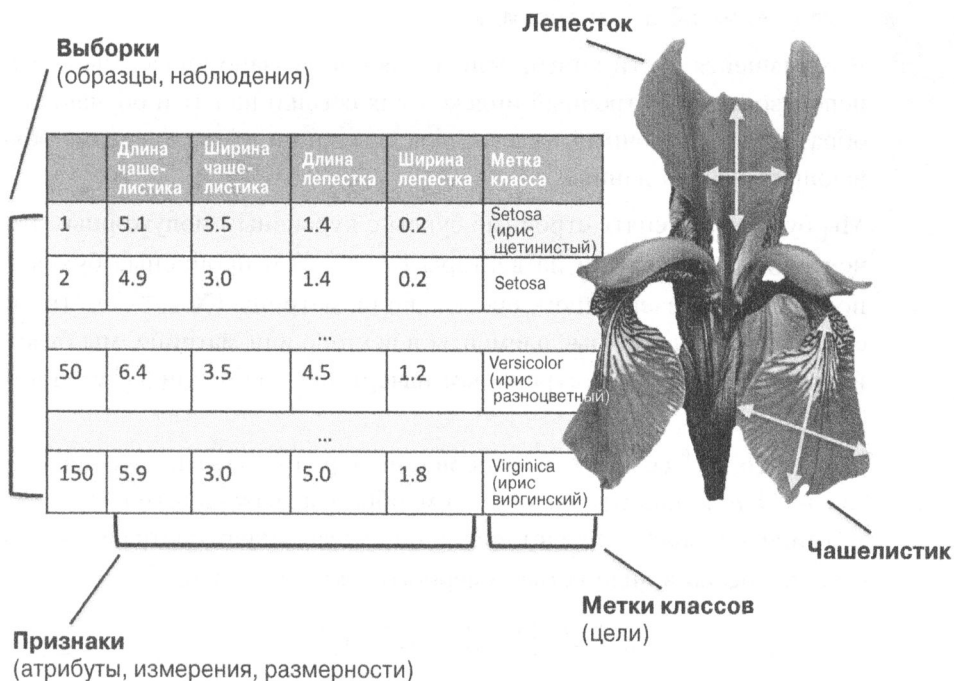


Рис. 1.8. Выборки из набора данных Iris

Чтобы сохранить систему обозначений и реализацию простыми, мы воспользуемся некоторыми основами линейной алгебры. В последующих главах для ссылки на данные мы будем применять матричную и векторную запись. Мы будем следовать общему соглашению, представляя каждый образец как отдельную строку в матрице признаков  $\mathbf{X}$ , где каждый признак хранится в обособленном столбце.

Тогда набор данных Iris, состоящий из 150 образцов и четырех признаков, можно записать в виде матрицы  $150 \times 4$ ,  $\mathbf{X} \in \mathbb{R}^{150 \times 4}$ :

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$



На  
заметку!

### Соглашение об обозначениях

В оставшейся части книги, если только не указано иначе, мы будем использовать надстрочный индекс  $i$  для ссылки на  $i$ -тый обучающий образец и подстрочный индекс  $j$  для ссылки на  $j$ -тое измерение обучающего набора данных.

Мы будем применять строчные буквы с курсивным полужирным начертанием для ссылки на векторы ( $\mathbf{x} \in \mathbb{R}^{n \times 1}$ ) и прописные буквы с полужирным начертанием для ссылки на матрицы ( $\mathbf{X} \in \mathbb{R}^{n \times m}$ ). Чтобы ссылаться на одиночные элементы в векторе или матрице, мы будем использовать буквы с курсивным начертанием ( $x^{(n)}$  или  $x_m^{(n)}$  соответственно).

Например,  $x_1^{150}$  ссылается на первое измерение образца цветка 150, т.е. на *длину чашелистика*. Таким образом, каждая строка в этой матрице признаков представляет один экземпляр цветка и может быть записана в виде четырехмерного вектора-строки,  $\mathbf{x}^{(i)} \in \mathbb{R}^{1 \times 4}$ :

$$\mathbf{x}^{(i)} = [x_1^{(i)} \quad x_2^{(i)} \quad x_3^{(i)} \quad x_4^{(i)}]$$

Каждое измерение внутри признаков представляет собой 150-мерный вектор-столбец,  $\mathbf{x}^{(i)} \in \mathbb{R}^{150 \times 4}$ , например:

$$\mathbf{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix}$$

Целевые переменные (здесь метки классов) будут храниться аналогично — в 150-мерном векторе-столбце:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} \left( y \in \{\text{Setosa, Versicolor, Virginica}\} \right)$$

## Терминология, связанная с машинным обучением

Машинное обучение является обширной и также междисциплинарной областью, поскольку оно объединяет многих ученых из других областей исследования. Как это часто происходит, многие термины и концепции были открыты заново или переопределены и могут быть уже знакомыми вам, но под другими названиями. Ради удобства в следующем списке вы найдете подборку часто применяемых терминов и их синонимов, которую вы можете счесть удобной при чтении этой книги и в целом литературы по МО.

- *Обучающий образец (training sample)*. Строка в таблице, которая представляет набор данных и синонимична наблюдению, записи, экземпляру или примеру (в большинстве контекстов под выборкой понимается коллекция обучающих образцов).
- *Обучение (training)*. Подгонка моделей, которая в случае параметрических моделей подобна оценке параметров.
- *Признак (feature)*, сокращенно *x*. Столбец в таблице данных или матрице (структуры) данных. Синонимичен прогнозатору, переменной, входу, атрибуту или коварианту.
- *Цель (target)*, сокращенно *y*. Синонимична исходу, выходу, переменной отклика, зависимой переменной, метке (классу) или истинной метке.
- *Функция потерь (loss function)*. Часто используется синонимично *функции издержек (cost function)*. Иногда функция потерь также на-



зывается *функцией ошибки (error function)*. Временами в литературе термин “потеря” относят к утрате, измеренной для одиночной точки данных, а издержки — это измерение, которое подсчитывает потерю (среднюю или суммарную) по целому набору данных.

## Дорожная карта для построения систем машинного обучения

В предшествующих разделах мы обсуждали базовые концепции МО и три типа обучения. В настоящем разделе мы рассмотрим другие важные части системы МО, сопровождающие алгоритм обучения.

Диаграмма на рис. 1.9 демонстрирует типичный рабочий поток для применения МО в прогнозирующем моделировании, который мы обсудим в последующих подразделах.

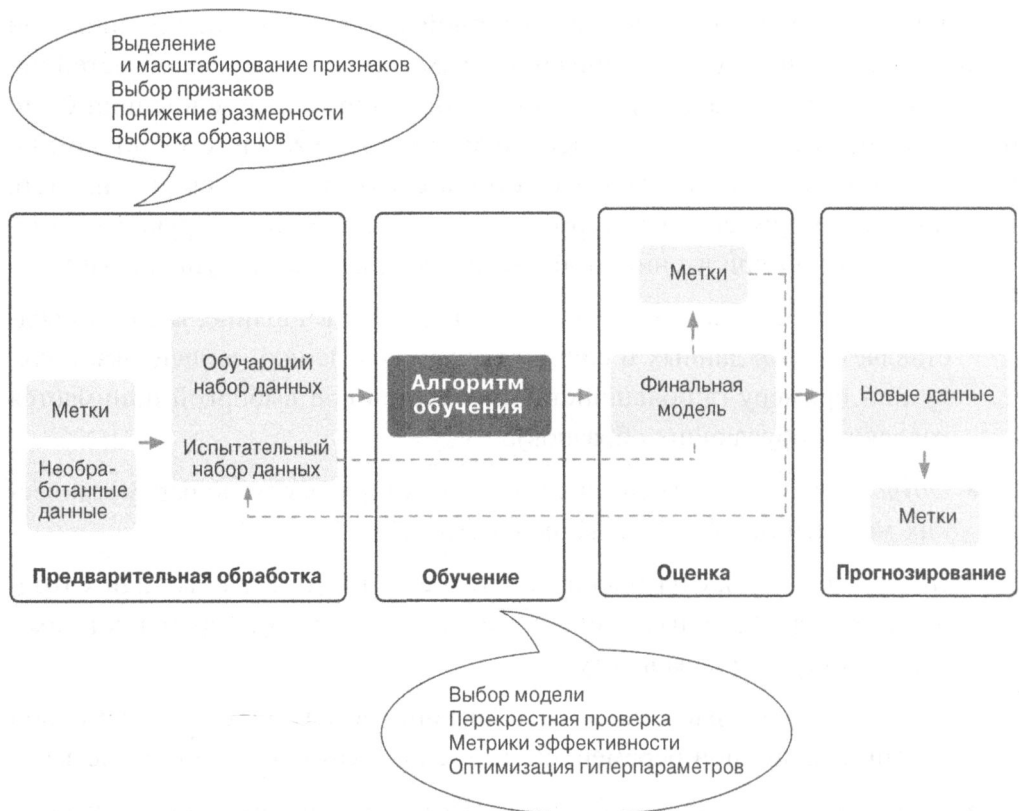


Рис. 1.9. Рабочий поток для применения МО в прогнозирующем моделировании

## **Предварительная обработка — приведение данных в приемлемую форму**

Давайте начнем обсуждение дорожной карты для построения систем МО. Необработанные данные редко поступают в форме, которая нужна для обеспечения оптимальной эффективности алгоритма обучения. Таким образом, предварительная обработка данных является одним из наиболее важных шагов в любом приложении МО.

Возьмем в качестве примера набор данных Iris из предыдущего раздела. Необработанными данными можно считать последовательность изображений цветков, из которых мы хотим выделить значащие признаки. Полезными признаками могли бы быть цвет, оттенок и яркость цветков или высота цветков вместе с длиной и шириной чашелистиков и лепестков.

Многие алгоритмы МО вдобавок требуют, чтобы выбранные признаки имели одинаковый масштаб для обеспечения оптимальной эффективности, что часто достигается преобразованием признаков в диапазон  $[0, 1]$  либо их приведением к стандартному нормальному распределению с нулевым средним и единичной дисперсией, как будет показано в более поздних главах.

Некоторые выбранные признаки могут оказаться сильно связанными и потому в определенной степени избыточными. В таких случаях полезно использовать методики понижения размерности для сжатия признаков в подпространство меньшей размерности. Понижение размерности пространства признаков дает еще одно преимущество: они занимают меньший объем в хранилище и алгоритм обучения может работать гораздо быстрее. В ряде ситуаций понижение размерности способно также улучшить эффективность прогнозирования модели, когда набор данных содержит большое число не относящихся к делу признаков (или шум), т.е. если набор данных имеет низкое соотношение сигнал/шум.

Чтобы установить, что алгоритм МО не только успешно функционирует на обучающем наборе, но также хорошо обобщается на новые данные, набор данных понадобится случайным образом разделить на обучающий и испытательный наборы. Обучающий набор применяется для обучения и оптимизации модели МО, тогда как испытательный набор хранится до самого конца и предназначен для оценки финальной модели.

## Обучение и выбор прогнозирующей модели

Как будет видно в последующих главах, для решения разнообразных задач было разработано много отличающихся друг от друга алгоритмов МО. Важный момент, который можно вынести из известных *теорем об отсутствии бесплатных завтраков* (*No free lunch theorems*) Дэвида Уолперта, касается того, что проводить обучение “бесплатно” нельзя (“The Lack of A Priori Distinctions Between Learning Algorithms” (Отсутствие априорных различий между обучающими алгоритмами), Д. Вольперт (1996 г.); “No free lunch theorems for optimization” (Теоремы об отсутствии бесплатных завтраков для оптимизации), Д. Вольперт и У. Макриди (1997 г.)). Мы можем увязать данную концепцию с популярным высказыванием: “Я полагаю, что заманчиво трактовать все вокруг как гвозди, если единственный инструмент, что есть у вас — это молоток” (Абрахам Маслоу, 1966 год) или, перефразируя, “Когда у тебя есть только молоток, то все похоже на гвоздь”. Например, каждый алгоритм классификации имеет присущие ему смещения, и никакая отдельно взятая классификационная модель не получает преимуществ перед другими, если только не сделаны некоторые допущения, касающиеся задачи. Следовательно, на практике важно сравнить хотя бы несколько разных алгоритмов, чтобы обучить и выбрать лучше всех работающую модель. Но прежде чем появится возможность сравнивать разные модели, мы должны принять решение относительно метрики для измерения эффективности. Одной из распространенных метрик является точность классификации, которая определяется как доля корректно классифицированных образцов.

Возникает вполне законный вопрос: *как мы узнаем, какая модель хорошо работает на финальном испытательном наборе и реальных данных, если мы не используем этот испытательный набор для выбора модели, а храним его для оценки окончательной модели?* Для решения проблемы, заложенной в сформулированном вопросе, можно применять различные приемы перекрестной проверки. При перекрестной проверке мы проводим дальнейшее разделение набора данных на обучающий и проверочный поднаборы, чтобы оценить эффективность обобщения модели. Наконец, мы не можем надеяться, что стандартные параметры разнообразных алгоритмов обучения, предлагаемых программными библиотеками, окажутся оптимальными для нашей специфической задачи. Таким образом, в последующих главах мы будем часто прибегать к методикам оптимизации гиперпараметров, которые помогут точно регулировать эффективность модели.

Интуитивно гиперпараметры можно считать параметрами, которые не узнаются из данных, а представляют собой своего рода ручки управления моделью, позволяющие улучшать ее эффективность. Сказанное станет гораздо яснее позже в книге, когда начнут демонстрироваться фактические примеры.

## **Оценка моделей и прогнозирование на не встречавшихся ранее образцах данных**

После выбора модели, подогнанной к обучающему набору данных, мы можем использовать испытательный набор данных для оценки, насколько хорошо модель работает на этих не встречавшихся ранее данных, чтобы приблизительно подсчитать ошибку обобщения. Если эффективность модели нас устраивает, тогда мы можем применять ее для прогнозирования на новых будущих данных. Важно отметить, что параметры для ранее упомянутых процедур, таких как масштабирование признаков и понижение размерности, получаются исключительно из обучающего набора данных. Те же самые параметры позже снова применяются для трансформации испытательного набора данных, а также любых новых образцов данных — иначе эффективность, измеренная на испытательных данных, может оказаться слишком оптимистичной.

## **Использование Python для машинного обучения**

Python является одним из наиболее популярных языков программирования, применяемых в науке о данных, и благодаря его очень активному сообществу разработчиков ПО с открытым кодом было создано большое количество полезных библиотек для научных расчетов и МО.

Хотя производительность интерпретируемых языков вроде Python при выполнении задач, требующих большого объема вычислений, хуже производительности низкоуровневых языков программирования, были разработаны библиотеки расширений, такие как NumPy и SciPy, которые задействуют низкоуровневые реализации на Fortran и C для быстрых векторизованных операций над многомерными массивами.

При решении задач программирования для МО мы будем обращаться главным образом к библиотеке scikit-learn, которая в настоящее время считается одной из самых популярных и доступных библиотек МО с открытым

кодом. Когда в последующих главах мы сосредоточим внимание на подобласти МО, называемой глубоким обучением, то будем использовать последнюю версию библиотеки TensorFlow, которая приспособлена для очень эффективного обучения моделей на основе глубоких нейронных сетей за счет применения графических плат.

## Установка Python и необходимых пакетов

Язык Python доступен для трех основных операционных систем, Microsoft Windows, macOS и Linux, а программу установки и документацию можно загрузить из официального веб-сайта Python: <https://www.python.org>.

Код в книге рассчитан на Python 3.7 или последующие версии, но рекомендуется использовать самую последнюю версию линейки Python 3, доступную на момент чтения книги. Некоторый код может быть совместимым с Python 2.7, но поскольку официальная поддержка Python 2.7 закончилась в 2019 году и большинство библиотек с открытым кодом перестали поддерживать Python 2.7 (<https://python3statement.org>), мы настоятельно советуем применять Python 3.7 или более новую версию.

Дополнительные пакеты, которые будут использоваться повсеместно в книге, можно установить с помощью программы установки `pip`, которая входит в состав стандартной библиотеки Python, начиная с версии Python 3.3. Больше сведений о `pip` можно найти по ссылке <https://docs.python.org/3/installing/index.html>.

После успешной установки Python можно запустить в терминальном окне `pip` и установить дополнительные пакеты Python:

```
pip install ИмяПакета
```

Уже установленные пакеты можно обновить, указав флаг `--upgrade`:

```
pip install ИмяПакета --upgrade
```

## Использование дистрибутива Anaconda и диспетчера пакетов

Крайне рекомендуемым альтернативным дистрибутивом Python для научных вычислений является Anaconda от Continuum Analytics. Дистрибутив Anaconda бесплатен, в том числе для коммерческого применения, и готов к работе в масштабах предприятия. В него упакованы все важные пакеты

Python, предназначенные для науки о данных, математики и инженерного искусства. Он дружелюбен к пользователю и поддерживает множество платформ. Программа установки Anaconda доступна для загрузки по ссылке <https://docs.anaconda.com/anaconda/install/>, а краткое руководство по Anaconda — по ссылке <https://docs.anaconda.com/anaconda/user-guide/getting-started/>.

После успешной установки Anaconda можно установить новые пакеты Python с помощью такой команды:

```
conda install ИмяПакета
```

Обновить существующие пакеты можно посредством следующей команды:

```
conda update ИмяПакета
```

## Пакеты для научных расчетов, науки о данных и машинного обучения

Повсюду в книге для хранения и манипулирования данными мы будем использовать главным образом многомерные массивы NumPy. Временами мы будем применять pandas — библиотеку, построенную поверх NumPy, которая предоставляет дополнительные высокоуровневые инструменты манипулирования данными, еще более облегчающие работу с табличными данными. Чтобы расширить наш опыт познания и визуализировать количественные данные, что зачастую исключительно полезно для интуитивного восприятия их смысла, мы будем использовать полностью настраиваемую библиотеку Matplotlib.

Ниже приведен список с номерами версий основных пакетов Python, которые применялись при написании этой книги. Для обеспечения корректной работы примеров кода удостоверьтесь в том, что номера версий установленных у вас пакетов совпадают или превышают указанные далее номера:

- NumPy 1.17.4
- SciPy 1.3.1
- scikit-learn 0.22.1
- Matplotlib 3.1.0
- pandas 0.25.3

## Резюме

В главе были проведены высокоуровневые исследования МО, а также представлена общая картина и главные концепции того, что более подробно будет рассматриваться в последующих главах. Мы выяснили, что обучение с учителем состоит из двух важных подобластей: классификация и регрессия. В то время как классификационные модели позволяют нам распределять объекты по известным классам, регрессионный анализ можно использовать для прогнозирования непрерывных результатов целевых переменных. Обучение без учителя не только предлагает удобные приемы обнаружения структур в непомеченных данных, но также может быть полезным для сжатия данных на шагах предварительной обработки признаков.

Мы кратко прошлись по типовой дорожной карте, регламентирующей применение МО к решаемым задачам, которая будет использоваться в качестве основы для более глубоких обсуждений и практических примеров в последующих главах. В заключение мы настроили среду Python, а также установили и обновили требующиеся пакеты, чтобы подготовиться к наблюдению за МО в действии.

Позже в книге вдобавок к самому МО мы представим разнообразные приемы предварительной обработки набора данных, которые помогут добиться наилучшей эффективности от различных алгоритмов МО. Наряду с тем, что повсюду в книге мы будем довольно широко раскрывать алгоритмы классификации, мы также исследуем приемы регрессионного анализа и кластеризации.

Нам предстоит захватывающее путешествие, которое раскроет многие мощные технологии в обширной области МО. Однако приближаться к МО мы будем шаг за шагом с опорой на знания, постепенно накапливаемые после каждой главы книги. В следующей главе мы начнем это путешествие с реализации одного из самых ранних алгоритмов МО для классификации, что обеспечит подготовку к чтению главы 3, где рассматриваются более сложные алгоритмы МО, использующие библиотеку `scikit-learn` с открытым кодом.

# ОБУЧЕНИЕ ПРОСТЫХ АЛГОРИТМОВ МО для КЛАССИФИКАЦИИ

**В** текущей главе мы будем использовать два варианта из группы первых алгоритмически описанных методов МО, предназначенных для классификации — перцептрон и адаптивные линейные нейроны. Мы начнем с пошаговой реализации перцептрона на Python и его обучения для классификации видов цветков ириса в наборе данных Iris. Это поможет понять концепцию алгоритмов МО для классификации и способы их эффективной реализации на Python.

Последующее обсуждение базовых понятий оптимизации с применением адаптивных линейных нейронов сформирует основу для использования более мощных классификаторов через библиотеку `scikit-learn` в главе 3.

В главе будут раскрыты следующие темы:

- обеспечение интуитивного понимания алгоритмов МО;
- применение `pandas`, `NumPy` и `Matplotlib` для чтения, обработки и визуализации данных;
- реализация алгоритмов линейной классификации на Python.



## Искусственные нейроны — беглое знакомство с ранней историей машинного обучения

Прежде чем перейти к более подробным обсуждениям персептрона и связанных с ним алгоритмов, давайте проведем краткий экскурс в раннюю историю МО. Пытаясь осмыслить, как работает биологический мозг, с целью проектирования искусственного интеллекта Уоррен Мак-Каллок и Уолтер Питтс в 1943 году написали статью “Логическое исчисление идей, присущих нервной деятельности” (“A Logical Calculus of Ideas Immanent in Nervous Activity”, *Bulletin of Mathematical Biophysics*, 5(4): с. 115–133). В статье была представлена первая концепция упрощенной клетки головного мозга, которая известна под названием *нейрон Мак-Каллока-Питтса* (McCulloch-Pitts — MCP). Биологические нейроны — это взаимосвязанные клетки головного мозга, участвующие в обработке и передаче химических и электрических сигналов, как иллюстрируется на рис. 2.1.

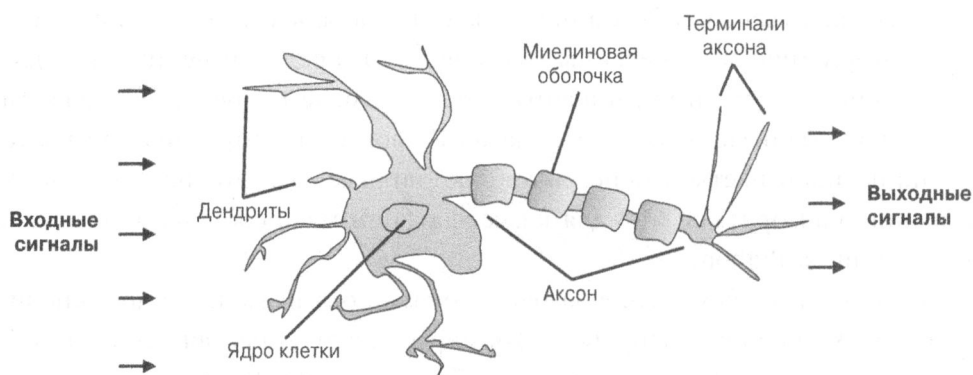


Рис. 2.1. Упрощенная схема биологического нейрона

Мак-Каллок и Питтс описали такую нервную клетку в виде простого логического вентиля с двоичными выходами; множество сигналов достигают дендритов, затем они встраиваются в тело клетки, и если накопленный сигнал превышает определенный порог, тогда генерируется выходной сигнал, который будет передан аксоном.

Через несколько лет в 1957 году Фрэнк Розенблатт опубликовал статью “Персептрон: воспринимающий и распознающий автомат” (“The Perceptron: A Perceiving and Recognizing Automaton”, Cornell Aeronautical Laboratory), где изложил первую идею относительно правила обучения персептрона,

основанного на модели нейрона Мак-Каллока-Питтса. Вместе с правилом персептрона Розенблатт предложил алгоритм, автоматически узнающий оптимальные весовые коэффициенты, на которые затем умножались входные признаки для принятия решения о том, активируется нейрон (передает сигнал) или нет. В контексте обучения с учителем и классификации алгоритм такого рода может использоваться для прогнозирования принадлежности новой точки данных к одному или к другому классу.

## Формальное определение искусственного нейрона

Более формально идею, лежащую в основе *искусственных нейронов*, можно поместить в контекст задачи двоичной классификации, в рамках которой для простоты мы ссылаемся на два класса как на 1 (положительный класс) и  $-1$  (отрицательный класс). Затем мы можем определить функцию решения  $\phi(z)$ , которая принимает линейную комбинацию некоторых входных значений  $x$  и соответствующий весовой вектор  $w$ , где  $z$  — так называемый *общий вход* (*net input*),  $z = w_1x_1 + w_2x_2 + \dots + w_mx_m$ :

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Если общий вход отдельного образца  $x^{(i)}$  превышает определенный порог  $\theta$ , тогда мы прогнозируем класс 1, а в противном случае — класс  $-1$ . В алгоритме персептрона функция решения  $\phi(\cdot)$  — это разновидность *единичной ступенчатой функции* (*unit step function*):

$$\phi(z) = \begin{cases} 1, & \text{если } z \geq \theta \\ -1 & \text{в противном случае} \end{cases}$$

Для простоты мы можем перенести порог  $\theta$  в левую часть равенства и определить нулевой вес как  $w_0 = -\theta$  и  $x_0 = 1$ , так что  $z$  запишется в более сжатой форме:

$$z = w_1x_1 + w_2x_2 + \dots + w_mx_m = w^T x$$

И:

$$\phi(z) = \begin{cases} 1, & \text{если } z \geq 0 \\ -1 & \text{в противном случае} \end{cases}$$

В литературе по МО отрицательный порог или вес,  $w_0 = -\theta$ , обычно называют членом смещения (*bias unit*).



На  
заметку!

### Основы линейной алгебры:

#### скалярное произведение и транспонирование матриц

В последующих разделах мы будем часто применять базовые обозначения из линейной алгебры. Скажем, мы сократим сумму произведений значений в  $x$  и  $w$ , используя скалярное произведение векторов, при этом  $^T$  означает *транспонирование*, которое представляет собой операцию, трансформирующую вектор-столбец в вектор-строку и наоборот:

$$z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{j=0}^m x_j w_j = w^T x$$

Вот пример:

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$$

Кроме того, операцию транспонирования можно также применять к матрице, чтобы поменять местами столбцы и строки относительно диагонали, например:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Обратите внимание, что операция транспонирования определена строго только для матриц; тем не менее, в контексте МО, когда мы используем термин “вектор”, то имеем в виду матрицы  $n \times 1$  и  $1 \times m$ .

В книге будут задействованы лишь самые элементарные понятия из линейной алгебры; однако если вам нужен краткий справочник по линейной алгебре, тогда можете загрузить великолепную работу Зико Колтера по ссылке [http://www.cs.cmu.edu/~zkolter/course/linalg/linalg\\_notes.pdf](http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf).

На рис. 2.2 слева видно, как функция решения персептрона сжимает общий вход  $z = w^T x$  в двоичный выход ( $-1$  или  $1$ ), а справа — каким образом это может использоваться для различения *двух линейно сепарабельных классов*.

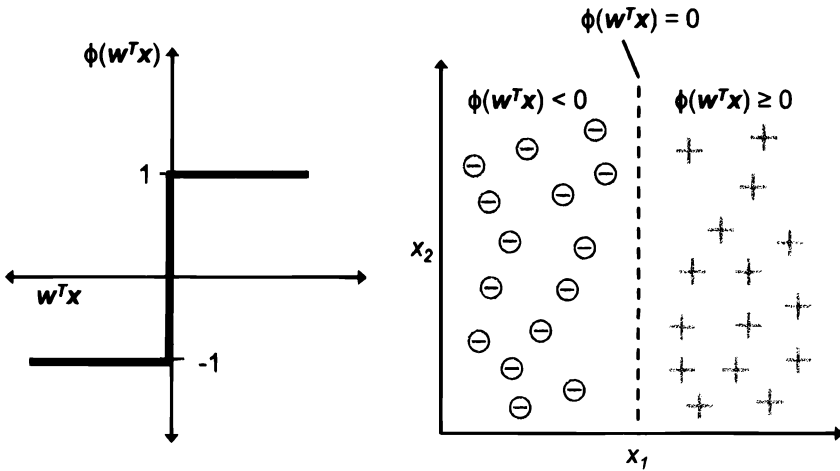


Рис. 2.2. Работа функции решения персептрона

## Правило обучения персептрона

Вся идея, лежащая в основе нейрона Мак-Каллока-Питтса и модели *порогового персептрона Розенблатта*, заключается в том, чтобы с помощью редукционистского подхода симитировать работу одиночного нейрона из головного мозга: он либо *активируется*, либо нет. Таким образом, начальное правило персептрона Розенблатта выглядит довольно простым и алгоритм персептрона может быть сведен к следующим шагам.

1. Инициализировать веса нулями или небольшими случайными числами.
2. Для каждого обучающего образца  $x^{(i)}$ :
  - а) вычислить выходное значение  $\hat{y}$ ;
  - б) обновить веса.

Здесь выходным значением является метка класса, прогнозируемая определенной ранее единичной ступенчатой функцией, а одновременное обновление всех весов  $w_j$  в весовом векторе  $w$  может быть формально записано так:

$$w_j := w_j + \Delta w_j$$

Обновляющее значение для  $w_j$  (либо изменение в  $w_j$ ), на которое мы ссылаемся как на  $\Delta w_j$ , вычисляется правилом обучения персептрона следующим образом:

$$\Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

Здесь  $\eta$  — скорость обучения (обычно константа между 0.0 и 1.0),  $y^{(i)}$  — настоящая метка класса  $i$ -того обучающего образца, а  $\hat{y}^{(i)}$  — спрогнозированная метка класса. Важно отметить, что все веса в весовом векторе обновляются одновременно, т.е. мы не вычисляем повторно спрогнозированную метку  $\hat{y}^{(i)}$  до того, как всех веса обновятся через соответствующие обновляющие значения  $\Delta w_j$ . Конкретно для двумерного набора данных мы могли бы записать обновление так:

$$\Delta w_0 = \eta (y^{(i)} - \text{выход}^{(i)})$$

$$\Delta w_1 = \eta (y^{(i)} - \text{выход}^{(i)}) x_1^{(i)}$$

$$\Delta w_2 = \eta (y^{(i)} - \text{выход}^{(i)}) x_2^{(i)}$$

Прежде чем заняться реализацией правила персептрона на Python, давайте проведем простой мысленный эксперимент с целью иллюстрации того, настолько замечательно простым в действительности является это правило обучения. В двух сценариях, когда персептрон корректно прогнозирует метку класса, веса остаются неизменными, т.к. обновляющие значения равны 0:

$$(1) \quad y^{(i)} = -1 \quad \hat{y}^{(i)} = -1 \quad \Delta w_j = \eta (-1 - (-1)) x_j^{(i)} = 0$$

$$(2) \quad y^{(i)} = 1 \quad \hat{y}^{(i)} = 1 \quad \Delta w_j = \eta (1 - 1) x_j^{(i)} = 0$$

Тем не менее, в случае неправильного прогноза веса продвигаются в направлении положительного или отрицательного целевого класса:

$$(3) \quad y^{(i)} = 1 \quad \hat{y}^{(i)} = -1 \quad \Delta w_j = \eta (1 - (-1)) x_j^{(i)} = \eta (2) x_j^{(i)}$$

$$(4) \quad y^{(i)} = -1 \quad \hat{y}^{(i)} = 1 \quad \Delta w_j = \eta (-1 - 1) x_j^{(i)} = \eta (-2) x_j^{(i)}$$

Чтобы лучше понять мультипликативный множитель  $x_j^{(i)}$ , рассмотрим еще один простой пример, где:

$$\hat{y}^{(i)} = -1, \quad y^{(i)} = +1, \quad \eta = 1$$

Допустим, что  $x_j^{(i)} = 0.5$ , и мы неправильно классифицировали данный образец как  $-1$ . В таком случае мы увеличили бы соответствующий вес на 1,

чтобы общий вход  $x_j^{(i)} \times w_j$  стал более положительным в следующий раз, когда встретится этот образец, и с большей вероятностью превысил порог единичной ступенчатой функции, обеспечив классификацию образца как +1:

$$\Delta w_j = (1 - (-1))0.5 = (2) 0.5 = 1$$

Обновление весов пропорционально значению  $x_j^{(i)}$ . Например, при наличии еще одного образца  $x_j^{(i)} = 2$ , который некорректно классифицируется как -1, мы продвинем границу решения еще дальше, чтобы в следующий раз данный образец классифицировался правильно:

$$\Delta w_j = (1^{(i)} - (-1)^{(i)})2^{(i)} = (2)2^{(i)} = 4$$

Важно отметить, что сходимость персептрона гарантируется лишь тогда, когда два класса линейно сепарабельны и скорость обучения достаточно мала (заинтересованные читатели могут ознакомиться с математическим доказательством в конспекте лекций: [https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03\\_perceptron\\_slides.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03_perceptron_slides.pdf)). Если два класса не могут быть разделены линейной границей решения, то можно установить максимальное количество проходов через обучающий набор данных (*epoch*) и/или пороговое число допустимых неправильных классификаций — в противном случае персептрон никогда не прекратит обновление весов (рис. 2.3).



Рис. 2.3. Классы, которые сепарабельны и не сепарабельны линейно



### Загрузка кода примеров

Исходный код всех примеров и наборы данных доступны для загрузки по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition>.

До того, как приступить к реализации в следующем разделе, давайте подведем итоги только что изученного с помощью простой диаграммы, которая иллюстрирует общую концепцию персептрона (рис. 2.4).

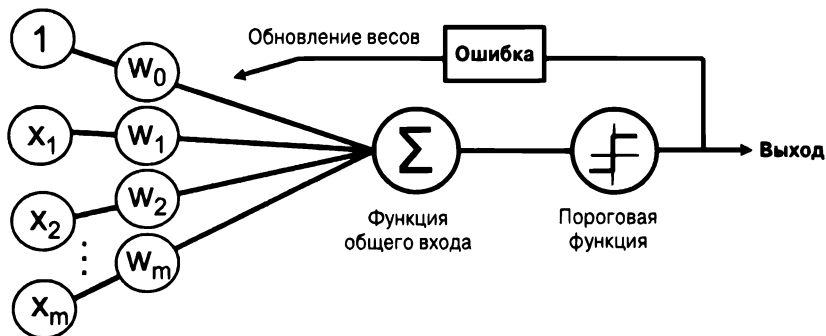


Рис. 2.4. Общая концепция персептрона

На рис. 2.4 видно, что персептрон получает входы образца  $x$  и объединяет их с весами  $w$ , чтобы вычислить общий вход. Затем общий вход передается пороговой функции, которая генерирует двоичный вывод  $-1$  или  $+1$  — спрогнозированную метку класса для образца. На стадии обучения этот вывод используется для вычисления ошибки прогнозирования и обновления весов.

## Реализация алгоритма обучения персептрона на Python

В предыдущем разделе вы узнали, как работает правило персептрона Розенблатта; теперь мы реализуем его на Python и применим к набору данных Iris, представленному в главе 1.

### Объектно-ориентированный API-интерфейс персептрона

Мы примем объектно-ориентированный подход, чтобы определить интерфейс персептрона в виде класса Python, позволяющий инициализировать новые объекты `Perceptron`, которые можно обучать на данных через метод `fit` и делать прогнозы посредством отдельного метода `predict`. В качестве соглашения мы добавляем символ подчеркивания (`_`) к именам атрибутов, которые не создаются во время инициализации объекта, но это будет делаться путем вызова других методов объекта, например, `self.w_`.



На  
заметку!

## Дополнительные ресурсы для стека научных расчетов на языке Python

Если вы пока не знакомы с библиотеками Python для научных расчетов или хотите освежить свои знания, тогда просмотрите следующие ресурсы:

- o NumPy: [https://sebastianraschka.com/pdf/books/dlb/appendix\\_f\\_numpy-intro.pdf](https://sebastianraschka.com/pdf/books/dlb/appendix_f_numpy-intro.pdf)
- o pandas: <https://pandas.pydata.org/pandas-docs/stable/10min.html>
- o Matplotlib: <https://matplotlib.org/tutorials/introductory/usage.html>

Ниже приведена реализация персептрона на Python.

```
import numpy as np

class Perceptron(object):
    """Классификатор на основе персептрона.

    Параметры
    -----
    eta : float
        Скорость обучения (между 0.0 и 1.0)
    n_iter : int
        Проходы по обучающему набору данных.
    random_state : int
        Начальное значение генератора случайных чисел
        для инициализации случайными весами.

    Атрибуты
    -----
    w_ : одномерный массив
        Веса после подгонки.
    errors_ : список
        Количество неправильных классификаций (обновлений) в каждой эпохе.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```



```
def fit(self, X, y):
    """Подгоняет к обучающим данным.

    Параметры
    -----
    X : {подобен массиву}, форма = [n_examples, n_features]
        Обучающие векторы, где n_examples - количество образцов
        и n_features - количество признаков.
    y : подобен массиву, форма = [n_examples]
        Целевые значения.

    Возвращает
    -----
    self : object
    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01,
                          size=1 + X.shape[1])
    self.errors_ = []
    for _ in range(self.n_iter):
        errors = 0
        for xi, target in zip(X, y):
            update = self.eta * (target - self.predict(xi))
            self.w_[1:] += update * xi
            self.w_[0] += update
            errors += int(update != 0.0)
        self.errors_.append(errors)
    return self

def net_input(self, X):
    """Вычисляет общий вход"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    """Возвращает метку класса после единичного шага"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Используя такую реализацию персептрона, мы можем инициализировать новые объекты `Perceptron` с заданной скоростью обучения `eta` и количеством эпох `n_iter` (проходов по обучающему набору).

Посредством метода `fit` мы инициализируем веса в `self.w_` вектором  $\mathbb{R}^{m+1}$ , где  $m$  обозначает количество измерений (признаков) в наборе данных,

к которому добавляется 1 для первого элемента в этом векторе. Не забывайте, что первый элемент вектора, `self.w_[0]`, представляет так называемый член смещения, который упоминался ранее.

Также имейте в виду, что данный вектор содержит небольшие случайные числа, взятые из нормального распределения со стандартным отклонением 0.01 с помощью метода `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, где `rgen` — генератор случайных чисел NumPy, который снабжается указанным пользователем начальным значением, позволяя при желании воспроизводить предыдущие результаты.

Важно помнить о том, что мы не инициализируем веса нулями, потому что скорость обучения  $\eta$  (eta) влияет на результат классификации, только если веса инициализированы ненулевыми значениями. Когда все веса инициализированы нулями, параметр скорости обучения `eta` влияет только на масштаб весового вектора, но не на его направление. Если вы знакомы с тригонометрией, тогда подумайте о векторе `v1 = [1 2 3]` с точно нулевым углом между ним и вектором `v2 = 0.5 × v1`, как демонстрируется в следующем фрагменте кода:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...           np.linalg.norm(v2)))
0.0
```

Здесь `np.arccos` — тригонометрический арккосинус, а `np.linalg.norm` — функция, вычисляющая длину вектора (наше решение извлекать случайные числа из нормального распределения, а не, например, равномерного, и выбор стандартного отклонения 0.01 было чисто произвольным; не забывайте, что нас всего лишь интересуют небольшие случайные значения, чтобы не иметь дела с особенностями векторов, содержащих все нули, как обсуждалось ранее.)



На заметку!

Индексация одномерных массивов в NumPy работает подобно индексации списков Python, использующих систему обозначений в виде квадратных скобок (`[]`). Для двумерных массивов первый индексатор ссылается на номер строки, а второй — на номер столбца. Например, для выбора элемента, находящегося в третьей строке и четвертом столбце двумерного массива `X`, мы будем применять `X[2, 3]`.

После инициализации весов метод `fit` проходит в цикле по всем индивидуальным образцам в обучающем наборе и обновляет веса согласно правилу обучения персептрона, которое мы обсуждали в предыдущем разделе.

Метки классов прогнозируются методом `predict`, который вызывается в методе `fit` на стадии обучения, чтобы получить метку класса для обновления весов, но `predict` может также использоваться для прогнозирования меток классов новых данных после подгонки модели. Кроме того, в списке `self.errors_` мы накапливаем число неправильных классификаций в течение каждой эпохи, чтобы позже можно было проанализировать, насколько хорошо работал наш персептрон во время обучения. Функция `np.dot`, применяемая в методе `net_input`, просто вычисляет скалярное произведение векторов  $w^T x$ .



На заметку!

Вместо использования NumPy для вычисления скалярного произведения векторов с двумя массивами `a` и `b` через `a.dot(b)` или `np.dot(a, b)` мы могли бы выполнить вычисление на чистом Python посредством `sum([i*j for i, j in zip(a, b)])`. Однако преимущество библиотеки NumPy перед классическими структурами циклов `for` языка Python заключается в том, что ее арифметические операции векторизованы. *Векторизация* означает, что элементарная арифметическая операция автоматически применяется ко всем элементам в массиве. За счет формулирования наших арифметических операций как последовательности инструкций на массиве вместо того, чтобы выполнять набор операций с каждым элементом, мы можем эффективнее использовать современные процессорные архитектуры с поддержкой *SIMD* (*Single Instruction, Multiple Data* — однопоточный поток команд, множественный поток данных). Более того, в NumPy применяются высокооптимизированные библиотеки линейной алгебры, такие как *BLAS* (*Basic Linear Algebra Subprograms* — базовые подпрограммы линейной алгебры) и *LAPACK* (*Linear Algebra Package* — пакет линейной алгебры), написанные на C или Fortran. Наконец, NumPy также делает возможным написание кода в более компактной и понятной манере с использованием основ линейной алгебры, таких как скалярные произведения векторов и матриц.

## Обучение модели персептрона на наборе данных Iris

При тестировании нашей реализации персептрона мы сузим последующий анализ и примеры в главе до двух переменных признаков (измерений). Хотя правило персептрона не ограничивается двумя измерениями, учет

только двух признаков, длины чашелистика (sepal length) и длины лепестка (petal length), позволит визуализировать области решений обученной модели на графике рассеяния в учебных целях.

К тому же, исходя из практических соображений, мы будем принимать во внимание только два вида цветков, Setosa и Versicolor, из набора данных Iris — вспомните о том, что перцептрон является двоичным классификатором. Тем не менее, алгоритм перцептрона можно расширить для выполнения многоклассовой классификации, скажем, по методике “один против всех” (*one-versus-all* — OvA).



На  
заметку!

### Методика OvA для многоклассовой классификации

Методика “один против всех” (OvA), также иногда называемая “один против остальных” (*one-versus-rest* — OvR), позволяет расширять любой двоичный классификатор с целью решения многоклассовых задач. С применением OvA мы можем обучать по одному классификатору на класс, при этом специфический класс трактуется как положительный, а образцы из всех остальных классов считаются принадлежащими отрицательным классам. Если бы мы классифицировали новый непомеченный образец данных, то использовали бы  $n$  классификаторов, где  $n$  — количество меток классов, и назначали бы отдельному образцу, подлежащему классификации, метку класса с наивысшей степенью доверия. В случае перцептрона мы применяли бы методику OvA для выбора метки класса, которая ассоциирована с наибольшим значением общего входа.

Для начала мы с помощью библиотеки pandas загрузим в объект DataFrame набор данных Iris прямо из *Хранилища машинного обучения Калифорнийского университета в Ирвайне (UCI Machine Learning Repository)* и посредством метода tail выведем последние пять строк, чтобы проверить, корректно ли загрузились данные:

```
>>> import os
>>> import pandas as pd
>>> s = os.path.join('https://archive.ics.uci.edu', 'ml',
...                  'machine-learning-databases',
...                  'iris', 'iris.data')
>>> print('URL:', s)
URL: https://archive.ics.uci.edu/ml/machine-learning-databases/
iris/iris.data
```

```
>>> df = pd.read_csv(s,
...                   header=None,
...                   encoding='utf-8')
>>> df.tail()
```

	0	1	2	3	4
<b>145</b>	6.7	3.0	5.2	2.3	Iris-virginica
<b>146</b>	6.3	2.5	5.0	1.9	Iris-virginica
<b>147</b>	6.5	3.0	5.2	2.0	Iris-virginica
<b>148</b>	6.2	3.4	5.4	2.3	Iris-virginica
<b>149</b>	5.9	3.0	5.1	1.8	Iris-virginica



На  
заметку!

### Загрузка набора данных Iris

Копия набора данных Iris (и всех других наборов данных, используемых в книге) включена в состав загружаемого архива с кодом примеров для книги. Указанные копии можно задействовать при автономной работе или в случае временной недоступности ресурса <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data> на сервере UCI. Скажем, чтобы загрузить набор данных Iris из какого-то локального каталога, следующий оператор:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/iris/iris.data',
    header=None, encoding='utf-8')
```

понадобится заменить таким оператором:

```
df = pd.read_csv(
    'ваш/локальный/путь/к/iris.data',
    header=None, encoding='utf-8')
```

Затем мы извлекаем первые 100 меток классов, которые соответствуют 50 цветкам Iris-setosa (ирис разноцветный) и 50 цветкам Iris-versicolor (ирис щетинистый). Далее мы преобразуем метки классов в две целочисленные метки классов 1 (разноцветный) и -1 (щетинистый), которые присваиваем вектору  $y$ ; метод values объекта DataFrame из pandas выдает надлежащее представление NumPy.

Аналогичным образом мы извлекаем из 100 обучающих образцов первый столбец признака (длина чашелистика) и третий столбец признака (длина лепестка), после чего присваиваем их матрице признаков X, которую можно визуализировать через двумерный график рассеяния:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np

>>> # выбрать ирис щетинистый и ирис разноцветный
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', -1, 1)

>>> # извлечь длину чашелистика и длину лепестка
>>> X = df.iloc[0:100, [0, 2]].values

>>> # вычертить график для данных
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='щетинистый')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='x', label='разноцветный')
>>> plt.xlabel('длина чашелистика [см]')
>>> plt.ylabel('длина лепестка [см]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

В результате выполнения приведенного примера кода должен отобразиться график рассеяния (рис. 2.5).

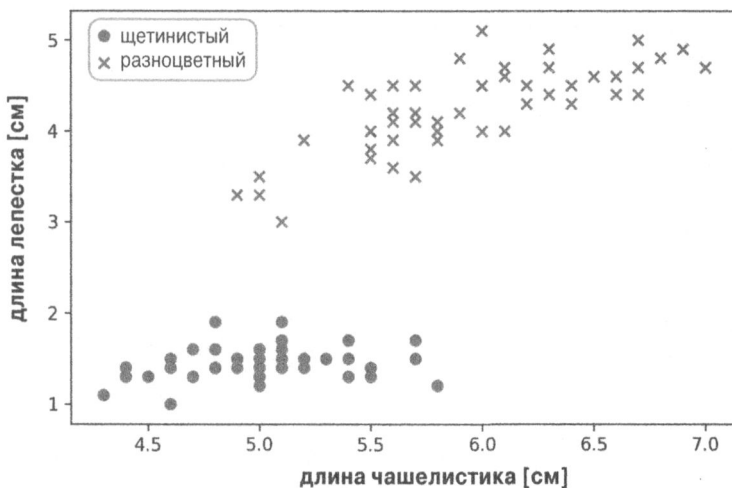


Рис. 2.5. График рассеяния

График рассеяния на рис. 2.5 показывает распределение образцов цветков в наборе данных Iris вдоль двух осей признаков — с длиной лепестка и с длиной чашелистика (измеренных в сантиметрах). В этом двумерном подпространстве признаков мы можем видеть, что линейной границы решений должно быть достаточно для отделения цветков ириса щетинистого от цветков ириса разноцветного. Таким образом, линейный классификатор, такой как персептрон, должен быть в состоянии прекрасно классифицировать цветки в имеющемся наборе данных.

Настало время обучить наш алгоритм персептрона на только что извлеченном поднаборе данных Iris. Мы также построим график ошибки неправильной классификации для каждой эпохи, чтобы проверить, сошелся ли алгоритм и отыскал ли он границу решений, которая разделяет два класса цветков ириса:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1),
...          ppn.errors_, marker='o')
>>> plt.xlabel('Эпохи')
>>> plt.ylabel('Количество обновлений')
>>> plt.show()
```

После выполнения предыдущего кода мы должны получить график ошибки неправильной классификации относительно количества эпох (рис. 2.6).

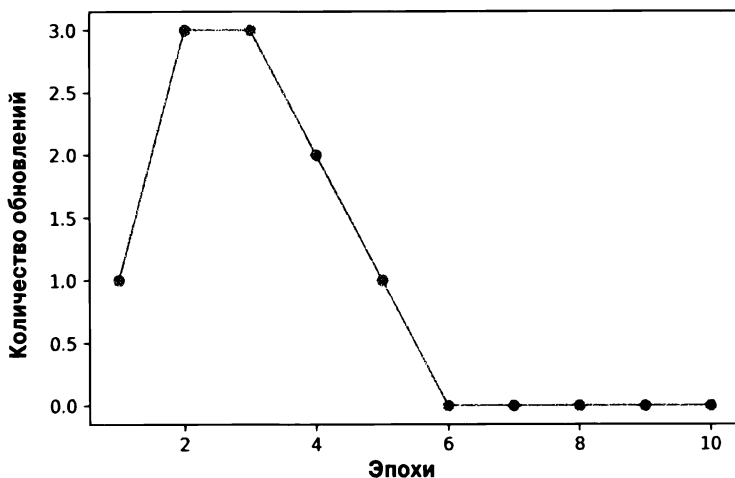


Рис. 2.6. График ошибки неправильной классификации

Как видно на рис. 2.6, наш персептрон сходится после шестой эпохи и теперь должен обладать возможностью в полной мере классифицировать обучающие образцы. Давайте реализуем небольшую удобную функцию, визуализирующую границы решений для двумерных наборов данных:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):
    # настроить генератор маркеров и карту цветов
    markers = ('s', 'x', 'o', '4', 4)
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # вывести поверхность решения
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # вывести образцы по классам
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=cl,
                    edgecolor='black')
```

Сначала мы определяем цвета (colors) и маркеры (markers), после чего посредством ListedColormap создаем карту цветов. Затем мы определяем минимальные и максимальные значения для двух признаков и применяем эти векторы признаков для создания пары сеточных массивов xx1 и xx2 через функцию meshgrid из NumPy. Поскольку мы обучали классификатор на основе персептрона на двух размерностях признаков, нам необходимо выпрямить сеточные массивы и создать матрицу, которая имеет такое же количество столбцов, как у обучающего поднабора Iris, чтобы можно было применять метод predict для прогнозирования меток классов Z соответствующих позиций сеток.



После изменения формы прогнозируемых меток классов  $Z$  на сеточный массив с такими же размерностями, как у `xx1` и `xx2`, мы можем вывести контурный график с помощью функции `contourf` из библиотеки `Matplotlib`, которая сопоставляет разные области решений с разными цветами для каждого прогнозируемого класса в сеточном массиве:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('длина чашелистика [см]')
>>> plt.ylabel('длина лепестка [см]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

В результате выполнения предыдущего примера кода должен отобразиться график с областями решений (рис. 2.7).

На рис. 2.7 видно, что персептрон узнал границу решений, которая способна идеально классифицировать все образцы цветков в обучающем поднаборе `Iris`.

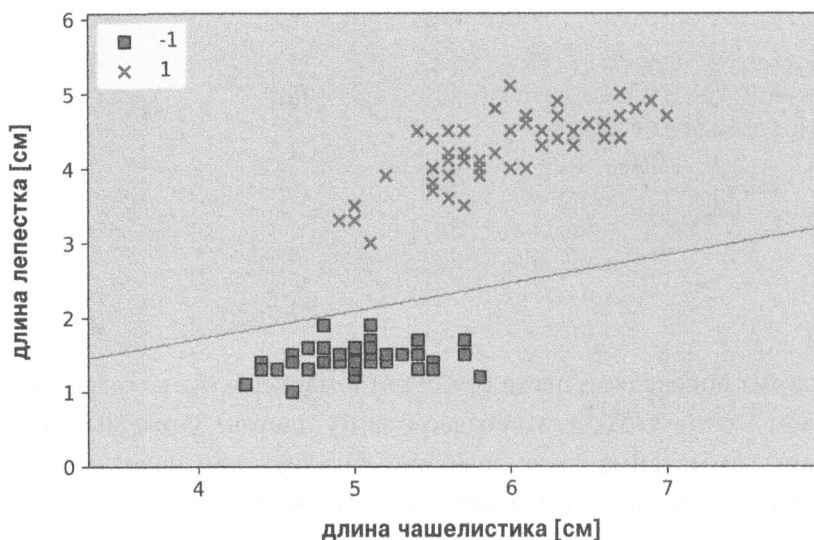


Рис. 2.7. График с областями решений



### Сходимость персептронов

Хотя персептрон прекрасно классифицирует два класса цветков ириса, одной из крупных проблем персептронов является сходимость. Фрэнк Розенблатт математически доказал, что правило обучения персептрона сходится, если два класса могут быть разделены линейной гиперплоскостью. Однако когда разделить классы с помощью такой линейной границы решений невозможно, то веса никогда не перестанут обновляться, если только мы не установим максимальное количество эпох.

## Адаптивные линейные нейроны и сходимость обучения

В этом разделе мы рассмотрим еще один тип однослойной нейронной сети: *Adaline* (*ADaptive Linear NEuron* — адаптивный линейный нейрон). Статья об *Adaline* была опубликована Бернардом Видроу и его докторантом Теддом Хоффом через несколько лет после публикации Фрэнком Розенблаттом статьи об алгоритме персептрона и может считаться усовершенствованием последнего (“An Adaptive ‘Adaline’ Neuron Using Chemical ‘Memistors’”) (Адаптивный нейрон ‘Adaline’, использующий химические ‘мемисторы’), Technical Report Number 1553-2, Б. Видроу и др., Stanford Electron Labs, Stanford, CA (октябрь 1960 года)).

Алгоритм *Adaline* особенно интересен тем, что он иллюстрирует ключевые концепции определения и минимизации непрерывных функций издержек. Это закладывает основу в плане понимания более развитых алгоритмов МО для классификации, таких как логистическая регрессия, методы опорных векторов и регрессионные модели, которые мы обсудим в последующих главах.

Ключевое отличие правила *Adaline* (также известного под названием *правило Видроу-Хоффа*) от правила персептрона Розенблатта связано с тем, что веса обновляются на основе линейной функции активации, а не единичной ступенчатой функции, как в персептроне. В *Adaline* линейная функция активации  $\phi(z)$  является просто тождественным отображением общего входа, так что:

$$\phi(\mathbf{w}^T \mathbf{x}) = \mathbf{w}^T \mathbf{x}$$

Наряду с тем, что для выяснения весов применяется линейная функция активации, для финального прогноза мы по-прежнему используем порого-

вую функцию, которая похожа на раскрытую ранее единичную ступенчатую функцию.

Главные отличия между персептроном и алгоритмом Adaline выделены на рис. 2.8.

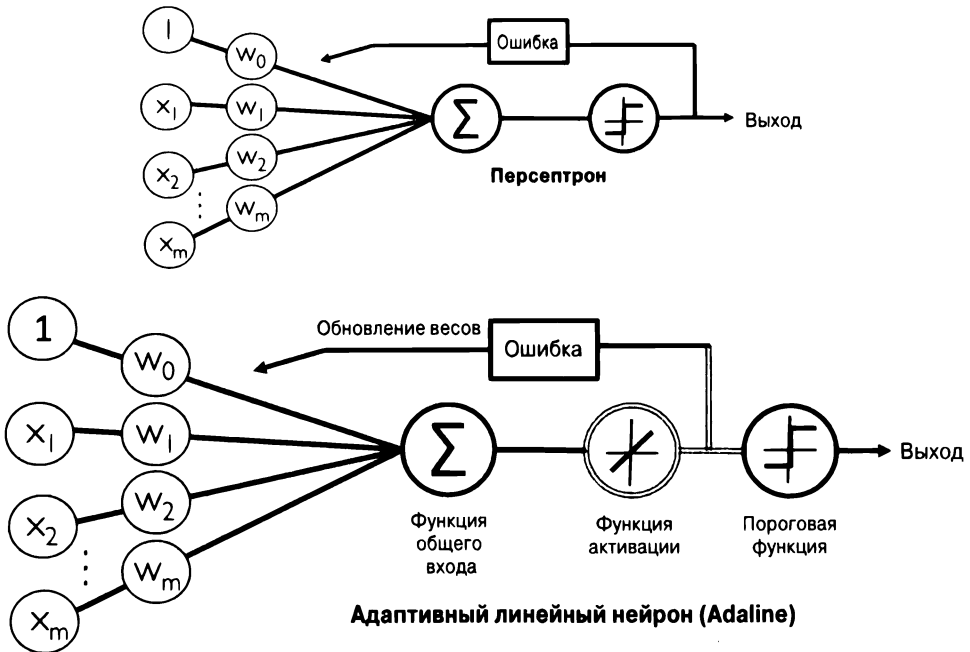


Рис. 2.8. Главные отличия между персептроном и алгоритмом Adaline

На рис. 2.8 видно, что алгоритм Adaline сравнивает настоящие метки классов с непрерывным выходом значений линейной функции активации, вычисляя ошибку модели и обновляя веса. Напротив, персептрон сравнивает настоящие и спрогнозированные метки классов.

## Минимизация функций издержек с помощью градиентного спуска

Одним из ключевых составных частей алгоритмов МО с учителем является определенная *целевая функция* (*objective function*), которая должна быть оптимизирована в течение процесса обучения. Часто такая целевая функция представляет собой функцию издержек, которую мы хотим минимизировать. В случае Adaline мы можем определить функцию издержек,  $J$ , для изучения весов как сумму *квадратичных ошибок* (*sum of squared errors — SSE*) между вычисленным результатом и настоящей меткой класса:

$$J(\mathbf{w}) = \frac{1}{2} \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right)^2$$

Член  $\frac{1}{2}$  добавлен просто ради удобства и, как будет показано ниже, он облегчит выведение градиента функции издержек или потерь относительно весовых параметров. Основное преимущество непрерывной линейной функции активации по сравнению с единичной ступенчатой функцией заключается в том, что функция издержек становится дифференцируемой. Еще одна приятная отличительная черта этой функции издержек — она выпуклая; таким образом, мы можем применять простой, но мощный алгоритм оптимизации под названием *градиентный спуск* (*gradient descent*) для нахождения весов, которые минимизируют функцию издержек при классификации образцов в наборе данных Iris. На рис. 2.9 продемонстрировано, что мы можем описать главную идею, лежащую в основе градиентного спуска, как *скатывание с возвышенности* до тех пор, пока не будет достигнут локальный или глобальный минимум издержек.

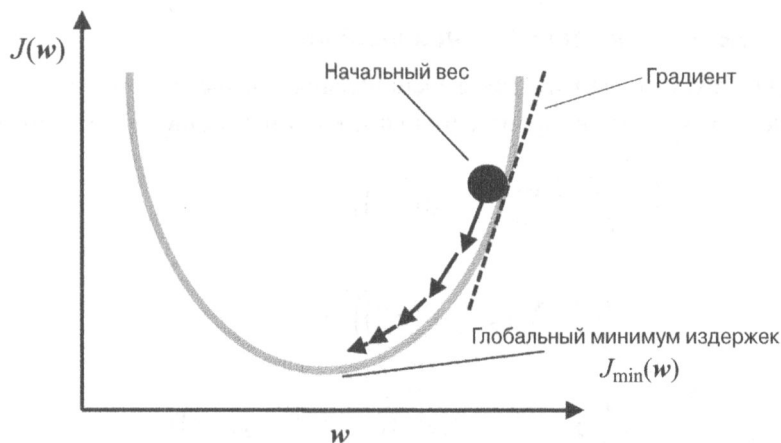


Рис. 2.9. Главная идея градиентного спуска

На каждой итерации мы делаем шаг в направлении, противоположном градиенту, где размер шага определяется величиной скорости обучения, а также наклоном градиента.

С использованием градиентного спуска мы можем обновлять веса путем совершения шага в направлении, противоположном градиенту  $\nabla J(\mathbf{w})$  нашей функции издержек  $J(\mathbf{w})$ :

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

Изменение весов  $\Delta \mathbf{w}$  определяется как отрицательный градиент, умноженный на скорость обучения  $\eta$ :

$$\Delta \mathbf{w} = -\eta \nabla J(\mathbf{w})$$

Чтобы рассчитать градиент функции издержек, нам понадобится вычислить частную производную функции издержек по каждому весу  $w_j$ :

$$\frac{\partial J}{\partial w_j} = -\sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}$$

Следовательно, теперь мы можем записать обновление веса  $w_j$ :

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}$$

Так как мы обновляем все веса одновременно, наше правило обучения Adaline приобретает вид:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$



На  
заметку!

### Производная квадратичной ошибки

Для тех, кто знаком с дифференциальным исчислением, вот как можно получить частную производную функции издержек SSE по  $j$ -тому весу:

$$\begin{aligned} \frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right)^2 = \\ &= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right)^2 = \\ &= \frac{1}{2} \sum_i 2 \left( y^{(i)} - \phi(z^{(i)}) \right) \frac{\partial}{\partial w_j} \left( y^{(i)} - \phi(z^{(i)}) \right) = \\ &= \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) \frac{\partial}{\partial w_j} \left( y^{(i)} - \sum_i \left( w_j x_j^{(i)} \right) \right) = \\ &= \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) \left( -x_j^{(i)} \right) = \\ &= -\sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)} \end{aligned}$$

Несмотря на то что правило обучения Adaline выглядит идентичным правилу персептрона, нужно отметить, что  $\phi(z^{(i)})$  с  $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)}$  — это вещественное число, а не целочисленная метка класса. Кроме того, обновление весов вычисляется на основе всех образцов в обучающем наборе (вместо пошагового обновления весов после каждого образца), отчего данный подход также называют *пакетным градиентным спуском* (*batch gradient descent*).

## Реализация Adaline на Python

Поскольку правило персептрона и Adaline очень похожи, мы возьмем реализацию персептрона, которую определили ранее, и модифицируем метод `fit`, чтобы веса обновлялись за счет минимизации функции издержек посредством градиентного спуска:

```
class AdalineGD(object):
    """Классификатор на основе адаптивного линейного нейрона.

    Параметры
    -----
    eta : float
        Скорость обучения (между 0.0 и 1.0)
    n_iter : int
        Проходы по обучающему набору данных.
    random_state : int
        Начальное значение генератора случайных чисел
        для инициализации случайными весами.

    Атрибуты
    -----
    w_ : одномерный массив
        Веса после подгонки.
    cost_ : список
        Значение функции издержек на основе суммы квадратов
        в каждой эпохе.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """Подгоняет к обучающим данным.
```

Параметры

-----  
*X* : {подобен массиву}, форма = [*n\_examples*, *n\_features*]  
 Обучающие векторы, где *n\_examples* - количество образцов,  
*n\_features* - количество признаков.  
*y* : подобен массиву, форма = [*n\_examples*]  
 Целевые значения.

Возвращает

-----  
*self* : object

"""

```
rgen = np.random.RandomState(self.random_state)
self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
self.cost_ = []
```

```
for i in range(self.n_iter):
    net_input = self.net_input(X)
    output = self.activation(net_input)
    errors = (y - output)
    self.w_[1:] += self.eta * X.T.dot(errors)
    self.w_[0] += self.eta * errors.sum()
    cost = (errors**2).sum() / 2.0
    self.cost_.append(cost)
```

```
return self
```

```
def net_input(self, X):
```

```
    """Вычисляет общий вход"""
```

```
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

```
def activation(self, X):
```

```
    """Вычисляет линейную активацию"""
```

```
    return X
```

```
def predict(self, X):
```

```
    """Возвращает метку класса после единичного шага"""
```

```
    return np.where(self.activation(self.net_input(X))
                    >= 0.0, 1, -1)
```

Вместо обновления весов после оценки каждого отдельного обучающего образца, как в персептроне, мы рассчитываем градиент на основе целого обучающего набора данных через `self.eta * errors.sum()` для члена смещения (нулевого веса) и посредством `self.eta * X.T.dot(errors)` для весов от 1 до *m*, где `X.T.dot(errors)` — матрично-векторное перемножение матрицы признаков и вектора ошибок.

Обратите внимание, что метод `activation` не имеет никакого влияния в коде, т.к. он является просто тождественным отображением. Здесь мы добавляем функцию активации (вычисляемую с помощью метода `activation`), чтобы проиллюстрировать общую концепцию того, каким образом информация протекает через однослойную нейронную сеть: признаки из входных данных, общий вход, активация и выход.

В следующей главе будет рассматриваться классификатор на основе логистической регрессии, в котором применяется отличающаяся от тождественного отображения нелинейная функция активации. Мы увидим, что логистическая регрессионная модель является близкородственной модели Adaline с единственным отличием, касающимся ее функций активации и издержек.

Подобно предыдущей реализации персептрона мы собираем значения издержек в списке `self.cost_`, чтобы после обучения проверить, сошелся ли алгоритм.



На заметку!

### Перемножение матриц

Выполнение перемножения матриц похоже на вычисление скалярного произведения векторов, где каждая строка в матрице трактуется как одиночный вектор-строка. Такой векторизованный подход обеспечивает более компактную систему обозначений и дает в результате более эффективное вычисление с использованием NumPy, например:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

Обратите внимание, что в предыдущем уравнении мы перемножаем матрицу и вектор, что математически не определено. Тем не менее, вспомните о принятии нами соглашения о том, что предыдущий вектор рассматривается как матрица  $3 \times 1$ .

На практике алгоритм часто требует некоторой доли экспериментирования, чтобы отыскать хорошую скорость обучения  $\eta$  для оптимального схождения. Итак, давайте в самом начале выберем две разных скорости обучения,  $\eta = 0.01$  и  $\eta = 0.0001$ , и построим графики функций издержек относительно количества эпох для выяснения, насколько эффективно реализация Adaline обучается на обучающих данных.





## Гиперпараметры персептрона

Скорость обучения  $\eta$  (eta) и количество эпох (`n_iter`) представляют собой так называемые гиперпараметры алгоритмов обучения персептрона и Adaline. В главе 6 мы взглянем на другие приемы автоматического нахождения значений для гиперпараметров, которые дают оптимальную эффективность классификационной модели.

А теперь давайте вычертим график зависимости издержек от количества эпох для двух разных скоростей обучения:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> adal = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
>>> ax[0].plot(range(1, len(adal.cost_) + 1),
...            np.log10(adal.cost_), marker='o')
>>> ax[0].set_xlabel('Эпохи')
>>> ax[0].set_ylabel('log(Сумма квадратичных ошибок)')
>>> ax[0].set_title('Adaline - скорость обучения 0.01')
>>> ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.cost_) + 1),
...            ada2.cost_, marker='o')
>>> ax[1].set_xlabel('Эпохи')
>>> ax[1].set_ylabel('Сумма квадратичных ошибок')
>>> ax[1].set_title('Adaline - скорость обучения 0.0001')
>>> plt.show()
```

На результирующих графиках для функции издержек (рис. 2.10) видно, что мы столкнулись с проблемами двух типов. График слева показывает, что могло бы произойти, если бы мы выбрали слишком высокую скорость обучения. Вместо минимизации функции издержек ошибка с каждой эпохой становится все больше, т.к. мы *проскакиваем* глобальный минимум. С другой стороны, на графике справа мы можем отметить уменьшение издержек, но выбранная скорость обучения  $\eta = 0.0001$  настолько мала, что алгоритм требует слишком большого количества эпох для схождения в глобальном минимуме издержек.

На рис. 2.11 показано, что может произойти, если мы изменим значение отдельного параметра веса для минимизации функции издержек  $J$ . График слева иллюстрирует случай удачно выбранной скорости обучения, при котором издержки постепенно уменьшаются, перемещаясь в направлении глобального минимума. График справа демонстрирует ситуацию, когда выбранная скорость обучения слишком высока — мы *проскакиваем* глобальный минимум.

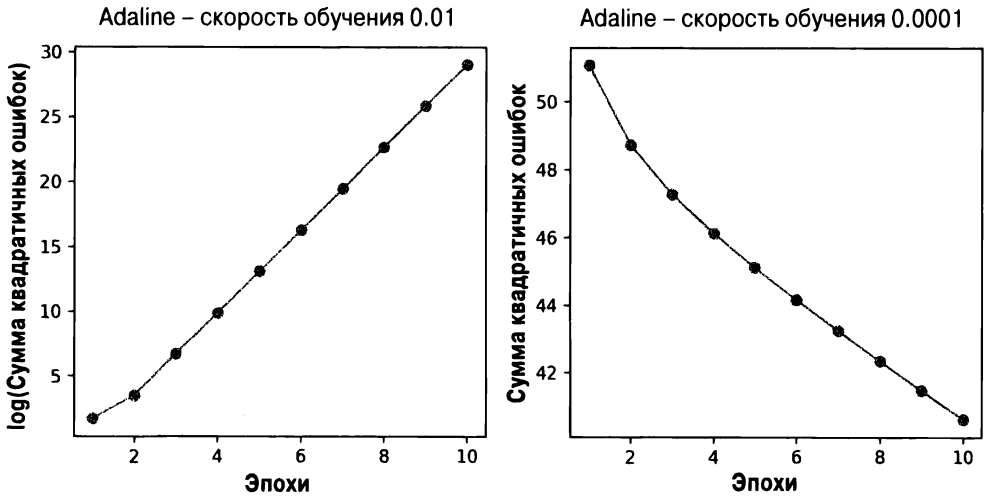


Рис. 2.10. Графики функции издержек для двух разных скоростей обучения

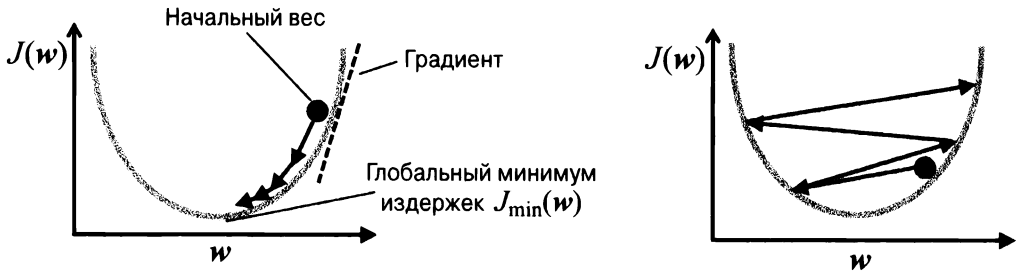


Рис. 2.11. Изменение значения отдельного параметра веса с целью минимизации функции издержек

## Улучшение градиентного спуска посредством масштабирования признаков

Многие алгоритмы МО, встречающиеся в книге, для оптимальной эффективности требуют определенного масштабирования признаков, которое мы более подробно обсудим в главах 3 и 4.

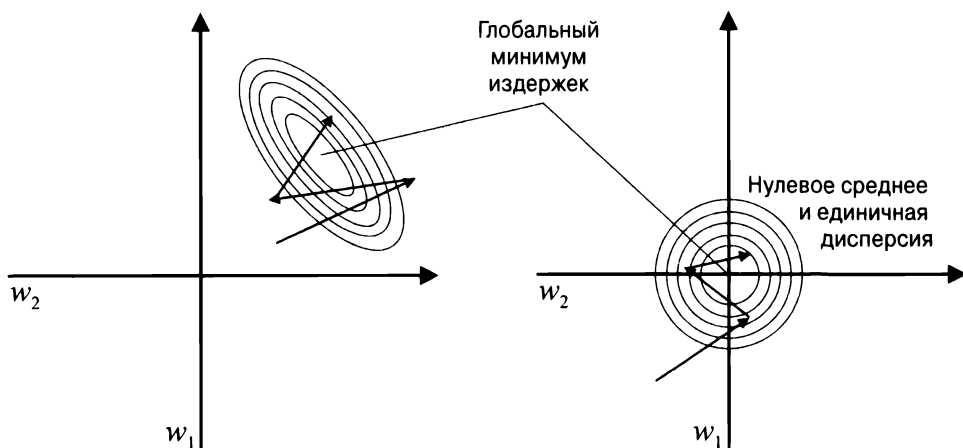
Градиентный спуск — один из многих алгоритмов, извлекающих пользу из масштабирования признаков. В этом разделе мы будем применять метод масштабирования признаков под названием *стандартизация*, который придает нашим данным характеристики стандартного нормального распре-

деления: нулевое среднее и единичную дисперсию. Такая процедура нормализации содействует более быстрому сходимости обучения с градиентным спуском; однако, она не делает исходный набор данных нормально распределенным. Стандартизация так смещает среднее каждого признака, что оно центрируется возле нуля, и каждый признак имеет стандартное отклонение 1 (единичную дисперсию). Например, чтобы стандартизировать  $j$ -тый признак, мы можем просто вычесть выборочное среднее  $\mu_j$  из каждого обучающего образца и разделить результат на стандартное отклонение  $\sigma_j$ :

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Здесь  $x_j$  — вектор, состоящий из значений  $j$ -того признака всех обучающих образцов  $n$ , и этот прием стандартизации применяется к каждому признаку  $j$  в нашем наборе данных.

Одна из причин, по которым стандартизация содействует обучению с градиентным спуском, заключается в том, что оптимизатору приходится проходить через меньшее число шагов в поиске хорошего или оптимального решения (глобального минимума издержек). Сказанное иллюстрируется на рис. 2.12, где пространство издержек представлено как функция от двух весов модели в задаче двумерной классификации.



**Рис. 2.12.** Влияние стандартизации на обучение с градиентным спуском

Стандартизацию можно легко обеспечить с использованием встроенных в NumPy методов `mean` и `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

Проведя стандартизацию, мы обучим алгоритм Adaline снова и заметим, что он сходится после небольшого количества эпох при скорости обучения  $\eta = 0.01$ :

```
>>> ada_gd = AdalineGD(n_iter=15, eta=0.01)
>>> ada_gd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>>> plt.title('Adaline - градиентный спуск')
>>> plt.xlabel('длина чашелистика [стандартизированная]')
>>> plt.ylabel('длина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

>>> plt.plot(range(1, len(ada_gd.cost_) + 1),
...          ada_gd.cost_, marker='o')
>>> plt.xlabel('Эпохи')
>>> plt.ylabel('Сумма квадратичных ошибок')
>>> plt.tight_layout()
>>> plt.show()
```

В результате выполнения приведенного кода мы должны получить изображение областей решений и график снижающихся издержек (рис. 2.13).

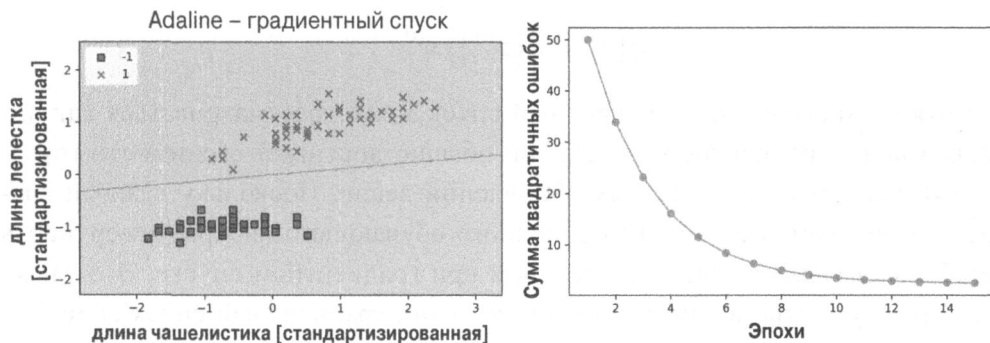


Рис. 2.13. Алгоритм Adaline с градиентным спуском после стандартизации

На рис. 2.13 видно, что алгоритм Adaline теперь сходится после обучения на стандартизированных признаках с применением скорости обучения  $\eta = 0.01$ . Тем не менее, следует отметить, что ошибка SSE остается ненулевой, даже когда все образцы цветков были классифицированы корректно.

## Крупномасштабное машинное обучение и стохастический градиентный спуск

В предыдущем разделе вы узнали, как минимизировать функцию издержек путем совершения шага в направлении, противоположном градиенту издержек, который вычисляется на основе полного обучающего набора; именно потому этот подход иногда называют также *пакетным градиентным спуском*. А теперь представим, что у нас есть очень крупный набор данных с миллионами точек данных, который нередко встречается во многих приложениях МО. В сценариях подобного рода выполнение пакетного градиентного спуска может оказаться довольно затратным в вычислительном плане, т.к. каждый раз, когда мы делаем один шаг по направлению к глобальному минимуму, нам необходимо заново оценивать полный обучающий набор данных.

Популярной альтернативой пакетного градиентного спуска является *стохастический градиентный спуск* (*stochastic gradient descent* — SGD), временами также называемый итеративным или динамическим градиентным спуском. Вместо обновления весов на основе суммы накопленных ошибок по всем образцам  $\mathbf{x}^{(i)}$ :

$$\Delta \mathbf{w} = \eta \sum_i \left( y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}$$

мы обновляем веса постепенно для каждого обучающего образца:

$$\eta \left( y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}$$

Хотя стохастический градиентный спуск может рассматриваться как аппроксимация градиентного спуска, он обычно достигает сходимости гораздо быстрее из-за более частых обновлений весов. Поскольку каждый градиент вычисляется на основе одиночного обучающего набора, поверхность ошибок оказывается зашумленной, чем при градиентном спуске. Это также дает преимущество в том, что стохастический градиентный спуск способен с большей легкостью избегать локальных минимумов, если мы работаем с нелинейными функциями издержек, как будет показано в главе 12. Чтобы

получить удовлетворительные результаты посредством стохастического градиентного спуска, важно подавать ему обучающие данные в случайном порядке; к тому же во избежание циклов нам необходимо тасовать обучающий набор для каждой эпохи.



### Регулирование скорости обучения

В реализациях стохастического градиентного спуска фиксированная скорость обучения  $\eta$  часто заменяется адаптивной скоростью обучения, которая увеличивается с течением временем, например:

$$\frac{c_1}{\left[ \text{количество итераций} \right] + c_2}$$

где  $c_1$  и  $c_2$  — константы. Следует отметить, что стохастический градиентный спуск не попадает в глобальный минимум, но оказывается в очень близкой к нему области. Используя адаптивную скорость обучения, мы можем достичь дальнейшего приближения к минимуму издержек.

Еще одно преимущество стохастического градиентного спуска связано с тем, что его можно применять для *динамического обучения* (*online learning*). При динамическом обучении наша модель обучается на лету по мере поступления новых обучающих данных, что особенно практично, когда мы накапливаем крупные объемы данных, скажем, информацию о заказчиках в веб-приложениях. За счет использования динамического обучения система может немедленно адаптироваться к изменениям, а в случае ограниченного пространства хранения обучающие данные могут отбрасываться сразу после обновления модели.



### Мини-пакетный градиентный спуск

Компромиссом между пакетным градиентным спуском и стохастическим градиентным спуском является так называемое *мини-пакетное обучение*. Мини-пакетное обучение можно понимать как применение пакетного градиентного спуска к небольшим поднаборам обучающих данных, например, к 32 обучающим образцам за раз. Преимущество по сравнению с пакетным градиентным спуском в том, что благодаря мини-пакетам сходимость достигается быстрее по причине более частых обновлений весов. Кроме того, мини-пакетное обучение дает нам возможность заменить цикл `for` по обуча-

ющим образцам при стохастическом градиентном спуске векторизованными операциями, которые задействуют концепции из линейной алгебры (скажем, реализация взвешенной суммы через скалярное произведение), что может дополнительно улучшить вычислительную эффективность нашего алгоритма обучения.

Поскольку мы уже реализовали правило обучения Adaline с использованием градиентного спуска, нам нужно лишь внести несколько изменений, чтобы заставить алгоритм обучения обновлять веса посредством стохастического градиентного спуска. Внутри метода `fit` мы теперь будем обновлять веса после прохождения каждого обучающего образца. Вдобавок мы реализуем дополнительный метод `partial_fit`, предназначенный для динамического обучения, который не будет повторно инициализировать веса. Для проверки, сходится ли наш алгоритм после обучения, мы будем вычислять издержки как усредненные издержки обучающих образцов в каждой эпохе. Кроме того, мы добавим возможность тасования обучающих данных перед каждой эпохой, чтобы избежать повторяющихся циклов во время оптимизации функции издержек; параметр `random_state` позволяет указывать случайное начальное значение для воспроизводимости:

```
class AdalineSGD(object):
    """Классификатор на основе адаптивного линейного нейрона.

    Параметры
    -----
    eta : float
        Скорость обучения (между 0.0 и 1.0)
    n_iter : int
        Проходы по обучающему набору данных.
    shuffle : bool (по умолчанию: True)
        Если True, тогда тасовать обучающие данные во избежание циклов.
    random_state : int
        Начальное значение генератора случайных чисел
        для инициализации случайными весами.

    Атрибуты
    -----
    w_ : одномерный массив
        Веса после подгонки.
    cost_ : список
        Значение функции издержек на основе суммы квадратов,
        усредненное по всем обучающим образцам в каждой эпохе.

    """
```

```

def __init__(self, eta=0.01, n_iter=10,
              shuffle=True, random_state=None):
    self.eta = eta
    self.n_iter = n_iter
    self.w_initialized = False
    self.shuffle = shuffle
    self.random_state = random_state

def fit(self, X, y):
    """Подгоняет к обучающим данным.

    Параметры
    -----
    X : {подобен массиву}, форма = [n_examples, n_features]
        Обучающие векторы, где n_examples - количество образцов,
        n_features - количество признаков.
    y : подобен массиву, форма = [n_examples]
        Целевые значения.

    Возвращает
    -----
    self : object
    """
    self._initialize_weights(X.shape[1])
    self.cost_ = []
    for i in range(self.n_iter):
        if self.shuffle:
            X, y = self._shuffle(X, y)
        cost = []
        for xi, target in zip(X, y):
            cost.append(self._update_weights(xi, target))
        avg_cost = sum(cost) / len(y)
        self.cost_.append(avg_cost)
    return self

def partial_fit(self, X, y):
    """Подгоняет к обучающим данным без повторной
    инициализации весов"""
    if not self.w_initialized:
        self._initialize_weights(X.shape[1])
    if y.ravel().shape[0] > 1:
        for xi, target in zip(X, y):
            self._update_weights(xi, target)
    else:
        self._update_weights(X, y)
    return self

```



```
def _shuffle(self, X, y):  
    """Тасует обучающие данные"""  
    r = self.rgen.permutation(len(y))  
    return X[r], y[r]  
  
def _initialize_weights(self, m):  
    """Инициализирует веса небольшими случайными числами"""  
    self.rgen = np.random.RandomState(self.random_state)  
    self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)  
    self.w_initialized = True  
  
def _update_weights(self, xi, target):  
    """Применяет правило обучения Adaline для обновления весов"""  
    output = self.activation(self.net_input(xi))  
    error = (target - output)  
    self.w_[1:] += self.eta * xi.dot(error)  
    self.w_[0] += self.eta * error  
    cost = 0.5 * error**2  
    return cost  
  
def net_input(self, X):  
    """Вычисляет общий вход"""  
    return np.dot(X, self.w_[1:]) + self.w_[0]  
  
def activation(self, X):  
    """Вычисляет линейную активацию"""  
    return X  
  
def predict(self, X):  
    """Возвращает метку класса после единичного шага"""  
    return np.where(self.activation(self.net_input(X))  
                    >= 0.0, 1, -1)
```

Метод `_shuffle`, который мы теперь используем в классификаторе `AdalineSGD`, работает следующим образом: посредством функции `permutation` из `np.random` мы генерируем случайную последовательность уникальных чисел в диапазоне от 0 до 100. Затем эти числа можно применять в качестве индексов для тасования нашей матрицы признаков и вектора меток классов.

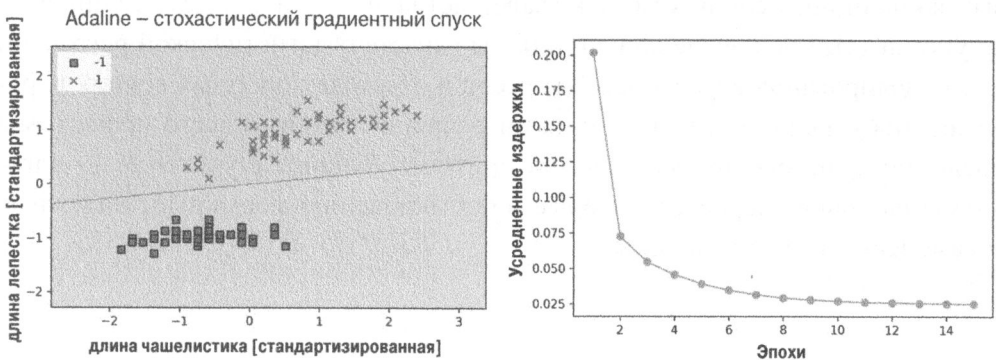
Далее мы можем использовать метод `fit` для обучения классификатора `AdalineSGD` и нашу функцию `plot_decision_regions` для отображения результатов обучения:

```

>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - стохастический градиентный спуск')
>>> plt.xlabel('длина чашелистика [стандартизированная]')
>>> plt.ylabel('длина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_sgd.cost_) + 1), ada_sgd.cost_,
...          marker='o')
>>> plt.xlabel('Эпохи')
>>> plt.ylabel('Усредненные издержки')
>>> plt.tight_layout()
>>> plt.show()

```

После запуска предыдущего примера кода мы получаем два графика, показанные на рис. 2.14.



**Рис. 2.14.** Алгоритм Adaline со стохастическим градиентным спуском после стандартизации

Несложно заметить, что усредненные издержки снижаются довольно быстро, а финальная граница решений после 15 эпох выглядит похожей на границу решений в алгоритме Adaline с пакетным градиентным спуском. Скажем, если необходимо обновить модель в сценарии динамического обучения с потоковыми данными, тогда мы могли бы просто вызывать метод `partial_fit` на индивидуальных обучающих образцах — например, `ada_sgd.partial_fit(X_std[0, :], y[0])`.

## Резюме

В этой главе вы получили хорошее представление о базовых концепциях линейных классификаторов для обучения с учителем. После реализации персептрона мы продемонстрировали, каким образом можно эффективно обучать адаптивные линейные нейроны через векторизованную реализацию градиентного спуска и проводить динамическое обучение посредством стохастического градиентного спуска.

Теперь, когда было показано, как реализовывать простые классификаторы на Python, вы готовы перейти к следующей главе, где будет применяться библиотека Python для МО под названием `scikit-learn` для получения доступа к более развитым и мощным классификаторам МО, которые обычно используются в научном сообществе и производственной среде.

Объектно-ориентированный подход, применяемый при реализации алгоритмов персептрона и Adaline, будет содействовать пониманию API-интерфейса `scikit-learn`, который реализован на основе тех же самых ключевых концепций, используемых в главе: методов `fit` и `predict`. Опираясь на упомянутые основные концепции, вы узнаете о логистической регрессии для моделирования вероятностей классов и о методах опорных векторов, работающих с нелинейными границами решений. Помимо всего прочего мы представим отличающийся класс алгоритмов обучения с учителем — алгоритмы на основе деревьев, которые по обыкновению формируют надежные ансамблевые классификаторы.

# ОБЗОР КЛАССИФИКАТОРОВ НА ОСНОВЕ МАШИННОГО ОБУЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ SCIKIT-LEARN

**В** этой главе мы пройдемся по подборке популярных и мощных алгоритмов МО, которые широко применяются как в научном сообществе, так и в производственной среде. В ходе исследования отличий между несколькими алгоритмами обучения с учителем, предназначенными для классификации, мы разовьем интуитивное понимание их индивидуальных достоинств и недостатков. Вдобавок мы совершим свои первые шаги в освоении библиотеки `scikit-learn`, которая предлагает дружелюбный к пользователю и согласованный интерфейс для эффективного и рационального использования таких алгоритмов.

В главе будут раскрыты следующие темы:

- введение в надежные и популярные алгоритмы классификации, в числе которых логистическая регрессия, методы опорных векторов и деревья принятия решений;

- примеры и пояснения сценариев применения библиотеки МО под названием *scikit-learn*, которая предоставляет широкий выбор алгоритмов МО через дружественный к пользователю API-интерфейс на основе Python;
- обсуждение достоинств и недостатков классификаторов с линейными и нелинейными границами решений.

## Выбор алгоритма классификации

Выбор подходящего алгоритма классификации для конкретной задачи требует опыта; каждый алгоритм обладает индивидуальными особенностями и основывается на определенных допущениях. Формулируя по-другому теорему Дэвида Вольперта об отсутствии бесплатных завтраков, можно сказать, что ни один классификатор не работает лучше других во всех возможных сценариях (“The Lack of A Priori Distinctions Between Learning Algorithms”) (Отсутствие априорных различий между обучающими алгоритмами), Д. Вольперт, *Neural Computation* 8.7: с. 1341–1390 (1996 год)). На практике всегда рекомендуется сравнивать эффективность хотя бы нескольких разных алгоритмов обучения, чтобы выбрать наилучшую модель для имеющейся задачи; они могут отличаться по количеству признаков или образцов, объему шума в наборе данных и тому, являются классы линейно сепарабельными или нет.

В итоге эффективность классификатора — вычислительная мощность и прогнозирующая способность — сильно зависят от лежащих в основе данных, доступных для обучения. Пять главных шагов, выполняемых при обучении алгоритма МО, могут быть подытожены следующим образом.

1. Отбор признаков и накопление помеченных обучающих образцов.
2. Выбор метрики эффективности.
3. Выбор алгоритма классификации и оптимизации.
4. Оценка эффективности модели.
5. Точная настройка алгоритма.

Поскольку принятый в книге подход предусматривает постепенное обретение знаний МО, в настоящей главе мы преимущественно будем концентрировать внимание на главных концепциях различных алгоритмов и поз-

же возвратимся к более подробному обсуждению таких тем, как выбор и предварительная обработка признаков, метрики эффективности и настройка гиперпараметров.

## Первые шаги в освоении *scikit-learn* — обучение персептрона

В главе 2 вы узнали о двух связанных алгоритмах обучения для классификации, правиле персептрона и Adaline, которые были самостоятельно реализованы с помощью Python и NumPy. Теперь мы посмотрим на API-интерфейс *scikit-learn*, объединяющий дружественный к пользователю и согласованный интерфейс с высокооптимизированной реализацией нескольких алгоритмов классификации. Библиотека *scikit-learn* предлагает не только большое разнообразие алгоритмов обучения, но также много удобных функций для предварительной обработки данных и точной настройки и оценки моделей. Мы обсудим их более детально вместе с лежащими в основе концепциями в главах 4 и 5.

В качестве начала работы с библиотекой *scikit-learn* мы обучим модель на базе персептрона, подобную той, которая была реализована в главе 2. Ради простоты во всех последующих разделах мы будем использовать хорошо знакомый набор данных Iris. Удобно то, что набор данных Iris уже доступен через *scikit-learn*, т.к. он представляет собой простой, но популярный набор данных, который часто применяется при проведении испытаний и экспериментов с алгоритмами. Подобно предыдущей главе для целей визуализации мы будем использовать только два признака из набора данных Iris.

Мы присвоим длины и ширины лепестков в 150 образцах цветков матрице признаков *X*, а метки классов, соответствующие видам цветков — вектору *y*:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Метки классов:', np.unique(y))
Метки классов: [0 1 2]
```

Функция `np.unique(y)` возвращает три уникальные метки классов, хранящиеся в `iris.target`, и как мы видим, имена классов цветков ириса `Iris-setosa`, `Iris-versicolor` и `Iris-virginica` уже сохранены в виде целых чисел (здесь 0, 1, 2). Хотя многие функции и методы классов `scikit-learn` также работают с метками классов в строковом формате, применение целочисленных меток является рекомендуемым подходом, позволяющим избежать технических сбоев и улучшить вычислительную эффективность благодаря меньшему объему занимаемой памяти. Кроме того, кодирование меток классов как целых чисел — общепринятое соглашение в большинстве библиотек МО.

Чтобы оценить, насколько хорошо обученная модель работает на не встречавшихся ранее данных, мы дополнительно разобьем набор данных на обучающий и испытательный наборы. В главе 6 мы более подробно обсудим установившуюся практику оценки моделей:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y)
```

С помощью функции `train_test_split` из модуля `model_selection` библиотеки `scikit-learn` мы случайным образом разбиваем массивы `X` и `y` на 30% испытательных данных (45 образцов) и 70% обучающих данных (105 образцов).

Обратите внимание, что внутренне функция `train_test_split` тасует обучающие наборы перед разбиением; иначе все образцы классов 0 и 1 в итоге попали бы в обучающий набор, а испытательный набор состоял бы из 45 образцов класса 2. Посредством параметра `random_state` мы предоставляем фиксированное случайное начальное значение (`random_state=1`) для внутреннего генератора псевдослучайных чисел, который используется при тасовании наборов данных, предворяющем их разбиение. Применение такого фиксированного значения `random_state` гарантирует, что наши результаты будут воспроизводимыми.

Наконец, через `stratify=y` мы используем в своих интересах встроенную поддержку стратификации. В нашем контексте стратификация означает, что метод `train_test_split` возвращает обучающий и испытательный поднаборы, имеющие такие же количественные соотношения меток классов, как у входного набора данных. Мы можем проверить, так ли это, с при-

менением функции `bincount` из NumPy, которая подсчитывает количество вхождений каждого значения в массиве:

```
>>> print('Количества меток в y:', np.bincount(y))
Количества меток в y: [50 50 50]
>>> print('Количества меток в y_train:', np.bincount(y_train))
Количества меток в y_train: [35 35 35]
>>> print('Количества меток в y_test:', np.bincount(y_test))
Количества меток в y_test: [15 15 15]
```

Как выяснилось в примере с *градиентным спуском* из главы 2, для достижения оптимальной эффективности многие алгоритмы МО и оптимизации также требуют масштабирования признаков. В текущем примере мы стандартизируем признаки, используя класс `StandardScaler` из модуля `preprocessing` библиотеки *scikit-learn*:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

В показанном выше коде мы загружаем класс `StandardScaler` из модуля `preprocessing`, инициализируем новый объект `StandardScaler` и присваиваем его переменной `sc`. С помощью метода `fit` объект `StandardScaler` оценивает параметры  $\mu$  (выборочное среднее) и  $\sigma$  (стандартное отклонение) для каждой размерности признаков из обучающих данных. Затем за счет вызова метода `transform` мы стандартизируем обучающие данные, указывая оценочные параметры  $\mu$  и  $\sigma$ . Обратите внимание, что при стандартизации испытательного набора мы применяем те же самые параметры масштабирования, обеспечивая возможность сравнения значений в обучающем и испытательном наборах данных.

Стандартизовав обучающие данные, мы можем обучить модель на основе перцептрона. Большинство алгоритмов в *scikit-learn* по умолчанию поддерживают многоклассовую классификацию посредством метода “*один против остальных*” (*one-versus-rest* — OvR), который позволяет предоставлять перцептрону сразу три класса цветков. Вот соответствующий код:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```



Интерфейс *scikit-learn* будет напоминать реализацию персептрона, построенную нами в главе 2. После загрузки класса *Perceptron* из модуля *linear\_model* мы инициализируем новый объект *Perceptron* и обучаем модель через метод *fit*. Здесь параметр модели *eta0* эквивалентен скорости обучения *eta*, которую мы использовали в собственной реализации персептрона, а параметр *n\_iter* определяет количество эпох (проходов по обучающему набору).

Как объяснялось в главе 2, нахождение подходящей скорости обучения требует некоторого экспериментирования. Если скорость обучения чересчур высока, то алгоритм проскочит глобальный минимум издержек. Если скорость обучения слишком низка, тогда алгоритм потребует для своего схождения большего числа эпох, что может замедлить обучение — особенно в случае крупных наборов данных. Мы также применяем параметр *random\_state*, чтобы обеспечить воспроизводимость начального тасования обучающего набора данных после каждой эпохи.

Обучив модель посредством *scikit-learn*, мы можем вырабатывать прогнозы через метод *predict*, как это делалось в нашей реализации персептрона из главы 2. Ниже приведен код:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Неправильно классифицированных образцов: %d'
        % (y_test != y_pred).sum())
Неправильно классифицированных образцов: 1
```

В результате выполнения кода мы видим, что персептрон неправильно классифицировал 1 из 45 образцов цветков. Таким образом, ошибка неправильной классификации на испытательном наборе данных составляет приблизительно 0.022, или 2.2% ( $1 / 45 \approx 0.022$ ).



### Ошибка классификации или правильность

Вместо ошибки неправильной классификации многие специалисты-практики по МО вычисляют правильность классификации модели:

$$1 - \text{ошибка} = 0.978, \text{ или } 97.8\%.$$

Выбор между ошибкой классификации или правильностью зависит только от личных предпочтений.

Следует отметить, что в библиотеке *scikit-learn* также реализовано широкое разнообразие различных метрик эффективности, которые доступны через модуль *metrics*. Например, вот как мы можем вычислить правильность классификации персептрона на испытательном наборе:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Правильность: %.3f' % accuracy_score(y_test, y_pred))
Правильность: 0.978
```

Здесь *y\_test* — настоящие метки классов, а *y\_pred* — метки классов, которые мы спрогнозировали ранее. В качестве альтернативы каждый классификатор в *scikit-learn* имеет метод *score*, который подсчитывает правильность прогнозирования классификатора, объединяя вызов *predict* с *accuracy\_score*:

```
>>> print('Правильность: %.3f' % ppn.score(X_test_std, y_test))
Правильность: 0.978
```



На  
заметку!

Обратите внимание, что в этой главе мы оцениваем эффективность моделей на основе испытательного набора. В главе 6 вы узнаете о полезных методиках, включая графический анализ, такой как кривые обучения, для обнаружения и предотвращения *переобучения* (*overfitting*). Переобучение, к обсуждению которого мы возвратимся позже в главе, означает, что модель хорошо захватывает шаблоны в обучающем наборе, но плохо обобщается на не встречавшиеся ранее данные.

Наконец, мы можем использовать нашу функцию *plot\_decision\_regions* из главы 2 для вывода графика *областей решений* недавно обученной модели на основе персептрона и визуализации того, насколько хорошо она отделяет друг от друга различные образцы цветков. Тем не менее, давайте внесем в нее небольшое изменение, чтобы выделять образцы данных из испытательного набора маленькими окружностями:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

def plot_decision_regions(X, y, classifier, test_idx=None,
                        resolution=0.02):

    # настроить генератор маркеров и цветовую карту
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
```

```

# вывести поверхность решения
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                        np.arange(x2_min, x2_max, resolution))
Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
Z = Z.reshape(xx1.shape)
plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                alpha=0.8, c=colors[idx],
                marker=markers[idx], label=cl,
                edgecolor='black')

# выделить образцы из испытательного набора
if test_idx:
    # вычертить все образцы
    X_test, y_test = X[test_idx, :], y[test_idx]

    plt.scatter(X_test[:, 0], X_test[:, 1],
                c='', edgecolor='black', alpha=1.0,
                linewidth=1, marker='o',
                s=100, label='испытательный набор')

```

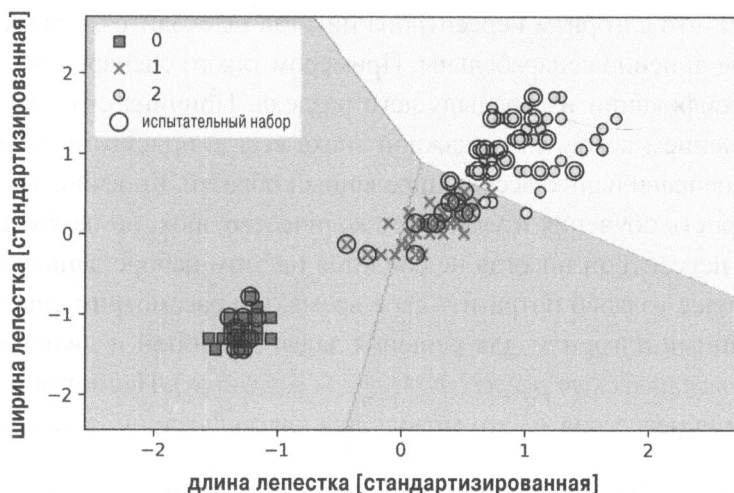
Благодаря такой небольшой модификации, внесенной в функцию `plot_decision_regions`, мы можем указывать индексы образцов, которые хотим пометить в результирующем графике. Код выглядит следующим образом:

```

>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105, 150))
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

```

На рис. 3.1 видно, что три класса цветков не могут разделяться в полной мере линейной границей решений.



*Рис. 3.1. Невозможность хорошего разделения трех класса цветков с помощью линейной границы решений*

Вспомните из обсуждения в главе 2, что алгоритм персептрона никогда не сходится на наборах данных, которые не полностью линейно сепарабельны, из-за чего применять алгоритм персептрона на практике обычно не рекомендуется. В последующих разделах мы рассмотрим более мощные линейные классификаторы, которые сходятся к минимуму издержек, даже если классы не являются идеально линейно сепарабельными.



### Дополнительные настройки персептрона

На заметку!

Класс `Perceptron`, равно как другие функции и классы *scikit-learn*, часто имеют дополнительные параметры, которые мы опустили ради ясности. Ознакомиться с более подробными сведениями об этих параметрах можно с использованием функции `help` в Python (например, `help(Perceptron)`) или обратившись к великолепной онлайн-документации *scikit-learn* по ссылке <http://scikit-learn.org/stable/>.

## Моделирование вероятностей классов посредством логистической регрессии

Хотя правило персептрона предлагает хорошее и естественное введение в алгоритмы МО, предназначенные для классификации, его крупнейший недо-

статок в том, что алгоритм персептрона никогда не сходится, если классы не в полной мере линейно сепарабельны. Примером такого сценария послужила бы задача классификации из предыдущего раздела. Причина связана с постоянным обновлением весов, т.к. в каждой эпохе всегда присутствует, по крайней мере, один неправильно классифицированный образец. Конечно, мы можем изменить скорость обучения и увеличить количество эпох, но необходимо иметь в виду, что персептрон никогда не сойдется на этом наборе данных.

Чтобы более толково потратить свое время, мы рассмотрим еще один простой, но мощный алгоритм для решения задач линейной и двоичной классификации: *логистическую регрессию (logistic regression)*. Несмотря на название, логистическая регрессия — это модель для классификации, а не регрессии.

## Понятие логистической регрессии и условные вероятности

Логистическая регрессия представляет собой классификационную модель, которая крайне проста в реализации, но очень хорошо работает на линейно сепарабельных классах. Логистическая регрессия — один из наиболее широко применяемых в производственной среде алгоритмов, предназначенных для классификации. Подобно персептрону и Adaline логистическая регрессионная модель в текущей главе является также линейной моделью для двоичной классификации.



На  
заметку!

### Логистическая регрессия для множества классов

Обратите внимание, что логистическая регрессия может быть без труда обобщена до многоклассовой конфигурации, которая известна как полиномиальная (multinomial) логистическая регрессия или многопеременная (softmax) логистическая регрессия. Более подробное раскрытие полиномиальной логистической регрессии выходит за рамки настоящей книги, но заинтересованные читатели могут почерпнуть дополнительную информацию из конспекта лекций: [https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L05\\_gradient-descent\\_slides.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L05_gradient-descent_slides.pdf) или [http://rasbt.github.io/mlxtend/user\\_guide/classifier/SoftmaxRegression/](http://rasbt.github.io/mlxtend/user_guide/classifier/SoftmaxRegression/).

Еще один способ использования логистической регрессии в многоклассовой конфигурации предусматривает применение обсуждавшейся ранее методики OvR.

Чтобы прояснить идею, лежащую в основе логистической регрессии как вероятностной модели для двоичной классификации, мы введем понятие *перевеса* (*odds*): перевеса в пользу определенного события. Перевес может быть записан как  $\frac{p}{(1-p)}$ , где  $p$  — вероятность положительного события. Термин “положительное событие” не обязательно означает “хорошее” событие, а имеет отношение к событию, которое мы хотим спрогнозировать, например, вероятность того, что у пациента имеется определенное заболевание; мы можем думать о положительном событии как о метке класса  $y = 1$ . Затем мы определим *логит-функцию* (*logit*), которая представляет собой просто логарифм перевеса:

$$\text{logit}(p) = \log \frac{p}{(1-p)}$$

Обратите внимание, что  $\log$  ссылается на натуральный логарифм, т.к. это распространенное соглашение в компьютерных науках. Логит-функция принимает входные значения в диапазоне от 0 до 1 и трансформирует их в значения по всему диапазону вещественных чисел, которые можно использовать для выражения линейной взаимосвязи между значениями признаков и логарифмом перевеса:

$$\text{logit}(p(y=1|\mathbf{x})) = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_ix_i = \mathbf{w}^T \mathbf{x}$$

Здесь  $p(y=1|\mathbf{x})$  — условная вероятность того, что специфический экземпляр принадлежит классу 1 при заданных признаках  $\mathbf{x}$ .

Нас фактически интересует прогнозирование вероятности того, что определенный образец принадлежит к конкретному классу, а это является обратной формой логит-функции. Она также называется *логистической сигмоидальной функцией*, а иногда просто *сигмоидальной функцией* из-за своей характерной S-образной формы:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Здесь  $z$  — общий вход, т.е. линейная комбинация весов и входов (признаков, ассоциированных с обучающими образцами):

$$z = \mathbf{w}^T \mathbf{x} = w_0x_0 + w_1x_1 + \dots + w_mx_m$$



На  
заметку!

Обратите внимание, что подобно принятому в главе 2 соглашению  $w_0$  означает член смещения, и мы еще предоставляем дополнительное входное значение  $x_0$ , которое установлено равным 1.

Давайте вычертим график сигмоидальной функции для значений в диапазоне от  $-7$  до  $7$ , чтобы посмотреть, как он выглядит:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> phi_z = sigmoid(z)
>>> plt.plot(z, phi_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\phi(z)$')
>>> # отметки и линия координатой сетки оси y
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.tight_layout()
>>> plt.show()
```

В результате выполнения приведенного выше кода мы получаем S-образную (сигмоидальную) кривую (рис. 3.2).

Мы можем заметить, что функция  $\phi(z)$  приближается к 1, если  $z$  стремится к бесконечности ( $z \rightarrow \infty$ ), т.к.  $e^{-z}$  становится очень малым для больших значений  $z$ . Подобным же образом  $\phi(z)$  стремится к 0 для  $z \rightarrow -\infty$  как результат все больше и больше увеличивающегося знаменателя. Подводя итоги, данная сигмоидальная функция принимает на входе вещественные числа и трансформирует их в значения из диапазона  $[0, 1]$  с точкой пересечения  $\phi(z) = 0.5$ .

Чтобы заложить основу для понимания логистической регрессионной модели, мы можем связать ее с моделью Adaline, представленной в главе 2. В качестве функции активации для Adaline мы применяли тождественное отображение  $\phi(z) = z$ . В логистической регрессии эта функция активации просто становится сигмоидальной функцией, которую мы определили ранее. На рис. 3.3 отражено отличие между Adaline и логистической регрессией.

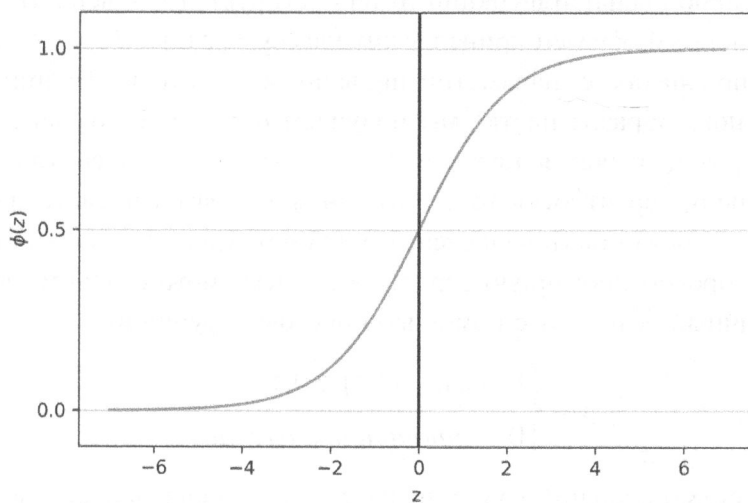


Рис. 3.2. График сигмоидальной функции для значений в диапазоне от  $-7$  до  $7$

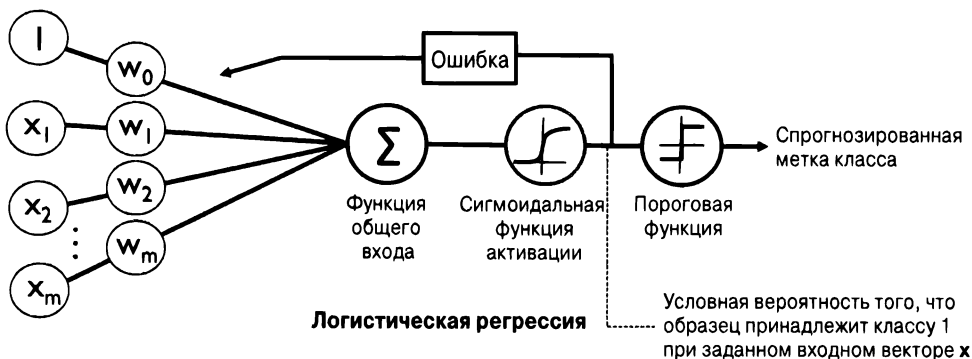
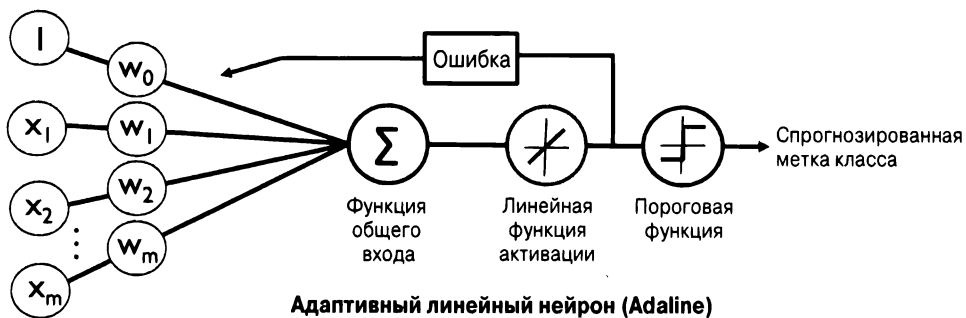


Рис. 3.3. Отличие между Adaline и логистической регрессией



Выход сигмоидальной функции интерпретируется как вероятность того, что определенный образец принадлежит классу 1,  $\phi(z) = P(y = 1 | x; \mathbf{w})$  при заданных признаках  $x$ , параметризованных весами  $\mathbf{w}$ . Например, если для отдельного образца цветка мы получаем  $\phi(z) = 0.8$ , то шанс того, что данный образец является цветком *Iris-versicolor*, составляет 80%. Следовательно, вероятность того, что данный образец представляет собой *Iris-setosa*, может быть вычислена как  $P(y=0 | x; \mathbf{w}) = 1 - P(y=1 | x; \mathbf{w}) = 0.2$ , или 20%. Спрогнозированную вероятность затем можно просто преобразовать в двоичный результат с помощью пороговой функции:

$$\hat{y} = \begin{cases} 1, & \text{если } \phi(z) \geq 0.5 \\ 0 & \text{в противном случае} \end{cases}$$

График сигмоидальной функции на рис. 3.2 эквивалентен следующему уравнению:

$$\hat{y} = \begin{cases} 1, & \text{если } z \geq 0.0 \\ 0 & \text{в противном случае} \end{cases}$$

На самом деле существует много приложений, где интересуют не только спрогнозированные метки классов, но и оценка вероятности членства в классе (выход сигмоидальной функции до применения пороговой функции). Скажем, когда логистическая регрессия используется для прогноза погоды, то с ее помощью не только прогнозируется, будет ли идти дождь в определенный день, но и сообщается, с какой вероятностью он пойдет. Кроме того, логистическая регрессия может применяться для прогнозирования шанса наличия у пациента определенной болезни при заданных симптомах, поэтому логистическая регрессия очень популярна в области медицины.

## Выяснение весов функции издержек для логистической регрессии

Ранее было показано, как использовать логистическую регрессионную модель для прогнозирования вероятностей и меток классов; теперь давайте кратко обсудим, каким образом подгонять параметры модели, например, веса  $\mathbf{w}$ . В предыдущей главе мы определяли функцию издержек как сумму квадратичных ошибок следующим образом:

$$J(\mathbf{w}) = \sum_i \frac{1}{2} \left( \phi(z^{(i)}) - y^{(i)} \right)^2$$

Мы минимизировали эту функцию с целью выяснения весов  $\mathbf{w}$  для классификационной модели Adaline. Чтобы пояснить, как можно вывести функцию издержек для логистической регрессии, первым делом определим *правдоподобие* (likelihood)  $L$ , которое мы хотим довести до максимума при построении логистической регрессионной модели с условием, что индивидуальные образцы в наборе данных независимы друг от друга. Вот соответствующая формула:

$$L(\mathbf{w}) = P(\mathbf{y} | \mathbf{x}; \mathbf{w}) = \prod_{i=1}^n P(y^{(i)} | \mathbf{x}^{(i)}; \mathbf{w}) = \prod_{i=1}^n \left( \phi(z^{(i)}) \right)^{y^{(i)}} \left( 1 - \phi(z^{(i)}) \right)^{1-y^{(i)}}$$

На практике легче довести до максимума (натуральный) логарифм этого уравнения, который называется функцией *логарифмического правдоподобия* (log-likelihood):

$$l(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^n \left[ y^{(i)} \log \left( \phi(z^{(i)}) \right) + (1 - y^{(i)}) \log \left( 1 - \phi(z^{(i)}) \right) \right]$$

Во-первых, применение логарифмической функции снижает возможность числовой потери значимости, которая может произойти, если величины правдоподобия очень малы. Во-вторых, мы можем преобразовать произведение сомножителей в сумму сомножителей, что облегчит получение производной этой функции посредством приема со сложением, который вы можете помнить из курса математики.

Далее мы могли бы воспользоваться алгоритмом оптимизации, таким как градиентный подъем, для доведения до максимума данной функции логарифмического правдоподобия. В качестве альтернативы давайте перепишем функцию логарифмического правдоподобия в виде функции издержек  $J$ , которую можно минимизировать с использованием градиентного спуска, как было показано в главе 2:

$$J(\mathbf{w}) = \sum_{i=1}^n \left[ -y^{(i)} \log \left( \phi(z^{(i)}) \right) - (1 - y^{(i)}) \log \left( 1 - \phi(z^{(i)}) \right) \right]$$

Чтобы лучше понять эту функцию издержек, рассмотрим издержки, подсчитанные для однократно выбранного обучающего образца:

$$J(\phi(z), y; \mathbf{w}) = -y \log(\phi(z)) - (1 - y) \log(1 - \phi(z))$$

Глядя на уравнение, мы можем заметить, что первый член обращается в ноль, если  $y = 0$ , а второй — если  $y = 1$ :

$$J(\phi(z), y; \mathbf{w}) = \begin{cases} -\log(\phi(z)), & \text{если } y = 1 \\ -\log(1 - \phi(z)), & \text{если } y = 0 \end{cases}$$

Давайте напишем короткий фрагмент кода построения графика, который проиллюстрирует издержки, связанные с классификацией одиночного обучающего образца для разных значений  $\phi(z)$ :

```
>>> def cost_1(z):
...     return - np.log(sigmoid(z))
>>> def cost_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> phi_z = sigmoid(z)
>>> c1 = [cost_1(x) for x in z]
>>> plt.plot(phi_z, c1, label='J(w), если y=1')
>>> c0 = [cost_0(x) for x in z]
>>> plt.plot(phi_z, c0, linestyle='--', label='J(w), если y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\phi(z)$')
>>> plt.ylabel('J(w)')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 3.4) по оси  $x$  изображена сигмоидальная активация в диапазоне от 0 до 1 (входы сигмоидальной функции, где значения  $z$  находятся в диапазоне от  $-10$  до  $10$ ), а по оси  $y$  — ассоциированные логистические издержки.

Легко заметить, что издержки приближаются к 0 (сплошная линия), если мы корректно прогнозируем, что образец относится к классу 1. Аналогично по оси  $y$  видно, что издержки также приближаются к 0, когда мы корректно прогнозируем  $y = 0$  (пунктирная линия). Однако если прогноз неправильный, тогда издержки стремятся к бесконечности. Суть в том, что мы штрафует за неправильные прогнозы все более и более высокими издержками.

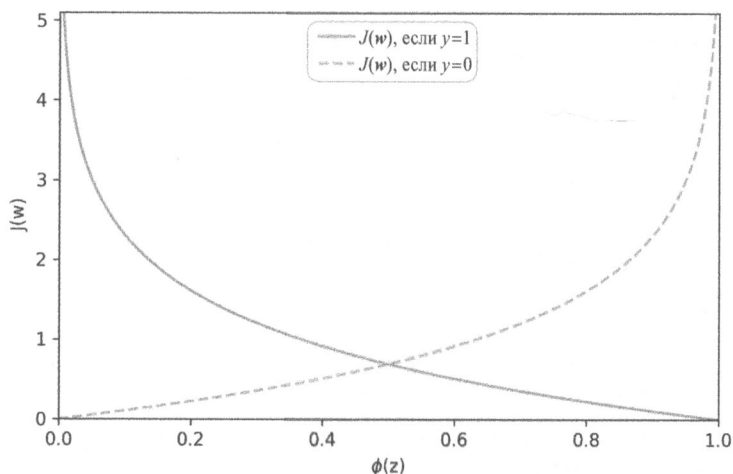


Рис. 3.4. Издержки, связанные с классификацией однократно выбранного образца для разных значений  $\phi(z)$

## Преобразование реализации Adaline в алгоритм для логистической регрессии

Если бы мы реализовывали логистическую регрессию самостоятельно, то могли бы заменить функцию издержек  $J$  в реализации Adaline из главы 2 новой функцией издержек:

$$J(\mathbf{w}) = -\sum_i y^{(i)} \log(\phi(z^{(i)})) + (1 - y^{(i)}) \log(1 - \phi(z^{(i)}))$$

Приведенная выше функция применяется для вычисления издержек от классификации всех обучающих образцов за эпоху. Кроме того, нам необходимо поменять линейную функцию активации на сигмоидальную функцию активации и изменить пороговую функцию с целью возвращения меток классов 0 и 1 вместо  $-1$  и  $1$ . Внеся указанные изменения в код реализации Adaline, мы получим работающую реализацию логистической регрессии:

```
class LogisticRegressionGD(object):
    """Классификатор на основе логистической регрессии,
        использующий градиентный спуск.

    Параметры
    -----
    eta : float
        Скорость обучения (между 0.0 и 1.0).
```

```
n_iter : int
    Проходы по обучающему набору данных.
random_state : int
    Начальное значение генератора случайных чисел
    для инициализации случайными весами.

Атрибуты
-----

w_ : одномерный массив
    Веса после подгонки.
cost_ : list
    Значение логистической функции издержек в каждой эпохе.
"""

def __init__(self, eta=0.05, n_iter=100, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

def fit(self, X, y):
    """Подгоняет к обучающим данным.

    Параметры
    -----
    X : {подобен массиву}, форма = [n_examples, n_features]
        Обучающие векторы, где n_examples - количество образцов,
        n_features - количество признаков.
    y : подобен массиву, форма = [n_examples]
        Целевые значения.

    Возвращает
    -----
    self : object
    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01,
                          size=1 + X.shape[1])
    self.cost_ = []

    for i in range(self.n_iter):
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
```

```

        # обратите внимание, что теперь мы вычисляем
        # логистические 'издержки', а не издержки в виде
        # суммы квадратичных ошибок
        cost = (-y.dot(np.log(output)) -
                ((1 - y).dot(np.log(1 - output))))
        self.cost_.append(cost)
    return self

def net_input(self, X):
    """Вычисляет общий вход"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def activation(self, z):
    """Вычисляет логистическую сигмоидальную активацию"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def predict(self, X):
    """Возвращает метку класса после единичного шага"""
    return np.where(self.net_input(X) >= 0.0, 1, 0)
    # эквивалентно:
    # return np.where(self.activation(self.net_input(X))
    #                 >= 0.5, 1, 0)

```

При подгонке логистической регрессионной модели мы должны иметь в виду, что она работает только для задач двоичной классификации.

Итак, давайте примем во внимание только цветки *Iris-setosa* и *Iris-versicolor* (классы 0 и 1) и проверим, работает ли наша реализация логистической регрессии:

```

>>> X_train_01_subset = X_train[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = LogisticRegressionGD(eta=0.05,
...                             n_iter=1000,
...                             random_state=1)
>>> lrgd.fit(X_train_01_subset,
...          y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset,
...                       y=y_train_01_subset,
...                       classifier=lrgd)
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

```

Результирующий график области решений показан на рис. 3.5.

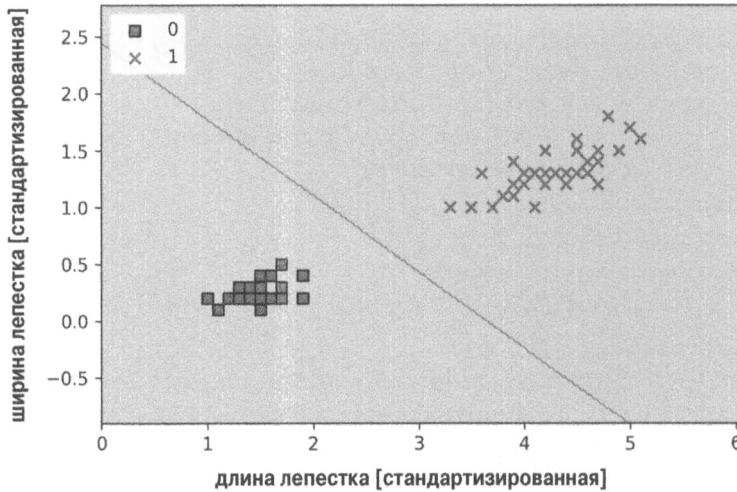


Рис. 3.5. Результат подгонки логистической регрессионной модели



На заметку!

### Алгоритм обучения на основе градиентного спуска для логистической регрессии

Используя математику, мы можем показать, что обновление весов логистической регрессии через градиентный спуск делается идентично уравнению, которое мы применяли в главе 2. Тем не менее, обратите внимание, что демонстрируемое ниже выведение правила обучения на основе градиентного спуска предназначено для читателей, которые интересуются математическими концепциями, лежащими в основе этого правила для логистической регрессии. Оно не является существенно важным для понимания оставшихся материалов главы.

Начнем с вычисления частной производной функции логарифмического правдоподобия относительно  $j$ -того веса:

$$\frac{\partial}{\partial w_j} l(\mathbf{w}) = \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z)$$

Прежде чем продолжить, давайте вычислим также и частную производную сигмоидальной функции:

$$\begin{aligned} \frac{\partial}{\partial z} \phi(z) &= \frac{\partial}{\partial z} \frac{1}{1+e^{-z}} = \frac{1}{(1+e^{-z})^2} e^{-z} = \frac{1}{1+e^{-z}} \left( 1 - \frac{1}{1+e^{-z}} \right) = \\ &= \phi(z)(1-\phi(z)) \end{aligned}$$

Теперь мы можем подставить  $\frac{\partial}{\partial z} \phi(z) = \phi(z)(1 - \phi(z))$  в наше первое уравнение, получив следующее:

$$\begin{aligned} & \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\partial}{\partial w_j} \phi(z) = \\ & = \left( y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \phi(z)(1-\phi(z)) \frac{\partial}{\partial w_j} z = \\ & = (y(1-\phi(z)) - (1-y)\phi(z)) x_j = \\ & = (y - \phi(z)) x_j \end{aligned}$$

Вспомним, что цель заключается в том, чтобы отыскать веса, которые доводят до максимума логарифмическое правдоподобие, поэтому мы выполняем обновление для каждого веса, как показано далее:

$$w_j := w_j + \eta \sum_{i=1}^n (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Поскольку мы обновляем все веса одновременно, то можем записать общее правило обновления такого вида:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w},$$

Мы определяем  $\Delta \mathbf{w}$  следующим образом:

$$\Delta \mathbf{w} = \eta \nabla l(\mathbf{w})$$

Так как доведение до максимума логарифмического правдоподобия эквивалентно минимизации определенной ранее функции издержек  $J$ , мы можем записать правило обновления на основе градиентного спуска, как показано ниже:

$$\begin{aligned} \Delta w_j &= -\eta \frac{\partial J}{\partial w_j} = \eta \sum_{i=1}^n (y^{(i)} - \phi(z^{(i)})) x_j^{(i)} \\ \mathbf{w} &:= \mathbf{w} + \Delta \mathbf{w}, \quad \Delta \mathbf{w} = -\eta \nabla J(\mathbf{w}) \end{aligned}$$

Данное правило равносильно правилу градиентного спуска для Adaline в главе 2.



## Обучение логистической регрессионной модели с помощью *scikit-learn*

В предыдущем подразделе мы проделали полезные упражнения по кодированию и математическим выкладкам, которые помогли проиллюстрировать концептуальные отличия между Adaline и логистической регрессией. А теперь давайте выясним, как использовать более оптимизированную реализацию логистической регрессии из библиотеки *scikit-learn*, которая также поддерживает многоклассовую конфигурацию. Обратите внимание, что в последних версиях *scikit-learn* методика, применяемая для многоклассовой классификации (полиномиальная логистическая регрессия или OvR), выбирается автоматически. В следующем примере кода мы будем применять класс `sklearn.linear_model.LogisticRegression`, а также знакомый метод `fit` для обучения модели на всех трех классах в стандартизированном обучающем наборе *Iris*. Кроме того, в целях иллюстрации мы устанавливаем `multi_class='ovr'`. В качестве упражнения для самостоятельной проработки можете сравнить результаты с теми, которые получаются при установке `multi_class='multinomial'`. Имейте в виду, что на практике настройка `multinomial` обычно рекомендуется для взаимно исключающих классов вроде находящихся в наборе данных *Iris*. Здесь “взаимно исключающие” означает, что каждый обучающий образец может принадлежать только одному классу (в противоположность *многозначной (multilabel)* классификации, где обучающий образец может быть членом множества классов).

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, random_state=1,
...                         solver='lbfgs', multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=lr,
...                       test_idx=range(105, 150))
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

После подгонки модели к обучающим данным мы вычерчиваем области решений, обучающие образцы и испытательные образцы (рис. 3.6).

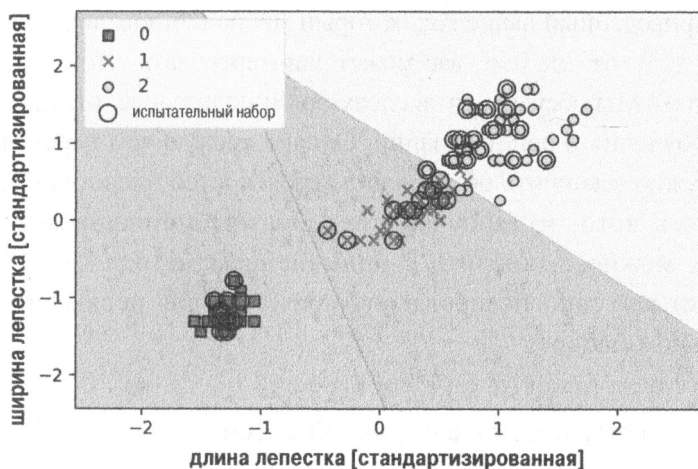


Рис. 3.6. Результат подгонки модели посредством *scikit-learn*



Обратите внимание, что существует много различных алгоритмов оптимизации для решения оптимизационных задач. Чтобы свести к минимуму выпуклую функцию потерь, такую как потери логистической регрессии, рекомендуется использовать более развитые методики, нежели обыкновенный стохастический градиентный спуск (SGD). На самом деле в *scikit-learn* реализован целый ряд таких алгоритмов оптимизации, которые можно указывать посредством параметра `solver`, в частности, `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'` и `'saga'`.

Наряду с тем, что функция потерь логистической регрессии выпуклая, большинство алгоритмов оптимизации должны с легкостью сходиться в глобальном минимуме потерь. Однако существуют определенные преимущества применения одного алгоритма перед другими. Например, в текущей версии (0.21) библиотеки *scikit-learn* по умолчанию используется решатель `'liblinear'`, который не способен обрабатывать полиномиальные потери и ограничен схемой OvR для многоклассовой классификации. Тем не менее, в будущих версиях *scikit-learn* (т.е. 0.22) стандартный решатель изменится на `'lbfgs'`, который означает алгоритм Бройдена–Флетчера–Гольдфарба–Шанно с ограниченной памятью (*limited-memory Broyden–Fletcher–Goldfarb–Shanno* — BFGS; [https://ru.wikipedia.org/wiki/Алгоритм\\_Бройдена\\_—\\_Флетчера\\_—\\_Гольдфарба\\_—\\_Шанно](https://ru.wikipedia.org/wiki/Алгоритм_Бройдена_—_Флетчера_—_Гольдфарба_—_Шанно)) и в этом отношении он более гибкий. Чтобы принять такой новый стандартный вариант, повсюду в книге в случае применения логистической регрессии мы будем явно указывать `solver='lbfgs'`.

Глядя на приведенный выше код, который мы использовали для обучения модели `LogisticRegression`, вас может заинтересовать, что это за таинственный параметр `C`? Мы обсудим его в следующем подразделе, где представим концепции переобучения и регуляризации. Однако прежде чем переходить к таким темам, нам нужно закончить обсуждение вероятностей членства в классах.

Вероятность того, что обучающие образцы принадлежат определенному классу, можно вычислить с применением метода `predict_proba`. Например, вот как спрогнозировать вероятности для первых трех образцов в испытательном наборе:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

В результате возвращается следующий массив:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

Первая строка соответствует вероятностям членства в классах первого цветка, вторая строка — вероятностям членства в классах второго цветка и т.д. Обратите внимание, что значения столбцов в строке в сумме дают единицу, как и ожидалось. (Проверить сказанное можно, выполнив `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`.)

Самое высокое значение в первой строке составляет приблизительно 0.85, т.е. первый образец принадлежит третьему классу (*Iris-virginica*) со спрогнозированной вероятностью 85%. Следовательно, как вы уже наверняка заметили, мы можем извлечь спрогнозированные метки классов, идентифицируя в каждой строке столбец с наибольшим значением, например, с использованием функции `argmax` из `NumPy`:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Ниже показаны возвращенные индексы классов (они соответствуют *Iris-virginica*, *Iris-setosa* и *Iris-setosa*):

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

В предыдущем примере кода мы вычисляли условные вероятности и вручную преобразовывали их в метки классов с помощью функции `argmax` из `NumPy`. Более удобный способ получения меток классов при работе с библиотекой `scikit-learn` предусматривает прямой вызов метода `predict`:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

И в заключение одно предостережение на случай, если желательно прогнозировать метку класса для одиночного образца цветка: библиотека *scikit-learn* ожидает получить в качестве входных данных двумерный массив; таким образом, нужную строку придется предварительно преобразовать в такой формат. Преобразовать одиночную строку в двумерный массив данных можно с применением метода `reshape` из NumPy, добавив новое измерение:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

## Решение проблемы переобучения с помощью регуляризации

*Переобучение* — распространенная проблема в МО, когда модель хорошо работает на обучающих данных, но плохо обобщается на не встречавшиеся ранее (испытательные) данные. Если модель страдает от переобучения, то мы также говорим, что она имеет высокую дисперсию, которая может быть связана с наличием слишком большого числа параметров, приводящих к получению чересчур сложной модели для лежащих в основе данных. Подобным же образом модель может также страдать от *недообучения* (*underfitting*), или высокого смещения, которое означает, что модель недостаточно сложна для того, чтобы хорошо выявлять структуру в обучающих данных, из-за чего она обладает низкой эффективностью на не встречавшихся ранее данных.

Хотя до сих пор мы встречались только с линейными моделями для классификации, проиллюстрировать проблемы переобучения и недообучения лучше всего путем сравнения линейной границы решений с более сложными нелинейными границами решений, как показано на рис. 3.7.

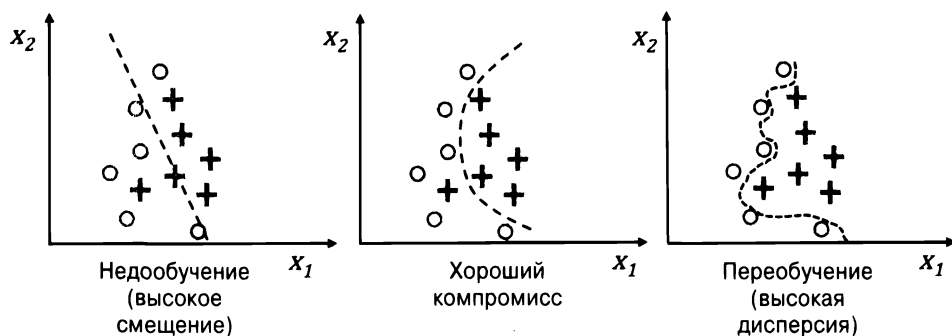


Рис. 3.7. Иллюстрация проблем переобучения и недообучения



### Компромисс между смещением и дисперсией

При описании эффективности модели исследователи часто используют термины “смещение” и “дисперсия” или “компромисс между смещением и дисперсией” — т.е. в беседах, книгах или статьях вы можете встретить заявления о том, что модель имеет “высокую дисперсию” или “высокое смещение”. Что же это означает? В целом мы можем говорить, что “высокая дисперсия” пропорциональна переобучению, а “высокое смещение” — недообучению.

В контексте моделей МО *дисперсия* измеряет постоянство (либо изменчивость) прогноза модели для классификации отдельного образца при многократном обучении модели, например, на разных подмножествах обучающего набора данных. Мы можем сказать, что модель чувствительна к случайности обучающих данных. Напротив, *смещение* измеряет, насколько далеко прогнозы находятся от корректных значений в целом при многократном обучении модели на разных обучающих наборах данных; смещение представляет собой меру систематической ошибки, которая не является результатом случайности.

Если вас интересует техническое описание и происхождение терминов “смещение” и “дисперсия”, тогда обратитесь к конспекту лекций: [https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/08\\_eval-intro\\_notes.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/08_eval-intro_notes.pdf).

Один из способов отыскания хорошего компромисса между смещением и дисперсией предусматривает настройку сложности модели посредством регуляризации. *Регуляризация* — очень полезный метод для обработки коллинеарности (сильной взаимосвязи между признаками), фильтрации шума из данных и в итоге предотвращения переобучения.

Концепция, положенная в основу регуляризации, заключается в том, чтобы вводить дополнительную информацию (смещение) для штрафования экстремальных значений параметров (весов). Самой распространенной формой регуляризации является так называемая *регуляризация L2* (иногда также называемая сокращением L2 или ослаблением весов), которую можно записать следующим образом:

$$\frac{\lambda}{2} \|\mathbf{w}\|^2 = \frac{\lambda}{2} \sum_{j=1}^m w_j^2$$

Здесь  $\lambda$  — параметр регуляризации.



## Регуляризация и нормализация признаков

Регуляризация — еще одна причина важности масштабирования признаков, такого как стандартизация. Для надлежащей работы регуляризации мы должны обеспечить наличие у всех признаков соизмеримых масштабов.

Функцию издержек для логистической регрессии можно регуляризовать, добавив простой член регуляризации, который будет сокращать веса во время обучения модели:

$$J(\mathbf{w}) = \sum_{i=1}^n \left[ -y^{(i)} \log(\phi(z^{(i)})) - (1 - y^{(i)}) \log(1 - \phi(z^{(i)})) \right] + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

Через параметр регуляризации  $\lambda$  мы можем управлять тем, насколько хорошо происходит подгонка к обучающим данным, одновременно сохраняя веса низкими. Увеличивая значение  $\lambda$ , мы увеличиваем силу регуляризации.

Параметр  $C$ , предусмотренный для класса `LogisticRegression` в библиотеке `scikit-learn`, происходит из соглашения, принятого в методах опорных векторов, которые будут темой следующего раздела. Член  $C$  напрямую связан с параметром регуляризации  $\lambda$ , являясь его инверсией. Следовательно, уменьшение значения инверсного параметра регуляризации  $C$  означает увеличение силы регуляризации, что можно визуализировать, построив график пути регуляризации  $L2$  для двух весовых коэффициентов:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c, random_state=1,
...                             solver='lbfgs', multi_class='ovr')
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='длина лепестка')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='ширина лепестка')
>>> plt.ylabel('весовой коэффициент')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

В результате выполнения показанного выше кода мы выполнили подгонку 10 логистических регрессионных моделей с разными инверсными параметрами регуляризации  $C$ . В демонстрационных целях мы собрали только весовые коэффициенты класса 1 (второго класса в наборе данных, *Iris-versicolor*) по отношению ко всем классификаторам — не забывайте, что мы применяем методику OvR для многоклассовой классификации.

Как видно на результирующем графике (рис. 3.8), в случае уменьшения параметра  $C$  весовые коэффициенты сокращаются, т.е. сила регуляризации увеличивается.

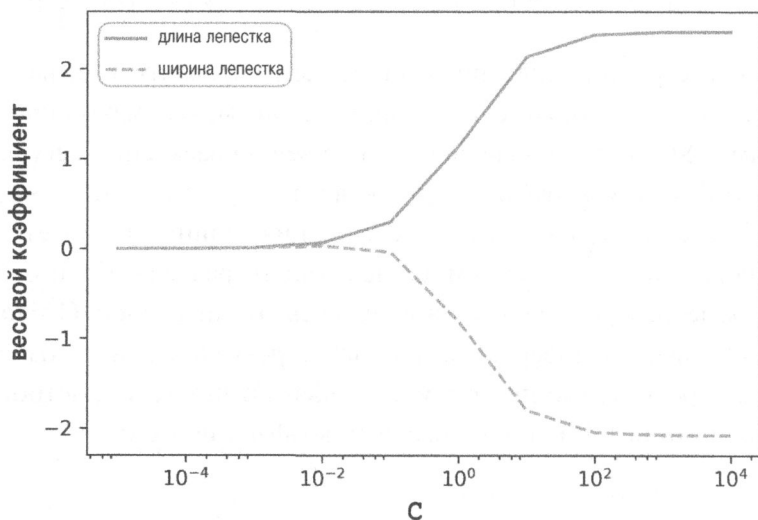


Рис. 3.8. Путь регуляризации  $L2$  для двух весовых коэффициентов



### Дополнительные ресурсы по логистической регрессии

Поскольку исчерпывающее раскрытие индивидуальных алгоритмов классификации выходит за рамки настоящей книги, тем читателям, которые заинтересованы в дальнейшем изучении логистической регрессии, настоятельно рекомендуется обратиться к книге Скотта Менаарда “Logistic Regression: From Introductory to Advanced Concepts and Applications” (Логистическая регрессия: от введения до расширенных концепций и приложений), Sage Publications (2009 г.).

## Классификация с максимальным зазором с помощью методов опорных векторов

Еще одним мощным и широко используемым алгоритмом обучения является *метод опорных векторов* (*Support Vector Machine* — *SVM*), который можно считать расширением персептрона. С применением алгоритма персептрона мы сводим к минимуму ошибки неправильной классификации. Тем не менее, в методах SVM цель оптимизации — довести до максимума зазор. Зазор определяется как расстояние между разделяющей гиперплоскостью (границей решений) и ближайшими к этой гиперплоскости обучающими образцами, которые называются *опорными векторами*. Сказанное проиллюстрировано на рис. 3.9.

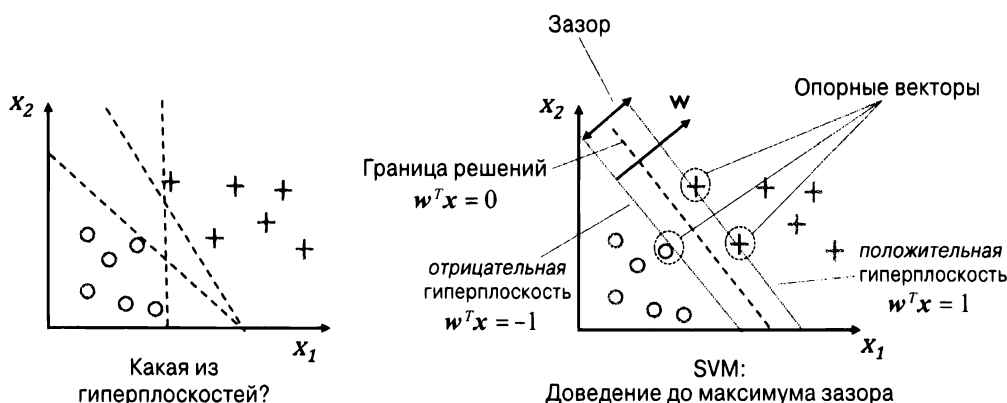


Рис. 3.9. Метод опорных векторов

### Понятие максимального зазора

Логическое обоснование наличия границ решений с широкими зазорами заключается в том, что такие модели обычно имеют меньшую ошибку обобщения, тогда как модели с маленькими зазорами больше предрасположены к переобучению. Чтобы получить представление о доведении до максимума зазора, давайте пристальнее взглянем на те положительные и отрицательные гиперплоскости, которые параллельны границе решений и могут быть выражены следующим образом:

$$w_0 + w^T x_{\text{положительная}} = 1 \quad (1)$$

$$w_0 + w^T x_{\text{отрицательная}} = -1 \quad (2)$$



Если мы вычтем два линейных уравнения (1) и (2) друг из друга, то получим:

$$\Rightarrow \mathbf{w}^T (\mathbf{x}_{\text{положительная}} - \mathbf{x}_{\text{отрицательная}}) = 2$$

Мы можем нормализовать это уравнение по длине вектора  $\mathbf{w}$ , которая определяется так:

$$\|\mathbf{w}\| = \sqrt{\sum_{j=1}^m w_j^2}$$

В итоге мы приходим к представленному ниже уравнению:

$$\frac{\mathbf{w}^T (\mathbf{x}_{\text{положительная}} - \mathbf{x}_{\text{отрицательная}})}{\|\mathbf{w}\|} = \frac{2}{\|\mathbf{w}\|}$$

Затем мы можем интерпретировать левую часть последнего уравнения как расстояние между положительной и отрицательной гиперплоскостями, которое также называется *зазором*, подлежащим доведению до максимума.

Теперь целевой функцией SVM становится доведение до максимума зазора путем максимизации  $\frac{2}{\|\mathbf{w}\|}$  при условии корректной классификации образцов, что может быть записано следующим образом:

$$\begin{aligned} w_0 + \mathbf{w}^T \mathbf{x}^{(i)} &\geq 1, \text{ если } y^{(i)} = 1 \\ w_0 + \mathbf{w}^T \mathbf{x}^{(i)} &\leq -1, \text{ если } y^{(i)} = -1 \\ &\text{для } i = 1 \dots N \end{aligned}$$

Здесь  $N$  — количество образцов в нашем наборе данных.

Два приведенных уравнения по существу говорят о том, что все образцы отрицательного класса должны находиться с одной стороны отрицательной гиперплоскости, а все образцы положительного класса быть позади положительной гиперплоскости, что можно записать более компактно:

$$y^{(i)} (w_0 + \mathbf{w}^T \mathbf{x}^{(i)}) \geq 1 \quad \forall_i$$

Однако на практике легче минимизировать обратный член  $\frac{1}{2} \|\mathbf{w}\|^2$ , что можно сделать посредством квадратичного программирования. Тем не менее, детальное обсуждение квадратичного программирования выходит за рамки книги. Дополнительные сведения о методах опорных векторов можно

найти в книге Владимира Вапника “The Nature of Statistical Learning Theory” (“Природа теории статистического обучения”), Springer Science+Business Media (2000 год) или в великолепной статье Криса Бурга “A Tutorial on Support Vector Machines for Pattern Recognition” (“Руководство по методам опорных векторов для распознавания образов”), Data Mining and Knowledge Discovery, 2(2): с. 121–167 (1998 г.).

## Обработка нелинейно сепарабельного случая с использованием фиктивных переменных

Хотя мы не намерены погружаться слишком глубоко в более сложные математические концепции, лежащие в основе классификации с максимальным зазором, давайте кратко упомянем о *фиктивной переменной* (*slack variable*)  $\xi$ , которая в 1995 году была введена Владимиром Вапником и привела к появлению так называемой *классификации с мягким зазором*. Мотивом введения фиктивной переменной оказались линейные ограничения, которые нужно было ослабить для нелинейно сепарабельных данных, чтобы сделать возможной сходимость оптимизации при наличии неправильных классификаций в условиях надлежащего штрафования издержек.

Фиктивная переменная с положительными значениями просто присоединяется к линейным ограничениям:

$$\begin{aligned} w_0 + \mathbf{w}^T \mathbf{x}^{(i)} &\geq 1 - \xi^{(i)}, \text{ если } y^{(i)} = 1 \\ w_0 + \mathbf{w}^T \mathbf{x}^{(i)} &\leq -1 + \xi^{(i)}, \text{ если } y^{(i)} = -1 \\ &\text{для } i = 1 \dots N \end{aligned}$$

Здесь  $N$  — количество образцов в нашем наборе данных. Таким образом, новой целью для минимизации (предметом ограничений) становится:

$$\frac{1}{2} \|\mathbf{w}\|^2 + C \left( \sum_i \xi^{(i)} \right)$$

Через переменную  $C$  мы можем впоследствии управлять штрафом за неправильную классификацию. Крупные значения  $C$  соответствуют крупным штрафам за ошибки, тогда как выбор меньших значений для  $C$  означает меньшую строгость в отношении ошибок неправильной классификации. Параметр  $C$  затем можно применять для управления шириной зазора и, следовательно, настраивать компромисс между смещением и дисперсией, как демонстрируется на рис. 3.10.

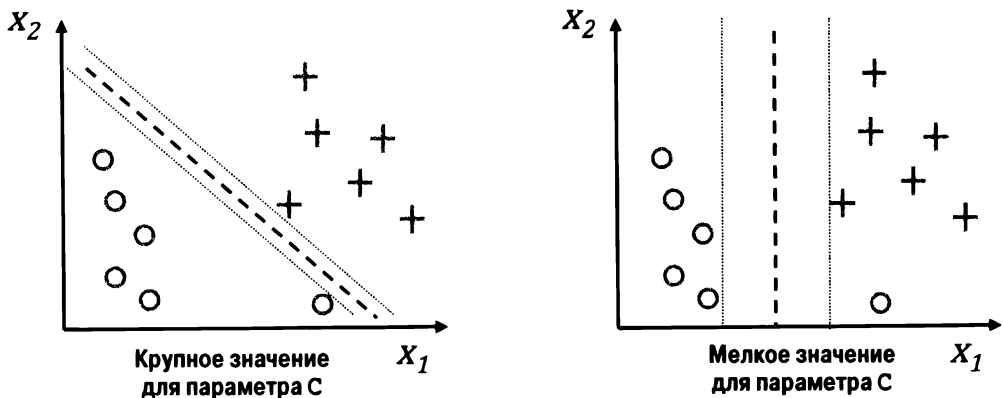


Рис. 3.10. Управление шириной зазора с помощью параметра  $C$

Эта концепция связана с регуляризацией, которую мы обсуждали в предыдущем разделе в контексте регуляризированной регрессии, где уменьшение значения  $C$  увеличивает смещение и понижает дисперсию модели.

Ознакомившись с базовыми концепциями, лежащими в основе линейного SVM, давайте обучим модель SVM для классификации различных цветков в наборе данных Iris:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=svm,
...                       test_idx=range(105, 150))
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 3.11 показаны три области решений SVM, визуализированные после обучения классификатора на наборе данных путем выполнения приведенного выше кода.

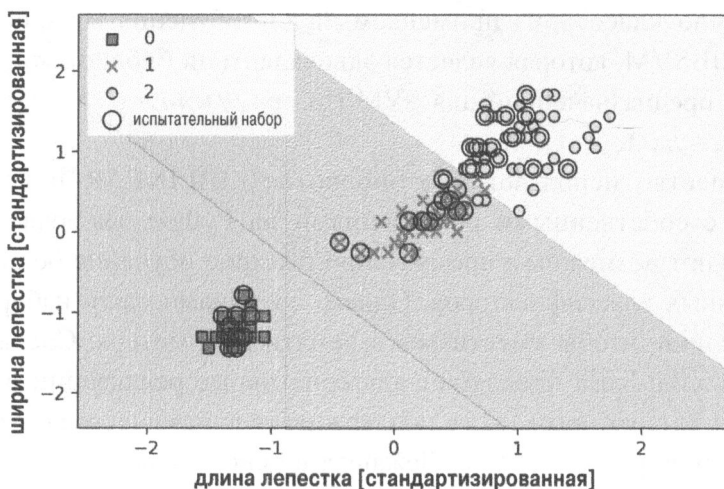


Рис. 3.11. Области решений после обучения классификатора на наборе данных *Iris*



На заметку!

### Логистическая регрессия в сравнении с методами опорных векторов

Для практических задач классификации линейная логистическая регрессия и линейные SVM часто выдают очень похожие результаты. Логистическая регрессия пытается довести до максимума условные правдоподобия обучающих данных, что делает их в большей степени подверженными выбросам, чем SVM, которые главным образом заботятся о точках, ближайших к границе решений (опорных векторах). С другой стороны, преимущество логистической регрессии связано с тем, что она является более простой моделью и ее легче реализовать. Кроме того, логистические регрессионные модели можно легко обновлять, что привлекательно при работе с потоковыми данными.

## Альтернативные реализации в *scikit-learn*

Класс `LogisticRegression` из библиотеки *scikit-learn*, которые мы использовали в предшествующих разделах, задействуют `LIBLINEAR` — высокооптимизированную библиотеку на C/C++, разработанную в Национальном университете Тайваня (<http://www.csie.ntu.edu.tw/~cjlin/liblinear/>).

Аналогично класс `SVC`, применяемый для обучения модели SVM, задействует `LIBSVM`, которая является эквивалентной библиотекой на C/C++, специально предназначенной для SVM (<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>).

Преимущество использования библиотек `LIBLINEAR` и `LIBSVM` по сравнению с собственными реализациями на Python заключается в том, что они делают возможным чрезвычайно быстрое обучение больших количеств линейных классификаторов. Однако временами наши наборы данных слишком велики, чтобы уместиться в памяти компьютера. Соответственно библиотека *scikit-learn* предлагает альтернативные реализации через класс `SGDClassifier`, который также поддерживает динамическое обучение посредством метода `partial_fit`. Лежащая в основе класса `SGDClassifier` концепция подобна концепции алгоритма стохастического градиентного спуска, который мы реализовали в главе 2 для *Adaline*. Мы могли бы инициализировать основанную на стохастическом градиентном спуске версию перцептрона, логистической регрессии и метода опорных векторов со стандартными параметрами следующим образом:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

## Решение нелинейных задач с применением ядерного метода опорных векторов

Еще одна причина высокой популярности SVM у специалистов-практиков МО связана с тем, что методы опорных векторов могут быть легко *параметрически редуцированы* (*kernelized*) для решения задач нелинейной классификации. Прежде чем обсуждать главную концепцию, положенную в основу так называемого *ядерного SVM* (*kernel SVM*), давайте сначала создадим синтетический набор данных, чтобы выяснить, на что похожа задача нелинейной классификации подобного рода.

### Ядерные методы для линейно сепарабельных данных

С помощью следующего кода мы создадим простой набор данных в форме логического вентиля XOR (“исключающее ИЛИ”), используя для этого

функцию `logical_or` из NumPy, где 100 образцам будет назначена метка класса 1, а 100 образцам — метка класса -1:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> np.random.seed(1)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0,
...                        X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, -1)
>>> plt.scatter(X_xor[y_xor == 1, 0],
...            X_xor[y_xor == 1, 1],
...            c='b', marker='x',
...            label='1')
>>> plt.scatter(X_xor[y_xor == -1, 0],
...            X_xor[y_xor == -1, 1],
...            c='r',
...            marker='s',
...            label='-1')
>>> plt.xlim([-3, 3])
>>> plt.ylim([-3, 3])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

В результате выполнения кода мы получим набор данных XOR со случайным шумом (рис. 3.12).

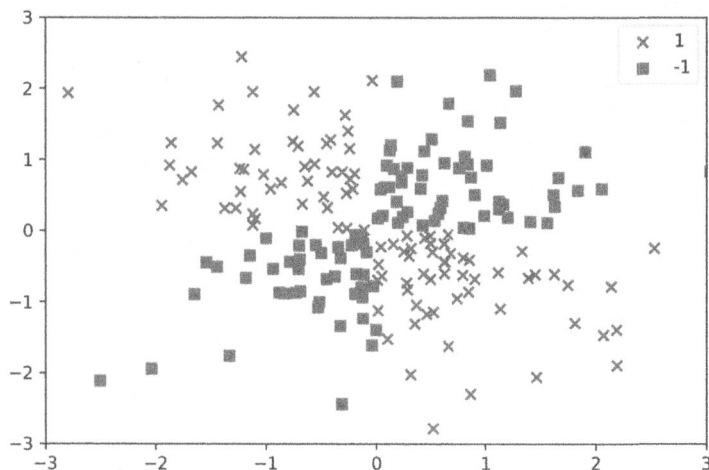


Рис. 3.12. Набор данных XOR со случайным шумом

Очевидно, что с применением линейной гиперплоскости в качестве границы решений нам не удастся очень хорошо отделить образцы положительного класса от образцов отрицательного класса, используя модель на основе линейной логистической регрессии или линейного SVM, которая обсуждалась в предшествующих разделах.

Базовая идея *ядерных методов*, предназначенных для работы с такими линейно несепарабельными данными, заключается в создании нелинейных комбинаций исходных признаков с целью их проецирования на пространство более высокой размерности через отображающую функцию  $\phi$ , где они становятся линейно сепарабельными. Посредством следующей проекции мы можем трансформировать двумерный набор данных в новое трехмерное пространство признаков, внутри которого классы становятся сепарабельными:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

В результате мы получаем возможность разделения двух классов (рис. 3.13) через линейную гиперплоскость, которая превращается в нелинейную границу решений при ее проецировании обратно на исходное пространство признаков.

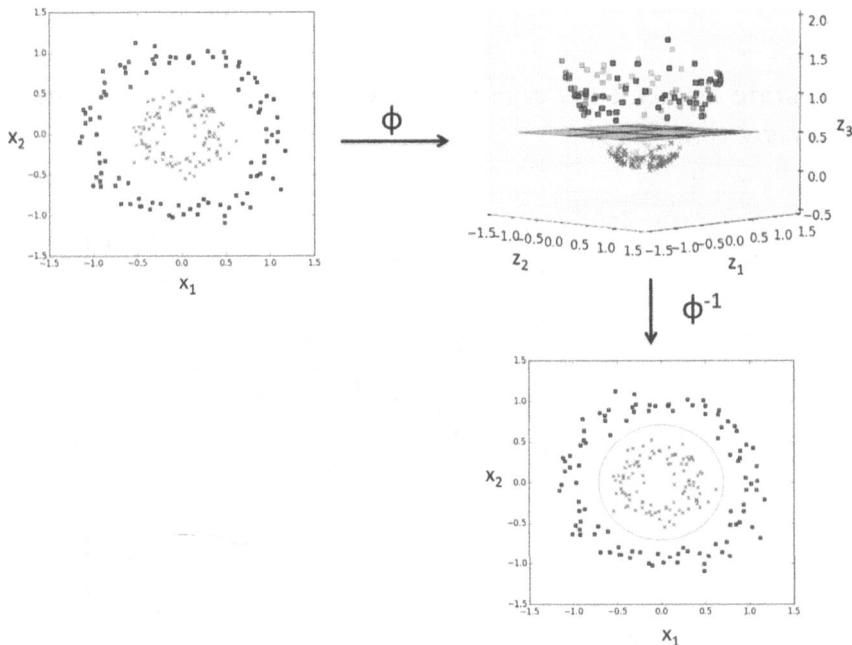


Рис. 3.13. Пример работы ядерного метода

## Использование ядерного трюка для нахождения разделяющих гиперплоскостей в пространстве высокой размерности

Чтобы решить нелинейную задачу с применением SVM, мы могли бы с помощью отображающей функции  $\phi$  трансформировать обучающие данные в пространство признаков более высокой размерности и обучить линейную модель SVM для классификации данных в новом пространстве признаков. Затем посредством той же самой отображающей функции  $\phi$  мы трансформировали бы новые, не встречавшиеся ранее данные для их классификации с использованием линейной модели SVM.

Тем не менее, этому подходу с отображением присуща одна проблема: конструирование новых признаков сопряжено с очень высокими вычислительными затратами, особенно если мы имеем дело с данными высокой размерности. Именно здесь в игру вступает то, что называется *ядерным трюком* (*kernel trick*). Хотя мы не будем вдаваться в мельчайшие подробности о том, как решать задачу квадратичного программирования для обучения SVM, на практике необходимо лишь заменить скалярное произведение  $\mathbf{x}^{(i)T} \mathbf{x}^{(j)}$  произведением  $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$ . Чтобы сэкономить на затратном шаге явного вычисления скалярного произведения двух точек, мы определяем так называемую *ядерную функцию* (*kernel function*):

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$$

Одним из самых широко применяемых ядер является ядро на основе *радиальной базисной функции* (*Radial Basis Function* — *RBF*), которое называют просто *гауссовым ядром*:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$$

Оно часто упрощается до:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Здесь  $\gamma = \frac{1}{2\sigma^2}$  — свободный параметр, подлежащий оптимизации.



Грубо говоря, термин “ядро” может интерпретироваться как *функция близости (similarity function)* между парой образцов. Знак “минус” обращает меру расстояния в показатель близости, и благодаря экспоненциальному члену результирующий показатель близости попадет в диапазон между 1 (для вполне близких образцов) и 0 (для очень несходных образцов).

Теперь, когда мы раскрыли общую картину с ядерным трюком, давайте посмотрим, сумеем ли мы обучить ядерный SVM, способный провести нелинейную границу решений, которая хорошо разделит данные XOR. Мы просто воспользуемся ранее импортированным классом SVC из библиотеки *scikit-learn*, заменив параметр `kernel='linear'` параметром `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике видно, что ядерный SVM разделяет данные XOR относительно хорошо (рис. 3.14).

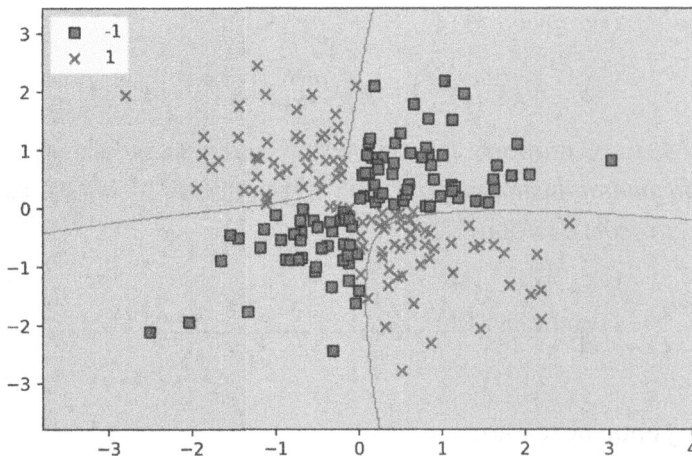


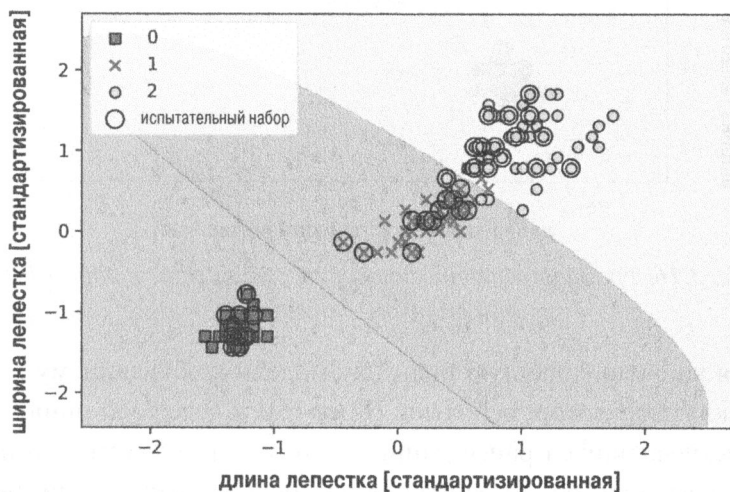
Рис. 3.14. Разделение данных XOR с помощью ядерного SVM

Параметр  $\gamma$ , установленный как `gamma=0.1`, можно считать параметром отсечения для гауссовой сферы. Если мы увеличим значение  $\gamma$ , то тем самым усилим влияние или охват обучающих образцов, что приведет к более плотной и бугристой границе решений.

Чтобы лучше понять смысл параметра  $\gamma$ , применим метод SVM с ядром RBF к нашему набору данных Iris:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Поскольку мы выбрали для  $\gamma$  сравнительно небольшое значение, результирующая граница решений модели на основе SVM с ядром RBF будет относительно мягкой, как показано на рис. 3.15.



**Рис. 3.15.** Граница решений модели на основе SVM с ядром RBF при низком значении  $\gamma$

А теперь давайте увеличим значение  $\gamma$  и понаблюдаем за его воздействием на границу решений:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
```

```
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике мы можем заметить, что при относительно большом значении  $\gamma$  граница решений между классами 0 и 1 оказывается гораздо компактнее (рис. 3.16).

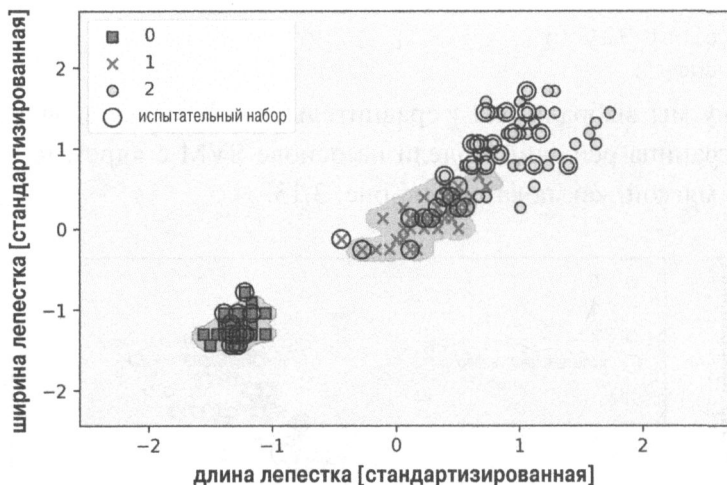


Рис. 3.16. Граница решений модели на основе SVM с ядром RBF при высоком значении  $\gamma$

Несмотря на очень хорошую подгонку модели к обучающему набору данных, такой классификатор, вероятно, будет иметь высокую ошибку обобщения на не встречавшихся ранее данных. Это говорит о том, что параметр  $\gamma$  также играет важную роль в контроле над переобучением или дисперсией, когда алгоритм слишком чувствителен к колебаниям в обучающем наборе.

## Обучение моделей на основе деревьев принятия решений

Классификаторы на основе *деревьев принятия решений* являются привлекательными моделями, когда нас заботит интерпретируемость. Как подсказывает само название, мы можем предположить, что такая модель разбивает наши данные, задавая последовательность вопросов.

Рассмотрим пример, представленный на рис. 3.17, в котором дерево принятия решений используется при определении вида деятельности в отдельно взятый день.



**Рис. 3.17.** Пример дерева принятия решений

Базируясь на признаках в обучающем наборе, модель на основе дерева принятия решений обучается последовательности вопросов, чтобы выводить метки классов для образцов. Хотя на рис. 3.17 иллюстрируется концепция дерева принятия решений, основанного на категориальных переменных, та же самая концепция применима и в случае, когда признаки представляют собой вещественные числа, как в наборе данных *Iris*. Например, мы могли бы просто определить отсекающее значение по оси признака “ширина чашелистика” и задать двоичный вопрос: ширина чашелистика  $\geq 2.8$  см?

Используя алгоритм принятия решения, мы начинаем с корня дерева и разбиваем данные по признаку, который дает в результате наибольший *прирост информации* (*information gain* — *IG*), как будет более подробно объясняться в следующем разделе. В рамках итерационного процесса мы повторяем описанную процедуру разбиения в каждом узле, пока листовые узлы не станут чистыми. Это означает, что все образцы в каждом узле принадлежат тому же самому классу. На практике результатом может оказаться очень глубокое дерево с многочисленными узлами, что легко способно привести к переобучению. Таким образом, обычно мы *подрезаем* дерево, устанавливая предел для его максимальной глубины.

## Доведение до максимума прироста информации — получение наибольшей отдачи

Для того чтобы разделить узлы в самых информативных признаках, нам понадобится определить целевую функцию, которую необходимо оптимизировать посредством алгоритма обучения дерева. В данном случае наша целевая функция заключается в доведении до максимума прироста информации при каждом разделении, что определяется следующим образом:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N_p} I(D_j)$$

Здесь  $f$  — признак для выполнения разбиения,  $D_p$  и  $D_j$  — набор данных родительского и  $j$ -того дочернего узла,  $I$  — мера *загрязненности*,  $N_p$  — общее количество образцов в родительском узле и  $N_j$  — количество образцов в  $j$ -том дочернем узле. Как мы увидим, прирост информации представляет собой просто разность между загрязненностью родительского узла и суммой загрязненностей дочерних узлов — чем ниже загрязненность дочерних узлов, тем выше прирост информации. Однако для простоты и ради сокращения пространства комбинаторного поиска в большинстве библиотек (включая *scikit-learn*) реализованы двоичные деревья принятия решений. Это значит, что каждый родительский узел разбивается на два дочерних узла,  $D_{\text{левый}}$  и  $D_{\text{правый}}$ :

$$IG(D_p, f) = I(D_p) - \frac{N_{\text{левый}}}{N_p} I(D_{\text{левый}}) - \frac{N_{\text{правый}}}{N_p} I(D_{\text{правый}})$$

Тремя мерами загрязненности или критериями разбиения, которые обычно применяются в двоичных деревьях принятия решений, являются *загрязненность Джини* ( $I_G$ ), *энтропия* ( $I_H$ ) и *ошибка классификации* ( $I_E$ ). Давайте начнем с определения энтропии для всех *непустых* классов ( $p(i | t) \neq 0$ ):

$$I_H(t) = - \sum_{i=1}^c p(i | t) \log_2 p(i | t)$$

Здесь  $p(i | t)$  — доля образцов, которые принадлежат классу  $i$  для индивидуального узла  $t$ . Следовательно, энтропия равна 0, если все образцы в узле принадлежат тому же самому классу, и максимальна при равномерном распределении классов. Скажем, в окружении с двоичными классами энтропия равна 0, если  $p(i = 1 | t) = 1$  или  $p(i = 0 | t) = 0$ . Если классы распределены равномерно с  $p(i = 1 | t) = 0.5$  и  $p(i = 0 | t) = 0.5$ , тогда энтропия составляет 1.

Таким образом, мы можем говорить, что критерий энтропии стремится довести до максимума полное количество информации в дереве.

Загрязненность Джини можно понимать как критерий для минимизации вероятности неправильной классификации:

$$I_G(t) = \sum_{i=1}^c p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2$$

Подобно энтропии загрязненность Джини максимальна, если классы в полной мере перемешаны, например, в окружении с двоичными классами ( $c = 2$ ):

$$I_G(t) = 1 - \sum_{i=1}^c 0.5^2 = 0.5$$

Тем не менее, на практике загрязненность Джини и энтропия обычно выдают очень похожие результаты, и часто не стоит тратить много времени на оценку деревьев с использованием различных критериев загрязненности вместо экспериментирования с разными значениями параметра отсечения при подрезке.

Еще одной мерой загрязненности является ошибка классификации:

$$I_E = 1 - \max \{p(i|t)\}$$

Ошибка классификации — полезный критерий для подрезки, но не рекомендуется для выращивания дерева принятия решений, т.к. она менее чувствительна к изменениям в вероятностях классов узлов. В качестве иллюстрации мы можем взглянуть на два возможных сценария разбиения, показанные на рис. 3.18.

Мы начинаем с набора данных  $D_p$  в родительском узле  $D_p$ , который содержит 40 образцов из класса 1 и 40 образцов из класса 2, которые мы разбиваем на два набора данных,  $D_{\text{левый}}$  и  $D_{\text{правый}}$ . Прирост информации с применением ошибки классификации в качестве критерия разбиения будет одинаковым ( $IG_E = 0.25$ ) в обоих сценариях, А и В:

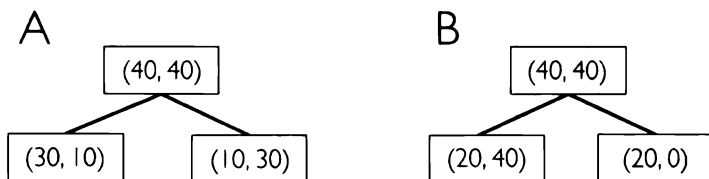


Рис. 3.18. Два возможных сценария разбиения

$$I_E(D_p) = 1 - 0.5 = 0.5$$

$$A : I_E(D_{\text{левый}}) = 1 - \frac{3}{4} = 0.25$$

$$A : I_E(D_{\text{правый}}) = 1 - \frac{3}{4} = 0.25$$

$$A : IG_E = 0.5 - \frac{4}{8} \cdot 0.25 - \frac{4}{8} \cdot 0.25 = 0.25$$

$$B : I_E(D_{\text{левый}}) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B : I_E(D_{\text{правый}}) = 1 - 1 = 0$$

$$B : IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

Однако для загрязненности Джини более привлекательно разбиение в сценарии В ( $IG_G = 0.1\bar{6}$ ) по сравнению со сценарием А ( $IG_G = 0.125$ ), т.к. сценарий В на самом деле чище:

$$I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$A : I_G(D_{\text{левый}}) = 1 - \left( \left( \frac{3}{4} \right)^2 + \left( \frac{1}{4} \right)^2 \right) = \frac{3}{8} = 0.375$$

$$A : I_G(D_{\text{правый}}) = 1 - \left( \left( \frac{1}{4} \right)^2 + \left( \frac{3}{4} \right)^2 \right) = \frac{3}{8} = 0.375$$

$$A : IG_G = 0.5 - \frac{4}{8} \cdot 0.375 - \frac{4}{8} \cdot 0.375 = 0.125$$

$$B : I_G(D_{\text{левый}}) = 1 - \left( \left( \frac{2}{6} \right)^2 + \left( \frac{4}{6} \right)^2 \right) = \frac{4}{9} = 0.\bar{4}$$

$$B : I_G(D_{\text{правый}}) = 1 - (1^2 + 0^2) = 0$$

$$B : IG_G = 0.5 - \frac{6}{8} \cdot 0.\bar{4} - 0 = 0.1\bar{6}$$

Аналогично критерий энтропии также отдает предпочтение сценарию В ( $IG_H = 0.31$ ) перед сценарием А ( $IG_H = 0.19$ ):

$$I_H(D_p) = -(0.5 \log_2(0.5) + 0.5 \log_2(0.5)) = 1$$

$$A : I_H(D_{left}) = -\left(\frac{3}{4} \log_2\left(\frac{3}{4}\right) + \frac{1}{4} \log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A : I_H(D_{right}) = -\left(\frac{1}{4} \log_2\left(\frac{1}{4}\right) + \frac{3}{4} \log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A : IG_H = 1 - \frac{4}{8} \cdot 0.81 - \frac{4}{8} \cdot 0.81 = 0.19$$

$$B : I_H(D_{left}) = -\left(\frac{2}{6} \log_2\left(\frac{2}{6}\right) + \frac{4}{6} \log_2\left(\frac{4}{6}\right)\right) = 0.92$$

$$B : I_H(D_{right}) = 0$$

$$B : IG_H = 1 - \frac{6}{8} \cdot 0.92 - 0 = 0.31$$

Чтобы более наглядно сравнить три обсуждавшихся ранее критерия загрязненности, давайте построим график индексов загрязненности для диапазона вероятностей  $[0, 1]$  класса 1. Обратите также внимание на добавление масштабированной версии энтропии (энтропия / 2) с целью наблюдения за тем, что загрязненность Джини является промежуточной мерой между энтропией и ошибкой классификации. Ниже приведен необходимый код:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return -p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
```

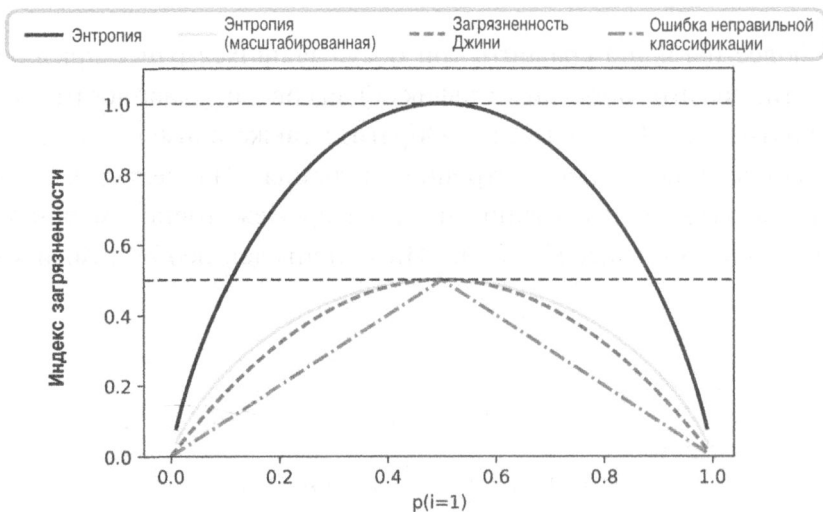


```

>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                          ['Энтропия', 'Энтропия (масштабированная)',
...                          'Загрязненность Джини',
...                          'Ошибка неправильной классификации'],
...                          ['-', '-', '--', '-.'],
...                          ['black', 'lightgray',
...                          'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                     linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=5, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('Индекс загрязненности')
>>> plt.show()

```

На рис. 3.19 показан график, полученный в результате выполнения предыдущего кода.



**Рис. 3.19.** Сравнение критериев загрязненности

## Построение дерева принятия решений

Деревья принятия решений способны строить сложные границы решений, разделяя пространство признаков на прямоугольники. Тем не менее, мы должны соблюдать осторожность, т.к. чем глубже дерево принятия решений, тем более сложными становятся границы решений, что легко может привести к переобучению. С помощью *scikit-learn* мы обучим дерево принятия решений с максимальной глубиной 4, используя загрязненность Джини в качестве критерия загрязненности. Хотя масштабирование признаков может быть желательным при визуализации, имейте в виду, что оно не является требованием для алгоритмов на основе деревьев принятия решений. Вот как выглядит код:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree_model = DecisionTreeClassifier(criterion='gini',
...                                   max_depth=4,
...                                   random_state=1)
>>> tree_model.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined,
...                       y_combined,
...                       classifier=tree_model,
...                       test_idx=range(105, 150))
>>> plt.xlabel('длина лепестка [см]')
>>> plt.ylabel('ширина лепестка [см]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

В результате выполнения примера кода мы получаем типовые параллельные осям границы решений для дерева принятия решений (рис. 3.20).

Удобная возможность библиотеки *scikit-learn* заключается в том, что после обучения она позволяет без труда визуализировать модель на основе дерева принятия решений посредством следующего кода:

```
>>> from sklearn import tree
>>> tree.plot_tree(tree_model)
>>> plt.show()
```

Итоговая визуализация показана на рис. 3.21.

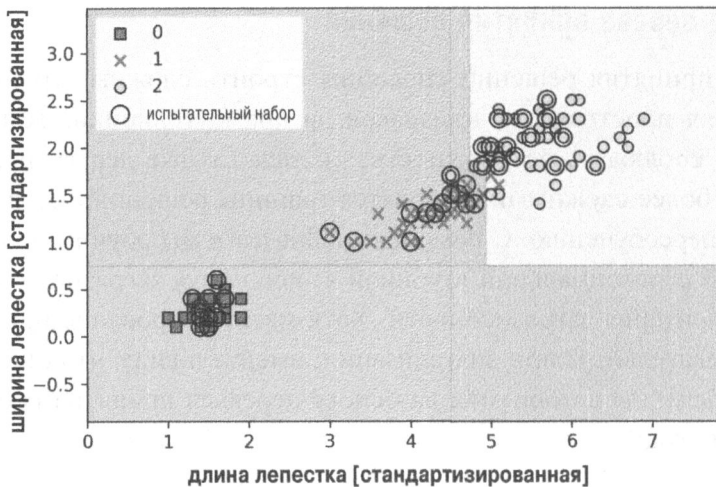


Рис. 3.20. Обучение дерева принятия решений с максимальной глубиной 4

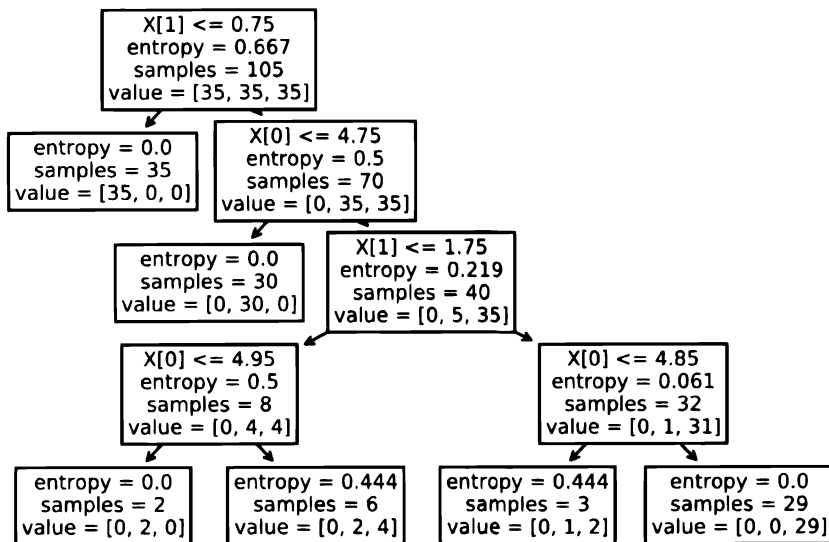


Рис. 3.21. Визуализация модели на основе дерева принятия решений

Однако более аккуратные визуализации можно получить с помощью программы Graphviz, предназначенной для вычерчивания деревьев принятия решений *scikit-learn*. Она доступна бесплатно на веб-сайте <http://www.graphviz.org> и поддерживается в средах Linux, Windows и macOS. Помимо Graphviz мы будем использовать библиотеку Python под названием PyDotPlus, которая обладает аналогичными Graphviz возможностями и поз-

воляет преобразовывать файлы данных `.dot` в файлы изображений с деревьями принятия решений. После установки Graphviz (согласно инструкциям, приведенным по ссылке <http://www.graphviz.org/download/>) можно установить pydotplus прямо через программу установки pip, выполнив в окне терминала следующую команду:

```
> pip3 install pydotplus
```



На  
заметку!

### Компоненты, требуемые для PyDotPlus

Обратите внимание, что в некоторых системах может возникнуть необходимость в ручной установке компонентов, требуемых для PyDotPlus, посредством приведенных ниже команд:

```
pip3 install graphviz
pip3 install pyparsing
```

Следующий код создаст файл с изображением нашего дерева принятия решений в формате PNG внутри локального каталога:

```
>>> from pydotplus import graph_from_dot_data
>>> from sklearn.tree import export_graphviz
>>> dot_data = export_graphviz(tree_model,
...                             filled=True,
...                             rounded=True,
...                             class_names=['Setosa',
...                                           'Versicolor',
...                                           'Virginica'],
...                             feature_names=['petal length',
...                                           'petal width'],
...                             out_file=None)
>>> graph = graph_from_dot_data(dot_data)
>>> graph.write_png('tree.png')
```

За счет применения настройки `out_file=None` мы напрямую присваиваем данные точек переменной `dot_data` вместо того, чтобы сохранять промежуточный файл `tree.dot` на диске. Аргументы `filled`, `rounded`, `class_names` и `feature_names` необязательны, но делают результирующее изображение более привлекательным, добавляя цвет, скругляя углы прямоугольников, показывая в каждом узле метку класса, к которому принадлежит большинство образцов, и отображая имена признаков в критериях разбиения. В результате мы получаем изображение дерева принятия решений, представленное на рис. 3.22.

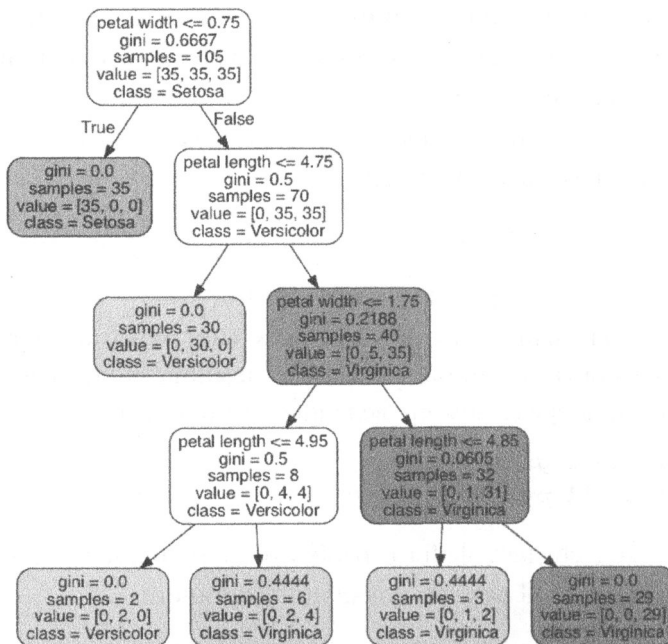


Рис. 3.22. Пример дерева принятия решений

Глядя на рис. 3.22, мы теперь можем легко отследить разделения, которые дерево принятия решений выявило из нашего обучающего набора данных. Мы начали со 105 образцов в корне и разделили их на два дочерних узла с 35 и 70 образцами, используя отсечение вида “ширина лепестка (petal width)  $\leq 0.75$  см”. После первого разделения можно заметить, что левый дочерний узел оказывается чистым и содержит только образцы из класса Iris-setosa (загрязненность Джини (gini) = 0). Затем справа производятся дальнейшие разделения, чтобы отделить образцы из классов Iris-versicolor и Iris-virginica.

Проанализировав дерево и график с областями решений дерева, мы можем сделать вывод, что дерево принятия решений не особенно хорошо выполняет работу по разделению классов цветков. К сожалению, в настоящее время в библиотеке scikit-learn не реализована функциональность для ручной подрезки построенного дерева принятия решений. Однако мы могли бы вернуться к предыдущему примеру, изменить значение параметра max\_depth дерева принятия решений на 3 и сравнить результат с текущей моделью; мы оставляем это заинтересованным читателям в качестве упражнения для самостоятельной проработки.

## Объединение множества деревьев принятия решений с помощью случайных лесов

За последнее десятилетие в приложениях МО огромную популярность обрели ансамблевые методы (*ensemble method*) благодаря высокой эффективности классификации и устойчивости к переобучению. Хотя мы рассмотрим различные ансамблевые методы, в том числе *бэггинг* (*bagging*) и *бустинг* (*boosting*), позже в главе 7, давайте обсудим алгоритм *случайного леса* (*random forest*), основанный на деревьях принятия решений, который известен своей хорошей масштабируемостью и легкостью в применении. Случайный лес можно рассматривать как *ансамбль* деревьев принятия решений. В основе случайного леса заложена идея усреднения множества (глубоких) деревьев принятия решений, которые по отдельности страдают от высокой дисперсии, с целью построения более надежной модели, обладающей большей эффективностью обобщения и меньшей восприимчивостью к переобучению. Алгоритм случайного леса можно подытожить в виде четырех простых шагов.

1. Извлечь случайную *бутстрэп*-выборку размером  $n$  (случайно выбрать  $n$  образцов из обучающего набора с возвращением).
2. Вырастить дерево принятия решений из бутстрэп-выборки. В каждом узле:
  - а) случайно выбрать  $d$  признаков без возвращения;
  - б) разделить узел, используя признак, который обеспечивает наилучшее разделение согласно целевой функции, например, доводящей до максимума прирост информации.
3. Повторить  $k$  раз шаги 1 и 2.
4. Объединить прогнозы всех деревьев путем назначения метки класса по *большинству голосов*. Мажоритарная система будет более подробно обсуждаться в главе 7.

Мы должны отметить одну небольшую модификацию на шаге 2, когда обучаем индивидуальные деревья принятия решений: вместо оценки всех признаков для установления наилучшего разделения в каждом узле во внимание принимается только случайный поднабор признаков.



### Выборка с возвращением и без возвращения

На тот случай, если вы не знакомы с терминами выборка “с” возвращением и выборка “без” возвращения, давайте проведем простой мысленный эксперимент. Предположим, что мы играем в лотерею, где числа случайным образом извлекаются из урны. Мы начинаем с урны, содержащей пять уникальных чисел, 0, 1, 2, 3 и 4, и на каждом шаге извлекаем в точности одно число. В первом раунде шанс выбрать определенное число из урны составляет  $1/5$ . При выборке без возвращения мы не помещаем число обратно в урну на каждом шаге. В результате вероятность извлечения определенного числа из оставшегося набора чисел в следующем раунде зависит от предыдущего раунда. Например, при наличии в оставшемся наборе чисел 0, 1, 2 и 4 шанс извлечения на следующем шаге числа 0 становится равным  $1/4$ .

Тем не менее, в случайной выборке с возвращением мы всегда возвращаем извлеченное число в урну, поэтому вероятность извлечения определенного числа на каждом шаге не изменяется; мы можем извлечь одно и то же число более одного раза. Другими словами, при выборке с возвращением образцы (числа) независимы и имеют нулевую ковариацию. Например, после пяти раундов извлечения случайных чисел результаты могли бы выглядеть так:

- случайная выборка без возвращения — 2, 1, 3, 4, 0;
- случайная выборка с возвращением — 1, 3, 3, 4, 1.

Хотя случайные леса не предлагают тот же самый уровень интерпретируемости, как у деревьев принятия решений, крупное преимущество случайных лесов связано с тем, что нам не приходится особо переживать по поводу выбора хороших значений для гиперпараметров. Случайный лес обычно не приходится подрезать, т.к. ансамблевая модель довольно устойчива к шуму от индивидуальных деревьев принятия решений. Единственный параметр, о котором действительно нужно заботиться — это количество деревьев  $k$  (шаг 3), выбираемое для случайного леса. Как правило, чем больше число деревьев, тем выше эффективность классификатора на основе случайного леса за счет увеличенных вычислительных затрат.

Несмотря на меньшее распространение на практике, другими гиперпараметрами классификатора на основе случайного леса, допускающими опти-

мизацию (с применением приемов, обсуждаемых в главе 6), являются размер  $n$  бутстрэп-выборки (шаг 1) и количество признаков  $d$ , которое выбирается случайным образом для каждого разделения (шаг 2.а). Посредством размера  $n$  бутстрэп-выборки мы управляем компромиссом между смещением и дисперсией случайного леса.

Уменьшение размера бутстрэп-выборки приводит к увеличению несходства между индивидуальными деревьями, т.к. вероятность того, что определенный обучающий образец включается в бутстрэп-выборку, становится низкой. Соответственно сокращение размера бутстрэп-выборки может увеличить *произвольность* случайного леса, что способно помочь ослабить влияние переобучения. Однако меньшие бутстрэп-выборки обычно дают в результате пониженную общую эффективность случайного леса — небольшой промежуток между эффективностью при обучении и эффективностью при испытаниях, но в целом низкую эффективность при испытаниях. И наоборот, увеличение размера бутстрэп-выборки может увеличить степень переобучения. Поскольку бутстрэп-выборки и, следовательно, индивидуальные деревья принятия решений становятся все больше похожими друг на друга, они учатся более тесно подгоняться к исходному обучающему набору данных.

В большинстве реализаций, включая реализацию `RandomForestClassifier` из *scikit-learn*, размер бутстрэп-выборки выбирается равным количеству обучающих образцов в исходном обучающем наборе, что обычно обеспечивает хороший компромисс между смещением и дисперсией. Для числа признаков  $d$  в каждом разделии мы хотим выбрать значение, которое меньше общего количества признаков в обучающем наборе. Разумным стандартным значением, используемым в *scikit-learn* и других реализациях, является  $d = \sqrt{m}$ , где  $m$  — число признаков в обучающем наборе.

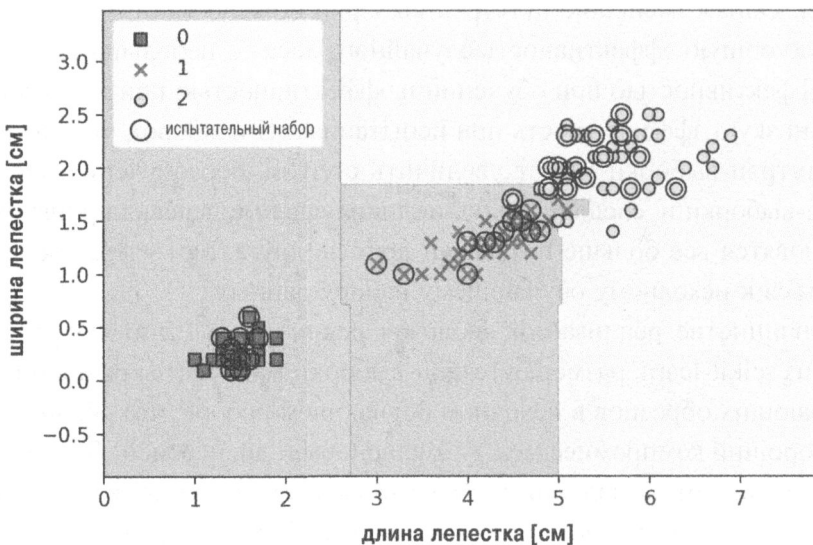
Удобно то, что нам не приходится самостоятельно строить классификатор на основе случайного леса из индивидуальных деревьев принятия решений, т.к. в *scikit-learn* имеется реализация, которую мы можем задействовать:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(criterion='gini',
...                               n_estimators=25,
...                               random_state=1,
...                               n_jobs=2)
>>> forest.fit(X_train, y_train)
```



```
>>> plot_decision_regions(X_combined, y_combined,  
...                       classifier=forest, test_idx=range(105,150))  
>>> plt.xlabel('длина лепестка [см]')  
>>> plt.ylabel('ширина лепестка [см]')  
>>> plt.legend(loc='upper left')  
>>> plt.tight_layout()  
>>> plt.show()
```

В результате выполнения предыдущего кода мы должны увидеть области решений, сформированные ансамблем деревьев в случайном лесе, как показано на рис. 3.23.



**Рис. 3.23.** Области решений, сформированные ансамблем деревьев в случайном лесе

С помощью приведенного выше кода мы обучаем случайный лес, состоящий из 25 деревьев принятия решений, с применением параметра `n_estimators` и при разделении узлов в качестве меры загрязненности используем критерий загрязненности Джини. Несмотря на то что мы выращиваем очень маленький случайный лес из совсем небольшого обучающего набора данных, в демонстрационных целях мы применяем параметр `n_jobs`, который дает возможность распараллеливания процесса обучения модели, используя множество процессорных ядер (здесь два ядра).

## Метод $k$ ближайших соседей — алгоритм ленивого обучения

Последним алгоритмом обучения с учителем, обсуждаемым в этой главе, будет классификатор на основе  $k$  ближайших соседей ( $k$ -nearest neighbor — KNN), который особенно интересен тем, что он фундаментально отличается от алгоритмов обучения, рассмотренных до сих пор.

Алгоритм KNN — типичный пример *ленивого ученика*. Ученик называется “ленивым” не из-за своей очевидной простоты, а оттого, что он не узнает различающую функцию из обучающих данных, а взамен запоминает обучающий набор данных.



На заметку!

### Параметрические модели в сравнении с непараметрическими моделями

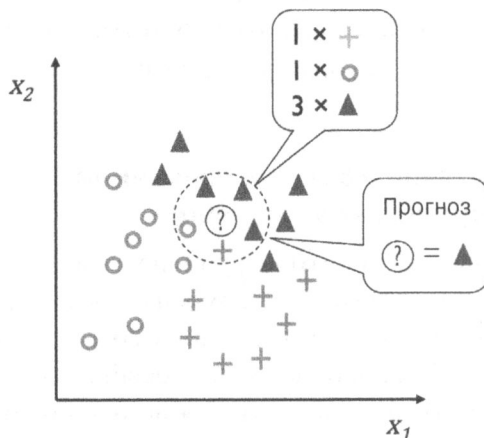
Алгоритмы МО могут быть сгруппированы в *параметрические* и *непараметрические* модели. С помощью параметрических моделей мы производим оценку параметров на обучающем наборе данных, чтобы вывести функцию, которая способна классифицировать новые точки данных, больше не требуя исходного обучающего набора данных. Типичными примерами параметрических моделей являются перцептрон, логистическая регрессия и линейный SVM. По контрасту с ними непараметрические модели не могут быть описаны посредством фиксированного набора параметров, и количество параметров растет вместе с обучающими данными. До сих пор мы встречались с двумя примерами непараметрических моделей: классификатором на основе дерева принятия решений/случайного леса и ядерным SVM.

Алгоритм KNN относится к подкатегории непараметрических моделей, которая описывается как *обучение на основе образцов*. Модели, опирающиеся на обучение на основе образцов, характеризуются запоминанием обучающего набора данных, а ленивое обучение — это специальный случай обучения на основе образцов, которое связано с отсутствием (нулем) издержек во время процесса обучения.

Сам алгоритм KNN довольно прямолинеен и может быть подытожен в виде следующих шагов.

1. Выбрать число  $k$  и метрику расстояния.
2. Найти  $k$  ближайших соседей образца, который нужно классифицировать.
3. Назначить метку класса по большинству голосов.

На рис. 3.24 показано, как новой точке данных (?) назначается класс треугольника на основе мажоритарного голосования среди ее пяти ближайших соседей.



**Рис. 3.24.** Назначение точке данных класса треугольника по большинству голосов

Базируясь на выбранной метрике расстояния, алгоритм KNN находит в обучающем наборе данных  $k$  образцов, ближайших (наиболее похожих) к точке, которую необходимо классифицировать. Затем метка класса для точки данных определяется на основе мажоритарного голосования среди ее  $k$  ближайших соседей.

Главное преимущество такого основанного на памяти подхода в том, что классификатор немедленно адаптируется по мере накопления нами новых обучающих данных. Тем не менее, недостаток связан с тем, что вычислительная сложность классификации новых образцов при худшем сценарии растет линейно с увеличением количества образцов в обучающем наборе данных — если только набор данных не обладает совсем незначительным числом измерений (признаков), а алгоритм был реализован с применением эффективных структур данных, таких как  $K$ -мерные деревья (“An Algorithm for Finding Best Matches in Logarithmic Expected Time”) (Алгоритм для нахождения наилучших совпадений за логарифмическое ожидаемое время),

Дж. Фридман, Дж. Бентли и Р. Финкель, *ACM transactions on mathematical software (TOMS)*, 3(3): стр. 209–226 (1977 год)). Кроме того, мы не можем отбрасывать обучающие образцы ввиду отсутствия шага *обучения*. Таким образом, при работе с крупными наборами данных проблемой может стать пространство хранения.

За счет выполнения следующего кода мы реализуем модель KNN из *scikit-learn*, используя метрику евклидова расстояния:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                           metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('длина лепестка [стандартизированная]')
>>> plt.ylabel('ширина лепестка [стандартизированная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Указав пять соседей в модели KNN для этого набора данных, мы получаем относительно гладкую границу решений, как показано на рис. 3.25.

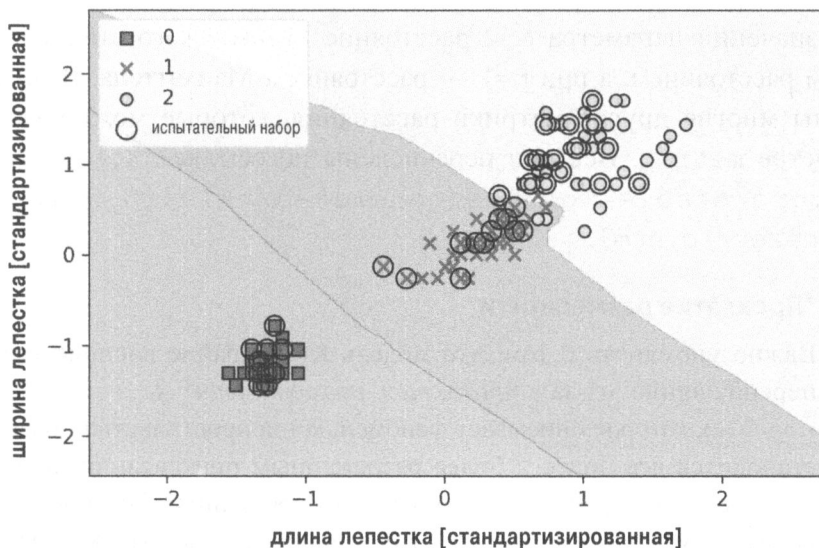


Рис. 3.25. Классификатор на основе  $k$  ближайших соседей



### Разрешение связей

В случае равного числа голосов реализация алгоритма KNN из *scikit-learn* будет отдавать предпочтение соседям с наименьшим расстоянием к образцу. Если соседи имеют подобные расстояния, тогда алгоритм выберет метку того класса, который поступает первым из обучающего набора данных.

Правильный выбор  $k$  критически важен для нахождения хорошего баланса между переобучением и недообучением. Мы также должны обеспечить выбор метрики расстояния, подходящей для признаков в наборе данных. Для образцов с вещественными значениями вроде цветков в наборе данных Iris, которые имеют признаки, измеренные в сантиметрах, часто применяется простая мера — евклидово расстояние. Однако при использовании меры в виде евклидова расстояния также важно стандартизировать данные, чтобы каждый признак в равной степени вносил вклад в расстояние. Расстояние Минковского (`metric='minkowski'`), которое применялось в предыдущем коде, представляет собой всего лишь обобщение евклидова расстояния и расстояния Манхэттена, что может быть записано следующим образом:

$$d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \sqrt[p]{\sum_k |\mathbf{x}_k^{(i)} - \mathbf{x}_k^{(j)}|^p}$$

При значении параметра  $p=2$  расстояние Минковского становится евклидовым расстоянием, а при  $p=1$  — расстоянием Манхэттена. В *scikit-learn* доступны многие другие метрики расстояния, которые можно указывать в параметре `metric`. Все они перечислены по ссылке <http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html>.



### “Проклятие размерности”

Важно упомянуть о том, что модель KNN крайне восприимчива к переобучению из-за *“проклятия размерности”* (*curse of dimensionality*), которое описывает феномен, когда пространство признаков становится все более и более разреженным при увеличивающемся количестве измерений в обучающем наборе данных фиксированного размера. Мы можем считать, что в пространстве высокой размерности даже ближайшие соседи находятся слишком далеко друг от друга, чтобы дать хорошую оценку.

В разделе, посвященном логистической регрессии, мы обсуждали концепцию регуляризации как способа избежать переобучения. Тем не менее, в моделях, где регуляризация неприменима, наподобие деревьев и KNN, мы можем использовать приемы выбора признаков и понижения размерности, которые помогают уклониться от “проклятия размерности”. Мы рассмотрим их более подробно в следующей главе.

## Резюме

В главе вы узнали о многих алгоритмах МО, которые применяются для решения линейных и нелинейных задач. Вы видели, что деревья принятия решений особенно привлекательны, если нас заботит интерпретируемость. Логистическая регрессия не только является удобной моделью для динамического обучения посредством стохастического градиентного спуска, но также позволяет прогнозировать вероятность определенного события.

Хотя методы опорных векторов представляют собой мощные линейные модели, расширяемые с помощью ядерного трюка для решения нелинейных задач, они имеют много параметров, которые должны быть настроены, чтобы вырабатывать хорошие прогнозы. В противоположность им ансамблевые методы, подобные случайным лесам, не требуют настройки многочисленных параметров и не подвергаются переобучению настолько легко, как деревья принятия решений, что на практике делает их привлекательными моделями для многих предметных областей. Классификатор KNN предлагает альтернативный подход к классификации через ленивое обучение, которое делает возможной выработку прогнозов без обучения модели, но с более затратным в вычислительном плане шагом прогнозирования.

Однако еще более важным аспектом, чем выбор подходящего алгоритма обучения, являются данные, доступные в обучающем наборе. Ни один алгоритм не будет способен вырабатывать хорошие прогнозы без информативных и отличительных признаков.

В следующей главе мы обсудим важные темы, касающиеся предварительной обработки данных, выбора признаков и понижения размерности, которые понадобятся для построения мощных моделей МО. Позже, в главе 6, мы покажем, как можно оценивать и сравнивать эффективность моделей, и продемонстрируем полезные трюки для точной настройки различных алгоритмов.



# ПОСТРОЕНИЕ ХОРОШИХ ОБУЧАЮЩИХ НАБОРОВ — ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Качество данных и объем содержащейся в них полезной информации — ключевые факторы, которые определяют, насколько хорошо алгоритмы МО способны обучаться. Следовательно, критически важно провести исследование и предварительную обработку набора данных, прежде чем передавать его алгоритму обучения. В настоящей главе мы обсудим основные приемы предварительной обработки данных, которые помогут строить хорошие модели МО.

В главе будут раскрыты следующие темы:

- удаление и *условный расчет* (*imputing*) недостающих значений в наборе данных;
- приведение категориальных данных в форму, подходящую для алгоритмов МО;
- выбор значимых признаков для построения модели.



## Решение проблемы с недостающими данными

В реальных приложениях внутри обучающих образцов нередко отсутствует одно или большее количество значений по различным причинам. Это может быть связано с ошибкой в процессе сбора данных, неприменимостью определенных измерений или просто оставлением некоторых полей пустыми, например, при опросе. Обычно мы воспринимаем недостающие значения как пробелы в нашей таблице данных либо как строковые заполнители вроде NaN для нечислового значения или NULL (широко используемый индикатор неизвестных значений в реляционных базах данных). К сожалению, большинство вычислительных инструментов либо не в состоянии обрабатывать недостающие значения такого рода, либо выдают непредсказуемые результаты, если мы просто игнорируем их. По указанной причине крайне важно позаботиться о недостающих значениях перед тем, как продолжать дальнейший анализ.

В текущем разделе мы проработаем несколько практических приемов решения проблемы с недостающими значениями, удаляя записи из набора данных или производя условный расчет недостающих значений из других образцов и признаков.

### Идентификация недостающих значений в табличных данных

Перед обсуждением приемов решения проблемы с недостающими данными давайте создадим простой кадр данных из файла *значений, отделенных друг от друга запятыми* (*comma-separated values* — CSV), чтобы лучше ухватить суть проблемы:

```
>>> import pandas as pd
>>> from io import StringIO

>>> csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>>> # В случае работы с Python 2.7 понадобится
>>> # преобразовать строку в формат Unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

В показанном выше коде посредством функции `read_csv` мы читаем данные CSV в объект `DataFrame` из `pandas` и замечаем, что две недостающие ячейки были заменены `NaN`. Функция `StringIO` применялась просто в целях иллюстрации. Она позволяет читать строку, присвоенную `csv_data`, в объект `DataFrame` из `pandas`, как если бы она была обычным файлом CSV на жестком диске.

В более крупном объекте `DataFrame` ручной поиск недостающих значений становится утомительным; в таком случае мы можем использовать метод `isnull` для возвращения объекта `DataFrame` с булевскими значениями, которые указывают, содержит ли ячейка числовую величину (`False`) или же данные отсутствуют (`True`). Затем с помощью метода `sum` мы можем получить количество недостающих значений по столбцам:

```
>>> df.isnull().sum()
A      0
B      0
C      1
D      1
dtype: int64
```

Таким способом мы способны подсчитать количество недостающих значений на столбец; в последующих подразделах мы рассмотрим различные стратегии решения проблемы с недостающими данными.



На заметку!

### Удобная обработка данных с помощью класса `DataFrame` из библиотеки `pandas`

Несмотря на то что библиотека `scikit-learn` изначально проектировалась для работы только с массивами `NumPy`, иногда удобнее предварительно обрабатывать данные с применением класса `DataFrame` из библиотеки `pandas`. В настоящее время большинство функций `scikit-learn` поддерживают на входе объекты `DataFrame`, но поскольку в API-интерфейсе `scikit-learn` более естественной является обработка массивов `NumPy`, рекомендуется по возможности использовать массивы `NumPy`. Следует отметить, что с помощью атрибута `values` мы

всегда можем получить доступ к массиву NumPy внутри DataFrame перед его передачей оценщику scikit-learn:

```
>>> df.values
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6., nan,  8.],
       [10., 11., 12., nan]])
```

## Исключение обучающих образцов или признаков с недостающими значениями

Один из простейших способов решения проблемы с недостающими данными предусматривает полное удаление соответствующих признаков (столбцов) или обучающих образцов (строк) из набора данных; строки с недостающими значениями могут быть легко удалены посредством метода `dropna`:

```
>>> df.dropna(axis=0)
   A    B    C    D
0  1.0  2.0  3.0  4.0
```

Подобным образом мы можем отбрасывать столбцы, которые имеют, по крайней мере, одно значение NaN в любой строке, устанавливая аргумент `axis` в 1:

```
>>> df.dropna(axis=1)
   A    B
0  1.0  2.0
1  5.0  6.0
2 10.0 11.0
```

Метод `dropna` поддерживает несколько дополнительных параметров, которые могут оказаться полезными:

```
# удалить только строки, где все столбцы содержат NaN
# (здесь возвращается полный массив, т.к. отсутствует
# строка со значениями NaN во всех столбцах)
```

```
>>> df.dropna(how='all')
   A    B    C    D
0  1.0  2.0  3.0  4.0
1  5.0  6.0  NaN  8.0
2 10.0 11.0 -12.0  NaN
```

```
# удалить строки, которые содержат менее 4 вещественных значений
```

```
>>> df.dropna(thresh=4)
   A    B    C    D
0  1.0  2.0  3.0  4.0
```

```
# удалить только строки, где NaN появляется
# в специфических столбцах (здесь: 'C')
>>> df.dropna(subset=['C'])
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
2	10.0	11.0	12.0	NaN

Хотя удаление недостающих данных кажется удобным подходом, ему также присущи определенные недостатки; скажем, мы можем в итоге удалить слишком много образцов, сделав надежный анализ невозможным. Либо же в случае удаления чересчур большого количества столбцов признаков возникнет риск утери ценной информации, которая нужна классификатору для различения классов. Таким образом, в следующем разделе мы взглянем на самые распространенные альтернативы для решения проблемы с недостающими значениями: методики интерполяции.

## Условный расчет недостающих значений

Часто удаление обучающих образцов или отбрасывание целых столбцов признаков попросту неосуществимо из-за возможности утери слишком многих ценных данных. В таком случае мы можем использовать различные методики интерполяции для оценки недостающих значений на основе остальных обучающих образцов в наборе данных. Одним из наиболее распространенных приемов интерполяции является *условный расчет на основе среднего* (*mean imputation*), когда мы просто заменяем недостающее значение средним значением полного столбца признака. Достичь этого удобно с применением класса `SimpleImputer` из `scikit-learn`, как демонстрируется в следующем коде:

```
>>> from sklearn.impute import SimpleImputer
>>> import numpy as np
>>> imr = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imr = imr.fit(df.values)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
```

array([[	1.,	2.,	3.,	4.],	
	[	5.,	6.,	7.5,	8.],
	[	10.,	11.,	12.,	6.]])

Здесь мы заменяем каждое значение NaN соответствующим средним, которое отдельно вычисляется для каждого столбца признака. Для параметра `strategy` доступны другие варианты — `median` и `most_frequent`, где последний заменяет недостающие значения самыми часто встречающимися значениями. Это полезно для условного расчета значений категориальных признаков, например, столбца признака, который хранит коды названий цветов, таких как красный, зеленый и синий; позже в главе будут приведены примеры данных подобного вида.

В качестве альтернативы можно использовать еще более удобный способ условного расчета недостающих значений — метод `fillna` из `pandas`, передавая ему в аргументе метод условного расчета. Например, с применением `pandas` мы могли бы обеспечить условный расчет на основе среднего прямо в объекте `DataFrame`:

```
>>> df.fillna(df.mean())
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.5	8.0
2	10.0	11.0	12.0	6.0

## Понятие API-интерфейса оценщиков `scikit-learn`

В предыдущем разделе мы использовали класс `SimpleImputer` из `scikit-learn` для условного расчета недостающих значений в нашем наборе данных. В рамках библиотеки `scikit-learn` класс `SimpleImputer` относится к так называемым классам *преобразователей* (*transformer*), которые применяются для трансформации данных. Двумя неотъемлемыми методами оценщиков являются `fit` и `transform`. Метод `fit` используется для того, чтобы узнать параметры из обучающих данных, а метод `transform` применяет выявленные параметры для трансформации данных. Любой массив данных, подлежащий трансформации, должен иметь такое же количество признаков, как у массива данных, который использовался для подгонки модели. На рис. 4.1 показано, каким образом преобразователь, подогнанный к обучающим данным, применяется для трансформации обучающего и нового испытательного набора данных.

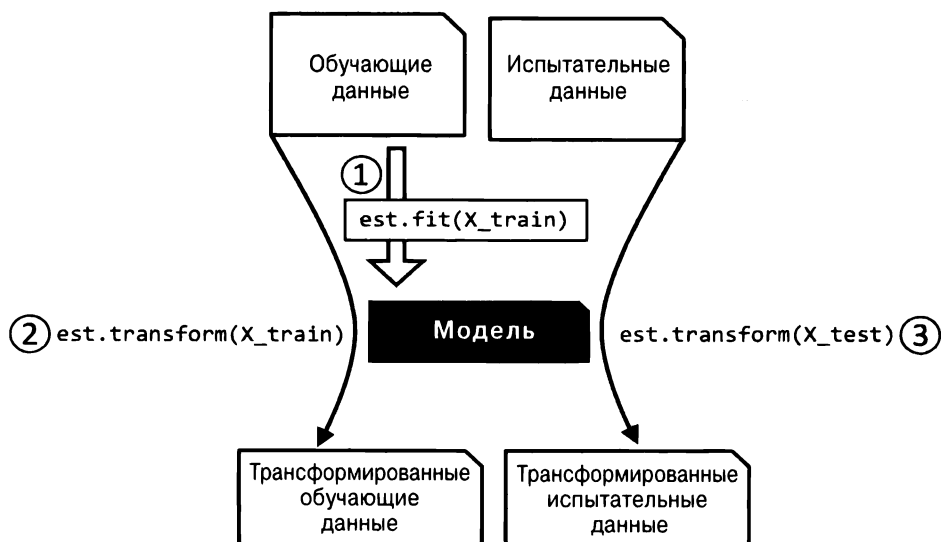


Рис. 4.1. Использование преобразователя для трансформации данных

Классификаторы, применяемые в главе 3, внутри библиотеки `scikit-learn` принадлежат к числу так называемых *оценщиков* с API-интерфейсом, который концептуально очень похож на API-интерфейс преобразователей. Оценщики имеют метод `predict`, но, как мы увидим позже в главе, также могут иметь метод `transform`. Вы можете вспомнить, что мы использовали метод `fit`, чтобы узнать параметры модели, когда обучали эти оценщики для классификации. Однако в задачах обучения с учителем мы дополнительно предоставляем метки классов для подгонки модели, которые затем можно применять для выработки прогнозов о новых непомеченных образцах данных через метод `predict`, как иллюстрируется на рис. 4.2.

## Обработка категориальных данных

До сих пор мы работали только с числовыми значениями. Тем не менее, реальные наборы данных нередко содержат один или большее число столбцов категориальных признаков. В этом разделе мы будем использовать простые, но эффективные примеры, чтобы посмотреть, как такой тип данных обрабатывается в библиотеках численных расчетов.

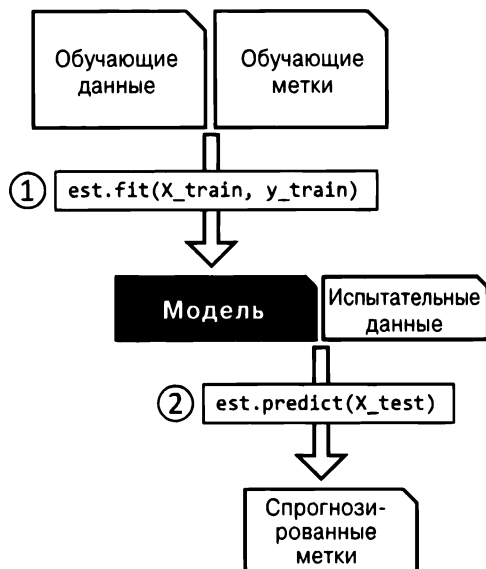


Рис. 4.2. Дополнительное предоставление меток классов для подгонки модели

Когда речь идет о категориальных данных, мы должны дополнительно проводить различие между *именными* и *порядковыми* признаками. Порядковые признаки можно понимать как категориальные значения, которые допускают сортировку или упорядочение. Например, размер футболки был бы порядковым признаком, потому что мы можем определить порядок  $XL > L > M$ . Напротив, именные признаки не подразумевают какого-либо порядка и в контексте примера с футболками мы могли бы считать именным признаком цвет футболки, т.к. обычно не имеет смысла говорить что-то вроде того, что красный больше синего.

## Кодирование категориальных данных с помощью pandas

Прежде чем заняться исследованием различных методик обработки таких категориальных данных, давайте создадим новый объект `DataFrame` для иллюстрации задачи:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...     ['green', 'M', 10.1, 'class2'],
...     ['red', 'L', 13.5, 'class1'],
...     ['blue', 'XL', 15.3, 'class2']])
```

```
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color  size  price  classlabel
0  green    M   10.1     class2
1   red    L   13.5     class1
2  blue   XL   15.3     class2
```

В предыдущем выводе видно, что вновь созданный объект `DataFrame` содержит столбцы с именованным признаком (`color`), порядковым признаком (`size`) и числовым признаком (`price`). Метки классов (при условии, что мы создали набор данных для задачи обучения с учителем) хранятся в последнем столбце (`classlabel`). Алгоритмы обучения для классификации, которые мы обсуждали в книге, не используют порядковую информацию в метках классов.

## Отображение порядковых признаков

Чтобы обеспечить корректную интерпретацию порядковых признаков алгоритмами обучения, нам нужно преобразовать строковые значения в целые числа. К сожалению, нет удобной функции, которая могла бы автоматически вывести правильный порядок меток в признаке `size`, поэтому нам придется определить отображение вручную. В представленном ниже простом примере мы предполагаем, что нам известна числовая разница между признаками, скажем,  $XL = L + 1 = M + 2$ :

```
>>> size_mapping = {'XL': 3,
...                 'L': 2,
...                 'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price  classlabel
0  green    1   10.1     class2
1   red    2   13.5     class1
2  blue    3   15.3     class2
```

Если на более поздней стадии целочисленные значения необходимо трансформировать в первоначальное строковое представление, тогда мы можем определить словарь обратного отображения `inv_size_mapping = {v: k for k, v in size_mapping.items() }`. Затем он может применяться при вызове метода `map` библиотеки `pandas` на трансформированном столбце признака подобно использованному ранее словарю `size_mapping`:



```
>>> inv_size_mapping = {v: k for k, v in size_mapping.items()}
>>> df['size'].map(inv_size_mapping)
0    M
1    L
2   XL
Name: size, dtype: object
```

## Кодирование меток классов

Многие библиотеки МО требуют, чтобы метки классов были закодированы как целочисленные значения. Несмотря на то что большинство оценщиков в библиотеке `scikit-learn`, ориентированных на классификацию, внутренне преобразуют метки классов в целые числа, установившаяся практика предполагает представление меток классов в виде целочисленных массивов во избежание технических затруднений. Для кодирования меток классов мы можем применять подход, подобный обсуждавшемуся ранее подходу с отображением порядковых признаков. Необходимо помнить, что метки классов *не* являются порядковыми, поэтому не играет роли, какое целое число будет присвоено определенной строковой метке. Таким образом, мы можем просто перечислить метки классов, начиная с 0:

```
>>> import numpy as np
>>> class_mapping = {label: idx for idx, label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Далее мы можем воспользоваться словарем отображения для трансформации меток классов в целые числа:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1            1
1   red     2   13.5            0
2  blue     3   15.3            1
```

Чтобы вернуть преобразованные метки классов в первоначальное строковое представление, мы можем обратить пары “ключ-значение” в словаре отображения, как показано ниже:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price  classlabel
0  green     1   10.1     class2
1   red     2   13.5     class1
2  blue     3   15.3     class2
```

В качестве альтернативы для достижения той же цели в библиотеке `scikit-learn` реализован удобный класс `LabelEncoder`:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([1, 0, 1])
```

Обратите внимание, что метод `fit_transform` — это лишь сокращение для вызова методов `fit` и `transform` по отдельности, а с помощью метода `inverse_transform` целочисленные метки классов можно трансформировать обратно в первоначальное строковое представление:

```
>>> class_le.inverse_transform(y)
array(['class2', 'class1', 'class2'], dtype=object)
```

## Выполнение унитарного кодирования на именных признаках

В предыдущем разделе для преобразования порядкового признака `size` в целые числа мы применяли простой подход со словарем отображения. Поскольку оценщики `scikit-learn`, предназначенные для классификации, трактуют метки классов как категориальные данные, что не подразумевает наличие любого порядка (именные данные), для кодирования строковых меток в целые числа мы использовали удобный класс `LabelEncoder`. Может показаться, что похожий подход удалось бы применить для трансформации именного столбца цвета (`color`) в нашем наборе данных:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

После выполнения предыдущего кода первый столбец в NumPy-массиве `X` теперь хранит новые значения цвета, закодированные следующим образом:

- `blue = 0`
- `green = 1`
- `red = 2`

Если мы на этом остановимся и передадим массив нашему классификатору, то допустим наиболее распространенную ошибку при работе с категориальными данными. Заметили проблему? Хотя значения цвета не должны поступать в каком-либо порядке, алгоритм обучения теперь предполагает, что `green` больше `blue`, а `red` больше `green`. Несмотря на некорректность такого предположения, алгоритм по-прежнему мог бы выдавать пригодные результаты. Однако результаты подобного рода не были бы оптимальными.

Общепринятый обход описанной проблемы предусматривает использование приема, называемого *унитарным кодированием* (*one-hot encoding*) или кодированием с одним активным состоянием. Идея подхода заключается в том, чтобы создать новый фиктивный признак для каждого уникального значения в столбце именного признака. В рассматриваемом примере мы преобразовали бы признак `color` в три новых признака: `blue`, `green` и `red`. Затем с помощью двоичных значений можно было бы указывать цвет (`color`) образца; скажем, образец синего (`blue`) цвета кодировался бы как `blue=1, green=0, red=0`. Для выполнения такой трансформации мы можем применить класс `OneHotEncoder`, реализованный в модуле `scikit-learn.preprocessing`:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> X = df[['color', 'size', 'price']].values
>>> color_ohe = OneHotEncoder()
>>> color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

Обратите внимание, что мы применили `OneHotEncoder` только к единственному столбцу, (`X[:, 0].reshape(-1, 1)`), чтобы избежать модификации остальных двух столбцов в массиве. Если мы хотим выборочно трансформировать столбцы в массиве с множеством признаков, то можем использовать класс `ColumnTransformer`, который принимает список (`(name, transformer, column(s))`) кортежей:

```
>>> from sklearn.compose import ColumnTransformer
>>> X = df[['color', 'size', 'price']].values
>>> c_transf = ColumnTransformer([
...     ('onehot', OneHotEncoder(), [0]),
...     ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)
array([[0.0, 1.0, 0.0, 1, 10.1],
       [0.0, 0.0, 1.0, 2, 13.5],
       [1.0, 0.0, 0.0, 3, 15.3]])
```

В предыдущем примере кода посредством аргумента 'passthrough' мы указали, что желаем модифицировать только первый столбец и оставить другие два столбца незатронутыми.

Еще более удобный способ создания таких фиктивных признаков через унитарное кодирование предполагает использование метода `get_dummies`, реализованного в `pandas`. Будучи примененным к объекту `DataFrame`, метод `get_dummies` преобразует только строковые столбцы и оставит все остальные столбцы неизменными:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
```

	price	size	color_blue	color_green	color_red
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0

При использовании закодированных унитарным кодом наборов данных мы должны помнить, что они привносят мультиколлинеарность, которая может стать проблемой для определенных методов (например, методов, требующих обращения матриц). Если признаки сильно взаимосвязаны, тогда обращаться матрицы будет трудно в вычислительном плане, что может привести к получению численно неустойчивых оценок. Чтобы сократить взаимосвязь между переменными, мы можем просто удалить один столбец признака из закодированного унитарным кодом массива. Обратите внимание, что в результате удаления столбца признака мы не теряем какую-то важную информацию. Скажем, после удаления столбца `color_blue` информация признака все равно сохраняется, т.к. из наблюдения `color_green=0` и `color_red=0` вытекает, что цветом должен быть `blue`.

Если мы применяем функцию `get_dummies`, то можем удалить первый столбец, указав `True` для параметра `drop_first`, как демонстрируется в следующем коде:

```
>>> pd.get_dummies(df[['price', 'color', 'size']],
...                 drop_first=True)
   price  size  color_green  color_red
0   10.1    1            1           0
1   13.5    2            0           1
2   15.3    3            0           0
```

Чтобы удалить избыточный столбец через `OneHotEncoder`, нам необходимо установить `drop='first'` и `categories='auto'`:

```
>>> color_ohe = OneHotEncoder(categories='auto', drop='first')
>>> c_transf = ColumnTransformer([
...     ('onehot', color_ohe, [0]),
...     ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)
array([[ 1. ,  0. ,  1. , 10.1],
       [ 0. ,  1. ,  2. , 13.5],
       [ 0. ,  0. ,  3. , 15.3]])
```



На  
заметку!

### Дополнительно: кодирование порядковых признаков

Если мы не уверены в числовых различиях между категориями порядковых признаков или разница между двумя порядковыми признаками не определена, тогда мы можем также кодировать их с использованием пороговой кодировки со значениями 0/1. Например, мы можем расщепить признак `size` со значениями `M`, `L` и `XL` на два новых признака, “`x > M`” и “`x > L`”. Давайте возьмем исходный объект `DataFrame`:

```
>>> df = pd.DataFrame([['green', 'M', 10.1,
...                     'class2'],
...                     ['red', 'L', 13.5,
...                     'class1'],
...                     ['blue', 'XL', 15.3,
...                     'class2']])
>>> df.columns = ['color', 'size', 'price',
...               'classlabel']
>>> df
```

Мы можем применять метод `apply` объектов `DataFrame` из `pandas` для написания специальных лямбда-выражений, чтобы кодировать эти переменные с использованием подхода с пороговыми значениями:

```
>>> df['x > M'] = df['size'].apply(
...     lambda x: 1 if x in {'L', 'XL'} else 0)
>>> df['x > L'] = df['size'].apply(
...     lambda x: 1 if x == 'XL' else 0)
>>> del df['size']
>>> df
```

## Разбиение набора данных на отдельные обучающий и испытательный наборы

В главах 1 и 3 мы кратко представили концепцию разбиения набора данных на отдельные поднаборы для обучения и испытаний. Вспомните, что сравнение прогнозов с настоящими метками классов можно понимать как несмещенную (неискаженную) оценку эффективности модели перед ее выпуском в реальный мир. В этом разделе мы подготовим новый набор данных с информацией о винах — *Wine*. После предварительной обработки набора данных мы исследуем различные приемы выбора признаков для понижения размерности набора данных.

Набор данных *Wine* — еще один набор данных с открытым кодом, который доступен в Хранилище машинного обучения Калифорнийского университета в Ирвайне (UCI) по ссылке <https://archive.ics.uci.edu/ml/datasets/Wine>; он содержит 178 образцов вин с 13 признаками, описывающими их химические свойства.



На заметку!

### Получение набора данных *Wine*

Копия набора данных *Wine* (и всех других наборов данных, используемых в книге) включена в состав загружаемого архива с кодом примеров для книги. Указанные копии можно задействовать при автономной работе или в случае временной недоступности <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data> на сервере UCI. Скажем, чтобы загрузить набор данных *Wine* из какого-то локального каталога, следующий оператор:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases/wine/wine.data',
                 header=None)
```

понадобится заменить таким оператором:

```
df = pd.read_csv('ваш/локальный/путь/к/wine.data',
                 header=None)
```

С помощью библиотеки `pandas` мы будем читать набор данных `Wine` непосредственно из Хранилища машинного обучения `UCI`:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/'
                          'ml/machine-learning-databases/'
                          'wine/wine.data', header=None)
>>> df_wine.columns = ['Метка класса', 'Алкоголь',
                        # 'Class label', 'Alcohol'
                        'Яблочная кислота', 'Зола',
                        # 'Malic acid', 'Ash'
                        'Щелочность золы', 'Магний',
                        # 'Alcalinity of ash', 'Magnesium'
                        'Всего фенолов', 'Флавоноиды',
                        # 'Total phenols', 'Flavanoids'
                        'Нефлавоноидные фенолы',
                        # 'Nonflavanoid phenols'
                        'Проантоцианидины',
                        # 'Proanthocyanins'
                        'Интенсивность цвета', 'Оттенок',
                        # 'Color intensity', 'Hue'
                        'OD280/OD315 разбавленных вин',
                        # 'OD280/OD315 of diluted wines'
                        'Пролин']
                        # 'Proline'
>>> print('Метки классов', np.unique(df_wine['Метка класса']))
Метки классов [1 2 3]
>>> df_wine.head()
```

Ниже в таблице перечислены 13 признаков в наборе данных `Wine`, которые описывают химические свойства 178 образцов вин.

	Метка класса	Алкоголь	Яблочная кислота	Зола	Щелочность золы	Магний	Всего фенолов	Флавоноиды	Нефлавоноидные фенолы	Проантоцианидины	Интенсивность цвета	Оттенок	OD280/OD315 разбавленных вин	Пролин
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.85	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Образцы принадлежат одному из трех классов, 1, 2 и 3, которые относятся к трем разным видам винограда, выращенного в одном и том же регионе Италии, но с отличающимися винными культурами, как описано в сводке по набору данных (<https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names>).

Удобный способ случайного разбиения этого набора данных на отдельные испытательный и обучающий поднаборы предусматривает применение функции `train_test_split` из подмодуля `model_selection` библиотеки `scikit-learn`:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.3,
...                       random_state=0,
...                       stratify=y)
```

Сначала мы присваиваем переменной `X` представление в виде массива NumPy столбцов признаков 1–13, а переменной `y` — метки классов из первого столбца. Затем мы используем функцию `train_test_split` для случайного разделения `X` и `y` на обучающий и испытательный наборы данных. Установкой `test_size=0.3` мы присваиваем 30% образцов вин `X_test` и `y_test`, а оставшиеся 70% образцов вин — `X_train` и `y_train`. Предоставление массива меток классов `y` как аргумента для `stratify` гарантирует, что и обучающий, и испытательный набор данных имеют такие же доли классов, как у исходного набора данных.



На заметку!

#### **Выбор подходящей пропорции для разделения набора данных на обучающий и испытательный наборы**

При разделении набора данных на обучающий и испытательный наборы следует иметь в виду, что мы удерживаем ценную информацию, из которой алгоритм обучения мог бы извлечь выгоду. Таким образом, выносить слишком много информации в испытательный набор нежелательно. Но чем меньше испытательный набор, тем более неточной будет оценка ошибки обобщения. Вся суть разделения набора данных на обучающий и испытательный наборы — уравнивание такого компромисса. На практике самыми часто применяемыми разделениями являются 60:40, 70:30 и 80:20 в зависимости от



размера первоначального набора данных. Однако для крупных наборов данных разделение 90:10 или 99:1 на обучающий и испытательный наборы также оказываются обычными и подходящими. Скажем, если набор данных содержит свыше 100 000 обучающих образцов, то может оказаться неплохим решение удерживать только 10 000 образцов для испытаний, чтобы получить хорошую оценку эффективности обобщения. Дополнительные сведения и иллюстрации ищите в первом разделе статьи “Model evaluation, model selection, and algorithm selection in machine learning” (“Оценка моделей, подбор моделей и выбор алгоритмов в машинном обучении”), которая свободно доступна по ссылке <https://arxiv.org/pdf/1811.12808.pdf>.

Кроме того, вместо отбрасывания выделенных испытательных данных после обучения и оценки модели классификатор обучают повторно на полном наборе данных, т.к. это может улучшить эффективность прогнозирования модели. Наряду с тем, что такой подход, как правило, рекомендуется, он может приводить к худшей эффективности обобщения, если набор данных мал и испытательный набор содержит выбросы, например. К тому же после повторной подгонки модели на полном наборе данных не остается никаких независимых данных для оценки ее эффективности.

## Приведение признаков к тому же самому масштабу

*Масштабирование признаков (feature scaling)* — критически важный шаг в конвейере предварительной обработки, о котором можно легко забыть. *Деревья принятия решений и случайные леса* представляют собой два из очень немногих алгоритмов МО, где не нужно беспокоиться о масштабировании признаков. Такие алгоритмы инвариантны к масштабу. Тем не менее, большинство алгоритмов МО и оптимизации работают гораздо лучше, когда признаки имеют один и тот же масштаб, как было показано в главе 2 при реализации алгоритма *оптимизации на основе градиентного спуска*.

Важность масштабирования признаков можно проиллюстрировать на простом примере. Пусть есть два признака, причем первый измеряется в масштабе от 1 до 10, а второй — в масштабе от 1 до 100 000. Вспомнив функцию суммы квадратичных ошибок в Adaline из главы 2, разумно говорить, что алгоритм будет занят главным образом оптимизацией весов на основании более крупных ошибок во втором признаке. Еще одним примером

является алгоритм *k* ближайших соседей (KNN) с мерой в виде евклидова расстояния; вычисленные расстояния между образцами будут преобладать на оси второго признака.

Существуют два общих подхода к приведению признаков к тому же самому масштабу: *нормализация* (*normalization*) и *стандартизация* (*standardization*). Указанные термины довольно свободно используются в разных областях, поэтому их смысл приходится выводить из контекста. Чаще всего под нормализацией понимается изменение масштаба признаков на диапазон  $[0, 1]$ , которое представляет собой частный случай *масштабирования по минимаксу* (*min-max scaling*). Для нормализации данных мы можем просто применить масштабирование по минимаксу к каждому столбцу признака, где новое значение  $x_{norm}^{(i)}$  образца  $x^{(i)}$  может быть вычислено как:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}$$

Здесь  $x^{(i)}$  — индивидуальный образец,  $x_{\min}$  — наименьшее значение в столбце признака, а  $x_{\max}$  — наибольшее значение в столбце признака.

Процедура масштабирования по минимаксу реализована в библиотеке *scikit-learn* и может использоваться следующим образом:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Несмотря на то что нормализация посредством масштабирования по минимаксу является ходовым приемом, который полезен, когда необходимы значения в ограниченном интервале, стандартизация может быть более практичной для многих алгоритмов МО, особенно для алгоритмов оптимизации наподобие градиентного спуска. Причина в том, что многие линейные модели, такие как рассмотренные в главе 3 логистическая регрессия и SVM, инициализируют веса нулями или небольшими случайными значениями, близкими к нулю. С помощью стандартизации мы центрируем столбцы признаков относительно среднего значения 0 со стандартным отклонением 1, так что столбцы признаков имеют те же параметры, как нормальное распределение (нулевое среднее и единичная дисперсия), облегчая выяснение весов. Кроме того, стандартизация сохраняет полезную информацию о вы-

бросах и делает алгоритм менее чувствительным к ним в отличие от масштабирования по минимаксу, которое приводит данные к ограниченному диапазону значений.

Процедура стандартизации может быть выражена с помощью такого уравнения:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Здесь  $\mu_x$  — выборочное среднее по определенному столбцу признака, а  $\sigma_x$  — соответствующее стандартное отклонение.

В представленной ниже таблице демонстрируется разница между двумя самыми распространенными приемами масштабирования признаков, стандартизацией и нормализацией, на простом наборе данных, состоящем из чисел от 0 до 5.

Входное значение	Стандартизированное значение	Нормализованное значение
0.0	-1.46385	0.0
1.0	-0.87831	0.2
2.0	-0.29277	0.4
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

Продемонстрированную в таблице стандартизацию и нормализацию можно выполнить вручную посредством следующего кода:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('стандартизированные значения:', (ex - ex.mean()) / ex.std())
стандартизированные значения: [-1.46385011 -0.87831007
                               -0.29277002  0.29277002
                               0.87831007  1.46385011]

>>> print('нормализованные значения:',
          (ex - ex.min()) / (ex.max() - ex.min()))
нормализованные значения: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Подобно классу `MinMaxScaler` в `scikit-learn` также реализован класс для стандартизации:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Также важно отметить, что мы подгоняем класс `StandardScaler` только один раз — на обучающих данных — и применяем найденные параметры для трансформирования испытательного набора или любой новой точки данных.

В библиотеке `scikit-learn` доступные и другие, более развитые методы для масштабирования признаков, такие как класс `RobustScaler`. Класс `RobustScaler` особенно удобен и рекомендуется в случае работы с небольшими наборами данных, которые содержат много выбросов. Аналогично, если алгоритм МО, применяемый к этому набору данных, устойчив к *переобучению*, то класс `RobustScaler` может оказаться хорошим вариантом. Работая с каждым столбцом признака независимо, класс `RobustScaler` удаляет медианное значение и масштабирует набор данных в соответствии с 1-м и 3-м квартилем набора данных (т.е. 25-й и 75-й квантили), так что более экстремальные значения и выбросы становятся менее резко выраженными. Заинтересованные читатели могут найти дополнительную информацию о классе `RobustScaler` в официальной документации по `scikit-learn`: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html>.

## Выбор значимых признаков

Если мы замечаем, что модель работает гораздо лучше на обучающем наборе данных, чем на испытательном наборе данных, то такое наблюдение является явным сигналом *переобучения*. Как обсуждалось в главе 3, переобучение означает, что модель слишком сильно подгоняется к параметрам касаясь отдельных наблюдений в обучающем наборе данных, но плохо обобщается на новые данные; мы говорим, что модель имеет *высокую дисперсию*. Причина переобучения состоит в том, что модель является слишком сложной для имеющихся обучающих данных. Ниже перечислены общепринятые решения, направленные на сокращение ошибки обобщения:

- накопление большего объема обучающих данных;
- введение штрафа за сложность через регуляризацию;

- выбор более простой модели с меньшим числом параметров;
- понижение размерности данных.

Накопление большего объема обучающих данных часто неприменимо. В главе 6 мы рассмотрим удобный прием, который позволяет проверить, полезны ли вообще дополнительные обучающие данные. В последующих разделах мы взглянем на распространенные способы сокращения переобучения за счет регуляризации и понижения размерности посредством выбора признаков, что приводит к более простой модели, которая требует меньшего количества параметров, подлежащих подгонке к данным.

## Регуляризация L1 и L2 как штрафы за сложность модели

Как известно из главы 3, *регуляризация L2* представляет собой подход к сокращению сложности модели путем штрафования крупных индивидуальных весов, при котором мы определяем норму L2 весового вектора  $\mathbf{w}$  следующим образом:

$$L2: \|\mathbf{w}\|_2^2 = \sum_{j=1}^m w_j^2$$

Другой подход к сокращению сложности модели — родственная *регуляризация L1*:

$$L1: \|\mathbf{w}\|_1 = \sum_{j=1}^m |w_j|$$

Здесь мы просто заменили сумму квадратов весов суммой абсолютных величин весов. В отличие от регуляризации L2 регуляризация L1 обычно выдает разреженные векторы признаков, и большинство весов будут нулевыми. Разреженность может быть практически полезной, если мы имеем набор данных высокой размерности с многочисленными признаками, не относящимися к делу, особенно в случаях, когда неподходящих измерений больше, чем обучающих образцов. В этом смысле регуляризацию L1 можно понимать как прием для выбора признаков.

## Геометрическая интерпретация регуляризации L2

Как упоминалось в предыдущем разделе, регуляризация L2 добавляет к функции издержек член штрафа, который фактически приведет к менее экс-

тремальным значениям весов по сравнению с моделью, обученной с нерегуляризированной функцией издержек.

Чтобы лучше понять, каким образом регуляризация L1 поддерживает разреженность, давайте сделаем шаг назад и ознакомимся с геометрической интерпретацией регуляризации. Мы изобразим контуры выпуклой функции издержек для двух весовых коэффициентов  $w_1$  и  $w_2$ .

Мы рассмотрим функцию издержек в виде суммы квадратичных ошибок (SSE), которая использовалась для Adaline в главе 2, т.к. она сферическая и удобнее в построении графика, чем функция издержек из логистической регрессии; однако к ней применимы те же самые концепции. Вспомните, что наша цель — отыскать такое сочетание весовых коэффициентов, которое сводит к минимуму функцию издержек для обучающих данных, как показано на рис. 4.3 (точка в центре эллипсов).

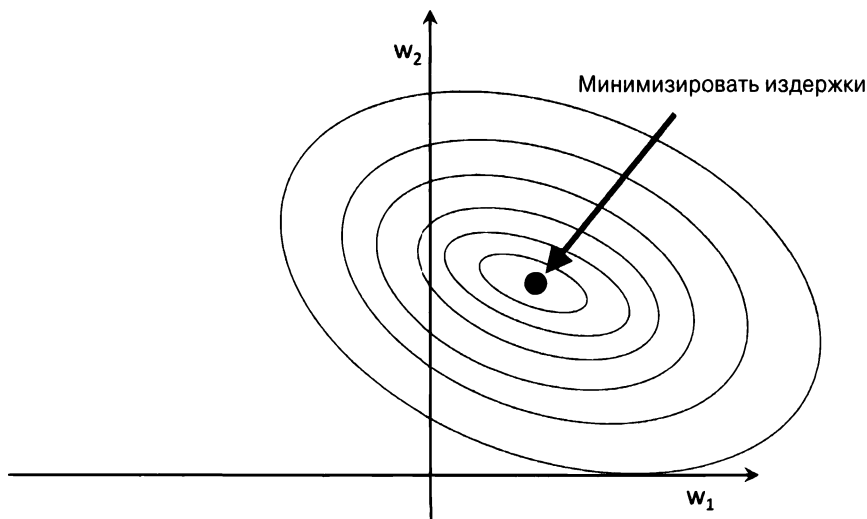
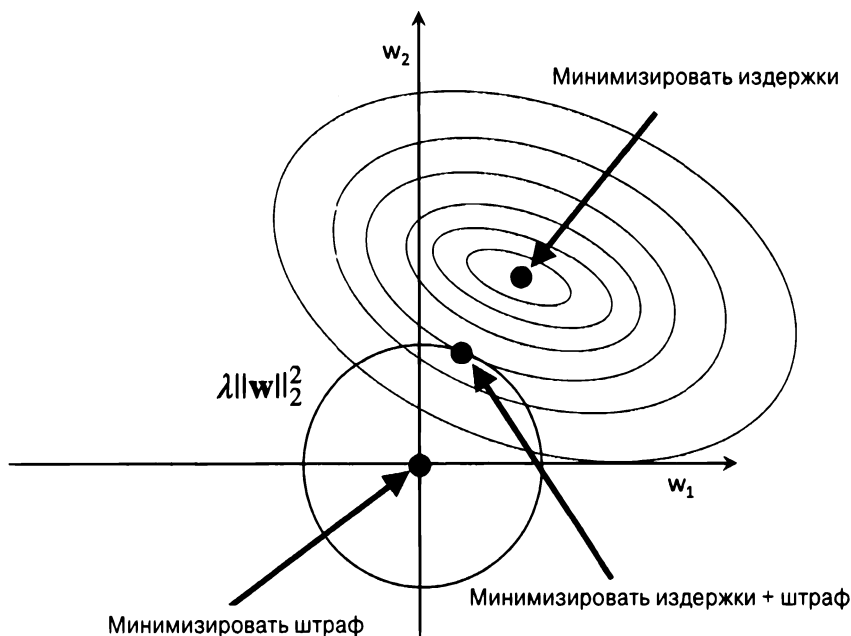


Рис. 4.3. Сведение к минимуму функции издержек для обучающих данных

Мы можем думать о регуляризации как о добавлении к функции издержек члена штрафа, чтобы поддерживать меньшие веса; другими словами, мы штрафуем крупные веса. Таким образом, за счет увеличения силы регуляризации через параметр регуляризации  $\lambda$  мы уменьшаем веса в сторону нуля и сокращаем зависимость модели от обучающих данных. На рис. 4.4 эта концепция иллюстрируется для члена штрафа при регуляризации L2.

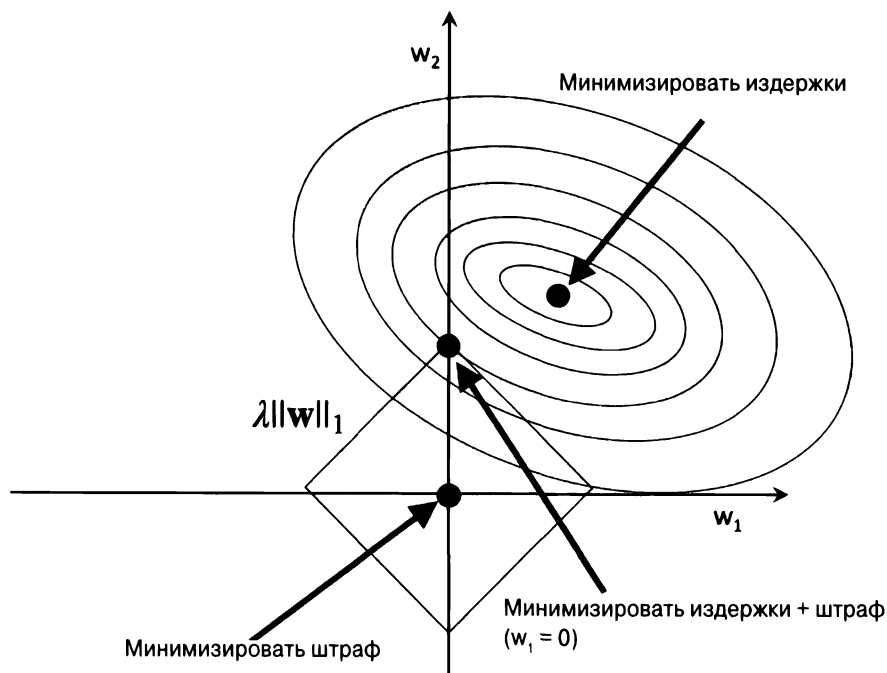


**Рис. 4.4.** Увеличение силы регуляризации через параметр регуляризации  $\lambda$

Квадратичный член штрафа регуляризации L2 представлен на рис. 4.4 затененным кругом. Наши весовые коэффициенты не могут превысить запас регуляризации, т.е. сочетание весовых коэффициентов не способно покинуть затененную область. С другой стороны, мы по-прежнему хотим минимизировать функцию издержек. В условиях ограничения в виде штрафа самое лучшее, что мы можем предпринять — выбрать точку, где круг регуляризации L2 пересекается с контурами функции издержек, не обложенной штрафом. Чем большее значение получает параметр регуляризации  $\lambda$ , тем быстрее растут издержки, обложенные штрафом, что приводит к меньшему кругу регуляризации L2. Например, если увеличивать параметр регуляризации  $\lambda$  в направлении бесконечности, то весовые коэффициенты станут фактически нулевыми, что отмечается центром круга регуляризации L2. Итак, наша цель в рассмотренном примере — минимизировать сумму издержек, не обложенных штрафом, плюс член штрафа. Это можно понимать как добавление смещения и поддержку более простой модели, что сократит дисперсию в условиях отсутствия достаточного объема обучающих данных для подгонки модели.

## Разреженные решения с регуляризацией L1

Теперь давайте обсудим регуляризацию L1 и разреженность. Основная идея регуляризации L1 похожа на ту, что мы раскрывали в предыдущем разделе. Тем не менее, поскольку штраф регуляризации L1 выражается суммой абсолютных значений весовых коэффициентов (вспомните, что член регуляризации L2 является квадратичным), мы можем представить его как запас ромбовидной формы (рис. 4.5).



**Рис. 4.5.** Графическое представление регуляризации L1

На рис. 4.5 видно, что контур функции издержек касается ромба регуляризации L1 в точке  $w_1 = 0$ . Так как контуры системы, подверженной регуляризации L1, оказываются остроконечными, более вероятно, что оптимум, т.е. пересечение эллипсов функции издержек и границы ромба регуляризации L1, расположится на осях, способствуя разреженности.





## Регуляризация L1 и разреженность

Выяснение математических деталей того, почему регуляризация L1 может приводить к разреженным решениям, выходит за рамки вопросов, рассматриваемых в книге. Если вам интересно, тогда можете ознакомиться с великолепным сравнением регуляризации L2 и L1 в разделе 3.4 книги “The Elements of Statistical Learning” (“Основы статистического обучения: интеллектуальный анализ данных, логический вывод и прогнозирование”, 2-е изд., пер. с англ., изд. “Диалектика”, 2020 г), написанной Тревором Хастем, Робертом Тибширани и Джеромом Фридманом (Springer Science+Business Media (2009 год)).

Для регуляризированных моделей, поддерживающих регуляризацию L1, в `scikit-learn` мы можем просто установить параметр `penalty` в `'l1'` и получить разреженное решение:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1',
...                    solver='liblinear',
...                    multi_class='ovr')
```

Обратите внимание, что нам также необходимо выбрать какой-то другой алгоритм оптимизации (скажем, `solver='liblinear'`), т.к. `'lbfgs'` в текущий момент не поддерживает оптимизацию потерь с регуляризацией L1. Логистическая регрессия с регуляризацией L1, примененная к стандартизированному набору данных `Wine`, выдаст следующее разреженное решение:

```
>>> lr = LogisticRegression(penalty='l1',
...                          C=1.0,
...                          solver='liblinear',
...                          multi_class='ovr')
# Следует отметить, что C=1.0 принимается по умолчанию.
# Вы можете увеличить или уменьшить значение C, чтобы сделать
# эффект регуляризации соответственно сильнее или слабее.
>>> lr.fit(X_train_std, y_train)
>>> print('Правильность при обучении:', lr.score(X_train_std, y_train))
Правильность при обучении: 1.0
>>> print('Правильность при испытании:', lr.score(X_test_std, y_test))
Правильность при испытании: 1.0
```

Правильность при обучении и правильность при испытании (составляющие 100%) указывают на то, что наша модель идеально работает на обоих наборах данных. В результате обращения к членам пересечения через атри-

бут `lr.intercept_` мы можем заметить, что возвращается массив с тремя значениями:

```
>>> lr.intercept_
array([-1.26346036, -1.21584018, -2.3697841 ])
```

Поскольку мы подгоняем объект `LogisticRegression` к многоклассовому набору данных посредством подхода “один против остальных” (OvR), первое значение — это пересечение модели, которая подгоняется к классу 1 против классов 2 и 3, второе значение является пересечением модели, подогнанной к классу 2 против классов 1 и 3, а третье значение представляет собой пересечение модели, подогнанной к классу 3 против классов 1 и 2:

```
>>> lr.coef_
array([[ 1.24590762,  0.18070219,  0.74375939, -1.16141503,
         0.          ,  0.          ,  1.16926815,  0.          ,
         0.          ,  0.          ,  0.          ,  0.54784923,
         2.51028042],
       [-1.53680415, -0.38795309, -0.99494046,  0.36508729,
        -0.05981561,  0.          ,  0.6681573 ,  0.          ,
         0.          , -1.93426485,  1.23265994,  0.          ,
        -2.23137595],
       [ 0.13547047,  0.16873019,  0.35728003,  0.          ,
         0.          ,  0.          , -2.43713947,  0.          ,
         0.          ,  1.56351492, -0.81894749, -0.49308407,
         0.          ]])
```

Массив весов, к которому мы получаем доступ через атрибут `lr.coef_`, содержит три строки весовых коэффициентов, по одному весовому вектору для каждого класса. Каждая строка состоит из 13 весов, и для вычисления общего входа каждый вес умножается на соответствующий признак в 13-мерном наборе данных Wine:

$$z = w_0x_0 + \dots + w_mx_m = \sum_{j=0}^m x_j w_j = \mathbf{w}^T \mathbf{x}$$



На  
заметку!

В `scikit-learn` атрибут `intercept_` соответствует  $w_0$ , а `coef_` — значениям  $w_j$  для  $j > 0$ .

В результате выполнения регуляризации L1, служащей методом для выбора признаков, мы просто обучаем модель, которая устойчива к потенциально не относящимся к делу признакам в наборе данных Wine. Но, строго

говоря, весовые векторы из предыдущего примера не обязательно являются разреженными, т.к. они содержат больше ненулевых, чем нулевых элементов. Однако мы могли бы навязать разреженность (увеличить число нулевых элементов), дополнительно повышая силу регуляризации, т.е. выбирая меньшие значения для параметра  $C$ .

В последнем примере главы, посвященном регуляризации, мы варьируем силу регуляризации и строим график пути регуляризации — весовых коэффициентов различных признаков для разных значений силы регуляризации:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...           'magenta', 'yellow', 'black',
...           'pink', 'lightgreen', 'lightblue',
...           'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4., 6.):
...     lr = LogisticRegression(penalty='l1', C=10.**c,
...                             solver='liblinear',
...                             multi_class='ovr', random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column + 1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('весовой коэффициент')
>>> plt.xlabel('C')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...          bbox_to_anchor=(1.38, 1.03),
...          ncol=1, fancybox=True)
>>> plt.show()
```

Результирующий график, показанный на рис. 4.6, содействует лучшему пониманию поведения регуляризации L1. Мы можем заметить, что веса всех признаков будут нулевыми, если штрафовать модель с помощью сильного параметра регуляризации ( $C < 0.01$ );  $C$  — инверсия параметра регуляризации  $\lambda$ .

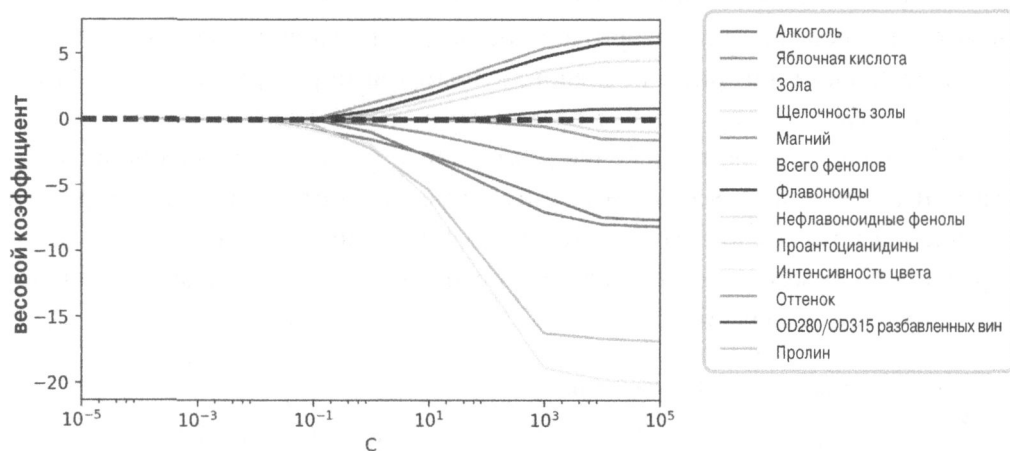


Рис. 4.6. График пути регуляризации

## Алгоритмы последовательного выбора признаков

Альтернативный способ сокращения сложности модели и избегания переобучения предусматривает *понижение размерности* (*dimensionality reduction*) посредством выбора признаков, что особенно полезно для нерегуляризованных моделей. Существуют две главные категории приемов понижения размерности: *выбор признаков* (*feature selection*) и *выделение признаков* (*feature extraction*). При выборе признаков мы отбираем подмножество исходных признаков, тогда как при выделении признаков мы выводим информацию из набора признаков для построения нового подпространства признаков.

В текущем разделе мы рассмотрим классическое семейство алгоритмов выбора признаков. В следующей главе мы исследуем различные приемы выделения признаков для сжатия набора данных в подпространство признаков меньшей размерности.

Алгоритмы последовательного выбора признаков — это семейство поглощающих (“жадных”) алгоритмов поиска, которые применяются для по-

нижения первоначального  $d$ -мерного пространства признаков до  $k$ -мерного подпространства признаков, где  $k < d$ . Мотивацией, лежащей в основе алгоритмов выбора признаков, является автоматический отбор подмножества признаков, наиболее значимых для задачи. Цель такого отбора — повысить вычислительную эффективность либо снизить ошибку обобщения модели за счет удаления не имеющих отношения к делу признаков или шума, что может быть полезно для алгоритмов, не поддерживающих регуляризацию.

Классический алгоритм последовательного выбора признаков называется *последовательным обратным выбором* (*sequential backward selection* — SBS). Он направлен на понижение размерности первоначального пространства признаков с минимальным спадом эффективности классификатора для улучшения показателей вычислительной эффективности. В ряде случаев алгоритм SBS может даже усилить прогнозирующую мощь модели, если модель страдает от переобучения.



На  
заметку!

### Поглощающие алгоритмы поиска

*Поглощающие* (“жадные”) алгоритмы делают локально оптимальные отборы на каждой стадии задачи комбинаторного поиска и в целом дают субоптимальное решение задачи в отличие от *алгоритмов исчерпывающего поиска*, которые оценивают все возможные комбинации и гарантируют нахождение оптимального решения. Тем не менее, на практике исчерпывающий поиск часто неосуществим в вычислительном плане, тогда как поглощающие алгоритмы делают возможным менее сложное и более эффективное с вычислительной точки зрения решение.

Идея, лежащая в основе алгоритма SBS, довольно проста: алгоритм SBS последовательно удаляет признаки из полного набора признаков до тех пор, пока новое подпространство признаков не станет содержать желательное количество признаков. Для установления, какой признак должен быть удален на каждой стадии, нам необходимо определить функцию критерия  $J$  и свести ее к минимуму. Вычисляемым этой функцией критерием может быть просто разница в эффективности классификатора до и после удаления индивидуального признака. Признаком, подлежащим удалению на каждой стадии, будет тот, который доводит до максимума данный критерий; или, выражаясь более понятными терминами, на каждой стадии мы исключаем признак, удаление которого приводит к наименьшей потере эффективности.

Опираясь на предыдущее определение SBS, мы можем представить алгоритм в виде четырех простых шагов.

1. Инициализировать алгоритм с  $k = d$ , где  $d$  – размерность полного пространства признаков  $\mathbf{X}_d$ .
2. Определить признак  $x^-$ , который доводит до максимума критерий  $x^- = \operatorname{argmax} J(\mathbf{X}_k - x)$ , где  $x \in \mathbf{X}_k$ .
3. Удалить признак  $x^-$  из набора признаков  $\mathbf{X}_{k+1} = \mathbf{X}_k - x^-$ ;  $k = k - 1$ .
4. Закончить, если  $k$  равно желательному количеству признаков; в противном случае перейти к шагу 2.



На  
заметку!

### Ресурс по алгоритмам последовательного выбора признаков

Подробный анализ некоторых алгоритмов последовательного выбора признаков можно найти в работе Ф. Ферри, П. Пудила, М. Хатефа и И. Киттлера “Comparative Study of Techniques for Large-Scale Feature Selection” (Сравнительное изучение приемов крупномасштабного выбора признаков), с. 403–413 (1994 г.).

К сожалению, алгоритм SBS в библиотеке scikit-learn пока еще не реализован. Но поскольку он довольно прост, давайте реализуем его на Python самостоятельно:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

class SBS():
    def __init__(self, estimator, k_features,
                 scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                            random_state=self.random_state)
```

```
dim = X_train.shape[1]
self.indices_ = tuple(range(dim))
self.subsets_ = [self.indices_]
score = self._calc_score(X_train, y_train,
                        X_test, y_test, self.indices_)
self.scores_ = [score]

while dim > self.k_features:
    scores = []
    subsets = []

    for p in combinations(self.indices_, r=dim - 1):
        score = self._calc_score(X_train, y_train,
                                X_test, y_test, p)

        scores.append(score)
        subsets.append(p)

    best = np.argmax(scores)
    self.indices_ = subsets[best]
    self.subsets_.append(self.indices_)
    dim -= 1

    self.scores_.append(scores[best])
self.k_score_ = self.scores_[-1]

return self

def transform(self, X):
    return X[:, self.indices_]

def _calc_score(self, X_train, y_train, X_test, y_test, indices):
    self.estimator.fit(X_train[:, indices], y_train)
    y_pred = self.estimator.predict(X_test[:, indices])
    score = self.scoring(y_test, y_pred)
    return score
```

В приведенной выше реализации мы определяем параметр `k_features` для указания желательного числа признаков, которые нужно возвратить. По умолчанию для оценки эффективности модели (оценщиком для классификации) на поднаборах признаков мы используем метрику `accuracy_score` из `scikit-learn`.

Внутри цикла `while` метода `fit` поднаборы признаков, созданные функцией `itertools.combination`, оцениваются и понижаются в размерности до тех пор, пока не приобретут желательную размерность. На каждой итерации показатель правильности наилучшего поднабора помещается в спи-

сок `self.scores_`, основанный на внутренне созданном испытательном наборе данных `X_test`. Мы задействуем эти показатели позже при оценке результатов. Индексы столбцов финального поднабора признаков присваиваются атрибуту `self.indices_`, который посредством метода `transform` можно применять для возвращения нового массива данных с выбранными столбцами признаков. Обратите внимание, что вместо вычисления критерия явно внутри метода `fit` мы просто удаляем признак, который отсутствует в поднаборе признаков с наилучшей эффективностью.

А теперь давайте посмотрим на нашу реализацию алгоритма SBS в работе, воспользовавшись классификатором KNN из `scikit-learn`:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Хотя наша реализация алгоритма SBS внутри функции `fit` уже разбивает набор данных на испытательный и обучающий поднаборы, мы по-прежнему передаем алгоритму обучающий набор данных `X_train`. Метод `fit` алгоритма SBS затем создает новые обучающие поднаборы для испытаний (проверки) и обучения, из-за чего испытательный набор называют также *проверочным набором данных*. Такой подход необходим для того, чтобы не позволить нашему *исходному* испытательному набору стать частью обучающего набора.

Вспомните, что на каждой стадии наш алгоритм SBS собирает показатель правильности наилучшего поднабора признаков, поэтому давайте перейдем к более захватывающей части реализации и построим для классификатора KNN график правильности классификации, которая была подсчитана на проверочном наборе данных. Вот как выглядит код:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.02])
>>> plt.ylabel('Правильность')
>>> plt.xlabel('Количество признаков')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```



Как видно на рис. 4.7, с сокращением количества признаков показатель правильности классификатора KNN на проверочном наборе данных улучшился, что произошло, вероятно, из-за ослабления “проклятия размерности”, которое мы обсуждали при рассмотрении алгоритма KNN в главе 3. На графике также можно заметить, что классификатор достигает стопроцентной правильности для  $k = \{3, 7, 8, 9, 10, 11, 12\}$ .

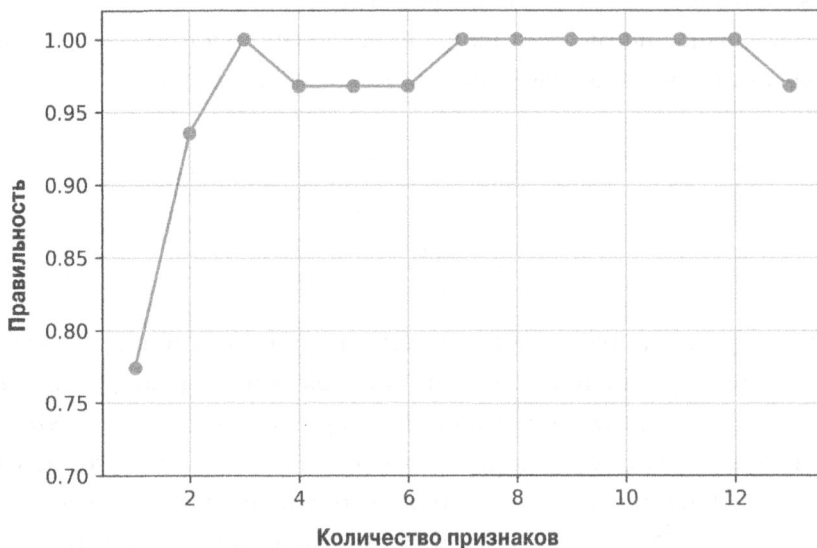


Рис. 4.7. График правильности классификатора KNN

Чтобы удовлетворить собственную любознательность, давайте посмотрим, на что похож наименьший поднабор признаков ( $k = 3$ ), который обеспечивает настолько хорошую эффективность на проверочном наборе данных:

```
>>> k3 = list(sbs.subsets_[10])
>>> print(df_wine.columns[1:][k3])
Index(['Алкоголь', 'Яблочная кислота', 'OD280/OD315 разбавленных вин'],
      dtype='object')
```

В предыдущем коде мы получаем индексы столбцов поднабора, включающего три признака, из 11-й позиции в атрибуте `sbs.subsets_` и возвращаем соответствующие имена признаков из столбцового индекса `pandas`-объекта `DataFrame` для набора данных `Wine`.

Далее оценим эффективность классификатора KNN на исходном испытательном наборе:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Правильность при обучении:',
          knn.score(X_train_std, y_train))
Правильность при обучении: 0.967741935484
>>> print('Правильность при испытании:',
          knn.score(X_test_std, y_test))
Правильность при испытании: 0.962962962963
```

В коде мы применяем полный набор признаков и получаем правильность почти 97% на обучающем наборе и правильность около 96% на испытательном наборе, что говорит о наличии у модели хорошей способности обобщения на новые данные. А теперь возьмем выбранный поднабор из трех признаков и посмотрим, насколько хорошо работает классификатор KNN:

```
>>> knn.fit(X_train_std[:, k3], y_train)
>>> print('Правильность при обучении:',
          ...      knn.score(X_train_std[:, k3], y_train))
Правильность при обучении: 0.951612903226
>>> print('Правильность при испытании:',
          ...      knn.score(X_test_std[:, k3], y_test))
Правильность при испытании: 0.925925925926
```

При использовании менее четверти исходных признаков в наборе данных Wine правильность прогноза на испытательном наборе слегка ухудшилась. Это может указывать на то, что выбранные три признака не предоставляют менее отличительную информацию, чем исходный набор данных. Однако мы также должны иметь в виду, что набор данных Wine имеет небольшой размер, а потому он крайне чувствителен к случайности, т.е. к способу его разделения на обучающий и испытательный поднаборы, и к тому, каким образом обучающий набор данных дополнительно делится на обучающий и проверочный поднаборы.

Хотя мы не увеличили эффективность модели KNN, уменьшив количество признаков, но сократили размер набора данных, что может быть полезно в реальных приложениях, которые нередко реализуют затратные шаги сбора данных. К тому же, существенно уменьшая количество признаков, мы получаем более простые модели, которые легче для интерпретации.



## Алгоритмы выбора признаков в библиотеке scikit-learn

В библиотеке scikit-learn доступно много других алгоритмов выбора признаков. В их число входят *рекурсивное обратное исключение* (*recursive backward elimination*) на основе весов признаков, методы выбора признаков по важности, основанные на деревьях, и одномерные статистические критерии. Всеобъемлющее обсуждение разнообразных методов выбора признаков выходит за рамки этой книги, но хорошую сводку с иллюстративными примерами можно найти по ссылке [http://scikit-learn.org/stable/modules/feature\\_selection.html](http://scikit-learn.org/stable/modules/feature_selection.html). Реализации нескольких вариантов последовательного выбора признаков, родственного реализованному ранее простому алгоритму SBS, доступны в пакете Python по имени mlxtend: [http://rasbt.github.io/mlxtend/user\\_guide/feature\\_selection/SequentialFeatureSelector/](http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/).

## Оценка важности признаков с помощью случайных лесов

В предшествующих разделах вы узнали, как применять регуляризацию L1 для обнуления не имеющих отношения к делу признаков через логистическую регрессию, а также как использовать алгоритм SBS для выбора признаков и применять его в классификаторе KNN. Еще один подход к выбору значимых признаков из набора данных предусматривает использование *случайного леса* — ансамблевого приема, который был введен в главе 3. Применяя случайный лес, мы можем оценивать важность признаков как усредненное уменьшение загрязненности, рассчитанное из всех деревьев принятия решений в лесе, не выдвигая никаких предположений о том, являются наши данные линейно сепарабельными или нет. Удобно то, что реализация случайных лесов в scikit-learn уже собирает показатели важности признаков, к которым можно получать доступ через атрибут `feature_importances_` после подгонки классификатора `RandomForestClassifier`. Выполнив приведенный ниже код, мы обучим лес из 10 000 деревьев на наборе данных Wine и расположим 13 признаков в порядке их показателей важности — вспомните из главы 3, что в моделях на основе деревьев использовать стандартизированные или нормализованные признаки не обязательно:

```

>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=500,
...                                random_state=1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
>>> plt.title('Важность признаков')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...             feat_labels[indices] rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()

```

1) Пролин	0.185453
2) Флавоноиды	0.174751
3) Интенсивность цвета	0.143920
4) OD280/OD315 разбавленных вин	0.136162
5) Алкоголь	0.118529
6) Оттенок	0.058739
7) Всего фенолов	0.050872
8) Магний	0.031357
9) Яблочная кислота	0.025648
10) Проантоцианидины	0.025570
11) Щелочность золы	0.022366
12) Нефлавоноидные фенолы	0.013354
13) Зола	0.013279

В результате выполнения кода отображается график, который располагает признаки в наборе данных Wine по их относительной важности (рис. 4.8); обратите внимание, что показатели важности признаков нормализованы, в сумме давая 1.0.

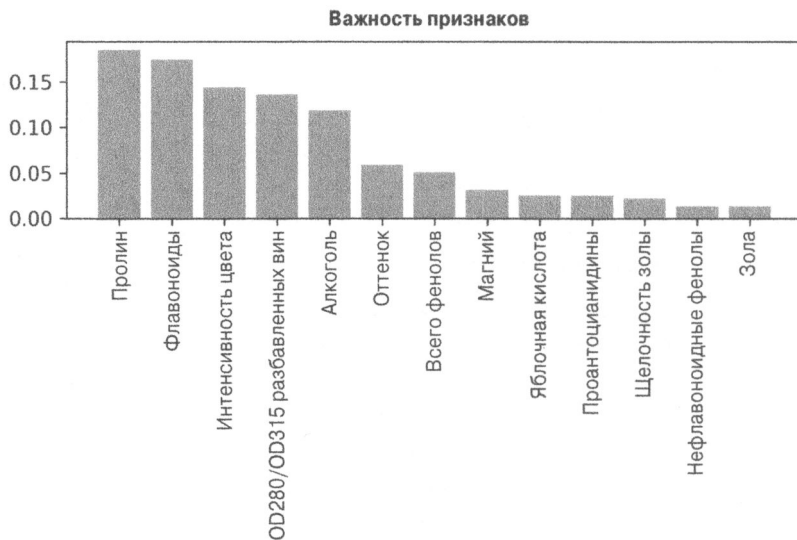


Рис. 4.8. График относительной важности признаков в наборе данных Wine

Мы можем сделать вывод, что уровни пролина и флавоноидов, интенсивность цвета, дифракция OD280/OD315 и концентрация алкоголя в вине являются самыми отличительными признаками в наборе данных, основываясь на усредненном уменьшении загрязненности в 500 деревьях принятия решений. Интересно отметить, что два признака из числа самых важных на графике также входят в состав выбранного поднабора с тремя признаками в показанной ранее реализации алгоритма SBS (концентрация алкоголя и OD280/OD315 разбавленных вин). Тем не менее, что касается интерпретируемости, то прием со случайными лесами сопровождается важным *затруднением*, заслуживающим особого упоминания. Если два или большее количество признаков сильно связаны друг с другом, тогда один признак может получить очень высокую важность, в то время как информация о другом признаке (либо признаках) может быть собрана не полностью. С другой стороны, нам нет нужды беспокоиться об этой проблеме, если нас интересует только эффективность прогнозирования модели, а не интерпретация показателей важности признаков.

В завершение раздела о показателях важности признаков и случайных лесах полезно отметить, что в библиотеке `scikit-learn` также реализован объект `SelectFromModel`, позволяющий выбирать признаки на основе указанного пользователем порога после подгонки модели. Объект `SelectFromModel` удобен, когда мы хотим применять объект `RandomForestClassifier` для

выбора признаков и в качестве промежуточного шага в объекте Pipeline из scikit-learn, который позволяет связывать различные шаги предварительной обработки с оценщиком, как будет показано в главе 6. Например, мы могли бы установить threshold в 0.1, чтобы сократить набор данных до пяти самых важных признаков:

```
>>> from sklearn.feature_selection import SelectFromModel
>>> sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
>>> X_selected = sfm.transform(X_train)
>>> print('Количество признаков, удовлетворяющих данному
критерию порога:',
...       X_selected.shape[1])
Количество признаков, удовлетворяющих данному критерию порога: 5
>>> for f in range(X_selected.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
1) Пролин                      0.185453
2) Флавоноиды                  0.174751
3) Интенсивность цвета          0.143920
4) OD280/OD315 разбавленных вин 0.136162
5) Алкоголь                     0.118529
```

## Резюме

В начале главы мы рассмотрели полезные приемы, обеспечивающие корректную обработку недостающих данных. Прежде чем передавать данные алгоритму МО, мы также должны удостовериться в правильности кодировки категориальных переменных, поэтому было показано, как отображать именные и порядковые признаки на целочисленные представления.

Кроме того, мы кратко обсудили регуляризацию L1, которая может помочь избежать переобучения за счет уменьшения сложности модели. В качестве альтернативы удалению признаков, не имеющих отношения к делу, мы использовали алгоритм последовательного выбора признаков для отбора значимых признаков из набора данных.

В следующей главе вы узнаете о другом удобном подходе к понижению размерности: выделении признаков. Он позволяет сжимать признаки в подпространство меньшей размерности, а не полностью удалять признаки как при выборе признаков.



# СЖАТИЕ ДАННЫХ С ПОМОЩЬЮ ПОНИЖЕНИЯ РАЗМЕРНОСТИ

В главе 4 вы узнали о различных подходах к понижению размерности набора данных с использованием разнообразных приемов выбора признаков. Подход к понижению размерности, альтернативный выбору признаков, называется *выделением признаков*. В этой главе будут рассматриваться три фундаментальных приема, которые помогут подытожить информационное содержимое набора данных, трансформируя его в новое подпространство признаков с меньшей, чем у исходного набора данных размерностью. Сжатие данных — важная тема в МО; сжатие содействует более эффективному хранению и анализу растущих объемов данных, которые вырабатываются и накапливаются в современную технологическую эпоху.

В главе будут раскрыты следующие темы:

- *анализ главных компонент (principal component analysis — PCA)* для сжатия данных без учителя;
- *линейный дискриминантный анализ (linear discriminant analysis — LDA)* как прием понижения размерности без учителя для доведения до максимума сепарабельности классов;
- *нелинейное понижение размерности посредством ядерного анализа главных компонент (kernel principal component analysis — KPCA)*.



## Понижение размерности без учителя с помощью анализа главных компонент

Подобно выбору признаков различные приемы выделения признаков можно применять для сокращения количества признаков в наборе данных. Отличие между выбором и выделением признаков связано с тем, что если при использовании алгоритмов выбора признаков, таких как *последовательный обратный выбор*, первоначальные признаки сохраняются, то выделение признаков применяется для трансформирования или проецирования данных в новое пространство признаков.

В контексте понижения размерности выделение признаков можно понимать как подход к сжатию данных с целью сохранения большей части значимой информации. На практике выделение признаков используется не только для улучшения характеристик пространства хранения или вычислительной продуктивности алгоритма обучения. Оно способно также повысить эффективность прогнозирования за счет ослабления “проклятия размерности” — особенно при работе с нерегуляризованными моделями.

### Основные шаги при анализе главных компонент

В текущем разделе мы обсудим PCA — прием линейной трансформации без учителя, который широко применяется в разнообразных областях, наиболее значительными из которых являются выделение признаков и понижение размерности. Другие популярные области использования PCA включают исследовательский анализ данных и устранение шумов в сигналах при торговле на фондовой бирже, а также анализ данных о геномах и уровнях проявления генов в биоинформатике.

Анализ главных компонент помогает распознавать шаблоны в данных, основываясь на корреляции между признаками. Выражаясь кратко, анализ PCA нацелен на обнаружение направлений максимальной дисперсии в данных высокой размерности и проецирование их в новое подпространство с равным или меньшим числом измерений. Ортогональные оси (главные компоненты) нового подпространства можно интерпретировать как направления максимальной дисперсии при условии ограничения, что новые оси признаков ортогональны друг к другу (рис. 5.1).

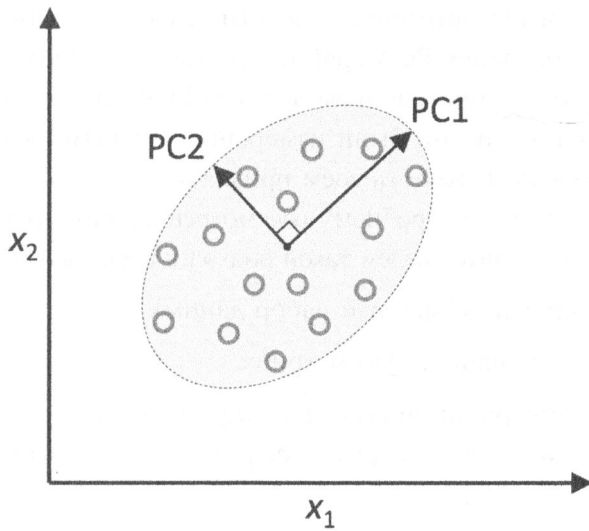


Рис. 5.1. Главные компоненты

Здесь  $x_1$  и  $x_2$  — это исходные оси признаков, а **PC1** и **PC2** — главные компоненты.

Когда мы применяем PCA для понижения размерности, то строим  $(d \times k)$ -мерную матрицу трансформации **W**, позволяющую отобразить вектор  $x$  с признаками обучающего образца на новое  $k$ -мерное подпространство признаков, которое имеет меньше измерений, чем исходное  $d$ -мерное пространство признаков. Ниже описан процесс. Предположим, у нас есть вектор признаков  $x$ :

$$x = [x_1, x_2, \dots, x_d], \quad x \in \mathbb{R}^d$$

который затем трансформируется посредством матрицы трансформации  $W \in \mathbb{R}^{d \times k}$ :

$$x W = z$$

давая выходной вектор:

$$z = [z_1, z_2, \dots, z_k], \quad z \in \mathbb{R}^k$$

В результате трансформации исходных  $d$ -мерных данных в новое  $k$ -мерное подпространство (обычно  $k \ll d$ ) первый главный компонент будет располагать наибольшей дисперсией из возможных. Все последующие главные компоненты будут иметь наибольшую дисперсию при условии, что они не взаимосвязаны (ортогональны) с другими главными компонентами — даже если входные признаки взаимосвязаны, то результирующие главные ком-

поненты будут взаимно ортогональными (не взаимосвязанными). Обратите внимание, что направления PCA крайне чувствительны к масштабированию данных, поэтому *перед* проведением анализа PCA нам придется стандартизировать признаки, если они были измерены с разными масштабами и мы хотим придать равную важность всем признакам.

Прежде чем перейти к подробному рассмотрению алгоритма PCA для понижения размерности, давайте сведем такой подход к нескольким простым шагам.

1. Стандартизировать  $d$ -мерный набор данных.
2. Построить ковариационную матрицу.
3. Разложить ковариационную матрицу на ее *собственные векторы* (*eigenvector*) и *собственные значения* (*eigenvalue*; характеристические числа).
4. Отсортировать собственные значения в порядке убывания, чтобы ранжировать соответствующие собственные векторы.
5. Выбрать  $k$  собственных векторов, которые соответствуют  $k$  наибольшим собственным значениям, где  $k$  — размерность нового подпространства признаков ( $k \leq d$ ).
6. Построить матрицу проекции  $W$  из “верхних”  $k$  собственных векторов.
7. Трансформировать  $d$ -мерный входной набор данных  $X$  с использованием матрицы проекции  $W$ , чтобы получить новое  $k$ -мерное подпространство признаков.

Далее в главе мы пошагово выполним алгоритм PCA с применением Python в виде учебного упражнения. Затем мы покажем, как более удобно выполнить алгоритм PCA с помощью `scikit-learn`.

## Выделение главных компонент шаг за шагом

В этом подразделе мы займемся первыми четырьмя шагами алгоритма PCA:

- 1) стандартизация данных;
- 2) построение ковариационной матрицы;
- 3) получение собственных значений и собственных векторов ковариационной матрицы;
- 4) сортировка собственных значений в порядке убывания для ранжирования собственных векторов.

Мы начнем с загрузки набора данных Wine, с которым работали в главе 4:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                       'machine-learning-databases/wine/wine.data',
...                       header=None)
```



На заметку!

### Получение набора данных Wine

Копия набора данных Wine (и всех других наборов данных, используемых в книге) включена в состав загружаемого архива с кодом примеров для книги. Указанные копии можно задействовать при автономной работе или в случае временной недоступности <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data> на сервере UCI. Скажем, чтобы загрузить набор данных Wine из какого-то локального каталога, следующий оператор:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases/wine/wine.data',
                 header=None)
```

понадобится заменить таким оператором:

```
df = pd.read_csv('ваш/локальный/путь/к/wine.data',
                 header=None)
```

Затем мы разделим данные Wine на обучающий и испытательный наборы (включающие 70% и 30% данных соответственно) и стандартизируем признаки, приведя к единичной дисперсии:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.3,
...                       stratify=y,
...                       random_state=0)
>>> # стандартизировать признаки
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

После завершения обязательной предварительной обработки путем выполнения приведенного выше кода мы переходим ко второму шагу: построению ковариационной матрицы. Симметричная  $(d \times d)$ -мерная ковариационная матрица, где  $d$  — количество измерений в наборе данных, хранит попарные ковариации между разными признаками. Например, ковариация между двумя признаками  $x_j$  и  $x_k$  на уровне генеральной совокупности может быть вычислена посредством следующего уравнения:

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k)$$

Здесь  $\mu_j$  и  $\mu_k$  — выборочные средние признаков  $j$  и  $k$ . Обратите внимание, что эти выборочные средние будут нулевыми, если набор данных стандартизирован. Положительная ковариация между двумя признаками указывает на то, что признаки вместе увеличиваются или уменьшаются, тогда как отрицательная — что признаки изменяются в противоположных направлениях. Скажем, ковариационную матрицу для трех признаков можно записать, как показано ниже (не путайте прописную греческую букву  $\Sigma$  (“сигма”) с символом, обозначающим сумму):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

Собственные векторы ковариационной матрицы представляют главные компоненты (направления максимальной дисперсии), а соответствующие собственные значения будут определять их величину. В случае набора данных Wine мы получим 13 собственных векторов и собственных значений из  $(13 \times 13)$ -мерной ковариационной матрицы.

Давайте для третьего шага получим собственные пары ковариационной матрицы. В качестве напоминания из вводного курса по линейной алгебре собственный вектор  $v$  удовлетворяет следующему условию:

$$\Sigma v = \lambda v$$

Здесь  $\lambda$  представляет собой скаляр — собственное значение. Поскольку ручное вычисление собственных векторов и собственных значений — отчасти утомительная и сложная задача, для получения собственных пар кова-

риационной матрицы Wine мы будем применять функцию `linalg.eig` из NumPy:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nСобственные значения \n%s' % eigen_vals)
Собственные значения
[ 4.84274532  2.41602459  1.54845825  0.96120438  0.84166161
  0.6620634   0.51828472  0.34650377  0.3131368   0.10754642
  0.21357215  0.15362835  0.1808613 ]
```

С использованием функции `numpy.cov` мы вычисляем ковариационную матрицу стандартизированного обучающего набора данных. С применением функции `linalg.eig` мы выполняем разложение по собственным значениям, что дает вектор (`eigen_vals`), состоящий из 13 собственных значений, и соответствующие собственные векторы, которые хранятся как столбцы в (13×13)-мерной ковариационной матрице (`eigen_vecs`).



На заметку!

### Разложение собственных значений в NumPy

Функция `numpy.linalg.eig` была спроектирована для работы с симметричными и несимметричными квадратными матрицами. Однако вы можете обнаружить, что в определенных случаях она возвращает комплексные собственные значения.

Связанная с `numpy.linalg.eig` функция, `numpy.linalg.eigh`, была реализована для разложения *эрмитовых* (*Hermitian*) матриц, которые обеспечивают численно более устойчивый подход к работе с симметричными матрицами, такими как ковариационная матрица; `numpy.linalg.eigh` всегда возвращает вещественные собственные значения.

## Полная и объясненная дисперсия

Так как мы хотим понизить размерность набора данных, сжимая его в новое подпространство признаков, мы выбираем только поднабор собственных векторов (главных компонент), которые содержат большую часть информации (дисперсии). Собственные значения определяют величину собственных векторов, поэтому мы должны отсортировать собственные значения в порядке убывания величины; нас интересуют верхние  $k$  собственных век-

торов на основе величин соответствующих собственных значений. Но прежде чем собирать  $k$  наиболее информативных собственных векторов, давайте построим график с *коэффициентами объясненной дисперсии* (*variance explained ratio*) собственных значений. Коэффициент объясненной дисперсии собственного значения  $\lambda_j$  представляет собой просто долю собственного значения  $\lambda_j$  в общей сумме собственных значений:

$$\text{Коэффициент объясненной дисперсии} = \frac{\lambda_j}{\sum_{j=1}^d \lambda_j}$$

Затем с использованием функции `cumsum` из NumPy мы можем подсчитать кумулятивную сумму объясненных дисперсий и отобразить ее на графике посредством функции `step` из Matplotlib:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...             sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
...         label='индивидуальная объясненная дисперсия')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...          label='кумулятивная объясненная дисперсия')
>>> plt.ylabel('Коэффициент объясненной дисперсии')
>>> plt.xlabel('Индекс главного компонента')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

Результирующий график (рис. 5.2) показывает, что на долю лишь одного первого главного компонента приходится приблизительно 40% дисперсии. Кроме того, можно заметить, что первые два главных компонента вместе объясняют почти 60% дисперсии в наборе данных.

Хотя график объясненной дисперсии в чем-то похож на показатели важности признаков, вычисляемые в главе 4 через случайные леса, мы должны напомнить себе, что PCA — метод без учителя, т.е. информация о метках классов игнорируется. В то время как случайный лес применяет информацию членства в классах для подсчета показателей загрязненности узлов, дисперсия измеряет разброс значений вдоль оси признака.

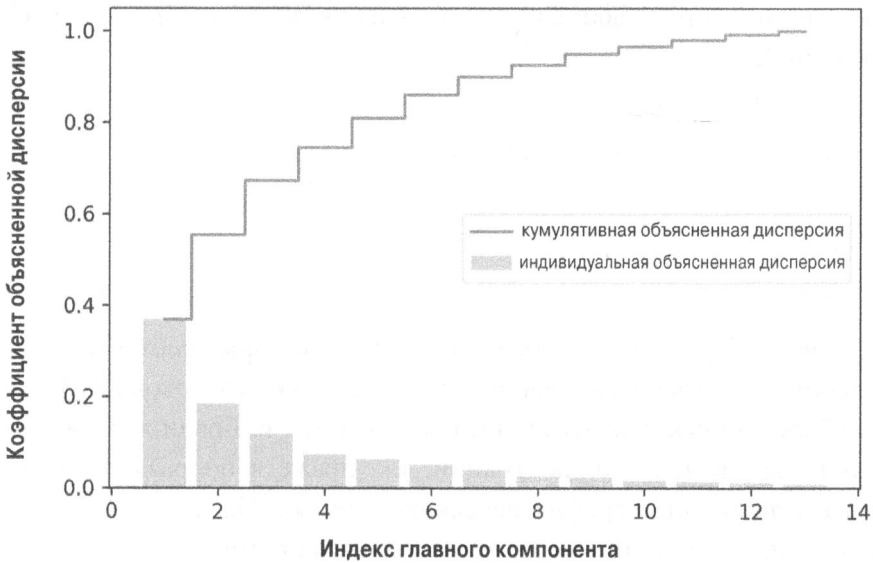


Рис. 5.2. График объясненной дисперсии

## Трансформация признаков

После успешного разложения ковариационной матрицы на собственные пары мы продолжим реализацией последних трех шагов, чтобы трансформировать набор данных Wine в новые оси главных компонент. Ниже перечислены оставшиеся шаги, которыми мы займемся в настоящем разделе:

- 1) выбор  $k$  собственных векторов, которые соответствуют  $k$  наибольшим собственным значениям, где  $k$  — размерность нового подпространства признаков ( $k \leq d$ );
- 2) построение матрицы проекции  $\mathbf{W}$  из “верхних”  $k$  собственных векторов;
- 3) трансформация  $d$ -мерного входного набора данных  $\mathbf{X}$  с использованием матрицы проекции  $\mathbf{W}$  для получения нового  $k$ -мерного подпространства признаков.

Или, выражаясь менее формально, мы отсортируем собственные пары в порядке убывания собственных значений, построим матрицу проекции из выбранных собственных векторов и применим готовую матрицу проекции для трансформирования данных в подпространство меньшей размерности.



Мы начнем с сортировки собственных пар в порядке убывания собственных значений:

```
>>> #создать список кортежей (собственное значение, собственный вектор)
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
...                 for i in range(len(eigen_vals))]
>>> # отсортировать кортежи (собственное значение,
>>> # собственный вектор) от высоких к низким
>>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Далее мы соберем два собственных вектора, которые соответствуют двум наибольшим собственным значениям, чтобы захватить около 60% дисперсии в наборе данных. Обратите внимание, что мы выбрали два собственных вектора только в целях иллюстрации, т.к. позже в подразделе планируем нарисовать двумерный график рассеяния данных. На практике количество главных компонент должно определяться компромиссом между вычислительной продуктивностью и эффективностью классификатора:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                 eigen_pairs[1][1][:, np.newaxis]))
>>> print('Матрица W:\n', w)
```

Матрица W:

```
[[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]
```

В результате выполнения предыдущего кода мы создаем (13×2)-мерную матрицу проекции **W** из двух верхних собственных векторов.



### Зеркальные проекции

В зависимости от используемых версий NumPy и LAPACK вы можете получить матрицу  $W$  с противоположными знаками. Следует отметить, что это не проблема; если  $v$  — собственный вектор матрицы  $\Sigma$ , то мы имеем:

$$\Sigma v = \lambda v$$

Здесь  $v$  — собственный вектор и  $-v$  — тоже собственный вектор, что мы можем доказать следующим образом. Вспомнив базовую алгебру, мы можем умножить обе стороны уравнения на скаляр  $\alpha$ :

$$\alpha \Sigma v = \alpha \lambda v$$

Поскольку перемножение матриц ассоциативно для умножения на скаляр, мы можем затем сделать такую перестановку:

$$\Sigma(\alpha v) = \lambda(\alpha v)$$

Теперь видно, что  $\alpha v$  — собственный вектор с тем же самым собственным значением  $\lambda$  для  $\alpha = 1$  и  $\alpha = -1$ . Отсюда  $v$  и  $-v$  являются собственными векторами.

С применением матрицы проекции мы можем трансформировать образец  $x$  (представленный как 13-мерный вектор-строка) в подпространство PCA (первый и второй главные компоненты), получив  $x'$  — теперь двумерный вектор образца, который состоит из двух новых признаков:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.38299011,  0.45458499])
```

Подобным образом мы можем трансформировать весь (124×13)-мерный обучающий набор данных в два главных компонента, вычислив скалярное произведение матриц:

$$X' = XW$$

```
>>> X_train_pca = X_train_std.dot(w)
```

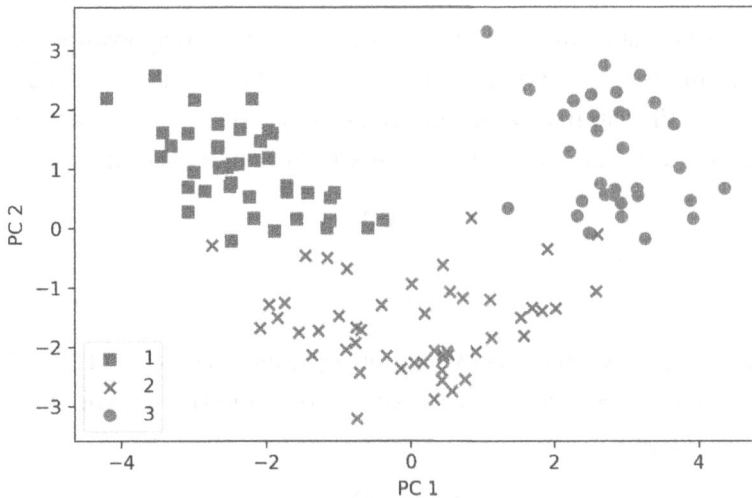
Наконец, давайте визуализируем трансформированный обучающий набор Wine, хранящийся в (124×2)-мерной матрице, в виде двумерного графика рассеяния:

```

>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=l, marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()

```

На результирующем графике (рис. 5.3) видно, что данные больше разбросаны вдоль оси  $x$  (первого главного компонента), нежели вдоль оси  $y$  (второго главного компонента), что согласуется с графиком объясненной дисперсии (см. рис. 5.2). Тем не менее, мы можем интуитивно представлять, что линейный классификатор вероятнее всего будет способен хорошо разделять классы.



**Рис. 5.3.** Двумерный график рассеяния для трансформированного обучающего набора *Wine*

Хотя ради демонстрации на предыдущем графике (см. рис. 5.3) мы кодировали сведения о метках классов, необходимо иметь в виду, что РСА является приемом без учителя, который не использует какую-либо информацию о метках классов.

## Анализ главных компонент в scikit-learn

Несмотря на то что подробный подход в предыдущем подразделе помог в исследовании внутреннего устройства PCA, далее мы обсудим применение класса PCA, реализованного в scikit-learn.

Класс PCA — еще один класс-преобразователь из scikit-learn, где мы сначала подгоняем модель, используя обучающие данные, а затем трансформируем обучающий и испытательный наборы данных с указанием тех же самых параметров модели. Давайте применим класс PCA из scikit-learn к обучающему набору данных Wine, классифицируем трансформированные образцы посредством логистической регрессии и визуализируем области решений с использованием функции `plot_decision_regions`, которую мы определили в главе 2:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):
    # настроить генератор маркеров и карту цветов
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # вывести поверхность решения
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # вывести образцы по классам
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.6,
                    color=cmap(idx),
                    edgecolor='black',
                    marker=markers[idx],
                    label=cl)
```

Для удобства код функции `plot_decision_regions` можно поместить в отдельный файл внутри рабочего каталога, например, `plot_decision_regions_script.py`, и импортировать его в текущем сеансе Python.

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> # инициализация преобразователя PCA и оценщика
>>> # на основе логистической регрессии:
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression(multi_class='ovr',
...                          random_state=1,
...                          solver='lbfgs')
>>> # понижение размерности:
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> # подгонка модели, основанной на логистической регрессии,
>>> # к сокращенному набору данных:
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

В результате выполнения предыдущего кода должны отобразиться области решений для обучающих данных, приведенные к двум осям главных компонентов (рис. 5.4).

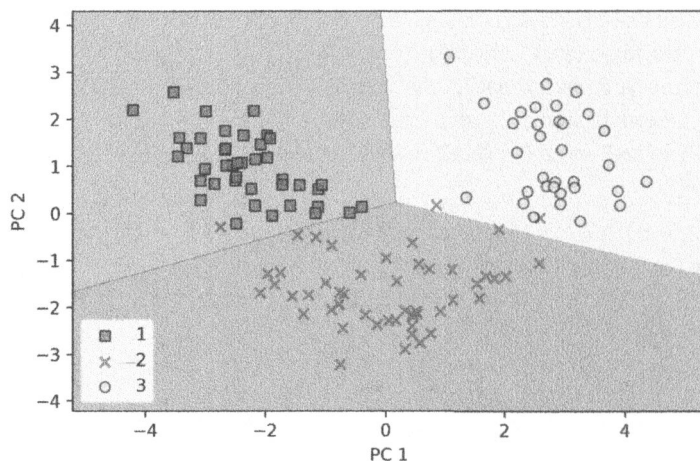


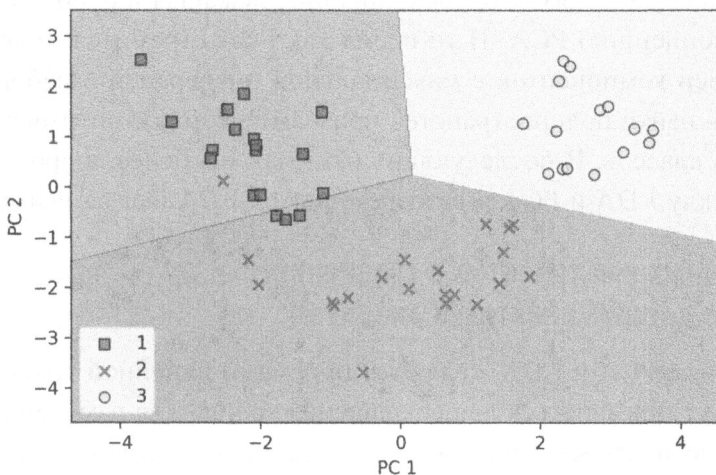
Рис. 5.4. Применение класса PCA к обучающему набору данных Wine

При сравнении проекций класса PCA из `scikit-learn` с нашей реализацией алгоритма PCA может оказаться, что результирующие графики являются зеркальными изображениями друг друга. Следует отметить, что причина такой разницы не связана с какой-то ошибкой в одной из двух реализаций. Дело в том, что в зависимости от собственного решателя собственные векторы могут иметь либо отрицательные, либо положительные знаки.

Хотя это не имеет значения, мы могли бы просто обратить зеркальное изображение, умножая данные на  $-1$ ; имейте в виду, что собственные векторы обычно масштабируются до единичной длины (1). Для полноты давайте построим график с областями решений логистической регрессии на трансформированном испытательном наборе данных, чтобы посмотреть, удастся ли ей хорошо разделить классы:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике с областями решений можно заметить, что логистическая регрессия на этом небольшом двумерном подпространстве признаков выполняется довольно хорошо, неправильно классифицируя совсем немного образцов (рис. 5.5).



**Рис. 5.5.** Области решений логистической регрессии на трансформированном испытательном наборе данных

Если нас интересуют коэффициенты объясненной дисперсии разных главных компонент, тогда мы можем просто инициализировать класс PCA с параметром `n_components`, установленным в `None`, так что сохраняются все главные компоненты, а коэффициент объясненной дисперсии будет доступен через атрибут `explained_variance_ratio_`:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.36951469, 0.18434927, 0.11815159, 0.07334252,
        0.06422108, 0.05051724, 0.03954654, 0.02643918,
        0.02389319, 0.01629614, 0.01380021, 0.01172226,
        0.00820609])
```

При инициализации класса PCA мы установили `n_components=None`, поэтому вместо того, чтобы выполнить понижение размерности, он возвращает все главные компоненты в отсортированном виде.

## Сжатие данных с учителем посредством линейного дискриминантного анализа

*Линейный дискриминантный анализ (LDA)* можно использовать как прием выделения признаков для повышения вычислительной продуктивности и уменьшения степени переобучения из-за “проклятия размерности” в нерегуляризованных моделях. Общая концепция, лежащая в основе LDA, очень похожа на концепцию PCA. В то время как PCA стремится отыскать ортогональные оси компонент с максимальной дисперсией в наборе данных, цель LDA — найти подпространство признаков, которое оптимизирует сепарабельность классов. В последующих разделах мы более подробно обсудим сходства между LDA и PCA и разберем подход LDA шаг за шагом.

### Анализ главных компонент в сравнении с линейным дискриминантным анализом

Оба анализа, PCA и LDA, являются приемами линейной трансформации, которые могут применяться для сокращения количества измерений в наборе данных; первый представляет собой алгоритм без учителя, тогда как второй — алгоритм с учителем. Таким образом, мы можем интуитивно считать LDA более совершенным приемом выделения признаков для задач класси-

фикации по сравнению с PCA. Однако в статье “PCA Versus LDA” (PCA в сравнении с LDA), А.М. Мартинес и А.К. Как, IEEE Transactions on Pattern Analysis and Machine Intelligence, 23(2): стр. 228–233 (2001 год) было показано, что предварительная обработка посредством PCA в определенных случаях имеет тенденцию давать лучшие результаты классификации в задаче распознавания изображений, например, если в состав каждого класса входит лишь небольшое число образцов.



На  
замечку!

### Анализ LDA Фишера

Временами анализ LDA также называют анализом LDA Фишера. Первоначально в 1936 году Роналд Э. Фишер сформулировал *линейный дискриминант Фишера* для задач двухклассовой классификации (“The Use of Multiple Measurements in Taxonomic Problems” (Использование множества измерений в таксономических задачах), Р.Э. Фишер, Annals of Eugenics, 7(2): с. 179–188 (1936 г.)). Позже в 1948 году К. Радхакришна Рао обобщил линейный дискриминант Фишера для решения многоклассовых задач при допущениях о равных ковариациях классов и о нормально распределенных классах, который теперь называется LDA (“The Utilization of Multiple Measurements in Problems of Biological Classification” (Применение множества измерений в задачах биологической классификации), К.Р. Рао, Journal of the Royal Statistical Society. Series B (Methodological), 10(2): с. 159–203 (1948 г.)).

На рис. 5.6 демонстрируется концепция LDA для двухклассовой задачи. Образцы из класса 1 изображены в виде окружностей, а образцы из класса 2 — в виде крестиков.

Линейный дискриминант, показанный на оси  $x$  (LD 1), хорошо разделял бы два нормально распределенных класса. Хотя иллюстративный линейный дискриминант, представленный на оси  $y$  (LD 2), захватывает много дисперсии в наборе данных, он не смог бы послужить хорошим линейным дискриминантом, т.к. не захватывает вообще никакой информации, различающей классы.

Одно из допущений в LDA заключается в том, что данные подчиняются нормальному распределению. Вдобавок мы предполагаем, что классы имеют идентичные ковариационные матрицы и признаки статистически независимы друг от друга.



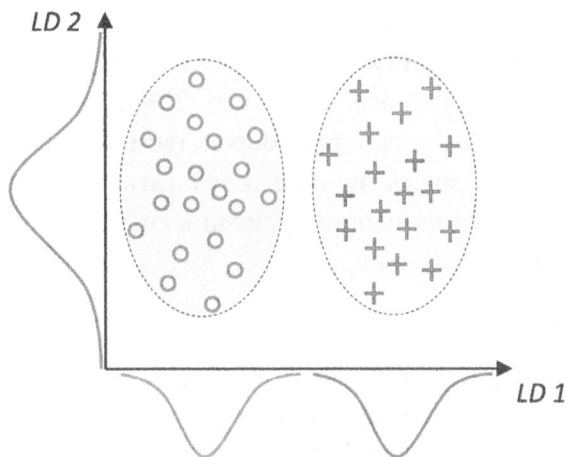


Рис. 5.6. Концепция LDA для двухклассовой задачи

Тем не менее, даже если одно или более таких допущений (слегка) нарушается, LDA для понижения размерности по-прежнему может работать достаточно хорошо (“Pattern Classification” (Классификация образов), 2-е издание, Р. Дуда, П. Харт, Д. Сторк (2001 г.)).

## Внутреннее устройство линейного дискриминантного анализа

Прежде чем приступить к написанию кода реализации, давайте подытожим основные шаги, требующиеся для выполнения LDA.

1. Стандартизировать  $d$ -мерный набор данных ( $d$  — количество признаков).
2. Для каждого класса вычислить  $d$ -мерный вектор средних.
3. Построить матрицу рассеяния между классами  $S_B$  и матрицу рассеяния внутри классов  $S_W$ .
4. Вычислить собственные векторы и соответствующие собственные значения матрицы  $S_W^{-1} S_B$ .
5. Отсортировать собственные значения в порядке убывания, чтобы ранжировать соответствующие собственные векторы.
6. Выбрать  $k$  собственных векторов, которые соответствуют  $k$  наибольшим собственным значениям, чтобы построить  $(d \times k)$ -мерную матрицу трансформации  $W$ ; собственные векторы будут столбцами этой матрицы.
7. Спроецировать образцы в новое подпространство признаков, используя матрицу трансформации  $W$ .

Можно заметить, что алгоритм LDA довольно похож на PCA в том смысле, что мы производим разложение матриц на собственные значения и собственные векторы, которые сформируют новое пространство признаков меньшей размерности. Однако, как упоминалось ранее, алгоритм LDA принимает во внимание информацию о метках классов, которая представлена в виде векторов средних, вычисляемых на шаге 2. В последующих разделах мы обсудим перечисленные семь шагов более подробно, сопровождая их иллюстративными реализациями.

## Вычисление матриц рассеяния

Поскольку в начале главы мы уже стандартизировали признаки набора данных Wine, то можем пропустить первый шаг и заняться вычислением векторов средних, которые будут применяться для построения матрицы рассеяния внутри классов и матрицы рассеяния между классами. Каждый вектор средних  $\mathbf{m}_i$  хранит среднее значение признака  $\mu_m$  по образцам класса  $i$ :

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{\mathbf{x} \in D_i} \mathbf{x}_m$$

Результатом будут три вектора средних:

$$\mathbf{m}_i = \begin{bmatrix} \mu_{i, \text{алкоголь}} \\ \mu_{i, \text{яблочная кислота}} \\ \vdots \\ \mu_{i, \text{пролин}} \end{bmatrix} \quad i \in \{1, 2, 3\}$$

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...         X_train_std[y_train==label], axis=0))
...     print('MV %s: %s\n' % (label, mean_vecs[label-1]))
MV 1: [ 0.9066 -0.3497  0.3201 -0.7189  0.5056  0.8807  0.9589 -0.5516
 0.5416  0.2338  0.5897  0.6563  1.2075]
MV 2: [-0.8749 -0.2848 -0.3735  0.3157 -0.3848 -0.0433  0.0635 -0.0946
 0.0703 -0.8286  0.3144  0.3608 -0.7253]
MV 3: [ 0.1992  0.866  0.1682  0.4148 -0.0451 -1.0286 -1.2876  0.8287
-0.7795  0.9649 -1.209 -1.3622 -0.4013]
```

Используя векторы средних, мы можем вычислить матрицу рассеяния внутри классов  $S_W$ :

$$S_W = \sum_{i=1}^c S_i$$

Это вычисляется путем суммирования индивидуальных матриц рассеяния  $S_i$  каждого отдельного класса  $i$ :

$$S_i = \sum_{x \in D_i} (x - m_i)(x - m_i)^T$$

```
>>> d = 13      # количество признаков
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.zeros((d, d))
>>>     for row in X_train_std[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row - mv).dot((row - mv).T)
...         S_W += class_scatter
>>> print('Матрица рассеяния внутри классов: %sx%s' % (
...         S_W.shape[0], S_W.shape[1]))
Матрица рассеяния внутри классов: 13x13
```

При вычислении матриц рассеяния мы делаем допущение о том, что метки классов в обучающем наборе имеют равномерное распределение. Тем не менее, если мы выведем числа меток классов, то увидим, что указанное допущение нарушается:

```
>>> print('Распределение меток классов: %s'
...       % np.bincount(y_train)[1:])
Распределение меток классов: [41 50 33]
```

Таким образом, мы хотим масштабировать индивидуальные матрицы рассеяния  $S_i$  перед их суммированием для получения матрицы рассеяния  $S_W$ . Разделив матрицы рассеяния на количество образцов класса  $n_i$ , мы сможем заметить, что вычисление матрицы рассеяния фактически будет таким же, как вычисление ковариационной матрицы  $\Sigma_i$  — ковариационная матрица является нормализованной версией матрицы рассеяния:

$$\Sigma_i = \frac{1}{n_i} S_W = \frac{1}{n_i} \sum_{x \in D_i} (x - m_i)(x - m_i)^T$$

Вот код для расчета масштабированной матрицы рассеяния внутри классов:

```
>>> d = 13      # количество признаков
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Масштабированная матрица рассеяния внутри классов: %s%s'
...       % (S_W.shape[0], S_W.shape[1]))
Масштабированная матрица рассеяния внутри классов: 13x13
```

После вычисления масштабированной матрицы рассеяния внутри классов (или ковариационной матрицы) мы можем переходить к следующему шагу и рассчитать матрицу рассеяния между классами  $S_B$ :

$$S_B = \sum_{i=1}^c n_i (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

Здесь  $\mathbf{m}$  — общее среднее, которое вычисляется, включая образцы из всех классов:

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> d = 13      # количество признаков
>>> S_B = np.zeros((d, d))
>>> for i, mean_vec in enumerate(mean_vecs):
...     n = X_train_std[y_train == i + 1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1) # создать вектор-столбец
...     mean_overall = mean_overall.reshape(d, 1)
...     S_B += n * (mean_vec - mean_overall).dot(
...         (mean_vec - mean_overall).T)
>>> print('Матрица рассеяния между классами: %s%s' % (
...     S_B.shape[0], S_B.shape[1]))
Матрица рассеяния между классами: 13x13
```

## Выбор линейных дискриминантов для нового подпространства признаков

Оставшиеся шаги LDA похожи на шаги PCA. Однако вместо разложения ковариационной матрицы на собственные значения мы решаем обобщенную задачу получения собственных значений матрицы  $S_W^{-1} S_B$ :

```
>>> eigen_vals, eigen_vecs = \
...     np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

После вычисления собственных пар мы можем отсортировать собственные значения в порядке убывания:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                 for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                       key=lambda k: k[0], reverse=True)
>>> print('Собственные значения в порядке убывания:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
```

Собственные значения в порядке убывания:

```
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

В алгоритме LDA число линейных дискриминантов не превышает  $c-1$ , где  $c$  — количество меток классов, т.к. матрица рассеяния внутри классов  $S_B$  представляет собой сумму  $c$  матриц с рангом 1 или меньше. На самом деле мы можем заметить, что имеем только два ненулевых собственных значения (собственные значения 3–13 не точно равны нулю, но это обусловлено особенностями арифметики с плавающей точкой в NumPy).



### Коллинеарность

На заметку!

Обратите внимание, что в редком случае идеальной коллинеарности (все выровненные точки образцов попадают на прямую линию) ковариационная матрица имела бы ранг 1, и в итоге получился бы только один собственный вектор с ненулевым собственным значением.

Чтобы измерить, сколько информации, различающей классы, захватывается линейными дискриминантами (собственными векторами), мы постро-

им график линейных дискриминантов по убыванию собственных значений, похожий на график объясненной дисперсии, который создавался в разделе, посвященном PCA. Для упрощения мы будем называть содержимое информации, различающей классы, *различимостью* (*discriminability*):

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, alpha=0.5, align='center',
...         label='индивидуальная "различимость"')
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='кумулятивная "различимость"')
>>> plt.ylabel('Коэффициент "различимости"')
>>> plt.xlabel('Линейные дискриминанты')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 5.7) видно, что первые два линейных дискриминанта без посторонней помощи захватывают 100% полезной информации в обучающем наборе Wine.

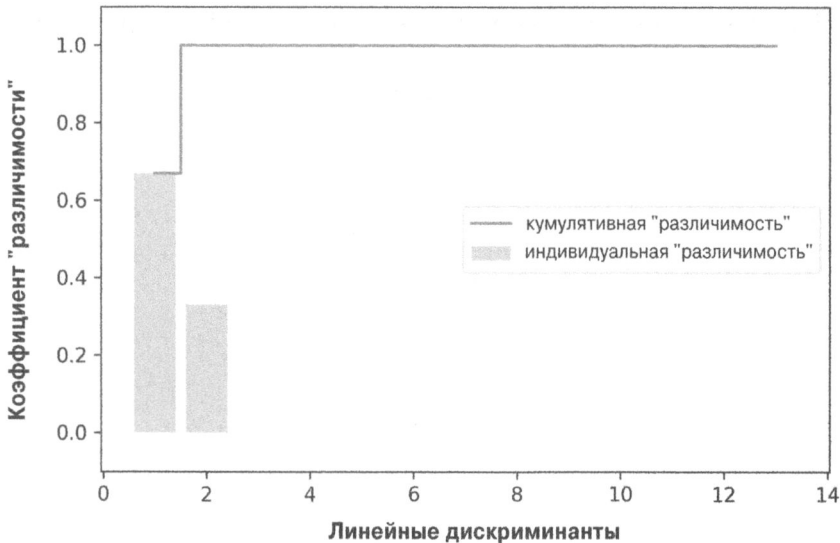


Рис. 5.7. График коэффициента "различимости"

Теперь давайте уложим в стопку два столбца с самыми различающимися собственными векторами, чтобы создать матрицу трансформации **W**:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                  eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Матрица W:\n', w)
Матрица W:
[ [-0.1481 -0.4092]
  [ 0.0908 -0.1577]
  [-0.0168 -0.3537]
  [ 0.1484  0.3223]
  [-0.0163 -0.0817]
  [ 0.1913  0.0842]
  [-0.7338  0.2823]
  [-0.075   -0.0102]
  [ 0.0018  0.0907]
  [ 0.294   -0.2152]
  [-0.0328  0.2747]
  [-0.3547 -0.0124]
  [-0.3915 -0.5958]]
```

## Проецирование образцов в новое подпространство признаков

С применением матрицы трансформации **W**, созданной в предыдущем подразделе, мы можем трансформировать обучающий набор, умножая матрицы:

$$\mathbf{X}' = \mathbf{X} \mathbf{W}$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['s', 'x', 'o']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1] * (-1),
...                 c=c, label=l, marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 5.8) можно заметить, что в новом подпространстве признаков три класса вин в полной мере линейно сепарабельны.

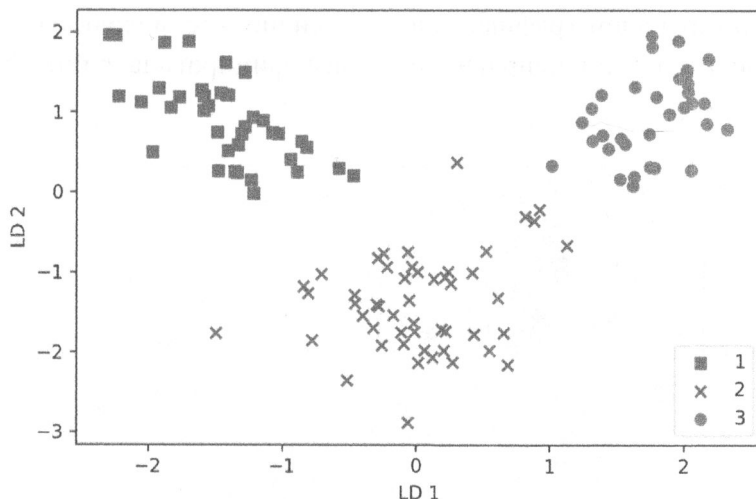


Рис. 5.8. Результат проецирования образцов в новое подпространство признаков

## Реализация LDA в scikit-learn

Пошаговая реализация была хорошим упражнением для понимания внутреннего устройства LDA, а также отличий между алгоритмами LDA и PCA. Пришло время взглянуть на класс LDA, реализованный в библиотеке scikit-learn:

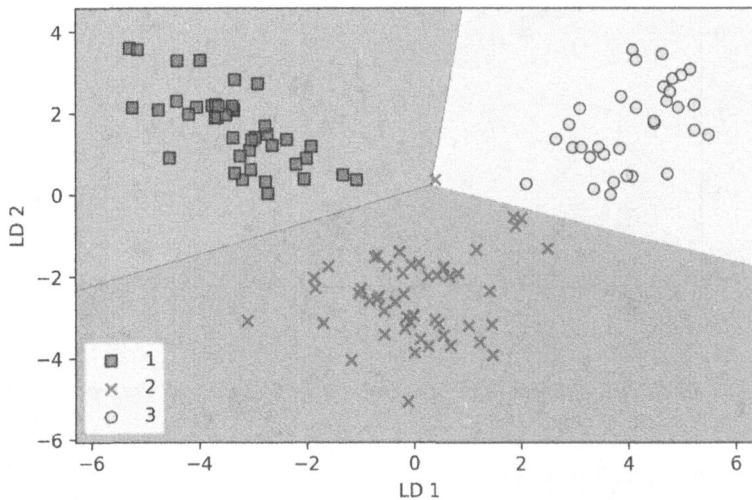
```
>>> # следующий оператор импортирования располагается в одной строке
>>> from sklearn.discriminant_analysis
...     import LinearDiscriminantAnalysis as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Далее мы посмотрим, как классификатор на основе логистической регрессии обрабатывает обучающий набор меньшей размерности после трансформации LDA:

```
>>> lr = LogisticRegression(multi_class='ovr', random_state=1,
...                          solver='lbfgs')
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```



На результирующем графике (рис. 5.9) видно, что модель на основе логистической регрессии неправильно классифицировала один образец из класса 2.



**Рис. 5.9.** Результат работы классификатора на обучающем наборе

Снижая силу регуляризации, вероятно, мы смогли бы переместить границы решений, так что модель на основе логистической регрессии корректно классифицировала бы все образцы в обучающем наборе данных. Тем не менее (и это важнее), давайте выясним, какие результаты получаются на испытательном наборе:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 5.10) можно заметить, что классификатор на основе логистической регрессии способен достичь идеальной меры правильности при классификации образцов в испытательном наборе данных, используя лишь двумерное подпространство признаков, а не исходные 13 признаков набора данных Wine.

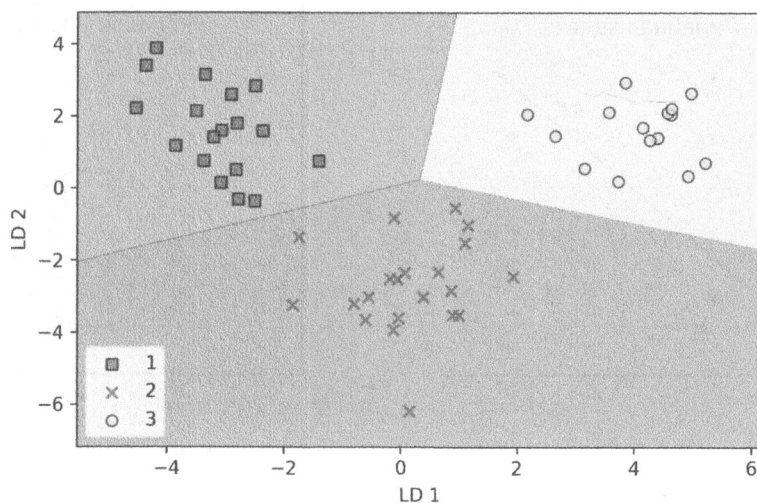


Рис. 5.10. Результат работы классификатора на испытательном наборе

## Использование ядерного анализа главных компонент для нелинейных отображающих функций

Многие алгоритмы МО выдвигают допущения о линейной сепарабельности входных данных. Вы узнали, что персептрон для сходимости даже требует идеально линейно сепарабельных обучающих данных. Другие раскрытые до сих пор алгоритмы предполагают, что идеальная линейная сепарабельность отсутствует из-за шума: Adaline, логистическая регрессия и (стандартный) метод SVM — перечень далеко не полон.

Однако если мы имеем дело с нелинейными задачами, которые весьма часто встречаются в реальных приложениях, то приемы линейной трансформации для понижения размерности, такие как PCA и LDA, могут оказаться не лучшим выбором.

В текущем разделе мы рассмотрим *ядерную (kernelized) версию PCA*, или KPCA, которая относится к концепции ядерного SVM из главы 3. С применением ядерного PCA мы научимся трансформировать данные, не являющиеся линейно сепарабельными, в новое подпространство меньшей размерности, которое подойдет для линейных классификаторов (рис. 5.11).

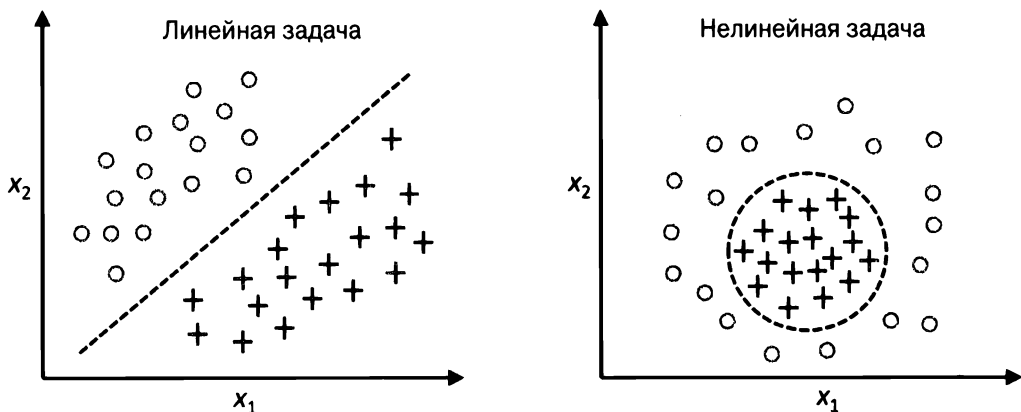


Рис. 5.11. Пример нелинейной задачи классификации

## Ядерные функции и ядерный трюк

Из обсуждения ядерных SVM в главе 3 несложно вспомнить, что мы можем решать нелинейные задачи путем проецирования на новое пространство признаков более высокой размерности, где классы становятся линейно сепарабельными. Чтобы трансформировать образцы  $x \in \mathbb{R}^d$  в такое более высокое  $k$ -мерное подпространство, мы определяли нелинейную отображающую функцию  $\phi$ :

$$\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k \quad (k \gg d)$$

Мы можем считать  $\phi$  функцией, которая создает нелинейные сочетания исходных признаков для отображения первоначального  $d$ -мерного набора данных на более крупное  $k$ -мерное пространство признаков. Например, при наличии вектора признаков  $x \in \mathbb{R}^d$  ( $x$  — вектор-столбец, состоящий из  $d$  признаков) с двумя измерениями ( $d = 2$ ) потенциальным отображением на трехмерное пространство могло бы быть:

$$x = [x_1, x_2]^T$$

$$\downarrow \phi$$

$$z = [x_1^2, \sqrt{2x_1x_2}, x_2^2]^T$$

Другими словами, мы выполняем нелинейное отображение с помощью ядерного PCA, трансформирующего данные в пространство более высокой размерности. Затем в новом пространстве более высокой размерности мы используем стандартный PCA, чтобы спроецировать данные обратно на пространство меньшей размерности, где образцы могут быть разделены линейным классификатором (при условии, что образцы во входном пространстве поддаются разделению по плотности). Тем не менее, недостатком такого подхода являются большие вычислительные затраты и именно здесь мы применяем *ядерный трюк* (*kernel trick*). Используя ядерный трюк, мы можем рассчитать близость между двумя векторами признаков высокой размерности в исходном пространстве признаков.

Прежде чем погружаться в детали ядерного трюка для решения этой затратной в вычислительном плане задачи, давайте вспомним подход со стандартным PCA, который был реализован в начале главы. Мы вычисляли ковариацию между двумя признаками  $k$  и  $j$  следующим образом:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j) (x_k^{(i)} - \mu_k)$$

Поскольку стандартизация признаков центрирует их в нулевом среднем, например,  $\mu_j$  и  $\mu_k$ , мы можем упростить уравнение, как показано ниже:

$$\sigma_{jk} = \frac{1}{n} \sum_{i=1}^n x_j^{(i)} x_k^{(i)}$$

Обратите внимание, что предыдущее уравнение ссылается на ковариацию между двумя признаками. А теперь запишем общее уравнения для вычисления ковариационной матрицы  $\Sigma$ :

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \mathbf{x}^{(i)} \mathbf{x}^{(i)T}$$

Учитывая то, что подход был обобщен Бернхардом Шолькопфом (“Kernel principal component analysis” (“Ядерный анализ главных компонентов”), Б. Шолькопф, А. Смола и К.Р. Мюллер, с. 583–588 (1997 г.)), мы можем заменить скалярные произведения образцов в исходном пространстве признаков нелинейными комбинациями признаков посредством  $\phi$ :

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T$$

Для получения собственных векторов — главных компонент — из ковариационной матрицы мы должны решить следующее уравнение:

$$\Sigma \mathbf{v} = \lambda \mathbf{v}$$

$$\Rightarrow \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \lambda \mathbf{v}$$

$$\Rightarrow \mathbf{v} = \frac{1}{n\lambda} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T \mathbf{v} = \frac{1}{n} \sum_{i=1}^n \mathbf{a}^{(i)} \phi(\mathbf{x}^{(i)})$$

Здесь  $\lambda$  и  $\mathbf{v}$  — собственные значения и собственные векторы ковариационной матрицы  $\Sigma$ ;  $\mathbf{a}$  можно получить путем извлечения собственных векторов из матрицы ядра (матрицы подобия)  $\mathbf{K}$ , как вы увидите далее.



На заметку!

Ниже показано, как можно вывести матрицу ядра. Сначала давайте запишем ковариационную матрицу в матричном представлении, где  $\phi(\mathbf{X})$  —  $n \times k$ -мерная матрица:

$$\Sigma = \frac{1}{n} \sum_{i=1}^n \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^T = \frac{1}{n} \phi(\mathbf{X})^T \phi(\mathbf{X})$$

Теперь мы можем записать уравнение собственного вектора следующим образом:

$$\mathbf{v} = \frac{1}{n} \sum_{i=1}^n \mathbf{a}^{(i)} \phi(\mathbf{x}^{(i)}) = \lambda \phi(\mathbf{X})^T \mathbf{a}$$

Так как  $\Sigma \mathbf{v} = \lambda \mathbf{v}$ , мы получаем:

$$\frac{1}{n} \phi(\mathbf{X})^T \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} = \lambda \phi(\mathbf{X})^T \mathbf{a}$$

Умножение уравнения с обеих сторон на  $\phi(\mathbf{X})$  дает следующий результат:

$$\frac{1}{n} \phi(\mathbf{X}) \phi(\mathbf{X})^T \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} = \lambda \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a}$$

$$\Rightarrow \frac{1}{n} \phi(\mathbf{X}) \phi(\mathbf{X})^T \mathbf{a} = \lambda \mathbf{a}$$

$$\Rightarrow \frac{1}{n} \mathbf{K} \mathbf{a} = \lambda \mathbf{a}$$

Здесь  $\mathbf{K}$  — матрица подобия (матрица ядра):

$$\mathbf{K} = \phi(\mathbf{X}) \phi(\mathbf{X})^T$$

Как объяснялось в разделе “Решение нелинейных задач с применением ядерного метода опорных векторов” главы 3, мы используем ядерный трюк, чтобы избежать явного вычисления попарных скалярных произведений образцов  $\mathbf{x}$  под  $\phi$ , за счет применения ядерной функции  $k$ , так что явно вычислять собственные векторы не понадобится:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$$

Другими словами, после использования ядерного РСА мы получаем образцы, уже спроецированные на соответствующие компоненты, а не строим матрицу трансформации, как при стандартном РСА. В своей основе ядерную функцию (или просто ядро) можно понимать как функцию, которая вычисляет скалярное произведение двух векторов — меру подобия.

Ниже перечислены наиболее широко применяемые ядра.

- Полиномиальное ядро:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = (\mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)^p$$

Здесь  $\theta$  — порог, а  $p$  — степень, которую должен указать пользователь.

- Ядро на основе гиперболического тангенса (сигмоидальное):

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \tanh(\eta \mathbf{x}^{(i)T} \mathbf{x}^{(j)} + \theta)$$

- Ядро на основе *радиальной базисной функции (RBF)*, или гауссова ядро, которое мы будем использовать в примерах в следующем подразделе:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$$

Его часто записывают в такой форме, вводя переменную  $\gamma = \frac{1}{2\sigma}$ :

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Чтобы подвести итог изученным до сих пор материалам, мы можем определить три шага для реализации алгоритма PCA с ядром RBF.

1. Мы рассчитываем матрицу ядра (матрицу подобия)  $\mathbf{K}$ , где необходимо вычислить следующее уравнение:

$$\mathcal{K}(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2\right)$$

Мы поступаем так для каждой пары образцов:

$$\mathbf{K} = \begin{bmatrix} \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \dots & \mathcal{K}(\mathbf{x}^{(1)}, \mathbf{x}^{(n)}) \\ \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \dots & \mathcal{K}(\mathbf{x}^{(2)}, \mathbf{x}^{(n)}) \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{K}(\mathbf{x}^{(n)}, \mathbf{x}^{(1)}) & \mathcal{K}(\mathbf{x}^{(n)}, \mathbf{x}^{(2)}) & \dots & \mathcal{K}(\mathbf{x}^{(n)}, \mathbf{x}^{(n)}) \end{bmatrix}$$

Например, если наш набор данных содержит 100 обучающих образцов, тогда симметричная матрица ядра с попарными подобиями оказалась бы 100×100-мерной.

2. Мы центрируем матрицу ядра  $\mathbf{K}$  с применением следующего уравнения:

$$\mathbf{K}' = \mathbf{K} - \mathbf{1}_n \mathbf{K} - \mathbf{K} \mathbf{1}_n + \mathbf{1}_n \mathbf{K} \mathbf{1}_n$$

Здесь  $\mathbf{1}_n$  —  $n \times n$ -мерная матрица (с таким же числом измерений, как у матрицы ядра), все значения в которой равны  $\frac{1}{n}$ .

3. Мы собираем верхние  $k$  собственных векторов центрированной матрицы ядра на основе соответствующих им собственных значений, которые ранжированы по уменьшению величины. В отличие от стандартного PCA собственные векторы являются не осями главных компонентов, а образцами, спроецированными на эти оси.

Вас может интересовать, почему мы должны центрировать матрицу ядра на втором шаге. Ранее, когда мы формулировали ковариационную матрицу и заменяли скалярные произведения нелинейными комбинациями признаков посредством  $\phi$ , было сделано допущение, что работа ведется со стандартизированными данными, где все признаки имеют нулевое среднее. Таким образом, на втором шаге возникает необходимость в центрировании матрицы ядра, поскольку мы не вычисляем новое пространство признаков явно, поэтому не можем гарантировать, что новое пространство признаков также будет центрировано в нуле.

В следующем разделе мы воплотим в жизнь описанные три шага, реализовав ядерный PCA на Python.

## Реализация ядерного анализа главных компонент на языке Python

В предыдущем подразделе мы обсуждали основные концепции ядерного PCA. Сейчас мы собираемся реализовать алгоритм PCA с ядром RBF на Python в соответствии с тремя шагами, которые подытоживали подход ядерного PCA. Мы покажем, что с использованием ряда вспомогательных функций SciPy и NumPy реализация ядерного PCA в действительности очень проста:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    Реализация алгоритма PCA с ядром RBF.

    Параметры
    -----
    X: {NumPy ndarray}, форма = [n_examples, n_features]
```



```
gamma: float
    Параметр настройки ядра RBF

n_components: int
    Количество главных компонентов, подлежащих возвращению

Возвращает
-----
X_pc: {NumPy ndarray}, форма = [n_examples, k_features]
    Спроецированный набор данных

"""
# Вычислить попарные квадратичные евклидовы расстояния
# в MxN-мерном наборе данных.
sq_dists = pdist(X, 'sqeuclidean')

# Преобразовать попарные расстояния в квадратную матрицу.
mat_sq_dists = squareform(sq_dists)

# Вычислить симметричную матрицу ядра.
K = exp(-gamma * mat_sq_dists)

# Центрировать матрицу ядра.
N = K.shape[0]
one_n = np.ones((N,N)) / N
K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

# Получить собственные пары из центрированной матрицы ядра;
# scipy.linalg.eigh возвращает их в порядке по возрастанию.
eigvals, eigvecs = eigh(K)
eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]

# Собрать верхние k собственных векторов (спроецированных образцов).
X_pc = np.column_stack([eigvecs[:, i]
                        for i in range(n_components)])

return X_pc
```

Недостаток применения алгоритма PCA с ядром RBF для понижения размерности заключается в том, что мы должны указывать параметр  $\gamma$  заранее. Нахождение подходящего значения для параметра  $\gamma$  требует экспериментирования и лучше всего делается с использованием алгоритмов настройки параметров, скажем, за счет выполнения решетчатого поиска, который мы подробно обсудим в главе 6.

### Пример 1 — разделение фигур в форме полумесяца

Давайте применим нашу функцию `rbf_kernel_pca` к ряду примеров нелинейных наборов данных. Мы начнем с создания двумерного набора данных с сотней образцов точек, представляющих две фигуры в форме полумесяца:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> plt.scatter(X[y==0, 0], X[y==0, 1],
...             color='red', marker='^', alpha=0.5)
>>> plt.scatter(X[y==1, 0], X[y==1, 1],
...             color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

В целях иллюстрации полумесяц из символов треугольника будет представлять образцы одного класса, а полумесяц из символов круга — образцы другого класса (рис. 5.12).

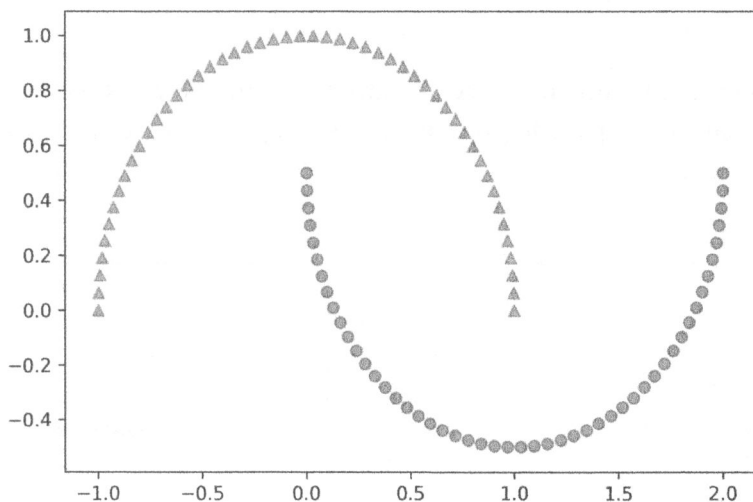


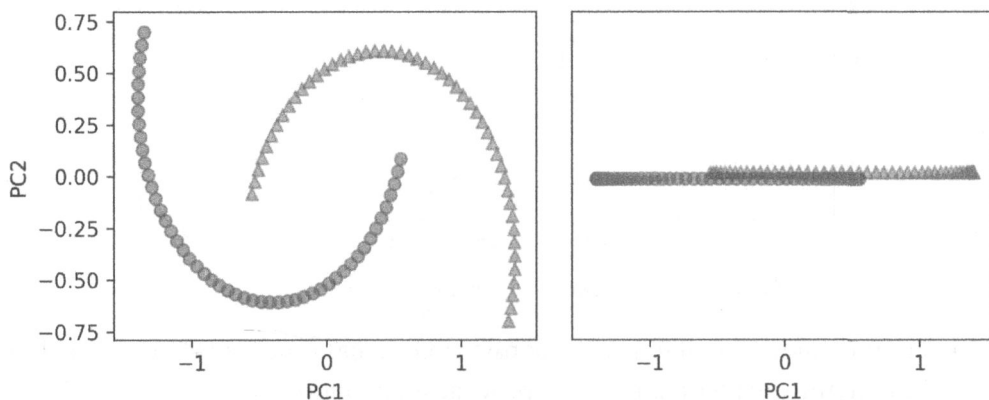
Рис. 5.12. Классы фигур в форме полумесяца

Очевидно, такие две фигуры в форме полумесяца не являются линейно сепарабельными и наша цель — посредством ядерного РСА *развернуть* полумесяцы, чтобы набор данных мог служить в качестве пригодного входа для линейного классификатора. Но сначала мы посмотрим, как выглядит

набор данных, если его спроецировать на главные компоненты через стандартный PCA:

```
>>> from sklearn.decomposition import PCA
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Безусловно, линейный классификатор не способен хорошо работать на наборе данных, трансформированном посредством стандартного PCA (рис. 5.13).



**Рис. 5.13.** Результат работы линейного классификатора на наборе данных, трансформированном посредством стандартного PCA

Обратите внимание, что при построении графика только первого главного компонента (правая часть графика) мы сместили образцы из треугольников слегка вверх, а образцы из кругов — чуть вниз, чтобы лучше видеть наложение классов. В левой части графика видно, что исходные фигуры в форме полумесяца совсем немного сдвинуты и зеркально отображены относительно центра по вертикали — эта трансформация не оказала линейному классификатору содействие в различении образцов из кругов и треугольников. Подобным же образом образцы из кругов и треугольников, соответствующие фигурам в форме полумесяца, не являются линейно сепарабельными, если мы спроецируем набор данных на одномерную ось признака, как показано в правой части графика.



На заметку!

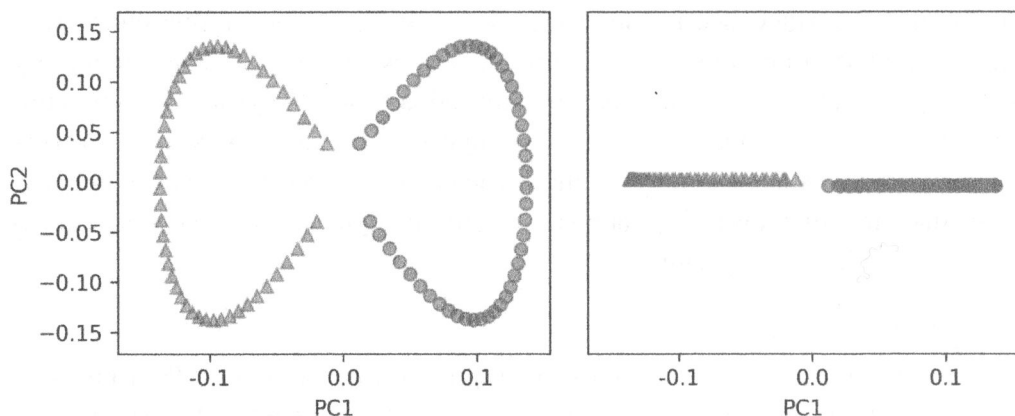
### PCA или LDA

Не забывайте, что в противоположность LDA алгоритм PCA представляет собой метод без учителя и не использует информацию о метках классов с целью доведения до максимума дисперсии. Символы кругов и треугольников здесь были добавлены для целей визуализации, чтобы показать степень разделения.

Давайте испытаем нашу функцию `rbf_kernel_pca` ядерного PCA, которую мы реализовали в предыдущем подразделе:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 5.14 мы видим, что два класса (круги и треугольники) хорошо линейно разделены, поэтому трансформированный набор данных подходит для обработки линейными классификаторами.



**Рис. 5.14.** Результат работы линейного классификатора на наборе данных, трансформированном посредством алгоритма PCA с ядром RBF

К сожалению, для параметра настройки  $\gamma$  нет какого-то универсального значения, которое хорошо бы работало с различными наборами данных. Отыскание значения  $\gamma$ , подходящего для заданной задачи, требует проведения экспериментов. В главе 6 мы обсудим приемы, которые могут помочь нам автоматизировать задачу оптимизации таких параметров настройки. Здесь для параметра  $\gamma$  будут применяться значения, которые согласно нашему опыту обеспечивают хорошие результаты.

### Пример 2 — разделение концентрических окружностей

В предыдущем подразделе мы показали, как разделить фигуры в форме полумесяца посредством ядерного PCA. Поскольку мы решили приложить немало сил, чтобы осмыслить концепции ядерного PCA, давайте взглянем на еще один интересный пример нелинейной задачи — разделение концентрических окружностей:

```
>>> from sklearn.datasets import make_circles
>>> X, y = make_circles(n_samples=1000,
...                     random_state=123, noise=0.1,
...                     factor=0.2)
>>> plt.scatter(X[y == 0, 0], X[y == 0, 1],
...             color='red', marker='^', alpha=0.5)
```

```
>>> plt.scatter(X[y == 1, 0], X[y == 1, 1],
...              color='blue', marker='o', alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

Мы снова предполагаем, что имеем дело с задачей с двумя классами, в которой фигуры из треугольников представляют один класс, а фигуры из кругов — другой класс (рис. 5.15).

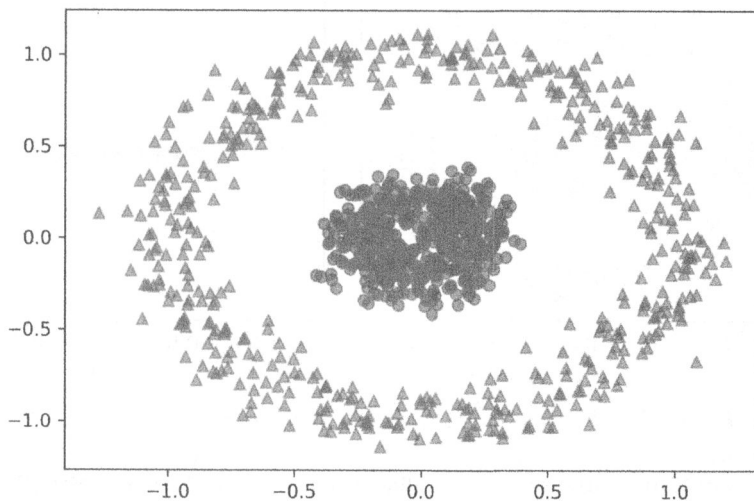


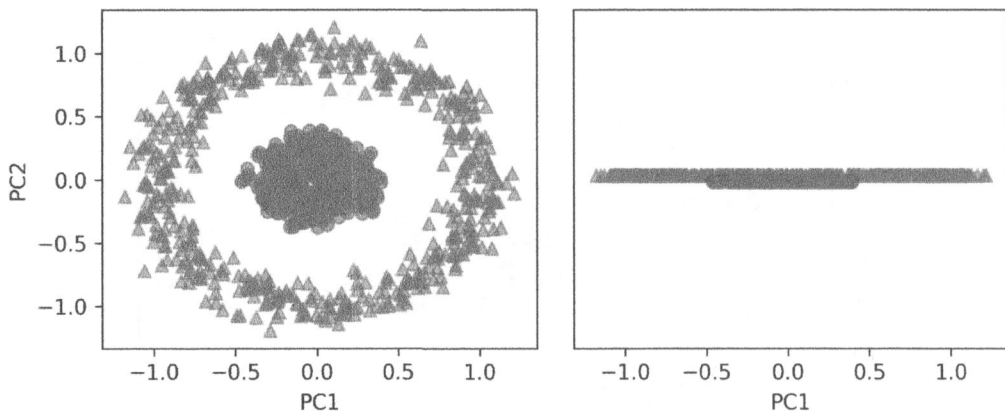
Рис. 5.15. Задача разделения концентрических окружностей

Мы начнем с использования стандартного PCA, чтобы сравнить его результаты с результатами применения PCA с ядром RBF:

```
>>> scikit_pca = PCA(n_components=2)
>>> X_spca = scikit_pca.fit_transform(X)
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
...              color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
...              color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
...              color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
...              color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
```

```
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Мы опять видим, что стандартный PCA не способен выдать результаты, подходящие для обучения линейного классификатора (рис. 5.16).



**Рис. 5.16.** Результаты использования стандартного PCA

Имея подходящее значение для параметра  $\gamma$ , давайте посмотрим, принесет ли нам удачу применение реализации PCA с ядром RBF:

```
>>> X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
>>> fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
>>> ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
...               color='red', marker='^', alpha=0.5)
>>> ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
...               color='blue', marker='o', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
...               color='red', marker='^', alpha=0.5)
>>> ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
...               color='blue', marker='o', alpha=0.5)
>>> ax[0].set_xlabel('PC1')
>>> ax[0].set_ylabel('PC2')
>>> ax[1].set_ylim([-1, 1])
>>> ax[1].set_yticks([])
>>> ax[1].set_xlabel('PC1')
>>> plt.tight_layout()
>>> plt.show()
```

Несложно заметить, что реализация PCA с ядром RBF спроецировала данные на новое подпространство, где два класса становятся линейно сепарабельными (рис. 5.17).

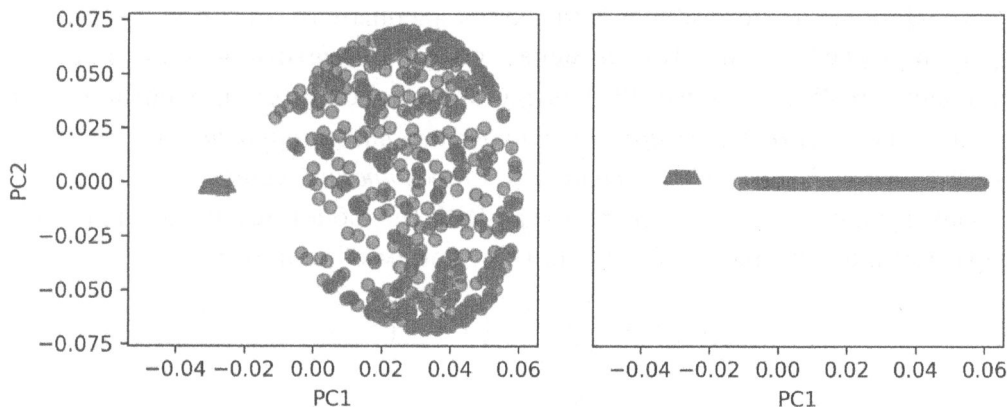


Рис. 5.17. Результаты использования PCA с ядром RBF

## Проецирование новых точек данных

В двух предшествующих примерах применения ядерного PCA (разделение фигур в форме полумесяца и разделение концентрических окружностей) мы проецировали одиночный набор данных на новое подпространство признаков. Однако в реальных приложениях мы можем иметь сразу несколько наборов данных, подлежащих трансформированию, скажем, обучающие и испытательные данные, и обычно также новые образцы, которые будут собираться после построения и оценки моделей. В этом разделе мы покажем, как проецировать точки данных, не являющиеся частью обучающего набора данных.

Как объяснялось при рассмотрении стандартного PCA в начале главы, мы проецируем данные путем вычисления скалярного произведения матрицы трансформации и входных образцов; столбцами матрицы проецирования будут верхние  $k$  собственных векторов ( $v$ ), которые получаются из ковариационной матрицы.

Вопрос теперь в том, как можно перенести данную концепцию на ядерный PCA. Вспоминая идею, лежащую в основе ядерного PCA, мы получали собственный вектор ( $a$ ) центрированной матрицы ядра (не ковариационной матрицы), а это значит, что они являются образцами, которые уже спроецированы на ось главного компонента  $v$ . Таким образом, если мы хотим спро-



ецировать новый образец  $\mathbf{x}'$  на указанную ось главного компонента, тогда должны рассчитать следующую проекцию:

$$\phi(\mathbf{x}')^T \mathbf{v}$$

К счастью, мы можем воспользоваться ядерным трюком и не вычислять проекцию  $\phi(\mathbf{x}')^T \mathbf{v}$  явно. Тем не менее, полезно отметить, что в отличие от стандартного PCA ядерный PCA представляет собой метод, основанный на памяти, т.е. *каждый раз при проецировании новых образцов мы должны повторно задействовать первоначальный обучающий набор*.

Нам нужно вычислить попарно ядро RBF (подобие) между каждым  $i$ -тым образцом в обучающем наборе данных и новым образцом  $\mathbf{x}'$ :

$$\begin{aligned}\phi(\mathbf{x}')^T \mathbf{v} &= \sum_i a^{(i)} \phi(\mathbf{x}')^T \phi(\mathbf{x}^{(i)}) \\ &= \sum_i a^{(i)} \kappa(\mathbf{x}', \mathbf{x}^{(i)})\end{aligned}$$

Собственные векторы  $\mathbf{a}$  и собственные значения  $\lambda$  матрицы ядра  $\mathbf{K}$  в уравнении удовлетворяют следующему условию:

$$\mathbf{K}\mathbf{a} = \lambda\mathbf{a}$$

После вычисления подобия между новыми образцами и образцами в обучающем наборе мы должны нормализовать собственный вектор  $\mathbf{a}$  по его собственному значению. Итак, модифицируем реализованную ранее функцию `rbf_kernel_pca`, чтобы она возвращала также и собственные значения матрицы ядра:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    Реализация алгоритма PCA с ядром RBF.

    Параметры
    -----
    X: {NumPy ndarray}, форма = [n_examples, n_features]

    gamma: float
        Параметр настройки ядра RBF
```

```

n_components: int
    Количество главных компонентов, подлежащих возвращению

Возвращает
-----
alphas: (NumPy ndarray), форма = [n_examples, k_features]
    Спроецированный набор данных

lambdas: список
    Собственные значения
"""
# Рассчитать попарные квадратичные евклидовы расстояния
# в MxN-мерном наборе данных.
sq_dists = pdist(X, 'sqeuclidean')

# Преобразовать попарные расстояния в квадратную матрицу.
mat_sq_dists = squareform(sq_dists)

# Вычислить симметричную матрицу ядра.
K = exp(-gamma * mat_sq_dists)

# Центрировать матрицу ядра.
N = K.shape[0]
one_n = np.ones((N,N)) / N
K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

# Получить собственные пары из центрированной матрицы ядра;
# scipy.linalg.eigh возвращает их в порядке по возрастанию.
eigvals, eigvecs = eigh(K)
eigvals, eigvecs = eigvals[::-1], eigvecs[:, ::-1]

# Собрать верхние k собственных векторов
# (спроецированных образцов).
alphas = np.column_stack([eigvecs[:, i]
                           for i in range(n_components)])

# Собрать соответствующие собственные значения.
lambdas = [eigvals[i] for i in range(n_components)]

return alphas, lambdas

```

А теперь создадим новый набор данных с фигурами в форме полумесяца и спроецируем его в одномерное подпространство, применяя модифицированную реализацию PCA с ядром RBF:

```

>>> X, y = make_moons(n_samples=100, random_state=123)
>>> alphas, lambdas = rbf_kernel_pca(X, gamma=15, n_components=1)

```

Чтобы проверить, реализовали ли мы код для проецирования новых образцов, предположим, что 26-я точка из набора данных с фигурами в форме полумесяца является новой точкой данных  $x'$ , и наша задача — спроецировать ее в новое подпространство:

```
>>> x_new = X[25]
>>> x_new
array([ 1.8713187 ,  0.00928245])
>>> x_proj = alphas[25]    # первоначальная проекция
>>> x_proj
array([ 0.07877284])
>>> def project_x(x_new, X, gamma, alphas, lambdas):
...     pair_dist = np.array([np.sum(
...         (x_new-row)**2) for row in X])
...     k = np.exp(-gamma * pair_dist)
...     return k.dot(alphas / lambdas)
```

После выполнения приведенного ниже кода мы в состоянии воспроизвести первоначальную проекцию. С использованием функции `project_x` мы также сможем проецировать любой новый образец данных. Вот как выглядит код:

```
>>> x_reproj = project_x(x_new, X,
...                       gamma=15, alphas=alphas,
...                       lambdas=lambdas)
>>> x_reproj
array([ 0.07877284])
```

Наконец, давайте визуализируем проекцию на первом главном компоненте:

```
>>> plt.scatter(alphas[y==0, 0], np.zeros((50)),
...              color='red', marker='^', alpha=0.5)
>>> plt.scatter(alphas[y==1, 0], np.zeros((50)),
...              color='blue', marker='o', alpha=0.5)
>>> plt.scatter(x_proj, 0, color='black',
...              label='первоначальная проекция точки X[25]',
...              marker='^', s=100)
>>> plt.scatter(x_reproj, 0, color='green',
...              label='повторно отображенная точка X[25]',
...              marker='x', s=500)
>>> plt.yticks([], [])
>>> plt.legend(scatterpoints=1)
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике рассеяния (рис. 5.18) видно, что образец  $x'$  корректно отображился на первый главный компонент.

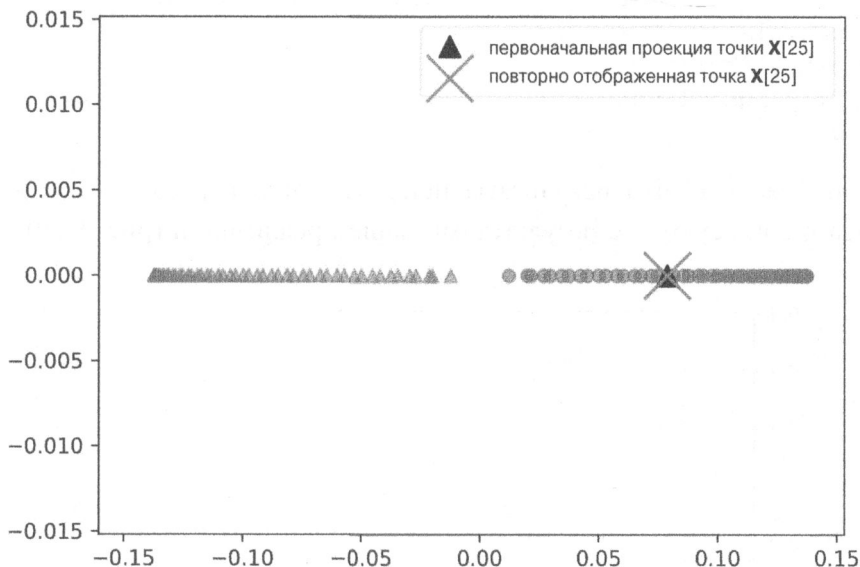


Рис. 5.18. Корректное отображение образца  $x'$  на первый главный компонент

## Ядерный анализ главных компонент в scikit-learn

Ради нашего удобства в подмодуле `sklearn.decomposition` библиотеки `scikit-learn` реализован класс ядерного PCA. Он применяется аналогично классу стандартного PCA, и с помощью параметра `kernel` можно указывать ядро:

```
>>> from sklearn.decomposition import KernelPCA
>>> X, y = make_moons(n_samples=100, random_state=123)
>>> scikit_kpca = KernelPCA(n_components=2,
...                           kernel='rbf', gamma=15)
>>> X_skernpca = scikit_kpca.fit_transform(X)
```

Чтобы проверить, согласуются ли полученные результаты с нашей реализацией ядерного PCA, мы построим график трансформированного набора данных с фигурами в форме полумесяца для первых двух главных компонент:

```

>>> plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
...              color='red', marker='^', alpha=0.5)
>>> plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
...              color='blue', marker='o', alpha=0.5)
>>> plt.xlabel('PC1')
>>> plt.ylabel('PC2')
>>> plt.tight_layout()
>>> plt.show()

```

Легко заметить, что результаты использования класса `KernelPCA` из `scikit-learn` согласуются с результатами нашей реализации (рис. 5.19).

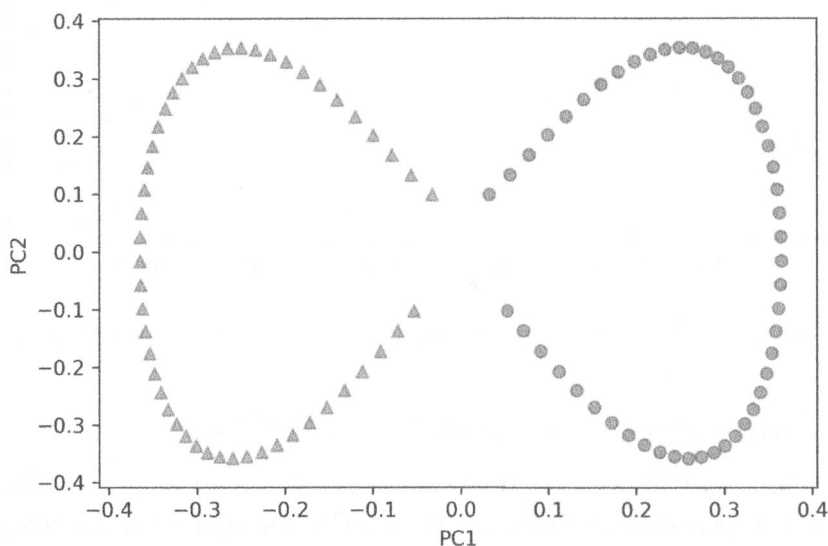


Рис. 5.19. Результаты применения класса `KernelPCA` из `scikit-learn`



На заметку!

### Обучение на основе многообразий (manifold learning)

В библиотеке `scikit-learn` также реализованы развитые приемы для нелинейного понижения размерности, рассмотрение которых выходит за рамки настоящей книги. Заинтересованные читатели могут ознакомиться с хорошим обзором текущих реализаций, сопровождаемым иллюстративными примерами, который доступен по ссылке <http://scikit-learn.org/stable/modules/manifold.html>.

## Резюме

В главе вы узнали три фундаментальных приема понижения размерности для выделения признаков: стандартный PCA, LDA и ядерный PCA. С использованием PCA мы проецируем данные в подпространство меньшей размерности, чтобы довести до максимума дисперсию по ортогональным осям признаков, игнорируя метки классов. В отличие от PCA прием LDA обеспечивает понижение размерности с учителем, что означает учет информации о метках классов в обучающем наборе данных при попытке довести до максимума сепарабельность классов в линейном пространстве признаков.

Наконец, вы ознакомились с приемом выделения нелинейных признаков — ядерным PCA. Применяя ядерный трюк и временную проекцию в пространство признаков более высокой размерности, мы в итоге получили возможность сжать наборы данных, состоящие из нелинейных признаков, в подпространство меньшей размерности, где классы становятся линейно сепарабельными.

Теперь, имея в своем распоряжении указанные важнейшие приемы предварительной обработки, вы полностью готовы к изучению в следующей главе установившейся практики рационального внедрения приемов такого рода и оценки эффективности различных моделей.



# ОСВОЕНИЕ ПРАКТИЧЕСКОГО ОПЫТА ОЦЕНКИ МОДЕЛЕЙ И НАСТРОЙКИ ГИПЕРПАРАМЕТРОВ

**В** предшествующих главах вы узнали о важнейших алгоритмах МО для классификации и о том, как привести данные в надлежащую форму до их передачи этим алгоритмам. Самое время ознакомиться с практическим опытом построения хороших моделей МО за счет точной настройки алгоритмов и оценки эффективности моделей. В главе будут раскрыты следующие темы:

- оценка эффективности моделей МО;
- диагностика распространенных проблем с алгоритмами МО;
- точная настройка моделей МО;
- оценка прогнозирующих моделей с использованием различных метрик эффективности.



## Модернизация рабочих потоков с помощью конвейеров

Применяя в предыдущих главах разнообразные приемы предварительной обработки, такие как стандартизация для масштабирования признаков в главе 4 или анализ главных компонент для сжатия данных в главе 5, мы должны были повторно использовать параметры, полученные во время подгонки к обучающим данным, чтобы масштабировать и сжимать любые новые данные вроде образцов в отдельном испытательном наборе данных. В текущем разделе вы узнаете о крайне полезном инструменте — классе `Pipeline` из `scikit-learn`. Он позволяет подгонять модель, включающую произвольное количество шагов трансформации, и применять ее для выработки прогнозов относительно новых данных.

### Загрузка набора данных Breast Cancer Wisconsin

В главе мы будем работать с набором данных Breast Cancer Wisconsin (диагнозы рака молочной железы в штате Висконсин), содержащим 569 образцов злокачественных и доброкачественных опухолевых клеток. В первых двух столбцах набора данных хранятся уникальные идентификационные номера образцов и соответствующие диагнозы (M = malignant (злокачественная), B = benign (доброкачественная)). Столбцы 3–32 содержат 30 признаков вещественного типа, вычисленных из оцифрованных изображений ядер клеток, которые могут использоваться для построения модели, прогнозирующей доброкачественность или злокачественность опухоли. Набор данных Breast Cancer Wisconsin был передан в Хранилище машинного обучения Калифорнийского университета в Ирвайне (UCI), а с более детальной информацией о нем можно ознакомиться по ссылке [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\(Diagnostic\)](https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)).



На заметку!

#### Получение набора данных Breast Cancer Wisconsin

Копия набора данных Breast Cancer Wisconsin (и всех других наборов данных, используемых в книге) включена в состав загружаемого архива с кодом примеров для книги. Указанные копии можно задействовать при автономной работе или в случае временной недоступности <https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data> на сервере UCI. Скажем, чтобы загрузить набор данных Wine из какого-то локального каталога, следующий оператор:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases'
    '/breast-cancer-wisconsin/wdbc.data',
    header=None)
```

**понадобится заменить таким оператором:**

```
df = pd.read_csv(
    'ваш/локальный/путь/к/wdbc.data',
    header=None)
```

Далее мы прочитаем набор данных Breast Cancer Wisconsin и разобьем его на обучающий и испытательный наборы данных, выполнив три простых шага.

1. Мы начнем с чтения набора данных прямо из веб-сайта UCI с применением `pandas`:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                  'machine-learning-databases'
...                  '/breast-cancer-wisconsin/wdbc.data',
...                  header=None)
```

2. Затем мы присвоим 30 признаков NumPy-массиву `X` и с использованием объекта `LabelEncoder` трансформируем метки классов из первоначального строкового представления ('М' и 'В') в целые числа:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> le.classes_
array(['B', 'M'], dtype=object)
```

После кодирования меток классов (диагнозов) в массиве `y` злокачественные опухоли представлены как класс 1, а доброкачественные — как класс 0. Мы можем еще раз проверить полученное отображение, вызвав метод `transform` подогнанного объекта `LabelEncoder` на двух фиктивных метках классов:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

3. До построения нашего первого конвейера модели в следующем разделе мы разобьем набор данных на обучающий (80% данных) и испытательный (20% данных) наборы:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.20,
...                       stratify=y,
...                       random_state=1)
```

## Объединение преобразователей и оценщиков в конвейер

В предыдущей главе вы узнали, что многие алгоритмы обучения для достижения оптимальной эффективности требуют входных признаков с одним и тем же масштабом. Поскольку признаки в наборе данных Breast Cancer Wisconsin измерены с разными масштабами, нам необходимо стандартизировать столбцы в нем, прежде чем мы сможем передать их линейному классификатору, например, на основе логистической регрессии. Кроме того, пусть мы хотим сжать данные от начальных 30 измерений до двумерного подпространства посредством *анализа главных компонент (PCA)* — приема выделения признаков для понижения размерности, который был представлен в главе 5.

Вместо прохождения через шаги подгонки модели и трансформации данных для обучающего и испытательного наборов по отдельности мы можем объединить объекты `StandardScaler`, `PCA` и `LogisticRegression` в конвейер:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import make_pipeline
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression(random_state=1,
...                                           solver='lbfgs'))
>>> pipe_lr.fit(X_train, y_train)
>>> y_pred = pipe_lr.predict(X_test)
>>> print('Правильность при испытании: %.3f'
...       % pipe_lr.score(X_test, y_test))
Правильность при испытании: 0.956
```

Функция `make_pipeline` принимает произвольное количество преобразователей `scikit-learn` (объектов, поддерживающих методы `fit` и `transform`), за которыми следует оценщик `scikit-learn`, реализующий методы `fit` и `predict`. В предыдущем примере кода мы предоставили функции `make_pipeline` в качестве входа два преобразователя, `StandardScaler` и `PCA`, а также оценщик `LogisticRegression`, в результате чего `make_pipeline` создаст из них объект `Pipeline`.

Мы можем считать объект `Pipeline` из `scikit-learn` метаоценщиком или оболочкой вокруг индивидуальных преобразователей и оценщиков. Если мы вызовем метод `fit` объекта `Pipeline`, тогда данные будут передаваться последовательно преобразователей посредством обращений к `fit` и `transform` на промежуточных шагах, пока они не достигнут объекта оценщика (финального элемента конвейера). Затем оценщик произведет подгонку к трансформированным обучающим данным.

Когда в показанном выше коде мы выполняем метод `fit` на конвейере `pipe_lr`, объект `StandardScaler` сначала осуществляет вызовы методов `fit` и `transform` на обучающих данных. Затем трансформированные обучающие данные передаются следующему объекту в конвейере, т.е. `PCA`. Подобно предыдущему шагу объект `PCA` также выполняет методы `fit` и `transform` на масштабированных входных данных и передает их финальному элементу конвейера — оценщику.

Наконец, оценщик `LogisticRegression` обеспечивает подгонку к обучающим данным после того, как они были подвергнуты трансформациям с помощью `StandardScaler` и `PCA`. Мы снова обязаны отметить, что на количество промежуточных шагов в конвейере никаких ограничений не накладывается; тем не менее, последним элементом конвейера должен быть оценщик.

Аналогично методу `fit` в конвейерах также реализован метод `predict`. Если мы передадим набор данных методу `predict` объекта `Pipeline`, то данные пройдут через все промежуточные шаги посредством обращений к `transform`. На последнем шаге объект оценщика возвратит прогноз на трансформированных данных.

Конвейеры из библиотеки `scikit-learn` — чрезвычайно полезные инструменты для создания оболочек, которые мы будем часто применять в оставшихся главах книги. Чтобы содействовать глубокому пониманию деталей работы объекта `Pipeline`, на рис. 6.1 приведена иллюстрация, подводящая итог предшествующих обсуждений.

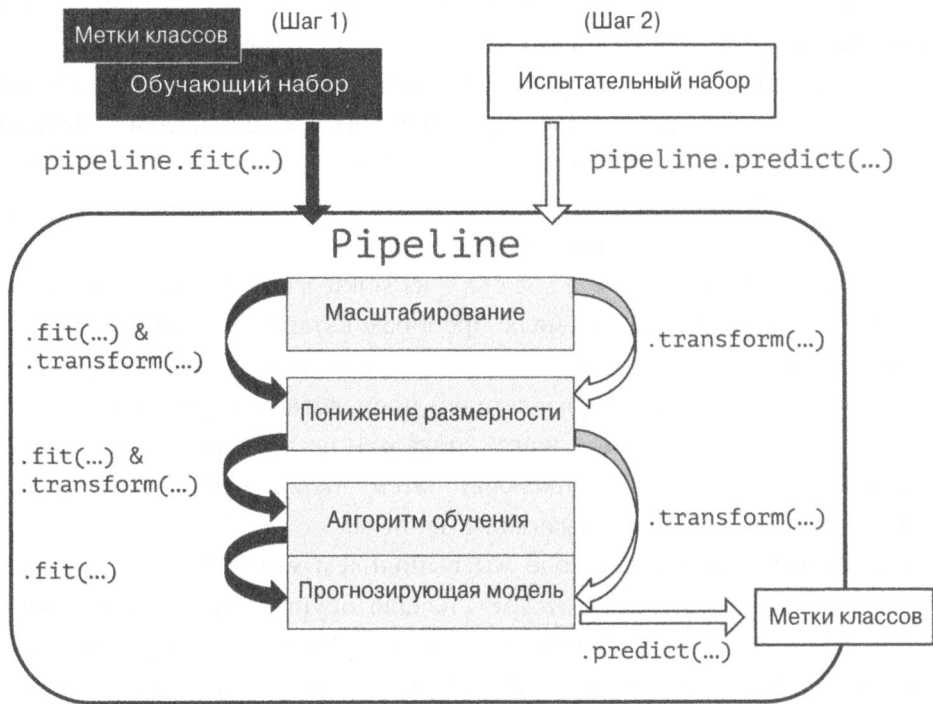


Рис. 6.1. Работа объекта Pipeline

## Использование перекрестной проверки по $k$ блокам для оценки эффективности модели

Одним из ключевых шагов при построении модели МО является оценка ее эффективности на данных, с которыми модель ранее не встречалась. Предположим, что мы подгоняем модель на обучающем наборе данных и применяем те же данные для оценки, насколько хорошо она работает на новых данных. Как мы помним из раздела “Решение проблемы переобучения с помощью регуляризации” главы 3, модель может либо страдать от недообучения (высокого смещения), если она слишком проста, либо переобучаться на обучающих данных (иметь высокую дисперсию), если она чересчур сложна для лежащих в основе обучающих данных.

Чтобы отыскать приемлемый компромисс между смещением и дисперсией, нам необходимо тщательно оценивать модель. В этом разделе вы узнаете о распространенных приемах *перекрестной проверки с удержанием*

(*holdout cross-validation*) и *перекрестной проверки по  $k$  блокам* (*k-fold cross-validation*), которые могут помочь получить надежные оценки эффективности обобщения модели, т.е. выяснить, насколько хорошо модель работает на не встречавшихся ранее данных.

## Метод перекрестной проверки с удержанием

Классическим и популярным подходом для оценки эффективности обобщения моделей МО является перекрестная проверка с удержанием. При такой проверке мы разбиваем начальный набор данных на отдельные обучающий и испытательный наборы; первый используется для обучения модели, а второй — для оценки ее эффективности обобщения. Однако в типовых приложениях МО нас также интересует настройка и сравнение значений различных параметров с целью дальнейшего повышения эффективности выработки прогнозов на не встречавшихся ранее данных. Такой процесс называется *отбором модели*, причем термин “отбор модели” относится к заданной задаче классификации, для *настраиваемых параметров* (также называемых *гиперпараметрами*) которой мы хотим подобрать *оптимальные* значения. Тем не менее, если во время отбора модели мы многократно применяем тот же самый испытательный набор данных, то он становится частью нашего обучающего набора и потому растет вероятность переобучения модели. Несмотря на упомянутую проблему, многие продолжают использовать испытательный набор для отбора модели, что не считается рекомендуемой практикой МО.

Более удачный способ применения метода перекрестной проверки с удержанием для отбора модели предусматривает разбиение данных на три части: обучающий набор, проверочный набор и испытательный набор. Обучающий набор используется для подгонки разных моделей, а показатели эффективности работы на проверочном наборе применяются при отборе модели. Преимущество наличия испытательного набора, с которым модель не встречалась ранее при выполнении шагов обучения и отбора, заключается в том, что мы можем получить менее смещенную оценку способности модели к обобщению на новые данные. На рис. 6.2 иллюстрируется концепция перекрестной проверки с удержанием, где мы неоднократно используем проверочный набор для оценки эффективности модели после ее обучения с применением разных значений параметров. После получения удовлетво-

рительных результатов настройки значений гиперпараметров мы оцениваем эффективность обобщения моделей на испытательном наборе.

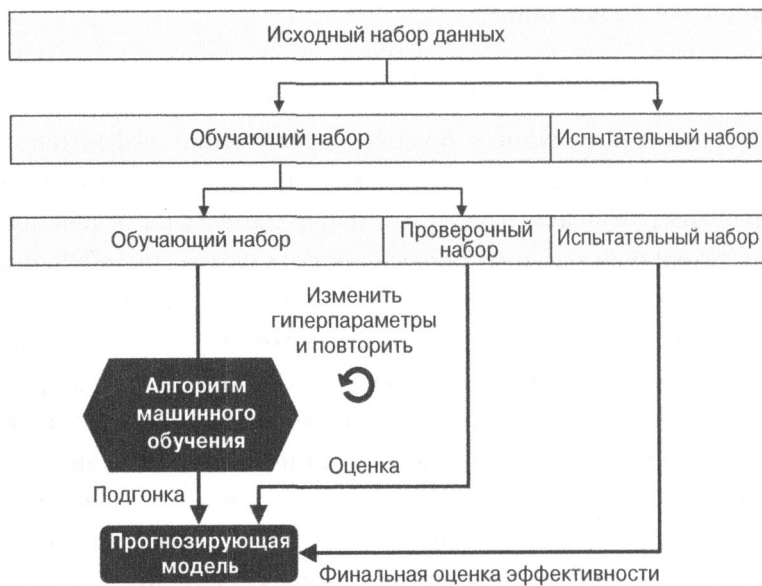


Рис. 6.2. Концепция перекрестной проверки с удержанием

Недостаток метода перекрестной проверки с удержанием в том, что оценка эффективности может быть очень чувствительной к способу разбиения обучающего набора на обучающий и проверочный поднаборы; для разных образцов данных оценка будет варьироваться. В следующем подразделе мы рассмотрим более надежный прием оценки эффективности — перекрестную проверку по  $k$  блокам, при которой мы повторяем метод перекрестной проверки с удержанием  $k$  раз на  $k$  поднаборах обучающих данных.

## Перекрестная проверка по $k$ блокам

При перекрестной проверке по  $k$  блокам мы случайным образом разбиваем обучающий набор данных на  $k$  блоков без возвращения, где  $k-1$  блоков используются для обучения моделей и один блок применяется для оценки эффективности. Указанная процедура повторяется  $k$  раз, так что мы получаем  $k$  моделей и оценок эффективности.



## Выборка с возвращением и без возвращения

В главе 3 мы приводили пример, иллюстрирующий выборку с возвращением и без возвращения. Если вы еще не читали эту главу или просто хотите освежить память, тогда обратитесь к врезке “Выборка с возвращением и без возвращения” в разделе “Объединение множества деревьев принятия решений с помощью случайных лесов” главы 3.

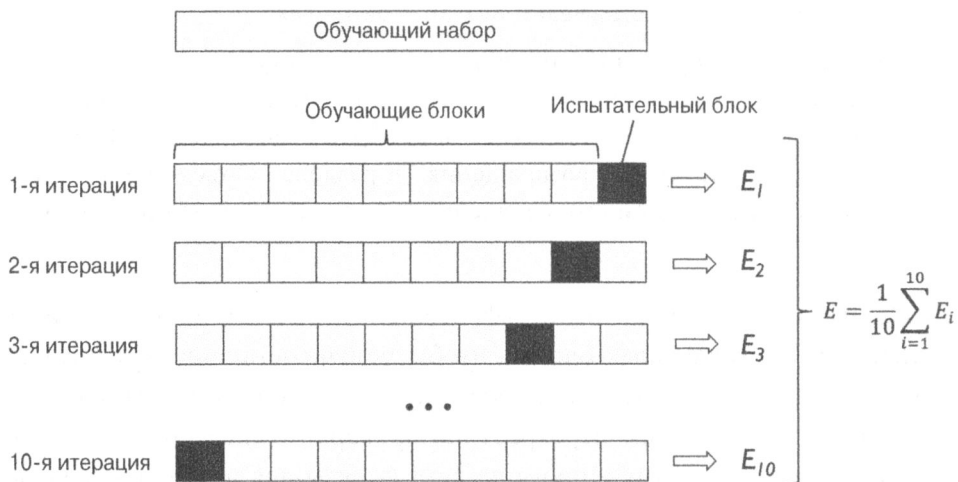
Затем мы определяем среднюю эффективность моделей на основе разных независимых испытательных блоков, чтобы получить оценку эффективности, которая менее чувствительна к добавочным разбиениям обучающих данных в сравнении с методом перекрестной проверки с удержанием. Обычно мы используем перекрестную проверку по  $k$  блокам для настройки моделей, т.е. нахождения оптимальных значений гиперпараметров, которые обеспечат удовлетворительную эффективность обобщения, полученную из оценки эффективности моделей на испытательных блоках.

После того, как приемлемые значения гиперпараметров найдены, мы можем повторно обучить модель на полном обучающем наборе и получить финальную оценку эффективности с применением независимого испытательного набора. Логическое обоснование подгонки модели к полному обучающему набору после перекрестной проверки по  $k$  блокам заключается в том, что предоставление алгоритму обучения большего количества обучающих образцов обычно дает в результате более точную и надежную модель.

Поскольку перекрестная проверка по  $k$  блокам является приемом повторной выборки без возвращения, ее преимущество в том, что каждая выборочная точка будет использоваться для обучения и проверки (как часть испытательного блока) в точности один раз, давая в итоге оценку с меньшей дисперсией, чем метод перекрестной проверки с удержанием.

На рис. 6.3 подытожена концепция перекрестной проверки по  $k$  блокам с  $k = 10$ . Обучающий набор данных разделяется на 10 блоков и в течение 10 итераций девять блоков применяются для обучения, а один блок будет использоваться как испытательный набор для оценки модели. Затем оценки эффективности  $E_i$  (например, правильность или ошибка классификации) для каждого блока применяются при вычислении средней оценочной эффективности  $E$  модели.





**Рис. 6.3.** Концепция перекрестной проверки по  $k$  блокам с  $k = 10$

Как показывают эмпирические данные, хорошим стандартным значением для  $k$  в перекрестной проверке по  $k$  блокам является 10. Например, эксперименты, проведенные Ронем Кохави на разнообразных реальных наборах данных, наводят на мысль, что перекрестная проверка по 10 блокам обеспечивает наилучший компромисс между смещением и дисперсией (“A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection” (“Исследование перекрестной проверки и бутстрэппинга для оценки правильности и отбора модели”), Рон Кохави, итоги международной конференции по искусственному интеллекту (IJCAI), 14 (12): стр. 1137–1143 (1995 год)).

Однако если мы работаем с относительно малыми обучающими наборами, то может быть полезно увеличить количество блоков. С увеличением значения  $k$  на каждой итерации будет использоваться больше обучающих данных, что дает в результате менее пессимистическое смещение по отношению к оценке эффективности обобщения путем усреднения оценок индивидуальных моделей. Тем не менее, крупные значения  $k$  также будут увеличивать время выполнения алгоритма перекрестной проверки, приводя к выдаче оценок с более высокой дисперсией, т.к. обучающие блоки станут в большей степени похожими друг на друга. С другой стороны, если мы работаем с крупными наборами данных, тогда можем выбирать меньшее значение для  $k$ , скажем,  $k=5$ , и по-прежнему получать точную оценку средней эффективности модели, одновременно сокращая вычислительные затраты на повторную подгонку и оценку модели на разных блоках.



## Перекрестная проверка по одному

Специальным случаем перекрестной проверки по  $k$  блокам является метод *перекрестной проверки по одному* (*leave-one-out cross-validation* — LOOCV). В методе LOOCV мы устанавливаем количество блоков равным числу обучающих образцов ( $k=n$ ), поэтому на каждой итерации для испытания применяется только один обучающий образец, что будет рекомендуемым подходом при работе с очень маленькими наборами данных.

Незначительным усовершенствованием подхода перекрестной проверки по  $k$  блокам считается стратифицированная перекрестная проверка по  $k$  блокам, которая может давать оценки с лучшим смещением и дисперсией, особенно в случаях неравных долей классов, как было показано в упомянутой ранее работе Рона Кохави. При стратифицированной перекрестной проверке доли классов предохраняются в каждом блоке, гарантируя тем самым, что каждый блок отображает доли классов в обучающем наборе данных, как мы продемонстрируем с использованием итератора `StratifiedKFold` из `scikit-learn`:

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold
>>> kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print('Блок: %2d, Распределение классов: %s, Правильность: %.3f'
...           % (k+1, np.bincount(y_train[train]), score))
Блок: 1, Распределение классов: [256 153], Правильность: 0.935
Блок: 2, Распределение классов: [256 153], Правильность: 0.935
Блок: 3, Распределение классов: [256 153], Правильность: 0.957
Блок: 4, Распределение классов: [256 153], Правильность: 0.957
Блок: 5, Распределение классов: [256 153], Правильность: 0.935
Блок: 6, Распределение классов: [257 153], Правильность: 0.956
Блок: 7, Распределение классов: [257 153], Правильность: 0.978
Блок: 8, Распределение классов: [257 153], Правильность: 0.933
Блок: 9, Распределение классов: [257 153], Правильность: 0.956
Блок: 10, Распределение классов: [257 153], Правильность: 0.956
```

```
>>> print('\nТочность перекрестной проверки: %.3f +/- %.3f' %  
...       (np.mean(scores), np.std(scores)))  
Точность перекрестной проверки: 0.950 +/- 0.014
```

Прежде всего, мы инициализируем итератор `StratifiedKFold` из модуля `sklearn.model_selection` метками классов `y_train` в обучающем наборе и с помощью параметра `n_splits` указываем количество блоков. Когда мы посредством итератора `kfold` проходим по  $k$  блокам, возвращаемые в `train` индексы применяются для подгонки конвейера с объектом логистической регрессии, который был настроен в начале главы. Используя конвейер `pipe_lr`, мы гарантируем, что на каждой итерации образцы надлежащим образом масштабированы (например, стандартизированы). Затем мы применяем индексы `test` для вычисления меры правильности модели, которую помещаем в список `scores`, чтобы вычислить среднюю точность и стандартное отклонение оценки.

Хотя приведенный выше пример кода был удобен в качестве иллюстрации работы перекрестной проверки по  $k$  блокам, в библиотеке `scikit-learn` также реализован счетчик перекрестной проверки по  $k$  блокам, который позволяет оценивать модель с использованием стратифицированной перекрестной проверки по  $k$  блокам более кратко:

```
>>> from sklearn.model_selection import cross_val_score  
>>> scores = cross_val_score(estimator=pipe_lr,  
...                          X=X_train,  
...                          y=y_train,  
...                          cv=10,  
...                          n_jobs=1)  
>>> print('Меры правильности перекрестной проверки: %s' % scores)  
Меры правильности перекрестной проверки: [  
0.93478261 0.93478261 0.95652174  
0.95652174 0.93478261 0.95555556  
0.97777778 0.93333333 0.95555556  
0.95555556 ]  
>>> print('Точность перекрестной проверки: %.3f +/- %.3f'  
...       % (np.mean(scores),  
...         np.std(scores)))  
Точность перекрестной проверки: 0.950 +/- 0.014
```

Исключительно полезная особенность подхода с `cross_val_score` связана с тем, что мы можем распределить оценку разных блоков по множеству

процессоров на машине. Если мы установим параметр `n_jobs` в 1, тогда для оценки эффективности будет применяться только один процессор, как было ранее в примере с итератором `StratifiedKFold`. Однако за счет установки `n_jobs=2` мы могли бы распределить 10 итераций перекрестной проверки по двум процессорам (при их наличии на машине), а путем установки `n_jobs=-1` задействовать все доступные процессоры для выполнения вычислений в параллельном режиме.



На  
заметку!

### Оценка эффективности обобщения

Обратите внимание, что подробное обсуждение способа, которым оценивается дисперсия эффективности обобщения при перекрестной проверке, выходит за рамки тематики настоящей книги. При желании вы можете обратиться к исчерпывающей статье об оценке моделей и перекрестной проверке (“Model evaluation, model selection, and algorithm selection in machine learning” (“Оценка моделей, подбор моделей и выбор алгоритмов в машинном обучении”), С. Рашка, препринт arXiv:1811.12808 (2018 г.)), где указанные темы раскрываются более глубоко. Статья свободно доступна по ссылке <https://arxiv.org/pdf/1811.12808.pdf>.

Вдобавок вы можете найти детальное обсуждение в великолепной статье М. Маркату и других “Analysis of Variance of Cross-validation Estimators of the Generalization Error” (“Анализ дисперсии оценщиков ошибки обобщения при перекрестной проверке”), *Journal of Machine Learning Research*, 6: с. 1127–1168 (2005 г.).

Вы также можете почитать об альтернативных приемах перекрестной проверки вроде метода перекрестной проверки с бутстрэппингом .632 (“Improvements on Cross-validation: The .632+ Bootstrap Method” (“Усовершенствования перекрестной проверки: метод бутстрэппинга .632”), Б. Эфрон и Р. Тибширани, *Journal of the American Statistical Association*, 92(438): с. 548–560 (1997 г.)).

## Отладка алгоритмов с помощью кривых обучения и проверки

В этом разделе мы взглянем на два очень простых, но мощных диагностических инструмента, которые могут помочь повысить эффективность алгоритма обучения: *кривые обучения* и *кривые проверки*. В последующих

подразделах мы обсудим, как можно использовать кривые обучения для установления, есть ли у алгоритма обучения проблема с переобучением (высокая дисперсия) или недообучением (высокое смещение). К тому же мы рассмотрим кривые проверки, которые могут помочь с решением распространенных проблем с алгоритмом обучения.

## **Диагностирование проблем со смещением и дисперсией с помощью кривых обучения**

Если модель чересчур сложная для имеющегося обучающего набора данных (в ней присутствует слишком много степеней свободы или параметров), то она имеет склонность к переобучению обучающими данными и не обобщается хорошо на не встречавшиеся ранее данные. Зачастую сократить степень переобучения может помочь сбор добавочных образцов.

Тем не менее, на практике сбор дополнительных данных часто оказывается крайне затратным или попросту неосуществимым. Построив графики показателей правильности обучения и проверки как функций размера обучающего набора, мы сможем легко обнаруживать, страдает ли модель от высокой дисперсии или высокого смещения, а также выяснять, поможет ли сбор добавочных данных решить имеющуюся проблему. Но прежде чем мы посмотрим, каким образом строить кривые обучения в `scikit-learn`, давайте обсудим две распространенные проблемы с моделями, проработав графики на рис. 6.4.

На графике вверху слева показана модель с высоким смещением. Эта модель характеризуется низкими показателями правильности обучения и перекрестной проверки, которые говорят о том, что она недообучена на обучающих данных. Общепринятые способы решения такой проблемы предусматривают увеличение количества параметров модели, например, путем сбора или конструирования дополнительных признаков, либо уменьшение степени регуляризации, скажем, в классификаторах на основе SVM или логистической регрессии.

На графике вверху справа представлена модель, которая страдает от высокой дисперсии, на что указывает крупный разрыв между показателями правильности обучения и перекрестной проверки. Чтобы решить проблему переобучения, мы можем собрать больше обучающих данных, снизить сложность модели либо увеличить значение параметра регуляризации.

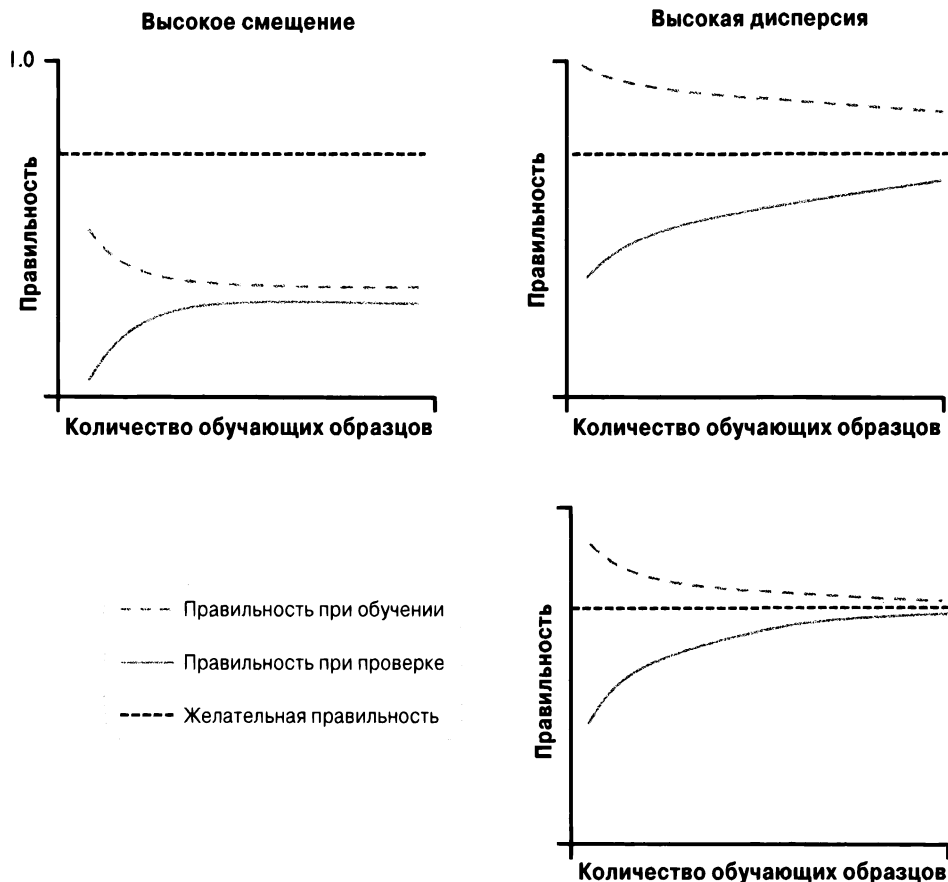


Рис. 6.4. Иллюстрация двух распространенных проблем с моделями

В случае нерегуляризованных моделей снижению степени переобучения может также помочь уменьшение количества признаков посредством выбора признаков (см. главу 4) или выделения признаков (см. главу 5). Наряду с тем, что сбор добавочных обучающих данных обычно имеет тенденцию снизить шансы переобучения, он не всегда может помочь, скажем, если обучающие данные крайне зашумлены или модель уже очень близка к оптимальной.

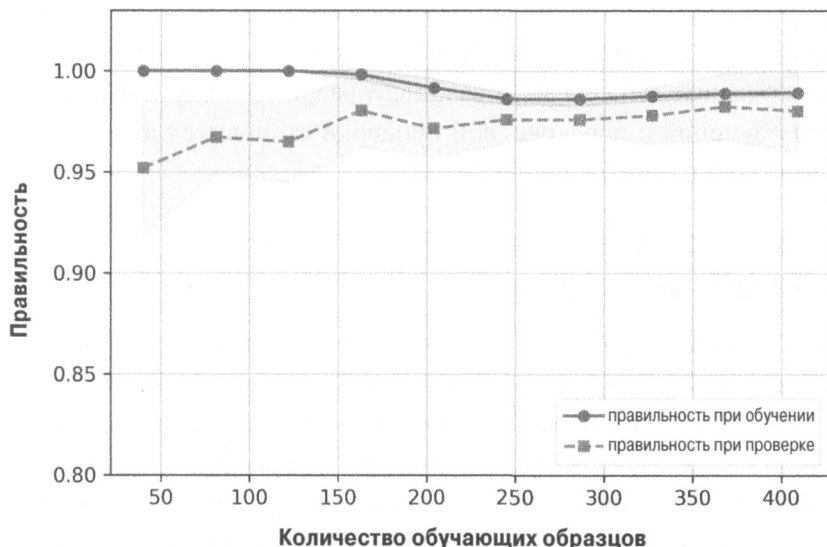
В следующем подразделе мы покажем, каким образом решать такие проблемы с моделью, применяя кривые проверки, но давайте сначала посмотрим, как можно оценивать модель посредством функции кривой обучения (`learning_curve`) из `scikit-learn`:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.model_selection import learning_curve

>>> pipe_lr = make_pipeline(StandardScaler(),
...                          LogisticRegression(penalty='l2',
...                                             random_state=1,
...                                             solver='lbfgs',
...                                             max_iter=10000))
>>> train_sizes, train_scores, test_scores = \
...     learning_curve(estimator=pipe_lr,
...                     X=X_train,
...                     y=y_train,
...                     train_sizes=np.linspace(
...                         0.1, 1.0, 10),
...                     cv=10,
...                     n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)

>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5, label='правильность при обучении')
>>> plt.fill_between(train_sizes,
...                   train_mean + train_std,
...                   train_mean - train_std,
...                   alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='правильность при проверке')
>>> plt.fill_between(train_sizes,
...                   test_mean + test_std,
...                   test_mean - test_std,
...                   alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Количество обучающих образцов')
>>> plt.ylabel('Правильность')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.03])
>>> plt.show()
```

Обратите внимание, что при создании объекта `LogisticRegression` (который по умолчанию использует 1000 итераций) мы передаем `max_iter=10000` в качестве дополнительного аргумента, чтобы избежать проблем со сходимостью для меньших размеров наборов данных или экстремальных значений параметра регуляризации (рассматривается в следующем разделе). В результате успешного выполнения предыдущего кода мы получаем график кривой обучения, показанный на рис. 6.5.



*Рис. 6.5. График кривой обучения, построенный с помощью функции `learning_curve` из `scikit-learn`*

С помощью параметра `train_sizes` в функции `learning_curve` мы можем управлять абсолютным или относительным количеством обучающих образцов, которые используются для генерации кривых обучения. Здесь мы устанавливаем `train_sizes=np.linspace(0.1, 1.0, 10)`, чтобы применять 10 равноотстоящих относительных интервалов для размеров обучающих наборов данных. По умолчанию функция `learning_curve` для вычисления правильности перекрестной проверки классификатора использует стратифицированную перекрестную проверку по  $k$  блокам, и мы устанавливаем  $k=10$  через параметр `cv`, чтобы применять стратифицированную перекрестную проверку по 10 блокам.



Далее мы просто рассчитываем средние показатели правильности на основе возвращенных показателей правильности обучения и перекрестной проверки для разных размеров обучающего набора и посредством функции `plot` из `Matplotlib` строим график. Кроме того, мы добавляем к графику стандартное отклонение средней правильности, используя функцию `fill_between` для обозначения дисперсии оценки.

Как можно было заметить на предыдущем графике с кривой обучения, наша модель работает на обучающем и проверочном наборах данных довольно хорошо, если во время обучения она встретила более 250 образцов. Также видно, что правильность обучения увеличивается для обучающих наборов, содержащих менее 250 образцов, а разрыв между показателями правильности обучения и перекрестной проверки становится шире, указывая на растущую степень переобучения.

## Решение проблем недообучения и переобучения с помощью кривых проверки

Кривые проверки являются полезным инструментом для повышения эффективности модели за счет решения таких проблем, как переобучение и недообучение. Кривые проверки имеют отношение к кривым обучения, но вместо построения графика с показателями правильности при обучении и испытаниях как функций размера выборки мы варьируем значения параметров модели, например, обратного параметра регуляризации  $C$  в логистической регрессии. Давайте посмотрим, как создавать кривые проверки посредством `scikit-learn`:

```
>>> from sklearn.model_selection import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...     estimator=pipe_lr,
...     X=X_train,
...     y=y_train,
...     param_name='logisticregression__C',
...     param_range=param_range,
...     cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
```

```

>>> plt.plot(param_range, train_mean,
...           color='blue', marker='o',
...           markersize=5, label='правильность при обучении')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...           color='green', linestyle='--',
...           marker='s', markersize=5,
...           label='правильность при проверке')
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Параметр C')
>>> plt.ylabel('Правильность')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()

```

В результате выполнения приведенного выше кода мы получаем график с кривой проверки для параметра  $C$  (рис. 6.6).

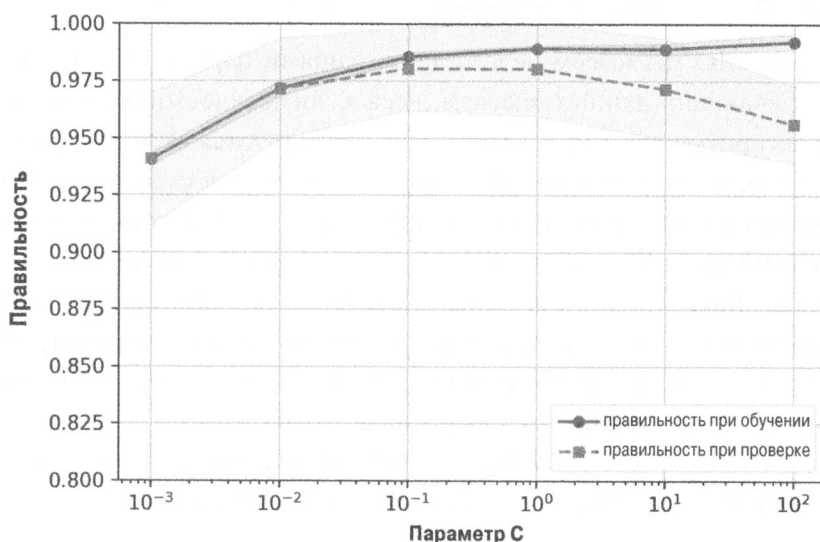


Рис. 6.6. График кривой проверки для параметра  $C$

Подобно функции `learning_curve` для оценки эффективности классификатора функция `validation_curve` по умолчанию применяет стратифицированную перекрестную проверку по  $k$  блокам. Внутри функции `validation_curve` мы указали параметр, который хотим оценить. В данном случае это  $C$ , обратный параметр регуляризации классификатора `LogisticRegression`, который мы записали как `'logisticregression__C'`, чтобы иметь доступ к объекту `LogisticRegression` внутри конвейера `scikit-learn` для указанного диапазона значений, установленного через параметр `param_range`. Как и в примере с кривой обучения в предыдущем разделе, мы строим график со средними показателями правильности при обучении и при перекрестной проверке и соответствующими стандартными отклонениями.

Хотя отличия в правильности для варьирующихся значений едва различимы, мы можем заметить, что модель слегка недообучается на данных при увеличении силы регуляризации (небольшие значения  $C$ ). Однако крупные величины  $C$  приводят к уменьшению силы регуляризации, поэтому модель склонна к незначительному переобучению данными. В рассматриваемом случае все выглядит так, что наилучшие значения  $C$  находятся в диапазоне между 0.01 и 0.1.

## Точная настройка моделей машинного обучения с помощью решетчатого поиска

В области МО мы имеем дело с двумя типами параметров: те, что узнаются из обучающих данных, скажем, веса в логистической регрессии, и параметры алгоритма обучения, которые оптимизируются отдельно. Последние являются параметрами настройки модели (или *гиперпараметрами*); примером гиперпараметра может служить параметр регуляризации в логистической регрессии или параметр глубины в дереве принятия решений.

В предыдущем разделе мы использовали кривые проверки для повышения эффективности модели, настраивая один из ее гиперпараметров. В текущем разделе мы рассмотрим популярный прием оптимизации гиперпараметров, который называется *решетчатым поиском* (*grid search*) и может дополнительно содействовать повышению эффективности модели, отыскивая *оптимальную* комбинацию значений гиперпараметров.

## Настройка гиперпараметров с помощью решетчатого поиска

Подход, лежащий в основе решетчатого поиска, довольно прост. Это парадигма исчерпывающего поиска методом грубой силы, когда мы указываем список значений для различных гиперпараметров, а компьютер оценивает эффективность модели для каждого их сочетания, чтобы получить оптимальную комбинацию значений из списка:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import SVC

>>> pipe_svc = make_pipeline(StandardScaler(),
...                           SVC(random_state=1))
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'svc__C': param_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': param_range,
...                'svc__gamma': param_range,
...                'svc__kernel': ['rbf']}]

>>> gs = GridSearchCV(estimator=pipe_svc,
...                    param_grid=param_grid,
...                    scoring='accuracy',
...                    cv=10,
...                    refit=True,
...                    n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.9846153846153847
>>> print(gs.best_params_)
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

В показанном выше коде мы инициализируем объект `GridSearchCV` из модуля `sklearn.model_selection` с целью обучения и настройки конвейера SVM. Мы устанавливаем параметр `param_grid` объекта `GridSearchCV` в список словарей, чтобы указать параметры, которые хотим настроить. Для линейного SVM мы оцениваем только обратный параметр регуляризации `C`; для SVM с ядром RBF мы настраиваем параметры `svc__C` и `svc__gamma`. Обратите внимание, что параметр `svc__gamma` специфичен для ядерных SVM.

После применения обучающих данных для проведения решетчатого поиска мы получаем меру лучше всех работающей модели через атрибут `best_score_` и просматриваем ее параметры, доступные посредством атрибута `best_params_`. В нашем конкретном случае модель SVM с ядром RBF при `svc__C=100.0` дает наилучшую правильность при перекрестной проверке по  $k$  блокам: 98.5%.

Наконец, мы будем использовать независимый испытательный набор данных для оценки эффективности выбранной лучшей модели с помощью оценщика, доступного через атрибут `best_estimator_` объекта `GridSearchCV`:

```
>>> clf = gs.best_estimator_  
>>> clf.fit(X_train, y_train)  
>>> print('Правильность при испытании: %.3f'  
...       % clf.score(X_test, y_test))  
Правильность при испытании: 0.974
```

Следует отметить, что после завершения решетчатого поиска подгонять модель с наилучшими настройками (`gs.best_estimator_`) к обучающему набору вручную посредством метода `clf.fit(X_train, y_train)` необязательно. Класс `GridSearchCV` имеет параметр `refit`, установка которого в `True` (стандартное значение) обеспечит автоматическую подгонку `gs.best_estimator_` к целому обучающему набору.



На заметку!

### Рандомизированный поиск гиперпараметров

Несмотря на то что решетчатый поиск — мощный подход для нахождения оптимального набора параметров, оценка всех возможных комбинаций параметров также сопряжена с очень большими вычислительными затратами. Альтернативным подходом к выборке различных комбинаций параметров с применением `scikit-learn` служит *рандомизированный поиск* (*randomized search*). Рандомизированный поиск обычно работает почти так же, как решетчатый поиск, но он гораздо экономичнее и эффективнее по времени. В частности, если мы отбираем только 60 комбинаций параметров через рандомизированный поиск, то уже имеем 95%-ную вероятность получения решений в пределах 5%-ной оптимальной эффективности (“Random search for hyper-parameter optimization” (Случайный поиск для оп-

тимизации гиперпараметров), Дж. Бергстра, Й. Бенджи, *Journal of Machine Learning Research*. с. 281–305 (2012 г.)).

С использованием класса `RandomizedSearchCV` из библиотеки `scikit-learn` мы осуществляем выборку случайных комбинаций параметров из выборочного распределения с указанным запасом. Дополнительные сведения и примеры применения класса `RandomizedSearchCV` можно найти по ссылке [http://scikit-learn.org/stable/modules/grid\\_search.html#randomized-parameter-optimization](http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization).

## Отбор алгоритма с помощью вложенной перекрестной проверки

Использование перекрестной проверки по  $k$  блокам в сочетании с решетчатым поиском — удобный подход для точной настройки эффективности модели МО путем варьирования значений ее гиперпараметров, как мы видели в предыдущем подразделе. Тем не менее, если мы хотим производить отбор среди разных алгоритмов МО, то еще одним рекомендуемым подходом является вложенная перекрестная проверка. Судхир Варма и Ричард Саймон в своем элегантном исследовании смещения в ошибках оценки сделали вывод о том, что при вложенной перекрестной проверке истинная ошибка оценки оказывается почти не смещенной относительно испытательного набора (“Bias in Error Estimation When Using Cross-validation for Model Selection” (Смещение в ошибках оценки при использовании перекрестной проверки для отбора модели), С. Варма и Р. Саймон, *BMC Bioinformatics*, 7(1): с. 91 (2006 г.)).

При вложенной перекрестной проверке мы организуем внешний цикл перекрестной проверки по  $k$  блокам для разбиения данных на обучающие и испытательные блоки, а также применяем внутренний цикл для отбора модели с использованием перекрестной проверки по  $k$  блокам на обучающем блоке. После отбора модели оценивается ее эффективность с применением испытательного блока. На рис. 6.7 объясняется концепция вложенной перекрестной проверки с пятью внешними и двумя внутренними блоками, которая может быть полезна для крупных наборов данных, где важна вычислительная эффективность. Указанный конкретный тип вложенной перекрестной проверки также известен как *перекрестная проверка 5×2*.

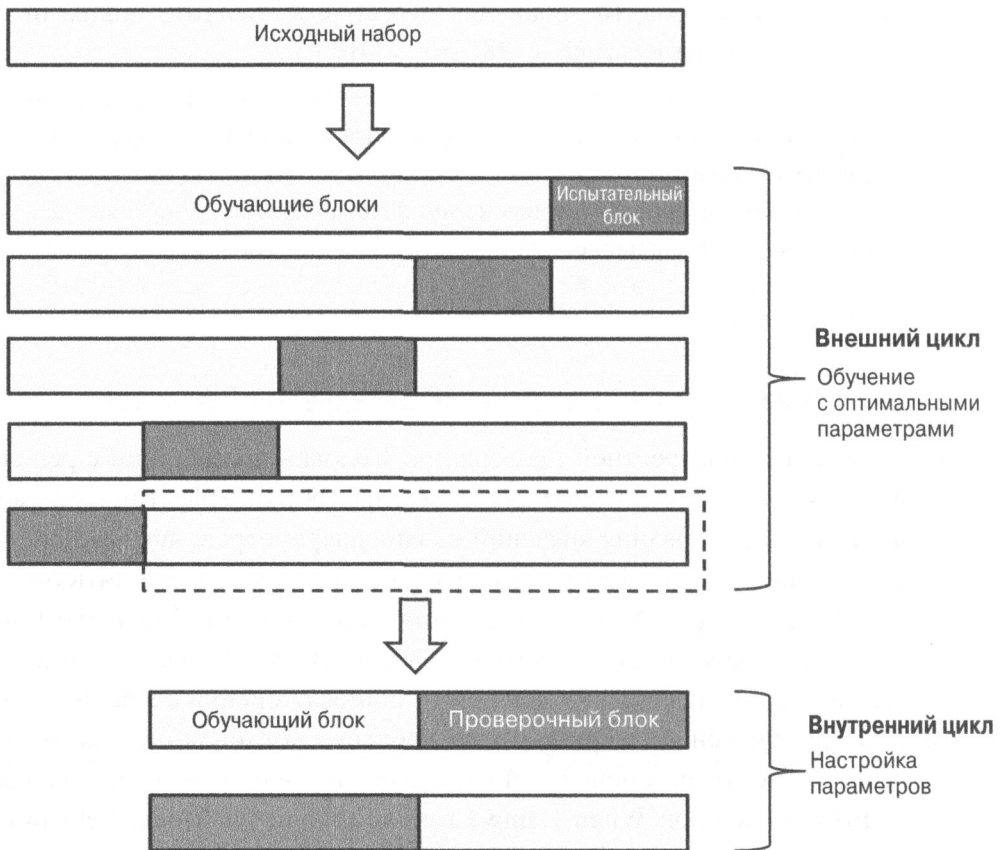


Рис. 6.7. Концепция перекрестной проверки 5×2

С помощью библиотеки `scikit-learn` мы можем выполнять вложенную перекрестную проверку следующим образом:

```
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2)
>>> scores = cross_val_score(gs, X_train, y_train,
...                           scoring='accuracy', cv=5)
>>> print('Точность перекрестной проверки: %.3f +/- %.3f'
...       % (np.mean(scores),
...          np.std(scores)))
Точность перекрестной проверки: 0.974 +/- 0.015
```

Возвращенный средний показатель правильности перекрестной проверки дает нам хорошую оценку того, чего можно ожидать после настройки гиперпараметров модели и ее использования на не встречавшихся ранее данных.

Например, мы можем применять подход с вложенной перекрестной проверкой для сравнения модели SVM с простым классификатором на основе дерева принятия решений; в целях упрощения мы будем настраивать только параметр глубины:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(estimator=DecisionTreeClassifier(
...                     random_state=0),
...                     param_grid=[{'max_depth': [1, 2, 3,
...                                                 4, 5, 6,
...                                                 7, None]}],
...                     scoring='accuracy',
...                     cv=2)
>>> scores = cross_val_score(gs, X_train, y_train,
...                             scoring='accuracy', cv=5)
>>> print('Точность перекрестной проверки: %.3f +/- %.3f'
...       % (np.mean(scores),
...          np.std(scores)))
Точность перекрестной проверки: 0.934 +/- 0.016
```

Мы видим, что эффективность вложенной перекрестной проверки модели SVM (97.4%) заметно выше эффективности при варианте с деревом принятия решений (93.4%). Вследствие этого мы ожидаем, что вариант с SVM может оказаться лучшим выбором для классификации новых данных, которые поступают из той же генеральной совокупности, откуда поступил наш задействованный выше набор данных.

## Использование других метрик оценки эффективности

В предшествующих разделах и главах мы оценивали разные модели МО с использованием правильности прогнозов — практичной метрики, с помощью которой количественно оценивается эффективность модели в целом. Однако существует несколько других метрик эффективности, которые можно применять для измерения степени соответствия модели, такие как *точность* (*precision*), *полнота* (*recall*) и *мера F1* (*F1 score*).



## Чтение матрицы неточностей

Прежде чем мы погрузимся в детали различных метрик подсчета, давайте посмотрим, что собой представляет *матрица неточностей* (*confusion matrix*), которая раскладывает эффективность алгоритма обучения.

Матрица неточностей — это просто квадратная матрица, описывающая численности *истинно положительных* (*true positive* — *TP*), *истинно отрицательных* (*true negative* — *TN*), *ложноположительных* (*false positive* — *FP*) и *ложноотрицательных* (*false negative* — *FN*) прогнозов классификатора (рис. 6.8).

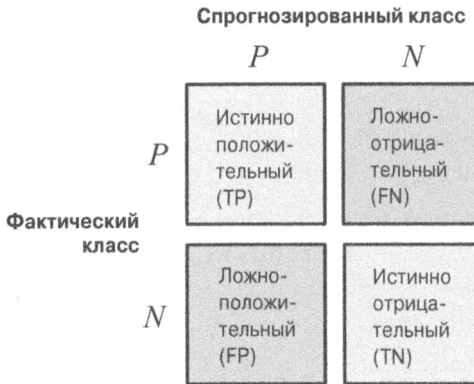


Рис. 6.8. Матрица неточностей

Невзирая на то что такие метрики легко вычислить вручную, сравнивая настоящие и спрогнозированные метки классов, библиотека *scikit-learn* предлагает удобную функцию `confusion_matrix`, которой мы и воспользуемся:

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[ 71  1]
 [ 2 40]]
```

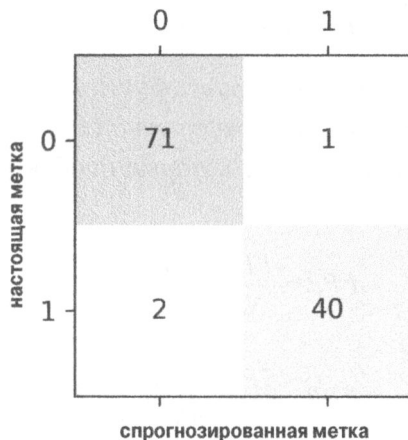
Возвращаемый в результате выполнения кода массив снабжает нас информацией об ошибках разных типов, допущенных классификатором на испытательном наборе данных. С применением функции `matshow` из *Matplotlib* мы можем сопоставить эту информацию с иллюстрацией матрицы неточностей, приведенной на рис. 6.8:

```

>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i,
...                 s=confmat[i, j],
...                 va='center', ha='center')
>>> plt.xlabel('Спрогнозированная метка')
>>> plt.ylabel('Настоящая метка')
>>> plt.show()

```

Полученная матрица неточностей с добавленными метками (рис. 6.9) должна немного облегчить интерпретацию результатов.



**Рис. 6.9.** Матрица неточностей с добавленными метками

Предполагая, что класс 1 (злокачественные опухоли) — положительный в рассматриваемом примере, наша модель корректно классифицировала 71 образец как принадлежащие классу 0 (истинно отрицательные прогнозы) и 40 образцов как принадлежащие классу 1 (истинно положительные прогнозы). Тем не менее, модель также неправильно классифицировала два образца из класса 1 как принадлежащие классу 0 (ложноотрицательные прогнозы) и спрогнозировала, что один образец относится к злокачественным опухолям, хотя он является доброкачественной опухолью (ложноположительный прогноз). В следующем разделе мы покажем, как использовать такую информацию для вычисления разнообразных метрик ошибок.

## Оптимизация точности и полноты классификационной модели

И *ошибка* (*error* — *ERR*), и *правильность* (*accuracy* — *ACC*) прогноза предоставляют общую информацию о том, сколько образцов классифицировано неправильно. Ошибку можно понимать как сумму всех ложных прогнозов, деленную на общее количество прогнозов, а правильность вычисляется как сумма корректных прогнозов, деленная на общее количество прогнозов:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

Тогда правильность прогноза можно вычислить прямо из ошибки:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

*Доля истинно положительных классификаций* (*true positive rate* — *TPR*) и *доля ложноположительных классификаций* (*false positive rate* — *FPR*) являются метриками эффективности, которые особенно полезны в задачах с дисбалансом классов:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Например, при диагностике новообразований нас больше заботит обнаружение злокачественных опухолей, чтобы помочь пациенту, назначив подходящее лечение. Однако важно также уменьшить количество случаев, когда доброкачественные опухоли неправильно классифицируются как злокачественные (прогнозы FP), чтобы без нужды не волновать пациентов. В отличие от FPR величина TPR предоставляет полезную информацию о части положительных (или значимых) образцов из общей совокупности положительных образцов, которые были идентифицированы корректно (P).

Метрики эффективности *точность* (*PRE*) и *полнота* (*REC*) связаны с долями истинно положительных и отрицательных классификаций, причем REC — фактически синоним TPR:

$$PRE = \frac{TP}{TP + FP}$$

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

В контексте примера выявления злокачественных опухолей оптимизация полноты помогает свести к минимуму вероятность не обнаружить злокачественную опухоль. Тем не менее, это происходит за счет прогнозирования злокачественных опухолей у здоровых пациентов (высокое число FP). С другой стороны, если мы оптимизируем точность, тогда придаем особое значение правильности, когда прогнозируем, что у пациента есть злокачественная опухоль. Однако это происходит за счет более частого упущения злокачественных опухолей (высокое число FN).

Чтобы достичь баланса между достоинствами и недостатками оптимизации PRE и REC, часто применяется комбинация PRE и REC — так называемая *мера F1*:

$$F1 = 2 \frac{PRE \times REC}{PRE + REC}$$



### Дополнительные сведения о точности и полноте

Если вас интересует более основательное обсуждение различных метрик эффективности, таких как точность и полнота, прочитайте технический отчет Дэвида М. У. Пауэрса “Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation” (“Оценка: от точности, полноты и F-меры до рабочей характеристики приемника, информированности, степени пометки и корреляции”), который бесплатно доступен по ссылке [https://www.researchgate.net/publication/228529307\\_Evaluation\\_From\\_Precision\\_Recall\\_and\\_F-Factor\\_to\\_ROC\\_Informedness\\_Markedness\\_Correlation](https://www.researchgate.net/publication/228529307_Evaluation_From_Precision_Recall_and_F-Factor_to_ROC_Informedness_Markedness_Correlation).

Все упомянутые метрики подсчета реализованы в `scikit-learn` и могут быть импортированы из модуля `sklearn.metrics`, как демонстрируется в следующем фрагменте кода:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics import recall_score, f1_score
>>> print('Точность: %.3f' % precision_score(
...     y_true=y_test, y_pred=y_pred))
Точность: 0.976
>>> print('Полнота: %.3f' % recall_score(
...     y_true=y_test, y_pred=y_pred))
Полнота: 0.952
>>> print('Мера F1: %.3f' % f1_score(
...     y_true=y_test, y_pred=y_pred))
Мера F1: 0.964
```

Кроме того, посредством параметра `scoring` в `GridSearchCV` мы можем использовать метрику подсчета, отличающуюся от метрики правильности. Полный список значений, принимаемых параметром `scoring`, приводится по ссылке [http://scikit-learn.org/stable/modules/model\\_evaluation.html](http://scikit-learn.org/stable/modules/model_evaluation.html).

Не забывайте, что положительный класс в `scikit-learn` — это класс, помеченный как класс 1. Если нужно указать другую *положительную метку*, тогда посредством функции `make_scorer` мы можем построить собственный счетчик, который затем предоставлять напрямую как аргумент параметру `scoring` в `GridSearchCV` (в текущем примере с применением `f1_score` в качестве метрики):

```
>>> from sklearn.metrics import make_scorer, f1_score
>>> c_gamma_range = [0.01, 0.1, 1.0, 10.0]
>>> param_grid = [{'svc__C': c_gamma_range,
...               'svc__kernel': ['linear']},
...               {'svc__C': c_gamma_range,
...               'svc__gamma': c_gamma_range,
...               'svc__kernel': ['rbf']}]
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.986202145696
>>> print(gs.best_params_)
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

## Построение кривой рабочей характеристики приемника

Кривая *рабочей характеристики приемника* (*receiver operating characteristic* — ROC) является удобным инструментом для отбора классификационных моделей на основе их эффективности относительно долей FPR и TPR, которые вычисляются путем сдвига порога принятия решения у классификатора. Диагональ кривой ROC можно интерпретировать как *случайное угадывание*, и классификационные модели, которые попадают ниже диагонали, считаются хуже случайного угадывания. Идеальный классификатор оказался бы в верхнем левом углу графа с величиной TPR, равной 1, и величиной FPR, равной 0. Основываясь на кривой ROC, мы можем вычислить так называемую *площадь под кривой ROC* (*ROC area under the curve* — ROC AUC) для получения характеристики эффективности классификационной модели.

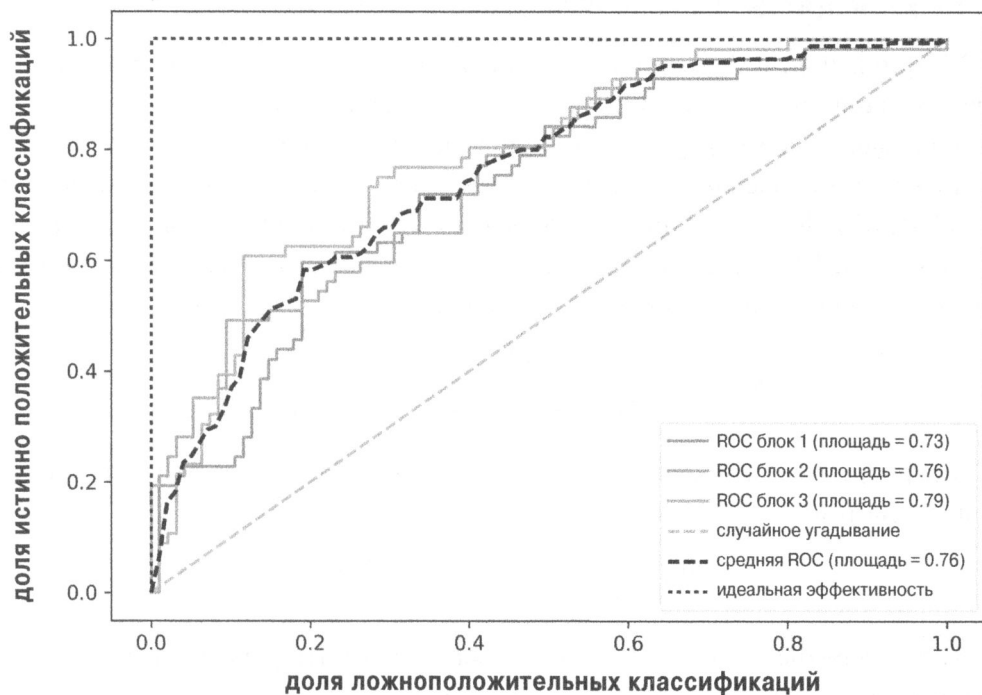
Подобно кривой ROC мы можем построить *кривые точности-полноты* для разных вероятностных порогов классификатора. Функция для построения таких кривых точности-полноты также реализована в scikit-learn и документирована по ссылке [http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision\\_recall\\_curve.html](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html).

Запустив представленный далее код, мы построим кривую ROC классификатора, который использует только два признака из набора данных Breast Cancer Wisconsin для прогнозирования, какой является опухоль — доброкачественной или злокачественной. Хотя мы собираемся применять тот же конвейер логистической регрессии, который определили ранее, на этот раз мы используем только два признака. В итоге задача классификации становится для классификатора более сложной из-за сокрытия полезной информации, содержащейся в остальных признаках, а потому результирующая кривая ROC будет визуально интереснее. По сходным соображениям мы также уменьшим количество блоков в объекте перекрестной проверки StratifiedKFold до трех. Итак, вот надлежащий код:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from scipy import interp
>>> pipe_lr = make_pipeline(StandardScaler(),
...                          PCA(n_components=2),
...                          LogisticRegression(penalty='l2',
...                                              random_state=1,
...                                              solver='lbfgs',
...                                              C=100.0))
```

```
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = list(StratifiedKFold(n_splits=3,
...                             random_state=1).split(X_train,
...                                                     y_train))
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []
>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(
...         X_train2[train],
...         y_train[train]).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                       probas[:, 1],
...                                       pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...               tpr,
...               label='ROC блок %d (площадь = %0.2f)'
...               % (i+1, roc_auc))
>>> plt.plot([0, 1],
...           [0, 1],
...           linestyle='--',
...           color=(0.6, 0.6, 0.6),
...           label='случайное угадывание')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label='средняя ROC (площадь = %0.2f)' % mean_auc, lw=2)
>>> plt.plot([0, 0, 1],
...           [0, 1, 1],
...           linestyle=':',
...           color='black',
...           label='идеальная эффективность')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('доля ложноположительных классификаций')
>>> plt.ylabel('доля истинно положительных классификаций')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

В предыдущем примере кода мы используем уже знакомый класс `StratifiedKfold` из `scikit-learn` и вычисляем эффективность ROC классификатора `LogisticRegression` в конвейере `pipe_lr` с применением функции `roc_curve` из модуля `sklearn.metrics` отдельно для каждой итерации. К тому же мы интерполируем кривую средней ROC трех блоков посредством функции `interp`, которая импортируется из `SciPy`, и вычисляем площадь под кривой с помощью функции `auc`. Результирующая кривая ROC указывает на наличие определенной дисперсии между разными блоками, а средняя площадь под кривой ROC (0.76) попадает между идеальным показателем (1.0) и случайным угадыванием (0.5), как видно на рис. 6.10.



*Рис. 6.10. Кривые ROC с разной площадью под кривой*

Обратите внимание, что если нас интересует только показатель ROC AUC, то мы могли бы также напрямую импортировать функцию `roc_auc_score` из подмодуля `sklearn.metrics`, которую можно использовать аналогично другим функциям подсчета (скажем, `precision_score`), представленным в предшествующих разделах.



Предоставление информации об эффективности классификатора в виде показателя ROC AUC способствует дальнейшему погружению в суть работы классификатора на несбалансированных образцах. Тем не менее, хотя мера правильности может интерпретироваться как одиночная точка отсечения на кривой ROC, Э.П. Брэдли в своей работе продемонстрировал, что показатель ROC AUC и мера правильности обычно согласуются друг с другом (“The use of the area under the ROC curve in the evaluation of machine learning algorithms” (Использование площади под кривой ROC при оценке алгоритмов машинного обучения), Э.П. Брэдли, *Pattern Recognition*, 30(7): с. 1145–1159 (1997 г.)).

## Метрики подсчета для многоклассовой классификации

Метрики подсчета, которые мы обсуждали до сих пор, характерны для систем двоичной классификации. Однако в *scikit-learn* реализованы также методы макро- и микроусреднения для распространения этих метрик подсчета на многоклассовые задачи с помощью классификации “один против всех” (OvA). Микросреднее вычисляется из индивидуальных прогнозов TP, TN, FP и FN системы. Например, микросреднее показателя точности в  $k$ -классовой системе может быть вычислено так:

$$PRE_{\text{микро}} = \frac{TP_1 + \dots + TP_k}{TP_1 + \dots + TP_k + FP_1 + \dots + FP_k}$$

Макросреднее вычисляется просто как среднее из показателей точности разных систем:

$$PRE_{\text{макро}} = \frac{PRE_1 + \dots + PRE_k}{k}$$

Микроусреднение полезно, когда нужно назначить равные веса всем образцам или прогнозам, тогда как макроусреднение назначает равные веса всем классам с тем, чтобы оценить общую эффективность классификатора относительно самых распространенных меток классов.

Если мы применяем двоичные метрики эффективности из *scikit-learn* для оценки моделей многоклассовой классификации, то по умолчанию используется нормализованный или взвешенный вариант макросреднего. Взвешенное макросреднее подсчитывается путем назначения весов показателям всех классов по числу настоящих образцов при вычислении средне-

го. Взвешенное макросреднее удобно, когда мы имеем дело с дисбалансом классов, т.е. разными количествами образцов для каждой метки.

Наряду с тем, что взвешенное макросреднее является стандартным для многоклассовых задач в `scikit-learn`, мы можем указывать метод усреднения через параметр `average` внутри различных функций подсчета из модуля `sklearn.metrics`, например, функции `precision_score` или `make_scorer`:

```
>>> pre_scorer = make_scorer(score_func=precision_score,  
...                          pos_label=1,  
...                          greater_is_better=True,  
...                          average='micro')
```

## Решение проблемы с дисбалансом классов

В главе мы несколько раз упоминали о дисбалансе классов, но пока фактически не обсуждали, как надлежащим образом обрабатывать такие сценарии, когда они возникают. Дисбаланс классов является довольно распространенной проблемой при работе с реальными данными, когда в наборе данных чрезмерно представлены образцы из одного или множества классов. Мы без особого труда можем вспомнить несколько предметных областей, где случается подобное — фильтрация спама, выявление подделок или скрининг заболеваний.

Пусть набор данных `Breast Cancer Wisconsin`, с которым мы работали ранее в главе, насчитывает 90% здоровых пациентов. В таком случае мы могли бы достичь 90%-ной правильности на испытательном наборе данных, просто прогнозируя для всех образцов мажоритарный класс (доброкачественная опухоль) и не прибегая к помощи какого-то алгоритма МО с учителем. Соответственно, обучение модели на наборе данных, которое достигает приблизительно 90%-ной правильности при испытании, означало бы, что наша модель не узнала ничего полезного из признаков, предоставляемых этим набором данных.

В настоящем разделе мы кратко рассмотрим ряд приемов, которые могли бы помочь справиться с несбалансированными наборами данных. Но прежде чем начать обсуждение различных методов решения проблемы, давайте создадим несбалансированный набор данных из набора данных `Breast Cancer Wisconsin`, который первоначально состоял из 357 диагнозов добро-

качественных опухолей (класс 0) и 212 диагнозов злокачественных опухолей (класс 1):

```
>>> X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
>>> y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

В приведенном фрагменте кода мы берем все 357 образцов доброкачественных опухолей и дополняем их первыми 40 образцами злокачественных опухолей, чтобы получить сильный дисбаланс классов. Если бы мы подсчитывали правильность модели, которая всегда прогнозирует мажоритарный класс (доброкачественные опухоли, класс 0), то достигли бы правильности прогнозирования около 90%:

```
>>> y_pred = np.zeros(y_imb.shape[0])
>>> np.mean(y_pred == y_imb) * 100
89.92443324937027
```

Таким образом, когда мы подгоняем классификаторы на наборах данных подобного рода, то при сравнении разных моделей имеет смысл сосредоточиться не на правильности, а на других метриках вроде точности, полноты, кривой ROC — т.е. на всем том, что нас заботит в строящемся приложении. Например, приоритетом могло бы быть установление большинства пациентов со злокачественными опухолями с целью рекомендации дополнительного скрининга; тогда предпочтительной метрикой служила бы полнота. В системе фильтрации спама, где мы не хотим пометчать сообщения как спамные, если в этом нет высокой уверенности, более подходящей метрикой могла бы быть точность.

Помимо оценки моделей МО дисбаланс классов оказывает влияние на алгоритм обучения во время самой подгонки модели. Поскольку алгоритмы МО обычно оптимизируют функцию награды либо издержек, вычисляемую как сумма по обучающим образцам, которые она встречает в течение подгонки, правило принятия решений, вероятно, будет смещенным в направлении мажоритарного класса.

Другими словами, алгоритм неявно обучает модель, которая оптимизирует прогнозы на основе наиболее распространенного класса в наборе данных, чтобы свести к минимуму издержки или довести до максимума награды во время обучения.

Один из способов справиться с несбалансированными долями классов в течение процесса подгонки модели предусматривает назначение более крупных штрафов неправильным прогнозам на миноритарном классе. В `scikit-learn` настройка такого штрафа сводится к установке параметра `class_weight` в `'balanced'`, что реализовано для большинства классификаторов.

Другие популярные стратегии решения проблемы с дисбалансом классов включают повышение дискретизации миноритарного класса, понижение дискретизации мажоритарного класса и генерацию искусственных обучающих образцов. К сожалению, универсального наилучшего решения не существует, и нет приема, который работал бы одинаково хорошо в разных предметных областях. Соответственно, на практике рекомендуется испытать разные стратегии на имеющейся задаче, оценить результаты и выбрать прием, который выглядит самым подходящим.

В библиотеке `scikit-learn` реализована простая функция `resample`, которая может помочь с повышением дискретизации миноритарного класса за счет выборки с возвращением новых образцов из набора данных. В следующем коде мы берем миноритарный класс из нашего несбалансированного набора данных `Breast Cancer Wisconsin` (здесь класс 1) и многократно производим выборку из него новых образцов, пока он не станет содержать такое же количество образцов, как класс 0:

```
>>> from sklearn.utils import resample
>>> print('Начальное количество образцов класса 1:',
...       X_imb[y_imb == 1].shape[0])
Начальное количество образцов класса 1: 40
>>> X_upsampled, y_upsampled = resample(
...     X_imb[y_imb == 1],
...     y_imb[y_imb == 1],
...     replace=True,
...     n_samples=X_imb[y_imb == 0].shape[0],
...     random_state=123)
>>> print('Конечное количество образцов класса 1:',
...       X_upsampled.shape[0])
Конечное количество образцов класса 1: 357
```

После повторной выборки мы можем скомпоновать образцы исходного класса 0 с поднабором образцов, появившихся в результате повышения дискретизации класса 1, чтобы получить сбалансированный набор данных:

```
>>> X_bal = np.vstack((X[y == 0], X_upsampled))
>>> y_bal = np.hstack((y[y == 0], y_upsampled))
```

Вследствие такого подхода правило мажоритарного прогнозирования будет достигать только 50%-ной правильности:

```
>>> y_pred = np.zeros(y_bal.shape[0])
>>> np.mean(y_pred == y_bal) * 100
50
```

Подобным же образом мы могли бы выполнить понижение дискретизации мажоритарного класса, удаляя обучающие образцы из набора данных. Для понижения дискретизации с применением функции `resample` мы просто меняем местами метки классов 1 и 0 в предыдущем коде.



### Генерация новых обучающих данных для решения проблемы с дисбалансом классов

На заметку!

Еще одним приемом решения проблемы с дисбалансом классов является генерация искусственных обучающих образцов, рассмотрение которой выходит за рамки тематики настоящей книги. Пожалуй, наиболее широко используемым алгоритмом генерации искусственных обучающих образцов следует считать прием SMOTE (Synthetic Minority Over-sampling Technique — искусственное увеличение экземпляров миноритарного класса); более подробные сведения о нем доступны в исследовательской статье, которую опубликовал Найтеш Чаула и другие: “SMOTE: Synthetic Minority Over-sampling Technique” (SMOTE: искусственное увеличение экземпляров миноритарного класса), *Journal of Artificial Intelligence Research*, 16: с. 321–357 (2002 г.). Крайне рекомендуем также ознакомиться с библиотекой `imbalanced-learn` для Python, которая целиком ориентирована на несбалансированные наборы данных и включает реализацию SMOTE. Дополнительную информацию о библиотеке `imbalanced-learn` можно найти по ссылке <https://github.com/scikit-learn-contrib/imbalanced-learn>.

## Резюме

В начале главы мы обсуждали, как связывать различные приемы трансформации и классификаторы в удобные конвейеры, которые помогают обучать и оценивать модели МО более рационально. Мы применяли эти конвейеры для выполнения перекрестной проверки по  $k$  блокам — одного из важнейших приемов отбора и оценки модели. Используя перекрестную проверку по  $k$  блокам, мы строили кривые обучения и кривые проверки, чтобы диагностировать распространенные проблемы алгоритмов обучения, такие как переобучение и недообучение.

С помощью решетчатого поиска мы производили дальнейшую настройку модели. Затем мы применяли матрицу неточностей и разнообразные метрики эффективности, чтобы оценить и оптимизировать эффективность модели для специфических задач. В завершение главы мы обсудили различные методы для решения проблемы с дисбалансом классов, которая часто встречается во многих реальных приложениях. Теперь вы должны хорошо владеть необходимыми приемами, чтобы успешно строить модели МО с учителем, предназначенные для классификации.

В следующей главе мы взглянем на ансамблевые методы — методы, которые позволяют комбинировать множество моделей и алгоритмов классификации, чтобы еще больше повысить эффективность прогнозирования системы МО.



# ОБЪЕДИНЕНИЕ РАЗНЫХ МОДЕЛЕЙ ДЛЯ АНСАМБЛЕВОГО ОБУЧЕНИЯ

В предыдущей главе наше внимание было сосредоточено на практическом опыте настройки и оценки моделей для классификации. Опираясь на описанные там приемы, в этой главе мы исследуем различные методы построения набора классификаторов, часто способного обладать лучшей эффективностью прогнозирования, чем любой индивидуальный член набора.

В главе будут раскрыты следующие темы:

- выработка прогнозов на основе большинства голосов;
- использование *бэггинга* (*bagging*) для сокращения степени переобучения путем выборки случайных комбинаций из обучающего набора с повторением;
- применение *бустинга* (*boosting*) для построения мощных моделей из слабых учеников, которые учатся на своих ошибках.

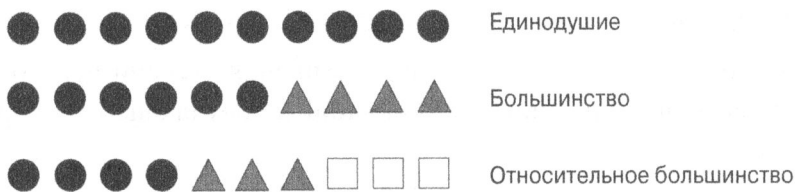
## Обучение с помощью ансамблей

Цель *ансамблевых методов* — объединить различные классификаторы в метаклассификатор, который обладает лучшей эффективностью обобщения, чем каждый индивидуальный классификатор сам по себе. Например, если предположить, что мы собрали прогнозы от 10 экспертов, то ансамблевые



методы позволили бы стратегически скомбинировать эти прогнозы для получения прогноза, более точного и надежного, нежели прогнозы каждого эксперта по отдельности. Как будет показано позже в главе, существует несколько подходов к созданию ансамбля классификаторов. В текущем разделе мы дадим общее представление о работе ансамблей и объясним, почему они обычно получают признание за обеспечение хорошей эффективности обобщения.

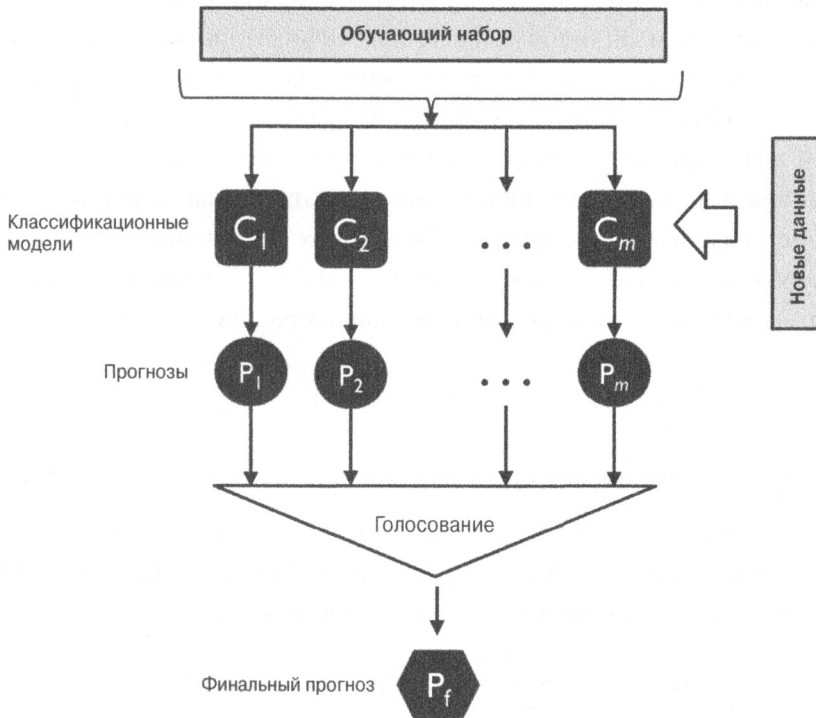
В настоящей главе мы сосредоточимся на самых популярных ансамблевых методах, которые используют принцип *мажоритарного голосования* (*majority voting*), или голосования большинством голосов. Мажоритарное голосование означает, что мы просто выбираем метку класса, которая была спрогнозирована большинством классификаторов, т.е. получила свыше 50% голосов. Строго говоря, термин *большинство голосов* относится только к конфигурациям с двоичными классами. Тем не менее, принцип мажоритарного голосования легко обобщить на многоклассовые конфигурации, в итоге получив так называемое *голосование относительным большинством голосов* (*plurality voting*). В таком случае мы выбираем метку класса, которая получила наибольшее число голосов (*моду (mode)*). На рис. 7.1 иллюстрируется концепция голосования большинством и относительным большинством голосов для ансамбля из 10 классификаторов, где каждый уникальный символ (треугольник, квадрат и круг) представляет уникальную метку класса.



**Рис. 7.1.** Концепция голосования большинством и относительным большинством голосов для ансамбля из 10 классификаторов

Мы начинаем с применения обучающего набора для обучения  $m$  разных классификаторов ( $C_1, \dots, C_m$ ). В зависимости от приема ансамбль может быть построен из разных алгоритмов классификации, скажем, деревьев принятия решений, методов опорных векторов, классификаторов на основе логистической регрессии и т.д. В качестве альтернативы мы также можем использовать один и тот же базовый алгоритм классификации, выполняя

подгонку к разным поднаборам обучающего набора. Известным примером такого подхода может служить алгоритм на базе случайного леса, который объединяет различные классификаторы, основанные на деревьях принятия решений. На рис. 7.2 демонстрируется концепция общего ансамблевого подхода, применяющего мажоритарное голосование.



**Рис. 7.2.** Концепция общего ансамблевого подхода, использующего мажоритарное голосование

Чтобы спрогнозировать метку класса посредством простого голосования большинством или относительным большинством голосов, мы объединяем метки классов, спрогнозированные каждым индивидуальным классификатором  $C_j$ , и выбираем метку класса  $\hat{y}$ , которая получила наибольшее количество голосов:

$$\hat{y} = \text{мода} \{C_1(x), C_2(x), \dots, C_m(x)\}$$

(В статистике мода представляет собой самое частое событие или результат в наборе. Скажем, мода  $\{1, 2, 1, 1, 2, 4, 5, 4\} = 1$ .)

Например, в задаче двоичной классификации, где класс 1 равен  $-1$ , а класс 2 равен  $+1$ , мы можем записать мажоритарный прогноз следующим образом:

$$C(\mathbf{x}) = \text{знак} \left[ \sum_j^m C_j(\mathbf{x}) \right] = \begin{cases} 1, & \text{если } \sum_i C_j(\mathbf{x}) \geq 0 \\ -1 & \text{в противном случае} \end{cases}$$

В целях иллюстрации причин, по которым ансамблевые методы способны работать лучше, чем индивидуальные классификаторы поодиночке, давайте применим простые принципы комбинаторики. В приведенном ниже примере мы делаем допущение о том, что все  $n$  базовых классификаторов для задачи двоичной классификации имеют одинаковые частоты ошибок  $\varepsilon$ . Кроме того, мы предполагаем, что классификаторы являются независимыми и частоты ошибок не связаны друг с другом. При таких допущениях мы можем выразить вероятность ошибки ансамбля базовых классификаторов просто как функцию вероятностной меры биномиального распределения:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1 - 0.25)^{n-k} = \varepsilon_{\text{ансамбля}}$$

Здесь  $\binom{n}{k}$  — биномиальный коэффициент из  $n$  по  $k$ . Другими словами, мы вычисляем вероятность того, что прогноз ансамбля неправильный. Теперь взглянем на более конкретный пример с 11 базовыми классификаторами ( $n = 11$ ), где каждый классификатор имеет частоту ошибок 0.25 ( $\varepsilon = 0.25$ ):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1 - \varepsilon)^{11-k} = 0.034$$



### Биномиальный коэффициент

Биномиальный коэффициент имеет отношение к количеству способов, которыми мы можем выбирать поднаборы  $k$  неупорядоченных элементов из набора с размером  $n$ ; соответственно его часто называют “из  $n$  по  $k$ ”. Поскольку порядок не имеет значения, на биномиальный коэффициент также иногда ссылаются как на *сочетание* или *комбинаторное число*, и в сокращенной форме он может быть записан следующим образом:

$$\frac{n!}{(n-k)!k!}$$

Здесь символом  $!$  обозначается факториал, например,  $3! = 3 \cdot 2 \cdot 1 = 6$ .

Несложно заметить, что частота ошибок ансамбля (0.034) гораздо меньше частоты ошибок каждого индивидуального классификатора (0.25), если удовлетворены все допущения. Обратите внимание, что в этой упрощенной иллюстрации разделение  $n$  классификаторов ровно пополам трактуется как ошибка, несмотря на то, что подобное происходит только в течение половины времени. Для сравнения такого идеалистического ансамблевого классификатора и базового классификатора с некоторым диапазоном частот базовых ошибок мы реализуем функцию вероятностной меры на Python:

```
>>> from scipy.special import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.))
...     probs = [comb(n_classifier, k) *
...               error**k *
...               (1-error)**(n_classifier - k)
...               for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.03432750701904297
```

После реализации функции `ensemble_error` мы можем подсчитать частоты ошибок ансамбля для диапазона частот базовых ошибок от 0.0 до 1.0, чтобы визуализировать связь между ансамблевыми и базовыми ошибками в виде линейного графика:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> plt.plot(error_range, ens_errors,
...          label='Ансамблевая ошибка',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Базовая ошибка',
...          linewidth=2)
>>> plt.xlabel('Базовая ошибка')
>>> plt.ylabel('Базовая/ансамблевая ошибка')
>>> plt.legend(loc='upper left')
>>> plt.grid(alpha=0.5)
>>> plt.show()
```

Как видно на результирующем графике (рис. 7.3), характеристика вероятности ошибки, допускаемой ансамблем, всегда лучше такой характеристики индивидуального базового классификатора до тех пор, пока базовые классификаторы работают лучше, чем случайное угадывание ( $\varepsilon < 0.5$ ). Обратите внимание, что ось  $y$  изображает базовую ошибку (пунктирная линия), а также ансамблевую ошибку (сплошная линия).

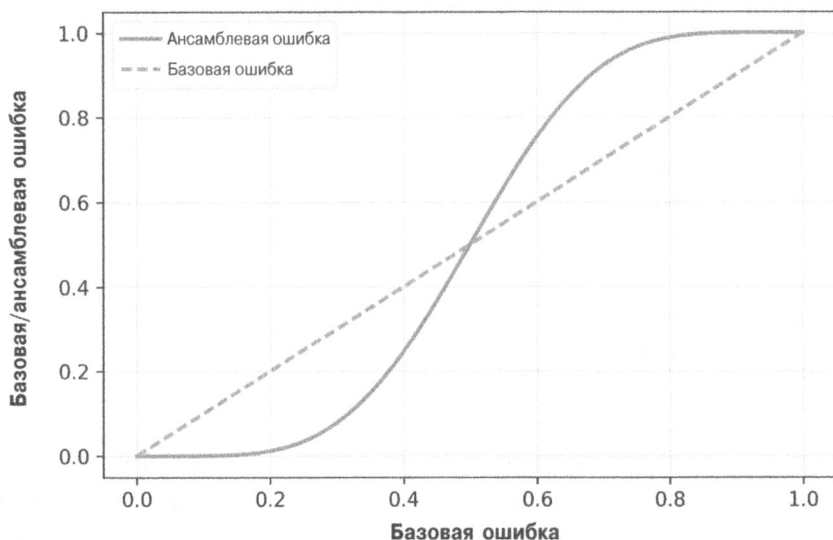


Рис. 7.3. Связь между ансамблевыми и базовыми ошибками

## Объединение классификаторов с помощью мажоритарного голосования

После краткого введения в ансамблевое обучение, предложенного в предыдущем разделе, давайте приступим к упражнению и реализуем простой ансамблевый классификатор с мажоритарным голосованием на Python.



### Голосование относительным большинством голосов

Хотя обсуждаемый в этом разделе алгоритм мажоритарного голосования также обобщается на многоклассовые конфигурации через голосование относительным большинством голосов, для простоты мы будем использовать термин “мажоритарное голосование”, как часто поступают в литературе по МО.

## Реализация простого классификатора с мажоритарным голосованием

Алгоритм, который мы собираемся реализовать в текущем разделе, позволит объединять различные алгоритмы классификации, связанные индивидуальными коэффициентами доверия в форме весов. Наша цель — построить более сильный метаклассификатор, который компенсирует слабость отдельных классификаторов на конкретном наборе данных. В математической форме мы можем записать взвешенное большинство голосов следующим образом:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i)$$

Здесь  $w_j$  — вес, ассоциированный с базовым классификатором  $C_j$ ,  $\hat{y}$  — метка класса, спрогнозированная ансамблем,  $\chi_A$  (греческая буква “хи”) — характеристическая или индикаторная функция, которая возвращает 1, если спрогнозированный класс  $j$ -того классификатора соответствует  $i$  ( $C_j(\mathbf{x}) = i$ ). Для равных весов уравнение можно упростить, записав его так:

$$\hat{y} = \text{мода}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}$$

Для лучшего понимания концепции *взвешивания* мы рассмотрим более конкретный пример. Пусть имеется ансамбль из трех базовых классификаторов,  $C_j (j \in \{0,1\})$ , и мы хотим спрогнозировать метку класса  $C_j(\mathbf{x}) \in \{0,1\}$  заданного образца  $\mathbf{x}$ . Два из трех базовых классификаторов прогнозируют метку класса 0 и один,  $C_3$ , вырабатывает прогноз, что образец принадлежит классу 1. Если назначить прогнозам всех базовых классификаторов одинаковые веса, тогда большинство голосов даст прогноз, что образец относится к классу 0:

$$C_1(\mathbf{x}) \rightarrow 0, C_2(\mathbf{x}) \rightarrow 0, C_3(\mathbf{x}) \rightarrow 1$$

$$\hat{y} = \text{мода}\{0, 0, 1\} = 0$$

Теперь давайте назначим  $C_3$  вес 0.6, а  $C_1$  и  $C_2$  взвесим с коэффициентом 0.2:

$$\begin{aligned} \hat{y} &= \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i) = \\ &= \arg \max_i [0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1] = 1 \end{aligned}$$

Поскольку  $3 \times 0.2 = 0.6$ , мы можем сказать, что прогноз, сделанный  $C_3$ , имеет в три раза больший вес, чем прогнозы от  $C_1$  или  $C_2$ , поэтому мы можем записать так:

$$\hat{y} = \text{мода } \{0, 0, 1, 1, 1\} = 1$$

Чтобы представить концепцию взвешенного большинства голосов в коде Python, мы можем воспользоваться удобными функциями `argmax` и `bincount` из библиотеки NumPy:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...                        weights=[0.2, 0.2, 0.6]))
1
```

Как мы помним из обсуждения логистической регрессии в главе 3, некоторые классификаторы в `scikit-learn` также способны возвращать вероятность спрогнозированной метки класса посредством метода `predict_proba`. Применение вероятностей спрогнозированных классов вместо меток классов для мажоритарного голосования может быть практичным, если наш ансамбль хорошо откалиброван. Модифицированную версию большинства голосов для прогнозирования меток классов из вероятностей можно записать следующим образом:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j p_{ij}$$

Здесь  $p_{ij}$  — спрогнозированная вероятность  $j$ -того классификатора для метки класса  $i$ .

Чтобы продолжить предыдущий пример, давайте предположим, что мы имеем задачу двоичной классификации с метками классов  $i \in \{0, 1\}$  и ансамблем из трех классификаторов  $C_j$  ( $j \in \{1, 2, 3\}$ ). Пусть классификаторы  $C_j$  возвращают такие вероятности членства в классах для отдельного образца  $x$ :

$$C_1(x) \rightarrow [0.9, 0.1], C_2(x) \rightarrow [0.8, 0.2], C_3(x) \rightarrow [0.4, 0.6]$$

Используя те же веса, что и ранее (0.2, 0.2 и 0.6), мы можем рассчитать вероятности индивидуальных классов следующим образом:

$$p(i_0 | \mathbf{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 | \mathbf{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

$$\hat{y} = \arg \max_i [p(i_0 | \mathbf{x}), p(i_1 | \mathbf{x})] = 0$$

Для реализации взвешенного большинства голосов на основе вероятностей классов мы снова можем задействовать функции `numpy.average` и `np.argmax` библиотеки NumPy:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

Теперь соберем все вместе и реализуем класс `MajorityVoteClassifier` на Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator

class MajorityVoteClassifier(BaseEstimator,
                           ClassifierMixin):
    """ Ансамблевый классификатор с мажоритарным голосованием

    Параметры
    -----
    classifiers : подобен массиву, форма = [n_classifiers]
        Различные классификаторы для ансамбля.

    vote : str, {'classlabel', 'probability'}
        По умолчанию: 'classlabel'
```



В случае 'classlabel' прогноз основывается на результате `argmax` меток классов. В случае 'probability' для прогнозирования метки класса применяется `argmax` суммы вероятностей (рекомендуется для откалиброванных классификаторов).

```
weights : подобен массиву, форма = [n_classifiers]
Необязательно, по умолчанию: None
Если предоставляется список значений int или float,
тогда классификаторам назначаются веса по важности;
в случае weights=None используются равномерные веса.

"""
def __init__(self, classifiers,
              vote='classlabel', weights=None):

    self.classifiers = classifiers
    self.named_classifiers = {key: value for
                              key, value in
                              _name_estimators(classifiers)}

    self.vote = vote
    self.weights = weights

def fit(self, X, y):
    """ Подгоняет классификаторы.

    Параметры
    -----
    X : {подобен массиву, разреженная матрица},
        форма = [n_examples, n_features]
        Матрица обучающих образцов.

    y : подобен массиву, форма = [n_examples]
        Вектор целевых меток классов.

    Возвращает
    -----
    self : object

    """
    if self.vote not in ('probability', 'classlabel'):
        raise ValueError("vote должно быть 'probability'"
                          "или 'classlabel'; получено (vote=%r)"
                          % self.vote)

    if self.weights and
       len(self.weights) != len(self.classifiers):
```

```

    raise ValueError("Количество классификаторов и
количество весов"
                    "должны быть равны; получено %d весов,"
                    "%d классификаторов"
                    % (len(self.weights),
                       len(self.classifiers)))
# Использовать LabelEncoder для гарантии того,
# что метки классов начинаются с 0;
# это важно при вызове np.argmax в self.predict.
self.lablenc_ = LabelEncoder()
self.lablenc_.fit(y)
self.classes_ = self.lablenc_.classes_
self.classifiers_ = []
for clf in self.classifiers:
    fitted_clf = clone(clf).fit(X,
                               self.lablenc_.transform(y))
    self.classifiers_.append(fitted_clf)
return self

```

В коде присутствует много комментариев, поясняющих его отдельные части. Однако прежде чем реализовывать остальные методы, давайте ненадолго отвлечемся и обсудим части кода, которые поначалу могут выглядеть сбивающими с толку. Мы применяли родительские классы `BaseEstimator` и `ClassifierMixin`, чтобы *даром* получить определенную функциональность, включая методы `get_params` и `set_params` для установки и возвращения параметров классификатора, а также метод `score` для подсчета правильности прогнозирования.

Далее мы добавим метод `predict` для прогнозирования метки класса через мажоритарное голосование, если новый объект `MajorityVoteClassifier` инициализируется с параметром `vote='classlabel'`. В качестве альтернативы мы будем иметь возможность инициализации ансамблевого классификатора с параметром `vote='probability'`, чтобы прогнозировать метку класса на основе вероятностей членства в классах. Кроме того, мы также добавим метод `predict_proba` для возвращения усредненных вероятностей, которые полезны при вычислении показателя ROC AUC:

```

def predict(self, X):
    """ Прогнозирует метки классов для X.

    Параметры
    -----

```

*X* : {подобен массиву, разреженная матрица},  
форма = [*n\_examples*, *n\_features*]  
Матрица обучающих образцов.

Возвращает

*maj\_vote* : подобен массиву, форма = [*n\_examples*]  
Спрогнозированные метки классов.

"""

```
if self.vote == 'probability':
    maj_vote = np.argmax(self.predict_proba(X),
                        axis=1)
else:    # vote='classlabel'

    # Получить результаты из вызовов clf.predict
    predictions = np.asarray([clf.predict(X)
                              for clf in
                              self.classifiers_]).T

    maj_vote = np.apply_along_axis(
        lambda x:
            np.argmax(np.bincount(x,
                                   weights=self.weights)),
        axis=1,
        arr=predictions)
maj_vote = self.lablenc_.inverse_transform(maj_vote)
return maj_vote
```

**def predict\_proba(self, X):**

""" Прогнозирует вероятности классов для *X*.

Параметры

*X* : {подобен массиву, разреженная матрица},  
форма = [*n\_examples*, *n\_features*]  
Обучающие векторы, где *n\_examples* - количество  
образцов, а *n\_features* - количество признаков.

Возвращает

*avg\_proba* : подобен массиву,  
форма = [*n\_examples*, *n\_classes*]  
Взвешенная усредненная вероятность  
для каждого класса на образце.

"""

```

        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas,
                               axis=0, weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        """ Получает имена параметров классификатора для
        решетчатого поиска """
        if not deep:
            return super(MajorityVoteClassifier,
                          self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in self.named_classifiers.items():
                for key, value in step.get_params(
                    deep=True).items():
                    out['%s__%s' % (name, key)] = value
            return out

```

Мы также определили собственную модифицированную версию метода `get_params` с целью использования функции `_name_estimators` для доступа к параметрам индивидуальных классификаторов в ансамбле. Сначала подход может выглядеть несколько замысловатым, но он обретет смысл, когда мы будем применять решетчатый поиск для настройки гиперпараметров в более поздних разделах.



На  
заметку!

### Класс `VotingClassifier` в `scikit-learn`

Хотя реализация `MajorityVoteClassifier` очень удобна для демонстрационных целей, мы реализовали более сложную версию данного классификатора с мажоритарным голосованием в библиотеке `scikit-learn`, основываясь на реализации из первого издания книги. Этот ансамблевый классификатор доступен в виде класса `sklearn.ensemble.VotingClassifier` в `scikit-learn` версии 0.17 и новее.

## Использование принципа мажоритарного голосования для выработки прогнозов

Наступило время задействовать класс `MajorityVoteClassifier`, реализованный в предыдущем разделе. Но для начала мы подготовим набор данных, на котором можно будет испытывать `MajorityVoteClassifier`.

Так как мы уже видели приемы для загрузки наборов данных из файлов CSV, мы примем сокращенный путь и загрузим набор данных Iris из модуля `datasets` библиотеки `scikit-learn`. К тому же мы выберем только два признака, *ширину чашелистика* (*sepal width*) и *длину лепестка* (*petal length*), чтобы сделать задачу классификации более подходящей для иллюстративных целей. Несмотря на то что `MajorityVoteClassifier` обобщается на многоклассовые задачи, мы будем классифицировать образцы цветков только из классов `Iris-versicolor` и `Iris-virginica`, а позже вычислим для них показатель ROC AUC. Ниже представлен соответствующий код:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
```



На заметку!

### Вероятности принадлежности классам в деревьях принятия решений

Обратите внимание, что для вычисления показателя ROC AUC в библиотеке `scikit-learn` используется метод `predict_proba` (если он применим). В главе 3 было показано, как вычисляются вероятности классов в логистических регрессионных моделях. В деревьях принятия решений вероятности вычисляются из частотного вектора, который создается для каждого узла во время обучения. Этот вектор накапливает значения частоты каждой метки класса, вычисленные из распределения меток классов в данном узле. Затем частоты нормализуются, чтобы давать в сумме 1. Подобным же образом группируются метки классов  $k$  ближайших соседей, чтобы возвратить нормализованные частоты меток классов в алгоритме  $k$  ближайших соседей (KNN). Хотя нормализованные вероятности, возвращаемые классификаторами на основе дерева принятия решений и метода  $k$  ближайших соседей, могут выглядеть похожими на вероятности, которые дает логистическая регрессионная модель, мы должны осознавать, что в действительности они не получаются из функций вероятностной меры.

Затем мы разделяем образцы Iris на 50% обучающих и 50% испытательных данных:

```
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.5,
...                       random_state=1,
...                       stratify=y)
```

Используя обучающий набор данных, мы обучим три разных классификатора:

- классификатор на основе логистической регрессии;
- классификатор на основе дерева принятия решений;
- классификатор на основе метода  $k$  ближайших соседей.

Перед объединением базовых классификаторов в ансамблевый классификатор мы оцениваем эффективность каждого классификатора посредством перекрестной проверки по 10 блокам на обучающем наборе:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           solver='lbfgs',
...                           random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipel = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Логистическая регрессия',
...               'Дерево принятия решений', 'KNN']
```

```
>>> print('Перекрестная проверка по 10 блокам:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
```

Получаемый вывод показывает, что эффективность прогнозирования индивидуальных классификаторов почти равна:

Перекрестная проверка по 10 блокам:

```
ROC AUC: 0.92 (+/- 0.15) [Логистическая регрессия]
ROC AUC: 0.87 (+/- 0.18) [Дерево принятия решений]
ROC AUC: 0.85 (+/- 0.13) [KNN]
```

Вас может интересовать причина, по которой мы обучаем классификаторы на основе логистической регрессии и метода  $k$  ближайших соседей в качестве части конвейера. Как обсуждалось в главе 3, дело в том, что в отличие от деревьев принятия решений алгоритмы логистической регрессии и  $k$  ближайших соседей (применяющие метрику евклидова расстояния) не являются масштабно-инвариантными. Несмотря на то что все признаки в наборе данных измерены с тем же самым масштабом (сантиметры), работа со стандартизированными признаками расценивается как хорошая привычка.

А теперь перейдем к более захватывающей части и объединим индивидуальные классификаторы по правилу мажоритарного голосования в классе `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...     classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Мажоритарное голосование']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
```

```

...     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...           % (scores.mean(), scores.std(), label))
ROC AUC: 0.92 (+/- 0.15) [Логистическая регрессия]
ROC AUC: 0.87 (+/- 0.18) [Дерево принятия решений]
ROC AUC: 0.85 (+/- 0.13) [KNN]
ROC AUC: 0.98 (+/- 0.05) [Мажоритарное голосование]

```

Глядя на результаты перекрестной проверки по 10 блокам, несложно заметить, что эффективность MajorityVotingClassifier повысилась по сравнению с эффективностью индивидуальных классификаторов.

## Оценка и настройка ансамблевого классификатора

В этом разделе мы построим кривые ROC для испытательного набора, чтобы проверить, хорошо ли MajorityVoteClassifier обобщается на не встречавшиеся ранее данные. Как вы помните, испытательный набор не должен использоваться при отборе модели; он предназначен для выдачи несмещенной оценки эффективности обобщения, которую обеспечивает классификационная система:

```

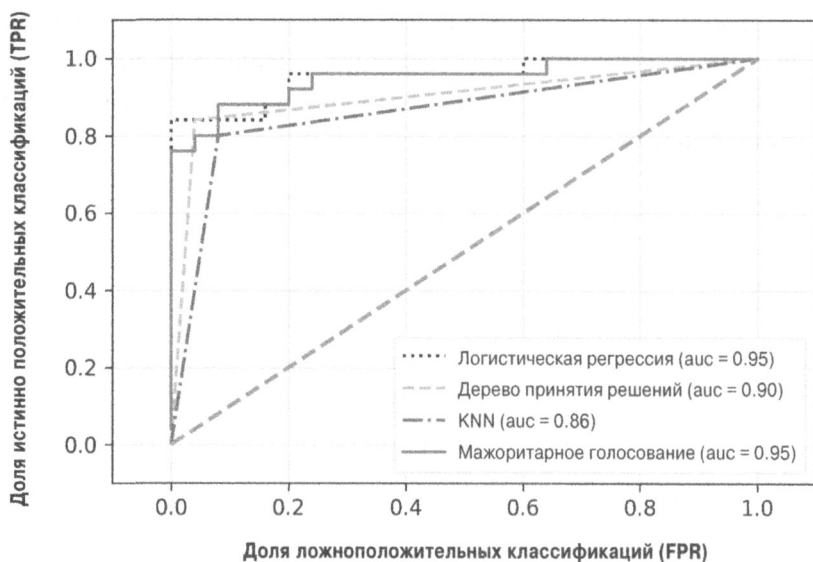
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyle = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...     in zip(all_clf, clf_labels, colors, linestyle):
...     # предполагается, что меткой положительного класса является 1
...     y_pred = clf.fit(X_train,
...                     y_train).predict_proba(X_test)[: , 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                     y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...              linestyle=ls,
...              label='%s (auc = %0.2f)' % (label, roc_auc))
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])

```



```
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid(alpha=0.5)
>>> plt.xlabel('Доля ложноположительных классификаций (FPR)')
>>> plt.ylabel('Доля истинно положительных классификаций (TPR)')
>>> plt.show()
```

На результирующих кривых ROC (рис. 7.4) легко заметить, что ансамблевый классификатор также хорошо работает на испытательном наборе (ROC AUC = 0.95). Тем не менее, мы видим, что классификатор на основе логистической регрессии на том же самом наборе данных работает в равной степени хорошо; вероятной причиной может быть высокая дисперсия (в рассматриваемом случае чувствительность к тому, как разделяется набор данных), учитывая небольшой размер набора данных.



**Рис. 7.4.** Кривые ROC для испытательного набора

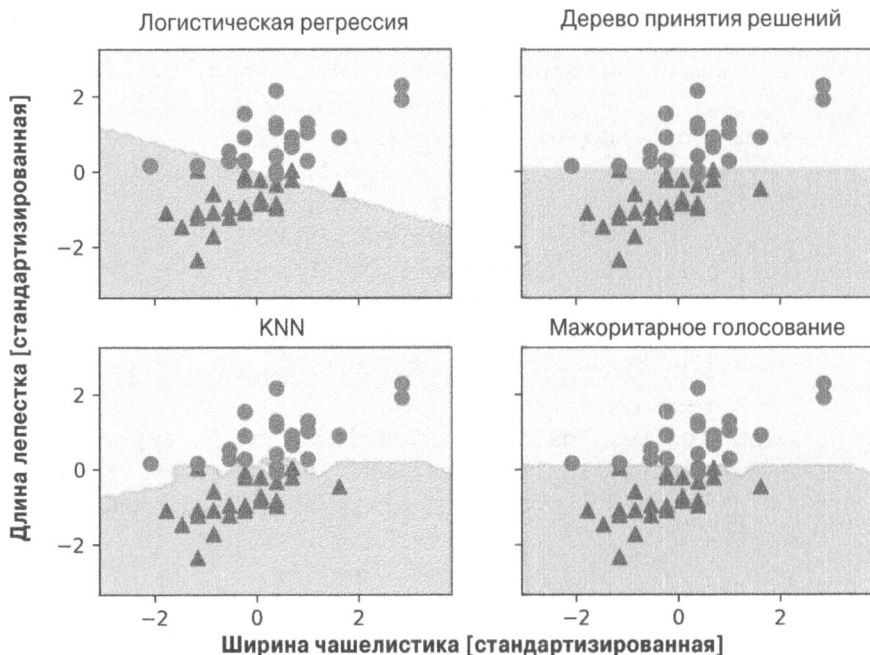
Так как для примеров классификации мы выбрали только два признака, было бы интересно посмотреть, на что в действительности похожа область решений ансамблевого классификатора.

Хотя стандартизировать обучающие признаки перед подгонкой модели вовсе не обязательно, поскольку конвейеры логистической регрессии и метода  $k$  ближайших соседей позаботятся об этом автоматически, мы стандарти-

зируем обучающий набор в визуальных целях, чтобы области решений у дерева принятия решений имели тот же самый масштаб. Ниже приведен код:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>>
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                       np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                          sharex='col',
...                          sharey='row',
...                          figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                          all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                     X_train_std[y_train==0, 1],
...                                     c='blue',
...                                     marker='^',
...                                     s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                     X_train_std[y_train==1, 1],
...                                     c='green',
...                                     marker='o',
...                                     s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -5.,
...           s='Ширина чашелистика [стандартизированная]',
...           ha='center', va='center', fontsize=12)
>>> plt.text(-12.5, 4.5,
...           s='Длина лепестка [стандартизированная]',
...           ha='center', va='center',
...           fontsize=12, rotation=90)
>>> plt.show()
```

Интересно, но вполне ожидаемо области решений ансамблевого классификатора выглядят как гибрид областей решений индивидуальных классификаторов (рис. 7.5). На первый взгляд граница решений классификатора с мажоритарным голосованием очень похожа на пенек дерева принятия решений, который перпендикулярен оси  $x$  для ширины чашелистика  $\geq 1$ . Однако мы также отмечаем примесь нелинейности со стороны классификатора на основе метода  $k$  ближайших соседей.



**Рис. 7.5.** Области решений ансамблевого классификатора и индивидуальных классификаторов

Прежде чем настраивать параметры индивидуальных классификаторов для ансамблевой классификации, давайте вызовем метод `get_params`, чтобы получить базовое представление о том, как обращаться к отдельным параметрам внутри объекта `GridSearchCV`:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier':
  DecisionTreeClassifier(class_weight=None, criterion='entropy',
                        max_depth=1, max_features=None,
                        max_leaf_nodes=None, min_samples_leaf=1,
```

```

        min_samples_split=2,
        min_weight_fraction_leaf=0.0,
        random_state=0, splitter='best'),
'decisiontreeclassifier__class_weight': None,
'decisiontreeclassifier__criterion': 'entropy',
[...]
'decisiontreeclassifier__random_state': 0,
'decisiontreeclassifier__splitter': 'best',
'pipeline-1':
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                with_std=True)),
                ('clf', LogisticRegression(C=0.001,
                                class_weight=None,
                                dual=False,
                                fit_intercept=True,
                                intercept_scaling=1,
                                max_iter=100,
                                multi_class='ovr',
                                penalty='l2',
                                random_state=0,
                                solver='liblinear',
                                tol=0.0001,
                                verbose=0))]),

'pipeline-1__clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
                    fit_intercept=True, intercept_scaling=1,
                    max_iter=100, multi_class='ovr',
                    penalty='l2', random_state=0,
                    solver='liblinear', tol=0.0001, verbose=0),

'pipeline-1__clf__C': 0.001,
'pipeline-1__clf__class_weight': None,
'pipeline-1__clf__dual': False,
[...]
'pipeline-1__sc__with_std': True,
'pipeline-2':
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                with_std=True)),
                ('clf', KNeighborsClassifier(algorithm='auto',
                                leaf_size=30,
                                metric='minkowski',
                                metric_params=None,
                                n_neighbors=1,

```

```

        p=2,
        weights='uniform')))),
'pipeline-2__clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                     metric='minkowski', metric_params=None,
                     n_neighbors=1, p=2, weights='uniform'),
'pipeline-2__clf__algorithm': 'auto',
[...]
'pipeline-2__sc__with_std': True}

```

Основываясь на значениях, которые возвратил метод `get_params`, мы знаем, как получать доступ к атрибутам индивидуальных классификаторов. Давайте теперь в демонстрационных целях с помощью решетчатого поиска настроим обратный параметр регуляризации `C` для классификатора на базе логистической регрессии и глубину дерева принятия решений:

```

>>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...           'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                      param_grid=params,
...                      cv=10,
...                      iid=False,
...                      scoring='roc_auc')
>>> grid.fit(X_train, y_train)

```

После завершения решетчатого поиска мы можем вывести различные комбинации значений гиперпараметров и средние показатели ROC AUC, вычисленные посредством перекрестной проверки по 10 блокам:

```

>>> for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...     print("%0.3f +/- %0.2f %r"
...           % (grid.cv_results_['mean_test_score'][r],
...              grid.cv_results_['std_test_score'][r] / 2.0,
...              grid.cv_results_['params'][r]))
0.944 +/- 0.07 {'decisiontreeclassifier__max_depth': 1,
               'pipeline-1__clf__C': 0.001}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 1,
               - 'pipeline-1__clf__C': 0.1}
0.978 +/- 0.03 {'decisiontreeclassifier__max_depth': 1,
               'pipeline-1__clf__C': 100.0}
0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 2,
               'pipeline-1__clf__C': 0.001}

```

```

0.956 +/- 0.07 {'decisiontreeclassifier__max_depth': 2,
               'pipeline-1__clf__C': 0.1}
0.978 +/- 0.03 {'decisiontreeclassifier__max_depth': 2,
               'pipeline-1__clf__C': 100.0}

>>> print('Наилучшие параметры: %s' % grid.best_params_)
Наилучшие параметры: {'decisiontreeclassifier__max_depth': 1,
                      'pipeline-1__clf__C': 0.001}

>>> print('Правильность: %.2f' % grid.best_score_)
Правильность: 0.98

```

Как видите, мы получили лучшие результаты перекрестной проверки, когда выбрали наименьшую силу регуляризации ( $C=0.001$ ), тогда как глубина дерева, кажется, вообще не влияет на эффективность, наводя на мысль о том, что пенек решения достаточен для разделения данных. Памятуя о том, что применение испытательного набора данных для оценки модели более одного раза — скверная практика, мы не собираемся в этом разделе оценивать эффективность обобщения модели с настроенными гиперпараметрами. Мы без промедления перейдем к рассмотрению альтернативного подхода для ансамблевого обучения — *бэггингу*.



На  
заметку!

### Построение ансамблей с использованием стекинга

Не следует путать подход мажоритарного голосования, реализованный в настоящем разделе, со *стекингом* (*stacking*). Алгоритм стекинга можно понимать как двухслойный ансамбль, в котором первый слой состоит из индивидуальных классификаторов, передающих свои прогнозы второму слою, где еще один классификатор (обычно основанный на логистической регрессии) подгоняется к прогнозам классификаторов первого слоя, чтобы вырабатывать финальные прогнозы. Алгоритм стекинга подробно описан Дэвидом Вольпертом в статье “Stacked generalization” (Многослойное обобщение), *Neural Networks*, 5(2): с. 241–259 (1992 г.). К сожалению, на момент написания книги описанный алгоритм не был реализован в *scikit-learn*, но работа над ним велась. Тем временем вы можете найти совместимые с библиотекой *scikit-learn* реализации стекинга по ссылкам [http://rasbt.github.io/mlxtend/user\\_guide/classifier/StackingClassifier/](http://rasbt.github.io/mlxtend/user_guide/classifier/StackingClassifier/) и [http://rasbt.github.io/mlxtend/user\\_guide/classifier/StackingCVClassifier/](http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/).

## Бэггинг — построение ансамбля классификаторов из бутстрэп-образцов

Бэггинг представляет собой прием ансамблевого обучения, тесно связанный с классификатором `MajorityVoteClassifier`, который мы реализовали в предыдущем разделе. Тем не менее, вместо использования для подгонки индивидуальных классификаторов в ансамбле того же самого обучающего набора мы производим выборку бутстрэп-образцов (случайных образцов с возвращением) из первоначального обучающего набора, что и является причиной, по которой бэггинг называют также *бутстрэп-агрегированием*.

Концепция бэггинга подытожена на рис. 7.6.

В последующих подразделах мы проработаем простой пример бэггинга вручную и применим библиотеку `scikit-learn` для классификации образцов вин.

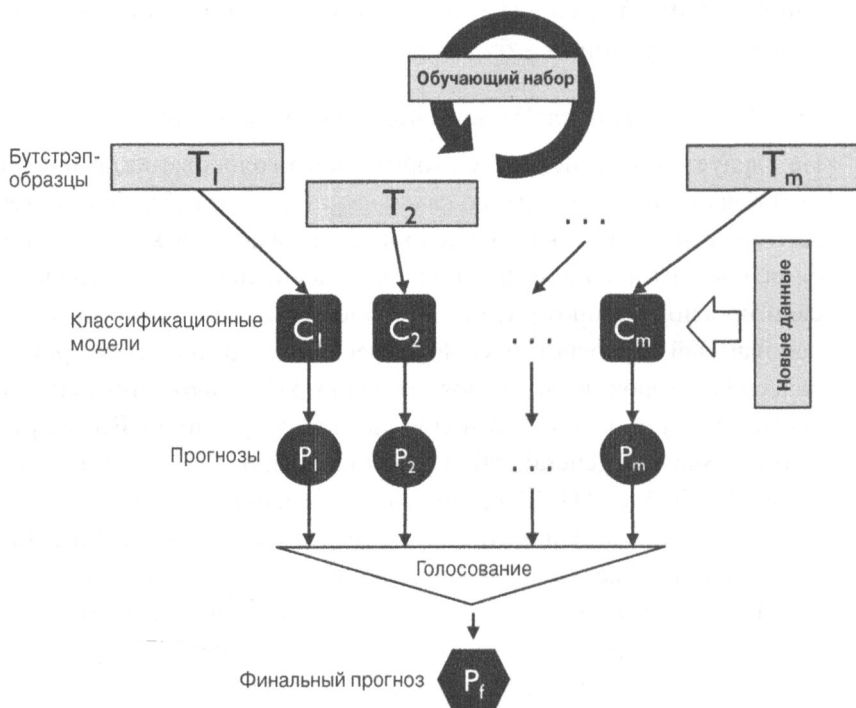


Рис. 7.6. Концепция бэггинга

## Коротко о бэггинге

Чтобы предоставить более конкретный пример, иллюстрирующий работу бутстрэп-агрегирования классификатора на основе бэггинга, рассмотрим таблицу на рис. 7.7. Здесь мы имеем семь обучающих образцов (обозначенных индексами 1–7), которые случайным образом выбираются с возвращением в каждом раунде бэггинга. Каждый бутстрэп-образец затем используется для подгонки классификатора  $C_j$ , которым чаще всего является неподрезанное дерево принятия решений.

Индексы образцов	Раунд 1 бэггинга	Раунд 2 бэггинга	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...

Рис. 7.7. Пример бэггинга

На рис. 7.7 видно, что каждый классификатор получает случайный поднабор образцов из обучающего набора. Мы обозначили такие случайные образцы, полученные через бэггинг, как “Раунд 1 бэггинга”, “Раунд 2 бэггинга” и т.д. Каждый поднабор содержит определенную долю дубликатов, а некоторые исходные образцы вообще не появляются в повторно выбранном наборе данных из-за выборки с возвращением. После того, как индивидуальные классификаторы подогнаны к бутстрэп-образцам, прогнозы объединяются с применением мажоритарного голосования.

Обратите внимание, что бэггинг находится в родстве с классификатором на основе случайного леса, который был представлен в главе 3. В сущности, случайные леса — это частный случай бэггинга, где при подгонке индиви-



дуальных деревьев принятия решений также используются случайные поднаборы признаков.



### Ансамбли моделей, использующие бэггинг

Бэггинг был впервые предложен Лео Брейманом в техническом отчете по МО за 1994 год; он также показал, что бэггинг может улучшить правильность нестабильных моделей и снизить степень переобучения. Настоятельно рекомендуем ознакомиться с его исследованием в статье “Bagging predictors” (Прогностаторы на основе бэггинга), Л. Брейман, *Machine Learning*, 24(2): с. 123–140 (1996 г.), свободно доступной в Интернете, которая позволит узнать больше подробностей о бэггинге.

## Применение бэггинга для классификации образцов в наборе данных Wine

Чтобы посмотреть на бэггинг в действии, давайте создадим более сложную задачу классификации, используя набор данных Wine, который был представлен в главе 4. Мы будем принимать во внимание только вина классов 2 и 3, к тому же выберем два признака: “Алкоголь” и “OD280/OD315 разбавленных вин”:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/
...                        'machine-learning-databases/wine/wine.data',
...                        header=None)
>>> df_wine.columns = ['Метка класса', 'Алкоголь',
...                    'Яблочная кислота', 'Зола',
...                    'Щелочность золы',
...                    'Магний', 'Всего фенолов',
...                    'Флавоноиды', 'Нефлавоноидные фенолы',
...                    'Проантоцианидины',
...                    'Интенсивность цвета', 'Оттенок',
...                    'OD280/OD315 разбавленных вин',
...                    'Пролин']
>>> # отбросить класс 1
>>> df_wine = df_wine[df_wine['Метка класса'] != 1]
>>> y = df_wine['Метка класса'].values
>>> X = df_wine[['Алкоголь',
...              'OD280/OD315 разбавленных вин']].values
```

Затем мы кодируем метки классов в двоичном формате и разбиваем набор данных на 80%-ный обучающий и 20%-ный испытательный наборы:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.2,
...                       random_state=1,
...                       stratify=y)
```



На  
заметку!

### Получение набора данных Wine

Копия набора данных Wine (и всех других наборов данных, используемых в книге) включена в состав загружаемого архива с кодом примеров для книги. Указанные копии можно задействовать при автономной работе или в случае временной недоступности <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data> на сервере UCI. Скажем, чтобы загрузить набор данных Wine из какого-то локального каталога, следующий оператор:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                  'machine-learning-databases/wine/wine.data',
                  header=None)
```

понадобится заменить таким оператором:

```
df = pd.read_csv('ваш/локальный/путь/к/wine.data',
                  header=None)
```

Алгоритм `BaggingClassifier` реализован в библиотеке `scikit-learn` и может импортироваться из подмодуля `ensemble`. Здесь мы будем применять в качестве базового классификатора неподрезанное дерево принятия решений и создадим ансамбль, включающий 500 таких деревьев, которые подгоняются на разных бутстрэп-образцах из обучающего набора данных:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          max_samples=1.0,
```

```
...             max_features=1.0,  
...             bootstrap=True,  
...             bootstrap_features=False,  
...             n_jobs=1,  
...             random_state=1)
```

Далее мы вычислим меру правильности прогноза на обучающем и испытательном наборах данных, чтобы сравнить эффективность классификатора на основе бэггинга с эффективностью одиночного неподрезанного дерева принятия решений:

```
>>> from sklearn.metrics import accuracy_score  
>>> tree = tree.fit(X_train, y_train)  
>>> y_train_pred = tree.predict(X_train)  
>>> y_test_pred = tree.predict(X_test)  
>>> tree_train = accuracy_score(y_train, y_train_pred)  
>>> tree_test = accuracy_score(y_test, y_test_pred)  
>>> print('Меры правильности дерева принятия решений при  
обучении/испытании %.3f/%.3f'  
...      % (tree_train, tree_test))  
Меры правильности дерева принятия решений при обучении/испытании  
1.000/0.833
```

Взглянув на выведенные значения мер правильности, можно сделать вывод, что неподрезанное дерево принятия решений корректно прогнозирует все метки классов для обучающих образцов; однако, существенно более низкая мера правильности при испытании указывает на высокую дисперсию (переобучение) модели:

```
>>> bag = bag.fit(X_train, y_train)  
>>> y_train_pred = bag.predict(X_train)  
>>> y_test_pred = bag.predict(X_test)  
>>> bag_train = accuracy_score(y_train, y_train_pred)  
>>> bag_test = accuracy_score(y_test, y_test_pred)  
>>> print('Меры правильности бэггинга при обучении/испытании  
...      %.3f/%.3f'  
...      % (bag_train, bag_test))  
Меры правильности бэггинга при обучении/испытании 1.000/0.917
```

Хотя меры правильности при обучении классификаторов на основе дерева принятия решений и бэггинга подобны на обучающем наборе (обе составляют 100%), мы можем заметить, что классификатор на базе бэггинга

обладает чуть лучшей эффективностью обобщения, как было оценено на испытательном наборе. Давайте сравним области решений классификаторов на основе дерева принятия решений и бэггинга:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                       np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                          sharex='col',
...                          sharey='row',
...                          figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                          [tree, bag],
...                          ['Дерево принятия решений', 'Бэггинг']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                          X_train[y_train==0, 1],
...                          c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                          X_train[y_train==1, 1],
...                          c='green', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('Алкоголь', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='OD280/OD315 разбавленных вин',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()
```

На результирующем графике (рис. 7.8) видно, что кусочно-линейная граница решений у дерева принятия решений длиной три узла в ансамбле на основе бэггинга выглядит более гладкой.

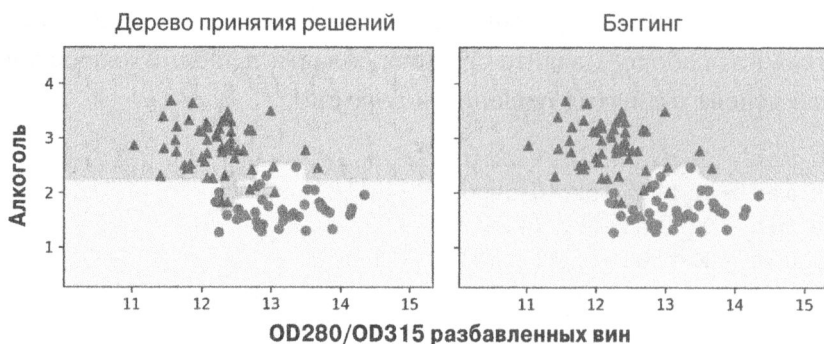


Рис. 7.8. Области решений классификаторов на основе дерева принятия решений и бэггинга

В этом разделе мы рассмотрели очень простой пример бэггинга. На практике более сложные задачи классификации и высокая размерность набора данных могут легко привести к переобучению в отдельных деревьях принятия решений, и как раз здесь алгоритм бэггинга способен по-настоящему проявить свои сильные стороны. Наконец, мы должны отметить, что алгоритм бэггинга может быть результативным подходом к снижению дисперсии модели. Тем не менее, бэггинг безрезультатен в плане сокращения смещения моделей, т.е. моделей, которые слишком просты, чтобы хорошо выявлять тенденцию в данных. Именно потому мы хотим выполнять бэггинг на ансамбле классификаторов с низким смещением, например, основанных на деревьях принятия решений.

## Использование в своих интересах слабых учеников посредством адаптивного бустинга

В последнем разделе, посвященном ансамблевым методам, мы обсудим бустинг с акцентированием особого внимания на его самой распространенной реализации — *адаптивном бустинге* (*Adaptive Boosting* — *AdaBoost*).



На заметку!

### Идентификация AdaBoost

Первоначальную идею, лежащую в основе AdaBoost, сформулировал Роберт Э. Шапир в 1990 году (“The Strength of Weak Learnability” (Достоинство слабой обучаемости), Р.Э. Шапир, *Machine Learning*, 5(2): с. 197–227 (1990 г.)). После того, как Роберт Шапир и Йоав Фройнд представили алгоритм AdaBoost в трудах 13-й Международной конференции по МО (ICML 1996), в последующие годы он

стал одним из наиболее широко применяемых ансамблевых методов (“Experiments with a New Boosting Algorithm” (Эксперименты с новым алгоритмом бустинга), Й. Фройнд, Р.Э. Шапир и др., ICML, том 96, с. 148–156 (1996 г.)). В 2003 году за свою новаторскую работу Фройнд и Шапир получили премию Геделя, которая является престижной наградой для большинства выдающихся публикаций в области компьютерных наук.

В бустинге ансамбль состоит из очень простых базовых классификаторов, часто именуемых *слабыми учениками*, которые нередко имеют лишь незначительное преимущество в эффективности над случайным угадыванием — типичным примером слабого ученика является пенек дерева принятия решения. Главная концепция, лежащая в основе бустинга, заключается в том, чтобы сконцентрироваться на обучающих образцах, которые трудно классифицировать, т.е. позволить слабым ученикам впоследствии обучиться на неправильно классифицированных обучающих образцах с целью повышения эффективности ансамбля.

В дальнейших подразделах будет представлена алгоритмическая процедура, которая положена в основу бустинга и AdaBoost. Наконец, мы будем использовать библиотеку `scikit-learn` в практическом примере классификации.

## Как работает бустинг

По контрасту с бэггингом алгоритм бустинга в своей первоначальной формулировке применяет случайные поднаборы обучающих образцов, выбранные из обучающего набора данных без возвращения; исходную процедуру бустинга можно подытожить в виде четырех основных шагов.

1. Произвести выборку случайного поднабора обучающих образцов  $d_1$  без возвращения из обучающего набора  $D$  для обучения слабого ученика  $C_1$ .
2. Произвести выборку второго случайного поднабора обучающих образцов  $d_2$  без возвращения из обучающего набора и добавить 50% образцов, которые ранее были неправильно классифицированы, для обучения слабого ученика  $C_2$ .
3. Найти в обучающем наборе  $D$  обучающие образцы  $d_3$ , по которым  $C_1$  и  $C_2$  расходятся, для обучения третьего слабого ученика  $C_3$ .
4. Объединить слабых учеников  $C_1$ ,  $C_2$  и  $C_3$  посредством мажоритарного голосования.

Как выяснил Лео Брейман (“Bias, variance, and arcing classifiers” (Смещение, дисперсия и дуговые классификаторы), Л. Брейман (1996 г.), бустинг может привести к уменьшению смещения и дисперсии в сравнении с моделями на основе бэггинга. Однако на практике алгоритмы бустинга, подобные AdaBoost, также известны своей высокой дисперсией, т.е. склонностью к переобучению обучающими данными (“An improvement of AdaBoost to avoid overfitting” (Усовершенствование AdaBoost во избежание переобучения), Г. Ретч, Т. Онода и К.Р. Мюллер, труды Международной конференции по нейронной обработке информации, CiteSeer (1998 г.)).

В отличие от описанной здесь исходной процедуры бустинга алгоритм AdaBoost для обучения слабых учеников использует полный обучающий набор, в котором обучающие образцы на каждой итерации заново взвешиваются, чтобы построить более сильный классификатор, обучающийся на ошибках предшествующих слабых учеников в ансамбле.

Прежде чем погрузиться в конкретные детали алгоритма AdaBoost, давайте взглянем на рис. 7.9, который содействует лучшему пониманию базовой концепции, лежащей в основе AdaBoost.

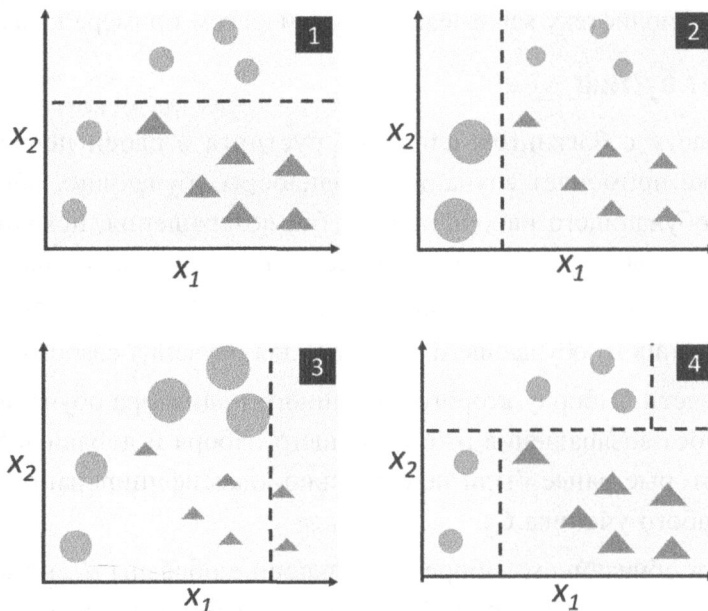


Рис. 7.9. Три раунда алгоритма AdaBoost

Пошаговый проход по иллюстрации AdaBoost мы начинаем с части 1 рисунка, которая представляет обучающий набор для двоичной классификации, где всем обучающим образцам назначены равные веса. На основе этого обучающего набора мы обучаем пенек решения (показанный пунктирной линией), который пытается классифицировать образцы двух классов (треугольники и круги), насколько возможно сводя к минимуму функцию издержек (или показатель загрязненности в специальном случае ансамблей из деревьев принятия решений).

Во втором раунде (часть 2 рисунка) мы назначаем более высокий вес двум ранее неправильно классифицированным образцам (кругам). Кроме того, мы снижаем вес корректно классифицированных образцов. Следующий пенек решения будет теперь больше сосредоточен на обучающих образцах, имеющих самые крупные веса — обучающих образцах, которые предположительно трудно классифицировать. Слабый ученик на части 2 рисунка неправильно классифицирует три образца из класса кругов, которым затем назначается больший вес, как видно на части 3 рисунка.

При условии, что ансамбль AdaBoost состоит только из трех раундов бустинга, мы объединяем трех слабых учеников, обученных на разных повторно взвешенных обучающих поднаборах, по взвешенному большинству голосов, как показано на части 4 рисунка.

Получив лучшее представление о базовой концепции AdaBoost, можно глубже исследовать алгоритм с применением псевдокода. Ради ясности мы будем обозначать поэлементное умножение крестиком ( $\times$ ), а скалярное произведение двух векторов точкой ( $\cdot$ ).

1. Установить в весовом векторе  $\mathbf{w}$  равномерные веса, где  $\sum_i w_i = 1$ .
2. Для  $j$ -того из  $m$  раундов бустинга выполнить следующие действия:
  - а) обучить взвешенного слабого ученика:  $C_j = \text{train}(\mathbf{X}, \mathbf{y}, \mathbf{w})$ ;
  - б) спрогнозировать метки классов:  $\hat{\mathbf{y}} = \text{predict}(C_j, \mathbf{X})$ ;
  - в) вычислить взвешенную частоту ошибок:  $\varepsilon = \mathbf{w} \cdot (\hat{\mathbf{y}} \neq \mathbf{y})$ ;
  - г) вычислить коэффициент:  $\alpha_j = 0.5 \log \frac{1 - \varepsilon}{\varepsilon}$ ;
  - д) обновить веса:  $\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$ ;
  - е) нормализовать веса до суммы, равной 1:  $\mathbf{w} := \mathbf{w} / \sum_i w_i$ .
3. Вычислить финальный прогноз:  $\hat{\mathbf{y}} = \left( \sum_{j=1}^m \left( \alpha_j \times \text{predict}(C_j, \mathbf{X}) \right) > 0 \right)$ .



Обратите внимание, что выражение ( $\hat{y} \neq y$ ) на шаге 2в ссылается на двоичный вектор, состоящий из единиц и нулей, где единица назначается, если прогноз неправильный, и ноль — если правильный.

Несмотря на внешнюю прямолинейность алгоритма AdaBoost, давайте проработаем более конкретный пример, используя обучающий набор из 10 образцов, как демонстрируется в таблице на рис. 7.10.

Индексы образцов	x	y	Вес	$\hat{y} (x \leq 3.0)?$	Правильно?	Обновленные веса
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

*Рис. 7.10. Пример работы алгоритма AdaBoost на обучающем наборе из 10 образцов*

В первом столбце таблицы описаны индексы обучающих образцов от 1 до 10. Во втором столбце приведены значения признаков индивидуальных образцов, исходя из предположения, что это одномерный набор данных. В третьем столбце показаны настоящие метки классов  $y_i$  для каждого обучающего образца  $x_i$ , где  $y_i \in \{1, -1\}$ . В четвертом столбце представлены начальные веса; мы инициализируем веса равномерно (присваивая то же самое константное значение) и нормализуем, чтобы они давали в сумме 1. Следовательно, в случае обучающего набора из 10 образцов мы присваиваем значение 0.1 каждому весу  $w_i$  в весовом векторе  $w$ . В пятом столбце приведены спрогнозированные метки классов  $\hat{y}$  при условии, что критерием разделения является  $x \leq 3.0$ . В последнем столбце таблицы показаны обновленные веса, основанные на правилах обновления, которые мы определили в псевдокоде.

Поскольку вычисление обновлений весов поначалу может выглядеть несколько сложным, давайте отследим его шаг за шагом. Мы начинаем с вычисления взвешенной частоты ошибок  $\varepsilon$ , как было описано в шаге 2в:

$$\begin{aligned}\varepsilon &= 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 0 + 0.1 \times 1 + 0.1 \times 1 \\ &\quad + 0.1 \times 1 + 0.1 \times 0 = \frac{3}{10} = 0.3\end{aligned}$$

Затем мы вычисляем коэффициент (указанный в шаге 2г), который позже применяется в шаге 2д для обновления весов, а также для весов в мажоритарном прогнозе (шаг 3):

$$\alpha_j = 0.5 \log \left( \frac{1 - \varepsilon}{\varepsilon} \right) \approx 0.424$$

После вычисления коэффициента  $\alpha_j$  мы можем обновить весовой вектор, используя следующее уравнение:

$$\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{y} \times y)$$

Здесь  $\hat{y} \times y$  — поэлементное умножение векторов спрогнозированных и настоящих меток классов. Соответственно, если прогноз  $\hat{y}_i$  корректен, то  $\hat{y}_i \times y_i$  будет иметь положительный знак, поэтому мы уменьшаем  $i$ -тый вес, потому что  $\alpha_j$  также является положительным числом:

$$0.1 \times \exp(-0.424 \times 1 \times 1) \approx 0.065$$

Подобным же образом мы будем увеличивать  $i$ -тый вес в случае неправильного прогнозирования метки:

$$0.1 \times \exp(-0.424 \times 1 \times (-1)) \approx 0.153$$

Вот альтернатива:

$$0.1 \times \exp(-0.424 \times (-1) \times (1)) \approx 0.153$$

Обновив каждый вес в весовом векторе, мы нормализуем веса, чтобы они в сумме давали 1 (шаг 2е):

$$\mathbf{w} := \frac{\mathbf{w}}{\sum_i w_i}$$

Здесь  $\sum_i w_i = 7 \times 0.065 + 3 \times 0.153 = 0.914$ .

Таким образом, каждый вес, который соответствует корректно классифицированному образцу, будет уменьшен от начального значения  $0.1$  до  $0.065 / 0.914 \approx 0.071$  для следующего раунда бустинга. Аналогично веса неправильно классифицированных образцов будут увеличиваться от  $0.1$  до  $0.153 / 0.914 \approx 0.167$ .

## Применение алгоритма AdaBoost с помощью scikit-learn

В предыдущем подразделе был кратко представлен алгоритм AdaBoost. Перейдя к части, больше связанной с практикой, мы обучим ансамблевый классификатор на основе AdaBoost посредством библиотеки `scikit-learn`. Мы будем использовать тот же поднабор `Wine`, который применяли в предыдущем разделе для обучения метаклассификатора, основанного на бэггинге. Через атрибут `base_estimator` мы обучим классификатор `AdaBoostClassifier` на 500 пеньках деревьев принятия решений:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           learning_rate=0.1,
...                           random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print('Меры правильности дерева принятия решений при
обучении/испытании %.3f/%.3f'
...       % (tree_train, tree_test))
Меры правильности дерева принятия решений при обучении/испытании
0.916/0.875
```

Легко заметить, что пеньки деревьев принятия решений кажутся недообученными на обучающих данных по контрасту с неподрезанным деревом принятия решений из предыдущего раздела:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
```

```
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print('Меры правильности AdaBoost при обучении/испытании
%.3f/%.3f'
...      % (ada_train, ada_test))
Меры правильности AdaBoost при обучении/испытании 1.000/0.917
```

Как видим, модель AdaBoost корректно прогнозирует все метки классов обучающего набора и показывает немного лучшую эффективность на испытательном наборе в сравнении с пеньком дерева принятия решений. Тем не менее, мы также можем отметить внесение дополнительной дисперсии за счет попытки снизить смещение модели — большой промежуток между эффективностью при обучении и при испытании.

Хотя в демонстрационных целях мы использовали другой простой пример, мы можем заметить, что эффективность классификатора AdaBoost слегка улучшилась по сравнению с пеньком дерева принятия решений и достигла мер правильности, очень похожих на те, которые имеет классификатор на базе бэггинга, обучаемый в предыдущем разделе. Однако мы обязаны отметить, что выбор модели на основе многократного применения испытательного набора — плохая практика. Оценка эффективности обобщения может быть излишне оптимистичной, как мы подробно обсуждали в главе 6.

В заключение давайте посмотрим, на что похожи области решений:

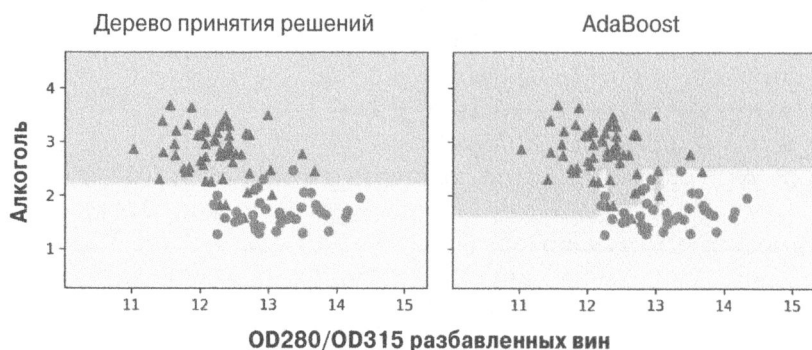
```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                          sharex='col',
...                          sharey='row',
...                          figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                          [tree, ada],
...                          ['Дерево принятия решений', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
```

```

...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('Алкоголь', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='OD280/OD315 разбавленных вин',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()

```

На рис. 7.11 мы видим, что граница решений модели AdaBoost значительно сложнее границы пенька решений. Кроме того, мы отмечаем, что модель AdaBoost разделяет пространство признаков очень похоже на то, как это делает классификатор на основе бэггинга, который мы обучали в предыдущем разделе.



**Рис. 7.11.** Области решений классификаторов на основе дерева принятия решений и алгоритма AdaBoost

В качестве заключительных замечаний об ансамблевых приемах полезно отметить, что ансамблевое обучение увеличивает вычислительную сложность в сравнении с индивидуальными классификаторами. На практике мы

должны хорошо обдумать, стоит ли расплачиваться повышенными вычислительными затратами за нередко относительно скромный рост эффективности прогнозирования.

Часто приводимым примером этого компромисса является известный приз *Netflix Prize* размером 1 миллион долларов, который был получен с использованием ансамблевых приемов. Детали алгоритма были опубликованы в статье А. Тошера, М. Ярепа и Р. Белла “The BigChaos Solution to the Netflix Grand Prize” (Решение команды BigChaos, получившее гран-при Netflix), документация по призу Netflix (2009 г.), которая доступна по ссылке [http://www.stat.osu.edu/~dmsl/GrandPrize2009\\_BPC\\_BigChaos.pdf](http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf). Победившая команда получила гран-при размером 1 миллион долларов; тем не менее, компании Netflix не удалось реализовать их модель из-за высокой сложности, которая делала ее неосуществимой в реальном приложении:

“Мы провели автономную оценку ряда новых методов, но дополнительный выигрыш в точности, который мы измерили, не кажется оправданием инженерных усилий, необходимых для их внедрения в производственную среду” (<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>).



На  
заметку!

### Градиентный бустинг

Еще одним популярным вариантом бустинга является *градиентный бустинг* (*gradient boosting*). Алгоритм AdaBoost и градиентный бустинг разделяют основную общую концепцию: подъем слабых учеников (таких как пеньки деревьев принятия решений) до уровня сильных учеников. Два подхода, адаптивный и градиентный бустинг, отличаются главным образом тем, как обновляются веса и как объединяются (слабые) классификаторы. Если вы знакомы с оптимизацией на базе градиентов и заинтересованы в градиентном бустинге, тогда мы рекомендуем ознакомиться с работой Джерома Фридмана “Greedy function approximation: a gradient boosting machine” (Жадная аппроксимация функций: машина градиентного бустинга, *Annals of Statistics* 2001, с. 1189–1232) и более поздней статьей по алгоритму XGBoost, который по существу является эффективной в плане вычислений реализацией исходного алгоритма градиентного бустинга (“XGBoost: A scalable tree boosting system” (XGBoost: масштабируемая система для бустинга на основе деревьев), Тяньцзи Чен и Карлос Гестрин, материалы 22-й Международной конферен-

ции ACM SIGKDD по обнаружению знаний и интеллектуальному анализу данных, ACM 2016, с.785–794). Обратите внимание, что наряду с реализацией `GradientBoostingClassifier` библиотека `scikit-learn` 0.21 теперь также включает более быструю версию градиентного бустинга `HistGradientBoostingClassifier`, которая даже быстрее, чем `XGBoost`. Дополнительные сведения о классах `GradientBoostingClassifier` и `HistGradientBoostingClassifier` в `scikit-learn` ищите в документации по ссылке <https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting>. Кроме того, краткое и общее объяснение градиентного бустинга можно найти в конспекте лекций по ссылке [https://sebastianraschka.com/pdf/lecture-notes/stat479fs19/07-ensembles\\_\\_notes.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479fs19/07-ensembles__notes.pdf).

## Резюме

В главе мы взглянули на несколько наиболее популярных и широко применяемых приемов для ансамблевого обучения. Ансамблевые методы объединяют различные классификационные модели, чтобы уравновесить их индивидуальные слабые стороны, часто давая в итоге стабильные и хорошо работающие модели, которые очень привлекательны для промышленных приложений и состязаний по МО.

В начале главы мы реализовали на Python классификатор `MajorityVoteClassifier`, который позволяет комбинировать разные алгоритмы классификации. Затем мы рассмотрели бэггинг — полезный прием снижения дисперсии модели за счет выборки случайных бутстрэп-образцов из обучающего набора и объединения отдельно обученных классификаторов посредством мажоритарного голосования. Наконец, мы обсудили `AdaBoost`, представляющий собой алгоритм, который основан на слабых учениках, впоследствии обучающихся на ошибках.

В предшествующих главах вы узнали многое об алгоритмах обучения, настройке и приемах оценки. В следующей главе мы рассмотрим конкретное приложение МО, смысловой анализ, которое стало интересной темой в эпоху Интернета и социальных сетей.

# ПРИМЕНЕНИЕ МАШИННОГО ОБУЧЕНИЯ ДЛЯ СМЫСЛОВОГО АНАЛИЗА

В нынешнюю эпоху Интернета и социальных сетей мнения, рецензии и рекомендации людей стали ценным ресурсом в политологии и бизнесе. Благодаря современным технологиям мы теперь в состоянии собирать и анализировать такие данные более рационально. В этой главе мы углубимся в подобласть *обработки естественного языка* (*natural language processing* — *NLP*), называемую *смысловым анализом* (*sentiment analysis*), и выясним, как использовать алгоритмы МО для классификации документов на основе их направленности: отношения со стороны автора. В частности, мы собираемся работать с набором данных, включающим 50 000 рецензий на фильмы из *базы данных фильмов в Интернете* (*Internet Movie Database* — *IMDb*), и построить прогнозатор, который будет способен проводить различие между положительными и отрицательными рецензиями.

В главе будут раскрыты следующие темы:

- очистка и подготовка текстовых данных;
- построение векторов признаков из текстовых документов;
- обучение модели МО для классификации положительных и отрицательных рецензий на фильмы;



- работа с крупными наборами текстовых данных с применением внешнего обучения (*out-of-core learning*);
- выведение тем из совокупностей документов в целях категоризации.

## Подготовка данных с рецензиями на фильмы IMDb для обработки текста

Как упоминалось ранее, смысловой анализ, иногда также называемый глубинным анализом мнений (*opinion mining*), представляет собой популярную дисциплину из более широкой области NLP; он занимается исследованием направленности документов. Популярной задачей при смысловом анализе является классификация документов на основе выраженных мнений или эмоций авторов в отношении определенной темы.

В главе мы будем иметь дело с крупным набором данных, содержащим рецензии на фильмы, из базы данных IMDb, который был собран Эндрю Маасом и другими (“Learning Word Vectors for Sentiment Analysis” (Изучение словарных векторов для смыслового анализа), Э. Маас, Р. Дели, П. Фам, Д. Хуан, Э. Ын и К. Поттс, труды 49-го ежегодного собрания Ассоциации компьютерной лингвистики: технологии естественного языка, с. 142–150, Портленд, Орегон, США, Ассоциация компьютерной лингвистики (июнь 2011 г.)). Набор данных с рецензиями на фильмы состоит из 50 000 диаметрально противоположных обзоров, которые помечены как положительные или отрицательные; здесь положительный означает, что фильм имеет рейтинг IMDb более шести звезд, а отрицательный — что рейтинг IMDb фильма меньше пяти звезд. В последующих разделах мы загрузим набор данных, обработаем его с целью приведения в формат, пригодный для инструментов МО, и извлечем значимую информацию из поднабора этих рецензий на фильмы, чтобы построить модель МО, которая способна прогнозировать, понравился ли фильм определенному рецензенту.

## Получение набора данных с рецензиями на фильмы

Сжатый посредством gzip архив с набором данных с рецензиями на фильмы (объемом 84,1 Мбайт) можно загрузить по ссылке <http://ai.stanford.edu/~amaas/data/sentiment/>:

- если вы работаете в среде Linux или macOS, то для распаковки набора данных можете открыть новое окно терминала, перейти с помощью `cd` в каталог загрузки и выполнить команду `tar -zxvf aclImdb_v1.tar.gz`;
- если вы работаете в среде Windows, то для распаковки набора данных можете загрузить бесплатную программу архивации, такую как 7-Zip (<http://www.7-zip.org>);
- в качестве альтернативы вы можете распаковать сжатый посредством gzip архив tarball прямо в Python:

```
>>> import tarfile
>>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:
...     tar.extractall()
```

## Предварительная обработка набора данных с целью приведения в более удобный формат

После успешного извлечения набора данных мы соберем индивидуальные текстовые документы из распакованного архива в одиночный файл CSV. В показанном ниже фрагменте кода мы будем читать рецензии на фильмы в объект `DataFrame` из `pandas`, что на стандартном настольном компьютере может потребовать вплоть до 10 минут. Для визуализации хода работ и оценки времени, оставшегося до окончания, мы будем использовать пакет реализации *индикатора выполнения на Python* (*Python Progress Indicator* (`PyPrind`); <https://pypi.python.org/pypi/PyPrind/>), который разработали несколько лет в таких целях. Пакет `PyPrind` можно установить, выполнив команду `pip install pyprind`.

```
>>> import pyprind
>>> import pandas as pd
>>> import os

>>> # укажите в basepath каталог, где находится распакованный
>>> # набор данных с рецензиями на фильмы

>>> basepath = 'aclImdb'
>>>
>>> labels = {'pos': 1, 'neg': 0}
>>> pbar = pyprind.ProgBar(50000)
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
```

```

...     path = os.path.join(basepath, s, 1)
...     for file in sorted(os.listdir(path)):
...         with open(os.path.join(path, file),
...                     'r', encoding='utf-8') as infile:
...             txt = infile.read()
...             df = df.append([[txt, labels[1]]],
...                             ignore_index=True)
...         pbar.update()
>>> df.columns = ['review', 'sentiment']
0%          100%
[#####] | ETA: 00:00:00
Total time elapsed: 00:02:05
Всего прошло времени: 00:02:05

```

В предыдущем коде мы сначала инициализируем новый объект индикатора выполнения `pbar`, указывая 50000 итераций, что соответствует количеству документов, подлежащих чтению. С помощью вложенных циклов `for` мы проходим по подкаталогам `train` и `test` в главном каталоге `aclImdb` и читаем отдельные текстовые файлы из подкаталогов `pos` и `neg`, которые в итоге добавляем к `pandas`-объекту `DataFrame` по имени `df` вместе с целочисленными метками классов (1 — положительный класс и 0 — отрицательный класс).

Поскольку метки классов в собранном наборе данных отсортированы, мы тасуем объект `DataFrame` с применением функции `permutation` из подмодуля `np.random` — это полезно для разбиения набора данных на обучающий и испытательные наборы в будущем, когда мы организуем поток данных напрямую из локального диска. Ради удобства мы также сохраняем собранный и перетасованный набор данных с рецензиями на фильмы в файле `CSV`:

```

>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('movie_data.csv', index=False, encoding='utf-8')

```

Так как мы планируем работать с набором данных позже в главе, давайте быстро удостоверимся в том, что данные были успешно сохранены в правильном формате, прочитав файл `CSV` и выведя выборку из первых трех образцов:

```

>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>>> df.head(3)

```

В случае выполнения примеров кода в Jupyter Notebook должна отобразиться таблица с первыми тремя образцами из набора данных, показанная на рис. 8.1.

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Рис. 8.1. Первые три образца из набора данных с рецензиями на фильмы

В качестве проверки работоспособности до перехода к следующему разделу давайте удостоверимся в том, что объект `DataFrame` содержит все 50 000 строк:

```
>>> df.shape
(50000, 2)
```

## Модель суммирования слов

В главе 4 объяснялось, что мы должны преобразовывать категориальные данные, такие как текст или слова, в числовую форму, прежде чем их можно будет передавать алгоритму МО. В настоящем разделе мы введем *модель суммирования*, или *модель мешка слов* (*bag-of-words model*), которая позволяет представлять текст в виде векторов числовых признаков. В основе модели суммирования слов лежит довольно простая идея, суть которой описана ниже.

1. Мы создаем глоссарий уникальных лексем — например, слов — из полного набора документов.
2. Из каждого документа мы создаем вектор признаков, который содержит счетчики частоты появления каждого слова в отдельном документе.

Поскольку уникальные слова в каждом документе представляют лишь небольшой поднабор всех слов в глоссарии модели суммирования слов, векторы признаков будут состоять главным образом из нулей, из-за чего мы называем их *разреженными*. Не переживайте, если это звучит чересчур аб-

страктно; в последующих подразделах мы пошагово пройдем через процесс создания простой модели суммирования слов.

## Трансформирование слов в векторы признаков

Чтобы построить модель суммирования слов на основе счетчиков слов в соответствующих документах, мы можем использовать класс `CountVectorizer`, реализованный в `scikit-learn`. Как демонстрируется в следующем фрагменте кода, `CountVectorizer` принимает массив текстовых данных, которые могут быть документами или предложениями, и конструирует модель суммирования слов:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array(['The sun is shining',
...                  'The weather is sweet',
...                  'The sun is shining, the weather is sweet, '
...                  'and one and one is two'])
>>> bag = count.fit_transform(docs)
```

Вызывая метод `fit_transform` на `CountVectorizer`, мы создаем глоссарий модели суммирования слов и трансформируем следующие три предложения в разреженные векторы признаков:

- 'The sun is shining' (Солнце светит);
- 'The weather is sweet' (Погода приятная);
- 'The sun is shining, the weather is sweet, and one and one is two' (Солнце светит, погода приятная и один плюс один равно двум).

Теперь давайте выведем содержимое глоссария, чтобы лучше понять лежащие в основе концепции:

```
>>> print(count.vocabulary_)
{'and': 0,
 'two': 7,
 'shining': 3,
 'one': 2,
 'sun': 4,
 'weather': 8,
 'the': 6,
 'sweet': 5,
 'is': 1}
```

После выполнения приведенной выше команды мы видим, что глоссарий хранится в словаре Python, который отображает уникальные слова на целочисленные индексы. Далее выведем только что созданные векторы признаков:

```
>>> print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Индексные позиции в показанных здесь векторах признаков соответствуют целочисленным значениям, которые хранятся как элементы словаря в глоссарии CountVectorizer. Например, первый признак в индексной позиции 0 отражает счетчик слова 'and', появляющегося только в последнем документе, а слово 'is' в индексной позиции 1 (второй признак в векторах документов) встречается во всех трех предложениях. Эти значения в векторах признаков также называются *сырыми частотами термов* (*raw term frequency*):  $tf(t, d)$  — сколько раз терм  $t$  встречается в документе  $d$ . Важно отметить, что в модели суммирования слов порядок следования слов или термов в предложении или документе не играет роли. Порядок, в котором частоты термов расположены в векторе признаков, определяется индексами в словаре, обычно назначаемыми по алфавиту.



На  
заметку!

### Модели n-грамм

Последовательность элементов в только что созданной модели суммирования слов также называется *1-граммной* (*1-gram*) или *униграммной* моделью — каждый элемент или лексема в глоссарии представляет одиночное слово. В более общем случае соприкасающаяся последовательность элементов в NLP — слов, букв или символов — называется *n-граммой* (*n-gram*). Выбор числа  $n$  в  $n$ -грамме зависит от конкретного приложения; скажем, исследование Иоанниса Канариса и других выявило, что  $n$ -граммы размером 3 и 4 обеспечивают хорошую эффективность в фильтровании спама из сообщений электронной почты (“Words versus character  $n$ -grams for anti-spam filtering” (Сравнение представлений в виде слов или буквенных  $n$ -грамм для фильтрования спама), И. Канарис, К. Канарис, И. Гувардас и Э. Стамататос, *International Journal on Artificial Intelligence Tools*, World Scientific Publishing Company, 16(06): с. 1047–1067 (2007 г.)).

Чтобы подытожить концепцию представления в форме  $n$ -грамм, ниже приведены представления в виде 1-граммы и 2-граммы первого документа, “the sun is shining”:

- 1-грамма: “the”, “sun”, “is”, “shining”
- 2-грамма: “the sun”, “sun is”, “is shining”

Класс `CountVectorizer` в `scikit-learn` позволяет применять модели на основе разных  $n$ -грамм через свой параметр `ngram_range`. Хотя по умолчанию используется представление в виде 1-граммы, мы могли бы переключиться на представление в виде 2-граммы, инициализируя новый объект `CountVectorizer` с указанием `ngram_range=(2, 2)`.

## Оценка важности слов с помощью приема `tf-idf`

При анализе текстовых данных мы часто сталкиваемся со словами, которые встречаются во множестве документов из обоих классов. Такие часто встречающиеся слова обычно не содержат полезной или различительной информации. В этом подразделе мы обсудим удобную меру под названием *частота термина — обратная частота документа* (*term frequency-inverse document frequency — tf-idf*), которая может применяться для снижения веса часто встречающихся слов в векторах признаков. Мера `tf-idf` может быть определена как произведение частоты термина на обратную частоту документа:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t, d)$$

Здесь  $\text{tf}(t, d)$  — частота термина, которую мы ввели в предыдущем разделе, а  $\text{idf}(t, d)$  — обратная частота документа, вычисляемая следующим образом:

$$\text{idf}(t, d) = \log \frac{n_d}{1 + \text{df}(d, t)}$$

В приведенном выше уравнении  $n_d$  представляет собой общее количество документов, а частота документа  $\text{df}(t, d)$  — количество документов  $d$ , содержащих терм  $t$ . Обратите внимание, что добавление константы 1 к знаменателю не является обязательным и имеет целью присвоить ненулевое значение терминам, которые встречаются во всех обучающих образцах;  $\log$  используется для того, чтобы гарантировать, что низким частотам документов не назначается слишком большой вес.

В библиотеке `scikit-learn` реализован еще один преобразователь, класс `TfidfTransformer`, который принимает в качестве входа сырые частоты термов от класса `CountVectorizer` и трансформирует их в меры `tf-idf`:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer(use_idf=True,
...                           norm='l2',
...                           smooth_idf=True)
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs))
...       .toarray())
[[ 0.   0.43  0.   0.56  0.56  0.   0.43  0.   0. ]
 [ 0.   0.43  0.   0.   0.   0.56  0.43  0.   0.56]
 [ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

Как было показано в предыдущем подразделе, слово `'is'` имело наибольшую частоту терма в третьем документе, будучи самым часто встречающимся словом. Однако после трансформирования того же вектора признаков в меры `tf-idf` мы видим, что теперь со словом `'is'` ассоциирована относительно небольшая мера `tf-idf` (0.45) в третьем документе, т.к. оно также присутствует в первом и втором документах и соответственно вряд ли содержит любую полезную различительную информацию.

Тем не менее, если бы мы вручную вычислили меры `tf-idf` индивидуальных термов в векторах признаков, то заметили бы, что `TfidfTransformer` вычисляет меры `tf-idf` несколько иначе по сравнению со стандартными уравнениями из учебника, которые мы приводили выше. Уравнение для вычисления обратной частоты документа, реализованное в `scikit-learn`, выглядит следующим образом:

$$\text{idf}(t, d) = \log \frac{1 + n_d}{1 + \text{df}(d, t)}$$

Подобным же образом уравнение для вычисления меры `tf-idf` в `scikit-learn` слегка отклоняется от стандартного уравнения, представленного ранее в главе:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times (\text{idf}(t, d) + 1)$$

Обратите внимание, что добавление “+1” в предыдущем уравнении связано с установкой `smooth_idf=True` в предшествующем примере кода, которая полезна для назначения нулевых весов (т.е.  $\text{idf}(t, d) = \log(1) = 0$ ) термам, встречающимся во всех документах.



Наряду с тем, что более привычно проводить нормализацию сырых частот термов перед вычислением мер `tf-idf`, класс `TfidfTransformer` нормализует меры `tf-idf` напрямую. По умолчанию (`norm='l2'`) класс `TfidfTransformer` из `scikit-learn` применяет нормализацию L2, которая возвращает вектор длиной 1 путем деления ненормализованного вектора признаков  $v$  на его норму L2:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^n v_i^2\right)^{1/2}}$$

Чтобы удостовериться в правильном понимании работы класса `TfidfTransformer`, давайте рассмотрим пример и вычислим меру `tf-idf` слова 'is' в третьем документе.

Слово 'is' имеет частоту термина 3 (`tf` = 3) в третьем документе, а частота документа этого термина равна 3, потому что терм 'is' встречается во всех трех документах (`df` = 3). Таким образом, мы можем вычислить обратную частоту документа, как показано ниже:

$$\text{idf}(\text{"is"}, d3) = \log \frac{1+3}{1+3} = 0$$

Теперь для вычисления меры `tf-idf` нам просто нужно добавить 1 к обратной частоте документа и умножить сумму на частоту термина:

$$\text{tf-idf}(\text{"is"}, d3) = 3 \times (0 + 1) = 3$$

Если мы повторим такое вычисление для всех термов в третьем документе, то получим следующий вектор `tf-idf`: [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]. Однако обратите внимание, что значения в этом векторе признаков отличаются от значений, которые мы получаем от ранее используемого класса `TfidfTransformer`. В приведенном вычислении меры `tf-idf` не хватает финального шага — нормализации L2, которую можно применить следующим образом:

$$\begin{aligned} \text{tf-idf}(d3)_{norm} &= \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}} \\ &= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19] \\ \text{tf-idf}(\text{"is"}, d3) &= 0.45 \end{aligned}$$

Несложно заметить, что сейчас результаты совпадают с результатами, возвращенными классом `TfidfTransformer` из `scikit-learn`, и поскольку теперь понятно, как вычисляются меры `tf-idf`, в следующем разделе мы применим рассмотренные концепции к набору данных с рецензиями на фильмы.

## Очистка текстовых данных

В предшествующих подразделах мы узнали о модели суммирования слов, частотах документов, частотах термов и мерах `tf-idf`. Тем не менее, первый важный шаг (перед построением модели суммирования слов) предусматривает очистку текстовых данных путем удаления всех нежелательных символов. Чтобы проиллюстрировать важность такого шага, мы отобразим последние 50 символов из первого документа в перетасованном наборе данных с рецензиями на фильмы:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

Здесь видно, что текст содержит HTML-разметку, а также знаки препинания и другие небуквенные символы. Хотя HTML-разметка не несет в себе много полезной семантики, знаки препинания в определенных контекстах NLP могут представлять важную дополнительную информацию. Однако ради простоты мы удалим все знаки препинания кроме эмотиконов вроде :), т.к. они, безусловно, полезны для смыслового анализа. Мы будем решать задачу с использованием библиотеки для работы с *регулярными выражениями* (`re`):

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?:|;|=)(?:-)?(?:\)|\(|D|P)',
...                             text)
...     text = (re.sub('[\W]+', ' ', text.lower()) +
...             ' '.join(emoticons).replace('-', ''))
...     return text
```

С помощью первого регулярного выражения, `<[^>]*>`, мы пытаемся удалить всю HTML-разметку из рецензий на фильмы. Несмотря на то что многие программисты в целом не советуют применять регулярные выражения при разборе HTML-разметки, нашего регулярного выражения должно быть достаточно для *очистки* этого конкретного набора данных. Поскольку мы заинтересованы только в удалении HTML-разметки и не планируем в дальней-

шем ее использовать, применение регулярного выражения для выполнения работы должно быть приемлемым. Тем не менее, если для удаления HTML-разметки из текста вы предпочитаете использовать сложно устроенные инструменты, тогда можете взглянуть на модуль `html.parser` в стандартной библиотеке Python, описание которого доступно по ссылке <https://docs.python.org/3/library/html.parser.html>. После удаления HTML-разметки мы задействуем чуть более сложное регулярное выражение для нахождения эмодзи и временно сохраняем их в `emoticons`. Далее мы удаляем из текста несловарные символы посредством регулярного выражения `[\W]+` и приводим все буквы к нижнему регистру.



### Работа со словами, написанными с заглавной буквы

В контексте проводимого анализа мы предполагаем, что написание слова с заглавной буквы (например, в начале предложения) не несет в себе семантически важной информации. Тем не менее, есть исключения — скажем, мы отказываемся от правильной записи имен. Но в рассматриваемом контексте мы опять выдвигаем упрощающее допущение, что регистр букв не содержит информации, которая важна для смыслового анализа.

Наконец, мы дополняем обработанную строку документа эмодзи, временно хранящимися в `emoticons`. Кроме того, в целях согласованности мы удаляем из эмодзи символ “носа”, т.е. “-” в “:-”).



### Регулярные выражения

Несмотря на то что регулярные выражения являются рациональным и удобным подходом для нахождения символов в строке, с ними связана крутая кривая обучения. К сожалению, подробное рассмотрение регулярных выражений выходит за рамки тем, раскрываемых в данной книге. Однако вы можете обратиться к великолепному руководству на портале разработчиков Google по ссылке <https://developers.google.com/edu/python/regular-expressions> или почитать официальную документацию по модулю `re` в Python: <https://docs.python.org/3.7/library/re.html>.

Хотя добавление эмодзи в конец очищенных строк документов может не выглядеть особо элегантным приемом, мы должны отметить, что по-

рядок слов в модели суммирования слов не имеет значения, если глоссарий состоит только из однословных лексем. Но прежде чем дальше обсуждать разбиение документов на отдельные термины, слова или лексемы, давайте выясним, корректно ли работает наша функция `preprocessor`:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Наконец, поскольку мы снова и снова будем использовать *очищенные* текстовые данные в последующих разделах, применим нашу функцию `preprocessor` ко всем рецензиям на фильмы в `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

## Переработка документов в лексемы

После успешной подготовки набора данных с рецензиями на фильмы нам теперь предстоит подумать о том, как разделить совокупность текстов на индивидуальные элементы. Один из способов разбиения документов на лексемы предусматривает разделение очищенных документов на отдельные слова по пробельным символам:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

В контексте разбиения на лексемы еще одним удобным приемом является *стемминг слов* (*word stemming*), или нахождение основы слова, который представляет собой процесс трансформирования слова в его корневую форму. Он позволяет отображать связанные слова на одну и ту же основу. Первоначальный алгоритм стемминга был разработан Мартином Ф. Портером в 1979 году и с тех пор известен как *стеммер Портера* (*Porter stemmer*) или алгоритм стемминга Портера (“An algorithm for suffix stripping” (Алгоритм для отбрасывания суффиксов), Мартин Ф. Портер, Program: Electronic Library and Information Systems, 14(3): с. 130–137 (1980 г.)). Алгоритм стемминга Портера реализован в *наборе инструментов обработки естественного языка* (*Natural Language Toolkit (NLTK)*; <http://www.>

nltk.org) для Python и мы будем его использовать в приведенном ниже фрагменте кода. Чтобы установить NLTK, можно просто выполнить команду `conda install nltk` или `pip install nltk`.



На  
заметку!

### Онлайновая книга по NLTK

Хотя NLTK не является центром внимания главы, настоятельно рекомендуем посетить веб-сайт NLTK и почитать официальную книгу по NLTK, которая свободно доступна по ссылке <http://www.nltk.org/book/>, если вас интересуют более сложные приложения NLP.

В следующем коде демонстрируется применение алгоритма стемминга Портера:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Используя класс `PorterStemmer` из пакета `nltk`, мы модифицировали нашу функцию `tokenizer`, чтобы сократить слова до их корневой формы, как было показано в примере выше, где слово `'running'` трансформировалось в свою корневую форму `'run'`.



На  
заметку!

### Алгоритмы стемминга

Алгоритм стемминга Портера, вероятно, представляет собой самый старый и простой алгоритм стемминга. Среди других алгоритмов стемминга важно отметить более новый *стеммер* “Снежный ком” (*Snowball stemmer*; стеммер Porter2 или стеммер английского языка) и *стеммер Ланкастерского университета* (*Lancaster stemmer*; стеммер Пейса/Хаска (*Paice/Husk stemmer*)). Наряду с тем, что стеммер “Снежный ком” и стеммер Ланкастерского университета быстрее первоначального стеммера Портера, стеммер Ланкастерского университета также печально известен своей большей агрессивностью по сравнению со стеммером Портера. Упомянутые альтернативные алгоритмы стемминга доступны в пакете NLTK (<http://www.nltk.org/api/nltk.stem.html>).

В то время как стемминг может создавать несуществующие в реальности слова вроде 'thu' (из 'thus'), что демонстрировалось в предыдущем примере, прием под названием *лемматизация* (*lemmatization*) помогает получить канонические (грамматически правильные) формы индивидуальных слов — так называемые *леммы*. Тем не менее, с вычислительной точки зрения лемматизация является более сложной и затратной, нежели стемминг, к тому же на практике выяснилось, что стемминг и лемматизация оказывают лишь небольшое влияние на эффективность классификации текстов (“Influence of Word Normalization on Text Classification” (Влияние нормализации слов на классификацию текстов), Михал Томан, Роман Тесар и Карел Йежек, труды InSciT, с. 354–358 (2006 г.)).

До того, как переходить к следующему разделу, где мы будем обучать модель МО с применением модели суммирования слов, давайте кратко обсудим еще одну полезную тему — *удаление стоп-слов* (*stop-word removal*). Стоп-слова — это просто такие слова, которые крайне распространены во всех видах текстов и вероятно не несут в себе (или несут, но мало) полезной информации, позволяющей проводить различие между разными классами документов. Примерами стоп-слов в английском языке могут служить “is”, “and”, “has” и “like”. Удаление стоп-слов может пригодиться, когда придется иметь дело с сырыми или нормализованными частотами термов вместо мер *tf-idf*, в которых веса часто встречающихся слов уже понижены.

Для удаления стоп-слов из рецензий на фильмы мы будем использовать набор из 127 стоп-слов английского языка, который доступен в библиотеке NLTK и может быть получен вызовом функции `nltk.download`:

```
>>> import nltk
>>> nltk.download('stopwords')
```

После загрузки набора стоп-слов мы можем загрузить и применить набор стоп-слов английского языка:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> [w for w in tokenizer_porter('a runner likes'
... ' running and runs a lot')[-10:]]
... if w not in stop]
['runner', 'like', 'run', 'run', 'lot']
```

## Обучение логистической регрессионной модели для классификации документов

В этом разделе мы обучим логистическую регрессионную модель, основанную на модели суммирования слов, которая будет классифицировать рецензии на *позитивные* и *негативные*. Первым делом мы разобьем объект DataFrame с очищенными текстовыми документами на 25 000 документов для обучения и 25 000 документов для испытания:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Затем мы воспользуемся объектом GridSearchCV, чтобы отыскать оптимальный набор параметров для нашей логистической регрессионной модели, применяя стратифицированную перекрестную проверку по 5 блокам:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer

>>> tfidf = TfidfVectorizer(strip_accents=None,
...                          lowercase=False,
...                          preprocessor=None)
>>> param_grid = [{'vect__ngram_range': [(1,1)],
...   'vect__stop_words': [stop, None],
...   'vect__tokenizer': [tokenizer,
...   'tokenizer_porter'],
...   'clf__penalty': ['l1', 'l2'],
...   'clf__C': [1.0, 10.0, 100.0]},
...   {'vect__ngram_range': [(1,1)],
...   'vect__stop_words': [stop, None],
...   'vect__tokenizer': [tokenizer,
...   'tokenizer_porter'],
...   'vect__use_idf':[False],
...   'vect__norm':[None],
...   'clf__penalty': ['l1', 'l2'],
...   'clf__C': [1.0, 10.0, 100.0]}
... ]
```

```
>>> lr_tfidf = Pipeline([('vect', tfidf),
...                       ('clf',
...                        LogisticRegression(random_state=0,
                                              solver='liblinear'))])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
...                              scoring='accuracy',
...                              cv=5, verbose=2,
...                              n_jobs=1)
>>> gs_lr_tfidf.fit(X_train, y_train)
```



На  
заметку!

### Многопроцессорная обработка посредством параметра `n_jobs`

Обратите внимание, что в предыдущем примере кода настоятельно рекомендуется устанавливать `n_jobs=-1` (вместо `n_jobs=1`), чтобы задействовать все свободные процессорные ядра и ускорить решетчатый поиск. Однако при выполнении кода с установкой `n_jobs=-1` ряд пользователей Windows сообщали о проблемах, связанных со сложностями, которые создают для многопроцессорной обработки в среде Windows функции `tokenizer` и `tokenizer_porter`. Другой обходной путь предусматривает замену указанных двух функций, `[tokenizer, tokenizer_porter]`, функцией `[str.split]`. Тем не менее, важно иметь в виду, что замена на простую функцию `str.split` не будет поддерживать стемминг.

Когда мы инициализируем объект `GridSearchCV` и его структуру параметров (`param_grid`) с использованием показанного выше кода, то ограничиваем себя лимитированным числом комбинаций параметров, т.к. количество векторов признаков и крупный глоссарий способны сделать решетчатый поиск очень затратным в вычислительном плане. На стандартном настольном компьютере выполнение нашего решетчатого поиска может занять до 40 минут.

В приведенном ранее примере кода мы заменили объекты `CountVectorizer` и `TfidfTransformer` из предыдущего подраздела объектом `TfidfVectorizer`, который комбинирует `CountVectorizer` с `TfidfTransformer`. Наш `param_grid` состоит из двух словарей параметров. В первом словаре мы применяем объект `TfidfVectorizer` со стандартными настройками (`use_idf=True`, `smooth_idf=True` и `norm='l2'`) для вычисления мер `tf-idf`. Во втором словаре мы устанавливаем эти параметры



в `use_idf=False`, `smooth_idf=False` и `norm=None`, чтобы обучить модель на сырых частотах термов. Кроме того, для самого классификатора на основе логистической регрессии мы обучили модели с использованием регуляризации L2 и L1 через параметр штрафа и сравнили силы регуляризации, определив диапазон значений для обратного параметра регуляризации C.

После завершения решетчатого поиска мы можем вывести наилучший набор параметров:

```
>>> print('Наилучший набор параметров: %s '
...       % gs_lr_tfidf.best_params_)
Наилучший набор параметров: {'clf__C': 10.0, 'vect__stop_words':
None, 'clf__penalty': 'l2', 'vect__tokenizer': <function
tokenizer at 0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

В выводе видно, что наилучшие результаты решетчатого поиска мы получили в случае применения обычного разбиения на лексемы без стеммера Портера, в отсутствие библиотеки стоп-слов и с использованием мер tf-idf в сочетании с классификатором на основе логистической регрессии, который применяет регуляризацию L2 с обратным параметром регуляризации C, равным 10.0.

При наличии наилучшей модели, найденной решетчатым поиском, давайте выведем среднюю меру правильности при перекрестной проверке по 5 блокам на обучающем наборе и правильность классификации на испытательном наборе данных:

```
>>> print('Правильность при перекрестной проверке: %.3f'
...       % gs_lr_tfidf.best_score_)
Правильность при перекрестной проверке: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print('Правильность при испытании: %.3f'
...       % clf.score(X_test, y_test))
Правильность при испытании: 0.899
```

Результаты показывают, что наша модель МО способна прогнозировать, является ли рецензия на фильм позитивной или негативной, почти с 90%-ной правильностью.



### Наивный байесовский классификатор

Все еще очень популярным классификатором для классификации текстов остается *наивный байесовский классификатор* (*naïve Bayes classifier*), который заработал популярность в приложениях фильтрации спама из сообщений электронной почты. Наивные байесовские классификаторы легко реализовывать, они рациональны с вычислительной точки зрения и имеют тенденцию работать особенно хорошо на относительно небольших наборах данных в сравнении с другими алгоритмами. Хотя обсуждать наивные байесовские классификаторы в книге мы не будем, заинтересованные читатели могут обратиться к статье “Naive Bayes and Text Classification I — Introduction and Theory” (Наивные байесовские классификаторы и классификация текстов I — введение и теория), С. Рашка, Computing Research Repository (CoRR), abs/1410.5329 (2014 г.), свободно доступной по ссылке <http://arxiv.org/pdf/1410.5329v3.pdf>.

## Работа с более крупными данными — динамические алгоритмы и внешнее обучение

Если вы выполняли примеры кода из предыдущего раздела, то наверняка заметили, что конструирование векторов признаков для набора данных, содержащего 50 000 рецензий на фильмы, во время решетчатого поиска может оказаться весьма затратным в вычислительном плане. Во многих реальных приложениях нередко приходится работать даже с более крупными наборами данных, которые могут не уместиться в памяти компьютера. Поскольку не каждый имеет доступ к суперкомпьютерам, мы теперь применим прием, называемый *внешним обучением*, который дает возможность работать с такими крупными наборами данных, постепенно подгоняя классификатор на меньших пакетах из набора данных.



### Классификация текста с помощью рекуррентных нейронных сетей

В главе 16 мы вернемся к этому набору данных и обучим классификатор, основанный на глубоком обучении (рекуррентной нейронной сети), для классификации рецензий в наборе данных IMDb. Такой основанный на нейронной сети классификатор следует тому же самому принципу внешнего обучения, используя алгоритм оптимизации на базе стохастического градиентного спуска, но не требует конструирования модели суммирования слов.

В главе 2 была представлена концепция *стохастического градиентного спуска* — алгоритма оптимизации, который обновляет веса модели с использованием одного образца за раз. В этом разделе мы задействуем функцию `partial_fit` класса `SGDClassifier` из `scikit-learn` для потоковой передачи документов прямо из локального диска и обучим логистическую регрессионную модель с применением небольших мини-пакетов документов.

Сначала мы определяем функцию `tokenizer`, которая очищает необработанные текстовые данные из файла `movie_data.csv`, созданного в начале главы, и разбивает его на лексемы, одновременно удаляя стоп-слова:

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[>]*>', '', text)
...     emoticons = re.findall('(?:[:|;|=](?:-)?(?:\)|\(|D|P))',
...                             text.lower())
...     text = re.sub('[\W]+', ' ', text.lower()) \
...         + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Далее мы определяем генераторную функцию `stream_docs`, которая читает и возвращает один документ за раз:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # пропустить заголовок
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

Чтобы проверить, корректно ли работает наша функция `stream_docs`, давайте прочитаем первый документ из файла `movie_data.csv`, что должно привести к возвращению кортежа, содержащего текст рецензии и соответствующую метку класса:

```
>>> next(stream_docs(path='movie_data.csv'))
('In 1974, the teenager Martha Moxley ... ', 1)
```

Затем мы определяем функцию `get_minibatch`, которая будет принимать поток документов от функции `stream_docs` и возвращать определенное количество документов, указанное в параметре `size`:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     try:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(label)
...     except StopIteration:
...         return None, None
...     return docs, y
```

К сожалению, мы не можем использовать для внешнего обучения класс `CountVectorizer`, т.к. он требует удержания полного глоссария в памяти. К тому же классу `TfidfVectorizer` необходимо хранить в памяти все векторы признаков обучающего набора, чтобы вычислять обратные частоты документов. Однако в библиотеке `scikit-learn` доступен еще один удобный векторизатор — `HashingVectorizer`, который не зависит от данных и задействует трюк с хешированием через 32-битную функцию `MurmurHash3`, разработанную Остином Эпплби (<https://sites.google.com/site/murmurhash/>):

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
...                          n_features=2**21,
...                          preprocessor=None,
...                          tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1)
>>> doc_stream = stream_docs(path='movie_data.csv')
```

В показанном выше коде мы инициализируем объект `HashingVectorizer` с нашей функцией `tokenizer` и устанавливаем количество признаков в `2**21`. Кроме того, мы повторно инициализируем классификатор на основе логистической регрессии, устанавливая параметр `loss` объекта `SGDClassifier` в `'log'`. Обратите внимание, что за счет выбора большого числа признаков в `HashingVectorizer` мы снижаем шанс возникновения

хеш-коллизий, но также увеличиваем количество коэффициентов в логистической регрессионной модели.

А теперь мы подошли к действительно интересной части. Создав все дополнительные функции, мы можем начать внешнее обучение с применением следующего кода:

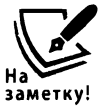
```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                               100%
[#####] | ETA: 00:00:00
Total time elapsed: 00:00:21
Всего прошло времени: 00:00:21
```

Мы снова используем пакет PyPrind для оценки продвижения нашего алгоритма обучения. Мы инициализируем объект индикатора выполнения 45 итерациями и в приведенном ниже цикле `for` проходим по 45 мини-пакетам документов, где каждый мини-пакет состоит из 1 000 документов. По завершении процесса постепенного обучения мы применим последние 5 000 документов для оценки эффективности модели:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print('Правильность: %.3f' % clf.score(X_test, y_test))
Правильность: 0.868
```

Как видим, правильность модели составляет приблизительно 87%, что чуть ниже правильности, которой мы достигали в предыдущем разделе, используя решетчатый поиск для настройки гиперпараметров. Тем не менее, внешнее обучение очень рационально в отношении расхода памяти и требует меньше минуты на свое завершение. Наконец, мы можем применить последние 5 000 документов для обновления модели:

```
>>> clf = clf.partial_fit(X_test, y_test)
```



## Модель word2vec

Более современной альтернативой модели суммирования слов является *word2vec* — алгоритм, который компания Google выпустила в 2013 году (“Efficient Estimation of Word Representations in Vector Space” (Эффективная оценка представлений слов в векторном пространстве), Т. Миколов, К. Чен, Г. Коррадо и Дж. Дин, препринт arXiv:1301.3781 (2013 г.)).

Алгоритм *word2vec* — это алгоритм обучения без учителя, основанный на нейронных сетях, который пытается автоматически узнать взаимосвязь между словами. Идея, лежащая в основе *word2vec*, заключается в том, чтобы помещать слова с похожим смыслом в подобные кластеры, и посредством искусной организации пространства векторов модель способна воспроизводить определенные слова с использованием простой векторной математики, например, *king – man + woman = queen* (король – мужчина + женщина = королева).

Исходную реализацию на языке C вместе с полезными ссылками на связанные работы и альтернативные реализации можно найти по адресу <https://code.google.com/p/word2vec/>.

## Тематическое моделирование с помощью латентного размещения Дирихле

*Тематическое моделирование (topic modeling)* описывает обширную задачу назначения тем непомеченным текстовым документам. Например, типичным приложением могла бы служить категоризация документов в крупном корпусе текстов газетных статей. В приложениях тематического моделирования мы затем стремимся назначить подобным статьям метки категорий — скажем, спорт, финансы, международные новости, политика, местные новости и т.д. Таким образом, в контексте широких категорий МО, обсуждавшихся в главе 1, мы можем считать тематическое моделирование задачей кластеризации, т.е. подкатегорией обучения без учителя.

В этом разделе мы обсудим популярный прием для тематического моделирования, который называется *латентным размещением Дирихле (Latent Dirichlet Allocation — LDA)*. Однако обратите внимание, что аббревиатуру LDA (Latent Dirichlet Allocation) не следует путать с аббревиатурой LDA, обозначающей линейный дискриминантный анализ (linear discriminant analysis) — прием понижения размерности с учителем, который был введен в главе 5.



На заметку!

## Встраивание классификатора рецензий на фильмы в веб-приложение

Прием LDA отличается от подхода обучения с учителем, который мы приняли в этой главе для классификации рецензий на фильмы как позитивные и негативные. Таким образом, если вас интересует встраивание моделей scikit-learn в веб-приложение посредством фреймворка Flask с применением в качестве примера рецензента фильмов, тогда перейдите к чтению следующей главы и позже возвратитесь в конец настоящей главы, чтобы изучить отдельный раздел, посвященный тематическому моделированию.

## Разбиение текстовых документов с помощью LDA

Поскольку лежащая в основе LDA математика довольно сложна и требует знания байесовского вывода, мы подойдем к этой теме с практической точки зрения и будем интерпретировать LDA, используя терминологию для непрофессионалов. Тем не менее, заинтересованные читатели могут почерпнуть дополнительную информацию о LDA из научной статьи “Latent Dirichlet Allocation” (Латентное размещение Дирихле), Дэвид Блей, Эндрю Ын и Майкл Джордан, *Journal of Machine Learning Research* 3, с. 993–1022 (январь 2003 г.).

LDA — порождающая вероятностная модель, которая пытается отыскать группы слов, часто появляющихся вместе в различных документах. Такие часто появляющиеся вместе слова представляют наши темы при допущении, что каждый документ является смесью разных слов. На входе LDA получает модель суммирования слов, которую мы обсуждали ранее в главе, и разлагает ее на две новые матрицы:

- матрица отображения документов на темы;
- матрица отображения слов на темы.

LDA разлагает матрицу суммирования слов на две матрицы таким образом, что если мы перемножим эти две матрицы, то будем в состоянии воспроизвести вход, т.е. матрицу суммирования слов, с самой низкой возможной ошибкой. На практике нас интересуют темы, которые прием LDA нашел в матрице суммирования слов. Единственный недостаток может заключаться в том, что мы обязаны определять количество тем заранее — количество тем является гиперпараметром LDA, который должен указываться вручную.

## Реализация LDA в библиотеке `scikit-learn`

В настоящем подразделе мы будем применять класс `LatentDirichlet Allocation`, реализованный в библиотеке `scikit-learn`, для разложения набора данных с рецензиями на различные темы. В приведенном ниже примере мы ограничиваем анализ 10 темами, но заинтересованные читатели могут поэкспериментировать с гиперпараметрами алгоритма, чтобы выяснить, какие еще темы могут быть найдены в этом наборе данных.

Первым делом мы загрузим набор данных в объект `DataFrame` из `pandas`, используя локальный файл `movie_data.csv` с рецензиями на фильмы, который мы создали в начале главы:

```
>>> import pandas as pd
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Далее мы применим уже знакомый класс `CountVectorizer`, чтобы создать матрицу суммирования слов, служащую входом для LDA. Ради удобства мы будем использовать встроенную в `scikit-learn` библиотеку стоп-слов английского языка через `stop_words='english'`:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer(stop_words='english',
...                         max_df=.1,
...                         max_features=5000)
>>> X = count.fit_transform(df['review'].values)
```

Обратите внимание, что мы устанавливаем максимальную частоту слов, подлежащую рассмотрению в документе, равной 10% (`max_df=.1`), чтобы исключить слова, которые встречаются в документах слишком часто. Основная причина удаления часто встречающихся слов связана с тем, что они могут быть употребительными словами, присутствующими во всех документах и, следовательно, с гораздо меньшей вероятностью ассоциироваться с индивидуальной тематической категорией заданного документа. Мы также ограничиваем количество учитываемых слов пятью тысячами чаще всего встречающихся слов (`max_features=5000`) для установления лимита на размерность набора данных, чтобы улучшить выведение, выполняемое LDA. Однако значения гиперпараметров `max_df=.1` и `max_features=5000` выбраны произвольно, и читатели могут регулировать их, попутно сравнивая результаты.



В следующем примере кода демонстрируется подгонка оценщика `LatentDirichletAllocation` к матрице суммирования слов и выведение 10 разных тем из документов (отметим, что подгонка модели на портативном или стандартном настольном компьютере может занять 5 и более минут):

```
>>> from sklearn.decomposition import LatentDirichletAllocation
>>> lda = LatentDirichletAllocation(n_components=10,
...                                random_state=123,
...                                learning_method='batch')
>>> X_topics = lda.fit_transform(X)
```

Устанавливая `learning_method` в `'batch'`, мы позволяем оценщику `lda` производить оценку на основе всех доступных обучающих данных (матрица суммирования слов) в одной итерации, что медленнее альтернативного метода обучения `'online'`, но может привести к более точным результатам (установка `learning_method` в `'online'` аналогична динамическому или мини-пакетному обучению, которое мы обсуждали в главе 2 и в текущей главе).



### Максимизация ожиданий

На заметку!

Реализация LDA в библиотеке `scikit-learn` применяет алгоритм максимизации ожиданий (*Expectation-Maximization* — EM) для многократного обновления оценок своих параметров. В главе алгоритм EM не обсуждается, но если вам интересно, тогда ознакомьтесь с обзором в Википедии (<https://ru.wikipedia.org/wiki/EM-алгоритм>) и подробным руководством по его использованию в LDA, свободно доступным по ссылке [http://obphio.us/pdfs/lda\\_tutorial.pdf](http://obphio.us/pdfs/lda_tutorial.pdf).

После подгонки LDA мы получаем доступ к атрибуту `components_` объекта `lda`, который хранит матрицу, содержащую значения важности слов (здесь 5000) для каждой из 10 тем в порядке возрастания:

```
>>> lda.components_.shape
(10, 5000)
```

Чтобы проанализировать результаты, давайте выведем пять самых важных слов для каждой из 10 тем. Обратите внимание, что значения важности слов расположены в порядке возрастания. Таким образом, для вывода верхних пяти слов нам необходимо отсортировать массив `topic` в обратном порядке:

```
>>> n_top_words = 5
>>> feature_names = count.get_feature_names()
>>> for topic_idx, topic in enumerate(lda.components_):
...     print("Тема %d:" % (topic_idx + 1))
...     print(" ".join([feature_names[i]
...                       for i in topic.argsort()\
...                           [:-n_top_words - 1:-1]]))
```

Тема 1:

worst minutes awful script stupid

Тема 2:

family mother father children girl

Тема 3:

american war dvd music tv

Тема 4:

human audience cinema art sense

Тема 5:

police guy car dead murder

Тема 6:

horror house sex girl woman

Тема 7:

role performance comedy actor performances

Тема 8:

series episode war episodes tv

Тема 9:

book version original read novel

Тема 10:

action fight guy guys cool

Прочитав пять самых важных слов для каждой темы, мы можем выдвинуть предположение о том, что LDA идентифицировал перечисленные далее темы.

1. В целом плохие фильмы (не является по-настоящему тематической категорией).
2. Семейные фильмы.
3. Военные фильмы.
4. Фильмы об искусстве.
5. Криминальные фильмы.
6. Фильмы ужасов.

7. Комедийные фильмы.
8. Фильмы, как-то связанные с телевизионными шоу.
9. Фильмы по мотивам книг.
10. Фильмы-боевики.

Для подтверждения, что категории имеют смысл на основе рецензий, выведем три фильма из категории фильмов ужасов (категория 6 в индексной позиции 5):

```
>>> horror = X_topics[:, 5].argsort()[::-1]
>>> for iter_idx, movie_idx in enumerate(horror[:3]):
...     print('\nФильм ужасов #%d:' % (iter_idx + 1))
...     print(df['review'][movie_idx][:300], '...')
```

Фильм ужасов #1:

House of Dracula works from the same basic premise as House of Frankenstein from the year before; namely that Universal's three most famous monsters; Dracula, Frankenstein's Monster and The Wolf Man are appearing in the movie together. Naturally, the film is rather messy therefore, but the fact that ...

*Дом Дракулы исходит из того же основного посыла, что вышедший годом ранее Дом Франкенштейна, а именно - три самых знаменитых чудовища от Universal; Дракула, монстр Франкенштейна и Человек-волк появляются в фильме вместе. Конечно, по этой причине фильм местами запутан, но сам факт того, что ...*

Фильм ужасов #2:

Okay, what the hell kind of TRASH have I been watching now? "The Witches' Mountain" has got to be one of the most incoherent and insane Spanish exploitation flicks ever and yet, at the same time, it's also strangely compelling. There's absolutely nothing that makes sense here and I even doubt there ...

*Ладно, что за невыносимую ЕРУНДУ я только что посмотрел?*

*"Гора ведьм" должна навсегда стать одним из самых бессвязных и абсурдных испанских киношек, но вместе с тем она удивительно захватывает. Здесь нет абсолютно никакого смысла, и я даже сомневаюсь, что ...*

Фильм ужасов #3:

<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a total freakfest from start to finish. A fun freakfest at that, but at times it was a tad too reliant on kitsch rather than the horror. The story is difficult to summarize succinctly: a carefree, normal teenage girl starts coming fac ...

*<br /><br />Время фильмов ужасов, японский стиль. Узумаки/Спираль от начала до конца был полным фестивалем фриков. При всей своей забавности фестиваль фриков временами он слишком полагался на китч, чем на ужас. Подытожить историю трудно: легкомысленная нормальная молоденькая девушка начинает сталкиваться с ...*

В предыдущем примере кода мы вывели первые 300 символов из верхних трех фильмов ужасов. Рецензии — несмотря на то, что мы не знаем, к какому фильму они относятся, — выглядят как рецензии на фильмы ужасов (хотя можно было бы утверждать, что Фильм ужасов #2 хорошо подошел бы и под тематическую категорию 1: *В целом плохие фильмы*).

## Резюме

В главе было показано, как применять алгоритмы МО для классификации текстовых документов на основе их направленности, которая является базовой задачей при смысловом анализе в области NLP. Вы не только научились кодировать документ в виде вектора признаков, используя модель суммирования слов, но также узнали, как взвешивать частоту термина по важности с применением меры tf-idf.

Работа с текстовыми данными может быть сопряжена с довольно высокими вычислительными затратами из-за больших векторов признаков, которые создаются во время этого процесса. В последнем разделе главы объяснялось, каким образом задействовать внешнее или постепенное обучение, чтобы обучать алгоритм МО, не загружая полный набор данных в память компьютера.

Наконец, мы представили концепцию тематического моделирования с использованием LDA для разделения рецензий на фильмы по разным категориям в манере без учителя.

В следующей главе мы будем применять наш классификатор документов и посмотрим, как его встроить в веб-приложение.



# ВСТРАИВАНИЕ МОДЕЛИ МАШИННОГО ОБУЧЕНИЯ В ВЕБ-ПРИЛОЖЕНИЕ

**В** предшествующих главах вы ознакомились со многими концепциями и алгоритмами МО, которые могут содействовать принятию лучших и более рациональных решений. Однако приемы МО не ограничиваются автономными приложениями и анализом; они стали прогнозирующими механизмами веб-служб. Например, популярные и удобные случаи употребления моделей МО в веб-приложениях включают обнаружение спама в отправляемых формах, поисковые механизмы, системы выдачи рекомендаций для медийных и продающих порталов, а также многие другие.

В этой главе вы узнаете, как встраивать модель МО в веб-приложение, которое способно не только классифицировать, но и учиться на данных в реальном времени.

В главе будут раскрыты следующие темы:

- сохранение текущего состояния обученной модели МО;
- использование баз данных SQLite для хранилищ данных;
- разработка веб-приложения с применением популярного веб-фреймворка Flask;
- развертывание приложения МО на публичном веб-сервере.

## Сериализация подогнанных оценщиков `scikit-learn`

Как было показано в главе 8, обучение модели МО может оказаться довольно затратным с точки зрения вычислений. Ведь не хотим же мы обучать модель заново каждый раз, когда закрыли интерпретатор Python и желаем выработать прогноз или перезагрузить веб-приложение?

Одним из вариантов обеспечения постоянства моделей является модуль `pickle` для Python (<https://docs.python.org/3.7/library/pickle.html>). Он делает возможными сериализацию и десериализацию структур объектов Python с целью сжатия байт-кода, чтобы мы могли сохранить классификатор в текущем состоянии и опять загрузить его, когда нужно классифицировать новые непомеченные образцы, не заставляя модель учиться на обучающих данных еще раз. Прежде чем выполнять следующий код, удостоверьтесь в том, что обучили внешнюю логистическую регрессионную модель из последнего раздела главы 8 и обеспечили ее готовность в текущем сеансе Python:

```
>>> import pickle
>>> import os
>>> dest = os.path.join('movieclassifier', 'pkl_objects')
>>> if not os.path.exists(dest):
...     os.makedirs(dest)
>>> pickle.dump(stop,
...             open(os.path.join(dest, 'stopwords.pkl'), 'wb'),
...             protocol=4)
>>> pickle.dump(clf,
...             open(os.path.join(dest, 'classifier.pkl'), 'wb'),
...             protocol=4)
```

В приведенном выше коде мы создаем каталог `movieclassifier`, где позже будем хранить файлы и данные для нашего веб-приложения. Внутри каталога `movieclassifier` мы создаем подкаталог `pkl_objects`, чтобы сохранять на локальном жестком диске или твердотельном накопителе сериализованные объекты Python. С помощью метода `dump` модуля `pickle` мы затем сериализуем обученную логистическую регрессионную модель, а также набор стоп-слов из библиотеки NLTK, чтобы устранить необходимость в установке глоссария NLTK на сервере.

Метод `dump` принимает в своем первом аргументе объект, который мы хотим законсервировать. Во втором аргументе предоставляется открытый

файловый объект, куда будет записываться объект Python. Посредством аргумента `wb` внутри функции `open` мы открываем файл в двоичном режиме для консервирования и устанавливаем `protocol=4`, чтобы выбрать самый последний и эффективный протокол консервирования, который появился в Python 3.4 и совместим с последующими версиями Python. В случае возникновения проблем с использованием `protocol=4` проверьте, работаете ли вы с самой последней версией Python 3 — в этой книге рекомендуется Python 3.7. Или же можете обдумать вопрос применения протокола с меньшим номером.

Также имейте в виду, что если вы имеете дело со специальным веб-сервером, то должны также проверить, совместима ли установленная на нем копия Python с версией протокола.



На  
заметку!

### Сериализация массивов NumPy с помощью библиотеки `joblib`

Наша логистическая регрессионная модель содержит несколько массивов NumPy, таких как весовой вектор, а более рациональный способ сериализации массивов NumPy предусматривает использование альтернативной библиотеки `joblib`. Чтобы обеспечить совместимость с серверной средой, которая будет применяться позже в главе, мы воспользуемся стандартным подходом к консервированию. Дополнительные сведения о библиотеке `joblib` доступны по ссылке <https://joblib.readthedocs.io>.

Консервировать объект `HashingVectorizer` нет никакой необходимости, потому что он не нуждается в подгонке. Взамен мы можем создать файл сценария Python, из которого импортировать векторизатор в текущий сеанс Python. Скопируем следующий код и сохраним его в файле `vectorizer.py` внутри каталога `movieclassifier`:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle

cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(os.path.join(
    cur_dir, 'pkl_objects', 'stopwords.pkl'),
    'rb'))
```



```
def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?:[:|;|=)(?:-)?(?:\)|\(|D|P)',
                           text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
        + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                        n_features=2**21,
                        preprocessor=None,
                        tokenizer=tokenizer)
```

После консервирования объектов Python и создания файла `vectorizer.py` имеет смысл перезапустить интерпретатор Python или ядро Jupyter Notebook, чтобы проверить, можем ли мы безошибочно десериализовать объекты.



На  
заметку!

### Консервирование может быть сопряжено с риском в плане безопасности

Тем не менее, имейте в виду, что расконсервирование данных из не заслуживающего доверия источника может быть сопряжено с риском в плане безопасности, т.к. модуль `pickle` не защищен против злонамеренного кода. Поскольку `pickle` проектировался для сериализации произвольных объектов, процесс расконсервирования выполнит код, который был сохранен в консервированном файле. Таким образом, если вы получаете консервированные файлы из не заслуживающего доверия источника (скажем, загружая их из Интернета), тогда предусмотрите дополнительные меры предосторожности — расконсервируйте элементы в виртуальной среде и/или на второстепенной машине, где отсутствуют важные данные, к которым никто кроме вас не должен иметь доступ.

В терминальном окне перейдем в каталог `movieclassifier`, запустим новый сеанс Python и выполним приведенный ниже код для проверки, можем ли мы импортировать `vectorizer` и расконсервировать классификатор:

```
>>> import pickle
>>> import re
>>> import os
>>> from vectorizer import vect
```

```
>>> clf = pickle.load(open(os.path.join(
...                         'pkl_objects', 'classifier.pkl'),
...                         'rb'))
```

После успешной загрузки `vectorizer` и расконсервирования классификатора мы можем использовать эти объекты для предварительной обработки образцов документов и выработки прогнозов об их отношениях:

```
>>> import numpy as np
>>> label = {0: 'негативный', 1: 'позитивный'}
>>> example = ["I love this movie. It's amazing."]
>>> X = vect.transform(example)
>>> print('Прогноз: %s\nВероятность: %.2f%%' % \
...       (label[clf.predict(X)[0]],
...       np.max(clf.predict_proba(X))*100))
Прогноз: позитивный
Вероятность: 95.55%
```

Поскольку наш классификатор возвращает спрогнозированные метки классов как целые числа, мы определили простой словарь Python для отображения таких целых чисел на их отношения ('негативный' или 'позитивный'). Хотя в итоге получилось простое приложение только с двумя классами, необходимо отметить, что подход с отображением посредством словаря также обобщается на многоклассовые конфигурации. Кроме того, такой словарь отображения должен архивироваться вместе с моделью.

В этом случае из-за того, что определение словаря включает лишь одну строку кода, мы не будем прилагать усилия по его сериализации с применением `pickle`. Однако в реальных приложениях с более обширными словарями отображения вы можете задействовать те же самые команды `pickle.dump` и `pickle.load`, которые использовались в предыдущем примере кода.

Продолжая обсуждение предыдущего примера кода, мы затем применили `HashingVectorizer` для трансформации простого примера документа в вектор слов `X`. Наконец, мы использовали метод `predict` классификатора на основе логистической регрессии для прогнозирования метки класса, а также метод `predict_proba` для возвращения соответствующей вероятности прогноза. Обратите внимание, что в результате вызова метода `predict_proba` возвращается массив со значениями вероятностей для всех уникальных метод классов. Так как метка класса с наибольшей вероятностью соответствует

метке класса, возвращаемой методом `predict`, для возвращения вероятности спрогнозированного класса мы применили функцию `np.max`.

## Настройка базы данных SQLite для хранилища данных

В текущем разделе мы настроим простую базу данных SQLite для сбора дополнительных отзывов о прогнозах от пользователей веб-приложения. Мы можем использовать такую обратную связь для обновления классификационной модели. SQLite — это механизм баз данных SQL с открытым кодом, не требующий для своей работы отдельного сервера, что делает его идеальным вариантом при разработке небольших проектов и простых веб-приложений. По существу базу данных SQLite можно считать одиночным самодостаточным файлом базы данных, который позволяет напрямую обращаться к хранилищу.

Кроме того, SQLite не требует конфигурирования, специфичного для системы, и поддерживается во всех распространенных операционных системах. Механизм SQLite известен своей высокой надежностью и применяется популярными компаниями, среди которых Google, Mozilla, Adobe, Apple, Microsoft и многие другие. Дополнительная информация о механизме SQLite доступна на официальном веб-сайте <http://www.sqlite.org>.

К счастью, согласно принятой в Python философии “батарейки в комплекте” стандартная библиотека Python предлагает API-интерфейс `sqlite3`, который позволяет работать с базами данных SQLite. (Больше сведений о модуле `sqlite3` можно получить по ссылке <https://docs.python.org/3.7/library/sqlite3.html>.)

Выполнив следующий код, мы создадим внутри каталога `movieclassifier` новую базу данных SQLite и сохраним в ней два образца рецензий на фильмы:

```
>>> import sqlite3
>>> import os

>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute('DROP TABLE IF EXISTS review_db')
>>> c.execute('CREATE TABLE review_db\'
...           ' (review TEXT, sentiment INTEGER, date TEXT)')
```

```

>>> example1 = 'I love this movie'
>>> c.execute("INSERT INTO review_db\"
...           " (review, sentiment, date) VALUES\"
...           " (?, ?, DATETIME('now'))", (example1, 1))
>>> example2 = 'I disliked this movie'
>>> c.execute("INSERT INTO review_db\"
...           " (review, sentiment, date) VALUES\"
...           " (?, ?, DATETIME('now'))", (example2, 0))
>>> conn.commit()
>>> conn.close()

```

В коде мы создаем подключение (conn) к файлу базы данных SQLite, вызывая метод connect из библиотеки sqlite3, который создает в каталоге movieclassifier файл базы данных reviews.sqlite, если он пока не существует.

Затем посредством метода cursor мы создаем курсор, который даст возможность проходить по записям базы данных, используя универсальный синтаксис SQL. Далее с применением первого вызова execute мы создаем новую таблицу базы данных review\_db, которую будем использовать для сохранения и последующего доступа к записям. В таблице review\_db мы создаем три столбца: review, sentiment и date. Они будут применяться для хранения примеров рецензий на фильмы и соответствующих меток классов (отношений).

С использованием SQL-команды DATETIME('now') мы добавляем к записям дату и отметку времени. Знаки вопроса (?) применяются для передачи методу execute текстов рецензий на фильмы (example1 и example2) и связанных с ними меток классов (1 и 0) в виде позиционных аргументов как членов кортежа. В заключение мы вызываем метод commit для сохранения изменений, внесенных в базу данных, и закрываем подключение посредством метода close.

Чтобы проверить, корректно ли сохранились записи в базе данных, мы снова откроем подключение к базе данных и с помощью SQL-команды SELECT извлечем из таблицы все строки, зафиксированные в период между началом 2017 года и текущей датой:

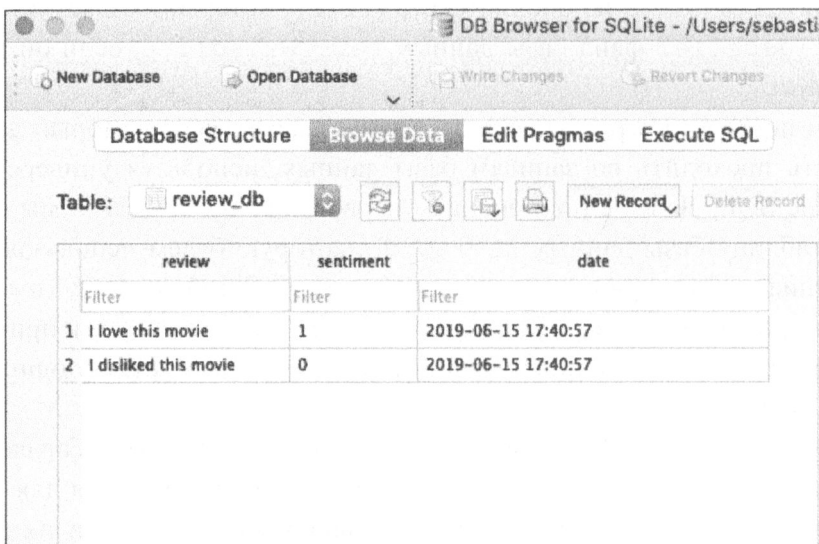
```

>>> conn = sqlite3.connect('reviews.sqlite')
>>> c = conn.cursor()
>>> c.execute("SELECT * FROM review_db WHERE date\"
...           " BETWEEN '2017-01-01 00:00:00' AND DATETIME('now')")

```

```
>>> results = c.fetchall()
>>> conn.close()
>>> print(results)
[('I love this movie', 1, '2020-04-15 17:53:46'), ('I disliked
this movie', 0, '2020-04-15 17:53:46')]
```

В качестве альтернативы мы могли бы также воспользоваться бесплатным приложением DB Browser for SQLite (Программа просмотра баз данных SQLite), доступным по ссылке <https://sqlitebrowser.org/dl/>; оно предлагает удобный графический пользовательский интерфейс для работы с базами данных SQLite (рис. 9.1).



*Рис. 9.1. Приложение DB Browser for SQLite*

## Разработка веб-приложения с помощью Flask

Теперь, когда код для классификации рецензий на фильмы готов, давайте обсудим основы веб-фреймворка Flask, необходимого для разработки нашего веб-приложения. После выпуска Армином Ронахером первоначальной версии Flask в 2010 году этот фреймворк с годами обрел огромную популярность и применяется в таких известных приложениях, как LinkedIn и Pinterest. Поскольку фреймворк Flask написан на языке Python, он снабжает программистов на Python удобным интерфейсом для встраивания существующего кода Python, подобного классификатору рецензий на фильмы.



## Микрофреймворк Flask

Flask также называют *микрофреймворком*, имея в виду тот факт, что его ядро сохранено небольшим и простым, но может легко расширяться с использованием других библиотек. Хотя кривая обучения легковесного API-интерфейса Flask не настолько крута, как у других популярных веб-фреймворков вроде Django, рекомендуется заглянуть в официальную документацию по ссылке <https://flask.palletsprojects.com/en/1.0.x/>, чтобы получить больше сведений о его функциональности.

Если библиотека Flask в текущей среде Python отсутствует, тогда ее можно установить с помощью команды `conda` или `pip` в терминальном окне (на момент написания главы последним стабильным выпуском был 1.0.2):

```
conda install flask
# или pip install flask
```

## Первое веб-приложение Flask

Перед тем, как заняться реализацией классификатора рецензий на фильмы, в настоящем подразделе мы разработаем очень простое веб-приложение с целью ознакомления с API-интерфейсом Flask. Первое приложение будет состоять из простой веб-страницы с полем формы, которое позволит вводить имя. После отправки формы веб-приложению визуализируется новая страница. Несмотря на свою простоту, этот пример веб-приложения поможет получить представление о том, как переменные сохраняются и передаются между разными частями кода внутри фреймворка Flask.

Прежде всего, мы создаем дерево каталогов:

```
1st_flask_app_1/
  app.py
  templates/
    first_app.html
```

Файл `app.py` содержит основной код, который будет выполняться интерпретатором Python для запуска веб-приложения Flask. В каталоге `templates` фреймворк Flask будет искать статические HTML-файлы для визуализации в веб-браузере. Теперь взглянем на содержимое файла `app.py`:

```
from flask import Flask, render_template

app = Flask(__name__)
@app.route('/')
def index():
    return render_template('first_app.html')

if __name__ == '__main__':
    app.run()
```

Давайте обсудим индивидуальные части кода по очереди.

1. Мы запускаем приложение как одиночный модуль. Таким образом, мы инициализируем новый экземпляр Flask с аргументом `__name__`, чтобы фреймворку Flask было известно, что подкаталог HTML-шаблонов (templates) располагается в том же каталоге, где находится приложение.
2. Затем мы применяем декоратор маршрута (`@app.route('/')`), чтобы указать URL, который должен инициировать выполнение функции `index`.
3. Далее функция `index` просто визуализирует HTML-файл `first_app.html`, находящийся в подкаталоге `templates`.
4. Наконец, мы используем функцию `run` для запуска приложения на сервере, когда этот сценарий непосредственно выполняется интерпретатором Python, что гарантируется оператором `if` с условием `__name__ == '__main__'`.

Рассмотрим содержимое файла `first_app.html`:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
  </head>
  <body>
    <div>Hi, this is my first Flask web app!</div>
  </body>
</html>
```



## Основы HTML

На  
заметку!

На тот случай, если синтаксис HTML вам не знаком, по ссылке <https://developer.mozilla.org/ru/docs/Web/HTML> доступно удобное руководство по основам HTML.

Здесь мы просто заполняем пустой файл HTML-шаблона с помощью элемента `<div>` (элемент блочного уровня), который содержит предложение: `Hi, this is my first Flask web app!`.

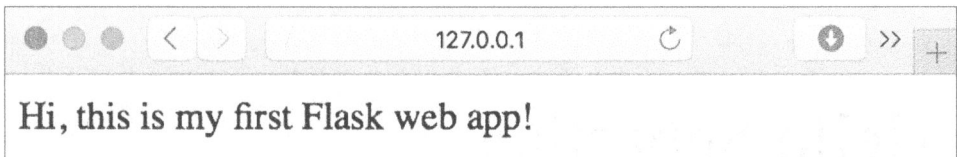
Фреймворк Flask позволяет запускать приложения локально, что удобно для разработки и тестирования веб-приложений перед их развертыванием на публичном веб-сервере. Давайте запустим наше веб-приложение, выполнив в терминальном окне следующую команду из текущего каталога `1st_flask_app_1`:

```
python3 app.py
```

Мы должны увидеть в терминальном окне строку такого вида:

```
* Running on http://127.0.0.1:5000/
* Выполняется на http://127.0.0.1:5000/
```

Строка содержит адрес локального сервера. Мы можем ввести этот адрес в адресной строке веб-браузера, чтобы посмотреть на веб-приложение в действии. Если все было сделано корректно, тогда отобразится простой веб-сайт с содержимым `Hi, this is my first Flask web app!` (рис. 9.2).



*Рис. 9.2. Пример веб-приложения Flask в действии*

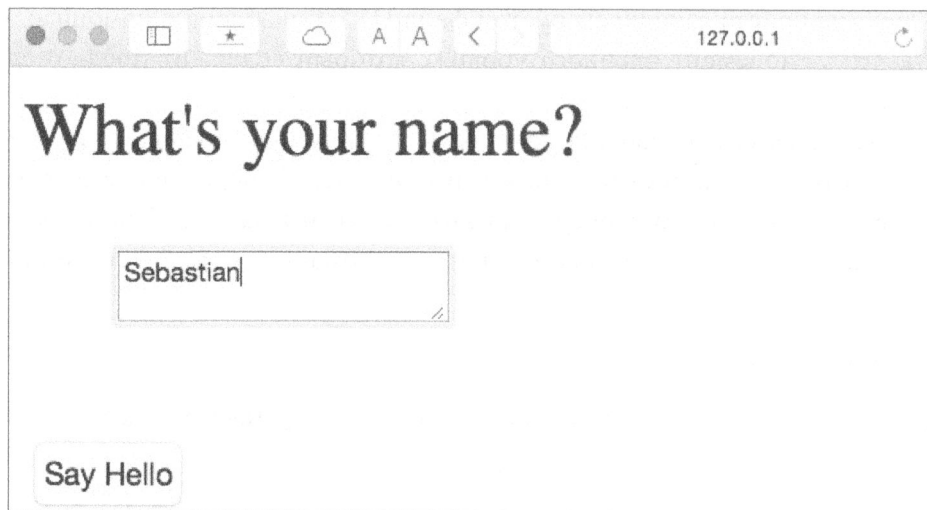
## Проверка достоверности и визуализация форм

В этом подразделе мы расширим наше простое веб-приложение Flask элементами HTML-формы, чтобы научиться собирать данные от пользователя с применением библиотеки WTForms (<https://wtforms.readthedocs.org/en/latest/>), которую можно установить через `conda` или `pip`:

```
conda install wtforms
# или pip install wtforms
```

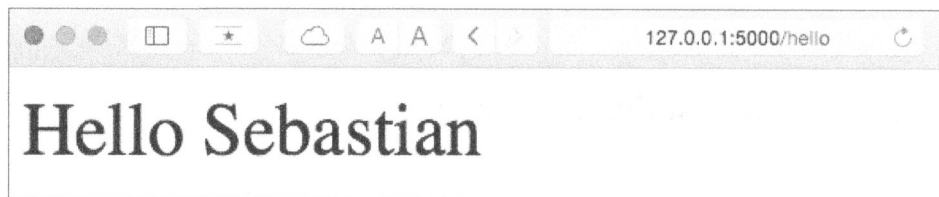
Веб-приложение будет предлагать пользователю ввести свое имя в текстовом поле (рис. 9.3).





*Рис. 9.3. Запрос имени у пользователя*

После щелчка на кнопке отправки (Say Hello (Поприветствовать)) и проверки достоверности формы будет визуализирована новая HTML-страница для отображения имени пользователя (рис. 9.4).



*Рис. 9.4. Отображение имени пользователя*

## Настройка структуры каталогов

Новая структура каталогов, которую нам нужно настроить для данного приложения, выглядит следующим образом:

```
1st_flask_app_2/  
  app.py  
  static/  
    style.css  
  templates/  
    _formhelpers.html  
    first_app.html  
    hello.html
```

Ниже показано модифицированное содержимое `app.py`:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('', [validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

Давайте проанализируем приведенный код шаг за шагом.

1. Используя `wtforms`, мы расширяем функцию `index` текстовым полем, которое будет встраиваться в стартовую страницу с применением класса `TextAreaField`, автоматически проверяющего наличие допустимого ввода от пользователя.
2. Кроме того, мы определяем новую функцию `hello`, которая будет визуализировать HTML-страницу `hello.html` после проверки достоверности HTML-формы.
3. Далее мы используем метод `POST` для передачи данных формы серверу в теле сообщения. В заключение установкой аргумента `debug=True` внутри метода `app.run` мы дополнительно активизируем отладчик `Flask`, который является полезным средством при разработке новых веб-приложений.

## Реализация макроса с использованием механизма шаблонизации Jinja2

Теперь с применением механизма шаблонизации Jinja2 мы реализуем обобщенный макрос в файле `_formhelpers.html`, который позже импортируем в файле `first_app.html`, чтобы визуализировать текстовое поле:

```
{% macro render_field(field) %}
<dt>{{ field.label }}
<dd>{{ field(**kwargs)|safe }}
{% if field.errors %}
<ul class=errors>
{% for error in field.errors %}
<li>{{ error }}</li>
{% endfor %}
</ul>
{% endif %}
</dd>
</dt>
{% endmacro %}
```

Подробное обсуждение языка шаблонизации Jinja2 выходит за рамки этой книги. Однако на веб-сайте <http://jinja.pocoo.org> можно найти всеобъемлющую документацию по синтаксису Jinja2.

## Добавление стиля в файл CSS

Далее мы создадим простой *CSS-файл* (*Cascading Style Sheet* — каскадная таблица стилей) `style.css` для демонстрации того, как можно изменять внешний вид и восприятие HTML-документов. Мы должны сохранить CSS-файл со следующим содержимым, которое просто удваивает размер шрифта HTML-элементов `body`, в подкаталоге по имени `static` — стандартном месте, где фреймворк Flask ищет статические файлы наподобие CSS:

```
body {
    font-size: 2em;
}
```

Вот модифицированное содержимое файла `first_app.html`, которое теперь визуализирует текстовое поле для ввода пользователем своего имени:

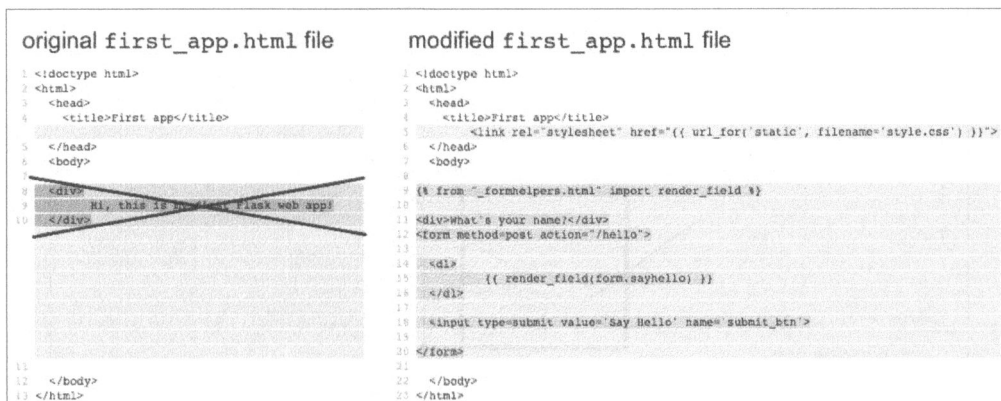
```
<!doctype html>
<html>
<head>
```

```

<title>First app</title>
<link rel="stylesheet"
      href="{{ url_for('static', filename='style.css') }}">
</head>
<body>
    {% from "_formhelpers.html" import render_field %}
    <div>What's your name?</div>
    <form method=post action="/hello">
        <dl>
            {{ render_field(form.sayhello) }}
        </dl>
        <input type=submit value='Say Hello' name='submit_btn'>
    </form>
</body>
</html>

```

Мы загружаем CSS-файл в разделе заголовка HTML-разметки `first_app.html`. Теперь размер всех текстовых элементов внутри HTML-элемента `body` должен измениться. В разделе тела HTML мы импортируем макрос формы из `_formhelpers.html` и визуализируем форму `sayhello`, которую указали в файле `app.py`. Кроме того, мы добавили кнопку к тому же самому элементу `form`, чтобы пользователь мог отправлять запись из текстового поля. Отличия между первоначальной и модифицированной версиями файла `first_app.html` демонстрируются на рис. 9.5.



**Рис. 9.5.** Отличия между первоначальной и модифицированной версиями файла `first_app.html`

## Создание результирующей страницы

Наконец, мы создадим файл `hello.html`, который будет визуализироваться посредством строки `return render_template('hello.html', name=name)` внутри функции `hello`, определенной в сценарии `app.py`, с целью отображения отправленного пользователем текста. Вот содержимое файла `hello.html`:

```
<!doctype html>
<html>
  <head>
    <title>First app</title>
    <link rel="stylesheet"
      href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>
    <div>Hello {{ name }}</div>
  </body>
</html>
```

Поскольку в предыдущем разделе мы раскрыли много основ, на рис. 9.6 представлен обзор ранее созданных файлов.

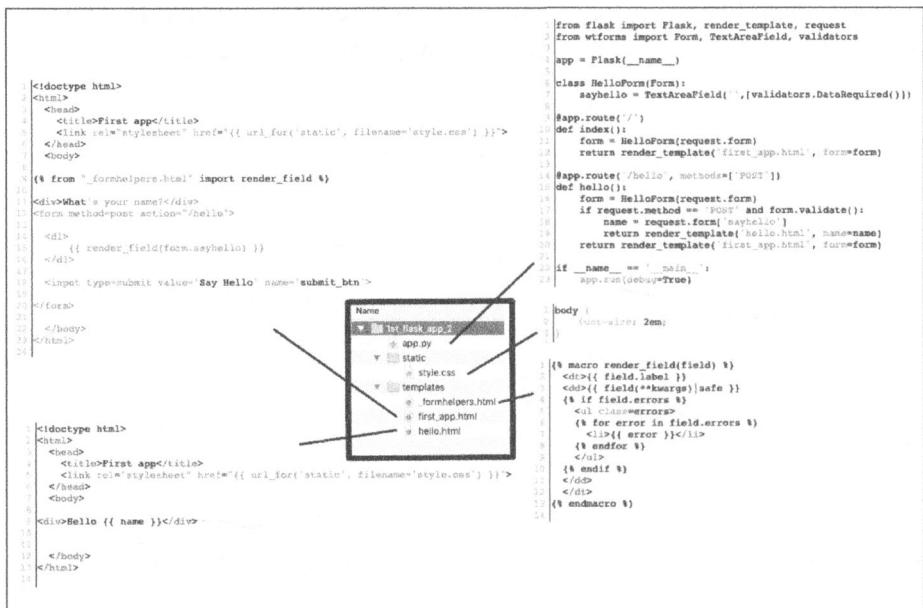


Рис. 9.6. Обзор ранее созданных файлов

Копировать какой-то код из показанного на рис. 9.6 не придется, т.к. содержимое всех файлов приводилось в предшествующих разделах. Для вашего удобства копии всех файлов также доступны по ссылке [https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch09/1st\\_flask\\_app\\_2](https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch09/1st_flask_app_2).

После модификации веб-приложения Flask мы можем запустить его локально, выполнив следующую команду из главного каталога приложения:

```
python3 app.py
```

Для просмотра результирующей веб-страницы введите в веб-браузере IP-адрес, отобразившийся в терминальном окне, которым обычно будет `http://127.0.0.1:5000/` (рис. 9.7).

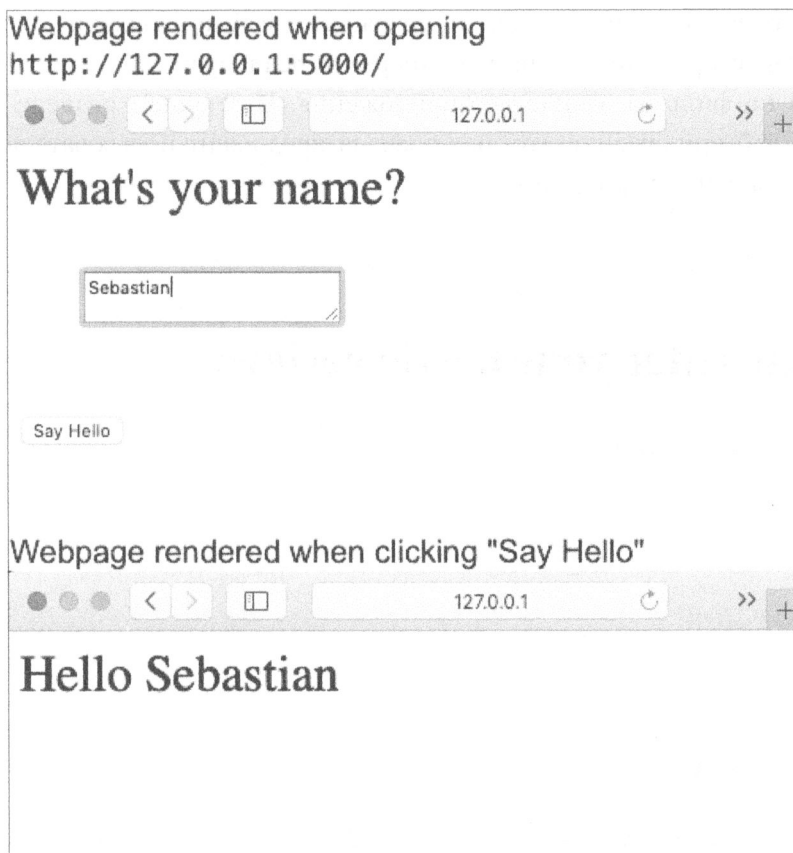


Рис. 9.7. Результирующая веб-страница



## Документация и примеры приложений Flask

Если вы — новичок в области разработки веб-приложений, то на первый взгляд некоторые концепции могут выглядеть очень сложными. В таком случае поместите предшествующие файлы в какой-то каталог на своем жестком диске и внимательно их исследуйте. Вы увидите, что веб-фреймворк Flask относительно прямолинеен и гораздо проще, чем может показаться поначалу. Вдобавок не забывайте обращаться к великолепной документации и примерам приложений Flask по ссылке <http://flask.pocoo.org/docs/1.0/>.

## Преобразование классификатора рецензий на фильмы в веб-приложение

После ознакомления с основами разработки веб-приложений с помощью Flask давайте продвинемся на шаг вперед и реализуем наш классификатор рецензий на фильмы в виде веб-приложения. В текущем разделе мы разработаем веб-приложение, которое сначала предложит пользователю ввести рецензию на фильм (рис. 9.8).

raschkas.pythonanywhere.com/

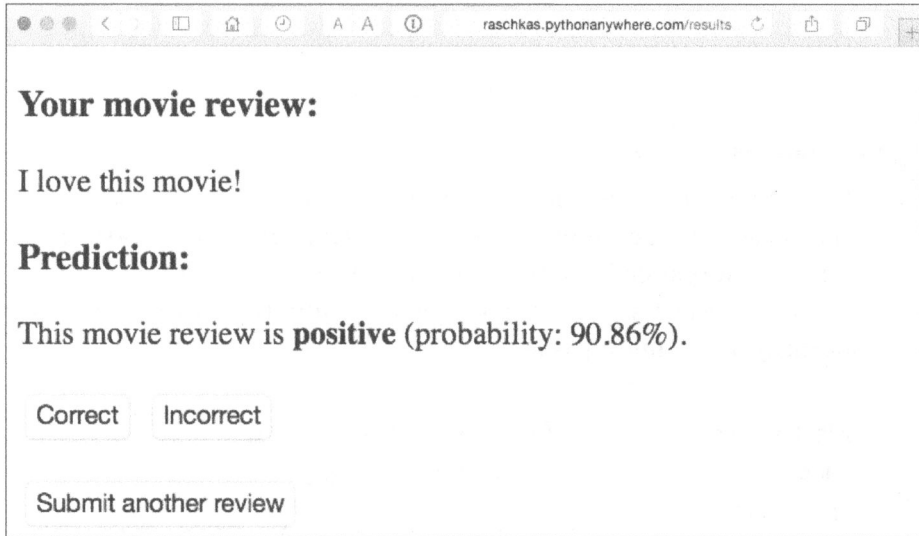
Please enter your movie review:

I love this movie!

Submit review

*Рис. 9.8. Запрос рецензии на фильм у пользователя*

После отправки рецензии пользователь увидит новую страницу, которая отобразит спрогнозированную метку класса и вероятность прогноза. К тому же пользователь получит возможность оставить отзыв о выработанном прогнозе, щелкая на кнопке **Correct** (Правильный) или **Incorrect** (Неправильный), как показано на рис. 9.9.



*Рис. 9.9. Отображение выработанного прогноза и возможность отправки отзыва о нем*

Если пользователь щелкнет на одной из кнопок **Correct** или **Incorrect**, тогда наша классификационная модель будет обновлена согласно пользовательскому отзыву. Кроме того, мы также сохраним введенный пользователем текст рецензии на фильм и предполагаемую метку класса, которую можно вывести из щелчка на кнопке, в базе данных SQLite для ссылки в будущем. (В качестве альтернативы пользователь мог бы пропустить шаг обновления и щелкнуть на кнопке **Submit another review** (Отправить еще одну рецензию), чтобы отправить еще одну рецензию.)

Третьей страницей, которую пользователь увидит после щелчка на одной из кнопок обратной связи, будет простой экран благодарности с кнопкой **Submit another review**, щелчок на которой приводит к перенаправлению пользователя обратно на стартовую страницу (рис. 9.10).





Рис. 9.10. Экран благодарности

**Живая демонстрация**

Прежде чем мы более пристально взглянем на код реализации рассматриваемого веб-приложения, рекомендуем поработать с живой демонстрацией, доступной по ссылке <http://raschkas.pythonanywhere.com>, чтобы лучше понимать, чего мы пытаемся достигнуть в данном разделе.

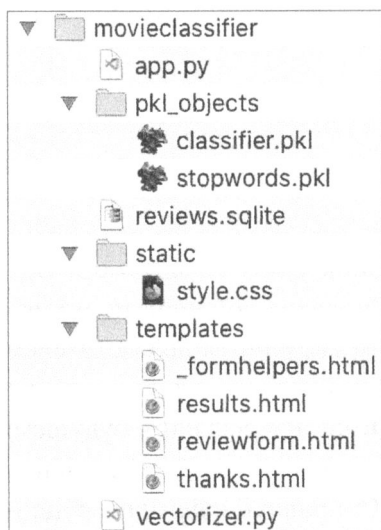


Рис. 9.11. Дерево каталогов для приложения классификации рецензий на фильмы

**Файлы и подкаталоги — дерево каталогов**

Чтобы начать с общей картины, давайте посмотрим на дерево каталогов, которое мы собираемся создать для приложения классификации рецензий на фильмы (рис. 9.11). Ранее в главе мы уже создали файл `vectorizer.py`, базу данных SQLite в файле `reviews.sqlite` и подкаталог `pk1_objects` с законсервированными объектами Python. Файл `app.py` в главном каталоге — это сценарий Python, который содержит код Flask, а файл базы данных `review.sqlite` будет использоваться для хранения рецензий на фильмы, отправленные нашему веб-приложению. В подкаталоге `templates` находятся HTML-шаблоны,

которые будут визуализироваться фреймворком Flask и отображаться в веб-браузере, а в подкаталоге `static` — простой CSS-файл, предназначенный для подстройки внешнего вида визуализируемой HTML-разметки.



## Получение файлов кода для классификатора рецензий на фильмы

Отдельный каталог, который содержит приложение классификатора рецензий на фильмы с кодом, обсуждаемым в настоящем разделе, входит в состав архива примеров кода для книги. Архив доступен для загрузки на GitHub по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition/>. Код, относящийся к текущему разделу, находится в подкаталоге `.../ch09/movieclassifier`.

## Реализация главного приложения как `app.py`

Поскольку содержимое файла `app.py` довольно длинное, мы разберем его в два этапа. Первая часть `app.py` импортирует модули и объекты Python, которые нам понадобятся, а также код для расконсервирования и настройки классификационной модели:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np

# импортировать HashingVectorizer из локального каталога
from vectorizer import vect

app = Flask(__name__)

##### подготовка классификатора
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                   'pkl_objects', 'classifier.pkl'),
                                   'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    # 0: 'негативный', 1: 'позитивный'
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = np.max(clf.predict_proba(X))
    return label[y], proba
```

```

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"
              " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()

```

К этому времени первая часть сценария `app.py` должна выглядеть хорошо знакомой. Мы просто импортируем `HashingVectorizer` и расконсервируем классификатор на основе логистической регрессии. Затем мы определяем функцию `classify` для возвращения спрогнозированной метки класса и вероятности прогноза, относящиеся к заданному текстовому документу. Функция `train` может применяться для обновления классификатора при условии, что были предоставлены документ и метка класса.

С использованием функции `sqlite_entry` мы можем сохранять в базе данных отправленную рецензию на фильм вместе с ее меткой класса и отметкой времени в регистрационных целях. Обратите внимание, что при перезапуске веб-приложения объект `clf` будет сбрасываться в исходное законсервированное состояние. В конце главы вы узнаете, как применять накопленную в базе данных SQLite информацию для постоянного обновления классификатора.

Концепции во второй части сценария `the app.py` также должны выглядеть довольно знакомыми:

```

##### Flask
class ReviewForm(Form):
    moviereview = TextAreaField('',
                                [validators.DataRequired(),
                                 validators.length(min=15)])

@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():

```

```

form = ReviewForm(request.form)
if request.method == 'POST' and form.validate():
    review = request.form['moviereview']
    y, proba = classify(review)
    return render_template('results.html',
                           content=review,
                           prediction=y,
                           probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)
@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negative': 0, 'positive': 1}
                # 'негативный': 0, 'позитивный': 1
    y = inv_label[prediction]
    if feedback == 'Incorrect':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')
if __name__ == '__main__':
    app.run(debug=True)

```

Мы определяем класс `ReviewForm`, создающий экземпляр `TextAreaField`, который будет визуализирован в файле шаблона `reviewform.html` (целевая страница нашего веб-приложения). В свою очередь это визуализируется функцией `index`. С помощью параметра `validators.length(min=15)` мы требуем, чтобы пользователь вводил рецензию, содержащую минимум 15 символов. Внутри функции `results` мы извлекаем содержимое отправленной веб-формы и передаем его классификатору для выработки прогноза отношения в рецензии на фильм, который затем отображается в визуализированном шаблоне `results.html`.

Функция `feedback`, которую мы реализовали в `app.py` в предыдущем подразделе, на первый взгляд может показаться несколько запутанной. По существу она извлекает спрогнозированную метку класса из шаблона `results.html`, если пользователь щелкнул на кнопке обратной связи `Correct` или `Incorrect`, и трансформирует спрогнозированное отношение в целочисленную метку класса, которая будет использоваться для обновления

классификатора посредством функции `train`, реализованной в первой части сценария `app.py`. К тому же с применением функции `sqlite_entry` в базу данных вносится новая запись, если был предоставлен отзыв о прогнозе, и в итоге визуализируется шаблон `thanks.html` для выражения благодарности пользователю за обратную связь.

## Настройка формы для рецензии

Давайте теперь взглянем на шаблон `reviewform.html`, который основывает стартовую страницу нашего приложения:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
    <link rel="stylesheet"
      href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>

    <h2>Please enter your movie review:</h2>

    {% from "_formhelpers.html" import render_field %}

    <form method=post action="/results">
      <dl>
        {{ render_field(form.moviereview, cols='30', rows='10') }}
      </dl>
      <div>
        <input type=submit value='Submit review' name='submit_btn'>
      </div>
    </form>

  </body>
</html>
```

Здесь мы просто импортируем тот же самый шаблон `_formhelpers.html`, который определили в разделе “Проверка достоверности и визуализация форм” ранее в главе. Функция `render_field` этого макроса используется для визуализации текстового поля `TextAreaField`, где пользователь может предоставить рецензию на фильм и отправить ее посредством кнопки `Submit review` (Отправить рецензию), отображаемой в нижней части страницы. Текстовое поле `TextAreaField` имеет ширину 30 столбцов и высоту 10 строк (рис. 9.12).

Рис. 9.12. Текстовое поле `TextAreaField`

## Создание шаблона страницы результатов

Наш следующий шаблон, `results.html`, выглядит чуть более интересным:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
    <link rel="stylesheet"
      href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>
    <h3>Your movie review:</h3>
    <div>{{ content }}</div>
    <h3>Prediction:</h3>
    <div>This movie review is <strong>{{ prediction }}</strong>
      (probability: {{ probability }}%).</div>
    <div id='button'>
      <form action="/"thanks" method="post">
        <input type=submit value='Correct' name='feedback_button'>
        <input type=submit value='Incorrect' name='feedback_button'>
```

```
<input type=hidden value='{{ prediction }}'  
      name='prediction'>  
<input type=hidden value='{{ content }}' name='review'>  
</form>  
</div>  
  
<div id='button'>  
  <form action="/">  
    <input type=submit value='Submit another review'>  
  </form>  
</div>  
  
</body>  
</html>
```

Первым делом мы вставляем отправленную рецензию и результаты прогноза в соответствующие поля `{{ content }}`, `{{ prediction }}` и `{{ probability }}`. Вы можете заметить, что мы второй раз применяем переменные-заполнители `{{ content }}` и `{{ prediction }}` (в данном контексте также известные как *скрытые поля*) внутри формы, которая содержит кнопки `Correct` и `Incorrect`. Это способ обойти отправку значений посредством `POST` обратно серверу для обновления классификатора и сохранения рецензии в случае щелчка пользователем на одной из двух кнопок.

Кроме того, в начале файла `results.html` мы импортировали CSS-файл (`style.css`). Его предназначение очень простое: он ограничивает ширину содержимого веб-приложения 600 пикселями и перемещает кнопки `Incorrect` и `Correct`, помеченные с помощью `div id='button'`, вниз на 20 пикселей:

```
body{  
  width:600px;  
}  
  
.button{  
  padding-top: 20px;  
}
```

Приведенный CSS-файл является просто заполнителем, поэтому вы можете свободно модифицировать его, чтобы придать веб-приложению желаемый внешний вид.

Последним HTML-файлом, который мы реализуем для нашего веб-приложения, будет шаблон `thanks.html`. Как подсказывает его имя, он просто

предоставляет пользователю сообщение с благодарностью за обратную связь через кнопку Correct или Incorrect. Вдобавок мы поместим в нижнюю часть страницы кнопки Submit another review, которая перенаправит пользователя на стартовую страницу. Ниже показано содержимое файла `thanks.html`:

```
<!doctype html>
<html>
  <head>
    <title>Movie Classification</title>
    <link rel="stylesheet"
      href="{{ url_for('static', filename='style.css') }}">
  </head>
  <body>
    <h3>Thank you for your feedback!</h3>
    <div id='button'>
      <form action="/">
        <input type=submit value='Submit another review'>
      </form>
    </div>
  </body>
</html>
```

Прежде чем переходить к следующему подразделу, посвященному развёртыванию веб-приложения на публичном веб-сервере, имеет смысл запустить его локально из терминального окна с помощью такой команды:

```
python3 app.py
```

Завершив тестирование приложения, мы должны удалить аргумент `debug=True` из вызова `app.run()` в сценарии `app.py`, как показано на рис. 9.13 (или установить `debug` в `False`).

68 69 70 71 72 73 74 75 76 77 78	<pre>         y = int(not(y))         train(review, y)         sqlite_entry(db, review, y)         return render_template('thanks.html')       if <u>name</u> == '<u>main</u>':         app.run(<u>debug=True</u>) </pre>
--	---

**Рис. 9.13.** Удаление аргумента `debug=True`



## Развертывание веб-приложения на публичном сервере

После того, как веб-приложение протестировано локально, оно готово к развертыванию на публичном веб-сервере. Мы будем использовать службу веб-хостинга PythonAnywhere, которая специализируется на хостинге веб-приложений Python, делая его исключительно простым и лишенным проблем. Вдобавок PythonAnywhere предлагает учетные записи для начинающих, которые позволяют запускать простые веб-приложения бесплатно.

### Создание учетной записи PythonAnywhere

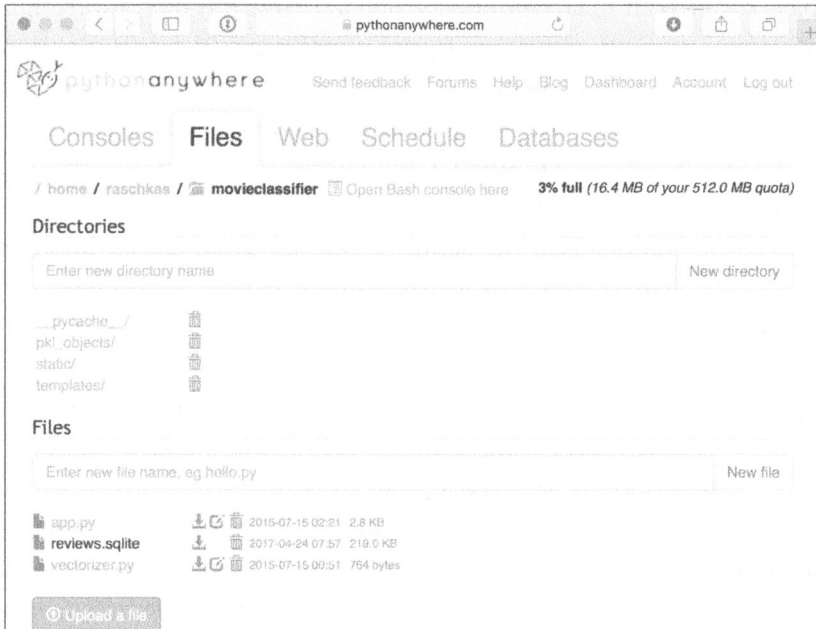
Чтобы создать учетную запись PythonAnywhere, мы заходим на веб-сайт <https://www.pythonanywhere.com/> и щелкаем на ссылке Pricing & signup (Ценообразование и оформление), расположенной в правом верхнем углу. Затем мы щелкаем на кнопке Create a Beginner account (Создать учетную запись для начинающих) и предоставляем имя пользователя, пароль и действительный адрес электронной почты. Прочитав и согласившись с условиями, мы должны заполучить в свое распоряжение новую учетную запись.

К сожалению, бесплатная учетная запись для начинающих не разрешает получать доступ к удаленному серверу через протокол SSH из терминального окна. Таким образом, для управления веб-приложением мы должны применять веб-интерфейс PythonAnywhere. Но прежде чем мы сможем загрузить локальные файлы приложения на сервер, нам необходимо создать новое веб-приложение для учетной записи PythonAnywhere. По щелчку на ссылке Dashboard (Инструментальная панель) в правом верхнем углу мы получаем доступ к панели управления. Перейдем на вкладку Web (Веб) и в ее левой части щелкнем на кнопке +Add a new web app (+Добавить новое веб-приложение), что позволит создать новое веб-приложение Python 3.7 Flask, которому мы назначаем имя `movieclassifier`.

### Загрузка файлов для приложения классификации рецензий на фильмы

Создав новое приложение для учетной записи PythonAnywhere, мы переходим на вкладку Files (Файлы), чтобы загрузить файлы из локального каталога `movieclassifier` с использованием веб-интерфейса PythonAnywhere. После загрузки файлов веб-приложения, которое было создано локально на компьютере, мы должны иметь каталог `movieclassifier` в учетной запи-

си PythonAnywhere. Он будет содержать те же самые подкаталоги и файлы, что и локальный каталог `movieclassifier` (рис. 9.14).



**Рис. 9.14.** Подкаталоги и файлы каталога `movieclassifier` в учетной записи PythonAnywhere

В заключение мы еще раз переходим на вкладку **Web** и щелкаем на кнопке `Reload <username>.pythonanywhere.com` (Перезагрузить `<имя_пользователя>.pythonanywhere.com`), чтобы распространить изменения и обновить наше веб-приложение. Теперь веб-приложение должно быть готово к работе и доступно публично через `<имя_пользователя>.pythonanywhere.com`.



### Поиск и устранение проблем

К сожалению, веб-серверы могут быть довольно восприимчивыми даже к мельчайшим проблемам, присутствующим в веб-приложении. Если вы сталкиваетесь с проблемами при выполнении веб-приложения на веб-сервере PythonAnywhere и получаете в браузере сообщения об ошибках, тогда с целью более точной диагностики проблемы можете просмотреть журналы сервера и ошибок, которые доступны на вкладке **Web** для учетной записи PythonAnywhere.

## Обновление классификатора рецензий на фильмы

Несмотря на то что наша прогнозирующая модель обновляется на лету всякий раз, когда пользователь предоставляет отзыв о результате классификации, обновления в объекте `clf` будут сбрасываться в случае аварийного отказа или перезапуска веб-сервера. Если мы перезагрузим веб-приложение, то объект `clf` будет заново инициализироваться из консервированного файла `classifier.pkl`. Одним из вариантов применения обновлений на постоянной основе было бы повторное консервирование объекта `clf` после каждого обновления. Тем не менее, с ростом количества пользователей такой подход стал бы крайне неэффективным и мог бы привести к порче консервированного файла в случае, если пользователи предоставят отзывы одновременно.

Альтернативное решение предусматривает обновление прогнозирующей модели данными обратной связи, которые собираются в базе данных SQLite. Мы можем загрузить базу данных SQLite из сервера PythonAnywhere, обновить объект `clf` локально на своем компьютере и загрузить новый консервированный файл на веб-сервер PythonAnywhere. Для обновления классификатора локально на компьютере мы создаем в каталоге `movieclassifier` сценарный файл `update.py` со следующим содержанием:

```
import pickle
import sqlite3
import numpy as np
import os

# импортировать HashingVectorizer из локального каталога
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):
    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
```

```

model.partial_fit(X_train, y, classes=classes)
results = c.fetchmany(batch_size)

conn.close()
return model

cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                   'pkl_objects',
                                   'classifier.pkl'), 'rb'))

db = os.path.join(cur_dir, 'reviews.sqlite')
clf = update_model(db_path=db, model=clf, batch_size=10000)

# Уберите символы комментария со следующих строк, если уверены в том,
# что хотите обновлять файл classifier.pkl на постоянной основе.

# pickle.dump(clf, open(os.path.join(cur_dir,
#                                   'pkl_objects', 'classifier.pkl'), 'wb'),
#             protocol=4)

```



На  
заметку!

### Получение файлов кода для классификатора рецензий на фильмы с обновленной функциональностью

Отдельный каталог, который содержит приложение классификатора рецензий на фильмы с кодом, обсуждаемым в настоящем разделе, входит в состав архива примеров кода для книги. Архив доступен для загрузки на GitHub по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition/>. Код, относящийся к текущему разделу, находится в подкаталоге `.../ch09/movieclassifier_with_update`.

Функция `update_model` будет извлекать записи из базы данных SQLite пакетами по 10 000 записей за раз, если только база данных не содержит меньшее число записей. В качестве альтернативы мы могли бы также извлекать по одной записи за раз, используя `fetchone` вместо `fetchmany`, что было бы крайне неэффективно с вычислительной точки зрения. Однако имейте в виду, что применение альтернативного метода `fetchall` может вызвать проблему при работе с крупными наборами данных, которые не умещаются в памяти компьютера или сервера.

Теперь, когда сценарий `update.py` создан, мы можем его загрузить в каталог `movieclassifier` на веб-сервере PythonAnywhere и импортировать функцию `update_model` в сценарии главного приложения `app.py`, чтобы

обновлять классификатор из базы данных SQLite при каждом перезапуске веб-приложения. Для этого нам понадобится лишь добавить в начало `app.py` строку кода, импортирующую функцию `update_model` из сценария `update.py`:

```
# импортировать функцию обновления из локального каталога
from update import update_model
```

Затем функцию `update_model` необходимо вызвать в теле главного приложения:

```
...
if __name__ == '__main__':
    clf = update_model(db_path=db,
                       model=clf,
                       batch_size=10000)
...
```

Как уже обсуждалось, модификация в предыдущем фрагменте кода обеспечивает обновление консервированного файла на веб-сервере PythonAnywhere. Тем не менее, на практике перезапускать веб-приложение придется нечасто, к тому же перед обновлением имеет смысл проверять достоверность пользовательского отзыва в базе данных SQLite, чтобы гарантировать наличие в нем ценной информации для классификатора.



На заметку!

### Создание резервных копий

В реальном приложении вы также можете сохранять резервные копии консервированного файла `classifier.pkl` через определенные промежутки времени, чтобы обеспечить защиту от разрушения файла, например, создавая версию с меткой времени до каждого обновления. Для создания резервных копий законсервированного классификатора вы можете выполнить следующее импортирование:

```
from shutil import copyfile
import time
```

Затем перед кодом, который обновляет законсервированный классификатор:

```
pickle.dump(
    clf, open(
        os.path.join(
```

```

        cur_dir, 'pkl_objects',
        'classifier.pkl'),
    'wb'),
    protocol=4)

```

вставьте приведенные ниже строки кода:

```

timestr = time.strftime("%Y%m%d-%H%M%S")
orig_path = os.path.join(
    cur_dir, 'pkl_objects', 'classifier.pkl')
backup_path = os.path.join(
    cur_dir, 'pkl_objects',
    'classifier_%s.pkl' % timestr)
copyfile(orig_path, backup_path)

```

В итоге файлы резервных копий законсервированного классификатора будут создаваться в соответствии с форматом ГодМесяцДень-Часы-МинутыСекунды, скажем, `classifier_20200420-092148.pkl`.

## Резюме

В главе вы ознакомились со многими полезными и практичными темами, которые расширяют теоретические знания в области МО. Вы узнали, как сериализовать модель после обучения и загружать ее в более поздних сценариях использования. Кроме того, были созданы база данных SQLite для рационального хранения данных и веб-приложение, которое делает доступным наш классификатор рецензий на фильмы внешнему миру.

До сих пор в книге мы раскрыли много концепций МО, установившихся приемов и моделей с учителем, предназначенных для классификации. В следующей главе мы рассмотрим еще одну подкатегорию обучения с учителем — регрессионный анализ, который позволяет вырабатывать прогнозы для выходных переменных с непрерывным масштабом в отличие от применяемых ранее категориальных меток классов в классификационных моделях.



# ПРОГНОЗИРОВАНИЕ ЗНАЧЕНИЙ НЕПРЕРЫВНЫХ ЦЕЛЕВЫХ ПЕРЕМЕННЫХ С ПОМОЩЬЮ РЕГРЕССИОННОГО АНАЛИЗА

**В** предшествующих главах вы узнали многое об основных концепциях обучения с учителем и обучили многочисленные модели, ориентированные на задачи классификации, чтобы вырабатывать прогнозы для членства в группах или для категориальных переменных. В настоящей главе мы углубимся в еще одну подкатегорию обучения с учителем — *регрессионный анализ*.

Регрессионные модели используются при прогнозировании значений целевых переменных с непрерывным масштабом, что делает их привлекательными для решения многих вопросов в науке. Они также имеют применения в индустрии, такие как выяснение взаимоотношений между переменными, оценивание тенденций или вырабатывание предсказаний. Примером может служить прогнозирование продаж компании в грядущих месяцах.



В главе будут обсуждены основные концепции регрессионных моделей и раскрыты следующие темы:

- исследование и визуализация наборов данных;
- анализ разных подходов к реализации линейных регрессионных моделей;
- обучение регрессионных моделей, устойчивых к выбросам;
- оценка регрессионных моделей и диагностика распространенных проблем;
- подгонка регрессионных моделей к нелинейным данным.

## Ведение в линейную регрессию

Цель линейной регрессии заключается в моделировании взаимоотношения между одним или множеством признаков и непрерывной целевой переменной. По контрасту с классификацией — другой подкатегорией обучения с учителем — регрессионный анализ направлен на прогнозирование выходов с непрерывным масштабом, а не категориальных меток классов.

В последующих подразделах мы представим самый базовый тип линейной регрессии, *простую линейную регрессию*, и свяжем ее с более общим многомерным случаем (линейной регрессией с множеством признаков).

### Простая линейная регрессия

Цель простой (*одномерной*) линейной регрессии — моделирование взаимоотношения между одиночным признаком (*объясняющей переменной  $x$* ) и *целью* с непрерывными значениями (*переменной ответа  $y$* ). Уравнение линейной модели с одной объясняющей переменной определяется следующим образом:

$$y = w_0 + w_1x$$

Здесь вес  $w_0$  представляет точку пересечения оси  $y$ , а  $w_1$  — весовой коэффициент объясняющей переменной. Наша цель — узнать веса линейного уравнения, чтобы описать взаимоотношение между объясняющей переменной и целевой переменной, которое затем используется для прогнозирования ответов новых объясняющих переменных, не входящих в состав обучающего набора данных.

Опираясь на определенное ранее линейное уравнение, линейную регрессию можно понимать как нахождение наилучшим образом подогнанной прямой линии, проходящей через точки образцов, как показано на рис. 10.1.

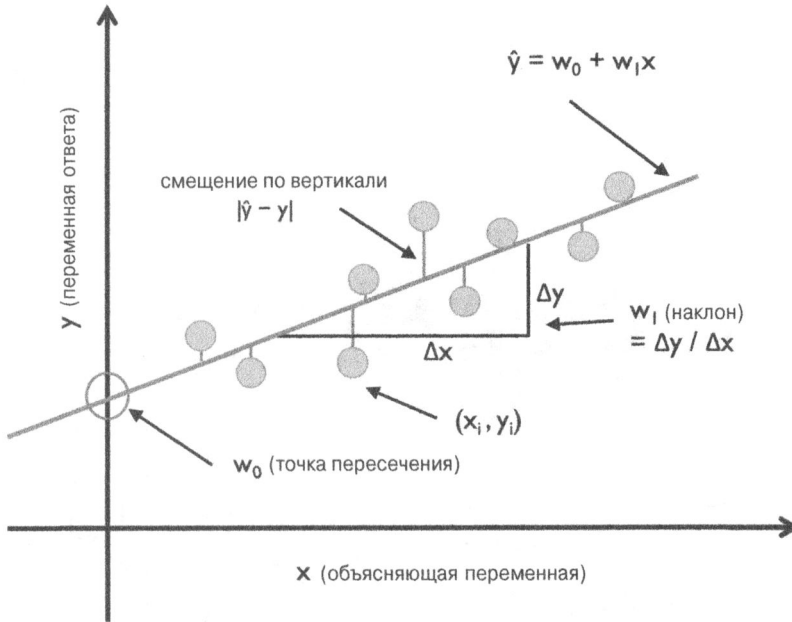


Рис. 10.1. Идея, лежащая в основе линейной регрессии

Наилучшим образом подогнанная линия также называется *линией регрессии*, а вертикальные линии от линии регрессии до обучающих образцов — это так называемые *смещения* или *остатки*, т.е. ошибки прогноза.

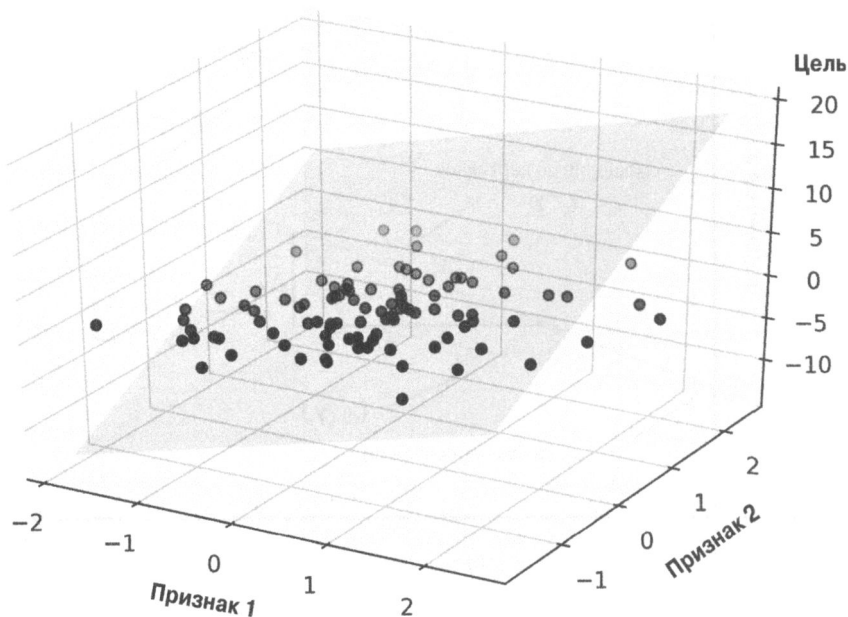
## Множественная линейная регрессия

В предыдущем разделе была представлена простая линейная регрессия как особый случай линейной регрессии с одной объясняющей переменной. Конечно, мы можем обобщить линейную регрессионную модель на множество объясняющих переменных; такой процесс называется *множественной линейной регрессией*:

$$y = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{i=0}^m w_i x_i = w^T x$$

Здесь  $w_0$  — точка пересечения оси  $y$  при  $x_0 = 1$ .

На рис. 10.2 показано, как могла бы выглядеть двумерная подогаданная гиперплоскость множественной линейной регрессионной модели с двумя признаками.



*Рис. 10.2. Двумерная подогаданная гиперплоскость множественной линейной регрессионной модели с двумя признаками*

Как можно заметить, визуализации гиперплоскостей множественной линейной регрессии на трехмерном графике рассеяния уже нелегко интерпретировать, глядя на статичные рисунки. Поскольку мы не располагаем хорошими средствами визуализации гиперплоскостей с двумя измерениями на графике рассеяния (множественной линейной регрессионной модели, подогаданной к наборам данных с тремя признаками и более), примеры и визуализации в главе будут сосредоточены на одномерном случае, когда применяется простая линейная регрессия. Однако простая и множественная линейные регрессии основаны на тех же самых концепциях и приемах оценки; код реализации, который мы будем обсуждать в главе, также совместим с обоими типами регрессионных моделей.

## Исследование набора данных Housing

Прежде чем реализовать первую линейную регрессионную модель, мы обсудим новый набор данных Housing, который содержит информацию о домах в окрестностях Бостона, собранную Д. Харрисоном и Д. Рубинфельдом в 1978 году. Набор данных Housing был сделан свободно доступным и включен в состав примеров кода для книги. Его недавно удалили из Хранилища машинного обучения UCI, но он доступен по ссылке <https://raw.githubusercontent.com/rasbt/python-machine-learning-book-3rd-edition/master/ch10/housing.data.txt> или в библиотеке `scikit-learn` ([https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/datasets/data/boston\\_house\\_prices.csv](https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/datasets/data/boston_house_prices.csv)). Как в случае любого нового набора данных, всегда полезно исследовать данные посредством простой визуализации, чтобы получить лучшее представление о том, с чем мы имеем дело.

### Загрузка набора данных Housing в объект DataFrame

В этом разделе мы загрузим набор данных Housing с использованием быстрой и универсальной функции `read_csv` из библиотеки `pandas` — рекомендованного инструмента для работы с табличными данными, хранящимися в виде неформатированного текста.

Ниже приведены краткие описания признаков 506 образцов в наборе данных, взятые из первоначального источника, который ранее находился по ссылке <https://archive.ics.uci.edu/ml/datasets/Housing>:

- CRIM — уровень преступности на душу населения по городу;
- ZN — доля жилых земельных участков с площадью более 25 000 квадратных футов (примерно 2 322.5 квадратных метров);
- INDUS — доля земель, занимаемых предприятиями нерозничной торговли, по городу;
- CHAS — фиктивная переменная реки Чарльз (равна 1, если граничит с рекой, и 0 в противном случае);
- NOX — концентрация окиси азота (в десятиллионных долях);

- RM — среднее количество комнат на дом;
- AGE — доля занимаемых собственниками домов, построенных до 1940 года;
- DIS — взвешенные расстояния до пяти центров занятости в Бостоне;
- RAD — индекс досягаемости до радиальных автомагистралей;
- TAX — полная ставка налога на недвижимость на \$10 000;
- PTRATIO — пропорция ученик-учитель по городу;
- B —  $1000(B_k - 0.63)^2$ , где  $B_k$  — доля людей афроамериканского происхождения по городу;
- LSTAT — процентная доля населения с более низким социальным статусом;
- MEDV — медианная стоимость занимаемых собственниками домов в тысячах долларов.

В оставшемся материале главы в качестве целевой переменной, т.е. переменной, для которой мы хотим вырабатывать прогнозы с применением одной или большего числа из 13 объясняющих переменных, мы будем считать цены на дома (MEDV). До проведения дальнейших исследований набора данных Housing давайте загрузим его в объект DataFrame из pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://raw.githubusercontent.com/rasbt/'
...                  'python-machine-learning-book-3rd-edition/'
...                  '/master/ch10/housing.data.txt',
...                  header=None,
...                  sep='\s+')
>>> df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
...               'NOX', 'RM', 'AGE', 'DIS', 'RAD',
...               'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
>>> df.head()
```

Чтобы проверить, успешно ли загрузился набор данных, мы отобразим из него первые пять строк (рис. 10.3).

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

Рис. 10.3. Первые пять строк набора данных Housing



На заметку!

### Получение набора данных Housing

Копия набора данных Housing (и всех других наборов данных, используемых в книге) включена в состав загружаемого архива с кодом примеров для книги. Указанные копии можно задействовать при автономной работе или в случае временной недоступности ресурса по ссылке <https://raw.githubusercontent.com/rasbt/python-machine-learning-book-3rd-edition/master/ch10/housing.data.txt>. Скажем, чтобы загрузить набор данных Housing из какого-то локального каталога, следующий оператор:

```
df = pd.read_csv(
    'https://raw.githubusercontent.com/rasbt/'
    'python-machine-learning-book-3rd-edition/'
    'master/ch10/housing.data.txt',
    header=None,
    sep='\s+')
```

понадобится заменить таким оператором:

```
df = pd.read_csv('./housing.data.txt',
    sep='\s+')
```

## Визуализация важных характеристик набора данных

Исследовательский анализ данных (*exploratory data analysis* — EDA) является важным и рекомендуемым первым шагом перед обучением модели МО. Далее в разделе мы будем применять ряд простых, но полезных инструментов из графического набора EDA, которые могут помочь визуально обнаружить присутствие выбросов, выяснить распределение данных и уловить взаимосвязи между признаками.

Первым делом мы создадим *матрицу графиков рассеяния*, которая позволит визуализировать в одном месте попарные взаимосвязи между разными признаками в наборе данных. Выводить матрицу графиков рассеяния

мы будем с использованием функции `scatterplotmatrix` из библиотеки `MLxtend` (<http://rasbt.github.io/mlxtend/>), которая содержит разнообразные удобные функции, предназначенные для приложений МО и науки о данных на языке Python.

Пакет `mlxtend` устанавливается с помощью команды `conda install mlxtend` или `pip install mlxtend`. После завершения установки пакет можно импортировать и создать матрицу графиков рассеяния, как показано ниже:

```
>>> import matplotlib.pyplot as plt
>>> from mlxtend.plotting import scatterplotmatrix
>>> cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
>>> scatterplotmatrix(df[cols].values, figsize=(10, 8),
...                   names=cols, alpha=0.5)
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 10.4 видно, что матрица графиков рассеяния снабжает нас полезной графической сводкой по взаимосвязям в наборе данных.

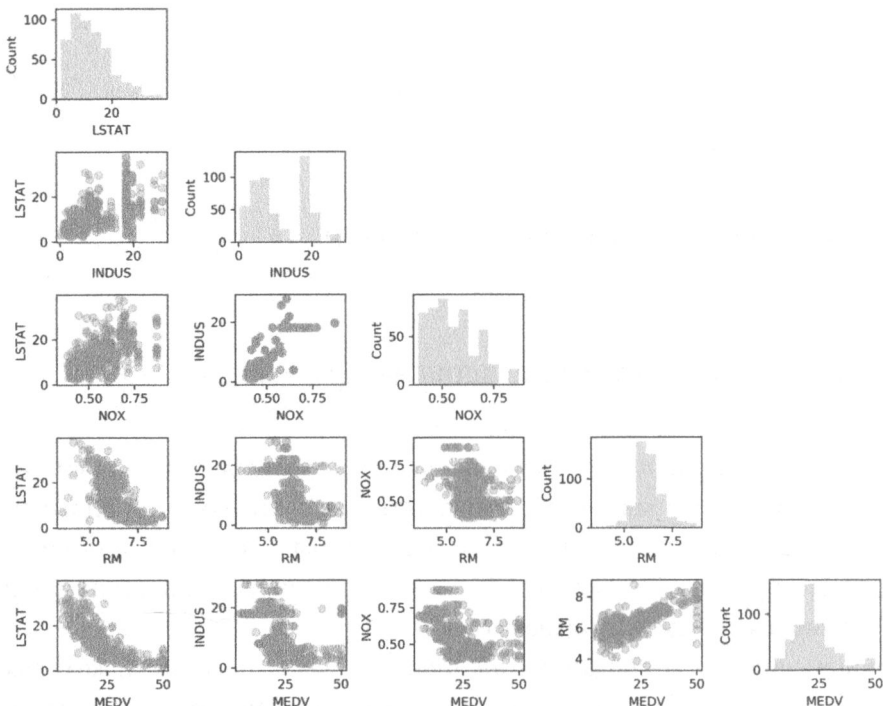


Рис. 10.4. Матрица графиков рассеяния

Из-за ограничений пространства и в интересах читабельности мы построили графики только для пяти столбцов из набора данных: LSTAT, INDUS, NOX, RM и MEDV. Тем не менее, вы можете самостоятельно создать матрицу графиков рассеяния полного объекта `DataFrame`, чтобы продолжить исследование набора данных, для чего передавать функции `scatterplotmatrix` различные имена столбцов или включить в матрицу все переменные, опустив селектор столбцов.

С помощью матрицы графиков рассеяния мы теперь можем быстро оценить на глаз, каким образом данные распределены и содержат ли они выбросы. Например, здесь видно, что есть линейная связь между RM и ценами на дома, MEDV (пятый столбец в четверной строке). Кроме того, глядя на гистограмму в нижнем правом углу матрицы графиков рассеяния, мы можем отметить, что переменная MEDV похоже имеет нормальное распределение, но содержит несколько выбросов.



На  
заметку!

### Предположение о нормальности линейной регрессии

Обратите внимание, что вопреки общему мнению обучение линейной регрессионной модели не требует наличия нормального распределения у объясняющих или целевых переменных. Предположение о нормальности является обязательным только для определенных статистических проверок и проверок гипотез, рассмотрение которых выходит за рамки настоящей книги (“Introduction to Linear Regression Analysis” (Введение в линейный регрессионный анализ), Д. Монтгомери, Э. Пек и Дж. Вайнинг, Wiley (2012 г.), с. 318–319).

## Просмотр взаимосвязей с использованием корреляционной матрицы

В предыдущем разделе мы визуализировали распределения данных для переменных набора данных Housing в форме гистограмм и графиков рассеяния. Далее мы создадим корреляционную матрицу для количественной оценки и суммирования линейных взаимосвязей между переменными. Корреляционная матрица тесно связана с ковариационной матрицей, которую мы обсуждали в разделе “Понижение размерности без учителя с помощью анализа главных компонент” главы 5. По существу мы можем трактовать корреляционную матрицу как масштабированную версию ковариационной матрицы. Фактически корреляционная матрица идентична ковариационной матрице, вычисленной на основе стандартизированных признаков.



Корреляционная матрица представляет собой квадратную матрицу, содержащую значения *коэффициента корреляции смешанного момента Пирсона* (Pearson product-moment correlation coefficient), часто сокращаемого до *r Пирсона*, который измеряет линейную зависимость между парой признаков. Коэффициент корреляции находится в диапазоне от  $-1$  до  $1$ . Два признака имеют идеальную положительную корреляцию, если  $r = 1$ , никакой корреляции, если  $r = 0$ , и идеальную отрицательную корреляцию, если  $r = -1$ . Как упоминалось ранее, коэффициент корреляции Пирсона может вычисляться просто как ковариация между двумя признаками  $x$  и  $y$  (числитель), деленная на произведение их стандартных отклонений (знаменатель):

$$r = \frac{\sum_{i=1}^n [(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)]}{\sqrt{\sum_{i=1}^n (x^{(i)} - \mu_x)^2} \sqrt{\sum_{i=1}^n (y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Здесь с помощью  $\mu$  обозначается выборочное среднее соответствующего признака,  $\sigma_{xy}$  — ковариация между признаками  $x$  и  $y$ , а  $\sigma_x$  и  $\sigma_y$  — стандартные отклонения признаков.



На заметку!

### Ковариация или корреляция для стандартизированных признаков

Мы можем показать, что ковариация между парой стандартизированных признаков фактически равна их линейному коэффициенту корреляции. Для начала стандартизируем признаки  $x$  и  $y$ , чтобы получить их меры  $z$ , которые мы обозначим как  $x'$  и  $y'$ :

$$x' = \frac{x - \mu_x}{\sigma_x}, y' = \frac{y - \mu_y}{\sigma_y}$$

Вспомните, что мы вычисляем ковариацию (генеральной совокупности) между двумя признаками следующим образом:

$$\sigma_{xy} = \frac{1}{n} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

Так как стандартизация центрирует переменную признака в нулевом среднем, мы можем вычислить ковариацию между масштабированными признаками:

$$\sigma'_{xy} = \frac{1}{n} \sum_i^n (x'_i - 0)(y'_i - 0)$$

С помощью повторной подстановки мы получаем такой результат:

$$\sigma'_{xy} = \frac{1}{n} \sum_i^n \left( \frac{x - \mu_x}{\sigma_x} \right) \left( \frac{y - \mu_y}{\sigma_y} \right)$$

$$\sigma'_{xy} = \frac{1}{n \cdot \sigma_x \sigma_y} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y)$$

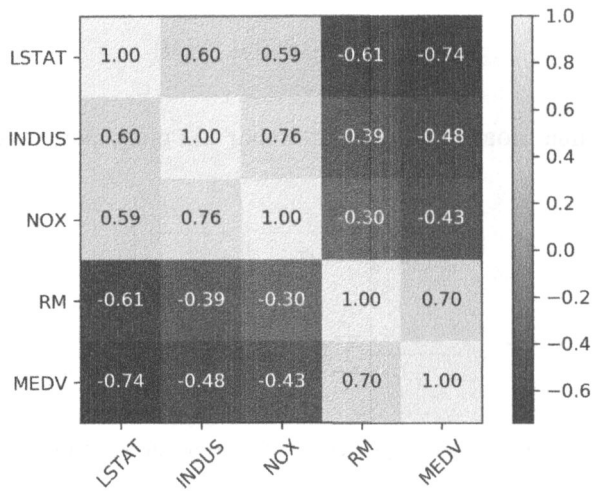
Наконец, мы можем упростить это уравнение до следующего вида:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

В приведенном ниже примере кода мы будем применять функцию `corrcoef` из NumPy на пяти столбцах признаков, которые ранее визуализировали в матрице графиков рассеяния, и также использовать функцию `heatmap` из MLxtend для вывода массива с корреляционной матрицей в виде тепловой карты:

```
>>> from mlxtend.plotting import heatmap
>>> import numpy as np
>>> cm = np.corrcoef(df[cols].values.T)
>>> hm = heatmap(cm,
...               row_names=cols,
...               column_names=cols)
>>> plt.show()
```

На результирующей тепловой карте (рис. 10.5) несложно заметить, что корреляционная матрица снабжает нас еще одной практичной итоговой диаграммой, которая способна помочь с выбором признаков на основе соответствующих линейных корреляций.



*Рис. 10.5. Корреляционная матрица, представленная в виде тепловой карты*

Для подгонки линейной регрессионной модели нас интересуют те признаки, которые имеют высокую корреляцию с целевой переменной MEDV. Взглянув на предыдущую корреляционную матрицу, мы заметим, что целевая переменная MEDV показывает наибольшую корреляцию с переменной LSTAT ( $-0.74$ ); однако, как вы можете помнить из исследования матрицы графиков рассеяния, между LSTAT и MEDV существует ясная нелинейная связь. С другой стороны, корреляция между RM и MEDV также относительно высока ( $0.70$ ). С учетом линейной связи между этими двумя переменными, наблюдаемой на графике рассеяния, RM выглядит хорошим выбором для объясняющей переменной, чтобы в следующем разделе успешно представить концепции простой линейной регрессионной модели.

## Реализация линейной регрессионной модели с использованием обычного метода наименьших квадратов

В начале главы упоминалось о том, что линейную регрессию можно понимать как получение наилучшим образом подогнанной прямой линии, проходящей через образцы наших обучающих данных. Тем не менее, мы не определяли термин “наилучшим образом подогнанная”, равно как и не обсуждали различные приемы подгонки такой модели. В последующих

подразделах мы заполним недостающие фрагменты головоломки, применяя *обычный метод наименьших квадратов* (*ordinary least squares* — OLS), иногда также называемый *линейным методом наименьших квадратов*, для оценки значений параметров прямой линейной регрессии, которые сводят к минимуму сумму квадратов расстояний по вертикали (остатков или ошибок) до обучающих образцов.

## Использование градиентного спуска для выяснения параметров регрессии

Вернемся к нашей реализации *адаптивного линейного нейрона* (*Adaline*) из главы 2. Вы должны помнить, что искусственный нейрон применяет линейную функцию активации. Мы также определяли функцию издержек  $J(\cdot)$ , которую минимизировали для выяснения весов через алгоритмы оптимизации, такие как *градиентный спуск* (GD) и *стохастический градиентный спуск* (SGD). Функцией издержек в Adaline была *сумма квадратичных ошибок* (SSE), которая идентична функции издержек, используемой для OLS:

$$J(w) = \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Здесь  $\hat{y}$  — спрогнозированное значение  $\hat{y} = w^T x$  (отметим, что член  $\frac{1}{2}$  применяется просто ради удобства для выведения правила обновления GD). По существу регрессию OLS можно понимать как Adaline без единичной ступенчатой функции, так что вместо меток классов  $-1$  и  $1$  мы получаем непрерывные целевые значения. Чтобы продемонстрировать это, давайте возьмем реализацию Adaline на основе GD из главы 2 и удалим единичную ступенчатую функцию с целью реализации нашей первой линейной регрессионной модели:

```
class LinearRegressionGD(object):
    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter
    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []
```

```

    for i in range(self.n_iter):
        output = self.net_input(X)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()
        cost = (errors**2).sum() / 2.0
        self.cost_.append(cost)
    return self

def net_input(self, X):
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    return self.net_input(X)

```



На  
заметку!

### Обновление весов с помощью градиентного спуска

Если вам необходимо освежить в памяти, каким образом обновляются веса, делая шаг в противоположном градиенту направлении, снова почитайте раздел “Адаптивные линейные нейроны и сходимость обучения” главы 2.

Чтобы увидеть регрессор `LinearRegressionGD` в действии, мы используем переменную `RM` (количество комнат) из набора данных `Housing` в качестве объясняющей переменной и обучим модель, которая сможет прогнозировать `MEDV` (цены на дома). Вдобавок мы стандартизируем переменные для лучшей сходимости алгоритма `GD`. Ниже приведен код:

```

>>> X = df[['RM']].values
>>> y = df['MEDV'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD()
>>> lr.fit(X_std, y_std)

```

Обратите внимание на обходной прием с применением `np.newaxis` и `flatten`, касающийся `y_std`. Большинство преобразователей в `scikit-learn` ожидают, что данные хранятся в двумерных массивах. В предыдущем примере кода с использованием `np.newaxis` в `y[:, np.newaxis]` к массиву добавляется новое измерение. Затем после того, как `StandardScaler` возвратит масштабированную переменную, для удобства мы преобразуем

ее в исходное представление в виде одномерного массива, применяя метод `flatten()`.

В главе 2 мы показали, что при использовании алгоритма оптимизации вроде градиентного спуска всегда имеет смысл строить график издержек как функцию количества эпох (завершенных итераций), пройденных по обучающему набору данных, чтобы проверить, сходится ли алгоритм к минимуму издержек (в рассматриваемом случае к *глобальному* минимуму издержек):

```
>>> plt.plot(range(1, lr.n_iter+1), lr.cost_)
>>> plt.ylabel('SSE')
>>> plt.xlabel('Эпоха')
>>> plt.show()
```

На рис. 10.6 видно, что алгоритм GD сходится после пятой эпохи.

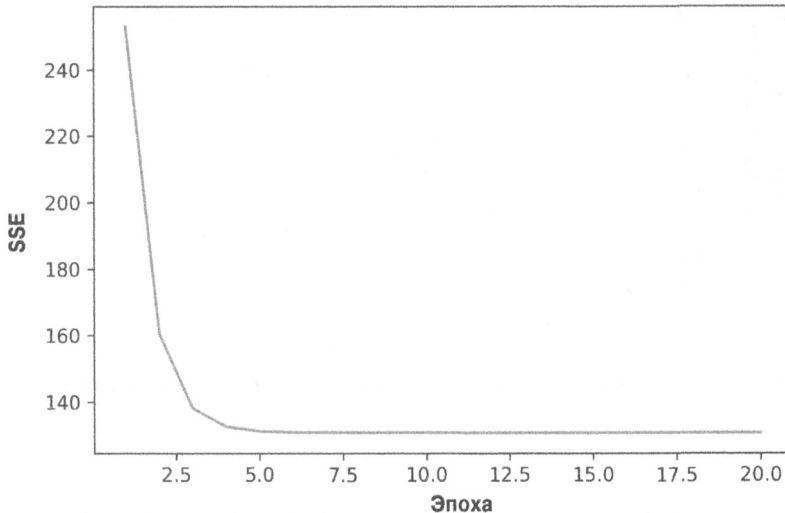


Рис. 10.6. График издержек алгоритма GD

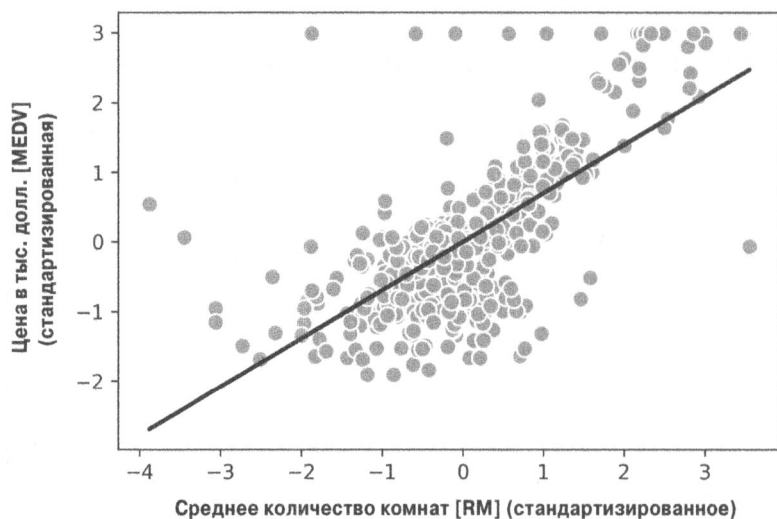
Давайте визуально выясним, насколько хорошо прямая линейной регрессии подогнана к обучающим данным. Для этого мы определим простую вспомогательную функцию, которая построит график рассеяния обучающих образцов и добавит прямую регрессии:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
...     return None
```

Теперь мы применим готовую функцию `lin_regplot` для вывода графика зависимости цены на дом от количества комнат:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Среднее количество комнат [RM] (стандартизированное)')
>>> plt.ylabel('Цена в тыс. долларов [MEDV] (стандартизированная)')
>>> plt.show()
```

На результирующем графике (рис. 10.7) можно заметить, что прямая линейной регрессии отражает общую тенденцию роста цен на дома с увеличением количества комнат.



**Рис. 10.7.** График зависимости цены на дом от количества комнат в модели, основанной на *GD*

Хотя такое наблюдение имеет смысл, данные также говорят нам о том, что во многих случаях количество комнат не особенно хорошо объясняет цены на дома. Позже в главе мы обсудим, как оценивать количественно эффективность регрессионной модели. Интересным наблюдением может служить выстраивание в линию нескольких точек данных в  $y = 3$ , что наводит на мысль об отсечении цен. В определенных приложениях также может быть важно сообщать спрогнозированные значения выходных переменных в исходном масштабе. Для масштабирования спрогнозированных цен обратно на ось “Цена в тыс. долларов” мы можем просто применить метод `inverse_transform` объекта `StandardScaler`:

```
>>> num_rooms_std = sc_x.transform(np.array([[5.0]]))
>>> price_std = lr.predict(num_rooms_std)
>>> print("Цена в тыс. долларов: %.3f" % \
...       sc_y.inverse_transform(price_std))
Цена в тыс. долларов: 10.840
```

В этом примере кода мы используем ранее обученную линейную регрессионную модель для прогнозирования цены пятикомнатного дома. Согласно нашей модели стоимость такого дома составляет \$10 840.

В качестве стороннего замечания стоит упомянуть о том, что формально мы не обязаны обновлять веса точки пересечения, если работаем со стандартизированными переменными, поскольку в таких случаях пересечение с осью  $y$  всегда равно 0. Быстро подтвердить сказанное можно, отобразив веса:

```
>>> print('Наклон: %.3f' % lr.w_[1])
Наклон: 0.695
>>> print('Точка пересечения: %.3f' % lr.w_[0])
Точка пересечения: -0.000
```

## Оценка коэффициентов регрессионной модели с помощью scikit-learn

В предыдущем разделе мы реализовали работающую модель для регрессионного анализа; однако в реальном приложении нас могут интересовать более рациональные реализации. Например, многие оценщики `scikit-learn`, предназначенные для регрессии, задействуют реализацию метода наименьших квадратов из библиотеки `SciPy` (`scipy.linalg.lstsq`), которая в свою очередь использует высоко оптимизированный код, основанный на пакете линейной алгебры `LAPACK`. Реализация линейной регрессии из библиотеки `scikit-learn` также (лучше) работает с нестандартизированными переменными, поэтому она не применяет оптимизацию на основе (стохастического) градиентного спуска, так что шаг стандартизации мы можем пропустить:

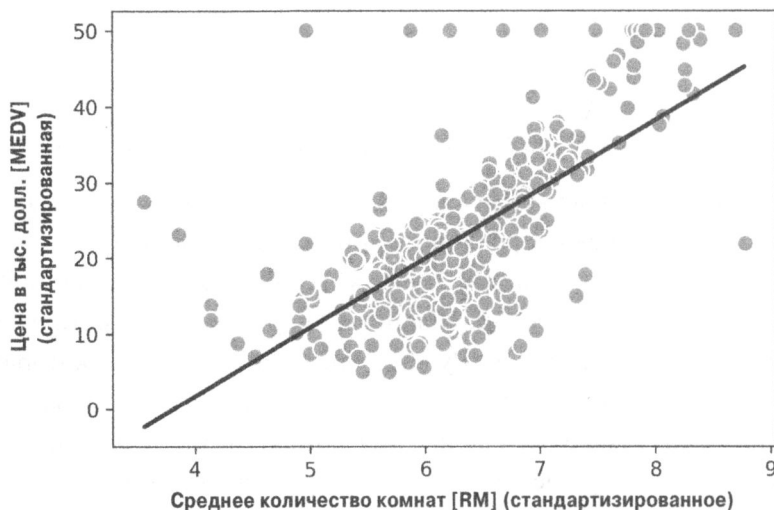
```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> y_pred = slr.predict(X)
>>> print('Наклон: %.3f' % slr.coef_[0])
Наклон: 9.102
>>> print('Точка пересечения: %.3f' % slr.intercept_)
Точка пересечения: -34.671
```



В результате выполнения приведенного выше кода мы можем заметить, что модель `LinearRegression` из `scikit-learn`, подогнанная к стандартизованным переменным `RM` и `MEDV`, выдала другие коэффициенты модели, т.к. признаки не были стандартизованы. Но когда мы сравним их с нашей реализацией на основе `GD`, построив график зависимости `MEDV` от `RM`, то сможем качественно увидеть, что она подобным образом хорошо подгоняется к данным:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Среднее количество комнат [RM] (стандартизированное)')
>>> plt.ylabel('Цена в тыс. долларов [MEDV] (стандартизированная)')
>>> plt.show()
```

Скажем, мы можем видеть, что общий результат выглядит идентичным нашей реализации на основе `GD` (рис. 10.8).



**Рис. 10.8.** График зависимости цены на дом от количества комнат в модели `LinearRegression`



На заметку!

### Аналитические решения задач линейной регрессии

В качестве альтернативы применению библиотек МО обычный метод наименьших квадратов можно решить в аналитическом виде, используя систему линейных уравнений, как показано в большинстве вводных учебниках по статистике:

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

Вот как реализовать решение на Python:

```
# добавление вектора-столбца единиц
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print('Наклон: %.3f' % w[1])
Наклон: 9.102
>>> print('Точка пересечения: %.3f' % w[0])
Точка пересечения: -34.671
```

Преимущество такого метода в том, что он гарантирует нахождение оптимального решения аналитически. Тем не менее, при работе с очень крупными базами данных получение обратной матрицы в этой формуле (иногда называемой *нормальным уравнением*) может оказаться слишком затратным в вычислительном плане или же матрица, содержащая обучающие образцы, может быть вырожденной (необращаемой), из-за чего в ряде случаев мы отдаем предпочтение итерационным методам.

Если вас интересуют дополнительные сведения о том, как получать нормальные уравнения, тогда ознакомьтесь с главой “The Classical Linear Regression Model” (Классическая линейная регрессионная модель) из курса лекций доктора Стивена Поллока, которые он читает в Лестерском университете (<http://www.le.ac.uk/users/dsgpl/COURSES/MESOMET/ECMETXT/06mesmet.pdf>).

Кроме того, если вы хотите сравнить решения на основе линейной регрессии, полученные посредством GD, SGD, нормальных уравнений, QR-разложения и сингулярного разложения, тогда можете воспользоваться классом `LinearRegression`, реализованным в библиотеке `MLxtend` ([http://rasbt.github.io/mlxtend/user\\_guide/regressor/LinearRegression/](http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/)), который позволяет переключаться между указанными вариантами. Еще одной великолепной библиотекой, рекомендуемой для работы с регрессионными моделями, является `Statsmodels`, которая реализует более сложные модели на основе линейной регрессии, как демонстрируется по ссылке <https://www.statsmodels.org/stable/examples/index.html#regression>.

## Подгонка надежной регрессионной модели с использованием RANSAC

Присутствие выбросов может сильно повлиять на линейные регрессионные модели. В некоторых ситуациях очень маленький поднабор данных способен оказывать большое воздействие на оценочные коэффициенты модели. Для обнаружения выбросов предусмотрено много статистических проверок, обсуждение которых выходит за рамки настоящей книги. Однако удаление выбросов всегда требует нашего суждения как специалистов в науке о данных, знающих предметную область, с которой мы имеем дело.

В качестве альтернативы устранению выбросов мы рассмотрим надежный метод регрессии, применяющий алгоритм *RANSAC* (*RAN*dом *S*ample *C*onsensus — соглашение на основе случайных выборок), который подгоняет модель к поднабору данных, называемых *не-выбросами* (*inlier*).

Итерационный алгоритм RANSAC можно подытожить следующим образом.

1. Выбрать случайное количество образцов, которые будут служить не-выбросами, и выполнить подгонку модели.
2. Проверить все остальные точки данных на подогнанной модели и добавить те точки, которые попадают внутрь предоставленного пользователем порога не-выбросов.
3. Повторно подогнать модель, используя все не-выбросы.
4. Оценить ошибку подогнанной модели относительно не-выбросов.
5. Закончить алгоритм, если эффективность достигла определенного пользователем порога или прошло фиксированное число итераций; в противном случае вернуться к шагу 1.

Давайте применим линейную модель в сочетании с алгоритмом RANSAC, как реализовано в классе `RANSACRegressor` библиотеки `scikit-learn`:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(LinearRegression(),
...                           max_trials=100,
...                           min_samples=50,
...                           loss='absolute_loss',
...                           residual_threshold=5.0,
...                           random_state=0)
>>> ransac.fit(X, y)
```

Мы указываем 100 для максимального количества итераций RANSAC Regressor и с использованием `min_samples=50` устанавливаем минимальное число случайно выбранных обучающих образцов в 50. Благодаря указанию аргумента `'absolute_loss'` для параметра `residual_metric` алгоритм вычисляет абсолютное расстояние по вертикали между подогнутой прямой и обучающими образцами.

Устанавливая параметр `residual_threshold` в 5.0, мы разрешаем включать в набор не-выбросов только образцы, у которых расстояние по вертикали до подогнутой прямой находится в пределах 5 единиц расстояния, что хорошо работает на этом конкретном наборе данных.

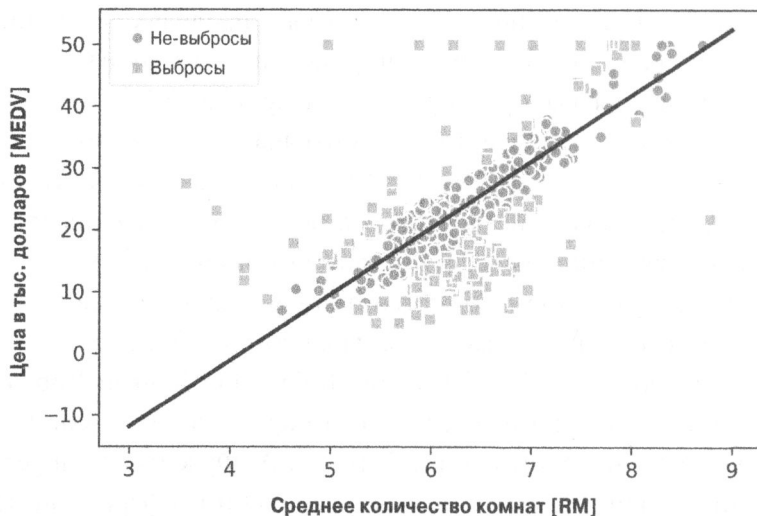
По умолчанию для выбора порога не-выбросов библиотека `scikit-learn` применяет оценку *MAD*, где *MAD* обозначает *медианное абсолютное отклонение* (*median absolute deviation*) целевых значений  $y$ . Тем не менее, выбор подходящего значения для порога не-выбросов специфичен для задачи, что является одним из недостатков алгоритма RANSAC. За последние годы было разработано множество разных подходов к автоматическому выбору хорошего порога не-выбросов. Подробные обсуждения можно найти в работе “Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting” (Автоматическая оценка порога не-выбросов в надежной подгонке множественных структур), Р. Толдо, А. Фузелло, Springer (2009 г.), представленной в книге с итогами международной конференции по анализу и обработке изображений (“Image Analysis and Processing — ICIAP 2009”, с. 123–131).

После подгонки модели RANSAC давайте получим не-выбросы и выбросы из подогнутой линейной регрессионной модели на основе RANSAC и добавим их к графику с линейной подгонкой:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='steelblue', edgecolor='white',
...             marker='o', label='Не-выбросы')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='limegreen', edgecolor='white',
...             marker='s', label='Выбросы')
>>> plt.plot(line_X, line_y_ransac, color='black', lw=2)
```

```
>>> plt.xlabel('Среднее количество комнат [RM]')
>>> plt.ylabel('Цена в тыс. долларов [MEDV]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

На результирующем графике (рис. 10.9) видно, что линейная регрессионная модель была подогнана на обнаруженном наборе не-выбросов, показанных в виде кружков.



**Рис. 10.9.** Результат подгонки линейной регрессионной модели на обнаруженном наборе не-выбросов

Если вывести наклон и точку пересечения модели, выполнив приведенный ниже код, то можно заметить, что прямая линейной регрессии будет слегка отличаться от подгонки, которую мы получили в предыдущем разделе, где алгоритм RANSAC не использовался:

```
>>> print('Slope: %.3f' % ransac.estimator_.coef_[0])
Slope: 10.735
>>> print('Intercept: %.3f' % ransac.estimator_.intercept_)
Intercept: -44.089
```

За счет применения RANSAC удалось уменьшить потенциальное влияние выбросов в этом наборе данных, но мы не знаем, покажет ли продемонстрированный подход положительный результат на не встречавшихся ранее данных. Таким образом, в следующем разделе мы рассмотрим другие подходы

к оценке регрессионной модели, которые представляют собой критически важную часть построения систем для прогнозирующего моделирования.

## Оценка эффективности линейных регрессионных моделей

В предыдущем разделе мы выяснили, как подгонять регрессионную модель к обучающим данным. Однако из предшествующих глав вы знаете, что критически важно испытать модель на данных, которые ей не встречались во время обучения, чтобы получить менее смещенную оценку эффективности модели.

Как обсуждалось в главе 6, мы хотим разбить набор данных на отдельные обучающий и испытательный поднаборы, где первый используется для подгонки модели, а второй — для оценки ее эффективности обобщения на не встречавшиеся ранее данные. Вместо продолжения работы с простой регрессионной моделью мы будем применять все переменные в наборе данных и обучать множественную регрессионную модель:

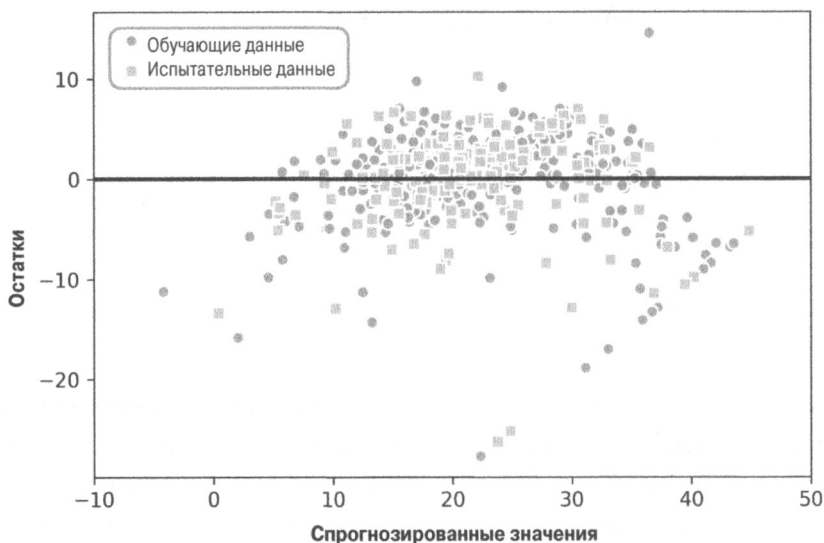
```
>>> from sklearn.model_selection import train_test_split
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=0)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Учитывая, что модель использует множество объясняющих переменных, на двумерном графике мы не можем визуализировать прямую (точнее гиперплоскость) линейной регрессии, но в состоянии вывести зависимость остатков (разностей или расстояний по вертикали между действительными и спрогнозированными значениями) от спрогнозированных значений для диагностирования нашей регрессионной модели. *Графики остатков (residual plot)* являются широко применяемым графическим инструментом для диагностирования регрессионных моделей. Они могут помочь выявить нелинейность и выбросы, а также проверить, распределены ли ошибки по случайному закону.

Используя следующий код, мы строим график остатков, где просто вычитаем настоящие целевые переменные из спрогнозированных ответов:

```
>>> plt.scatter(y_train_pred, y_train_pred - y_train,
...             c='steelblue', marker='o', edgecolor='white',
...             label='Обучающие данные')
>>> plt.scatter(y_test_pred, y_test_pred - y_test,
...             c='limegreen', marker='s', edgecolor='white',
...             label='Испытательные данные')
>>> plt.xlabel('Спрогнозированные значения')
>>> plt.ylabel('Остатки')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, color='black', lw=2)
>>> plt.xlim([-10, 50])
>>> plt.show()
```

В результате выполнения кода мы должны получить график остатков с линией, проходящей вдоль оси  $x$  через начало отсчета (рис. 10.10).



**Рис. 10.10.** График остатков

В случае идеального прогноза остатки были бы в точности нулевыми, что вряд ли когда-нибудь встретится в более реалистичных и практических приложениях. Тем не менее, для хорошей регрессионной модели мы могли бы ожидать, что ошибки распределены по случайному закону, а остатки должны быть случайным образом разбросаны вокруг средней линии. Если

мы видим на графике остатков повторяющиеся шаблоны, то это значит, что наша модель неспособна захватывать какую-то объясняющую информацию, которая просочилась в остатки, как немного просматривается на рис. 10.10. Кроме того, графики остатков можно также применять для выявления выбросов, которые представляются точками с большим отклонением от средней линии.

Еще одной полезной количественной мерой эффективности модели является так называемая *среднеквадратическая ошибка* (*mean squared error* — *MSE*), которая представляет собой просто усредненную величину издержек SSE, сведенную нами к минимуму с целью подгонки линейной регрессионной модели. Мера MSE удобна для сравнения разных регрессионных моделей или для настройки их параметров посредством решетчатого поиска и перекрестной проверки, т.к. она нормализует SSE по размеру выборки:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2$$

Давайте вычислим меру MSE при обучении и испытании:

```
>>> from sklearn.metrics import mean_squared_error
>>> print('MSE при обучении: %.3f, при испытании: %.3f' % (
...     mean_squared_error(y_train, y_train_pred),
...     mean_squared_error(y_test, y_test_pred)))
MSE при обучении: 19.958, при испытании: 27.196
```

Как видите, мера MSE на обучающем наборе составляет 19.96, а MSE на испытательном наборе гораздо больше — 27.20; это сигнализирует о том, что в рассматриваемом случае модель переобучается обучающими данными. Однако имейте в виду, что мера MSE является неограниченной в отличие от правильности классификации, например. Другими словами, интерпретация меры MSE зависит от масштабирования набора данных и признаков. Скажем, если бы цены на дома были представлены как числа, кратные 1000 (с суффиксом K), тогда та же самая модель выдавала бы более низкую меру MSE в сравнении с моделью, которая работает с немасштабированными признаками. В качестве дальнейшей иллюстрации этого момента:  $(\$10K - \$15K)^2 < (\$10\,000 - \$15\,000)^2$ .

Таким образом, чтобы лучше интерпретировать эффективность модели, временами более полезно сообщать *коэффициент детерминации*  $R^2$  (*coefficient of determination*), который можно понимать как стандартизи-



ванную версию MSE. Говоря иначе,  $R^2$  представляет собой долю дисперсии ответа, которая захватывается моделью. Вот как определяется значение  $R^2$ :

$$R^2 = 1 - \frac{SSE}{SST}$$

Здесь SSE — сумма квадратичных ошибок, а SST — полная сумма квадратов:

$$SST = \sum_{i=1}^n (y^{(i)} - \mu_y)^2$$

Другими словами, SST представляет собой просто дисперсию ответа.

Давайте быстро покажем, что на самом деле  $R^2$  — всего лишь масштабированная версия MSE:

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} = \\ &= \frac{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \mu_y)^2} = \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

Для обучающего набора данных значение  $R^2$  ограничивается диапазоном от 0 до 1, но может стать отрицательным для испытательного набора. Если  $R^2 = 1$ , тогда модель идеально подогнана к данным и соответственно  $MSE = 0$ .

При оценке на обучающих данных значение  $R^2$  модели равно 0.765, что не выглядит слишком плохо. Однако  $R^2$  на испытательном наборе данных составляет лишь 0.673. Мы можем вычислить значения  $R^2$  с помощью такого кода:

```
>>> from sklearn.metrics import r2_score
>>> print('R^2 при обучении: %.3f, при испытании: %.3f' %
...       (r2_score(y_train, y_train_pred),
...       r2_score(y_test, y_test_pred)))
R^2 при обучении: 0.765, при испытании: 0.673
```

## Использование регуляризированных методов для регрессии

В главе 3 мы обсуждали регуляризацию — подход к решению проблемы переобучения, который предусматривает добавление дополнительной информации и тем самым сокращение значений параметров, чтобы порождать штраф за сложность. Самые популярные подходы в отношении регуляризированной линейной регрессии включают *гребневую регрессию* (*ridge regression*), *регрессию методом наименьшего абсолютного сокращения и выбора* (*least absolute shrinkage and selection operator — LASSO*) и *эластичную сеть* (*elastic net*).

Гребневая регрессия представляет собой штрафруемую с помощью L2 модель, где мы просто добавляем к функции издержек, основанной на методе наименьших квадратов, квадратичную сумму весов:

$$J(w)_{\text{Гребневая}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_2^2$$

Здесь:

$$\text{L2: } \lambda \|w\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

Повышая значение гиперпараметра  $\lambda$ , мы увеличиваем силу регуляризации и посредством этого сокращаем веса модели. Обратите внимание, что мы не регуляризируем член для точки пересечения  $w_0$ .

Альтернативным подходом, который может привести к разреженным моделям, является LASSO. В зависимости от силы регуляризации определенные веса могут стать нулевыми, что также делает LASSO полезным в качестве приема выбора признаков с учителем:

$$J(w)_{\text{LASSO}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|w\|_1$$

Здесь штраф L1 для LASSO определяется как сумма абсолютных величин весов модели:

$$\text{L1: } \lambda \|w\|_1 = \lambda \sum_{j=1}^m |w_j|$$

Тем не менее, ограничение метода LASSO заключается в том, что он выбирает самое большее  $n$  признаков, если  $m > n$ , где  $n$  — количество обучающих образцов. В определенных случаях применения выбора признаков это может оказаться нежелательным. Однако на практике такая характеристика LASSO часто будет преимуществом, потому что она избегает насыщенных моделей. Насыщение модели происходит, если количество обучающих экземпляров равно количеству признаков, что является формой чрезмерной параметризации. Как следствие, насыщенная модель всегда способна идеально подгоняться к обучающим данным, но представляет собой просто разновидность интерполяции и, следовательно, не должна хорошо обобщаться.

Компромиссом между гребневой регрессией и методом LASSO считается эластичная сеть, которая предусматривает штраф L1 для порождения разреженности и штраф L2, так что ее можно использовать для выбора более  $n$  признаков, когда  $m > n$ :

$$J(w)_{\text{Эластичная сеть}} = \sum_{i=1}^n \left( y^{(i)} - \hat{y}^{(i)} \right)^2 + \lambda_1 \sum_{j=1}^m w_j^2 + \lambda_2 \sum_{j=1}^m |w_j|$$

Все упомянутые регрессионные модели с регуляризацией доступны в библиотеке `skikit-learn`. Они применяются подобно обычной регрессионной модели, но с указанием в параметре  $\lambda$  силы регуляризации, оптимизированной с помощью перекрестной проверки по  $k$  блокам, например.

Вот как можно инициализировать модель на основе гребневой регрессии:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Обратите внимание, что сила регуляризации управляется параметром `alpha`, который похож на параметр  $\lambda$ . Аналогично мы можем инициализировать регрессор LASSO посредством подмодуля `linear_model`:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Наконец, реализация `ElasticNet` позволяет варьировать соотношение L1 к L2:

```
>>> from sklearn.linear_model import ElasticNet
>>> elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Скажем, если мы установим `l1_ratio` в 1.0, то регрессор `ElasticNet` будет эквивалентен регрессору `LASSO`. Более детальные сведения о различных реализациях линейной регрессии ищите в документации по ссылке [https://scikit-learn.org/stable/modules/linear\\_model.html](https://scikit-learn.org/stable/modules/linear_model.html).

## Превращение линейной регрессионной модели в криволинейную — полиномиальная регрессия

В предшествующих разделах мы предполагали наличие линейной связи между объясняющими переменными и переменными ответов. При нарушении допущения о линейности один из подходов предусматривает использование полиномиальной регрессионной модели путем добавления полиномиальных членов:

$$y = w_0 + w_1x + w_2x^2 + \dots + w_dx^d$$

Здесь  $d$  обозначает степень полинома. Хотя мы можем применять полиномиальную регрессию для моделирования нелинейной связи, она по-прежнему считается множественной линейной регрессионной моделью из-за коэффициентов линейной регрессии  $w$ . В последующих подразделах мы покажем, как без труда добавлять такие полиномиальные члены к существующему набору данных и подгонять полиномиальную регрессионную модель.

### Добавление полиномиальных членов с использованием `scikit-learn`

Далее мы выясним, каким образом применять класс преобразователя `PolynomialFeatures` из `scikit-learn` для добавления квадратичного члена ( $d=2$ ) в простую задачу регрессии с одной объясняющей переменной. Затем мы сравним полиномиальную подгонку с линейной, следуя описанным ниже шагам.

#### 1. Добавить полиномиальный член второй степени:

```
from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([ 258.0, 270.0, 294.0, 320.0, 342.0,
...               368.0, 396.0, 446.0, 480.0, 586.0])\
...          [:, np.newaxis]
>>> y = np.array([ 236.4, 234.4, 252.8, 298.6, 314.2,
...               342.2, 360.8, 368.0, 391.2, 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Выполнить подгонку простой линейной регрессионной модели для сравнения:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[: , np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

3. Выполнить подгонку множественной регрессионной модели на трансформированных признаках для полиномиальной регрессии:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Построить график с результатами:

```
>>> plt.scatter(X, y, label='обучающие точки')
>>> plt.plot(X_fit, y_lin_fit,
...         label='линейная подгонка', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...         label='квадратичная подгонка')
>>> plt.xlabel('Объясняющая переменная')
>>> plt.ylabel('Спрогнозированные или известные целевые значения')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 10.11) видно, что полиномиальная подгонка улавливает связь между переменной ответа и объясняющей переменной гораздо лучше, чем линейная подгонка.

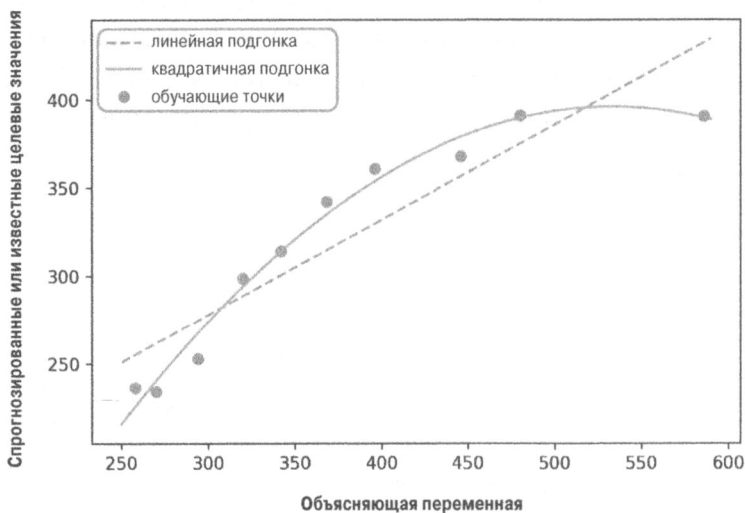


Рис. 10.11. Сравнение линейной и полиномиальной регрессии

Далее мы рассчитаем метрики оценки MSE и  $R^2$ :

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> print('MSE при обучении линейной модели: %.3f, квадратичной
модели: %.3f' % (
...     mean_squared_error(y, y_lin_pred),
...     mean_squared_error(y, y_quad_pred)))
MSE при обучении линейной модели: 569.780, квадратичной модели:
61.330
>>> print('R^2 при обучении линейной модели: %.3f, квадратичной
модели: %.3f' % (
...     r2_score(y, y_lin_pred),
...     r2_score(y, y_quad_pred)))
R^2 при обучении линейной модели: 0.832, квадратичной модели:
0.982
```

После выполнения кода можно заметить, что MSE уменьшается от приблизительно 570 (линейная подгонка) до примерно 61 (квадратичная подгонка). К тому же коэффициент детерминации отражает более тесную подгонку квадратичной модели ( $R^2 = 0.982$ ) по сравнению с линейной моделью ( $R^2 = 0.832$ ) в этой конкретной игровой задаче.

## Моделирование нелинейных связей в наборе данных Housing

В предыдущем подразделе вы узнали, как строить полиномиальные признаки для подгонки к нелинейным связям в игровой задаче; давайте теперь рассмотрим более конкретный пример и применим такие концепции к набору данных Housing. Выполнив следующий код, мы смоделируем связь между ценами на дома и LSTAT (процентная доля населения с более низким социальным статусом) с использованием полиномов второго (квадратичного) и третьего (кубического) порядков и сравним ее с линейной подгонкой:

```
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> regr = LinearRegression()
>>> # создать квадратичные признаки
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)
```

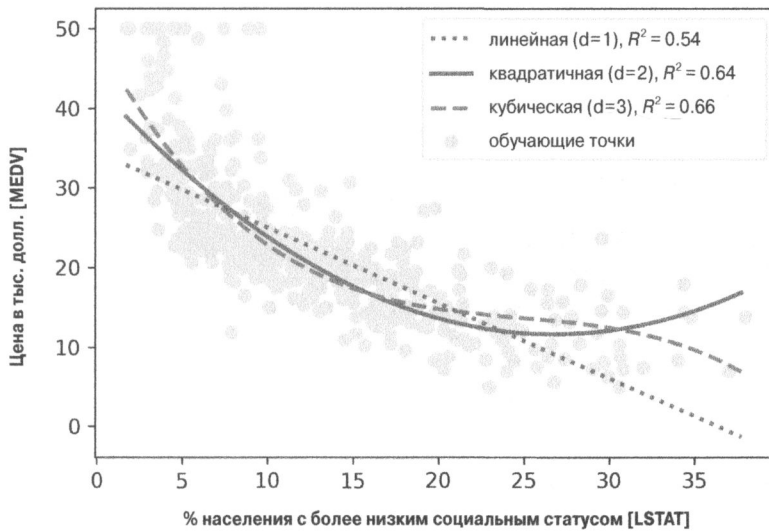
```

>>> # выполнить подгонку к признакам
>>> X_fit = np.arange(X.min(), X.max(), 1)[: , np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))
>>> # вывести результаты
>>> plt.scatter(X, y, label='обучающие точки', color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label='линейная (d=1), $R^2=%.2f$' % linear_r2,
...          color='blue',
...          lw=2,
...          linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...          label='квадратичная (d=2), $R^2=%.2f$' % quadratic_r2,
...          color='red',
...          lw=2,
...          linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
...          label='кубическая (d=3), $R^2=%.2f$' % cubic_r2,
...          color='green',
...          lw=2,
...          linestyle='--')
>>> plt.xlabel('%населения с более низким социальным статусом [LSTAT]')
>>> plt.ylabel('Цена в тыс. долларов [MEDV]')
>>> plt.legend(loc='upper right')
>>> plt.show()

```

На рис. 10.12 показан результирующий график.

Как можно заметить, кубическая подгонка улавливает связь между ценами на дома и LSTAT лучше, чем линейная и квадратичная. Однако важно помнить о том, что добавление все большего и большего количества признаков увеличивает сложность модели и, следовательно, повышает риск переобучения.



**Рис. 10.12.** График зависимости цены на дом от процентной доли населения с более низким социальным статусом

Таким образом, на практике рекомендуется оценивать эффективность модели на отдельном испытательном наборе данных, чтобы получить оценку эффективности обобщения.

Вдобавок полиномиальные признаки не всегда являются наилучшим выбором для моделирования нелинейных связей. Например, при наличии некоторого опыта или интуиции, просто взглянув на график рассеяния MEDV–LSTAT, можно выдвинуть гипотезу о том, что логарифмическое преобразование переменной признака LSTAT и квадратный корень MEDV могут спроецировать данные на линейное пространство признаков, подходящее для подгонки с помощью линейной регрессии. Скажем, по нашим ощущениям связь между упомянутыми двумя переменными выглядит очень похожей на экспоненциальную функцию:

$$f(x) = e^{-x}$$

Поскольку натуральный логарифм экспоненциальной функции представляет собой прямую линию, мы предполагаем, что такое логарифмическое преобразование может здесь пригодиться:

$$\log(f(x)) = -x$$

Давайте проверим эту гипотезу, выполнив следующий код:



```

>>> # трансформировать признаки
>>> X_log = np.log(X)
>>> y_sqrt = np.sqrt(y)
>>>
>>> # выполнить подгонку к признакам
>>> X_fit = np.arange(X_log.min()-1,
...                   X_log.max()+1, 1)[: , np.newaxis]
>>> regr = regr.fit(X_log, y_sqrt)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y_sqrt, regr.predict(X_log))
>>>
>>> # вывести результаты
>>> plt.scatter(X_log, y_sqrt,
...             label='обучающие точки',
...             color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...           label='линейная (d=1), $R^2$=%.2f$' % linear_r2,
...           color='blue',
...           lw=2)
>>> plt.xlabel('log(%населения с более низким социальным статусом
[LSTAT])')
>>> plt.ylabel('$\sqrt{\text{Цена \; в \; тыс. долларов \; [MEDV]}}$')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()

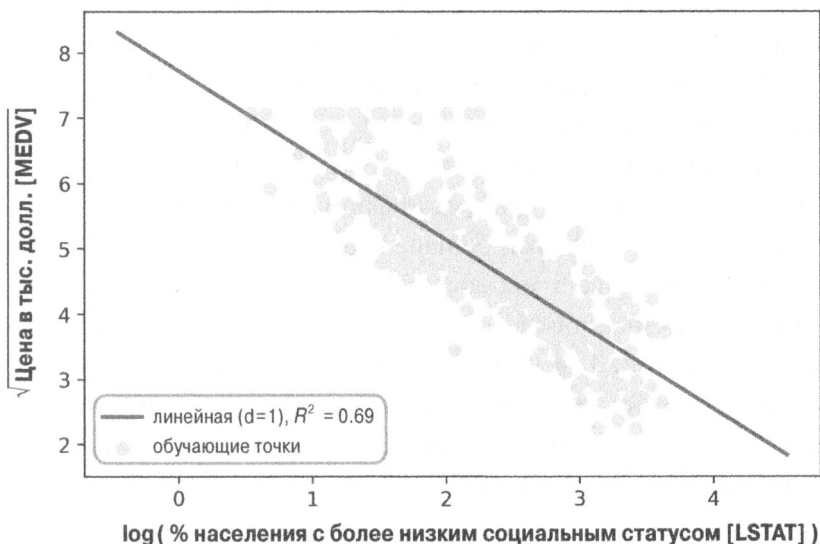
```

После преобразования объясняющей переменной в логарифмическое пространство и взятия квадратного корня из целевой переменной нам удалось уловить связь между двумя переменными с помощью прямой линейной регрессии, которая, кажется, лучше подгоняется к данным ( $R^2 = 0.69$ ), чем все рассмотренные ранее полиномиальные трансформации признаков (рис. 10.13).

## Обработка нелинейных связей с использованием случайных лесов

В настоящем разделе мы ознакомимся с регрессией на основе *случайных лесов*, которая концептуально отличается от регрессионных моделей, описанных выше в главе. Случайный лес, представляющий собой ансамбль из множества *деревьев принятия решений*, можно понимать как сумму кусочно-линейных функций в противоположность глобальным линейным и полиномиальным регрессионным моделям, которые мы обсуждали ранее. Другими

словами, посредством алгоритма дерева принятия решений мы подразделяем входное пространство на меньшие области, которые становятся более управляемыми.



*Рис. 10.13. График зависимости цены на дом от процентной доли населения с более низким социальным статусом после применения преобразования*

## Регрессия на основе дерева принятия решений

Преимущество алгоритма дерева принятия решений заключается в том, что при работе с нелинейными данными он не требует каких-либо трансформаций признаков, т.к. деревья принятия решений анализируют по одному признаку за раз, а не принимают во внимание взвешенные комбинации. (Аналогичным образом для деревьев принятия решений не требуется нормализация или стандартизация признаков.) Благодаря главе 3 вы знаете, что дерево принятия решений выращивается за счет последовательного разделения его узлов до тех пор, пока листовые узлы не станут чистыми или не будет удовлетворен критерий остановки. Когда мы применяем деревья принятия решений для классификации, то определяем энтропию как меру загрязненности, чтобы выяснить, какое разделение признака доводит до максимума *прирост информации* ( $IG$ ), который в случае двоичного разделения может быть записан так:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{\text{левый}}}{N_p} I(D_{\text{левый}}) - \frac{N_{\text{правый}}}{N_p} I(D_{\text{правый}})$$

Здесь  $x_i$  — признак, подлежащий разделению,  $N_p$  — количество обучающих образцов в родительском узле,  $I$  — функция загрязненности,  $D_p$  — поднабор обучающих образцов в родительском узле, а  $D_{\text{левый}}$  и  $D_{\text{правый}}$  — поднаборы обучающих образцов в левом и правом дочерних узлах после разделения. Вспомните, что наша цель — отыскать разделение признака, которое доводит до максимума прирост информации; другими словами, мы хотим найти разделение признака, больше всего сокращающее загрязненности в дочерних узлах. В главе 3 мы обсуждали загрязненность Джини и энтропию как меры загрязненности, и обе они являются критериями, пригодными для классификации. Тем не менее, чтобы использовать дерево принятия решений для регрессии, нам понадобится метрика загрязненности, которая применима для непрерывных переменных, а потому взамен мы определяем меру загрязненности узла  $t$  как MSE:

$$I(t) = \text{MSE}(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2$$

Здесь  $N_t$  — количество обучающих образцов в узле  $t$ ,  $D_t$  — поднабор обучающих образцов в узле  $t$ ,  $y^{(i)}$  — настоящее целевое значение и  $\hat{y}_t$  — спрогнозированное целевое значение (выборочное среднее):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

В контексте регрессии на основе дерева принятия решений на MSE часто ссылаются как на *внутриузловую дисперсию*, из-за чего критерий разделения лучше известен как *понижение дисперсии*. Чтобы посмотреть, на что похожа линия подгонки, давайте воспользуемся классом `DecisionTreeRegressor`, реализованным в `scikit-learn`, для моделирования нелинейной связи между переменными MEDV и LSTAT:

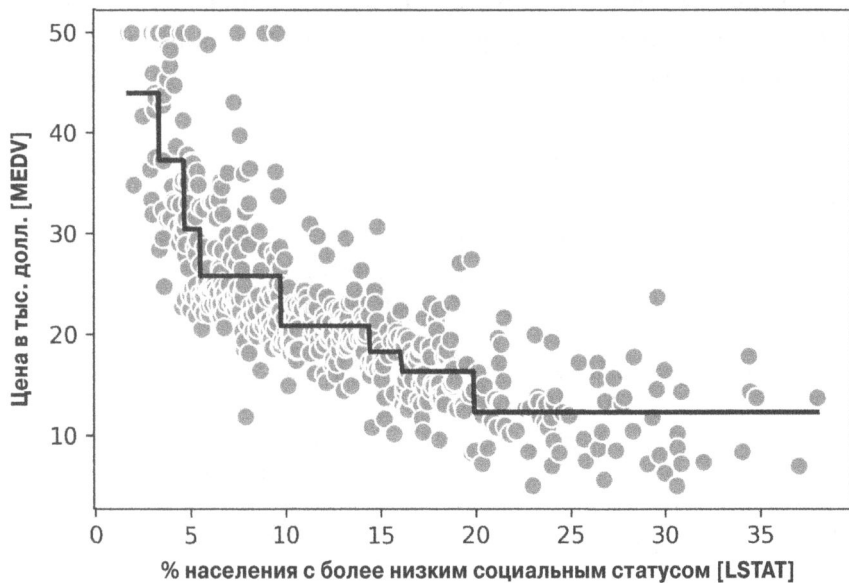
```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['LSTAT']].values
>>> y = df['MEDV'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
```

```

>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('% населения с более низким социальным статусом [LSTAT]')
>>> plt.ylabel('Цена в тыс. долларов [MEDV]')
>>> plt.show()

```

На результирующем графике (рис. 10.14) видно, что дерево принятия решений выявляет основную тенденцию в данных. Однако ограничение этой модели в том, что она не улавливает непрерывность и дифференцируемость желательного прогноза. Кроме того, нам необходимо проявлять осмотрительность при выборе подходящего значения для глубины дерева, чтобы не допустить переобучения или недообучения модели; в данном случае глубина 3 представляется хорошим выбором.



**Рис. 10.14.** Линия подгонки при регрессии на основе дерева принятия решений

В следующем разделе мы рассмотрим более надежный способ подгонки деревьев регрессии: случайные леса.

## Регрессия на основе случайного леса

Как было показано в главе 3, алгоритм случайного леса представляет собой ансамблевый прием, который объединяет множество деревьев принятия

решений. Случайный лес в большинстве случаев характеризуется лучшей эффективностью обобщения, чем отдельно взятое дерево принятия решений, благодаря случайности, которая помогает снизить дисперсию модели. Еще одно преимущество случайных лесов заключается в том, что они менее чувствительны к выбросам в наборе данных и не требуют особо объемной настройки параметров. Единственный параметр в случайных лесах, с которым обычно приходится экспериментировать — количество деревьев в ансамбле. Базовый алгоритм случайного леса для регрессии почти идентичен алгоритму случайного леса для классификации, рассмотренному в главе 3, с тем лишь отличием, что для выращивания индивидуальных деревьев принятия решений мы используем критерий MSE, и спрогнозированная целевая переменная вычисляется как средний прогноз по всем деревьям принятия решений.

А теперь давайте задействуем все признаки в наборе данных Housing для подгонки регрессионной модели на основе случайного леса с применением 60% образцов и оценим ее эффективность на оставшихся 40% образцов. Вот необходимый код:

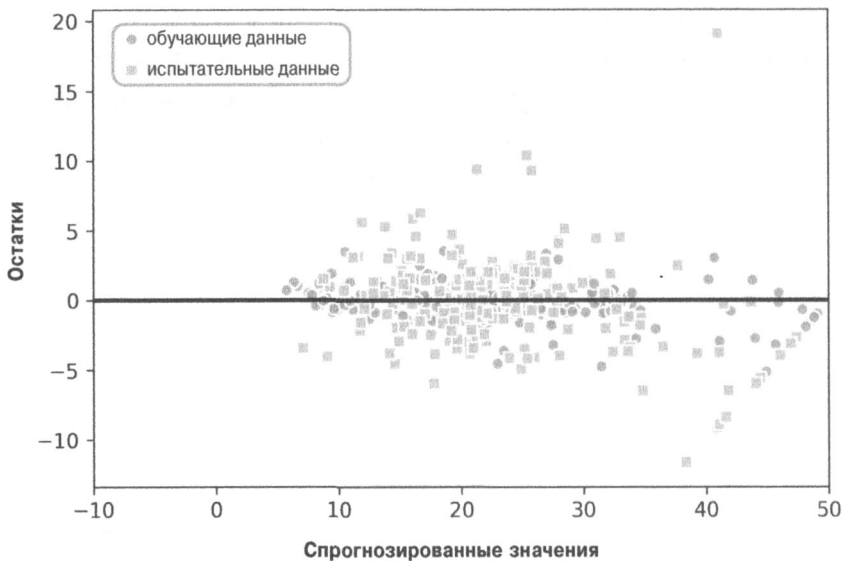
```
>>> X = df.iloc[:, :-1].values
>>> y = df['MEDV'].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                       test_size=0.4,
...                       random_state=1)
>>>
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(n_estimators=1000,
...                                criterion='mse',
...                                random_state=1,
...                                n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> print('MSE при обучении: %.3f, при испытании: %.3f' % (
...     mean_squared_error(y_train, y_train_pred),
...     mean_squared_error(y_test, y_test_pred)))
MSE при обучении: 1.642, при испытании: 11.052
>>> print('R^2 при обучении: %.3f, при испытании: %.3f' % (
...     r2_score(y_train, y_train_pred),
...     r2_score(y_test, y_test_pred)))
R^2 при обучении: 0.979, при испытании: 0.878
```

К сожалению, мы видим, что случайный лес склонен к переобучению обучающими данными. Тем не менее, он по-прежнему способен относительно хорошо раскрывать связь между целевой и объясняющей переменными ( $R^2 = 0.878$  на испытательном наборе данных).

Наконец, взглянем на остатки, относящиеся к прогнозу:

```
>>> plt.scatter(y_train_pred,
...             y_train_pred - y_train,
...             c='steelblue',
...             edgecolor='white',
...             marker='o',
...             s=35,
...             alpha=0.9,
...             label='обучающие данные')
>>> plt.scatter(y_test_pred,
...             y_test_pred - y_test,
...             c='limegreen',
...             edgecolor='white',
...             marker='s',
...             s=35,
...             alpha=0.9,
...             label='испытательные данные')
>>> plt.xlabel('Спрогнозированные значения')
>>> plt.ylabel('Остатки')
>>> plt.legend(loc='upper left')
>>> plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='black')
>>> plt.xlim([-10, 50])
>>> plt.tight_layout()
>>> plt.show()
```

Как уже было подытожено коэффициентом  $R^2$ , по выбросам в направлении оси  $y$  мы видим, что модель подгоняется к обучающим данным лучше, чем к испытательным данным. К тому же распределение остатков вокруг нулевой центральной точки не выглядит полностью случайным, указывая на то, что модель неспособна захватывать всю исследовательскую информацию. Однако график остатков, показанный на рис. 10.15, демонстрирует крупное улучшение в сравнении с графиком остатков линейной модели, который мы строили ранее в главе.



**Рис. 10.15.** График остатков модели на основе случайного леса

В идеале ошибка модели должна быть случайной или непредсказуемой. Другими словами, ошибка прогнозов не должна быть связана с любой информацией, содержащейся в объясняющих переменных; наоборот, она обязана отражать случайность реальных распределений или повторяющихся шаблонов. Если мы наблюдаем повторяющиеся шаблоны в ошибках прогнозов, например, изучая график остатков, то это означает, что график остатков содержит прогнозирующую информацию. Распространенная причина такого явления — просачивание объясняющей информации в остатки.

К сожалению, не существует универсального подхода к решению проблемы отсутствия случайности в графиках остатков; здесь требуется экспериментирование. В зависимости от доступных данных мы можем быть в состоянии улучшить модель, трансформируя переменные, настраивая гиперпараметры алгоритма обучения, выбирая более простые или более сложные модели, удаляя выбросы или включая дополнительные переменные.



На  
заметку!

### Регрессия с помощью метода опорных векторов

В главе 3 также был представлен ядерный трюк, который может использоваться в сочетании с *методом опорных векторов (SVM)* для классификации и полезен, когда мы имеем дело с нелинейными задачами. Хотя обсуждение выходит за рамки книги, методы SVM можно

также применять в нелинейных задачах регрессии. Заинтересованные читатели могут найти дополнительные сведения о методах SVM для регрессии в замечательном отчете “Support Vector Machines for Classification and Regression” (Методы опорных векторов для классификации и регрессии), С. Ганн и другие (технический отчет ISIS номер 14, 1998 г.), доступном по ссылке <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.579.6867&rep=rep1&type=pdf>. Регрессор SVM также реализован в библиотеке `scikit-learn`; описание его использования предлагается по ссылке <http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR>.

## Резюме

В начале главы вы узнали о простом линейном регрессионном анализе для моделирования связи между единственной объясняющей переменной и непрерывной переменной ответа. Затем мы обсудили удобный прием исследовательского анализа данных, направленный на поиск повторяющихся шаблонов и аномалий в данных, который является важным первым шагом при решении задач прогнозирующего моделирования.

Мы построили первую модель путем реализации линейной регрессии с применением подхода, основанного на градиентной оптимизации. Затем мы показали, каким образом задействовать линейные модели из библиотеки `scikit-learn`, предназначенные для регрессии, и также реализовали надежный метод регрессии RANSAC как прием обработки выбросов. Для оценки прогнозирующей эффективности регрессионных моделей мы вычисляли сумму квадратичных ошибок и связанную с ней метрику  $R^2$ . Вдобавок мы обсудили полезный графический подход к диагностированию проблем, касающихся регрессионных моделей: график остатков.

После выяснения, как применять к регрессионным моделям регуляризацию, чтобы снизить сложность моделей и избежать переобучения, мы представили несколько подходов к моделированию нелинейных связей, включая регрессоры на основе полиномиальной трансформации признаков и случайных лесов.

В предшествующих главах мы очень подробно обсудили обучение с учителем, классификацию и регрессионный анализ. В следующей главе мы займемся еще одной интересной подобластью МО — обучением без учителя — и объясним, как использовать кластерный анализ для нахождения скрытых структур в данных при отсутствии целевых переменных.





# РАБОТА С НЕПОМЕЧЕННЫМИ ДАНЫМИ — КЛАСТЕРНЫЙ АНАЛИЗ

В предшествующих главах для построения моделей МО мы применяли приемы обучения с учителем, используя данные с уже известными ответами, т.к. в обучающих данных были доступны метки классов. В этой главе мы переключимся на исследование кластерного анализа — категории приемов *обучения без учителя*, позволяющей обнаруживать скрытые структуры в данных, где правильный ответ заранее не известен. *Кластеризация* преследует цель найти в данных естественное группирование, при котором элементы в одном кластере будут больше похожими друг на друга, чем на элементы из других кластеров.

Учитывая исследовательскую природу, кластеризация является захватывающей темой, и в главе вы ознакомитесь со следующими концепциями, которые могут помочь организовать данные в содержательные структуры:

- нахождение центров подобия с применением популярного алгоритма *K-Means* (метод К-средних);
- принятие восходящего подхода к построению иерархических деревьев кластеризации;
- идентификация произвольных форм объектов с использованием подхода с кластеризацией на основе плотности.

## Группирование объектов по подобию с применением алгоритма K-Means

В текущем разделе мы обсудим один из самых популярных алгоритмов кластеризации — K-Means, который широко используется в научном сообществе и индустрии. Кластеризация (или кластерный анализ) представляет собой прием, позволяющий находить группы подобных объектов, т.е. объектов, которые больше связаны друг с другом, чем с объектами из других групп. Примеры бизнес-ориентированных приложений кластеризации включают группирование документов, музыкальных произведений и фильмов по разным темам или поиск заказчиков, разделяющих похожие интересы, на базе общих покупательских линий поведения в качестве основы для механизмов выдачи рекомендаций.

### Кластеризация K-Means с использованием scikit-learn

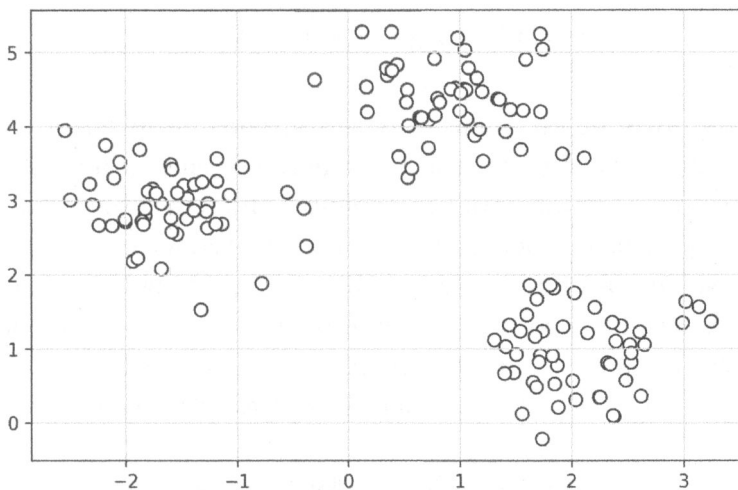
Как вскоре будет показано, алгоритм K-Means не только чрезвычайно легко реализовать, но он еще и очень эффективен с вычислительной точки зрения в сравнении с другими алгоритмами кластеризации, что может объяснять его популярность. Алгоритм K-Means относится к категории *кластеризации на основе прототипов*. Позже в главе мы обсудим остальные две категории кластеризации — *иерархическую* кластеризацию и *кластеризацию на основе плотности*.

Кластеризация на основе прототипов означает, что каждый кластер представляется прототипом, который обычно будет либо *центроидом* (средним) подобных точек с непрерывными признаками, либо *медоидом* (наиболее репрезентативной точкой или точкой, сводящей к минимуму расстояние до всех остальных точек, которые принадлежат индивидуальному кластеру) в случае категориальных признаков. Наряду с тем, что алгоритм K-Means очень хорош в идентификации кластеров со сферической формой, одним из его недостатков является необходимость *заранее* указывать количество кластеров  $k$ . Выбор неподходящего значения для  $k$  может привести к плохой эффективности кластеризации. Далее в главе мы рассмотрим метод *локтя* (*elbow method*) и *графики силуэтов* (*silhouette plot*) — удобные приемы для оценки качества кластеризации, которые помогают определить оптимальное количество кластеров  $k$ .

Хотя кластеризацию K-Means можно применять к данным высокой размерности, ради удобства визуализации мы проработаем примеры, используя простой двумерный набор данных:

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                    n_features=2,
...                    centers=3,
...                    cluster_std=0.5,
...                    shuffle=True,
...                    random_state=0)
>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:, 0],
...             X[:, 1],
...             c='white',
...             marker='o',
...             edgecolor='black',
...             s=50)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

Только что созданный набор данных состоит из 150 случайно сгенерированных точек, которые приблизительно сгруппированы в три области с более высокой плотностью, как видно на двумерном графике рассеяния на рис. 11.1.



*Рис. 11.1. График рассеяния для простого двумерного набора данных*

В реальных приложениях кластеризации мы не располагаем какой-либо изначально точной информацией о категориях образцов (информация предоставляется как эмпирические данные в противоположность выведению); если бы у нас были метки классов, тогда задача попала бы в область обучения с учителем. Таким образом, наша цель — сгруппировать образцы на основе подобия их признаков, чего можно достичь с применением алгоритма K-Means, который сводится к следующим четырем шагам.

1. Случайным образом выбрать  $k$  центроидов из образцов в качестве начальных центров кластеров.
2. Назначить каждый образец ближайшему центроиду  $\mu^{(j)}, j \in \{1, \dots, k\}$ .
3. Переместить каждый центроид в центр образцов, которые ему были назначены.
4. Повторять шаги 2 и 3 до тех пор, пока назначения кластеров не перестанут изменяться или не будет достигнуто максимальное количество итераций.

Теперь возникает вопрос: *как мы измерим подобие между объектами?* Мы можем определить подобие как противоположность расстоянию, и широко используемым расстоянием при кластеризации образцов с непрерывными признаками является *квадратичное евклидово расстояние* между двумя точками  $x$  и  $y$  в  $m$ -мерном пространстве:

$$d(x, y)^2 = \sum_{j=1}^m (x_j - y_j)^2 = \|x - y\|_2^2$$

Обратите внимание, что в предыдущем уравнении индекс  $j$  относится к  $j$ -тому измерению (столбцу признака) входных образцов  $x$  и  $y$ . До конца этого раздела мы будем применять надстрочные  $i$  и  $j$  для ссылки соответственно на индекс образца (запись данных) и индекс кластера.

Основываясь на такой метрике евклидова расстояния, мы можем описать алгоритм K-Means как простую задачу оптимизации — итерационный подход для минимизации *внутрикластерной суммы квадратичных ошибок* (*Sum of Squared Errors — SSE*), которую иногда называют *инерцией кластера*:

$$SSE = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)} \|x^{(i)} - \mu^{(j)}\|_2^2$$

Здесь  $\mu^{(j)}$  — репрезентативная точка (центроид) для кластера  $j$ ;  $w^{(ij)} = 1$ , если образец  $x^{(i)}$  находится в кластере  $j$ , и  $w^{(ij)} = 0$  в противном случае:

Теперь, когда мы знаем, как работает простой алгоритм K-Means, давайте применим его к нашему примеру набора данных, используя класс KMeans из модуля cluster библиотеки scikit-learn:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

С помощью приведенного выше кода мы устанавливаем желательное количество кластеров в 3; указание количества кластеров *заранее* — одно из ограничений алгоритма K-Means. Мы устанавливаем `n_init=10`, чтобы 10 раз выполнить кластеризацию K-Means независимо с разными случайными центроидами для выбора в качестве финальной модели с наименьшим значением SSE. Посредством параметра `max_iter` мы указываем максимальное количество итераций для каждого запуска (300). Обратите внимание, что реализация K-Means в scikit-learn останавливается раньше, если сходится до того, как выполнено максимальное количество итераций. Тем не менее, существует вероятность, что алгоритм K-Means не достигнет сходимости в определенном запуске, из-за чего может возникнуть проблема (высокие вычислительные затраты), если для `max_iter` было выбрано относительно большое значение. Решение проблемы со сходимостью предусматривает выбор более высоких значений `tol` — параметра, который управляет порогом изменений во внутрикластерной сумме SSE для объявления сходимости. В предыдущем коде мы выбрали порог `1e-04` ( $= 0.0001$ ).

Проблема с алгоритмом K-Means в том, что один или большее число кластеров могут быть пустыми. Отметим, что такая проблема не возникает для алгоритма *K-Medoids* (метод K-медоидов) или *FCM* (*fuzzy C-Means* — метод нечетких C-средних), который мы обсудим позже в разделе.

Однако в текущей реализации K-Means внутри библиотеки scikit-learn проблема учтена. Если кластер оказывается пустым, тогда реализация ищет

образец, наиболее отдаленный от центроида пустого кластера. Затем найденная наиболее отдаленная точка становится центроидом.



### Масштабирование признаков

Когда мы применяем алгоритм K-Means к реальным данным, используя метрику евклидова расстояния, то хотим гарантировать, что признаки измеряются с тем же самым масштабом, и при необходимости применяем стандартизацию z-оценкой или масштабирование по минимаксу.

После получения спрогнозированных меток кластеров `y_km` и обсуждения ряда сложностей, касающихся метода K-Means, мы визуализируем кластеры, которые реализация K-Means идентифицировала в наборе данных, вместе с центроидами кластеров. Центроиды хранятся в атрибуте `cluster_centers_` подогнанного объекта `KMeans`:

```
>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             marker='s', edgecolor='black',
...             label='кластер 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50, c='orange',
...             marker='o', edgecolor='black',
...             label='кластер 2')
>>> plt.scatter(X[y_km == 2, 0],
...             X[y_km == 2, 1],
...             s=50, c='lightblue',
...             marker='v', edgecolor='black',
...             label='кластер 3')
>>> plt.scatter(km.cluster_centers_[0, 0],
...             km.cluster_centers_[0, 1],
...             s=250, marker='*',
...             c='red', edgecolor='black',
...             label='центроиды')
>>> plt.legend(scatterpoints=1)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

Как видно на графике рассеяния, представленном на рис. 11.2, реализация K-Means разместила три центроида в центрах сфер, что выглядит подходящим группированием для этого набора данных.

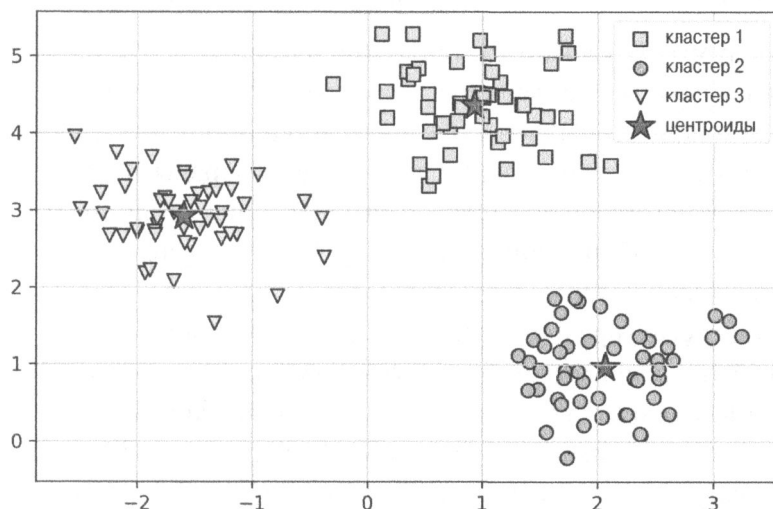


Рис. 11.2. График рассеяния после применения алгоритма K-Means к набору данных

Хотя алгоритм K-Means хорошо работает на таком игрушечном наборе данных, следовало бы акцентировать внимание на еще одном недостатке K-Means: мы обязаны *заранее* указывать количество кластеров  $k$ . В реальных приложениях выбор числа кластеров не всегда настолько очевиден, как в приведенном выше примере, особенно если мы работаем с набором данных более высокой размерности, который не может быть визуализирован. Другие особенности алгоритма K-Means включают тот факт, что кластеры не перекрываются и не иерархичны, и мы также предполагаем наличие в каждом кластере, по крайней мере, одного элемента. Позже в главе мы встретим дополнительные типы алгоритмов кластеризации — иерархические и основанные на плотности. Ни один из таких типов алгоритмов не требует заблаговременного указания количества кластеров или допущения о наличии сферических структур в наборе данных.

В следующем подразделе мы раскроем популярный вариант классического алгоритма K-Means под названием *K-Means++*. Несмотря на то что K-Means++ не избавляется от ранее упомянутых допущений и недостатков



алгоритма K-Means, он способен значительно улучшить результаты кластеризации посредством более искусного выбора начальных центров кластеров.

## Более интеллектуальный способ размещения начальных центроидов кластеров с использованием алгоритма K-Means++

До сих пор мы обсуждали классический алгоритм K-Means, который для размещения исходных центроидов применяет случайное начальное значение, что временами приводит к плохой кластеризации или медленной сходимости в случае, если начальные центроиды выбраны неудачно. Один из способов решения проблемы предусматривает прогон алгоритма K-Means несколько раз на наборе данных и выбор модели, лучше всех выполняющей с точки зрения SSE.

Другая стратегия предполагает размещение исходных центроидов далеко друг от друга посредством алгоритма K-Means++, который дает лучшие и более согласованные результаты, чем классический алгоритм K-Means (“K-Means++: The Advantages of Careful Seeding” (Метод K-Means++: преимущества аккуратной инициализации), Д. Артур и С. Васильвицкий, материалы восемнадцатого ежегодного симпозиума ACM-SIAM по дискретным алгоритмам, с. 1027–1035, Society for Industrial and Applied Mathematics (2007 г.)).

Инициализацию в алгоритме K-Means++ можно свести к следующим шагам.

1. Инициализировать пустой набор  $\mathbf{M}$  для хранения выбираемых  $k$  центроидов.
2. Случайным образом выбрать из входных образцов первый центроид  $\mu^{(j)}$  и присвоить его  $\mathbf{M}$ .
3. Для каждого образца  $\mathbf{x}^{(i)}$ , отсутствующего в  $\mathbf{M}$ , найти минимальное квадратичное расстояние  $d(\mathbf{x}^{(i)}, \mathbf{M})^2$  до всех центроидов в  $\mathbf{M}$ .
4. Для случайного выбора следующего центроида  $\mu^{(p)}$  использовать взвешенное распределение вероятностей вида 
$$\frac{d(\mu^{(p)}, \mathbf{M})^2}{\sum_i d(\mathbf{x}^{(i)}, \mathbf{M})^2}.$$
5. Повторять шаги 2 и 3 до тех пор, пока не будет выбрано  $k$  центроидов.
6. Продолжить выполнение согласно классическому алгоритму K-Means.

Чтобы применить алгоритм K-Means++ посредством объекта KMeans из scikit-learn, нам просто нужно установить параметр `init` в `'k-means++'`. В действительности `'k-means++'` является стандартным аргументом для параметра `init`, который настоятельно рекомендуется использовать на практике. Единственной причиной, по которой мы не применяли его в предыдущем примере, было нежелание вводить множество концепций сразу. Далее в разделе будет использоваться метод K-Means++, но заинтересованные читатели могут продолжить эксперименты с двумя разными подходами (классическим алгоритмом K-Means через `init='random'` в сравнении с алгоритмом K-Means++ через `init='k-means++'`) для размещения исходных центроидов кластеров.

## Жесткая или мягкая кластеризация

*Жесткая кластеризация* описывает семейство алгоритмов, где каждый образец из набора данных назначается в точности одному кластеру, как в алгоритмах K-Means и K-Means++, которые обсуждались в настоящей главе. Напротив, алгоритмы *мягкой кластеризации* (иногда называемой *нечеткой кластеризацией*) назначают образец одному или нескольким кластерам. Популярным примером мягкой кластеризации является алгоритм FCM (метод нечетких С-средних, также называемый методом *мягких К-средних* или методом *нечетких К-средних*). Первоначальная идея восходит еще к 1970-м годам, когда Джозеф Данн впервые предложил раннюю версию нечеткой кластеризации с целью улучшения алгоритма K-Means (“A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters” (Нечеткий родственник процесса ISODATA и его использование при обнаружении компактных вполне разделенных кластеров), Дж. Данн (1973 г.)). Спустя почти десятилетие Джеймс Бездек опубликовал свою работу по усовершенствованию алгоритма нечеткой кластеризации, который также известен как алгоритм FCM (“Pattern Recognition with Fuzzy Objective Function Algorithms” (Распознавание образов с помощью алгоритмов с нечеткой целевой функцией), Дж. Бездек, Springer Science+Business Media (2013 г.)).

Процедура FCM очень похожа на алгоритм K-Means, но жесткое назначение кластера каждой точке заменяется вероятностями ее принадлежности к каждому кластеру. В алгоритме K-Means мы могли бы выразить членство в кластерах образца  $x$  посредством разреженного вектора двоичных значений:

$$\left[ \begin{array}{l} x \in \mu^{(1)} \rightarrow w^{(i,j)} = 0 \\ x \in \mu^{(2)} \rightarrow w^{(i,j)} = 1 \\ x \in \mu^{(3)} \rightarrow w^{(i,j)} = 0 \end{array} \right]$$

Здесь индексная позиция со значением 1 указывает центроид кластера  $\mu^{(j)}$ , которому назначен образец (предполагая, что  $k = 3, j \in \{1, 2, 3\}$ ). В противоположность этому вектор членства в FCM можно было бы представить следующим образом:

$$\left[ \begin{array}{l} x \in \mu^{(1)} \rightarrow w^{(i,j)} = 0.1 \\ x \in \mu^{(2)} \rightarrow w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} \rightarrow w^{(i,j)} = 0.05 \end{array} \right]$$

Каждое значение находится в диапазоне  $[0, 1]$  и представляет вероятность членства в соответствующем центроиде кластера. Сумма вероятностей членства для заданного образца равна 1. Подобно алгоритму K-Means алгоритм FCM может быть резюмирован в виде к четырех шагов.

1. Указать количество центроидов  $k$  и случайным образом назначить каждой точке членство в кластерах.
2. Вычислить центроиды кластеров  $\mu^{(j)}, j \in \{1, \dots, k\}$ .
3. Для каждой точки обновить членство в кластерах.
4. Повторять шаги 2 и 3 до тех пор, пока коэффициенты членства не перестанут изменяться либо не будет достигнут определенный пользователем порог или максимальное количество итераций.

Целевая функция FCM — мы обозначим ее как  $J_m$  — выглядит очень похожей на внутрикластерную сумму квадратичных ошибок, которая минимизируется в методе K-Means:

$$J_m = \sum_{i=1}^n \sum_{j=1}^k w^{m(i,j)} \|x^{(i)} - \mu^{(j)}\|_2^2$$

Тем не менее, следует отметить, что указатель членства  $w^{(i,j)}$  является не двоичной величиной как в алгоритме K-Means ( $w^{(i,j)} \in \{0, 1\}$ ), а вещественным значением, которое обозначает вероятность членства ( $w^{(i,j)} \in [0, 1]$ ). Кроме того, мы добавили к  $w^{(i,j)}$  дополнительный показатель степени  $m$  —

любое число, которое больше или равно 1 (обычно  $m = 2$ ). Показатель степени  $m$  называется *коэффициентом нечеткости* (или *оператором размытия* (*fuzzifier*)), который управляет степенью нечеткости.

Чем выше значение  $m$ , тем ниже становится вероятность членства в кластерах  $w^{(i,j)}$ , что приводит к более нечетким кластерам. Сама вероятность членства в кластерах рассчитывается следующим образом:

$$w^{(i,j)} = \left[ \sum_{c=1}^k \left( \frac{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(c)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Скажем, если выбрать три центра кластеров, как в предыдущем примере с алгоритмом К-Means, тогда вычислить вероятность членства образца  $\mathbf{x}^{(i)}$  в кластере  $\boldsymbol{\mu}^{(j)}$  можно было бы так:

$$w^{(i,j)} = \left[ \left( \frac{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(1)}\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(2)}\|_2} \right)^{\frac{2}{m-1}} + \left( \frac{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)}\|_2}{\|\mathbf{x}^{(i)} - \boldsymbol{\mu}^{(3)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

Центр  $\boldsymbol{\mu}^{(j)}$  кластера рассчитывается как среднее всех образцов, взвешенное по степени принадлежности каждого образца к этому кластеру ( $w^{(i,j)^m}$ ):

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^n w^{(i,j)^m} \mathbf{x}^{(i)}}{\sum_{i=1}^n w^{(i,j)^m}}$$

Просто взглянув на уравнение для вычисления вероятности членства в кластерах, мы можем сказать, что каждая итерация в алгоритме FCM является более затратной, чем любая итерация в К-Means. Однако для достижения сходимости алгоритм FCM обычно требует меньшего числа итераций. К сожалению, в текущий момент алгоритм FCM в библиотеке scikit-learn не реализован. Тем не менее, на практике было обнаружено, что алгоритмы К-Means и FCM выдают очень похожие результаты кластеризации, как описано в работе С. Гоша и С. Дуби “Comparative Analysis of k-means and Fuzzy C-Means Algorithms” (Сравнительный анализ алгоритмов К-Means и нечетких С-средних), IJACSA, 4: с. 35–38 (2013 г.).

## Использование метода локтя для нахождения оптимального количества кластеров

Одно из основных затруднений в обучении без учителя связано с тем, что окончательный ответ нам не известен. Мы не располагаем изначально точными метками классов, которые позволили бы применять приемы, описанные в главе 6, для оценки эффективности модели обучения с учителем. Следовательно, для количественной оценки качества кластеризации нам необходимо использовать внутренние метрики, такие как внутрикластерное значение SSE (искажение), чтобы сравнивать эффективность разных результатов кластеризации K-Means.

Удобно то, что в случае применения библиотеки `scikit-learn` явно вычислять внутрикластерное значение SSE не придется, т.к. после подгонки модели `KMeans` оно уже доступно через атрибут `inertia_`:

```
>>> print('Искажение: %.2f' % km.inertia_)  
Искажение: 72.48
```

На основе внутрикластерного значения SSE мы можем воспользоваться графическим инструментом — так называемым *методом локтя* — для оценки оптимального количества кластеров  $k$  в имеющейся задаче. Мы можем сказать, что с увеличением  $k$  искажение будет понижаться. Причина в том, что образцы станут располагаться ближе к центроидам, которым они назначены. Идея метода локтя заключается в том, чтобы идентифицировать значение  $k$ , при котором искажение начинает увеличиваться быстрее всего, что прояснится, если построить график искажения для разных значений  $k$ :

```
>>> distortions = []  
>>> for i in range(1, 11):  
...     km = KMeans(n_clusters=i,  
...                 init='k-means++',  
...                 n_init=10,  
...                 max_iter=300,  
...                 random_state=0)  
...     km.fit(X)  
...     distortions.append(km.inertia_)  
>>> plt.plot(range(1,11), distortions, marker='o')  
>>> plt.xlabel('Количество кластеров')  
>>> plt.ylabel('Искажение')  
>>> plt.tight_layout()  
>>> plt.show()
```

Как видно на результирующем графике (рис. 11.3), *локоть* находится в точке  $k = 3$ , свидетельствуя о том, что  $k = 3$  — хороший выбор для этого набора данных.

## Количественная оценка качества кластеризации через графики силуэтов

Еще одной внутренней метрикой для оценки качества кластеризации является *анализ силуэтов*, который может применяться также к рассматриваемым далее в главе алгоритмам кластеризации, отличающимся от K-Means. Анализ силуэтов может использоваться как графический инструмент для построения графика, который отображает меру плотности группирования образцов в кластерах.

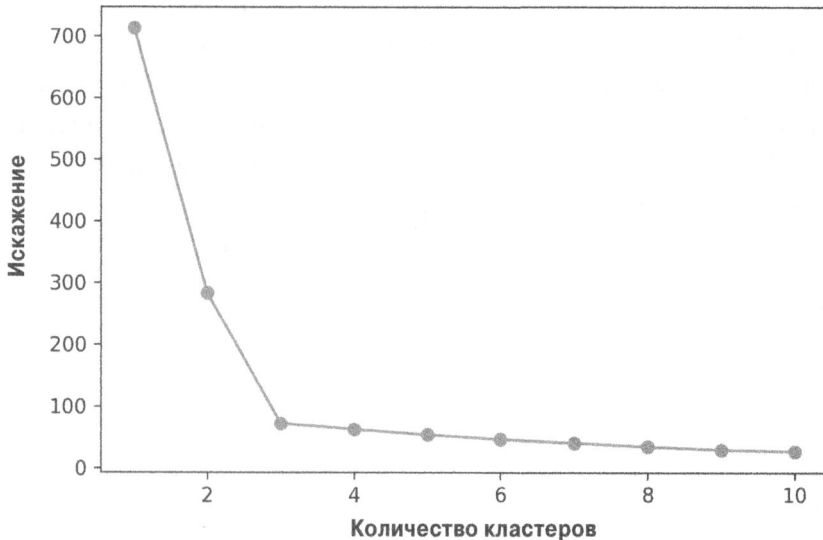


Рис. 11.3. График искажения для разного числа кластеров

Чтобы вычислить *коэффициент силуэта* (*silhouette coefficient*) одиночного образца в наборе данных, мы можем выполнить описанные ниже три шага.

1. Вычислить *связность кластера*  $a^{(i)}$  как среднее расстояние между образцом  $x^{(i)}$  и всеми остальными точками в том же самом кластере.
2. Вычислить *отделение кластера*  $b^{(i)}$  от следующего ближайшего кластера как среднее расстояние между образцом  $x^{(i)}$  и всеми образцами в ближайшем кластере.

3. Вычислить силуэт  $s^{(i)}$  как разность между связностью и отделением кластера, деленную на большее среди них значение:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}$$

Коэффициент силуэта ограничен диапазоном от  $-1$  до  $1$ . Основываясь на предыдущем уравнении, мы можем отметить, что коэффициент силуэта равен  $0$  в случае равенства отделения и связности кластера ( $b^{(i)} = a^{(i)}$ ). Кроме того, мы приближаемся к идеальному коэффициенту силуэта, если  $b^{(i)} \gg a^{(i)}$ , т.к.  $b^{(i)}$  количественно определяет, насколько образец не похож на образцы из других кластеров, и  $a^{(i)}$  сообщает о том, в какой степени он похож на остальные образцы в своем кластере.

Коэффициент силуэта доступен в виде функции `silhouette_samples` из модуля `metrics` библиотеки `scikit-learn`, а для удобства можно дополнительно импортировать функцию `silhouette_scores`, которая вычисляет средний коэффициент силуэта по всем образцам, что эквивалентно вызову `numpy.mean(silhouette_samples(...))`. С помощью приведенного ниже кода мы строим график коэффициентов силуэта для кластеризации K-Means при  $k = 3$ :

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

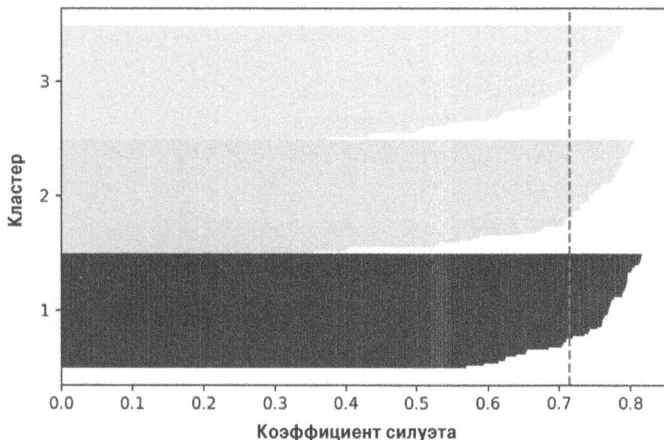
>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                     y_km,
...                                     metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
```

```

>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...               c_silhouette_vals,
...               height=1.0,
...               edgecolor='none',
...               color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...              color="red",
...              linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Кластер')
>>> plt.xlabel('Коэффициент силуэта')
>>> plt.tight_layout()
>>> plt.show()

```

Визуальное обследование результирующего графика коэффициентов силуэта (рис. 11.4) позволяет быстро разглядеть размеры разных кластеров и идентифицировать кластеры, содержащие *выбросы*.



**Рис. 11.4.** График коэффициентов силуэта для кластеризации K-Means при  $k = 3$



Однако на рис. 11.4 можно заметить, что коэффициенты силуэта далеки от 0, что в данном случае является индикатором *хорошей* кластеризации. Для подведения итогов выполненной кластеризации мы добавили к графику средний коэффициент силуэта (изображенный пунктирной линией).

Чтобы посмотреть, как выглядит график коэффициентов силуэта для относительно *плохой* кластеризации, давайте инициализируем алгоритм K-Means только двумя центроидами:

```
>>> km = KMeans(n_clusters=2,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             edgecolor='black',
...             marker='s',
...             label='кластер 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50,
...             c='orange',
...             edgecolor='black',
...             marker='o',
...             label='кластер 2')
>>> plt.scatter(km.cluster_centers_[0, 0],
...             km.cluster_centers_[0, 1],
...             s=250,
...             marker='*',
...             c='red',
...             label='центроиды')
>>> plt.legend()
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 11.5) видно, что один из центроидов попал между двумя из трех сферических группирований точек образцов.

Хотя такая кластеризация не считается абсолютно плохой, она субоптимальна.

Важно отметить, что при решении реальных задач обычно мы не располагаем такой роскошью, как визуализация наборов данных в виде двумерных графиков рассеяния, поскольку имеем дело с данными более высокой размерности.

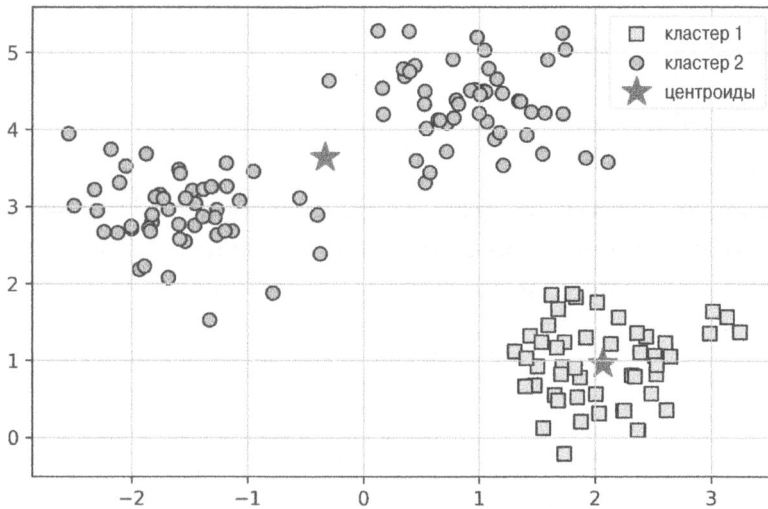


Рис. 11.5. Кластеризация K-Means с двумя центроидами

Далее мы создаем график коэффициентов силуэта для оценки результатов:

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(X,
...                                     y_km,
...                                     metric='euclidean')
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
```

```

...     plt.barh(range(y_ax_lower, y_ax_upper),
...               c_silhouette_vals,
...               height=1.0,
...               edgecolor='none',
...               color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Кластер')
>>> plt.xlabel('Коэффициент силуэта')
>>> plt.tight_layout()
>>> plt.show()

```

На полученном графике коэффициентов силуэта (рис. 11.6) видно, что теперь силуэты заметно отличаются по длине и ширине, что подтверждает относительно плохую или, во всяком случае, субоптимальную кластеризацию.

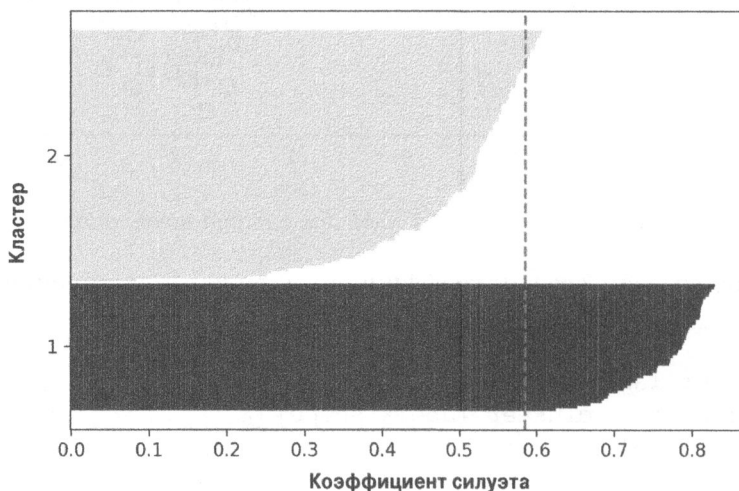


Рис. 11.6. График коэффициентов силуэта для кластеризации K-Means при  $k = 2$

## Организация кластеров в виде иерархического дерева

В этом разделе мы рассмотрим альтернативный подход к кластеризации на основе прототипов — *иерархическую кластеризацию*. Одно из преимуществ алгоритмов иерархической кластеризации заключается в том, что они позволя-

ют строить *дендрограммы* (древовидные диаграммы с визуализацией двоичной иерархической кластеризации), которые могут помочь с интерпретацией результатов за счет содержательной систематизации. Другим преимуществом является то, что нам не нужно указывать количество кластеров заранее.

Существуют два основных подхода к иерархической кластеризации — *агломеративный* (*agglomerative*) и *дивизивный* (*divisive*). В дивизивной иерархической кластеризации мы начинаем с одного кластера, который охватывает весь набор данных, и итеративно разделяем его на меньшие кластеры до тех пор, пока каждый кластер не станет содержать один образец. В настоящем разделе мы сосредоточим внимание на агломеративной иерархической кластеризации, где принят противоположный подход. Мы начинаем с того, что делаем каждый образец индивидуальным кластером и объединяем ближайшие пары кластеров до тех пор, пока не останется один кластер.

## Группирование кластеров в восходящей манере

Есть два стандартных алгоритма для агломеративной иерархической кластеризации — метод *одиночной связи* (*single linkage*) и метод *полной связи* (*complete linkage*). С применением метода одиночной связи мы вычисляем расстояния между наиболее похожими членами для каждой пары кластеров и объединяем два кластера с наименьшим расстоянием между самыми похожими членами. Метод полной связи аналогичен методу одиночной связи, но вместо сравнения наиболее похожих членов в каждой паре кластеров для выполнения объединения мы сравниваем самые непохожие члены. Сказанное иллюстрируется на рис. 11.7.

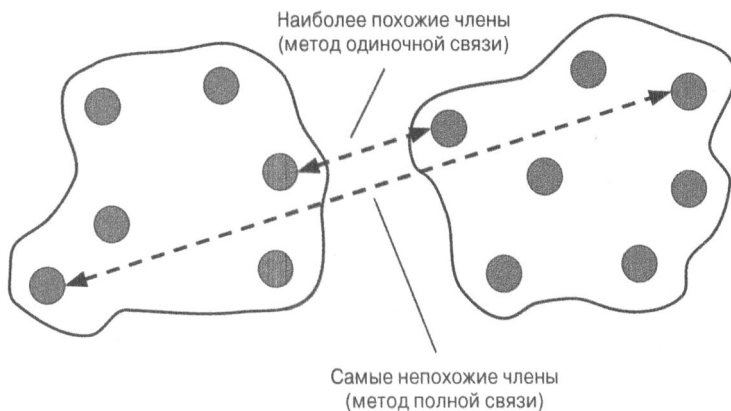


Рис. 11.7. Метод одиночной связи и метод полной связи



### Альтернативные типы связей

Другие широко используемые алгоритмы для агломеративной иерархической кластеризации включают метод *средней связи* (*average linkage*) и метод *связи Уорда* (*Ward's linkage*). В методе средней связи мы объединяем пары кластеров, основываясь на минимальных средних расстояниях между всеми членами групп в двух кластерах. В методе связи Уорда объединяются два кластера, которые приводят к минимальному увеличению совокупной внутрикластерной ошибки SSE.

В текущем разделе мы сконцентрируемся на агломеративной иерархической кластеризации, применяя метод полной связи. Такая кластеризация является итерационной процедурой, которую можно описать в виде следующих шагов.

1. Рассчитать матрицу расстояний для всех образцов.
2. Представить каждую точку данных как одноэлементный кластер.
3. Объединить два ближайших кластера, основываясь на расстоянии между самыми непохожими (дальними) членами.
4. Обновить матрицу подобия.
5. Повторять шаги 2–4 до тех пор, пока не останется единственный кластер.

Далее мы обсудим, как рассчитывать матрицу расстояний (шаг 1). Но сначала давайте сгенерируем случайные данные образцов, чтобы работать с ними: строки представляют различные наблюдения (идентификаторы ID\_0 — ID\_4), а столбцы — разные признаки (X, Y, Z) образцов:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random_sample([5, 3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

После выполнения показанного выше кода мы должны увидеть кадр данных, содержащий случайным образом сгенерированные образцы (рис. 11.8).

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

*Рис. 11.8. Кадр данных со случайно сгенерированными образцами*

## Выполнение иерархической кластеризации на матрице расстояний

Чтобы рассчитать матрицу расстояний для передачи на вход алгоритму иерархической кластеризации, мы будем использовать функцию `pdist` из подмодуля `spatial.distance` библиотеки `SciPy`:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...                         pdist(df, metric='euclidean')),
...                         columns=labels, index=labels)
>>> row_dist
```

В коде мы вычисляем евклидово расстояние между каждой парой входных образцов в наборе данных, основываясь на признаках X, Y и Z.

Плотная матрица расстояний, возвращенная функцией `pdist`, предоставляется в качестве входа функции `squareform`, которая создаст симметричную матрицу попарных расстояний (рис. 11.9).

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

*Рис. 11.9. Симметричная матрица попарных расстояний*

Затем мы применим к кластерам метод полной связи, используя функцию `linkage` из подмодуля `cluster.hierarchy` библиотеки `SciPy`, который возвратит так называемую *матрицу связей* (*linkage matrix*).

Тем не менее, прежде чем вызывать функцию `linkage`, давайте внимательно посмотрим на документацию по ней:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[...]
Параметры:
  y : ndarray
      Плотная или избыточная матрица расстояний. Сжатая матрица
      расстояний представляет собой плоский массив, содержащий верхний
      треугольник матрицы расстояний. Это та форма, которую возвращает
      pdist. В качестве альтернативы коллекцию из m векторов
      наблюдений в n измерениях можно передать как массив m на n.

  method : str, необязательный
      Используемый метод связи. Полное описание ищите
      ниже в разделе "Методы связи".

  metric : str, необязательный
      Используемая метрика расстояния. Список допустимых метрик
      расстояния ищите в документации по функции distance.pdist.

Возвращает:
  Z : ndarray
      Иерархическая кластеризация, закодированная как матрица связей.
  [...]
```

Из описания функции мы делаем вывод, что для входного атрибута можно применять сжатую матрицу расстояний (верхний треугольник), возвращенную из `pdist`. В качестве альтернативы мы могли бы предоставить первоначальный массив данных и в аргументе функции `linkage` указать метрику `'euclidean'`. Однако мы не должны использовать определенную ранее матрицу расстояний `squareform`, т.к. она приведет к выдаче не тех значений расстояний, которые ожидалось. Подводя итоги, вот три возможных сценария.

- *Некорректный подход* — применение матрицы расстояний `squareform`, как в следующем фрагменте кода, приведет к получению неправильных результатов:

```
>>> row_clusters = linkage(row_dist,
...                         method='complete',
...                         metric='euclidean')
```

- *Корректный подход* — использование сжатой матрицы расстояний, как в показанном ниже фрагменте кода, в результате даст правильную матрицу связей:

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                         method='complete')
```

- *Корректный подход* — аналогично предыдущему сценарию применение полной входной матрицы образцов, как в приведенном далее фрагменте кода, также обеспечит получение правильной матрицы связей:

```
>>> row_clusters = linkage(df.values,
...                         method='complete',
...                         metric='euclidean')
```

Чтобы более тщательно рассмотреть результаты кластеризации, мы можем превратить их в объект `DataFrame` из `pandas` (лучше всего просматриваемый в Jupyter Notebook):

```
>>> pd.DataFrame(row_clusters,
...               columns=['Метка строки 1',
...                       'Метка строки 2',
...                       'Расстояние',
...                       'К-во элементов в кластере'],
...               index=['Кластер %d' % (i+1) for i in
...                       range(row_clusters.shape[0])])
```

На рис. 11.10 видно, что матрица связей состоит из нескольких строк, где каждая строка представляет одно объединение. Первый и второй столбцы отмечают самые непохожие члены в каждом кластере, а третий столбец указывает расстояние между этими членами. В последнем столбце приводится количество членов в каждом кластере.

	Метка строки 1	Метка строки 2	Расстояние	К-во элементов в кластере
Кластер 1	0.0	4.0	3.835396	2.0
Кластер 2	1.0	2.0	4.347073	2.0
Кластер 3	3.0	5.0	5.899885	3.0
Кластер 4	6.0	7.0	8.316594	5.0

Рис. 11.10. Результирующая матрица связей



Имея рассчитанную матрицу связей, мы можем визуализировать результаты в форме дендрограммы:

```
>>> from scipy.cluster.hierarchy import dendrogram
>>> # сделать дендрограмму в черном цвете (часть 1 из 2)
>>> # from scipy.cluster.hierarchy import set_link_color_palette
>>> # set_link_color_palette(['black'])
>>> row_dendr = dendrogram(row_clusters,
...                         labels=labels,
...                         # сделать дендрограмму в черном цвете (часть 2 из 2)
...                         # color_threshold=np.inf
...                         )
>>> plt.tight_layout()
>>> plt.ylabel('Евклидово расстояние')
>>> plt.show()
```

Выполнив предыдущий код, легко заметить, что ветви в результирующей дендрограмме показаны разными цветами (рис. 11.11). Цветовая схема происходит из списка цветов библиотеки Matplotlib, который зациклен для порогов расстояния в дендрограмме. Например, чтобы отобразить дендрограмму черным цветом, можно убрать символы комментария в соответствующих разделах приведенного выше кода.

Дендрограмма такого рода подытоживает различные кластеры, которые были сформированы во время агломеративной иерархической кластеризации; скажем, мы видим, что на основе метрики евклидова расстояния наиболее похожими оказываются образцы ID\_0 и ID\_4, за которыми следуют образцы ID\_1 и ID\_2.

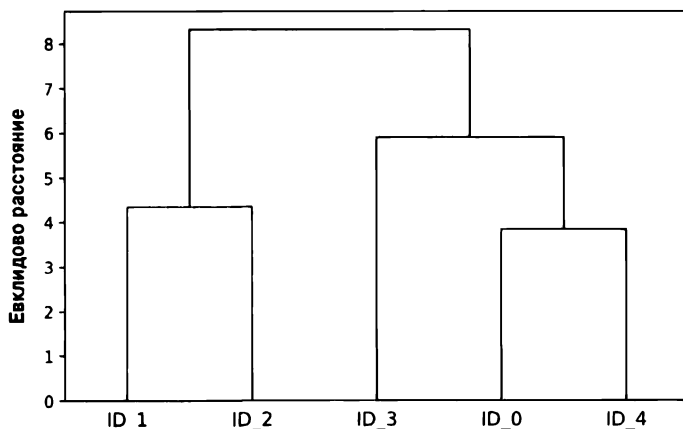


Рис. 11.11. Результирующая дендрограмма

## Прикрепление дендрограмм к тепловой карте

В практических приложениях дендрограммы иерархической кластеризации часто используются в сочетании с *тепловой картой*, которая позволяет представлять индивидуальные значения в массиве или матрице данных, содержащей обучающие образцы, с помощью цветового кода. В текущем разделе мы обсудим, как прикреплять дендрограмму к графику тепловой карты и соответствующим образом упорядочивать строки в тепловой карте.

Тем не менее, прикрепление дендрограммы к тепловой карте может быть слегка запутанным, так что давайте разберем процедуру шаг за шагом.

1. Мы создаем новый объект `figure` и определяем позицию оси  $x$ , позицию оси  $y$ , ширину и высоту дендрограммы через атрибут `add_axes`. Вдобавок мы поворачиваем дендрограмму на 90 градусов против часовой стрелки. Вот необходимый код:

```
>>> fig = plt.figure(figsize=(8, 8), facecolor='white')
>>> axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
>>> row_dendr = dendrogram(row_clusters,
...                         orientation='left')
>>> # примечание: для версии matplotlib, предшествующей v1.5.1,
>>> # используйте orientation='right'
```

2. Затем посредством ключа `leaves` мы переупорядочиваем данные в первоначальном объекте `DataFrame` согласно меткам кластеров, доступ к которым можно получать из объекта дендрограммы, по существу представляющего собой словарь Python. Код представлен ниже:

```
>>> df_rowclust = df.iloc[row_dendr['leaves'][:, -1]]
```

3. Теперь из переупорядоченного объекта `DataFrame` мы строим тепловую карту и размещаем ее рядом с дендрограммой:

```
>>> axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])
>>> cax = axm.matshow(df_rowclust,
...                   interpolation='nearest',
...                   cmap='hot_r')
```

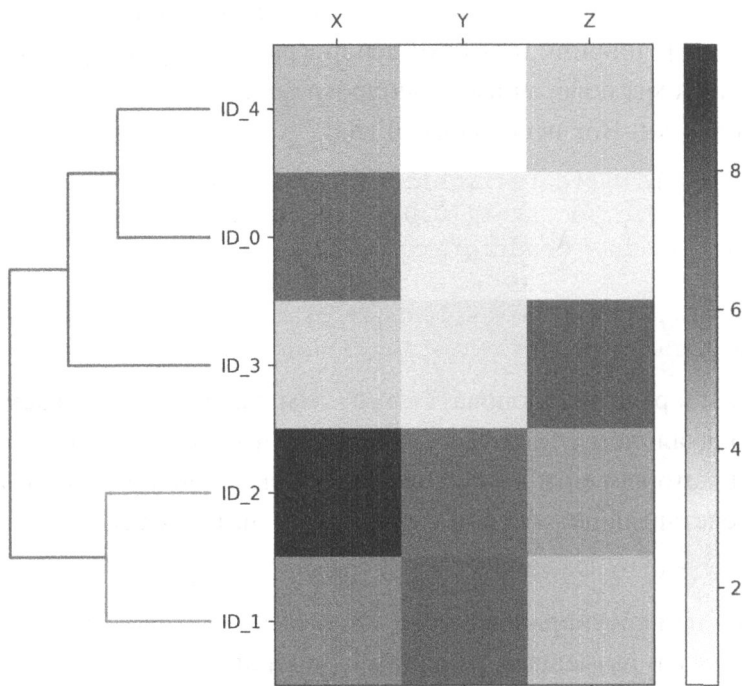
4. Наконец, мы улучшим эстетическое восприятие дендрограммы, удалив отметки осей и скрыв их линии. Кроме того, мы добавим цветовую полосу и назначим имена признаков и записей данных меткам осей  $x$  и  $y$  соответственно:

```

>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()

```

После выполнения всех ранее описанных шагов должна отобразиться тепловая карта с прикрепленной дендрограммой, как показано на рис. 11.12.



**Рис. 11.12.** Тепловая карта с прикрепленной дендрограммой

Как несложно заметить, порядок следования строк в тепловой карте отражает кластеризацию образцов на дендрограмме. В дополнение к простой дендрограмме закодированные цветом значения каждого образца и признака в тепловой карте снабжают нас симпатичной сводкой по набору данных.

## Применение агломеративной иерархической кластеризации с помощью `scikit-learn`

В предыдущем подразделе мы показали, как выполнять агломеративную иерархическую кластеризацию с использованием `SciPy`. Однако в библиотеке `scikit-learn` имеется также реализация `AgglomerativeClustering`, позволяющая выбирать количество кластеров, которые желательно возвратить. Класс `AgglomerativeClustering` полезен, если мы хотим подрезать иерархическое дерево кластеров. Устанавливая параметр `n_cluster` в 3, мы теперь кластеризируем образцы в три группы с применением того же самого метода полной связи, основанного на метрике евклидова расстояния, что и ранее:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=3,
...                               affinity='euclidean',
...                               linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Метки кластеров: %s' % labels)
Метки кластеров: [1 0 0 2 1]
```

По спрогнозированным меткам кластеров можно сделать вывод, что первый и пятый образцы (`ID_0` и `ID_4`) были назначены одному кластеру (метка 1), а образцы `ID_1` и `ID_2` — другому кластеру (метка 0). Образец `ID_3` был помещен в собственный кластер (метка 2). В целом результаты согласуются с результатами, которые мы наблюдали на дендрограмме. Но мы должны отметить, что образец `ID_3` больше похож на образцы `ID_4` и `ID_0`, чем на образцы `ID_1` и `ID_2`, как показано на рис. 11.12; в результатах кластеризации с помощью `scikit-learn` это не очевидно. Снова воспользуемся классом `AgglomerativeClustering`, но с `n_cluster=2`:

```
>>> ac = AgglomerativeClustering(n_clusters=2,
...                               affinity='euclidean',
...                               linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print('Метки кластеров: %s' % labels)
Метки кластеров: [0 1 1 0 0]
```

Как и ожидалось, в такой *подрезанной* иерархии кластеров метка `ID_3` была назначена тому же кластеру, что и метки `ID_0` и `ID_4`.

## Нахождение областей высокой плотности с помощью DBSCAN

Хотя мы не в состоянии раскрыть в этой главе подавляющее большинство алгоритмов кластеризации, давайте бегло взглянем на еще один подход к проведению кластеризации: *основанную на плотности пространственную кластеризацию для приложений с шумами (density-based spatial clustering of applications with noise — DBSCAN)*. Подход DBSCAN не выдвигает допущений о сферических кластерах, как делает алгоритм K-Means, и не разделяет набор данных на иерархии, требующие указываемой вручную точки подрезания. Из самого названия следует, что кластеризация на базе плотности назначает метки кластеров, основываясь на плотных областях точек. В DBSCAN понятие плотности определяется как количество точек внутри указанного радиуса  $\epsilon$ .

Согласно алгоритму DBSCAN каждому образцу (точке данных) назначается специальная метка с применением перечисленных ниже критериев.

- Точка считается *ядерной (core point)*, если, по крайней мере, указанное количество (MinPts) соседних точек попадают внутрь заданного радиуса  $\epsilon$ .
- *Граничная точка (border point)* — это точка, имеющая число соседей меньше MinPts в пределах  $\epsilon$ , но лежащая внутри радиуса  $\epsilon$  ядерной точки.
- Все остальные точки, которые не являются ядерными или граничными, трактуются как *шумовые (noise point)*.

После пометки точек как ядерных, граничных или шумовых алгоритм DBSCAN сводится к выполнению двух простых шагов.

1. Сформировать отдельный кластер для каждой ядерной точки или связанной группы ядерных точек. (Ядерные точки считаются связными, если расположены не дальше чем  $\epsilon$ .)
2. Назначить каждую граничную точку кластеру, к которому принадлежит соответствующая ей ядерная точка.

Чтобы лучше понять, на что могут быть похожи результаты выполнения алгоритма DBSCAN до того, как приступить к его реализации, на рис. 11.13

подытожены представленные выше сведения о ядерных, граничных и шумовых точках.

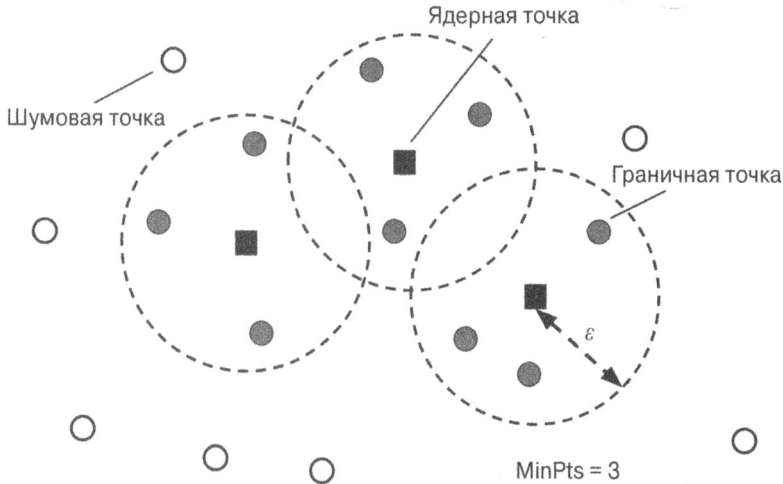


Рис. 11.13. Ядерные, граничные и шумовые точки

Одним из главных достоинств алгоритма DBSCAN является то, что он не выдвигает допущения относительно сферической формы кластеров, как делает алгоритм K-Means. Более того, DBSCAN отличается от алгоритмов K-Means и иерархической кластеризации еще и тем, что необязательно назначает каждую точку кластеру, но способен устранять шумовые точки.

В качестве более иллюстративного примера давайте создадим новый набор данных со структурами в форме полумесяца, чтобы сравнить на нем кластеризацию K-Means, иерархическую кластеризацию и кластеризацию DBSCAN:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                   noise=0.05,
...                   random_state=0)
>>> plt.scatter(X[:, 0], X[:, 1])
>>> plt.tight_layout()
>>> plt.show()
```

На результирующем графике (рис. 11.14) видно, что есть две хорошо заметные группы в форме полумесяца, каждая из которых состоит из 100 образцов (точек данных).

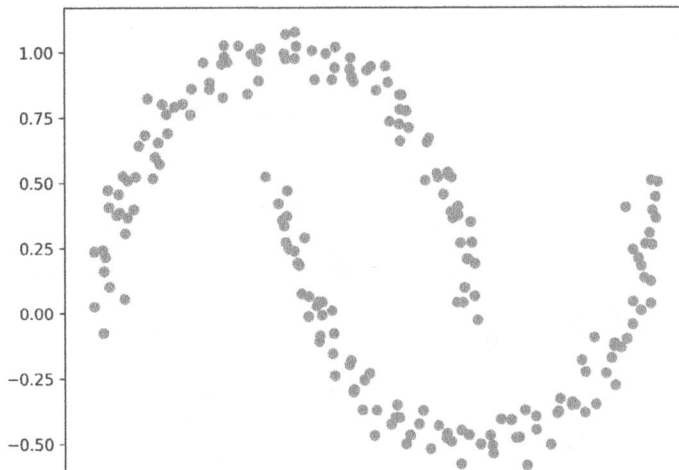


Рис. 11.14. Набор данных с группами в форме полумесяца

Мы начнем с использования кластеризации K-Means и агломеративной иерархической кластеризации методом полной связи, чтобы выяснить, могут ли они успешно идентифицировать группы в форме полумесяца как отдельные кластеры. Ниже приведен код:

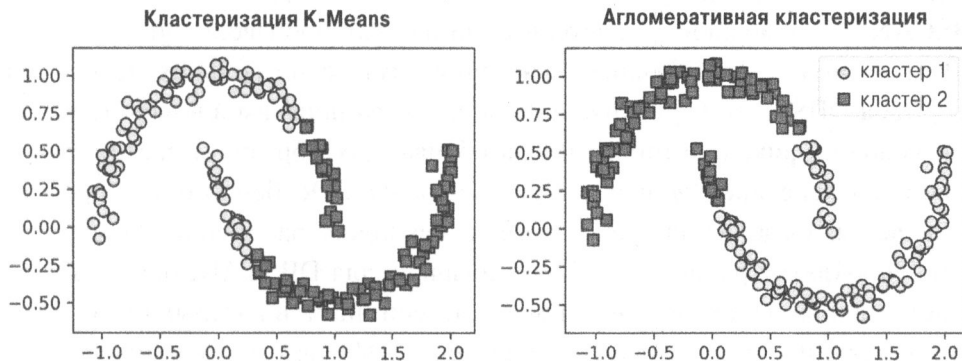
```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
>>> km = KMeans(n_clusters=2,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='кластер 1')
>>> ax1.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='кластер 2')
>>> ax1.set_title('Кластеризация K-Means')
```

```

>>> ac = AgglomerativeClustering(n_clusters=2,
...                               affinity='euclidean',
...                               linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac == 0, 0],
...             X[y_ac == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='кластер 1')
>>> ax2.scatter(X[y_ac == 1, 0],
...             X[y_ac == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='кластер 2')
>>> ax2.set_title('Агломеративная кластеризация')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()

```

Глядя на визуализированные результаты кластеризации (рис. 11.15), мы видим, что метод K-Means оказался неспособным разделить два кластера, и алгоритм иерархической классификации тоже не справился с этими сложными формами.



**Рис. 11.15.** Результаты кластеризации K-Means и агломеративной иерархической кластеризации методом полной связи



В заключение мы опробуем на имеющемся наборе данных алгоритм DBSCAN, чтобы посмотреть, в состоянии ли он найти два кластера в форме полумесяца, применяя подход на основе плотности:

```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db == 0, 0],
...             X[y_db == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='кластер 1')
>>> plt.scatter(X[y_db == 1, 0],
...             X[y_db == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='кластер 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

Как показано на рис. 11.16, алгоритм DBSCAN сумел успешно выявить формы в виде полумесяца, что подчеркивает одну из сильных сторон DBSCAN — возможность кластеризации данных произвольной формы.

Тем не менее, мы обязаны также упомянуть о нескольких недостатках алгоритма DBSCAN. С увеличением количества признаков в наборе данных (при условии фиксированного числа обучающих образцов) растет отрицательное воздействие *“проклятия размерности”*. Особенно остро проблема проявляется, когда используется метрика евклидова расстояния. Однако проблема *“проклятия размерности”* не уникальна для DBSCAN; она также оказывает влияние на другие алгоритмы кластеризации, в которых применяется метрика евклидова расстояния, например, K-Means и алгоритмы иерархической кластеризации. Вдобавок в алгоритме DBSCAN есть два гиперпараметра (MinPts и  $\epsilon$ ), которые нуждаются в оптимизации для выдачи хороших результатов кластеризации.

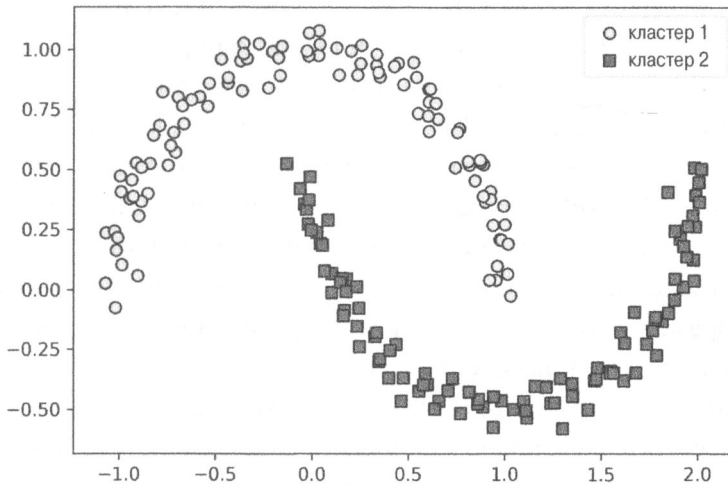


Рис. 11.16. Результаты кластеризации по алгоритму DBSCAN

Нахождение подходящей комбинации MinPts и  $\epsilon$  может стать проблематичным, если разницы в плотности внутри набора данных относительно велики.



На заметку!

### Кластеризация на основе графов

До сих пор мы видели три самых фундаментальных категории алгоритмов кластеризации: кластеризацию на основе прототипов K-Means, агломеративную иерархическую кластеризацию и кластеризацию на основе плотности посредством DBSCAN. Тем не менее, следует также упомянуть о четвертой категории более сложных алгоритмов кластеризации, которая в главе не раскрывалась — *кластеризация на основе графов*. Вероятно, наиболее знаменитыми членами семейства алгоритмов кластеризации на основе графов являются алгоритмы *спектральной кластеризации*.

Несмотря на наличие множества разных реализаций спектральной кластеризации, все они разделяют одну общую черту — используют для вывода кластерных связей собственные векторы подобия или матрицу расстояний. Поскольку обсуждение спектральной кластеризации выходит за рамки настоящей книги, узнать о ней больше можно, прочитав великолепную работу Ульрики фон Люксбург “A tutorial on spectral clustering” (Учебное пособие по спектральной кластеризации), Statistics and Computing, 17(4): с. 395–416 (2007 г.), которая свободно доступна по ссылке <http://arxiv.org/pdf/0711.0189v1.pdf>.

Обратите внимание, что на практике не всегда очевидно, какой алгоритм кластеризации будет выполняться лучше на конкретном наборе данных, особенно если данные поступают с высокой размерностью, затрудняя или препятствуя их визуализации. Кроме того, важно сделать особый акцент на том, что успешность кластеризации зависит не только от алгоритма и его гиперпараметров. Напротив, выбор подходящей метрики расстояния и знание предметной области, которое способно помочь должным образом сориентировать проведение экспериментов, могут оказаться намного важнее.

Таким образом, в контексте “проклятия размерности” устоявшаяся практика предусматривает применение приемов понижения размерности до выполнения самой кластеризации. Такие приемы понижения размерности для непомеченных наборов данных включают анализ главных компонент и ядерный анализ главных компонент с ядром RBF, которые были раскрыты в главе 5. Вдобавок чрезвычайно распространенной практикой является сжатие наборов данных до двумерных подпространств, которые делают возможной визуализацию кластеров и назначение меток с использованием двумерных графиков рассеяния, особенно полезных для оценки результатов.

## Резюме

В главе вы узнали о трех алгоритмах кластеризации, которые могут помочь в обнаружении скрытых структур либо информации в данных. Сначала рассматривался подход на основе прототипов — алгоритм K-Means, который кластеризирует образцы в сферические формы, базируясь на указанном количестве центроидов кластеров. Поскольку кластеризация является методом без учителя, мы лишены роскоши наличия изначально точных меток для оценки эффективности модели. Таким образом, мы применяем внутренние метрики эффективности вроде метода локтя или анализа силуэтов как попытки количественно оценить качество кластеризации.

Затем мы взглянули на другой подход к кластеризации — агломеративную иерархическую кластеризацию. Иерархическая кластеризация не требует предварительного указания количества кластеров, а результаты могут быть визуализированы в виде дендрограмм, которые помогают интерпретировать результаты. Последним алгоритмом кластеризации, который мы обсудили, был DBSCAN — алгоритм, группирующий точки на основе локальных плотностей и способный обрабатывать выбросы, а также идентифицировать несферические формы.

После такого экскурса в область обучения без учителя самое время представить некоторые из наиболее захватывающих алгоритмов МО для обучения с учителем: многослойные искусственные нейронные сети. Пройдя через недавнее возрождение, нейронные сети еще раз стали самой горячей темой для исследовательских работ в области МО. Благодаря разработанным в последнее время алгоритмам глубокого обучения нейронные сети считаются передовой технологией для решения многих сложных задач, таких как классификация изображений и распознавание речи. В главе 12 мы построим собственную многослойную нейронную сеть. В главе 13 мы будем работать с библиотекой TensorFlow, которая предназначена для очень эффективного обучения нейросетевых моделей с множеством слоев за счет использования графических процессоров.



# РЕАЛИЗАЦИЯ МНОГОСЛОЙНОЙ ИСКУССТВЕННОЙ НЕЙРОННОЙ СЕТИ С НУЛЯ

**К**ак вам наверняка известно, глубокому обучению уделяется большое внимание в прессе и вне всяких сомнений оно является самой горячей темой в области МО. Глубокое обучение (ГО) можно понимать как подобласть МО, которая ориентирована на эффективное обучение искусственных нейронных сетей с множеством слоев. В настоящей главе вы ознакомитесь с базовыми концепциями искусственных нейронных сетей, чтобы хорошо подготовиться к чтению последующих глав, где будут представлены основанные на Python библиотеки для ГО и архитектуры *глубоких нейронных сетей* (*deep neural network* — *DNN*), которые особенно хорошо подходят для анализа изображений и текстов.

В главе будут раскрыты следующие темы:

- представление о концепциях многослойных нейронных сетей;
- реализация с нуля фундаментального алгоритма обучения с обратным распространением для нейронных сетей;
- обучение базовой многослойной нейронной сети для классификации изображений.

## Моделирование сложных функций с помощью искусственных нейронных сетей

Наше путешествие по алгоритмам МО в этой книге начиналось с искусственных нейронов в главе 2. Искусственные нейроны исполняют роль строительных блоков для многослойных искусственных нейронных сетей, которые мы обсудим в данной главе.

Базовая концепция, лежащая в основе искусственных нейронных сетей, опиралась на гипотезы и модели функционирования человеческого мозга при решении сложных задач. Хотя в последние годы искусственные нейронные сети приобрели высокую популярность, ранние исследования нейронных сетей уходят корнями в 1940-е годы, когда Уоррен Мак-Каллок и Уолтер Питтс впервые описали, как могли бы работать нейроны (“A logical calculus of the ideas immanent in nervous activity”) (Логическое исчисление идей, присущих нервной деятельности), У. Мак-Каллок и У. Питтс, *The Bulletin of Mathematical Biophysics*, 5(4): с. 115–133, 1943 г.).

Тем не менее, в течение десятилетий, следующих за первой реализацией модели *нейрона Мак-Каллока-Питтса* — персептрона Розенблатта в 1950-х годах, — многие исследователи и специалисты-практики в области МО начали постепенно терять интерес к нейронным сетям, т.к. никто не располагал хорошим решением для обучения нейронной сети с множеством слоев. Затем интерес к нейронным сетям снова вспыхнул в 1986 году, когда Д. Румельхарт, Дж. Хинтон и Р. Уильямс (повторно) открыли и популяризировали алгоритм обратного распространения для более рационального обучения нейронных сетей, который более подробно обсуждается позже в главе (“Learning representations by back-propagating errors”) (Изучение представлений путем обратного распространения ошибок), Д. Румельхарт, Дж. Хинтон, Р. Уильямс, *Nature*, 323 (6088): с. 533–536, 1986 г.). Читатели, интересующиеся историей *искусственного интеллекта* (*artificial intelligence* — AI), МО и нейронных сетей, могут ознакомиться со статьей в Википедии, посвященной так называемой *зиме искусственного интеллекта* — периоду времени, в течение которого значительная часть научного сообщества утратила интерес к исследованию нейронных сетей ([https://ru.wikipedia.org/wiki/Зима\\_искусственного\\_интеллекта](https://ru.wikipedia.org/wiki/Зима_искусственного_интеллекта)).

Однако нейронные сети никогда не были настолько популярными, как в наши дни, благодаря многим крупным открытиям, совершенным в предыду-

щем десятилетия, которые в итоге привели к тому, что мы теперь называем алгоритмами и архитектурами ГО — нейронными сетями, состоящими из множества слоев. Нейронные сети являются горячей темой не только в научном сообществе, но также в больших технологических компаниях вроде Facebook, Microsoft, Amazon, Uber и Google, делающих крупные инвестиции в искусственные нейронные сети и исследования ГО.

На сегодняшний день сложные нейронные сети, приводимые в действие алгоритмами ГО, считаются современными решениями таких сложных задач, как распознавание изображений и речи. В число широко распространенных примеров повседневно применяемых продуктов, движимых ГО, входят поиск картинок и переводчик Google — приложение для смартфонов, которое способно автоматически распознавать текст в изображениях с целью его перевода в реальном времени более чем на 20 языков.

Крупные технологические и фармацевтические компании создали множество захватывающих приложений на основе сетей DNN, далеко не полный список которых представлен ниже:

- приложение DeepFace для снабжения метками изображений от Facebook (“DeepFace: Closing the Gap to Human-Level Performance in Face Verification” (DeepFace: сокращение отрыва от человеческого уровня эффективности в верификации лиц), Я. Тейгман, М. Янь, М. Ранзато и Л. Вулф, Конференция по компьютерному зрению и распознаванию образов IEEE (CVPR), с. 1701–1708 (2014 г.));
- приложение DeepSpeech от Baidu, которое способно обрабатывать голосовые запросы на китайском языке (“DeepSpeech: Scaling up end-to-end speech recognition” (DeepSpeech: масштабирование сквозного распознавания речи), А. Ханнун, К. Кейс, Дж. Каспер, Б. Катанзаро, Г. Даймос, Э. Элсен, Р. Пренджер, С. Сатиш, Ш. Сенгупта, А. Коутс и Э. Ын, препринт arXiv:1412.5567 (2014 г.));
- новая служба перевода на разные языки от Google (“Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation” (Нейронная система машинного перевода Google: преодоление разрыва между человеческим и машинным переводом), препринт arXiv:1412.5567 (2016 г.));



- новаторские приемы для обнаружения наркотиков и прогнозирования токсичности (“Toxicity prediction using Deep Learning” (Прогнозирование токсичности с использованием глубокого обучения), А. Майр, Г. Кламбауэр, Т. Унтерсинер, С. Хохрайтер, препринт arXiv:1503.01445 (2015 г.));
- мобильное приложение, которое способно обнаруживать рак кожи с точностью, аналогичной точности диагностики профессионально обученными дерматологами (“Dermatologist-level classification of skin cancer with deep neural networks” (Классификация рака кожи уровня дерматологов с помощью глубоких нейронных сетей), А. Эстева, Б. Купрель, Р. Новоа, Дж. Ко, С. Светтер, Х. Блау и С. Трун, Nature 542, номер 7639, с. 115–118 (2017 г.));
- прогнозирование трехмерной структуры белка по последовательностям генов (“De novo structure prediction with deep-learning based scoring” (Снова о прогнозировании структуры с помощью глубокого обучения на основе подсчета), Р. Эванс, Д. Джампер, Д. Киркпатрик, Л. Сифр, Т. Грин, С. Цинь, А. Зидек, С. Нельсон, А. Бридгленд, Х. Пенедонес, С. Петерсен, К. Симонян, С. Кроссан, Д. Джонс, Д. Сильвер, К. Кавукчуоглу, Д. Хассабис и Э. Сеньор, в тринадцатом эксперименте CASP (Critical Assessment of Techniques for Protein Structure Prediction — критическая оценка прогнозирования белковых структур), 1–4 декабря 2018 г.);
- обучение вождению при плотном трафике на основе исключительно данных наблюдений, таких как видеопотоки из камер (“Model-predictive policy learning with uncertainty regularization for driving in dense traffic” (Политика обучения прогнозирующих моделей с регуляризацией неопределенности для вождения при плотном трафике), М. Хенафф, А. Чанзани, Я. Лекун, Conference Proceedings of the International Conference on Learning Representations, ICLR, 2019 г.).

## Краткое повторение однослойных нейронных сетей

Настоящая глава целиком посвящена многослойным нейронным сетям, особенностям их работы и способам их обучения для решения сложных задач. Тем не менее, прежде чем погружаться в исследование конкретной архитектуры многослойной нейронной сети, давайте кратко повторим ряд

концепций однослойных нейронных сетей, которые мы представляли в главе 2, а именно — алгоритма *адаптивного линейного нейрона (Adaline)*, показанного на рис. 12.1.

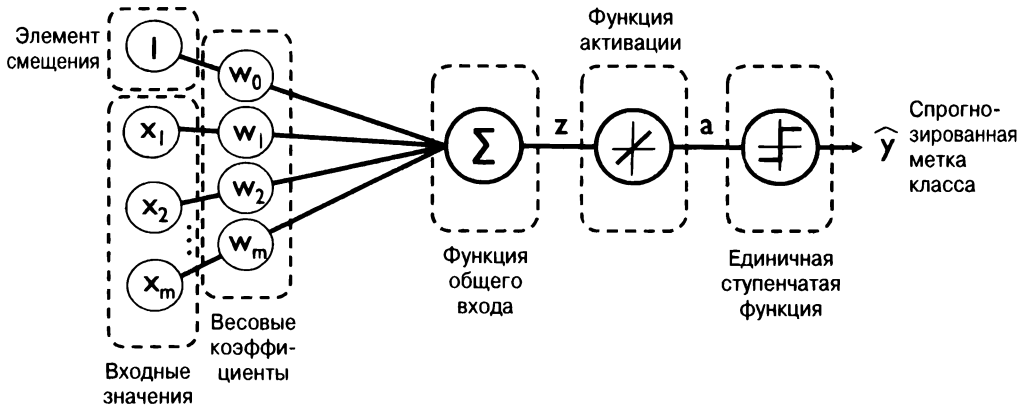


Рис. 12.1. Алгоритм Adaline

В главе 2 мы реализовали алгоритм Adaline для выполнения двоичной классификации и применяли оптимизацию методом градиентного спуска, чтобы выяснить весовые коэффициенты модели. В каждой эпохе (проходе по обучающему набору) мы обновляли весовой вектор  $w$ , используя следующее правило обновления:

$$w := w + \Delta w, \text{ где } \Delta w = -\eta \nabla J(w)$$

Другими словами, мы вычисляли градиент на основе полного обучающего набора данных и обновляли веса модели, делая шаг в направлении, противоположном градиенту  $\nabla J(w)$ . Для нахождения оптимальных весов модели мы оптимизировали целевую функцию, которую определяли как функцию издержек  $J(w)$  в форме суммы квадратичных ошибок (SSE). Вдобавок мы умножали градиент на коэффициент — *скорость обучения*  $\eta$ , который должен был тщательно подбираться, чтобы обеспечить приемлемый баланс между темпом обучения и риском проскочить глобальный минимум функции издержек.

При оптимизации посредством градиентного спуска мы обновляем все веса одновременно после каждой эпохи, и вот как определяется частная производная для каждого веса  $w_j$  в весовом векторе  $w$ :

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = -\sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}$$

Здесь  $y^{(i)}$  — целевая метка класса индивидуального образца  $x^{(i)}$  и  $a^{(i)}$  — активация нейрона, которая в особом случае алгоритма Adaline представляет собой линейную функцию.

Кроме того, мы определили функцию активации  $\phi(\cdot)$  следующим образом:

$$\phi(z) = z = a$$

Здесь общий вход  $z$  является линейной комбинацией весов, которые соединяют входной слой с выходным:

$$z = \sum_j w_j x_j = \mathbf{w}^T \mathbf{x}$$

Наряду с применением функции активации  $\phi(z)$  для вычисления обновления градиентов мы реализовали пороговую функцию для заталкивания выхода с непрерывными значениями в двоичные метки классов с целью выработки прогноза:

$$\hat{y} = \begin{cases} 1, & \text{если } g(z) \geq 0; \\ -1 & \text{в противном случае} \end{cases}$$



На  
заметку!

### Соглашение относительно названия “однослойная сеть”

Обратите внимание, что хотя сеть Adaline состоит из двух слоев, одного входного и одного выходного, она называется однослойной сетью из-за наличия единственной связи между входным и выходным слоями.

Мы также представили некоторый *трюк* для ускорения процесса обучения модели — так называемую оптимизацию методом *стохастического градиентного спуска*. Стохастический градиентный спуск аппроксимирует издержки из одиночного обучающего образца (динамическое обучение) или небольшого поднабора обучающих образцов (мини-пакетное обучение). Мы задействуем эту концепцию позже в главе, когда реализуем и обучим многослойный персептрон. Помимо более быстрого обучения — вследствие более частых обновлений весов в сравнении с градиентным спуском — его зашум-

ленная природа также расценивается как преимущество при обучении многослойных нейронных сетей с нелинейными функциями активации, которые не имеют выпуклой функции издержек. В такой ситуации добавленный шум может помочь избежать локальных минимумов издержек, но мы обсудим данную тему более подробно позже в главе.

## Введение в архитектуру многослойных нейронных сетей

В текущем разделе мы выясним, как подключать множество одиночных нейронов к многослойной нейронной сети прямого распространения (*feedforward neural network*); специальный тип *полносвязной сети* подобного рода называется также *многослойным персептроном* (*multilayer perceptron* — *MLP*). На рис. 12.2 иллюстрируется концепция MLP из трех слоев.

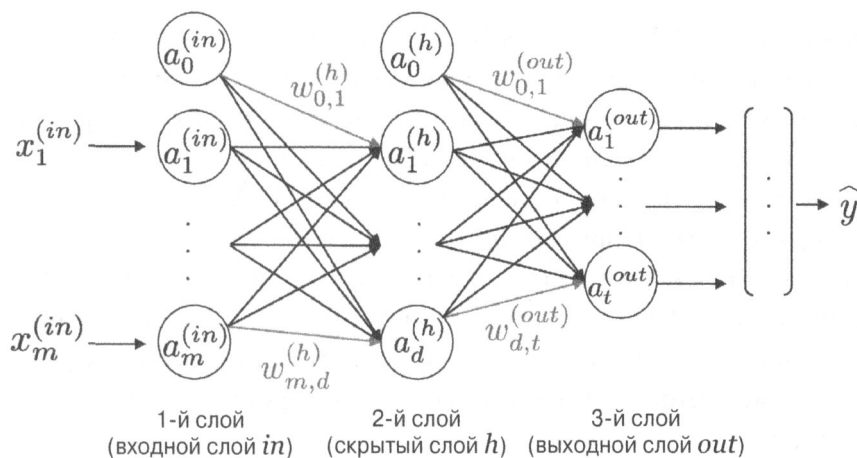


Рис. 12.2. Концепция MLP

Многослойный персептрон на рис. 12.2 имеет один входной слой, один скрытый слой и один выходной слой. Элементы в скрытом слое полносвязны с входным слоем, а выходной слой полносвязен со скрытым слоем. Если такая сеть имеет более одного скрытого слоя, тогда мы называем ее *глубокой искусственной нейронной сетью*.



### Добавление дополнительных скрытых слоев

Для создания более глубоких сетевых архитектур мы можем добавлять к MLP любое количество скрытых слоев. С практической точки зрения мы можем считать количество слоев и элементов в нейронной сети дополнительными гиперпараметрами, которые нужно оптимизировать для имеющейся задачи с использованием методики перекрестной проверки, рассмотренной в главе 6.

Однако градиенты ошибок, которые мы позже вычисляем через обратное распространение, по мере добавления дополнительных слоев к сети будут все больше и больше снижаться. Такая проблема исчезновения градиентов делает обучение модели более трудным. По этой причине были разработаны специальные алгоритмы, помогающие обучать структуры глубоких нейронных сетей, и результирующий процесс стал известен как *глубокое обучение*.

Согласно рис. 12.2, мы обозначаем  $i$ -тый элемент активации в  $l$ -том слое как  $a_i^{(l)}$ . Чтобы сделать математические выкладки и реализацию в коде более интуитивно понятными, для ссылки на слои мы будем применять не числовые индексы, а использовать надстрочный индекс *in* для входного слоя, надстрочный индекс *h* для скрытого слоя и надстрочный индекс *out* для выходного слоя. Например,  $a_i^{(in)}$  ссылается на  $i$ -тое значение во входном слое,  $a_i^{(h)}$  — на  $i$ -тый элемент в скрытом слое и  $a_i^{(out)}$  — на  $i$ -тый элемент в выходном слое. Здесь элементы активации  $a_0^{(in)}$  и  $a_0^{(h)}$  представляют собой *элементы смещения* (*bias unit*), которые мы устанавливаем равными 1. Активация элементов во входном слое — это просто его входы плюс элемент смещения:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$



### Соглашение об обозначении элементов смещения

Позже в главе мы реализуем многослойный персептрон с применением отдельных векторов для элементов смещения, что сделает кодовую реализацию рациональнее и легче в восприятии. Такой принцип используется и в TensorFlow — библиотеке для ГО, которую мы представим в главе 13. Тем не менее, последующие математические уравнения показались бы более сложными или запутанными, если бы нам пришлось работать с дополнительными переменными для смещения. Однако обратите внимание, что вычисление посредством дополнения единицами входного вектора (как было показано ранее) и применения переменной веса в качестве смещения — в точности то же самое, что и оперирование с отдельными векторами смещения; это только другое соглашение.

Каждый элемент в слое  $l$  связан со всеми элементами в слое  $l+1$  через весовой коэффициент. Скажем, связь между  $k$ -тым элементом в слое  $l$  и  $j$ -тым элементом в слое  $l+1$  будет записываться как  $w_{kj}^{(l)}$ . Возвращаясь к рис. 12.2, мы обозначаем весовую матрицу, которая связывает входной слой со скрытым слоем, как  $\mathbf{W}^{(h)}$ , а матрицу, связывающую скрытый слой с выходным слоем, как  $\mathbf{W}^{(out)}$ .

Хотя одного элемента в выходном слое было бы достаточно для задачи двоичной классификации, на рис. 12.2 мы видели более общую форму нейронной сети, которая позволяет выполнять многоклассовую классификацию посредством обобщения в соответствии с методикой “один против всех” (OvA). Чтобы лучше понять, как она работает, вспомним представление на основе унитарного кодирования категориальных переменных, которое было введено в главе 4. Например, мы можем закодировать три метки классов в знакомом наборе данных Iris (0=Setosa, 1=Versicolor, 2=Virginica) следующим образом:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Такое векторное представление на основе унитарного кодирования позволяет решать задачи классификации с произвольным количеством уникальных меток классов, присутствующих в обучающем наборе.

Если вы — новичок в представлениях нейронных сетей, то индексная система обозначений (подстрочные и надстрочные индексы) поначалу может показаться слегка сбивающей с толку. То, что выглядит чрезмерно сложным на первых порах, обретет гораздо больше смысла в последующих разделах, когда мы векторизуем представление нейронной сети. Как было указано ранее, мы сводим веса связей между входным и скрытым слоями в матрицу  $W^{(h)} \in \mathbb{R}^{m \times d}$ , где  $d$  — количество скрытых элементов, а  $m$  — количество входных элементов, включая элемент смещения. Поскольку эту систему обозначений важно усвоить для понимания концепций, обсуждаемых позже в главе, мы подведем итог приведенным выше сведениям на описательной иллюстрации многослойного персептрона 3-4-3 (рис. 12.3).

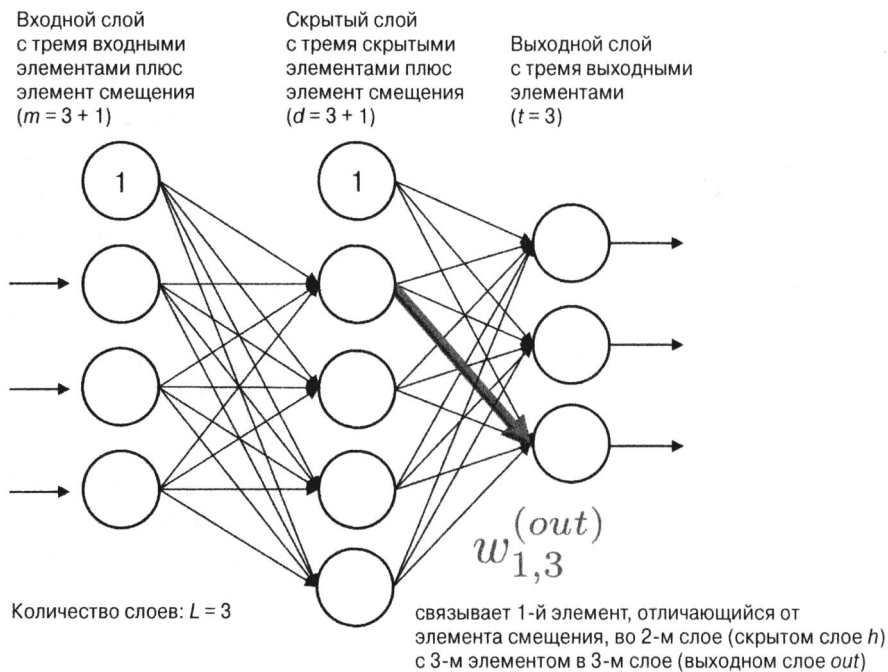


Рис. 12.3. Многослойный персептрон 3-4-3

## Активация нейронной сети посредством прямого распространения

В этом разделе мы опишем процесс *прямого распространения* для расчета выхода модели MLP. Чтобы понять, как он вписывается в контекст обучения модели MLP, давайте сведем процедуру обучения MLP к трем простым шагам.

1. Начиная с входного слоя, мы распространяем в прямом направлении образцы обучающих данных по сети для генерации выхода.
2. На основе выхода сети мы подсчитываем ошибку, подлежащую сведению к минимуму с использованием функции издержек, которая будет описана позже.
3. Мы распространяем в обратном направлении ошибку, находим ее производную по каждому весу в сети и обновляем модель.

Наконец, после повторения указанных выше шагов для множества эпох и выяснения весов модели MLP с помощью прямого распространения мы рассчитываем выход сети и применяем описанную в предыдущем разделе пороговую функцию, чтобы получить спрогнозированные метки классов, представленные в унитарном коде.

А теперь давайте пройдемся по индивидуальным шагам прямого распространения для порождения выхода из шаблонов в обучающих данных. Так как каждый элемент в скрытом слое связан со всеми элементами во входном слое, мы сначала вычисляем элемент активации скрытого слоя  $a_1^{(h)}$  следующим образом:

$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \dots + a_m^{(in)} w_{m,1}^{(h)}$$

$$a_1^{(h)} = \phi(z_1^{(h)})$$

Здесь  $z_1^{(h)}$  — общий вход и  $\phi(\cdot)$  — функция активации, которая должна быть дифференцируемой для выяснения весов связей между нейронами с использованием подхода на основе градиентов. Чтобы иметь возможность решать сложные задачи, такие как классификация изображений, в модели MLP нам нужны нелинейные функции активации, например, сигмоидальная (логистическая) функция активации, которая применялась в разделе, посвященном логистической регрессии, главы 3:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

Несложно вспомнить, что сигмоидальная функция представляет собой S-образную кривую, которая отображает общий вход  $z$  на логистическое распределение в диапазоне от 0 до 1, пересекая ось  $y$  в точке  $z = 0$  (рис. 12.4).



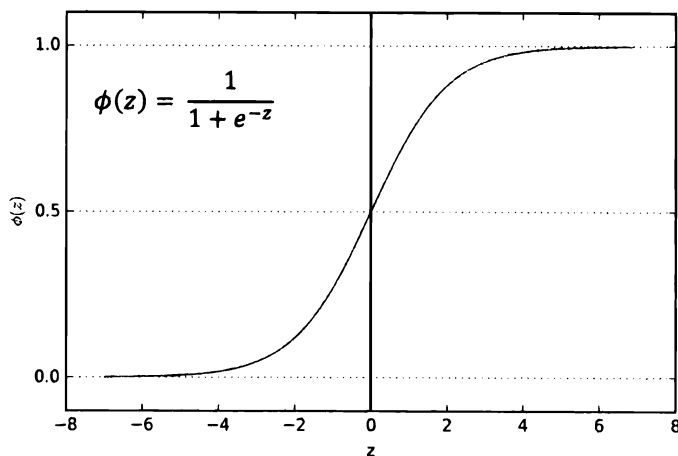


Рис. 12.4. График сигмоидальной функции

Многослойный персептрон (MLP) — типичный пример нейронной сети прямого распространения. Термин “прямое распространение” относится к тому факту, что каждый слой служит входом в следующий слой без циклов по контрасту с рекуррентными нейронными сетями — архитектурой, которую мы кратко затронем далее в текущей главе и более подробно обсудим в главе 16. Термин “многослойный персептрон” может показаться несколько путающим, поскольку искусственные нейроны в такой сетевой архитектуре — это обычно сигмоидальные элементы, а не персептроны. Мы можем думать о нейронах в MLP как о логистических регрессионных элементах, которые возвращают значения из непрерывного диапазона между 0 и 1.

В целях продуктивности и читабельности кода мы запишем активацию в более компактной форме с использованием концепций базовой линейной алгебры, что позволит векторизовать реализацию посредством библиотеки NumPy, а не применять множество вложенных и затратных в вычислительном плане циклов `for` языка Python:

$$\mathbf{z}^{(h)} = \mathbf{a}^{(in)} \mathbf{W}^{(h)}$$

$$\mathbf{a}^{(h)} = \phi(\mathbf{z}^{(h)})$$

Здесь  $\mathbf{a}^{(in)}$  —  $(1 \times m)$ -мерный вектор признаков образца  $\mathbf{x}^{(in)}$  плюс элемент смещения, а  $\mathbf{W}^{(h)}$  —  $(m \times d)$ -мерная матрица весов, где  $d$  — количество элементов в скрытом слое. После умножения матрицы на вектор мы получаем  $(1 \times d)$ -мерный вектор общего входа  $\mathbf{z}^{(h)}$  для расчета активации  $\mathbf{a}^{(h)}$  (где  $\mathbf{a}^{(h)} \in \mathbb{R}^{1 \times d}$ ).

Кроме того, мы можем обобщить это вычисление на все  $n$  образцов в обучающем наборе:

$$\mathbf{Z}^{(h)} = \mathbf{A}^{(in)} \mathbf{W}^{(h)}$$

Теперь  $\mathbf{A}^{(in)}$  —  $(n \times m)$ -мерная матрица, а умножение матрицы на матрицу даст в результате  $(n \times d)$ -мерную матрицу общего входа  $\mathbf{Z}^{(h)}$ . Наконец, мы применяем к каждому значению в матрице общего входа функцию активации  $\phi(\cdot)$ , чтобы получить  $(n \times d)$ -мерную матрицу активации  $\mathbf{A}^{(h)}$  для следующего слоя (в данном случае выходного слоя):

$$\mathbf{A}^{(h)} = \phi(\mathbf{Z}^{(h)})$$

Аналогично мы можем записать активацию выходного слоя в векторизованной форме для множества образцов:

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)} \mathbf{W}^{(out)}$$

Мы перемножаем  $(d \times t)$ -мерную матрицу  $\mathbf{W}^{(out)}$  ( $t$  — количество выходных элементов) и  $(n \times d)$ -мерную матрицу  $\mathbf{A}^{(h)}$ , чтобы получить  $(n \times t)$ -мерную матрицу  $\mathbf{Z}^{(out)}$  (столбцы в ней представляют выходы для каждого образца).

В заключение мы применяем сигмоидальную функцию активации для получения выхода в виде непрерывных значений нашей сети:

$$\mathbf{A}^{(out)} = \phi(\mathbf{Z}^{(out)}), \quad \mathbf{A}^{(out)} \in \mathbb{R}^{n \times d}$$

## Классификация рукописных цифр

В предыдущем разделе мы привели много теоретических сведений о нейронных сетях, которые могут слегка ошеломить новичков в данной теме. Прежде чем продолжить обсуждение алгоритма для выяснения весов модели MLP — обратного распространения, — давайте ненадолго отвлечемся от теории и посмотрим на нейронную сеть в действии.



На заметку!

### Дополнительные ресурсы по обратному распространению

Теория нейронных сетей может оказаться довольно сложной для освоения, поэтому рекомендуется обратиться к двум дополнительным ресурсам, которые глубже раскрывают концепции, представленные в настоящей главе:

- глава 6, “Deep Feedforward Networks” (Глубокие нейронные сети прямого распространения), из книги “Deep Learning” (Глубокое обучение), Я. Гудфеллоу, Й. Бенджи и А. Корвилл, MIT Press (2016 г.), свободно доступной по ссылке <http://www.deeplearningbook.org>;
- книга “Pattern Recognition and Machine Learning” (“Распознавание образов и машинное обучение”, пер. с англ., изд. “Диалектика”, 2020 г.), К. Бишоп и другие, том 1, Springer New York (2006 г.);
- Слайды из курса лекций по глубокому обучению, читаемого в Висконсинском университете в Мэдисоне:  
[https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L08\\_logistic\\_slides.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L08_logistic_slides.pdf)  
[https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L09\\_mlp\\_slides.pdf](https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L09_mlp_slides.pdf)

В текущем разделе мы реализуем и обучим нашу первую многослойную нейронную сеть для классификации рукописных цифр из широко известного набора данных *MNIST* (*Mixed National Institute of Standards and Technology* — смешанный набор данных Национального института стандартов и технологий США), который был создан Яном Лекуном и другими и служит популярным эталонным набором при испытаниях алгоритмов МО (“Gradient-Based Learning Applied to Document Recognition” (Основанное на градиентах обучение, примененное к распознаванию документов), Я. Леун, Л. Ботту, Й. Бенджи и П. Хаффнер, работы IEEE, 86(11): с. 2278–2324 (ноябрь 1998 г.)).

## Получение и подготовка набора данных MNIST

Набор данных MNIST публично доступен по ссылке <http://yann.lecun.com/exdb/mnist/> и состоит из следующих четырех частей:

- *изображения обучающего набора данных* — `train-images-idx3-ubyte.gz` (9.5 Мбайт, 47 Мбайт в распакованном виде, 60 000 образцов);
- *метки обучающего набора данных* — `train-labels-idx1-ubyte.gz` (29 Кбайт, 60 Кбайт в распакованном виде, 60 000 меток);
- *изображения испытательного набора данных* — `t10k-images-idx3-ubyte.gz` (1.6 Мбайт, 7.8 Мбайт в распакованном виде, 10 000 образцов);
- *метки испытательного набора данных* — `t10k-labels-idx1-ubyte.gz` (4.5 Кбайт, 10 Кбайт в распакованном виде, 10 000 меток).

Набор данных MNIST был создан из двух наборов данных *Национального института стандартов и технологий США* (*US National Institute of Standards and Technology — NIST*). Обучающий набор данных содержит рукописные цифры от 250 разных людей, 50% которых были учащимися средней школы, а другие 50% — служащими Бюро переписи населения США. Обратите внимание, что испытательный набор данных включает рукописные цифры от разных людей в той же пропорции.

После загрузки файлы понадобится распаковать утилитой `gzip` для Unix/Linux с помощью приведенной ниже команды, вводимой в локальном каталоге с загруженными файлами MNIST:

```
gzip *ubyte.gz -d
```

При работе в среде Microsoft Windows можно воспользоваться любым предпочитаемым инструментом для распаковки.

Изображения хранятся в байтовом формате, и мы прочитаем их в массивы NumPy, которые будут применяться для обучения и испытания нашей реализации MLP. Для этого мы определяем следующую вспомогательную функцию:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Загружает данные MNIST из пути 'path'"""
    labels_path = os.path.join(path,
                                '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path,
                                '%s-images-idx3-ubyte' % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                lbpath.read(8))
        labels = np.fromfile(lbpath,
                              dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                                imgpath.read(16))
```

```

images = np.fromfile(imgpath,
                      dtype=np.uint8).reshape(
                      len(labels), 784)
images = ((images / 255.) - .5) * 2
return images, labels

```

Функция `load_mnist` возвращает два массива, первый из которых является  $(n \times m)$ -мерным массивом NumPy (`images`), где  $n$  — количество образцов и  $m$  — число признаков (точек в нашем случае). Обучающий набор состоит из 60 000 обучающих образцов цифр, а испытательный набор — из 10 000 образцов.

Изображения в наборе данных MNIST содержат  $28 \times 28$  пикселей, причем каждый пиксель представлен значением интенсивности в оттенках серого. Мы разворачиваем изображение из  $28 \times 28$  пикселей в одномерные векторы-строки, которые представляют строки в массиве `images` (784 значения на строку или изображение). Второй массив (`labels`), возвращаемый функцией `load_mnist`, содержит соответствующую целевую переменную — метки классов (целые числа от 0 до 9) рукописных цифр. Способ чтения изображения поначалу может показаться несколько странным:

```

magic, n = struct.unpack('>II', lopath.read(8))
labels = np.fromfile(lopath, dtype=np.uint8)

```

Чтобы понять, как работают эти две строки кода, давайте посмотрим описание набора данных MNIST, доступное по ссылке <http://yann.lecun.com/exdb/mnist/>:

[смещение]	[тип]	[значение]	[описание]
0000	32-битное целое	0x00000801 (2049)	магическое число (сначала старший бит)
0004	32-битное целое	60000	количество элементов
0008	байт без знака	??	метка
0009	байт без знака	??	метка
.....			
xxxx	байт без знака	??	метка

С помощью приведенных ранее двух строк кода мы сначала читаем из файлового буфера магическое число, которое является описанием файлового протокола, и количество элементов ( $n$ ), а затем с использованием метода `fromfile` загружаем последующие байты в массив NumPy.

Значение '>II' параметра `fmt`, передаваемое как аргумент функции `struct.unpack`, можно разбить на две части:

- > указывает на порядок хранения последовательности байтов — от старшего к младшему; если вам не знакомо понятие порядка от старшего к младшему и от младшего к старшему, тогда почитайте статью “Порядок байтов” в Википедии ([https://ru.wikipedia.org/wiki/Порядок\\_байтов](https://ru.wikipedia.org/wiki/Порядок_байтов));
- I указывает на целое число без знака.

Наконец, мы также нормализуем значения пикселей MNIST в диапазоне от  $-1$  до  $1$  (первоначально было от  $0$  до  $255$ ) посредством следующей строки кода:

```
images = ((images / 255.) - .5) * 2
```

Как обсуждалось в главе 2, причина в том, что при таких условиях градиентная оптимизация будет гораздо более стабильной. Обратите внимание на масштабирование изображений на пиксельной основе, что отличается от подхода масштабирования признаков, который был принят в предшествующих главах.

Ранее мы выводили параметры масштабирования из обучающего набора и применяли их для масштабирования каждого столбца в обучающем и испытательном наборах. Тем не менее, при работе с пикселями изображений их центрирование относительно нуля и масштабирование в диапазоне  $[-1, 1]$  — распространенный подход, показывающий хорошие результаты на практике.



На заметку!

### Пакетная нормализация

Распространенным трюком, направленным на улучшение сходимости градиентной оптимизации через масштабирование входов, является *пакетная нормализация*, представляющая собой сложную тему, которая будет раскрыта в главе 17. Кроме того, вы можете ознакомиться с дополнительными сведениями о пакетной нормализации, прочитав великолепную статью Сергея Иоффе и Кристиана Сегеди “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” (Пакетная нормализация: ускорение обучения глубоких сетей путем сокращения внутреннего ковариационного сдвига), написанную в 2015 году (<https://arxiv.org/abs/1502.03167>).

Выполнив приведенный далее код, мы загрузим 60 000 обучающих образцов и 10 000 испытательных образцов из локального каталога, где производилась распаковка архивов с набором данных MNIST. (В следующем фрагменте кода предполагается, что загруженные файлы MNIST были распакованы в том же каталоге, в котором выполняется код.)

```
>>> X_train, y_train = load_mnist('', kind='train')
>>> print('Строк: %d, столбцов: %d'
...       % (X_train.shape[0], X_train.shape[1]))
Строк: 60000, столбцов: 784

>>> X_test, y_test = load_mnist('', kind='t10k')
>>> print('Строк: %d, столбцов: %d'
...       % (X_test.shape[0], X_test.shape[1]))
Строк: 10000, столбцов: 784
```

Чтобы получить представление о том, как выглядят изображения в наборе данных MNIST, давайте визуализируем примеры цифр 0–9 после восстановления 784-пиксельных векторов из матрицы признаков в исходное изображение  $28 \times 28$ , которое мы можем вывести посредством функции `imshow` библиотеки `Matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                         sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X_train[y_train == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')

>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Мы должны увидеть график из  $2 \times 5$  фигур, показывающих типичное изображение каждой уникальной цифры (рис. 12.5).

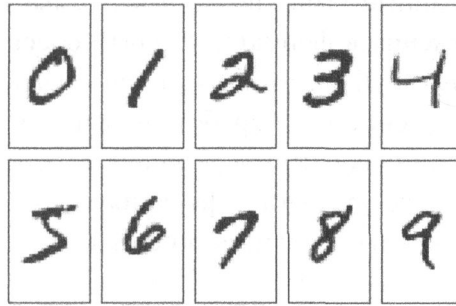


Рис. 12.5. Типичные изображения цифр

Вдобавок мы выведем также примеры одной цифры, чтобы посмотреть, как на самом деле отличается почерк:

```
>>> fig, ax = plt.subplots(nrows=5,
...                          ncols=5,
...                          sharex=True,
...                          sharey=True)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X_train[y_train == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

После выполнения кода мы должны увидеть первые 25 вариантов написания цифры 7 (рис. 12.6).

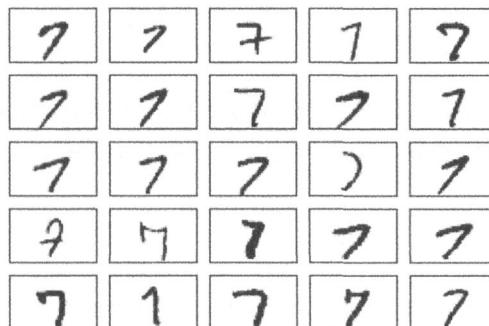


Рис. 12.6. Варианты написания цифры 7



Завершив все описанные выше шаги, имеет смысл сохранить отмасштабированные изображения в формате, который обеспечит более быструю загрузку в новом сеансе Python, чтобы избежать накладных расходов, связанных с повторным чтением и обработкой данных. Когда используются массивы NumPy, рациональный и удобный способ сохранения многомерных массивов на диск предлагает функция `savez` библиотеки NumPy. (Официальная документация находится по ссылке <https://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html>.)

Выражаясь кратко, функция `savez` аналогична модулю `pickle` из Python, который мы применяли в главе 9, но оптимизирована для сохранения массивов NumPy. Функция `savez` создает zip-архивы данных, выпуская файлы `.npz`, которые содержат файлы в формате `.npy`. Если вы хотите узнать больше об этом формате, то можете обратиться к удобному пояснению, включающему обсуждение достоинств и недостатков, в документации по NumPy: <https://docs.scipy.org/doc/numpy/reference/generated/numpy.lib.format.html#module-numpy.lib.format>. Кроме того, вместо `savez` мы будем использовать функцию `savez_compressed`, которая имеет такой же синтаксис, как у `savez`, но дополнительно сжимает выходной файл до существенно меньшего размера (в данном случае с около 400 Мбайт до приблизительно 22 Мбайт). В следующем фрагменте кода мы сохраняем обучающий и испытательный наборы данных в архивный файл `mnist_scaled.npz`:

```
>>> import numpy as np
>>> np.savez_compressed('mnist_scaled.npz',
...                     X_train=X_train,
...                     y_train=y_train,
...                     X_test=X_test,
...                     y_test=y_test)
```

После создания файлов `.npz` мы можем загружать предварительно обработанные массивы изображений MNIST с применением функции `load` библиотеки NumPy:

```
>>> mnist = np.load('mnist_scaled.npz')
```

Переменная `mnist` теперь ссылается на объект, способный получать доступ к четырем массивам данных, которые мы предоставили в ключевых

аргументах функции `savez_compressed`. Эти входные массивы теперь перечисляются в списке атрибута `files` объекта `mnist`:

```
>>> mnist.files
['X_train', 'y_train', 'X_test', 'y_test']
```

Например, для загрузки обучающих данных в текущий сеанс Python мы получаем доступ к массиву `X_train` следующим образом (подобно доступу к словарию Python):

```
>>> X_train = mnist['X_train']
```

Вот как мы можем извлечь все четыре массива данных, используя *списковое включение* (*list comprehension*):

```
>>> X_train, y_train, X_test, y_test = [mnist[f] for
...                                     f in mnist.files]
```

Обратите внимание, что хотя предшествующие примеры с функциями `np.savez_compressed` и `np.load` несущественны для выполнения кода в этой главе, они служат демонстрацией того, как удобно и рационально сохранять и загружать массивы NumPy.



На заметку!

### Загрузка набора данных MNIST с использованием `scikit-learn`

В настоящее время загрузить набор данных MNIST можно более удобно с применением новой функции `fetch_openml` из библиотеки `scikit-learn`. Например, с помощью приведенного ниже кода вы можете создать обучающий набор с 50 000 образцов и испытательный набор с 10 000 образцов, извлекая набор данных из <https://www.openml.org/d/554>:

```
>>> from sklearn.datasets import fetch_openml
>>> from sklearn.model_selection import train_test_split
>>> X, y = fetch_openml('mnist_784', version=1,
...                     return_X_y=True)
>>> y = y.astype(int)
>>> X = (X / 255.) - .5 * 2
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(
...         X, y, test_size=10000,
...         random_state=123, stratify=y)
```

Имейте в виду, что распределение записей MNIST на обучающий и испытательный наборы данных будет отличаться от такого распределения при ручном подходе, описанном ранее в разделе. Таким образом, в последующих разделах вы будете наблюдать немного другие результаты при загрузке набора данных с использованием функций `fetch_openml` и `train_test_split`.

## Реализация многослойного персептрона

В этом подразделе мы реализуем с чистого листа многослойный персептрон, предназначенный для классификации изображений в наборе данных MNIST. Чтобы не усложнять код, мы реализуем многослойный персептрон, имеющий только один скрытый слой. Поскольку на первый взгляд подход может выглядеть чуть сложнее, имеет смысл загрузить исходный код примеров из хранилища GitHub (<https://github.com/rasbt/python-machine-learning-book-3rd-edition>) и работать с реализацией MLP, снабженной комментариями и выделением синтаксиса для лучшего восприятия.

Если вы не выполняете код из сопровождающего файла Jupyter Notebook или не располагаете доступом в Интернет, тогда скопируйте код `NeuralNetMLP` из данной главы в сценарный файл Python (скажем, `neuralnet.py`) внутри текущего рабочего каталога и затем импортируйте его в текущем сеансе Python посредством следующей команды:

```
from neuralnet import NeuralNetMLP
```

Код будет содержать разделы, которые мы еще не раскрывали, такие как алгоритм обратного распространения, но большая часть кода должна выглядеть знакомой; она основана на реализации `Adaline` в главе 2 и обсуждении прямого распространения в предшествующих разделах.

Не переживайте, если не все в коде станет понятным незамедлительно; мы разберем определенные части далее в главе. Однако отслеживание кода на этой стадии может облегчить усвоение теории позже.

Ниже приведена реализация многослойного персептрона:

```
import numpy as np
import sys

class NeuralNetMLP(object):
    """ Нейронная сеть прямого распространения / классификатор
        на основе многослойного персептрона.
```

### Параметры

```

-----
n_hidden : int (по умолчанию: 30)
    Количество скрытых элементов.
l2 : float (по умолчанию: 0.)
    Значения лямбда для регуляризации L2.
    Регуляризация отсутствует, если l2=0 (принято по умолчанию).
epochs : int (по умолчанию: 100)
    Количество проходов по обучающему набору.
eta : float (по умолчанию: 0.001)
    Скорость обучения.
shuffle : bool (по умолчанию: True)
    Если True, тогда обучающие данные тасуются
    каждую эпоху, чтобы предотвратить циклы.
minibatch_size : int (по умолчанию: 1)
    Количество обучающих образцов на мини-пакет.
seed : int (по умолчанию: None)
    Случайное начальное значение для инициализации весов
    и тасования.

```

### Атрибуты

```

-----
eval_ : dict
    Словарь, в котором собираются показатели издержек,
    правильности при обучении и правильности при испытании
    для каждой эпохи во время обучения.
"""
def __init__(self, n_hidden=30,
              l2=0., epochs=100, eta=0.001,
              shuffle=True, minibatch_size=1, seed=None):
    self.random = np.random.RandomState(seed)
    self.n_hidden = n_hidden
    self.l2 = l2
    self.epochs = epochs
    self.eta = eta
    self.shuffle = shuffle
    self.minibatch_size = minibatch_size

def _onehot(self, y, n_classes):
    """ Кодирует метки в представление с унитарным кодом

```

### Параметры

```

-----
y : массив, форма = [n_examples]
    Целевые значения.

```

```

    Возвращает
    -----
    onehot : массив, форма = (n_examples, n_labels)
    """
    onehot = np.zeros((n_classes, y.shape[0]))
    for idx, val in enumerate(y.astype(int)):
        onehot[val, idx] = 1.
    return onehot.T

def _sigmoid(self, z):
    """ Вычисляет логистическую (сигмоидальную) функцию """
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def _forward(self, X):
    """ Вычисляет шаг прямого распространения """

    # шаг 1: общий вход скрытого слоя
    # скалярное произведение [n_examples, n_features]
    #                и [n_features, n_hidden]
    # -> [n_examples, n_hidden]
    z_h = np.dot(X, self.w_h) + self.b_h

    # шаг 2: активация скрытого слоя
    a_h = self._sigmoid(z_h)

    # шаг 3: общий вход выходного слоя
    # скалярное произведение [n_examples, n_hidden]
    #                и [n_hidden, n_classlabels]
    # -> [n_examples, n_classlabels]
    z_out = np.dot(a_h, self.w_out) + self.b_out

    # шаг 4: активация выходного слоя
    a_out = self._sigmoid(z_out)

    return z_h, a_h, z_out, a_out

def _compute_cost(self, y_enc, output):
    """ Вычисляет функцию издержек.

    Параметры
    -----
    y_enc : массив, форма = (n_examples, n_labels)
            Метки классов в унитарном коде.
    output : массив, форма = [n_examples, n_output_units]
            Активация выходного слоя (прямое распространение)

    Возвращает
    -----

```

```

cost : float
    Регуляризированные издержки
"""
L2_term = (self.l2 *
            (np.sum(self.w_h ** 2.) +
             np.sum(self.w_out ** 2.)))

term1 = -y_enc * (np.log(output))
term2 = (1. - y_enc) * np.log(1. - output)
cost = np.sum(term1 - term2) + L2_term
return cost

def predict(self, X):
    """ Прогнозирует метки классов.

    Параметры
    -----
    X : массив, форма = [n_examples, n_features]
        Входной слой с первоначальными признаками.

    Возвращает
    -----
    y_pred : массив, форма = [n_examples]
        Спрогнозированные метки классов.

    """
    z_h, a_h, z_out, a_out = self._forward(X)
    y_pred = np.argmax(z_out, axis=1)
    return y_pred

def fit(self, X_train, y_train, X_valid, y_valid):
    """ Выясняет веса из данных.

    Параметры
    -----
    X_train : массив, форма = [n_examples, n_features]
        Входной слой с первоначальными признаками.
    y_train : массив, форма = [n_examples]
        Целевые метки классов.
    X_valid : array, shape = [n_examples, n_features]
        Признаки образцов для проверки во время обучения
    y_valid : массив, форма = [n_examples]
        Метки образцов для проверки во время обучения

    Возвращает
    -----
    self
    """

```

```

n_output = np.unique(y_train).shape[0]    # количество
                                           # меток классов

n_features = X_train.shape[1]

#####
# Инициализация весов #
#####

# веса для входного слоя -> скрытого слоя
self.b_h = np.zeros(self.n_hidden)
self.w_h = self.random.normal(loc=0.0, scale=0.1,
                               size=(n_features,
                                      self.n_hidden))

# веса для скрытого слоя -> выходного слоя
self.b_out = np.zeros(n_output)
self.w_out = self.random.normal(loc=0.0, scale=0.1,
                                 size=(self.n_hidden,
                                       n_output))

epoch_strlen = len(str(self.epochs)) # для формата
self.eval_ = {'cost': [], 'train_acc': [], 'valid_acc': \
              []}

y_train_enc = self._onehot(y_train, n_output)

# итерация по эпохам обучения
for i in range(self.epochs):

    # итерация по мини-пакетам
    indices = np.arange(X_train.shape[0])

    if self.shuffle:
        self.random.shuffle(indices)

    for start_idx in range(0, indices.shape[0] - \
                           self.minibatch_size + \
                           1, self.minibatch_size):
        batch_idx = indices[start_idx:start_idx + \
                           self.minibatch_size]

        # прямое распространение
        z_h, a_h, z_out, a_out = \
            self._forward(X_train[batch_idx])

        #####
        # Обратное распространение #
        #####

```

```

# [n_examples, n_classlabels]
delta_out = a_out - y_train_enc[batch_idx]

# [n_examples, n_hidden]
sigmoid_derivative_h = a_h * (1. - a_h)

# скалярное произведение [n_examples, n_classlabels]
# и [n_classlabels, n_hidden]
# -> [n_examples, n_hidden]
delta_h = (np.dot(delta_out, self.w_out.T) *
           sigmoid_derivative_h)

# скалярное произведение [n_features, n_examples]
# и [n_examples, n_hidden]
# -> [n_features, n_hidden]
grad_w_h = np.dot(X_train[batch_idx].T, delta_h)
grad_b_h = np.sum(delta_h, axis=0)

# скалярное произведение [n_hidden, n_examples]
# и [n_examples, n_classlabels]
# -> [n_hidden, n_classlabels]
grad_w_out = np.dot(a_h.T, delta_out)
grad_b_out = np.sum(delta_out, axis=0)

# Регуляризация и обновления весов
delta_w_h = (grad_w_h + self.l2*self.w_h)
delta_b_h = grad_b_h # смещение не регуляризируется
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h

delta_w_out = (grad_w_out + self.l2*self.w_out)
delta_b_out = grad_b_out # смещение не регуляризируется
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out

#####
# Оценка #
#####

# Оценка после каждой эпохи во время обучения
z_h, a_h, z_out, a_out = self._forward(X_train)

cost = self._compute_cost(y_enc=y_train_enc,
                          output=a_out)

y_train_pred = self.predict(X_train)
y_valid_pred = self.predict(X_valid)

```



```

train_acc = ((np.sum(y_train ==
                    y_train_pred)).astype(np.float) /
             X_train.shape[0])
valid_acc = ((np.sum(y_valid ==
                    y_valid_pred)).astype(np.float) /
             X_valid.shape[0])

sys.stderr.write('\r%0*d/%d | Издержки: %.2f '
                '| Правильность при обучении/
при проверке: %.2f%%/%.2f%% '
                %
                (epoch_strlen, i+1, self.epochs,
                 cost,
                 train_acc*100, valid_acc*100))

sys.stderr.flush()

self.eval_['cost'].append(cost)
self.eval_['train_acc'].append(train_acc)
self.eval_['valid_acc'].append(valid_acc)

return self

```

После подготовки кода мы инициализируем новый многослойный персептрон 784-100-10 — нейронную сеть, которая имеет 784 входных элемента (`n_features`), 100 скрытых элементов (`n_hidden`) и 10 выходных элементов (`n_output`):

```

>>> nn = NeuralNetMLP(n_hidden=100,
...                    l2=0.01,
...                    epochs=200,
...                    eta=0.0005,
...                    minibatch_size=100,
...                    shuffle=True,
...                    seed=1)

```

Если вы исследовали код `NeuralNetMLP`, то вероятно уже догадались, для чего эти параметры предназначены. Ниже представлена краткая сводка.

- 12. Параметр  $\lambda$  для регуляризации L2 с целью понижения степени переобучения.
- `epochs`. Количество проходов по обучающему набору.
- `eta`. Скорость обучения  $\eta$ .
- `shuffle`. Необходимость тасования обучающего набора перед каждой эпохой, чтобы предотвратить попадание алгоритма в циклы.

- `seed`. Случайное начальное значение для тасования и инициализации весов.
- `minibatch_size`. Количество обучающих образцов в каждом мини-пакете, когда производится разделение обучающих данных в каждой эпохе для стохастического градиентного спуска. Чтобы обеспечить более быстрое обучение, градиент вычисляется для каждого мини-пакета, а не для полных обучающих данных.

Затем мы обучаем многослойный персептрон с применением 55 000 образцов из уже перетасованного обучающего набора данных MNIST, а оставшиеся 5 000 образцов используем для проверки во время обучения. Обратите внимание, что обучение нейронной сети на стандартном настольном компьютере может занять вплоть до пяти минут.

В предыдущем коде можно было заметить, что мы реализовали метод `fit` как принимающий четыре входных аргумента: обучающие образцы, обучающие метки, проверочные образцы и проверочные метки. Во время обучения нейронной сети по-настоящему полезно сравнивать правильность при обучении и правильность при проверке, что помогает судить о том, хорошо ли работает сетевая модель с имеющейся архитектурой и гиперпараметрами. Например, если мы наблюдаем низкую правильность при обучении и при проверке, то весьма вероятно наличие проблемы с обучающим набором или же настройки гиперпараметров не являются идеальными. Относительно большой разрыв между правильностью при обучении и правильностью при проверке указывает на то, что модель, по всей видимости, переобучается обучающим набором данных, поэтому желательно уменьшить количество параметров в модели или увеличить силу регуляризации. Если правильность при обучении и правильность при проверке высоки, тогда модель, скорее всего, хорошо обобщается на новые данные, скажем, на испытательный набор, который мы используем для оценки финальной модели.

В общем случае обучение (глубоких) нейронных сетей является относительно затратным в сравнении с другими моделями, которые мы обсуждали до сих пор. Таким образом, в определенных обстоятельствах мы хотели бы остановить обучение и начать заново с другими установками гиперпараметров. В качестве альтернативы, если мы обнаруживаем, что сеть имеет все большую и большую тенденцию к переобучению обучающими данными (заметную по увеличению разрыва между ее эффективностью на обучающем

и проверочном наборах), то также может иметь смысл остановить обучение раньше.

Для запуска обучения мы выполняем следующий код:

```
>>> nn.fit(X_train=X_train[:55000],
...        y_train=y_train[:55000],
...        X_valid=X_train[55000:],
...        y_valid=y_train[55000:])
200/200 | Издержки: 5065.78 | Правильность при обучении/при проверке:
99.28%/97.98%
```

В нашей реализации `NeuralNetMLP` мы также определили атрибут `eval_`, в котором собираются показатели издержек, правильности при обучении и правильности при испытании для каждой эпохи, так что мы можем визуализировать результаты с применением библиотеки `Matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(range(nn.epochs), nn.eval_['cost'])
>>> plt.ylabel('Издержки')
>>> plt.xlabel('Эпохи')
>>> plt.show()
```

В итоге мы получаем график издержек за 200 эпох, показанный на рис. 12.7.

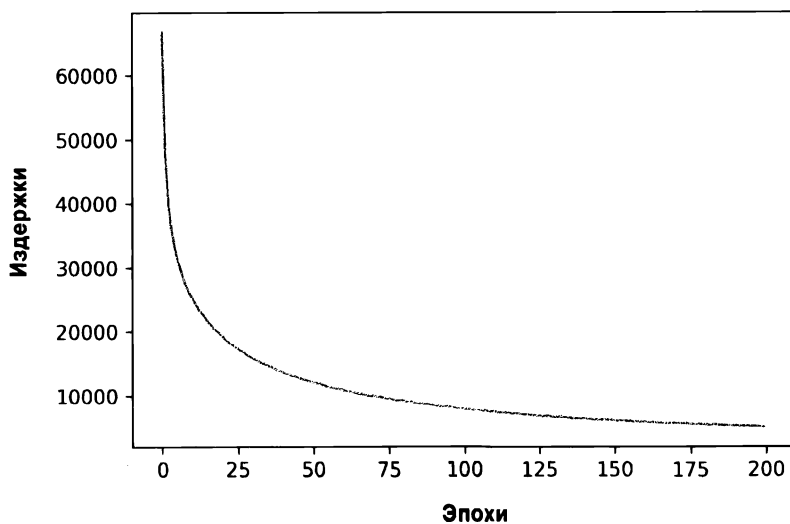


Рис. 12.7. График издержек за 200 эпох

На графике видно, что издержки существенно уменьшаются в течение первых 100 эпох и, кажется, медленно сходятся в последних 100 эпохах. Тем не менее, небольшой наклон между эпохами 175 и 200 указывает на то, что издержки продолжают снижаться, если обучение продолжится в дополнительных эпохах.

Давайте взглянем на правильность при обучении и при проверке:

```
>>> plt.plot(range(nn.epochs), nn.eval_['train_acc'],
...           label='Обучение')
>>> plt.plot(range(nn.epochs), nn.eval_['valid_acc'],
...           label='Проверка', linestyle='--')
>>> plt.ylabel('Правильность')
>>> plt.xlabel('Эпохи')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

В результате выполнения предыдущего кода мы получаем график правильности в течение 200 эпох обучения (рис. 12.8).

На графике обнаруживается, что разрыв между правильностью при обучении и правильностью при проверке растет с увеличением числа эпох, в течение которых мы обучаем сеть. Примерно на 50-й эпохе значения правильности при обучении и правильности при проверке становятся равными, а затем сеть начинает переобучаться обучающими данными.

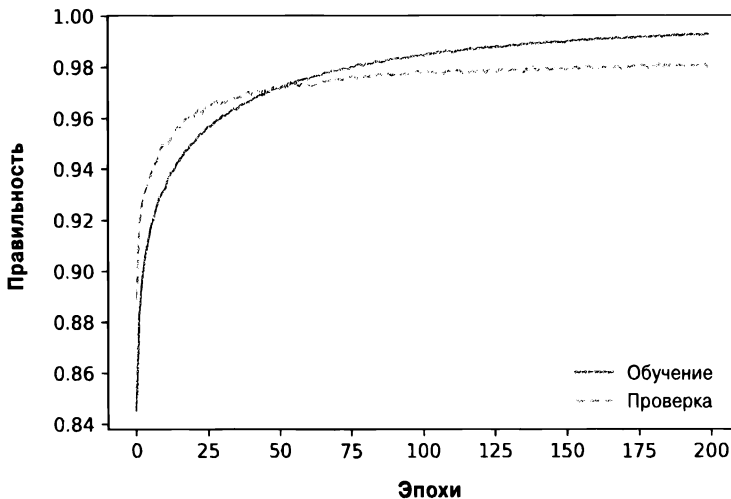


Рис. 12.8. График правильности за 200 эпох

Обратите внимание, что этот пример был выбран умышленно, чтобы проиллюстрировать эффект от переобучения и продемонстрировать, почему во время обучения полезно сравнивать значения правильности при проверке и правильности при обучении. Один из способов снижения эффекта переобучения предусматривает увеличение силы регуляризации — скажем, путем установки  $\lambda_2=0.1$ . Другой прием решения проблемы переобучения в нейронных сетях — *отключение (dropout)* — будет раскрыт в главе 15.

В заключение давайте оценим эффективность обобщения модели, подчитав правильность прогнозирования на испытательном наборе данных:

```
>>> y_test_pred = nn.predict(X_test)
>>> acc = (np.sum(y_test == y_test_pred)
...         .astype(np.float) / X_test.shape[0])
>>> print('Правильность при испытании: %.2f%%' % (acc * 100))
Правильность при испытании: 97.54%
```

Несмотря на незначительное переобучение обучающими данными, наша относительно простая нейронная сеть с одним скрытым слоем достигает довольно хорошей эффективности на испытательном наборе данных, которая сравнима с правильностью на проверочном наборе (97.98%).

Для дальнейшей настройки модели мы могли бы изменять количество скрытых элементов, значения параметров регуляризации и скорость обучения либо использовать другие трюки, разработанные за многие годы, но их рассмотрение выходит за рамки книги. В главе 15 вы узнаете о другой архитектуре нейронных сетей, которая известна своей высокой эффективностью на наборах данных с изображениями. Там же будут представлены дополнительные приемы улучшения эффективности, такие как адаптивные скорости обучения, более развитые алгоритмы оптимизации на основе стохастического градиентного спуска, пакетная нормализация и отключение.

Ниже перечислены другие часто применяемые приемы, которые выходят за рамки тематики последующих глав.

- Добавление обходящих связей, которые представляют собой основной вклад остаточных нейронных сетей (“Deep residual learning for image recognition”) (Глубокое остаточное обучение для распознавания изображений), К. Хе, Х. Чжан, Ш. Рен, Ц. Сунь (2016 г.), материалы конференции IEEE по компьютерному зрению и распознаванию образов, с. 770–778).

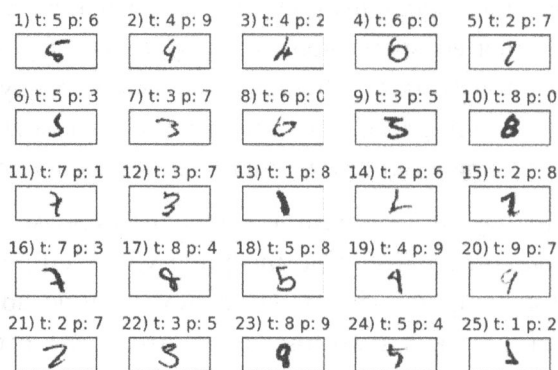
- Использование планировщиков скоростей обучения, которые изменяют скорость обучения во время процесса обучения (“Cyclical learning rates for training neural networks” (Циклические скорости обучения для обучения нейронных сетей), Л. Смит (2017 г.), материалы конференции IEEE (зима 2017 г.) по приложениям компьютерного зрения, с. 464–472).
- Присоединение функций потерь к начальным слоям в сетях, как делалось в популярной архитектуре Inception-v3 (“Rethinking the Inception architecture for computer vision” (Переосмысление архитектуры с модулем начала для компьютерного зрения), К. Сегеди, В. Ванхаук, С. Иоффе, Д. Шленс, З. Война (2016 г.), материалы конференции IEEE по компьютерному зрению и распознаванию образов, с. 2818–2826).

Наконец, давайте посмотрим, с какими изображениями сталкивался наш многослойный персептрон:

```
>>> miscl_img = X_test[y_test != y_test_pred][:25]
>>> correct_lab = y_test[y_test != y_test_pred][:25]
>>> miscl_lab = y_test_pred[y_test != y_test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True,)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = miscl_img[i].reshape(28, 28)
...     ax[i].imshow(img,
...                  cmap='Greys',
...                  interpolation='nearest')
...     ax[i].set_title('%d t: %d p: %d'
...                     % (i+1, correct_lab[i], miscl_lab[i]))
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Должна появиться матрица размером  $5 \times 5$  с графиками (рис. 12.9), где первое число в надписях указывает индекс графика, второе число представляет настоящую метку класса (t), а третье число обозначает спрогнозированную метку класса (p).



*Рис. 12.9. Изображения, обработанные многослойным персептроном*

На рис. 12.9 можно заметить, что некоторые изображения нелегко корректно классифицировать даже людям. Например, цифра 6 на графике 8 действительно похожа на небрежно написанную цифру 0, а цифра 8 на графике 23 могла бы быть цифрой 9 из-за узкой нижней части в сочетании с жирной линией.

## Обучение искусственной нейронной сети

Теперь, когда вы увидели нейронную сеть в действии и получили базовое представление о том, как она работает, просмотрев код, давайте немного углубимся в ряд концепций вроде логистической функции издержек и алгоритма обратного распространения, реализованного для узнавания весов.

### Вычисление логистической функции издержек

Логистическая функция издержек, реализованная нами в виде метода `_compute_cost`, в действительности очень проста, т.к. это та же самая функция издержек, которую мы описывали в разделе, посвященном логистической регрессии, главы 3:

$$J(\mathbf{w}) = -\sum_{i=1}^n y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]})$$

Здесь  $a^{[i]}$  — сигмоидальная активация  $i$ -того образца в наборе данных, которая вычисляется на шаге прямого распространения:

$$a^{[i]} = \phi(z^{[i]})$$

И снова в приведенном контексте надстрочный индекс  $^{[i]}$  является индексом для обучающих образцов, а не слоев.

Теперь добавим член регуляризации, который позволит снизить степень переобучения. Как было показано в предшествующих главах, член регуляризации L2 определяется следующим образом (вспомните, что мы не регуляризируем элементы смещения):

$$L2 = \lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^m w_j^2$$

Добавив к нашей логистической функции издержек член регуляризации L2, мы получим такое уравнение:

$$J(\mathbf{w}) = - \left[ \sum_{i=1}^n y^{[i]} \log(a^{[i]}) + (1 - y^{[i]}) \log(1 - a^{[i]}) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$$

Ранее мы реализовали многослойный персептрон для многоклассовой классификации, возвращающий выходной вектор из  $t$  элементов, который необходимо сравнить с  $(t \times 1)$ -мерным целевым вектором, представленным в унитарном коде. Если мы применяем этот многослойный персептрон для прогнозирования метки класса входного изображения с меткой 2, то вот как может выглядеть активация третьего слоя и цели:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Таким образом, нам нужно обобщить логистическую функцию издержек на все  $t$  элементов активации в сети.

Функция издержек (без члена регуляризации) принимает следующий вид:

$$J(\mathbf{W}) = - \sum_{i=1}^n \sum_{j=1}^t y_j^{[i]} \log(a_j^{[i]}) + (1 - y_j^{[i]}) \log(1 - a_j^{[i]})$$

Как и ранее, надстрочный индекс  $^{[i]}$  здесь относится к конкретному образцу в обучающем наборе данных.



Показанный ниже обобщенный член регуляризации поначалу может показаться несколько сложным, но мы лишь вычисляем сумму всех весов слоя  $l$  (без члена смещения) и добавляем ее к первому столбцу:

$$J(\mathbf{W}) = - \left[ \sum_{i=1}^n \sum_{j=1}^t y_j^{[i]} \log(a_j^{[i]}) + (1 - y_j^{[i]}) \log(1 - a_j^{[i]}) \right] + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{u_l} \sum_{j=1}^{u_{l+1}} (w_{j,i}^{(l)})^2$$

Здесь  $u_l$  — количество элементов в заданном слое  $l$ , а следующее выражение представляет член штрафа:

$$\frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{u_l} \sum_{j=1}^{u_{l+1}} (w_{j,i}^{(l)})^2$$

Вспомните, что наша цель заключается в сведении к минимуму функции издержек  $J(\mathbf{W})$ ; соответственно, мы должны вычислить частную производную параметров  $\mathbf{W}$  по каждому весу для всех слоев в сети:

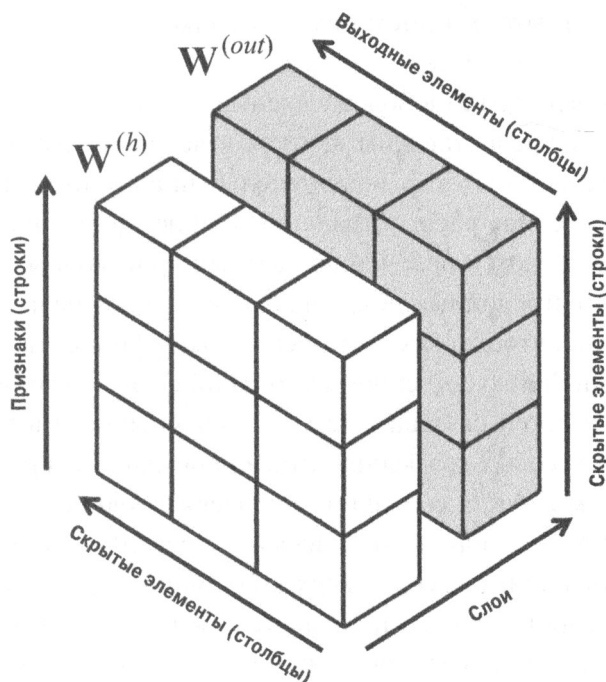
$$\frac{\partial}{\partial w_{j,i}^{(l)}} J(\mathbf{W})$$

В следующем разделе речь пойдет об алгоритме обратного распространения, который дает возможность вычислить такие частные производные для минимизации функции издержек.

Обратите внимание, что  $\mathbf{W}$  состоит из множества матриц. В многослойном персептроне с одним скрытым элементом мы имеем матрицу весов  $\mathbf{W}^{(h)}$ , связывающую вход со скрытым слоем, и матрицу весов  $\mathbf{W}^{(out)}$ , которая связывает скрытый слой с выходным слоем. На рис. 12.10 представлена поясняющая визуализация трехмерного тензора  $\mathbf{W}$ .

На приведенной упрощенной визуализации может показаться, что  $\mathbf{W}^{(h)}$  и  $\mathbf{W}^{(out)}$  имеют то же самое количество строк и столбцов, но обычно это не так, если только мы не инициализируем многослойный персептрон с одинаковым числом скрытых элементов, выходных элементов и входных признаков.

Если приведенное выше утверждение сбивает с толку, тогда дождитесь следующего раздела, где мы более подробно обсудим размерность  $\mathbf{W}^{(h)}$  и  $\mathbf{W}^{(out)}$  в контексте алгоритма обратного распространения.

Рис. 12.10. Трехмерный тензор  $W$ 

Также настоятельно рекомендуем снова просмотреть код `NeuralNetMLP`, который снабжен полезными комментариями относительно размерности различных матриц и векторных преобразований. Аннотированный код доступен для загрузки в хранилище GitHub по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition>.

## Выработка общего понимания обратного распространения

Несмотря на то что обратное распространение было заново открыто и популяризовано более 30 лет назад (“Learning Internal Representations by Error Propagation” (Изучение внутренних представлений путем распространения ошибки), Д. Румельхарт, Г. Хинтон, Р. Уильямс, *Nature*, 323: 6088, с. 533–536 (1986 г.)), оно по-прежнему остается одним из самых широко применяемых алгоритмов для очень рационального обучения искусственных нейронных сетей. Если вас интересуют дополнительные ссылки, касающиеся истории обратного распространения, то Юрген Шмидхубер написал хорошую обзорную статью “Who Invented Backpropagation?” (Кто изобрел обратное

распространение?), которая доступна по ссылке <http://people.idsia.ch/~juergen/who-invented-backpropagation.html>.

Прежде чем погружаться в более сложные математические детали, в данном разделе будут предоставлены краткая и ясная сводка и более крупная картина того, как работает этот восхитительный алгоритм. По существу мы можем считать обратное распространение крайне эффективным в вычислительном плане подходом к получению частных производных сложной функции издержек в многослойных нейронных сетях. Наша цель — использовать такие производные, чтобы выяснить весовые коэффициенты для параметризации многослойной искусственной нейронной сети. Сложность параметризации нейронных сетей связана с тем, что мы обычно имеем дело с очень большим числом весовых коэффициентов в пространстве признаков высокой размерности. По контрасту с функциями издержек однослойных нейронных сетей, таких как Adaline или логистическая регрессия, которые рассматривались в предшествующих главах, поверхность ошибок функции издержек для многослойной нейронной сети не является выпуклой или гладкой относительно параметров. На поверхности издержек высокой размерности существует много изгибов (локальных минимумов), которые мы должны преодолеть, чтобы отыскать глобальный минимум функции издержек.

Вы можете вспомнить цепное правило математики, т.е. подход к вычислению производной сложной вложенной функции вроде  $f(g(x))$ :

$$\frac{d}{dx} [f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Подобным образом мы можем применить цепное правило для произвольно длинной композиции функций. Например, пусть у нас есть пять разных функций,  $f(x)$ ,  $g(x)$ ,  $h(x)$ ,  $u(x)$  и  $v(x)$ , а  $F$  — композиция функций:  $F(x) = f(g(h(u(v(x)))))$ . Используя цепное правило, мы можем вычислить производную композиции функций следующим образом:

$$\frac{dF}{dx} = \frac{d}{dx} F(x) = \frac{d}{dx} f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

В контексте символьных вычислений был разработан набор приемов для рационального решения таких задач, который также называется *автоматическим дифференцированием*. Если вы хотите узнать больше об автоматическом дифференцировании в приложениях МО, тогда почитайте статью А.Г. Байдина и Б. Перлмуттера “Automatic Differentiation of Algorithms

for Machine Learning” (Автоматическое дифференцирование алгоритмов для машинного обучения), препринт arXiv:1404.7456 (2014 г.), которая свободно доступна по ссылке <http://arxiv.org/pdf/1404.7456.pdf>.

Автоматическое дифференцирование поддерживает два режима — прямой и обратный; обратное распространение является просто особым случаем автоматического дифференцирования в обратном режиме. Ключевой момент в том, что применение цепного правила в прямом режиме может оказаться довольно затратным, т.к. нам пришлось бы перемножать крупные матрицы для каждого слоя (якобианы), которые умножить в итоге на вектор, чтобы получить вывод.

Тонкость обратного режима связана с тем, что мы двигаемся справа налево: умножаем матрицу на вектор, получая в результате вектор, затем умножаем его на следующую матрицу и т.д. Вычислительные затраты при умножении матрицы на вектор гораздо ниже, чем при умножении матрицы на матрицу, что и объясняет высочайшую популярность алгоритма обратного распространения в области обучения нейронных сетей.



На  
заметку!

### Памятка о дифференциальном исчислении

Для полноценного понимания обратного распространения необходимо усвоить ряд концепций из дифференциального исчисления, объяснение которых выходит за рамки этой книги. Однако по ссылке [https://sebastianraschka.com/pdf/books/dlb/appendix\\_d\\_calculus.pdf](https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus.pdf) доступно приложение (на английском языке) с обзором наиболее фундаментальных концепций, который вы можете считать полезным для таких целей. В нем обсуждаются производные функций, частные производные, градиенты и якобианы. Если вы не знакомы с дифференциальным исчислением или хотите освежить свою память, тогда считайте чтение обзорного приложения дополнительным поддерживающим ресурсом перед переходом к изучению материала в следующем разделе.

## Обучение нейронных сетей с помощью обратного распространения

В этом разделе мы пройдемся по математическим выкладкам обратного распространения, чтобы понять, как можно очень эффективно выяснить веса в нейронной сети. В зависимости от того, насколько комфортно вы себя чувствуете с математическими выкладками, приводимые ниже уравнения на первых порах могут казаться относительно сложными или нет.

В предыдущем разделе мы показали, как вычислять издержки в виде разности между активацией последнего слоя и целевыми метками классов. Теперь мы посмотрим, каким образом алгоритм обратного распространения работает для обновления весов в модели MLP с математической точки зрения, которое было реализовано в разделе кода # Обратное распространение внутри метода `fit`. Как упоминалось в начале главы, сначала мы должны применить прямое распространение, чтобы получить активацию выходного слоя, что формулируется следующим образом:

$$\begin{aligned} \mathbf{Z}^{(h)} &= \mathbf{A}^{(in)} \mathbf{W}^{(h)} && \text{(общий вход скрытого слоя)} \\ \mathbf{A}^{(h)} &= \phi(\mathbf{Z}^{(h)}) && \text{(активация скрытого слоя)} \\ \mathbf{Z}^{(out)} &= \mathbf{A}^{(h)} \mathbf{W}^{(out)} && \text{(общий вход выходного слоя)} \\ \mathbf{A}^{(out)} &= \phi(\mathbf{Z}^{(out)}) && \text{(активация выходного слоя)} \end{aligned}$$

Выражаясь кратко, мы всего лишь распространяем в прямом направлении входные признаки через связи в сети, как иллюстрируется на рис. 12.11.

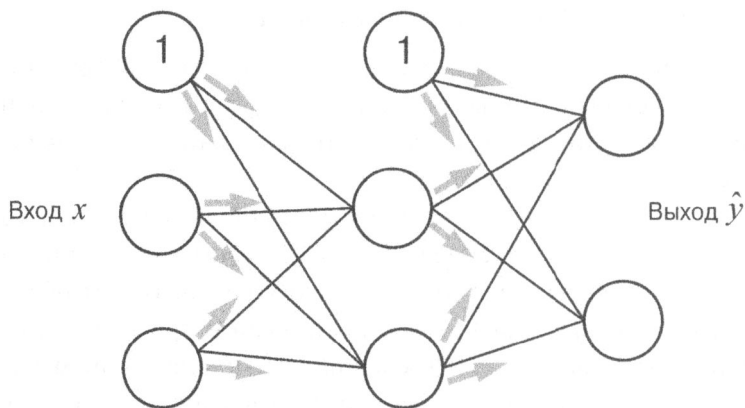


Рис. 12.11. Прямое распространение

При обратном распространении мы распространяем ошибку справа налево. Мы начинаем с вычисления вектора ошибок выходного слоя:

$$\delta^{(out)} = a^{(out)} - y$$

В уравнении  $y$  — вектор настоящих меток классов (соответствующая переменная в коде `NeuralNetMLP` называется `delta_out`).

Далее мы вычисляем член ошибки скрытого слоя:

$$\delta^{(h)} = \delta^{(out)} \left( \mathbf{W}^{(out)} \right)^T \odot \frac{\partial \phi \left( z^{(h)} \right)}{\partial z^{(h)}}$$

Здесь  $\frac{\partial \phi \left( z^{(h)} \right)}{\partial z^{(h)}}$  — просто производная сигмоидальной функции активации, которую мы вычисляем как `sigmoid_derivative_h = a_h * (1. - a_h)` в методе `fit` реализации `NeuralNetMLP`:

$$\frac{\partial \phi(z)}{\partial z} = \left( a^{(h)} \odot (1 - a^{(h)}) \right)$$

Обратите внимание, что символ  $\odot$  в данном контексте означает поэлементное умножение.



На заметку!

### Производная функции активации

Хотя понимать следующие уравнения вовсе не обязательно, вам может быть любопытно, как была получена производная функции активации; ниже представлено пошаговое выведение:

$$\begin{aligned} \phi'(z) &= \frac{\partial}{\partial z} \left( \frac{1}{1 + e^{-z}} \right) = \\ &= \frac{e^{-z}}{(1 + e^{-z})^2} = \\ &= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \left( \frac{1}{1 + e^{-z}} \right)^2 = \\ &= \frac{1}{(1 + e^{-z})} - \left( \frac{1}{1 + e^{-z}} \right)^2 = \\ &= \phi(z) - (\phi(z))^2 = \\ &= \phi(z)(1 - \phi(z)) = \\ &= a(1 - a) \end{aligned}$$

Затем мы вычисляем член матрицу ошибок  $\delta^{(h)}$  скрытого слоя ( $\delta_{\text{delta\_h}}$ ):

$$\delta^{(h)} = \delta^{(out)} \left( \mathbf{W}^{(out)} \right)^T \odot \left( a^{(h)} \odot (1 - a^{(h)}) \right)$$

Чтобы лучше понять, как мы вычисляем член  $\delta^{(h)}$ , давайте разберем его более подробно. В предыдущем уравнении мы используем транспонирование  $\left( \mathbf{W}^{(out)} \right)^T$   $(h \times t)$ -мерной матрицы  $\mathbf{W}^{(out)}$ . Здесь  $t$  — количество выходных меток классов, а  $h$  — число скрытых элементов. Произведение  $(n \times t)$ -мерной матрицы  $\delta^{(out)}$  на  $(t \times h)$ -мерную матрицу  $\left( \mathbf{W}^{(out)} \right)^T$  дает в результате  $(n \times h)$ -мерную матрицу, которую мы поэлементно умножаем на производную сигмоидальной функции той же размерности для получения  $(n \times h)$ -мерной матрицы  $\delta^{(h)}$ .

В конце концов, после получения членов  $\delta$  мы можем записать производную функции издержек следующим образом:

$$\frac{\partial}{\partial w_{i,j}^{(out)}} J(\mathbf{W}) = a_j^{(h)} \delta_i^{(out)}$$

$$\frac{\partial}{\partial w_{i,j}^{(h)}} J(\mathbf{W}) = a_j^{(in)} \delta_i^{(h)}$$

Далее нам необходимо накапливать частные производные всех узлов в каждом слое и ошибки узлов в следующем слое. Тем не менее, вспомните, что мы должны вычислять  $\Delta_{i,j}^{(l)}$  для каждого образца в обучающем наборе данных. Таким образом, легче реализовать его в виде векторизованной версии, похожей на то, как было сделано в реализации NeuralNetMLP:

$$\Delta^{(h)} = \left( \mathbf{A}^{(in)} \right)^T \delta^{(h)}$$

$$\Delta^{(out)} = \left( \mathbf{A}^{(h)} \right)^T \delta^{(out)}$$

После накопления частных производных мы можем добавить член регуляризации:

$$\Delta^{(l)} := \Delta^{(l)} + \lambda^{(l)} \mathbf{W}^{(l)}$$

(Имейте в виду, что элементы смещения обычно не регуляризуются.)

Предшествующие два математических уравнения соответствуют переменным  $\delta_{w\_h}$ ,  $\delta_{b\_h}$ ,  $\delta_{w\_out}$  и  $\delta_{b\_out}$  в коде NeuralNetMLP.

Наконец, после расчета градиентов мы можем обновить веса за счет выполнения шага в направлении, противоположном градиенту, для каждого слоя  $l$ :

$$\mathbf{W}^{(l)} := \mathbf{W}^{(l)} - \eta \Delta^{(l)}$$

Это реализуется следующим образом:

```
self.w_h -= self.eta * delta_w_h
self.b_h -= self.eta * delta_b_h
self.w_out -= self.eta * delta_w_out
self.b_out -= self.eta * delta_b_out
```

Чтобы собрать все вместе, на рис. 12.12 показана графическая иллюстрация обратного распространения.

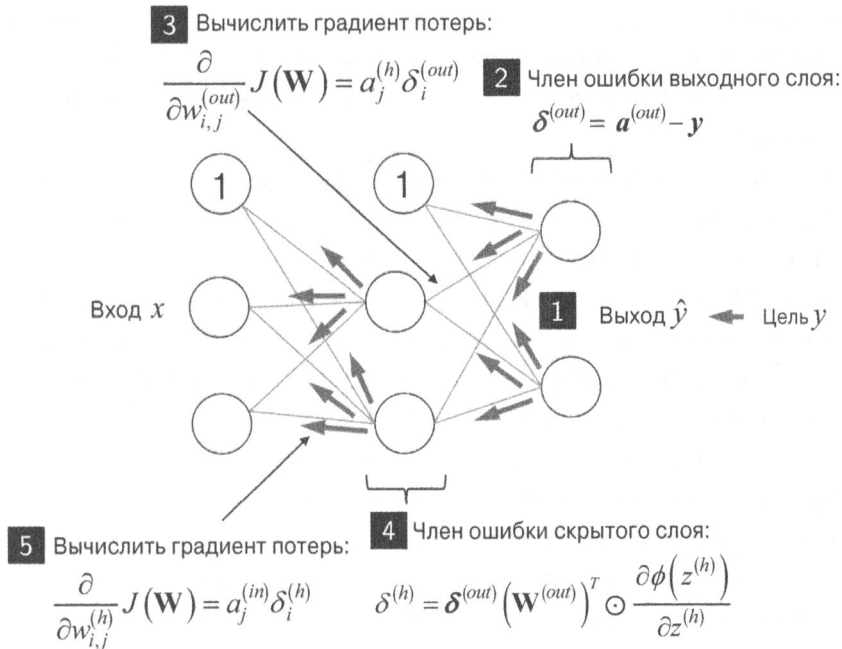


Рис. 12.12. Реализация обратного распространения



## 0 сходимости в нейронных сетях

Вас может интересовать, почему мы не применяли обычный градиентный спуск, а взамен использовали мини-пакетное обучение нейронной сети, предназначенной для классификации рукописных цифр. Вспомним обсуждение стохастического градиентного спуска, который применялся для реализации динамического обучения. При динамическом обучении для обновления весов мы вычисляем градиент на основе одного обучающего образца ( $k = 1$ ) за раз. Несмотря на то что подход стохастический, он часто приводит к очень точным решениям с гораздо более быстрой сходимостью, чем обычный градиентный спуск. Мини-пакетное обучение — это специальная форма стохастического градиентного спуска, где мы рассчитываем градиент, основываясь на поднаборе  $k$  из  $n$  обучающих образцов при  $1 < k < n$ . Преимущество мини-пакетного обучения перед динамическим обучением заключается в том, что мы можем задействовать наши векторизованные реализации для улучшения вычислительной продуктивности. Однако мы можем обновлять веса намного быстрее, чем при обычном градиентном спуске. Мини-пакетное обучение можно интуитивно представлять как прогнозирование явки избирателей на президентских выборах на базе опроса только репрезентативного поднабора населения, а не всего населения (что было бы эквивалентным проведению фактических выборов).

Многослойные нейронные сети гораздо труднее обучать, нежели более простые алгоритмы, такие как Adaline, логистическая регрессия или методы опорных векторов. В многослойных нейронных сетях мы, как правило, имеем сотни, тысячи, а то и миллиарды весов, которые необходимо оптимизировать. К сожалению, выходная функция характеризуется шероховатой поверхностью и потому алгоритм оптимизации может легко застрять в локальных минимумах, как показано на рис. 12.13.

Обратите внимание, что это представление крайне упрощено, поскольку наша нейронная сеть имеет много измерений; в итоге видимое для человеческого глаза представление действительной поверхности издержек становится невозможным. Здесь мы можем показать лишь поверхность издержек для одиночного веса по оси  $x$ . Тем не менее, главная мысль заключается в том, что мы не хотим, чтобы алгоритм застревал в локальных минимумах. За счет увеличения скорости обучения будет легче избегать таких локальных минимумов. С другой стороны, мы также увеличиваем шансы не попасть в

глобальный оптимум, если скорость обучения окажется слишком высокой. Поскольку мы инициализируем веса случайным образом, то начинаем с решения задачи оптимизации, которое обычно безнадежно неверно.

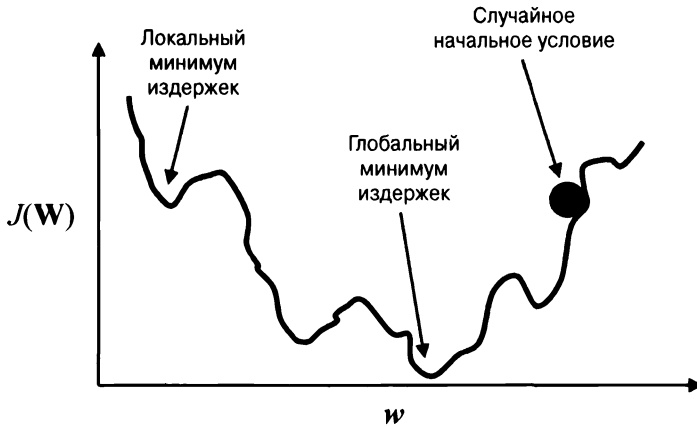


Рис. 12.13. Выходная функция с шероховатой поверхностью

## Несколько слов о реализации нейронных сетей

Вас может интересовать, почему мы привели столько теоретических выкладок лишь для того, чтобы реализовать простую многослойную нейронную сеть, способную классифицировать рукописные цифры, а не воспользовались взамен какой-то библиотекой Python для МО с открытым кодом. Надо сказать, что в последующих главах мы представим более сложные модели на основе нейронных сетей, которые будем обучать с применением библиотеки с открытым кодом TensorFlow (<https://www.tensorflow.org>).

Хотя рассмотренная в этой главе реализация с нуля поначалу кажется немного утомительной, она была хорошим упражнением, позволяющим лучше понять основы обратного распространения и обучения нейронной сети, а владение основами алгоритмов критически важно для надлежащей и успешной эксплуатации приемов МО.

Теперь, когда известно, каким образом работают нейронные сети прямого распространения, мы готовы к исследованию более сложных глубоких нейронных сетей с использованием библиотеки TensorFlow, которая позволяет конструировать нейронные сети более эффективно, что будет видно в главе 13.

После выхода в ноябре 2015 года библиотека TensorFlow обрела высокую популярность у исследователей в области МО, которые применяли TensorFlow для построения глубоких нейронных сетей из-за ее способности оптимизировать математические выражения в вычислениях с многомерными массивами благодаря использованию *графических процессоров*. Наряду с тем, что TensorFlow может считаться низкоуровневой библиотекой для глубокого обучения, были разработаны упрощенные API-интерфейсы вроде Keras, которые делают построение распространенных моделей ГО еще более удобным процессом, как будет показано в главе 13.

## Резюме

В настоящей главе были представлены базовые концепции, лежащие в основе многослойных искусственных нейронных сетей, которые в текущий момент являются самой горячей темой исследований в области МО. Мы начали путешествие в главе 2 с рассмотрения простых структур однослойных нейронных сетей и теперь связали множество нейронов в мощную нейросетевую архитектуру, чтобы решать сложные задачи, подобные распознаванию рукописных цифр. Мы прояснили популярный алгоритм обратного распространения, который выступает в качестве одного из строительных блоков во многих моделях на основе нейронных сетей, используемых в ГО. После освоения алгоритма обратного распространения мы хорошо подготовлены к анализу более сложных архитектур глубоких нейронных сетей. В оставшихся главах мы раскроем TensorFlow — библиотеку с открытым кодом, которая приводит в движение ГО, позволяя реализовывать и обучать многослойные нейронные сети более эффективно.

# РАСПАРАЛЛЕЛИВАНИЕ ПРОЦЕССА ОБУЧЕНИЯ НЕЙРОННЫХ СЕТЕЙ С ПОМОЩЬЮ TENSORFLOW

**В** настоящей главе мы перейдем от математических основ МО и ГО к рассмотрению TensorFlow — одной из наиболее популярных библиотек для ГО, доступных в текущий момент. Она делает возможной гораздо более эффективную реализацию нейронных сетей, чем любые предшествующие реализации NumPy. В этой главе мы приступим к использованию библиотеки TensorFlow и посмотрим, как она приводит к значительному выигрышу в производительности обучения.

Глава начинается еще один этап нашего путешествия в область МО и ГО; в ней будут раскрыты следующие темы:

- причины улучшения производительности обучения из-за применения библиотеки TensorFlow;
- работа с API-интерфейсом Dataset библиотеки TensorFlow (`tf.data`) для построения входных конвейеров и рационального обучения моделей;
- работа с TensorFlow для написания оптимизированного кода МО;
- использование высокоуровневых API-интерфейсов TensorFlow для построения многослойной нейронной сети;

- выбор функций активации для искусственных нейронных сетей;
- введение в Keras (`tf.keras`) — высокоуровневую оболочку вокруг TensorFlow, которую можно применять для удобной реализации пространственных архитектур ГО.

## TensorFlow и производительность обучения

Библиотека TensorFlow способна значительно ускорить выполнение задач МО. Чтобы понять, как она может это делать, давайте начнем с обсуждения проблем, связанных с производительностью, с которыми обычно приходится сталкиваться при выполнении интенсивных вычислений на имеющемся оборудовании. Затем мы выясним, что собой представляет библиотека TensorFlow с высокоуровневой точки зрения, и посмотрим, какой подход к обучению будет использоваться в данной главе.

### Проблемы, связанные с производительностью

Несомненно, производительность компьютерных процессоров в последние годы постоянно улучшалась, что позволило нам обучать более мощные и сложные системы МО и, следовательно, повысить эффективность прогнозирования наших моделей МО. В настоящее время даже недорогие настольные компьютеры располагают процессорами с множеством ядер.

В предшествующих главах мы видели, что многие функции в `scikit-learn` дают возможность разносить вычисления по нескольким обрабатывающим блокам. Однако по умолчанию Python ограничивается выполнением в одном ядре из-за *глобальной блокировки интерпретатора* (*global interpreter lock* — *GIL*). Таким образом, хотя мы действительно пользуемся преимуществами библиотеки многопроцессорной обработки Python, распределяя вычисления по множеству ядер, следует иметь в виду, что даже самые современные настольные компьютеры редко укомплектованы процессорами, имеющими больше 8 или 16 ядер.

Вспомните, что в главе 12 мы реализовали простейший многослойный персептрон с единственным скрытым слоем, состоящим из 100 элементов. Чтобы обучить модель для выполнения очень простой задачи классификации изображений, нам пришлось оптимизировать приблизительно 80 000 весовых параметров ( $[784 * 100 + 100] + [100 * 10] + 10 = 79\,510$ ). Изображения в наборе данных MNIST довольно малы (28×28 пикселей), и можно толь-

ко представить себе бурный рост количества параметров, если понадобится добавить дополнительные скрытые слои или обрабатывать изображения с большей плотностью пикселей. Такая задача быстро становится невыполнимой на одиночном процессорном блоке. В результате возникает вопрос: как более рационально решать задачи подобного рода?

Очевидным ответом будет применение графических процессоров (ГП), являющихся настоящими рабочими лошадками. Графическую плату можно считать небольшим вычислительным кластером внутри компьютера. Еще одно преимущество современных ГП в том, что они дешевле в сравнении с новейшими ЦП (рис. 13.1).

Характеристики	Intel® Core™ i9-9960X X-series Processor	NVIDIA GeForce® RTX™ 2080 Ti
Тактовая частота	3.1 ГГц	1.35 ГГц
Количество ядер	16 (32 потока)	4 352
Пропускная способность памяти	79.47 Гбайт/с	616 Гбайт/с
Количество операций с плавающей точкой	1290 Гфлопс	13 400 Гфлопс
Цена	~ \$1700.00	~ \$1100.00

**Рис. 13.1.** Сравнительные характеристики ЦП и ГП

Источниками информации в таблице, приведенной на рис. 13.1, послужили следующие ссылки (по данным на октябрь 2019 г.):

- <https://ark.intel.com/content/www/us/en/ark/products/189123/intel-core-i9-9960x-x-series-processor-22m-cache-up-to-4-50-ghz.html>
- <https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/>

За 65% цены современного ЦП мы можем заполучить ГП, который имеет в 272 раза больше ядер и способен выполнять примерно в 10 раз больше операций с плавающей точкой за секунду. Итак, что же нас удерживает от использования ГП для задач МО? Проблема в том, что писать код, ориентированный на ГП, не настолько просто, как выполнять код Python в интерпретаторе. Существуют специальные пакеты, такие как CUDA и OpenCL, которые позволяют ориентироваться на ГП. Тем не менее, написание кода в CUDA или OpenCL вряд ли можно считать самым удобным подходом к

реализации и запуску алгоритмов МО. Хорошая новость в том, что именно по этой причине была разработана библиотека TensorFlow!

## Что такое TensorFlow?

Библиотека TensorFlow представляет собой масштабируемый и мультиплатформенный программный интерфейс для реализации и выполнения алгоритмов МО, включающий удобные оболочки для ГО. Библиотека TensorFlow была разработана исследователями и инженерами группы Google Brain. Наряду с тем, что основная разработка велась группой исследователей и специалистов по программному обеспечению в Google, в процесс были также вовлечены многие участники из сообщества открытого кода. Первоначально библиотека TensorFlow строилась для внутреннего потребления в Google, но затем в ноябре 2015 года она была выпущена под либеральной лицензией ПО с открытым кодом. Многие исследователи и специалисты-практики в области МО из научного сообщества и индустрии стали применять библиотеку TensorFlow при разработке решений для ГО.

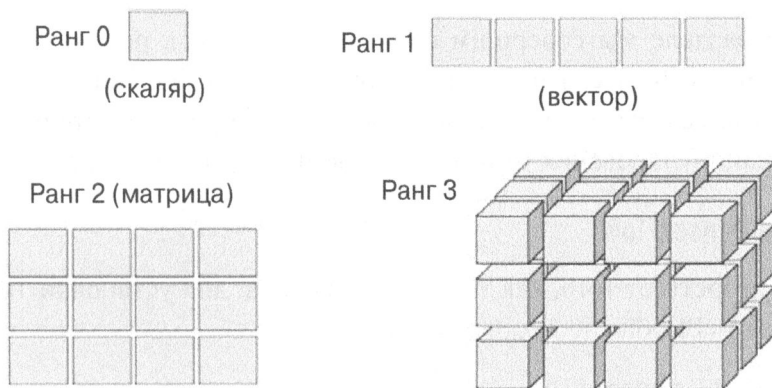
Чтобы улучшить производительность обучения моделей МО, библиотека TensorFlow делает возможным выполнение сразу как на ЦП, так и на ГП. Однако наилучших характеристик производительности можно добиться в случае применения ГП. Библиотека TensorFlow официально поддерживает ГП с возможностями CUDA. Поддержка устройств с возможностями OpenCL пока еще экспериментальна. Тем не менее, в ближайшем будущем OpenCL, вероятно, станет поддерживаться официально. В текущий момент TensorFlow поддерживает внешние интерфейсы для нескольких языков программирования.

К счастью пользователей Python в настоящее время API-интерфейс для Python библиотеки TensorFlow является самым полным, привлекая тем самым многих практикующих специалистов в области МО и ГО. Кроме того, в TensorFlow имеется официальный API-интерфейс для C++. Вдобавок были выпущены новые инструменты, основанные на TensorFlow, TensorFlow.js и TensorFlow Lite, которые направлены на выполнение и развертывание моделей МО в веб-браузере, а также мобильных и IoT-устройствах (Internet of Things — Интернет вещей). Что касается API-интерфейсов для других языков, таких как Java, Haskell, Node.js и Go, то они пока еще нестабильны, но сообщество открытого кода и разработчики TensorFlow постоянно улучшают их.

Библиотека TensorFlow построена вокруг вычислительного графа, состоящего из множества узлов. Каждый узел представляет операцию, которая может иметь ноль и более входов и выходов. Тензор создается как символический дескриптор для ссылки на вход и выход таких операций.

С математической точки зрения тензоры можно понимать как обобщение скаляров, векторов, матриц и т.д. Более конкретно скаляр можно определить как тензор ранга 0, вектор — как тензор ранга 1, матрицу — как тензор ранга 2, а матрицы, уложенные стопкой в третьем измерении, — как тензор ранга 3. Но имейте в виду, что в библиотеке TensorFlow значения хранятся в массивах NumPy и тензоры предоставляют ссылки на такие массивы.

Чтобы сделать концепцию тензора яснее, на рис. 13.2 в первой строке представлены тензоры рангов 0 и 1, а во второй — тензоры рангов 2 и 3:



*Рис. 13.2. Тензоры разных рангов*

В первоначальном выпуске вычисления TensorFlow основывались на конструировании статического ориентированного графа для представления потока данных. Поскольку использование статических вычислительных графов для многих оказалось серьезной проблемой, в недавней версии 2.0 библиотека TensorFlow подверглась крупной модернизации, в результате чего построение и обучение нейросетевых моделей значительно упростилось. Хотя библиотека TensorFlow 2.0 по-прежнему поддерживает статические вычислительные графы, теперь в ней применяются динамические вычислительные графы, что делает возможной более высокую гибкость.



## Как мы будем изучать TensorFlow

Первым делом мы раскроем программную модель TensorFlow, в частности, создание тензоров и манипулирование ими. Затем мы выясним, как загружать данные и задействовать объекты Dataset из TensorFlow, которые позволят эффективно проходить по набору данных. Вдобавок мы обсудим существующие готовые к использованию наборы данных в подмодуле `tensorflow_datasets` и научимся работать с ними.

После ознакомления с основами будет представлен API-интерфейс `tf.keras` и мы займемся построением моделей МО, их компиляцией и обучением, а также сохранением обученных моделей на диске для будущей оценки.

## Первые шаги при работе с библиотекой TensorFlow

В этом разделе мы совершим свои первые шаги в использовании низкоуровневого API-интерфейса TensorFlow. После установки TensorFlow мы покажем, как создавать тензоры в TensorFlow, и продемонстрируем разные способы манипулирования ими: изменение их формы, типа данных и т.д.

### Установка TensorFlow

В зависимости от того, как настроена система, для установки TensorFlow из каталога PyPI обычно можно применять `pip`:

```
pip install tensorflow
```

Введенная команда установит последнюю *стабильную* версию TensorFlow, которой на момент написания книги была 2.0.0. Чтобы обеспечить надлежащее выполнение предлагаемого в главе кода, мы рекомендуем использовать версию TensorFlow 2.0.0, которую можно установить, явно указывая номер версии:

```
pip install tensorflow==[желаемая-версия]
```

При желании использовать ГП (что рекомендуется) понадобится совместимая графическая плата NVIDIA наряду с установленным комплектом инструментов CUDA Toolkit и библиотекой NVIDIA cuDNN. Если ваш компьютер удовлетворяет таким требованиям, тогда вы можете установить TensorFlow с поддержкой ГП:

```
pip install tensorflow-gpu
```

Дополнительные сведения о процессе установки и настройки доступны по ссылке <https://www.tensorflow.org/install/gpu>.

Следует отметить, что библиотека TensorFlow все еще находится в условиях активной разработки; по этой причине каждые несколько месяцев выпускаются более новые версии со значительными изменениями. На время написания главы последней версией была 2.0. Проверить версию установленной библиотеки TensorFlow можно посредством такой команды:

```
python -c 'import tensorflow as tf; print(tf.__version__)'
```



На заметку!

### Устранение проблем, связанных с установкой TensorFlow

Если вы столкнулись с проблемами при выполнении процедуры установки, тогда ознакомьтесь с требованиями к системе и платформе по ссылке <https://www.tensorflow.org/install/>. Обратите внимание, что весь код, приведенный в главе, может выполняться в ЦП; применять ГП не обязательно, но желательно, если нужно задействовать все преимущества TensorFlow. Например, в то время как обучение ряда нейросетевых моделей на ЦП может занять неделю, на современном ГП для этого потребуется всего несколько часов. При наличии графической платы обратитесь на страницу документации, чтобы настроить ее надлежащим образом. Кроме того, может оказаться полезным руководство по установке TensorFlow-GPU, где объясняется, как устанавливать драйверы графических плат NVIDIA, CUDA и cuDNN в среде Ubuntu (не обязательные, но рекомендуемые условия для функционирования TensorFlow на ГП): [https://sebastianraschka.com/pdf/books/dlb/appendix\\_h\\_cloud-computing.pdf](https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf). Вдобавок, как вы увидите в главе 17, модели можно также бесплатно обучать с использованием ГП посредством Google Colab.

## Создание тензоров в TensorFlow

Давайте теперь рассмотрим несколько отличающихся способов создания тензоров, исследуем некоторые их свойства и выясним, как ими манипулировать. Прежде всего, мы можем легко создать тензор из списка или массива NumPy с применением функции `tf.convert_to_tensor`:

```
>>> import tensorflow as tf
>>> import numpy as np
>>> np.set_printoptions(precision=3)
>>> a = np.array([1, 2, 3], dtype=np.int32)
>>> b = [4, 5, 6]
>>> t_a = tf.convert_to_tensor(a)
>>> t_b = tf.convert_to_tensor(b)
>>> print(t_a)
>>> print(t_b)

tf.Tensor([1 2 3], shape=(3,), dtype=int32)
tf.Tensor([4 5 6], shape=(3,), dtype=int32)
```

Результатом выполнения показанного выше кода оказываются тензоры `t_a` и `t_b` со свойствами `shape=(3,)` и `dtype=int32`, полученными из их источника. Подобно массивам NumPy мы можем просмотреть эти свойства:

```
>>> t_ones = tf.ones((2, 3))
>>> t_ones.shape
TensorShape([2, 3])
```

Чтобы получить доступ к значениям, на которые ссылается тензор, мы можем просто вызвать метод `.numpy()` на тензоре:

```
>>> t_ones.numpy()
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

Наконец, вот как создать тензор константных значений:

```
>>> const_tensor = tf.constant([1.2, 5, np.pi],
...                             dtype=tf.float32)
>>> print(const_tensor)
tf.Tensor([1.2  5.   3.142], shape=(3,), dtype=float32)
```

## Манипулирование типом данных и формой тензора

Освоить способы манипулирования тензорами необходимо для того, чтобы делать их совместимыми с входом модели или операции. В этом разделе мы выясним, как манипулировать типами данных и формами тензоров с помощью нескольких функций TensorFlow, которые приводят, придают другую форму, транспонируют и сжимают тензоры.

Функцию `tf.cast()` можно использовать для смены типа данных тензора на желаемый тип:

```
>>> t_a_new = tf.cast(t_a, tf.int64)
>>> print(t_a_new.dtype)
<dtype: 'int64'>
```

Как будет показано в последующих главах, отдельные операции требуют, чтобы входные тензоры имели определенное количество измерений (т.е. ранг), ассоциированных с определенным числом элементов (форма). Таким образом, может возникнуть необходимость в изменении формы тензора, добавлении нового измерения или сжатии ненужного измерения. Для указанных целей библиотека TensorFlow предлагает практичные функции (или операции), такие как `tf.transpose()`, `tf.reshape()` и `tf.squeeze()`. Ниже приведено несколько примеров.

- Транспонирование тензора:

```
>>> t = tf.random.uniform(shape=(3, 5))
>>> t_tr = tf.transpose(t)
>>> print(t.shape, ' --> ', t_tr.shape)
(3, 5) --> (5, 3)
```

- Изменение формы тензора (скажем, превращение одномерного вектора в двумерный массив):

```
>>> t = tf.zeros((30,))
>>> t_reshape = tf.reshape(t, shape=(5, 6))
>>> print(t_reshape.shape)
(5, 6)
```

- Удаление излишних измерений (измерений, имеющих размер 1, которые не нужны):

```
>>> t = tf.zeros((1, 2, 1, 4, 1))
>>> t_sqz = tf.squeeze(t, axis=(2, 4))
>>> print(t.shape, ' --> ', t_sqz.shape)
(1, 2, 1, 4, 1) --> (1, 2, 4)
```

## Применение математических операций к тензорам

Применять математические операции, в частности операции линейной алгебры, необходимо при построении большинства моделей МО. В этом разделе мы раскроем ряд широко используемых операций линейной алгебры вроде поэлементного произведения, перемножения матриц и вычисления нормы тензора.

Первым делом давайте создадим два случайных тензора, один с равномерным распределением в диапазоне  $[-1, 1)$  и еще один со стандартным нормальным распределением:

```
>>> tf.random.set_seed(1)
>>> t1 = tf.random.uniform(shape=(5, 2),
...                          minval=-1.0, maxval=1.0)
>>> t2 = tf.random.normal(shape=(5, 2),
...                         mean=0.0, stddev=1.0)
```

Обратите внимание, что  $t1$  и  $t2$  имеют ту же самую форму. Теперь для вычисления поэлементного произведения  $t1$  и  $t2$  мы можем применять следующий код:

```
>>> t3 = tf.multiply(t1, t2).numpy()
>>> print(t3)
[[-0.27 -0.874]
 [-0.017 -0.175]
 [-0.296 -0.139]
 [-0.727  0.135]
 [-0.401  0.004]]
```

Чтобы рассчитать среднее, сумму и стандартное отклонение по определенной оси (или осям), мы можем использовать `tf.math.reduce_mean()`, `tf.math.reduce_sum()` и `tf.math.reduce_std()`. Например, среднее каждого столбца в  $t1$  можно вычислить так:

```
>>> t4 = tf.math.reduce_mean(t1, axis=0)
>>> print(t4)
tf.Tensor([0.09  0.207], shape=(2,), dtype=float32)
```

Матричное произведение  $t1$  и  $t2$  (т.е.  $t_1 \times t_2^T$ , где  $T$  означает транспонирование) можно вычислить с применением функции `tf.linalg.matmul()`:

```
>>> t5 = tf.linalg.matmul(t1, t2, transpose_b=True)
>>> print(t5.numpy())
[[-1.144  1.115 -0.87  -0.321  0.856]
 [ 0.248 -0.191  0.25  -0.064 -0.331]
 [-0.478  0.407 -0.436  0.022  0.527]
 [ 0.525 -0.234  0.741 -0.593 -1.194]
 [-0.099  0.26   0.125 -0.462 -0.396]]
```

С другой стороны, вычисление  $t_1 \times t_2^T$  выполняется путем транспонирования  $t1$ , что в результате дает массив  $2 \times 2$ :

```
>>> t6 = tf.linalg.matmul(t1, t2, transpose_a=True)
>>> print(t6.numpy())
[[-1.711  0.302]
 [ 0.371 -1.049]]
```

Наконец, функция `tf.norm()` удобна для расчета нормы  $L^p$  тензора. Скажем, мы можем вычислить нормы  $L^2$  тензора `t1` следующим образом:

```
>>> norm_t1 = tf.norm(t1, ord=2, axis=1).numpy()
>>> print(norm_t1)
[1.046 0.293 0.504 0.96  0.383]
```

Чтобы проверить корректность расчета нормы  $L^2$  тензора `t1` приведенным выше кодом, можно сравнить результаты посредством функции из библиотеки NumPy: `np.sqrt(np.sum(np.square(t1), axis=1))`.

## Расщепление, укладывание стопкой и объединение тензоров

В этом подразделе мы рассмотрим операции TensorFlow для расщепления одного тензора на множество тензоров и наоборот: укладывание стопкой и объединение тензоров в единственный тензор.

Предположим, что у нас есть одиночный тензор и его нужно расщепить на два или большее число тензоров. Для такого действия в TensorFlow предусмотрена удобная функция `tf.split()`, которая разделяет входной тензор на список тензоров равных размеров. Мы можем определить желательное количество расщеплений как целое число с использованием аргумента `num_or_size_splits` и расщепить тензор по заданному измерению, указанному с помощью аргумента `axis`. В таком случае общий размер входного тензора по указанному измерению должен быть кратным желательному числу расщеплений. В качестве альтернативы мы можем предоставить желательные размеры в списке. Ниже приведены примеры для обоих вариантов.

- Предоставление количества расщеплений (общий размер должен быть кратным указанному количеству):

```
>>> tf.random.set_seed(1)
>>> t = tf.random.uniform((6,))
>>> print(t.numpy())
[0.165 0.901 0.631 0.435 0.292 0.643]
>>> t_splits = tf.split(t, num_or_size_splits=3)
>>> [item.numpy() for item in t_splits]
[array([0.165, 0.901], dtype=float32),
 array([0.631, 0.435], dtype=float32),
 array([0.292, 0.643], dtype=float32)]
```

В показанном примере тензор размера 6 был разделен на список из трех тензоров, каждый из которых имеет размер 2.

- Предоставление размеров отличающихся расщеплений.

Вместо определения количества расщеплений мы также можем напрямую указать размеры выходных тензоров. Здесь мы разделяем тензор размера 5 на тензоры с размерами 3 и 2:

```
>>> tf.random.set_seed(1)
>>> t = tf.random.uniform((5,))
>>> print(t.numpy())
[0.165 0.901 0.631 0.435 0.292]

>>> t_splits = tf.split(t, num_or_size_splits=[3, 2])
>>> [item.numpy() for item in t_splits]
[array([0.165, 0.901, 0.631], dtype=float32),
 array([0.435, 0.292], dtype=float32)]
```

Иногда мы работаем с множеством тензоров и нуждаемся в их объединении или укладывании стопкой, чтобы создать единственный тензор. В таком случае окажутся полезными функции `tf.stack()` и `tf.concat()` библиотеки TensorFlow. Например, давайте создадим одномерный тензор A размера 3, содержащий единицы, и одномерный тензор B размера 2, содержащий нули, после чего объединим их в одномерный тензор C размера 5:

```
>>> A = tf.ones((3,))
>>> B = tf.zeros((2,))
>>> C = tf.concat([A, B], axis=0)
>>> print(C.numpy())
[1. 1. 1. 0. 0.]
```

Если мы создадим одномерные тензоры A и B, оба с размером 3, тогда можем уложить их стопкой, образовав двумерный тензор S:

```
>>> A = tf.ones((3,))
>>> B = tf.zeros((3,))
>>> S = tf.stack([A, B], axis=1)
>>> print(S.numpy())
[[1. 0.]
 [1. 0.]
 [1. 0.]]
```

В API-интерфейсе TensorFlow есть много операций, которые вы можете применять для построения модели, обработки данных и решения дру-

гих задач. Однако целью настоящей книги является раскрытие не абсолютно всех функций, а только самых важных из них. Полный список операций и функций ищите в документации TensorFlow по ссылке [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf).

## **Построение входных конвейеров с использованием `tf.data` — API-интерфейса `Dataset` библиотеки TensorFlow**

Обычно мы обучаем модель на основе глубокой нейронной сети постепенно, применяя итеративный алгоритм оптимизации, такой как стохастический градиентный спуск, который вы видели в предшествующих главах.

Как упоминалось в начале текущей главы, API-интерфейс Keras представляет собой оболочку вокруг TensorFlow, предназначенную для построения нейросетевых моделей. Для обучения моделей API-интерфейс Keras предлагает метод `.fit()`. В случаях, когда обучающий набор данных довольно мал и может быть загружен в виде тензора в память, модели TensorFlow (построенные с помощью API-интерфейса Keras) могут напрямую задействовать этот тензор через их метод `.fit()` для обучения. Тем не менее, в типовых сценариях использования, когда набор данных слишком большой и потому не помещается в памяти, нам понадобится загружать данные из основного устройства хранения (скажем, жесткого диска или твердотельного накопителя) порциями, т.е. пакет за пакетом (обратите внимание на употребление в этой главе термина “пакет” вместо “мини-пакет”, что позволяет оставаться ближе к терминологии TensorFlow). Кроме того, может возникнуть необходимость сконструировать конвейер обработки данных для применения к данным определенных трансформаций и шагов предварительной обработки вроде центрирования по среднему, масштабирования или добавления шума, чтобы дополнить процедуру обучения и предотвратить переобучение.

Постоянное ручное применение функций предварительной обработки может стать крайне утомительным. К счастью, библиотека TensorFlow предоставляет специальный класс для конструирования эффективных и удобных конвейеров предварительной обработки. В этом разделе мы предложим обзор различных методов для конструирования объектов `Dataset` из TensorFlow, включая трансформации наборов данных и распространенные шаги предварительной обработки.



## Создание объекта Dataset из существующих тензоров

Если данные уже существуют в форме объекта тензора, списка Python или массива NumPy, тогда мы можем легко создать набор данных с использованием функции `tf.data.Dataset.from_tensor_slices()`. Указанная функция возвращает объект класса `Dataset`, который мы можем применять для прохода по индивидуальным элементам во входном наборе данных. В качестве примера рассмотрим следующий код, который создает набор данных из списка значений:

```
>>> a = [1.2, 3.4, 7.5, 4.1, 5.0, 1.0]
>>> ds = tf.data.Dataset.from_tensor_slices(a)
>>> print(ds)
<TensorSliceDataset shapes: (), types: tf.float32>
```

Вот как проходить по записям набора данных:

```
>>> for item in ds:
...     print(item)
tf.Tensor(1.2, shape=(), dtype=float32)
tf.Tensor(3.4, shape=(), dtype=float32)
tf.Tensor(7.5, shape=(), dtype=float32)
tf.Tensor(4.1, shape=(), dtype=float32)
tf.Tensor(5.0, shape=(), dtype=float32)
tf.Tensor(1.0, shape=(), dtype=float32)
```

Если необходимо создать из этого набора данных пакеты с желательным размером 3, то мы можем поступить так:

```
>>> ds_batch = ds.batch(3)
>>> for i, elem in enumerate(ds_batch, 1):
...     print('пакет {}'.format(i), elem.numpy())
пакет 1: [1.2 3.4 7.5]
пакет 2: [4.1 5. 1. ]
```

В результате из набора данных создаются два пакета, причем первые три элемента попадают в пакет #1, а оставшиеся элементы — в пакет #2. Метод `.batch()` имеет необязательный аргумент `drop_remainder`, который будет полезен в случаях, когда количество элементов в тензоре не кратно желательному размеру пакета. Стандартным значением `drop_remainder` является `False`. Дополнительные примеры, иллюстрирующие поведение метода `.batch()`, будут приведены в разделе “Тасование, создание пакетов и повторение” далее в главе.

## Объединение двух тензоров в общий набор данных

Часто данные могут находиться в двух или большем числе тензоров. Скажем, мы могли бы иметь тензор для признаков и тензор для меток. В таких случаях нам необходимо построить набор данных, который объединит эти тензоры и позволит извлекать элементы тензоров в кортежах.

Предположим, что у нас есть два тензора, `t_x` и `t_y`. Тензор `t_x` хранит значения признаков, размером 3 каждое, а тензор `t_y` содержит метки классов. В рассматриваемом примере мы сначала создаем эти два тензора:

```
>>> tf.random.set_seed(1)
>>> t_x = tf.random.uniform([4, 3], dtype=tf.float32)
>>> t_y = tf.range(4)
```

Теперь мы хотим создать из двух тензоров общий набор данных. Обратите внимание на наличие обязательного соответствия “один к одному” между элементами тензоров:

```
>>> ds_x = tf.data.Dataset.from_tensor_slices(t_x)
>>> ds_y = tf.data.Dataset.from_tensor_slices(t_y)
>>>
>>> ds_joint = tf.data.Dataset.zip((ds_x, ds_y))
>>> for example in ds_joint:
...     print(' x:', example[0].numpy(),
...           ' y:', example[1].numpy())
x: [0.165 0.901 0.631] y: 0
x: [0.435 0.292 0.643] y: 1
x: [0.976 0.435 0.66 ] y: 2
x: [0.605 0.637 0.614] y: 3
```

Здесь мы сначала создаем два набора данных с именами `ds_x` и `ds_y`. Затем мы используем функцию `zip` для построения общего набора данных. В качестве альтернативы мы можем создать общий набор данных с применением метода `tf.data.Dataset.from_tensor_slices()`:

```
>>> ds_joint = tf.data.Dataset.from_tensor_slices((t_x, t_y))
>>> for example in ds_joint:
...     print(' x:', example[0].numpy(),
...           ' y:', example[1].numpy())
x: [0.165 0.901 0.631] y: 0
x: [0.435 0.292 0.643] y: 1
x: [0.976 0.435 0.66 ] y: 2
x: [0.605 0.637 0.614] y: 3
```

В результате мы получаем тот же самый вывод.

Следует отметить, что частым источником ошибки может быть утрата элементного соответствия между исходными признаками ( $x$ ) и метками ( $y$ ), скажем, если два набора данных тасуются по отдельности. Однако после объединения в один набор данных можно безопасно применять операции.

Давайте выясним, как применять трансформации к отдельным элементам набора данных. Для этого мы будем использовать предыдущий набор данных `ds_joint` и применять масштабирование признаков с целью приведения значений к диапазону  $[-1, 1)$ , поскольку в текущий момент значения `t_x` находятся в диапазоне  $[0, 1)$ , основанном на случайном равномерном распределении:

```
>>> ds_trans = ds_joint.map(lambda x, y: (x*2-1.0, y))
>>> for example in ds_trans:
...     print(' x:', example[0].numpy(),
...           ' y:', example[1].numpy())
x: [-0.67  0.803  0.262] y: 0
x: [-0.131 -0.416  0.285] y: 1
x: [ 0.952 -0.13  0.32 ] y: 2
x: [ 0.21  0.273  0.229] y: 3
```

Применение трансформации такого рода может требоваться в функции, определяемой пользователем. Например, если есть набор данных, созданный из списка имен файлов с изображениями на диске, тогда мы можем определить функцию для загрузки изображений по именам файлов и применять ее посредством вызова метода `.map()`. Позже в главе вы увидите пример применения множества трансформаций к набору данных.

## Тасование, создание пакетов и повторение

Как упоминалось в главе 2, для обучения нейросетевой модели с использованием оптимизации в форме стохастического градиентного спуска важно подавать обучающие данные в виде случайно перетасованных пакетов. Вы уже знаете, как создавать пакеты за счет вызова метода `.batch()` объекта набора данных. Теперь в дополнение к созданию пакетов вы научитесь тасовать наборы данных и проходить по ним. Мы продолжим работать с предыдущим набором данных `ds_joint`.

Первым делом давайте создадим перетасованную версию набора данных `ds_joint`:

```
>>> tf.random.set_seed(1)
>>> ds = ds_joint.shuffle(buffer_size=len(t_x))
>>> for example in ds:
...     print(' x:', example[0].numpy(),
...           ' y:', example[1].numpy())
x: [0.976 0.435 0.66 ] y: 2
x: [0.435 0.292 0.643] y: 1
x: [0.165 0.901 0.631] y: 0
x: [0.605 0.637 0.614] y: 3
```

Строки тасуются без потери соответствия “один к одному” между элементами в `x` и `y`. Метод `.shuffle()` требует аргумента по имени `buffer_size`, который определяет количество элементов в наборе данных, группируемых перед тасованием. Элементы в буфере выбираются случайным образом, а их место в буфере назначается следующим элементам в исходном (не перетасованном) наборе данных. Таким образом, если мы выберем небольшое значение для `buffer_size`, то не сможем обеспечить идеальное тасование набора данных.

Когда набор данных мал, выбор относительно небольшого значения `buffer_size` может отрицательно сказаться на эффективности прогнозирования нейронной сети, т.к. набор данных, возможно, окажется рандомизированным не полностью. Тем не менее, на практике обычно это не дает заметного эффекта в случае работы с относительно крупными наборами данных, которые распространены в глубоком обучении. В качестве альтернативы для обеспечения полной рандомизации в течение каждой эпохи мы можем просто выбрать размер буфера, равный количеству обучающих экземпляров, как в предшествующем коде (`buffer_size=len(t_x)`).

Вспомните, что набор данных разделяется на пакеты для обучения модели путем вызова метода `.batch()`. Давайте создадим такие пакеты из набора данных `ds_joint` и посмотрим, как выглядит один пакет:

```
>>> ds = ds_joint.batch(batch_size=3,
...                      drop_remainder=False)
>>> batch_x, batch_y = next(iter(ds))
>>> print('Пакет-x:\n', batch_x.numpy())
```

```
Пакет-х:  
[[0.165 0.901 0.631]  
 [0.435 0.292 0.643]  
 [0.976 0.435 0.66 ]]  
  
>>> print('Пакет-y: ', batch_y.numpy())  
Пакет-y: [0 1 2]
```

Вдобавок при обучении модели на протяжении множества эпох нам необходимо тасовать набор данных и проходить по нему в течение желаемого количества эпох. Вот как повторить разбитый на пакеты набор данных два раза:

```
>>> ds = ds_joint.batch(3).repeat(count=2)  
>>> for i, (batch_x, batch_y) in enumerate(ds):  
...     print(i, batch_x.shape, batch_y.numpy())  
0 (3, 3) [0 1 2]  
1 (1, 3) [3]  
2 (3, 3) [0 1 2]  
3 (1, 3) [3]
```

В результате появляются две копии каждого пакета. Если мы изменим порядок следования этих операций, т.е. выполним сначала создание пакетов и затем повторение, то результат будет другим:

```
>>> ds = ds_joint.repeat(count=2).batch(3)  
>>> for i, (batch_x, batch_y) in enumerate(ds):  
...     print(i, batch_x.shape, batch_y.numpy())  
0 (3, 3) [0 1 2]  
1 (3, 3) [3 0 1]  
2 (2, 3) [2 3]
```

Обратите внимание на отличие между пакетами. Когда мы сначала создаем пакеты и затем повторяем, то получаем четыре пакета. С другой стороны, когда повторение осуществляется первым, то создаются три пакета.

Наконец, для лучшего понимания поведения рассмотренных трех операций (создание пакетов, тасование и повторение) мы проведем ряд экспериментов с ними в различных порядках. Первым делом мы объединим операции в порядке (1) тасование, (2) создание пакетов и (3) повторение:

```
## Порядок 1: тасование -> создание пакетов -> повторение
>>> tf.random.set_seed(1)
>>> ds = ds_joint.shuffle(4).batch(2).repeat(3)
>>> for i, (batch_x, batch_y) in enumerate(ds):
...     print(i, batch_x.shape, batch_y.numpy())
0 (2, 3) [2 1]
1 (2, 3) [0 3]
2 (2, 3) [0 3]
3 (2, 3) [1 2]
4 (2, 3) [3 0]
5 (2, 3) [1 2]
```

А теперь давайте испытаем другой порядок — (2) создание пакетов, (1) тасование и (3) повторение:

```
## Порядок 2: создание пакетов -> тасование -> повторение
>>> tf.random.set_seed(1)
>>> ds = ds_joint.batch(2).shuffle(4).repeat(3)
>>> for i, (batch_x, batch_y) in enumerate(ds):
...     print(i, batch_x.shape, batch_y.numpy())
0 (2, 3) [0 1]
1 (2, 3) [2 3]
2 (2, 3) [0 1]
3 (2, 3) [2 3]
4 (2, 3) [2 3]
5 (2, 3) [0 1]
```

В то время как первый пример кода (тасование, создание пакетов, повторение), похоже, перетасовал набор данных ожидаемым образом, во втором сценарии (создание пакетов, тасование, повторение) мы видим, что элементы внутри пакета вообще не тасовались. Мы можем заметить отсутствие тасования, пристальнее взглянув на тензор, который содержит целевые значения,  $y$ . Все пакеты включают либо пару значений [ $y=0$ ,  $y=1$ ], либо оставшуюся пару значений [ $y=2$ ,  $y=3$ ]; мы не наблюдаем другие возможные перестановки: [ $y=2$ ,  $y=0$ ], [ $y=1$ ,  $y=3$ ] и т.д. Обратите внимание, что для гарантии несовпадения этих результатов вы можете принять решение сделать повторение с числом более 3, скажем, `.repeat(20)`.

Итак, сумеете ли вы предсказать, что произойдет в случае применения операции тасования после операции повторения, например, (2) создание пакетов, (3) повторение, (1) тасование? Проверьте.

**Совет**

Распространенной ошибкой является двукратный вызов метода `.batch()` для строки в наборе данных. В таком случае извлечение элементов из результирующего набора данных создаст пакет пакетов образцов. По существу каждый вызов `.batch()` на наборе данных будет увеличивать на единицу ранг извлеченных тензоров.

## Создание набора данных из файлов на локальном диске

В настоящем разделе мы построим набор данных из файлов с изображениями, которые хранятся на диске. В архиве с примерами для главы имеется подкаталог `cat_dog_images`, содержащий шесть файлов с изображениями котов и собак в формате JPEG.

С помощью этого небольшого набора данных будет показано, как в целом выглядит сама процедура построения на основе сохраненных файлов. Мы собираемся использовать два дополнительных модуля библиотеки TensorFlow: `tf.io` для чтения содержимого файла с изображением и `tf.image` для декодирования низкоуровневого содержимого и изменения размеров изображения.

**На заметку!**

### Модули `tf.io` и `tf.image`

Модули `tf.io` и `tf.image` предоставляют много дополнительных и полезных функций, рассмотрение которых не входит в цели, преследуемые книгой. Вы можете ознакомиться с этими функциями в официальной документации:

- [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/io](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/io) для `tf.io`
- [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/image](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/image) для `tf.image`

Прежде чем начать, давайте взглянем на содержимое файлов с изображениями. Мы будем генерировать список файлов с изображениями посредством библиотеки `pathlib`:

```
>>> import pathlib
>>> imgdir_path = pathlib.Path('cat_dog_images')
>>> file_list = sorted([str(path) for path in
...                     imgdir_path.glob('*.jpg')])
```

```
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg',  
'cat_dog_images/cat-02.jpg', 'cat_dog_images/cat-03.jpg',  
'cat_dog_images/dog-01.jpg', 'cat_dog_images/dog-02.jpg']
```

Далее мы визуализируем примеры изображений с применением Matplotlib:

```
>>> import matplotlib.pyplot as plt  
>>> fig = plt.figure(figsize=(10, 5))  
>>> for i, file in enumerate(file_list):  
...     img_raw = tf.io.read_file(file)  
...     img = tf.image.decode_image(img_raw)  
...     print('Форма изображения: ', img.shape)  
...     ax = fig.add_subplot(2, 3, i+1)  
...     ax.set_xticks([]); ax.set_yticks([])  
...     ax.imshow(img)  
...     ax.set_title(os.path.basename(file), size=15)  
>>> plt.tight_layout()  
>>> plt.show()  
Форма изображения: (900, 1200, 3)  
Форма изображения: (900, 1200, 3)  
Форма изображения: (900, 1200, 3)  
Форма изображения: (900, 742, 3)  
Форма изображения: (800, 1200, 3)  
Форма изображения: (800, 1200, 3)
```

Примеры изображений показаны на рис. 13.3.

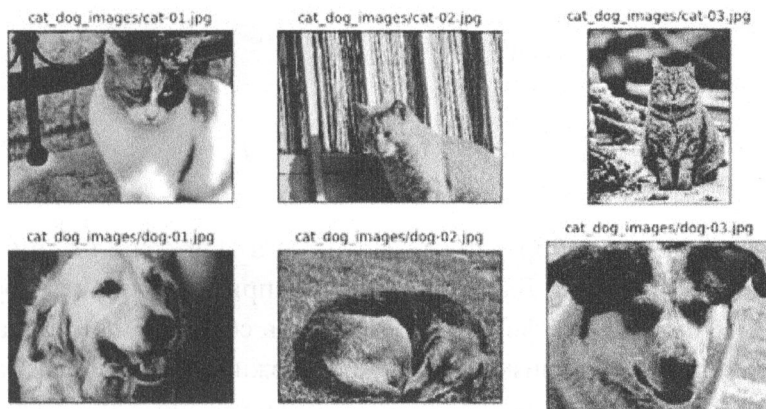


Рис. 13.3. Примеры изображений котов и собак



Благодаря одной лишь визуализации и выведенным формам изображений мы уже видим, что изображения имеют разные соотношения размеров сторон. Если вы выведете соотношения размеров сторон (или формы массивов данных) этих изображений, то заметите, что некоторые изображения обладают высотой 900 пикселей и шириной 1200 пикселей ( $900 \times 1200$ ), некоторые являются  $800 \times 1200$ , а одно —  $900 \times 742$ . Позже мы выполним предварительную обработку изображений, приведя их к согласованным размерам. Еще один момент, который необходимо учесть, заключается в том, что метки для изображений предоставляются внутри их имен файлов. Таким образом, мы извлекаем метки для изображений из списка имен файлов, назначая метку 1 изображениям собак и метку 0 изображениям котов:

```
>>> labels = [1 if 'dog' in os.path.basename(file) else 0
...           for file in file_list]
>>> print(labels)
[1, 0, 0, 0, 1, 1]
```

Теперь у нас есть два списка: список имен файлов (или путей к каждому изображению) и список их меток. В предыдущем разделе вы узнали два способа создания общего набора данных из двух тензоров. Здесь мы будем использовать второй подход:

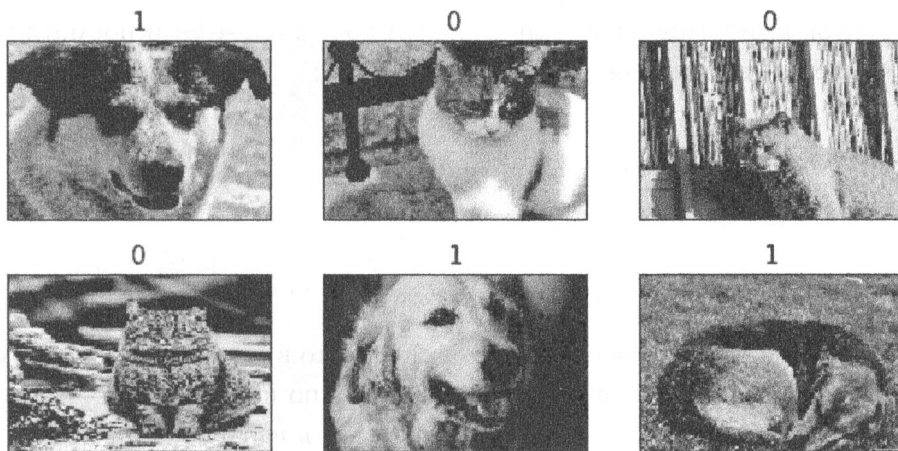
```
>>> ds_files_labels = tf.data.Dataset.from_tensor_slices(
...     (file_list, labels))
>>> for item in ds_files_labels:
...     print(item[0].numpy(), item[1].numpy())
b'cat_dog_images/dog-03.jpg' 1
b'cat_dog_images/cat-01.jpg' 0
b'cat_dog_images/cat-02.jpg' 0
b'cat_dog_images/cat-03.jpg' 0
b'cat_dog_images/dog-01.jpg' 1
b'cat_dog_images/dog-02.jpg' 1
```

Мы назвали этот набор данных `ds_files_labels`, т.к. он содержит имена файлов и метки. Далее нам нужно применить к набору данных `ds_files_labels` трансформации: загрузить содержимое изображения из его файла, декодировать низкоуровневое содержимое и изменить размеры до желаемых, скажем,  $80 \times 120$ . Ранее уже было показано, как применять лямбда-функцию с использованием метода `.map()`. Однако поскольку на этот

раз нам необходимо применить множество шагов предварительной обработки, мы взамен напишем вспомогательную функцию и воспользуемся ею при вызове метода `.map()`:

```
>>> def load_and_preprocess(path, label):
...     image = tf.io.read_file(path)
...     image = tf.image.decode_jpeg(image, channels=3)
...     image = tf.image.resize(image, [img_height, img_width])
...     image /= 255.0
...     return image, label
>>> img_width, img_height = 120, 80
>>> ds_images_labels = ds_files_labels.map(load_and_preprocess)
>>>
>>> fig = plt.figure(figsize=(10, 6))
>>> for i, example in enumerate(ds_images_labels):
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(example[0])
...     ax.set_title('{}'.format(example[1].numpy()),
...                   size=15)
>>> plt.tight_layout()
>>> plt.show()
```

Результатом будет визуализация извлеченных примеров изображений вместе с их метками, представленная на рис. 13.4.



**Рис. 13.4.** Извлеченные примеры изображений вместе с их метками

Функция `load_and_preprocess()` является оболочкой для всех четырех шагов, включая загрузку низкоуровневого содержимого, его декодирование и изменение размеров изображений. Затем функция возвращает набор данных, по которому мы можем проходить и применять к нему другие операции, описанные в предшествующих разделах.

## Извлечение доступных наборов данных из библиотеки `tensorflow_datasets`

Библиотека `tensorflow_datasets` предлагает аккуратную коллекцию бесплатно доступных наборов данных для обучения и оценки моделей ГО. Наборы данных хорошо сформатированы и снабжены информативными описаниями, включающими формат признаков и меток вместе с их типом и размерностью, а также ссылку на исходную статью, где был представлен тот или иной набор данных, в форме BibTeX. Еще одно преимущество в том, что все наборы данных предварительно обработаны и готовы к использованию как объекты `tf.data.Dataset`, поэтому можно напрямую применять все функции, раскрытые в предшествующих разделах. Итак, давайте взглянем на эти наборы данных в действии.

Прежде всего, потребуется установить библиотеку `tensorflow_datasets` посредством `pip` в командной строке:

```
pip install tensorflow-datasets
```

А теперь импортируем модуль `tensorflow_datasets` и посмотрим на список доступных наборов данных:

```
>>> import tensorflow_datasets as tfds
>>> print(len(tfds.list_builders()))
101
>>> print(tfds.list_builders()[:5])
['abstract_reasoning', 'aflw2k3d', 'amazon_us_reviews',
 'bair_robot_pushing_small', 'bigearthnet']
```

Выполнение предыдущего кода показывает, что в текущее время доступен 101 набор данных (на момент написания главы, но их количество, вероятно, будет увеличиваться), и мы выводим первые пять наборов данных. Извлечь набор данных можно двумя способами, которые мы раскроем ниже на примере извлечения двух наборов данных — CelebA (`celeb_a`) и MNIST.

Первый подход состоит из трех шагов.

1. Вызов функции построителя набора данных.
2. Выполнение метода `download_and_prepare()`.
3. Вызов метода `as_dataset()`.

Давайте отработаем первый шаг для набора данных CelebA и выведем соответствующее описание, которое предоставляется внутри библиотеки:

```
>>> celeba_bldr = tfds.builder('celeb_a')
>>> print(celeba_bldr.info.features)
FeaturesDict({'image': Image(shape=(218, 178, 3), dtype=tf.uint8),
'landmarks': FeaturesDict({'lefteye_x': Tensor(shape=(),
dtype=tf.int64), 'lefteye_y': Tensor(shape=(), dtype=tf.int64),
'righteye_x': Tensor(shape=(), dtype=tf.int64), 'righteye_y':
...
>>> print(celeba_bldr.info.features['image'])
Image(shape=(218, 178, 3), dtype=tf.uint8)
>>> print(celeba_bldr.info.features['attributes'].keys())
dict_keys(['5_o_Clock_Shadow', 'Arched_Eyebrows', ...
>>> print(celeba_bldr.info.citation)
@inproceedings{conf/iccv/LiuLWT15,
  added-at = {2018-10-09T00:00:00.000+0200},
  author = {Liu, Ziwei and Luo, Ping and Wang, Xiaogang and Tang, Xiaoou},
  biburl = {https://www.bibsonomy.org/bibtex/250e4959be61db325d2f02c1d8cd7bfbdb/dblp},
  booktitle = {ICCV},
  crossref = {conf/iccv/2015},
  ee = {http://doi.ieeecomputersociety.org/10.1109/ICCV.2015.425},
  interhash = {3f735aaa11957e73914bbe2ca9d5e702},
  intrahash = {50e4959be61db325d2f02c1d8cd7bfbdb},
  isbn = {978-1-4673-8391-2},
  keywords = {dblp},
  pages = {3730-3738},
  publisher = {IEEE Computer Society},
  timestamp = {2018-10-11T11:43:28.000+0200},
  title = {Deep Learning Face Attributes in the Wild.},
  url = {http://dblp.uni-trier.de/db/conf/iccv/iccv2015.html#LiuLWT15},
  year = 2015
}
```

В результате мы получаем полезную информацию, способствующую пониманию структуры набора данных CelebA. Признаки хранятся в виде словаря с тремя ключами: 'image', 'landmarks' и 'attributes'.

Элемент 'image' ссылается на изображение лица знаменитости. Элемент 'landmarks' ссылается на словарь извлеченных точек лица, таких как местоположение глаз, носа и т.д. Элемент 'attributes' представляет собой словарь из 40 атрибутов лица человека на изображении вроде мимики, макияжа, прически и т.п.

Затем мы вызовем метод `download_and_prepare()`. Он загрузит данные и сохранит их на диске в указанном подкаталоге для всех наборов данных TensorFlow. Если вы уже делали это ранее, то метод просто проверит, загружены ли данные, и не будет загружать их повторно в случае существования данных в указанном местоположении:

```
>>> celeba_bldr.download_and_prepare()
```

Далее мы создаем объект набора данных:

```
>>> datasets = celeba_bldr.as_dataset(shuffle_files=False)
>>> datasets.keys()
dict_keys(['test', 'train', 'validation'])
```

Набор данных уже расщеплен на обучающий, испытательный и проверочный наборы. Чтобы посмотреть, как выглядят примеры изображений, мы можем выполнить следующий код:

```
>>> ds_train = datasets['train']
>>> assert isinstance(ds_train, tf.data.Dataset)
>>> example = next(iter(ds_train))
>>> print(type(example))
<class 'dict'>
>>> print(example.keys())
dict_keys(['image', 'landmarks', 'attributes'])
```

Обратите внимание, что элементы в этом наборе данных поступают в словаре. Если мы хотим передать такой набор данных модели ГО с учителем во время обучения, тогда должны переформатировать его в виде кортежа (признаки, метка). Для метки мы будем использовать категорию 'Male' из атрибутов. Мы достигнем этого за счет применения трансформации через `map()`:

```
>>> ds_train = ds_train.map(lambda item:
...                          (item['image'],
...                          tf.cast(item['attributes']['Male'], tf.int32)))
```

В заключение давайте создадим пакеты из набора данных, извлечем пакет, содержащий 18 образцов, и визуализируем их вместе с метками:

```
>>> ds_train = ds_train.batch(18)
>>> images, labels = next(iter(ds_train))
>>> print(images.shape, labels)
(18, 218, 178, 3) tf.Tensor([0 0 0 1 1 1 0 1 1 0 1 1 0 1 0 1 1 1],
shape=(18,), dtype=int32)
>>> fig = plt.figure(figsize=(12, 8))
>>> for i, (image, label) in enumerate(zip(images, labels)):
...     ax = fig.add_subplot(3, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image)
...     ax.set_title('{}'.format(label), size=15)
>>> plt.show()
```

Образцы и их метки, извлеченные из набора данных `ds_train`, показаны на рис. 13.5.

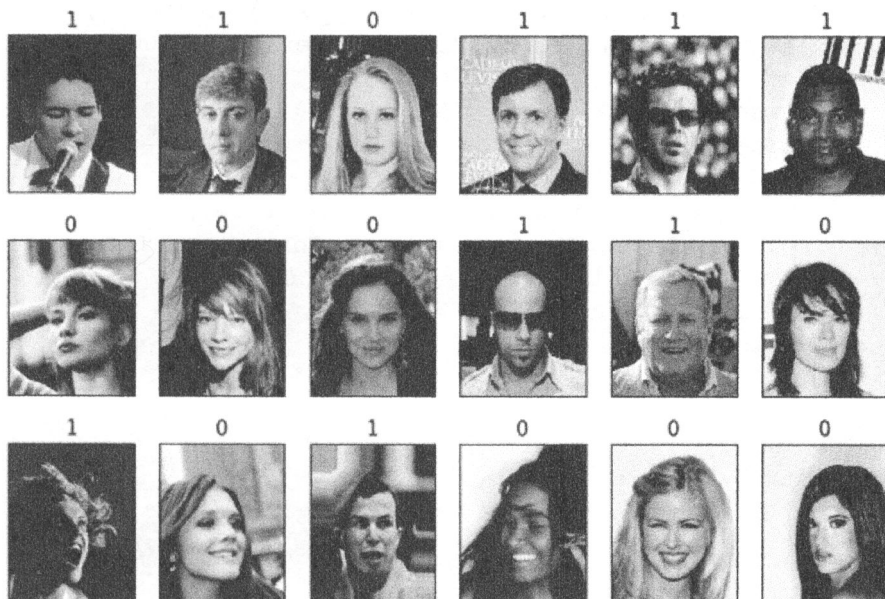


Рис. 13.5. Образцы и их метки, извлеченные из набора данных `ds_train`

Вот и все, что потребовалось предпринять для получения и использования набора данных с изображениями CelebA.

Мы продолжим исследованием второго подхода к получению набора данных из `tensorflow_datasets`. Существует функция-оболочка по имени `load()`, которая объединяет три шага по извлечению набора данных в один. Давайте посмотрим, как ее можно задействовать для получения набора данных MNIST:

```
>>> mnist, mnist_info = tfds.load('mnist', with_info=True,
...                               shuffle_files=False)
>>> print(mnist_info)
tfds.core.DatasetInfo(
  name='mnist',
  version=1.0.0,
  description='The MNIST database of handwritten digits.',
  urls=['https://storage.googleapis.com/cvdf-datasets/mnist/'],
  features=FeaturesDict({
    'image': Image(shape=(28, 28, 1), dtype=tf.uint8),
    'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=10)
  }),
  total_num_examples=70000,
  splits={
    'test': <tfds.core.SplitInfo num_examples=10000>,
    'train': <tfds.core.SplitInfo num_examples=60000>
  },
  supervised_keys=('image', 'label'),
  citation="""
    @article{lecun2010mnist,
      title={MNIST handwritten digit database},
      author={LeCun, Yann and Cortes, Corinna and Burges, CJ},
      journal={ATT Labs [Online]. Availablist},
      volume={2},
      year={2010}
    }

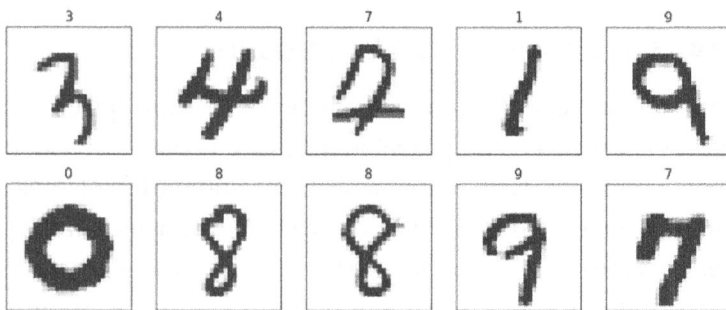
    """,
  redistribution_info=,
)

>>> print(mnist.keys())
dict_keys(['test', 'train'])
```

Легко заметить, что набор данных MNIST разбит на две части. Теперь мы можем взять часть для обучения, применить трансформацию с целью преобразования элементов из словаря в кортеж и визуализировать 10 образцов:

```
>>> ds_train = mnist['train']
>>> ds_train = ds_train.map(lambda item:
...                          (item['image'], item['label']))
>>> ds_train = ds_train.batch(10)
>>> batch = next(iter(ds_train))
>>> print(batch[0].shape, batch[1])
(10, 28, 28, 1) tf.Tensor([8 4 7 7 0 9 0 3 3 3], shape=(10,),
...                        dtype=int64)
>>> fig = plt.figure(figsize=(15, 6))
>>> for i, (image, label) in enumerate(zip(batch[0], batch[1])):
...     ax = fig.add_subplot(2, 5, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image[:, :, 0], cmap='gray_r')
...     ax.set_title('{}'.format(label), size=15)
>>> plt.show()
```

Извлеченные примеры рукописных цифр из набора данных MNIST продемонстрированы на рис. 13.6.



**Рис. 13.6.** Извлеченные примеры рукописных цифр из набора данных MNIST

На этом наш обзор построения наборов данных и манипулирования ими, а также получения наборов данных из библиотеки `tensorflow_datasets` завершен. Далее мы выясним, как строить нейросетевые модели в TensorFlow.





На  
заметку!

### Руководство по стилевому оформлению кода TensorFlow

Обратите внимание, что в официальном руководстве по стилевому оформлению кода TensorFlow ([https://www.tensorflow.org/community/style\\_guide](https://www.tensorflow.org/community/style_guide)) рекомендуется использовать для отступов в коде двухсимвольные промежутки. Тем не менее, в настоящей книге для отступов применяются четыре символа, т.к. это лучше согласуется с официальным руководством по стилевому оформлению кода Python и помогает корректно выделять синтаксис во многих текстовых редакторах и сопровождающих тетрадах Jupyter Notebook (<https://github.com/rasbt/python-machine-learning-book-3rd-edition>).

## Построение нейросетевой модели в TensorFlow

До сих пор в главе речь шла о базовых служебных компонентах TensorFlow, предназначенных для манипулирования тензорами и упорядочения данных в форматы, которые допускают выполнение прохода во время обучения. В текущем разделе мы, наконец, реализуем первую прогнозирующую модель в TensorFlow. Поскольку библиотека TensorFlow немного гибче, но также и сложнее библиотек для МО вроде `scikit-learn`, мы начнем с простой линейной регрессионной модели.

### API-интерфейс Keras в TensorFlow (`tf.keras`)

Keras представляет собой высокоуровневый API-интерфейс для нейронных сетей, который изначально разрабатывался с целью запуска поверх других библиотек, таких как TensorFlow и Theano. Библиотека Keras предлагает дружественный к пользователю и модульный программный интерфейс, позволяющий легко прототипировать и строить сложные модели с помощью всего лишь нескольких строк кода. Keras можно установить независимо от PyPI и затем сконфигурировать для использования TensorFlow в качестве своего внутреннего механизма. Библиотека Keras тесно интегрирована в TensorFlow и ее модули доступны через `tf.keras`. В версии TensorFlow 2.0 модуль `tf.keras` стал основным и рекомендуемым подходом к реализации моделей. Его преимущество связано с тем, что он поддерживает специфичную для TensorFlow функциональность, такую как конвейеры наборов дан-

ных посредством `tf.data`, которые обсуждались в предыдущем разделе. Для построения нейросетевых моделей в книге мы будем применять модуль `tf.keras`.

Как вы увидите в последующих подразделах, API-интерфейс Keras (`tf.keras`) делает построение нейросетевых моделей исключительно легким. Наиболее часто используемый подход к построению нейронной сети в TensorFlow предусматривает применение класса `tf.keras.Sequential`, который позволяет укладывать стопкой слои для формирования сети. Стопка слоев может быть предоставлена модели, определенной как `tf.keras.Sequential()`, в списке Python. В качестве альтернативы слои могут добавляться по одному с использованием метода `.add()`.

Кроме того, `tf.keras` делает возможным определение модели за счет создания подкласса класса `tf.keras.Model`. Это дает нам больший контроль над прямым проходом путем определения в классе модели метода `call()` для явного указания прямого прохода. Мы рассмотрим примеры обоих подходов к построению нейросетевых моделей с применением API-интерфейса `tf.keras`.

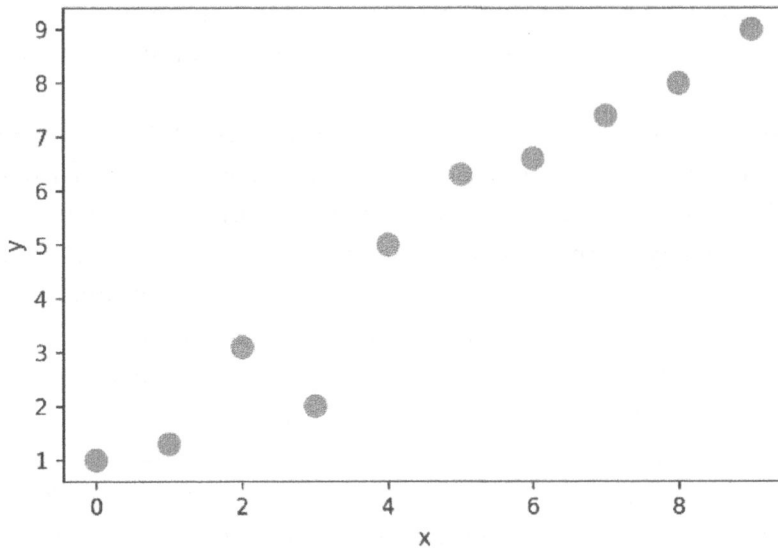
Наконец, как будет показано в последующих подразделах, построенные с использованием API-интерфейса `tf.keras` модели могут быть скомпилированы и обучены с помощью методов `.compile()` и `.fit()`.

## Построение линейной регрессионной модели

В текущем подразделе мы построим простую модель для решения задачи линейной регрессии. Первым делом мы создадим игрушечный набор данных в NumPy и визуализируем его:

```
>>> X_train = np.arange(10).reshape((10, 1))
>>> y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0, 6.3,
...                     6.6, 7.4, 8.0, 9.0])
>>> plt.plot(X_train, y_train, 'o', markersize=10)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

В результате обучающие образцы отобразятся в виде графика рассеяния, показанного на рис. 13.7.



*Рис. 13.7. График рассеяния с обучающими образцами*

Затем мы стандартизируем признаки (центрировав вокруг среднего и поделив на стандартное отклонение) и создаем объект `Dataset` из TensorFlow:

```
>>> X_train_norm = (X_train - np.mean(X_train))/np.std(X_train)
>>> ds_train_orig = tf.data.Dataset.from_tensor_slices(
...     (tf.cast(X_train_norm, tf.float32),
...     tf.cast(y_train, tf.float32)))
```

Далее мы можем определить модель для линейной регрессии как  $z = wx + b$ . Здесь мы собираемся воспользоваться API-интерфейсом Keras. В `tf.keras` имеются предварительно определенные слои для построения сложных нейросетевых моделей, но сначала нужно выяснить, как определять модель с нуля. Позже в главе вы увидите, каким образом применять эти предварительно определенные слои.

Для задачи регрессии мы определим новый класс, производный от класса `tf.keras.Model`. Создание подкласса `tf.keras.Model` позволяет использовать инструменты Keras для исследования, обучения и оценки модели. В конструкторе нашего класса мы определим параметры модели,  $w$  и  $b$ , которые соответствуют параметрам веса и смещения. В заключение мы определим метод `call()` для установления того, как модель будет генерировать свой выход на основе входных данных:

```
>>> class MyModel(tf.keras.Model):
...     def __init__(self):
...         super(MyModel, self).__init__()
...         self.w = tf.Variable(0.0, name='weight')
...         self.b = tf.Variable(0.0, name='bias')
...
...     def call(self, x):
...         return self.w * x + self.b
```

Затем мы создадим из класса `MyModel()` объект новой модели, который можно обучить с помощью обучающих данных. В API-интерфейсе Keras из TensorFlow предоставляется метод по имени `.summary()` для моделей, производных от `tf.keras.Model`, который позволяет получить послойную сводку по компонентам модели и количество параметров в каждом слое. Поскольку мы создали подкласс нашей модели из класса `tf.keras.Model`, то доступен также и метод `.summary()`. Но чтобы иметь возможность обращаться к методу `model.summary()`, нам необходимо указать модели размерность входа (количество признаков), для чего вызвать метод `model.build()` с ожидаемой формой входных данных:

```
>>> model = MyModel()
>>> model.build(input_shape=(None, 1))
>>> model.summary()
Model: "my_model"
```

Layer (type)	Output Shape	Param #
=====		
Total params: 2		
Trainable params: 2		
Non-trainable params: 0		

Модель: "my\_model"

Слой (тип)	Форма выхода	Кол-во параметров
=====		
Всего параметров: 2		
Обучаемых параметров: 2		
Необучаемых параметров: 0		

Обратите внимание на применение `None` в качестве заполнителя для первого измерения ожидаемого входного тензора в `model.build()`, что позволяет использовать произвольный размер пакета. Однако количество признаков фиксировано (1), т.к. напрямую соответствует количеству весовых параметров модели. Компоновка слоев и параметров модели после создания ее объекта вызовом метода `.build()` называется *поздним созданием переменных*. Для такой простой модели мы уже создали ее параметры в конструкторе; следовательно, указание `input_shape` в `build()` не оказывает дополнительного влияния на наши параметры, но все-таки оно необходимо, если мы хотим вызвать `model.summary()`.

После определения модели мы можем определить функцию издержек, которую нужно минимизировать с целью нахождения оптимальных весов модели. В рассматриваемом примере для функции издержек мы выберем *среднеквадратическую ошибку* (*mean squared error* — *MSE*). Кроме того, чтобы узнать весовые параметры модели, мы будем применять стохастический градиентный спуск. В текущем подразделе мы самостоятельно реализуем такое обучение посредством процедуры стохастического градиентного спуска, но в следующем подразделе для этого будем использовать методы `compile()` и `fit()` из библиотеки Keras.

Для реализации алгоритма стохастического градиентного спуска нам необходимо вычислять градиенты. Вместо того чтобы рассчитывать градиенты вручную, мы будем применять API-интерфейс `tf.GradientTape` из TensorFlow. Мы раскроем API-интерфейс `tf.GradientTape` и его разнообразные линии поведения в главе 14. Ниже приведен код:

```
>>> def loss_fn(y_true, y_pred):
...     return tf.reduce_mean(tf.square(y_true - y_pred))

>>> def train(model, inputs, outputs, learning_rate):
...     with tf.GradientTape() as tape:
...         current_loss = loss_fn(model(inputs), outputs)
...         dW, db = tape.gradient(current_loss, [model.w, model.b])
...         model.w.assign_sub(learning_rate * dW)
...         model.b.assign_sub(learning_rate * db)
```

Теперь мы можем установить гиперпараметры и обучить модель в течение 200 эпох. Мы создадим пакеты из набора данных и повторим набор данных с помощью `count=None`, что в результате даст бесконечно повторенный набор данных:

```

>>> tf.random.set_seed(1)
>>> num_epochs = 200
>>> log_steps = 100
>>> learning_rate = 0.001
>>> batch_size = 1
>>> steps_per_epoch = int(np.ceil(len(y_train) / batch_size))
>>> ds_train = ds_train_orig.shuffle(buffer_size=len(y_train))
>>> ds_train = ds_train.repeat(count=None)
>>> ds_train = ds_train.batch(1)
>>> Ws, bs = [], []

>>> for i, batch in enumerate(ds_train):
...     if i >= steps_per_epoch * num_epochs:
...         # прекратить бесконечный цикл
...         break
...     Ws.append(model.w.numpy())
...     bs.append(model.b.numpy())
...     bx, by = batch
...     loss_val = loss_fn(model(bx), by)
...     train(model, bx, by, learning_rate=learning_rate)
...     if i%log_steps==0:
...         print('Эпоха {:4d} Шаг {:2d} Потеря {:6.4f}'.format(
...             int(i/steps_per_epoch), i, loss_val))
Эпоха   0 Шаг   0 Потеря 43.5600
Эпоха  10 Шаг 100 Потеря 0.7530
Эпоха  20 Шаг 200 Потеря 20.1759
Эпоха  30 Шаг 300 Потеря 23.3976
Эпоха  40 Шаг 400 Потеря 6.3481
Эпоха  50 Шаг 500 Потеря 4.6356
Эпоха  60 Шаг 600 Потеря 0.2411
Эпоха  70 Шаг 700 Потеря 0.2036
Эпоха  80 Шаг 800 Потеря 3.8177
Эпоха  90 Шаг 900 Потеря 0.9416
Эпоха 100 Шаг 1000 Потеря 0.7035
Эпоха 110 Шаг 1100 Потеря 0.0348
Эпоха 120 Шаг 1200 Потеря 0.5404
Эпоха 130 Шаг 1300 Потеря 0.1170
Эпоха 140 Шаг 1400 Потеря 0.1195
Эпоха 150 Шаг 1500 Потеря 0.0944
Эпоха 160 Шаг 1600 Потеря 0.4670
Эпоха 170 Шаг 1700 Потеря 2.0695
Эпоха 180 Шаг 1800 Потеря 0.0020
Эпоха 190 Шаг 1900 Потеря 0.3612

```

Давайте взглянем на обученную модель и вычертим график рассеяния. Для испытательных данных мы создадим массив NumPy значений, равноотстоящих между 0 и 9. Поскольку мы обучали модель со стандартизированными признаками, то также применим стандартизацию к испытательным данным:

```
>>> print('Финальные параметры: ', model.w.numpy(), model.b.numpy())
Финальные параметры:  2.6576622 4.8798566

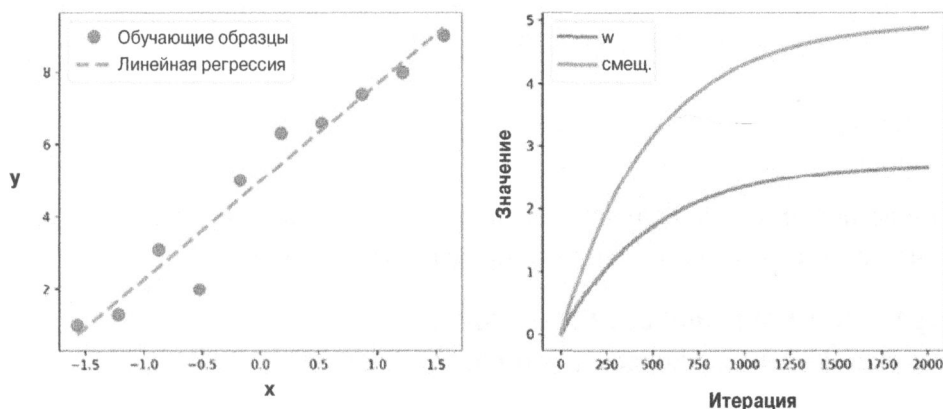
>>> X_test = np.linspace(0, 9, num=100).reshape(-1, 1)
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> y_pred = model(tf.cast(X_test_norm, dtype=tf.float32))

>>> fig = plt.figure(figsize=(13, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(X_train_norm, y_train, 'o', markersize=10)
>>> plt.plot(X_test_norm, y_pred, '--', lw=3)
>>> plt.legend(['Обучающие образцы', 'Линейная регрессия'],
...           fontsize=15)
>>> ax.set_xlabel('x', size=15)
>>> ax.set_ylabel('y', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> plt.plot(Ws, lw=3)
>>> plt.plot(bs, lw=3)
>>> plt.legend(['W', 'смещ.'], fontsize=15)
>>> ax.set_xlabel('Итерация', size=15)
>>> ax.set_ylabel('Значение', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

На рис. 13.8 показан график рассеяния обучающих образцов и обученной линейной регрессионной модели, а также хронология сходимости весов  $w$  и элемента смещения  $b$ :

## Обучение модели с помощью методов `.compile()` и `.fit()`

В предыдущем примере вы узнали, как обучить модель за счет написания специальной функции `train()` и применения оптимизации на базе стохастического градиентного спуска. Тем не менее, написание функции `train()` может оказаться повторяющейся задачей среди различных проектов.



**Рис. 13.8.** График рассеяния и хронология сходимости весов  $w$  и элемента смещения  $b$

В API-интерфейсе Keras библиотеки TensorFlow предлагается удобный метод `.fit()`, который можно вызывать на созданном объекте модели. Чтобы посмотреть, как работает прием, давайте создадим новую модель, после чего скомпилируем ее, выбрав оптимизатор, функцию потерь и метрики оценки:

```
>>> tf.random.set_seed(1)
>>> model = MyModel()
>>> model.compile(optimizer='sgd',
...               loss=loss_fn,
...               metrics=['mae', 'mse'])
```

Теперь мы можем просто вызвать метод `fit()` для обучения модели и передать ему набор данных в виде пакетов (наподобие `ds_train`, созданного в предыдущем примере). Однако на этот раз вы увидите, что мы можем передавать массивы NumPy для  $x$  и  $y$  напрямую без необходимости в создании набора данных:

```
>>> model.fit(X_train_norm, y_train,
...           epochs=num_epochs, batch_size=batch_size,
...           verbose=1)

Train on 10 samples
Epoch 1/200
10/10 [=====] - 0s 4ms/sample - loss:
27.8578 - mae: 4.5810 - mse: 27.8578
Epoch 2/200
```



```
10/10 [=====] - 0s 738us/sample - loss:
18.6640 - mae: 3.7395 - mse: 18.6640
...
Epoch 200/200
10/10 [=====] - 0s 1ms/sample - loss:
0.4139 - mae: 0.4942 - mse: 0.4139
```

После обучения модели визуализируйте результаты и удостоверьтесь в том, что они похожи на результаты предыдущего метода.

## **Построение многослойного персептрона для классификации цветков в наборе данных Iris**

В предыдущем примере мы строили модель с нуля. Мы обучали созданную модель с использованием оптимизации на основе стохастического градиентного спуска. Хотя мы начали свое путешествие с простейшего примера, вы видели, что определение модели с нуля даже для такого простого случая не является ни привлекательной, ни рекомендуемой практикой. Взамен TensorFlow предоставляет через `tf.keras.layers` уже определенные слои, которые могут применяться в качестве строительных блоков нейросетевой модели. В текущем разделе вы узнаете, как использовать предварительно определенные слои для решения задачи классификации с применением набора данных Iris и построения двухслойного персептрона с помощью API-интерфейса Keras. Первым делом мы получим данные из `tensorflow_datasets`:

```
>>> iris, iris_info = tfds.load('iris', with_info=True)
>>> print(iris_info)
```

В результате выводится информация о наборе данных Iris (не показанная здесь ради экономии пространства). Тем не менее, вы заметите в этой информации, что набор данных Iris поступает с единственной частью, а потому его придется самостоятельно расщепить на обучающую и испытательную части (и проверочную для надлежащего порядка МО). Давайте предположим, что мы хотим задействовать две трети набора данных для обучения и приберечь оставшиеся образцы для испытаний. В библиотеке `tensorflow_datasets` имеется удобный инструмент — объект `DatasetBuilder`, который позволяет определять части и расщеплять набор данных перед загрузкой. Дополнительные сведения о нем доступны по ссылке <https://www.tensorflow.org/datasets/splits>.

Альтернативный подход предусматривает загрузку полного набора данных и последующее использование методов `.take()` и `.skip()` для его расщепления на две части. Если набор данных изначально не перетасован, тогда мы также можем перетасовать его. Однако мы должны проявить крайнюю осторожность, потому что тасование способно привести к смешиванию обучающих/испытательных образцов, что в МО неприемлемо. Во избежание подобной ситуации при вызове метода `.shuffle()` мы обязаны установить аргумент `reshuffle_each_iteration` в `False`. Вот как выглядит код для расщепления набора данных на обучающий и испытательный наборы:

```
>>> tf.random.set_seed(1)
>>> ds_orig = iris['train']
>>> ds_orig = ds_orig.shuffle(150, reshuffle_each_iteration=False)
>>> ds_train_orig = ds_orig.take(100)
>>> ds_test = ds_orig.skip(100)
```

Далее, как уже демонстрировалось в предшествующих разделах, нам необходимо применить трансформацию посредством метода `.map()`, чтобы преобразовать словарь в кортеж:

```
>>> ds_train_orig = ds_train_orig.map(
...     lambda x: (x['features'], x['label']))
>>> ds_test = ds_test.map(
...     lambda x: (x['features'], x['label']))
```

Теперь мы готовы воспользоваться API-интерфейсом Keras для эффективного построения модели. В частности, с применением класса `tf.keras.Sequential` мы можем уложить стопкой несколько слоев Keras и построить нейронную сеть. Список всех доступных слоев Keras находится по ссылке [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/layers](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers). Для решения нашей задачи мы собираемся использовать слой `Dense` (`tf.keras.layers.Dense`), который также известен как *полносвязный* (*fully connected* — FC) или *линейный* слой и лучше всего может быть представлен посредством  $f(w \times x + b)$ , где  $x$  — входные признаки,  $w$  и  $b$  — матрица весов и вектор смещений, а  $f$  — функция активации.

Если подумать о слоях в нейронной сети, то каждый слой получает свои входы от предыдущего слоя; следовательно, его размерность (ранг и форма) является фиксированной. Обычно нам приходится самостоятельно за-

ботиться о размерности выхода только при проектировании архитектуры нейронной сети. (Примечание: первый слой будет исключением, но библиотека TensorFlow/Keras после определения модели позволяет принять решение относительно размерности входа первого слоя через *позднее создание переменных*.) Здесь мы хотим определить модель с двумя скрытыми слоями. Первый слой получает вход из четырех признаков и проецирует их на 16 нейронов. Второй слой получает выход предыдущего слоя (с размером 16) и проецирует его на три выходных нейрона, т.к. мы имеем три метки классов. Цель может быть достигнута с применением класса `Sequential` и слоя `Dense` из Keras:

```
>>> iris_model = tf.keras.Sequential([
...     tf.keras.layers.Dense(16, activation='sigmoid',
...                             name='fc1', input_shape=(4,)),
...     tf.keras.layers.Dense(3, name='fc2',
...                             activation='softmax')])
>>> iris_model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
fc1 (Dense)	(None, 16)	80
fc2 (Dense)	(None, 3)	51
Total params: 131		
Trainable params: 131		
Non-trainable params: 0		

Обратите внимание, что мы определили форму входа для первого слоя посредством `input_shape=(4,)` и потому уже не обязаны вызывать `.build()` для использования `iris_model.summary()`.

Выведенная сводка по модели показывает, что первый слой (`fc1`) имеет 80 параметров, а второй — 51 параметр. Вы можете проверить это с помощью выражения  $(n_{in} + 1) \times n_{out}$ , где  $n_{in}$  — количество входных элементов и  $n_{out}$  — количество выходных элементов. Вспомните, что для полносвязного (плотного) слоя обучаемыми параметрами являются матрица весов размера  $n_{in} \times n_{out}$  и вектор смещений размера  $n_{out}$ . Кроме того, как видите, мы при-

меняем сигмоидальную функцию активации для первого слоя и многопеременную активацию для последнего (выходного) слоя. Многопеременная активация в последнем слое используется с целью поддержки многоклассовой классификации, поскольку у нас есть три метки классов (вот почему выходной слой содержит три нейрона). Мы обсудим различные функции активации и их приложения позже в главе.

Затем мы скомпилируем модель, указав функцию потерь, оптимизатор и метрики для оценки:

```
>>> iris_model.compile(optimizer='adam',
...                     loss='sparse_categorical_crossentropy',
...                     metrics=['accuracy'])
```

Теперь мы можем обучить модель. Мы укажем 100 для количества эпох и 2 для размера пакета. В приведенном ниже коде мы построим бесконечно повторяющийся набор данных, который будет передаваться методу `fit()` для обучения модели. В рассматриваемом случае, чтобы метод `fit()` имел возможность отслеживать эпохи, он должен знать количество шагов в каждой эпохе.

Имея размер обучающих данных (100) и размер пакета (`batch_size`), мы можем определить количество шагов в каждой эпохе (`steps_per_epoch`):

```
>>> num_epochs = 100
>>> training_size = 100
>>> batch_size = 2
>>> steps_per_epoch = np.ceil(training_size / batch_size)
>>> ds_train = ds_train_orig.shuffle(buffer_size=training_size)
>>> ds_train = ds_train.repeat()
>>> ds_train = ds_train.batch(batch_size=batch_size)
>>> ds_train = ds_train.prefetch(buffer_size=1000)
>>> history = iris_model.fit(ds_train, epochs=num_epochs,
...                          steps_per_epoch=steps_per_epoch,
...                          verbose=0)
```

Возвращенная переменная `history` хранит потерю при обучении и правильность при обучении (т.к. они были указаны в качестве метрик в вызове `iris_model.compile()`) после каждой эпохи. Мы можем задействовать ее для визуализации кривых обучения:

```

>>> hist = history.history
>>> fig = plt.figure(figsize=(12, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(hist['loss'], lw=3)
>>> ax.set_title('Потеря при обучении', size=15)
>>> ax.set_xlabel('эпохи', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(hist['accuracy'], lw=3)
>>> ax.set_title('Правильность при обучении', size=15)
>>> ax.set_xlabel('эпохи', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()

```

Кривые обучения (потеря при обучении и правильность при обучении) показаны на рис. 13.9.

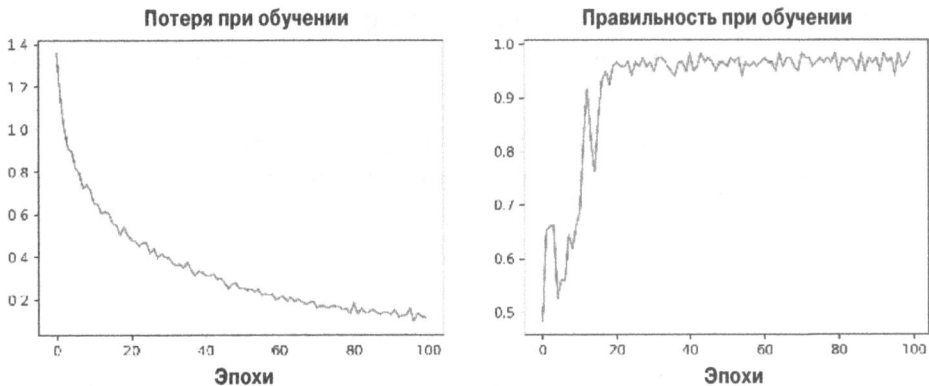


Рис. 13.9. Кривые обучения

## Оценка обученной модели на испытательном наборе данных

Поскольку для метрики оценки в `iris_model.compile()` было указано значение 'accuracy', теперь мы можем напрямую оценить модель на испытательном наборе данных:

```

>>> results = iris_model.evaluate(ds_test.batch(50), verbose=0)
>>> print('Потеря при испытании: {:.4f} Правильность при
испытании: {:.4f}'.format(*results))
Потеря при испытании: 0.0692 Правильность при испытании: 0.9800

```

Обратите внимание, что мы также должны создать пакеты из испытательного набора данных, чтобы обеспечить корректность измерения (ранг) входа модели. Как обсуждалось ранее, вызов `.batch()` увеличивает ранг извлеченных тензоров на 1. Входные данные для `.evaluate()` обязаны иметь одно назначенное измерение для пакета, хотя здесь (при оценке) размер пакета роли не играет. Таким образом, если мы передаем `ds_batch.batch(50)` методу `.evaluate()`, то весь испытательный набор данных будет обработан в одном пакете размером 50, но случае передачи `ds_batch.batch(1)` обработаются 50 пакетов размером 1.

## Сохранение и повторная загрузка обученной модели

Обученные модели можно сохранять на диске для применения в будущем. Вот как это делается:

```
>>> iris_model.save('iris-classifier.h5',  
...                 overwrite=True,  
...                 include_optimizer=True,  
...                 save_format='h5')
```

В первом параметре указывается имя файла. Вызов `iris_model.save()` сохранит архитектуру модели и все параметры, которые она узнала. Тем не менее, если вы хотите сохранить только архитектуру, тогда можете воспользоваться методом `iris_model.to_json()`, который сохраняет конфигурацию модели в формате JSON. Если же вы хотите сохранить только веса модели, то можете вызывать метод `iris_model.save_weights()`. В `save_format` можно указать либо `'h5'` для формата HDF5, либо `'tf'` для формата TensorFlow.

А теперь давайте повторно загрузим сохраненную модель. Поскольку мы сохранили и архитектуру, и веса модели, повторно загрузить параметры очень легко с помощью единственной строки кода:

```
>>> iris_model_new = tf.keras.models.load_model('iris-classifier.h5')
```

Попробуйте проконтролировать архитектуру модели, вызвав `iris_model_new.summary()`.

Наконец, мы оценим новую повторно загруженную модель на испытательном наборе данных, чтобы проверить, будут ли результаты такими же, как прежде:

```
>>> results = iris_model_new.evaluate(ds_test.batch(33), verbose=0)
>>> print('Потеря при испытании: {:.4f} Правильность при
испытании: {:.4f}'.format(*results))
Потеря при испытании: 0.0692 Правильность при испытании: 0.9800
```

## Выбор функций активации для многослойных нейронных сетей

До сих пор ради простоты в контексте многослойных нейронных сетей прямого распространения мы обсуждали только сигмоидальную функцию активации; она использовалась в скрытом и выходном слоях при реализации многослойного персептрона в главе 12.

Обратите внимание, что для краткости в этой книге мы называем сигмоидальную логистическую функцию  $\sigma(z) = \frac{1}{1 + e^{-z}}$  просто *сигмоидальной функцией*, что общепринято в литературе по МО. В последующих подразделах мы обсудим альтернативные нелинейные функции, которые полезны для реализации многослойных нейронных сетей.

Формально в качестве функции активации в многослойных нейронных сетях мы можем применять любую функцию при условии, что она дифференцируема. Можно использовать даже линейные функции активации, такие как в Adaline (см. главу 2). Однако на практике было бы не особенно полезно применять линейные функции активации для скрытого и выходного слоев, потому что мы хотим привнести нелинейность в типовую искусственную нейронную сеть, чтобы получить возможность решения сложных задач. В конце концов, сумма линейных функций дает линейную функцию.

Логистическая (сигмоидальная) функция активации, которую мы использовали в главе 12, пожалуй, наиболее близко имитирует концепцию нейрона в мозге — мы можем думать о ней, как о вероятности, возбудится нейрон или нет. Тем не менее, логистическая (сигмоидальная) функция активации может оказаться проблематичной при наличии высокого отрицательного входа, т.к. в этом случае выход сигмоидальной функции будет близким к нулю. Если сигмоидальная функция возвращает выход, близкий к нулю, тогда нейронная сеть будет обучаться очень медленно и возрастет вероятность того, что во время обучения она попадет в локальные минимумы. Именно потому разработчики часто отдают предпочтение гиперболическому тангенсу в качестве функции активации внутри скрытых слоев.

До того, как перейти к обсуждению функции гиперболического тангенса, давайте подведем краткие итоги по основам логистической функции и рассмотрим обобщение, которое делает ее более полезной для задач многозначной классификации.

## Краткое повторение логистической функции

Как упоминалось в начале раздела, логистическая функция фактически является особым случаем сигмоидальной функции. В разделе, посвященном логистической регрессии, главы 3 речь шла о том, что мы можем применять логистическую функцию для моделирования вероятности того, что образец  $x$  принадлежит к положительному классу (классу 1) в задаче двоичной классификации.

Общий вход  $z$  показан в следующем уравнении:

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_ix_i = w^T x$$

Вот как будет вычисляться логистическая функция:

$$\phi_{\text{логистическая}}(z) = \frac{1}{1 + e^{-z}}$$

Обратите внимание, что  $w_0$  представляет собой элемент смещения (пересечение с осью  $y$ , т.е.  $x_0 = 0$ ). Чтобы привести более конкретный пример, предположим, что у нас есть модель с двумерными точками данных  $x$  и следующими весовыми коэффициентами, присвоенными вектору  $w$ :

```
>>> import numpy as np
>>> X = np.array([1, 1.4, 2.5]) ## первое значение должно быть равно 1
>>> w = np.array([0.4, 0.3, 0.5])
>>> def net_input(X, w):
...     return np.dot(X, w)
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)
>>> print('P(y=1|x) = %.3f' % logistic_activation(X, w))
P(y=1|x) = 0.888
```



Если мы рассчитаем общий вход ( $z$ ) и используем его для логистической активации нейрона с указанными значениями признаков и весовыми коэффициентами, то получим значение 0.888, которое можно интерпретировать как 88.8%-ную вероятность того, что конкретный образец  $x$  принадлежит положительному классу.

В главе 12 мы применяли прием унитарного кодирования для представления многоклассовых достоверных меток и спроектировали выходной слой, состоящий из множества элементов с логистической активацией. Однако, как демонстрируется в примере кода ниже, выходной слой, включающий множество элементов с логистической активацией, не выпускает содержательные и интерпретируемые значения вероятностей:

```
>>> # W : массив с формой = (n_output_units, n_hidden_units+1)
>>> #    первый столбец содержит элементы смещения
>>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...              [0.2, 0.4, 1.0, 0.2],
...              [0.6, 1.5, 1.2, 0.7]])

>>> # A : массив данных с формой = (n_hidden_units + 1, n_samples)
>>> #    первый столбец должен содержать значения 1
>>> A = np.array([[1, 0.1, 0.4, 0.6]])

>>> Z = np.dot(W, A[0])
>>> y_probas = logistic(Z)
>>> print('Общий вход: \n', Z)
Общий вход:
[ 1.78  0.76  1.65]
>>> print('Выходные элементы:\n', y_probas)
Выходные элементы:
[ 0.85569687  0.68135373  0.83889105]
```

В выводе видно, что результирующие значения не могут интерпретироваться как вероятности для задачи с тремя классами. Причина в том, что значения не дают в сумме 1. Тем не менее, этот факт не является крупной проблемой, если мы используем модель только для прогнозирования меток классов, а не вероятностей членства в классах. Один из способов прогнозирования метки класса на основе полученных ранее выходных элементов предусматривает применение значения максимума:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Спрогнозированная метка класса: %d' % y_class)
Спрогнозированная метка класса: 0
```

В определенных контекстах может быть полезно вычислять содержательные вероятности для многоклассовых прогнозов. В следующем разделе мы взглянем на обобщение логистической функции — многопеременную логистическую функцию `softmax`, которая поможет решить такую задачу.

## Оценка вероятностей классов в многоклассовой классификации через многопеременную логистическую функцию

В предыдущем разделе мы показали, как можно получить метку класса с использованием функции `argmax`. Ранее в разделе “Построение многослойного персептрона для классификации цветков в наборе данных Iris” в последнем слое модели на основе многослойного персептрона мы определяли `activation='softmax'`. Функция `softmax` является мягкой формой функции `argmax`; вместо предоставления одиночного индекса класса она выдает вероятность каждого класса. Следовательно, она позволяет вычислять содержательные вероятности в многоклассовых окружениях (полиномиальная логистическая регрессия).

В функции `softmax` вероятность того, что определенный образец с общим входом  $z$  принадлежит  $i$ -тому классу, может быть вычислена с помощью члена нормализации в знаменателе, т.е. суммы экспоненциально взвешенных линейных функций:

$$p(z) = \phi(z) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}}$$

Чтобы посмотреть на функцию `softmax` в действии, давайте напомним соответствующий код Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))

>>> y_probab = softmax(Z)
>>> print('Вероятности:\n', y_probab)
Вероятности:
[ 0.44668973  0.16107406  0.39223621]

>>> np.sum(y_probab)
1.0
```

Несложно заметить, что спрогнозированные вероятности классов теперь дают в сумме 1, как и можно было ожидать. Также примечательно то, что спрогнозированная метка класса оказывается такой же, как и в случае применения функции `argmax` к логистическому выходу.

Содействовать пониманию результата функции `softmax` может ее представление как *нормализованного* выхода, который удобен для получения прогнозов принадлежности к классам в многоклассовых конфигурациях. Следовательно, при построении модели для многоклассовой классификации в TensorFlow мы можем использовать функцию `tf.keras.activations.softmax()`, чтобы оценить вероятности членства в каждом классе для входного пакета образцов. В приведенном ниже коде мы покажем, как применять функцию активации `softmax` из TensorFlow, преобразовав `Z` в тензор с дополнительным измерением, зарезервированным для размера пакета:

```
>>> import tensorflow as tf
>>> Z_tensor = tf.expand_dims(Z, axis=0)
>>> tf.keras.activations.softmax(Z_tensor)
<tf.Tensor: id=21, shape=(1, 3), dtype=float64,
numpy=array([[0.44668973, 0.16107406, 0.39223621]])>
```

## Расширение выходного спектра с использованием гиперболического тангенса

Еще одной сигмоидальной функцией, часто применяемой в скрытых слоях искусственных нейронных сетей, является *гиперболический тангенс* (`tanh`), который можно интерпретировать как масштабированную версию логистической функции:

$$\phi_{\text{логистическая}}(z) = \frac{1}{1 + e^{-z}}$$

$$\phi_{\text{tanh}}(z) = 2 \times \phi_{\text{логистическая}}(2z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Преимущество гиперболического тангенса перед логистической функцией в том, что он имеет более широкий выходной спектр и диапазоны в открытом интервале  $(-1, 1)$ , которые могут улучшить сходимость алгоритма обратного распространения (“Neural Networks for Pattern Recognition”

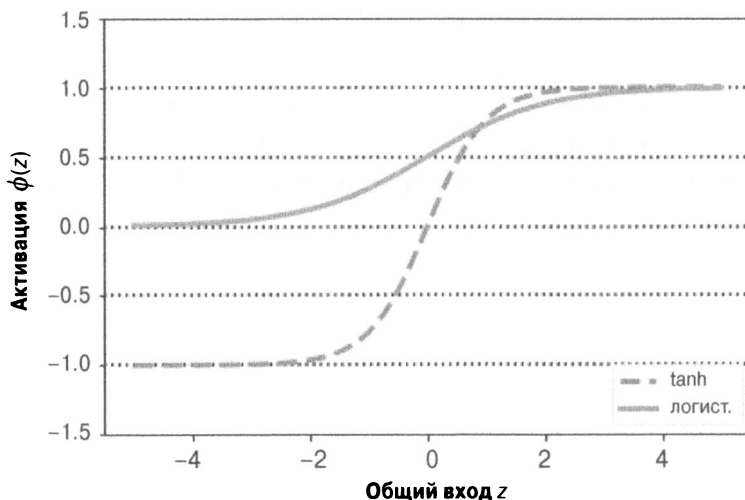
(Нейронные сети для распознавания образов), К. Бишоп, Oxford University Press, с. 500–501 (1995 г.)).

Напротив, логистическая функция возвращает выходной сигнал, находящийся в открытом интервале (0, 1). Чтобы сравнить логистическую функцию и гиперболический тангенс, мы построим графики двух сигмоидальных функций:

```
>>> import matplotlib.pyplot as plt
>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('общий вход $z$')
>>> plt.ylabel('активация $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle=':')
>>> plt.axhline(0.5, color='black', linestyle=':')
>>> plt.axhline(0, color='black', linestyle=':')
>>> plt.axhline(-0.5, color='black', linestyle=':')
>>> plt.axhline(-1, color='black', linestyle=':')
>>> plt.plot(z, tanh_act,
...         linewidth=3, linestyle='--',
...         label='tanh')
>>> plt.plot(z, log_act,
...         linewidth=3,
...         label='логист.')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
```

Как видно на рис. 13.10, формы двух сигмоидальных кривых выглядят очень похожими, но выходное пространство у функции `tanh` в два раза больше, чем у функции `logistic`.

Обратите внимание, что в целях иллюстрации мы реализовали функции `logistic` и `tanh` многословно. На практике мы можем использовать функцию `tanh` из NumPy.



**Рис. 13.10.** Графики двух сигмоидальных функций для сравнения логистической функции и гиперболического тангенса

В качестве альтернативы для получения тех же результатов при построении нейросетевой модели можно применять функцию `tf.keras.activations.tanh()` из TensorFlow:

```
>>> np.tanh(z)
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])

>>> tf.keras.activations.tanh(z)
<tf.Tensor: id=14, shape=(2000,), dtype=float64, numpy=
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])>
```

Вдобавок логистическая функция доступна в модуле `special` из SciPy:

```
>>> from scipy.special import expit
>>> expit(z)
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669,
        0.99324034, 0.99327383])
```

Подобным образом для выполнения того же самого вычисления можно использовать функцию `tf.keras.activations.sigmoid()` из TensorFlow:

```
>>> tf.keras.activations.sigmoid(z)
<tf.Tensor: id=16, shape=(2000,), dtype=float64, numpy=
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669,
        0.99324034, 0.99327383])>
```

## Активация на основе выпрямленного линейного элемента

*Выпрямленный линейный элемент* (*rectified linear unit* — *ReLU*) — еще одна функция активации, которая часто применяется в глубоких нейронных сетях. Прежде чем погружаться в детали ReLU, мы должны сделать шаг назад и понять проблему исчезновения градиентов, присущую активациям на основе гиперболического тангенса и логистической функции.

Пусть у нас первоначально имеется общий вход  $z_1 = 20$ , который изменяется на  $z_2 = 25$ . Вычислив активацию в форме гиперболического тангенса, мы получаем  $\phi(z_1) = 1.0$  и  $\phi(z_2) = 1.0$ , что указывает на отсутствие изменений в выходе (из-за асимптотического поведения функции гиперболического тангенса и численных ошибок).

Это означает, что производная активаций относительно общего входа уменьшается, когда  $z$  становится большим. В результате выяснение весов на стадии обучения крайне замедляется, т.к. члены градиента могут быть очень близкими к нулю. Активация ReLU решает такую проблему. Математически ReLU определяется следующим образом:

$$\phi(z) = \max(0, z)$$

Кроме того, ReLU — нелинейная функция, которая хорошо подходит для изучения сложных функций с нейронными сетями. Более того, производная ReLU относительно входа всегда равна 1 для положительных входных значений. По этой причине она решает проблему исчезновения градиентов, что делает ее подходящей для глубоких нейронных сетей. Вот как можно применять активацию ReLU в TensorFlow:

```
>>> tf.keras.activations.tanh(z)
<tf.Tensor: id=23, shape=(2000,), dtype=float64, numpy=array([0.    ,
 0.    , 0.    , ..., 4.985, 4.99 , 4.995])>
```

В следующей главе мы будем использовать ReLU в качестве функции активации для многослойных сверточных нейронных сетей.

Теперь, больше зная о различных функциях активации, которые обычно применяются в искусственных нейронных сетях, мы завершим настоящий раздел обзором разнообразных функций активации, встречающихся в книге (рис. 13.11).

Список всех функций активации, доступных в API-интерфейсе Keras, находится по ссылке [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/activations](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/activations).

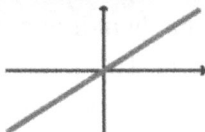
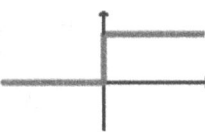
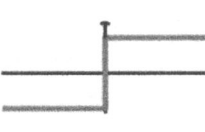
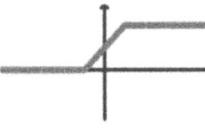

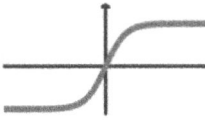
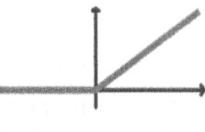
Функция активации	Уравнение	Пример	Одномерный граф
Линейная	$\phi(z) = z$	Adaline, линейная регрессия	
Единичная ступенчатая (функция Хевисайда)	$\phi(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Разновидность персептрона	
Знака (знаковая)	$\phi(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Разновидность персептрона	
Кусочно-линейная	$\phi(z) = \begin{cases} 0 & z \leq -1/2 \\ z + 1/2 & -1/2 \leq z \leq 1/2 \\ 1 & z \geq 1/2 \end{cases}$	Метод опорных векторов	
Логистическая (сигмоидальная)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Логистическая регрессия, многослойная нейронная сеть	
Гиперболический тангенс (tanh)	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Многослойная нейронная сеть, рекуррентные нейронные сети	
ReLU	$\phi(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Многослойная нейронная сеть, сверточные нейронные сети	

Рис. 13.11. Функции активации, встречающиеся в книге

## Резюме

В этой главе вы научились пользоваться TensorFlow — библиотекой с открытым кодом для численных расчетов, особо сконцентрированной на ГО. Хотя из-за дополнительной сложности, связанной с поддержкой ГП, библиотека TensorFlow менее удобна в применении по сравнению с NumPy, она позволяет крайне рационально определять и обучать крупные многослойные нейронные сети.

Кроме того, вы узнали об использовании API-интерфейса Keras библиотеки TensorFlow для построения сложных моделей МО и нейронных сетей и эффективного их запуска. Мы исследовали построение моделей в TensorFlow путем их определения с нуля посредством создания подклассов класса `tf.keras.Model`. Реализация моделей может оказаться утомительной, поскольку нам приходится оперировать на уровне умножений матрицы на вектор и определять все детали каждой операции. Однако преимущество в том, что мы как разработчики получаем возможность комбинировать такие базовые операции и строить более сложные модели. Вдобавок вы ознакомились с модулем `tf.keras.layers`, который позволяет строить нейросетевые модели гораздо легче, чем их реализация с нуля.

Наконец, вы узнали о различных функциях активации, а также ознакомились с их поведением и случаями применения. В частности, мы рассмотрели функции `tanh`, `softmax` и `ReLU`.

В следующей главе мы продолжим путешествие и погрузимся глубже в TensorFlow, открыв для себя работу с декорированием функций TensorFlow и оценщиками TensorFlow. Попутно вы освоите много новых концепций, таких как переменные и столбцы признаков.





## ПОГРУЖАЕМСЯ ГЛУБЖЕ — МЕХАНИКА TENSORFLOW

В главе 13 мы объяснили, как определять и манипулировать тензорами, а также работать с API-интерфейсом `tf.data` для построения входных конвейеров. Более того, мы построили и обучили многослойный персептрон для классификации цветков в наборе данных, используя API-интерфейс Keras библиотеки TensorFlow (`tf.keras`).

Теперь, когда у вас есть практический опыт работы с обучением нейронной сети TensorFlow и МО, самое время углубиться в библиотеку TensorFlow и исследовать предлагаемый ею богатый набор функциональных средств, который сделает возможным реализацию более развитых моделей ГО в последующих главах.

В этой главе мы задействуем различные аспекты API-интерфейса TensorFlow для реализации нейронных сетей. В частности, мы снова будем применять API-интерфейс Keras, который предоставляет множество уровней абстракции, делая реализацию стандартных архитектур очень удобной. Кроме того, TensorFlow позволяет реализовывать специальные слои нейронных сетей, что крайне полезно в исследовательских проектах, которые требуют дополнительной подстройки. Позже в главе мы реализуем специальный слой такого рода.

Чтобы проиллюстрировать разные способы построения моделей с использованием API-интерфейса Keras, мы также рассмотрим классическую задачу *исключающего ИЛИ (XOR)*. Первым делом мы построим многослойный персептрон с применением класса `Sequential`. Затем мы исследуем

другие методы, такие как создание подклассов класса `tf.keras.Model` для определения специальных слоев. В заключение мы раскроем высокоуровневый API-интерфейс TensorFlow по имени `tf.estimator`, который инкапсулирует шаги МО от низкоуровневого входа до прогнозирования.

В главе будут обсуждаться следующие темы:

- понятие графов TensorFlow и работа с ними, а также переход на версию TensorFlow v2;
- декорирование функций для компиляции графов;
- работа с переменными TensorFlow;
- решение классической задачи XOR и понятие емкости модели;
- построение сложных нейросетевых моделей с использованием класса `Model` и функционального API-интерфейса библиотеки Keras;
- расчет градиентов с применением автоматического дифференцирования и класса `tf.GradientTape`;
- работа с оценщиками TensorFlow.

## Ключевые средства TensorFlow

Библиотека TensorFlow предлагает масштабируемый, многоплатформенный программный интерфейс для реализации и запуска алгоритмов МО. Относительную стабильность и зрелость API-интерфейс TensorFlow обрел в выпуске 1.0, появившемся в 2017 году, а в своем последнем выпуске 2.0, который стал доступным в 2019 году и применяется в настоящей книге, он был серьезно реконструирован.

С момента первоначального выпуска в 2015 году TensorFlow стала самой широко используемой библиотекой для ГО. Однако один из проблемных аспектов заключался в том, что библиотека TensorFlow была построена на основе статических вычислительных графов. Статические вычислительные графы обладают рядом преимуществ, среди которых лучшая внутренняя оптимизация и поддержка более широкого спектра аппаратных устройств; тем не менее, статические вычислительные графы требуют отдельных шагов объявления и оценки, что затрудняет интерактивное конструирование и работу с нейронными сетями.

Серьезно относясь ко всем пользовательским отзывам, команда разработчиков в версии TensorFlow 2.0 решила сделать принятыми по умолчанию динамические вычислительные графы, которые повышают удобство создания и обучения нейронных сетей. В следующем разделе мы раскроем ряд важных изменений версии TensorFlow v2 по сравнению с TensorFlow v1.x. Динамические вычислительные графы позволяют чередовать шаги объявления и оценки, так что версия библиотеки TensorFlow 2.0 выглядит более естественной для пользователей Python и NumPy, нежели ее предшествующие версии. Однако имейте в виду, что TensorFlow 2.0 по-прежнему разрешает пользователям работать со “старым” API-интерфейсом TensorFlow v1.x через подмодуль `tf.compat`. Это содействует более плавному переходу пользователей на новый API-интерфейс TensorFlow v2.

Ключевым средством TensorFlow, которое уже упоминалось в главе 13, является способность работать с одним или несколькими ГП, что позволяет пользователям очень эффективно обучать модели ГО на больших наборах данных в крупномасштабных системах.

Хотя TensorFlow представляет собой библиотеку с открытым кодом, которая может свободно использоваться кем угодно, ее разработка финансируется и поддерживается Google. В процесс вовлечена крупная команда специалистов по программному обеспечению, постоянно расширяющих и совершенствующих библиотеку. Благодаря открытости кода библиотека TensorFlow также имеет сильную поддержку со стороны разработчиков вне компании Google, которые активно вносят свой вклад и обеспечивают пользовательскую обратную связь.

В итоге библиотека TensorFlow становится более полезной как для исследователей из научных кругов, так и для разработчиков. Добавочное следствие из указанных факторов заключается в том, что TensorFlow снабжается обширной документацией и учебными пособиями, оказывающими помощь новым пользователям.

Последней, но не менее важной характеристикой библиотеки TensorFlow, является поддержка мобильного развертывания, что делает ее весьма подходящим инструментом для производственной среды.

## Вычислительные графы TensorFlow: переход на TensorFlow v2

Библиотека TensorFlow выполняет свои вычисления на базе *ориентированного ациклического графа* (*directed acyclic graph* — DAG). В TensorFlow v1.x такие графы можно явно определять посредством низкоуровневого API-интерфейса, хотя это было не особо тривиально для крупных и сложных моделей. В настоящем разделе мы посмотрим, как определять графы подобного рода для простой арифметической операции. Затем мы выясним, каким образом переносить граф в TensorFlow v2, обсудим *энергичное выполнение* (*eager execution*) и парадигму динамических графов, а также рассмотрим декорирование функций для более быстрых вычислений.

### Понятие вычислительных графов

Внутренне библиотека TensorFlow опирается на построение вычислительного графа, который затем применяется для выведения взаимосвязей между тензорами на всем пути от входа до выхода. Пусть у нас есть тензоры  $a$ ,  $b$  и  $c$  ранга 0 (скаляры) и мы хотим оценить уравнение  $z = 2 \times (a - b) + c$ . Такая оценка может быть представлена как вычислительный граф, показанный на рис. 14.1.



**Рис. 14.1.** Вычислительный граф для оценки уравнения  $z = 2 \times (a - b) + c$

На рис. 14.1 видно, что вычислительный граф выглядит просто как сеть из узлов. Каждый узел представляет операцию, которая применяет функцию к своему входному тензору или тензорам и возвращает ноль или большее количество тензоров в качестве выхода. Библиотека TensorFlow строит этот вычислительный граф и использует его для надлежащего расчета градиентов. В последующих подразделах мы рассмотрим несколько примеров создания графа в стиле версий TensorFlow v1.x и v2 для вычисления, приведенного на рис. 14.1.

## Создание графа в TensorFlow v1.x

В ранней версии низкоуровневого API-интерфейса TensorFlow (v1.x) граф должен был объявляться явно. Ниже перечислены индивидуальные шаги для построения, компиляции и оценки вычислительного графа в TensorFlow v1.x.

1. Создать новый пустой вычислительный граф.
2. Добавить в вычислительный граф узлы (тензоры и операции).
3. Провести оценку (выполнить) граф:
  - а) начать новый сеанс;
  - б) инициализировать переменные в графе;
  - в) запустить вычислительный граф в этом сеансе.

Прежде чем взглянуть на динамический подход, принятый в TensorFlow v2, давайте рассмотрим простой пример, который проиллюстрирует создание графа в TensorFlow v1.x для оценки, показанной на рис. 14.1. Переменные  $a$ ,  $b$  и  $c$  являются скалярами (одиночными числами) и мы определим их как константы TensorFlow. Затем можно создать граф, вызвав `tf.Graph()`. Переменные и операции представляют узлы графа, которые мы определяем следующим образом:

```
## Стил TF v1.x
>>> g = tf.Graph()

>>> with g.as_default():
...     a = tf.constant(1, name='a')
...     b = tf.constant(2, name='b')
...     c = tf.constant(3, name='c')
...     z = 2*(a-b) + c
```

В коде мы сначала определяем граф `g` посредством `g=tf.Graph()`. Затем мы добавляем узлы в граф `g` с применением `with g.as_default()`. Тем не менее, имейте в виду, что если мы не создаем граф явно, то всегда существует стандартный граф, к которому автоматически будут добавляться переменные и операции.

В TensorFlow v1.x сеанс представляет собой среду, в которой могут выполняться операции и тензоры графа. Класс `Session` был удален из версии TensorFlow v2; однако на данный момент он все еще доступен через подмодуль `tf.compat`, чтобы обеспечить совместимость с TensorFlow v1.x. Объект сеанса можно создать вызовом `tf.compat.v1.Session()`, который способен принимать в качестве аргумента существующий граф (здесь `g`), как в `tf.Session(graph=g)`.

После запуска графа в сеансе TensorFlow мы можем выполнить его узлы, т.е. оценить его тензоры или выполнить его операции. Оценка каждого отдельного тензора влечет за собой вызов его метода `eval()` внутри текущего сеанса. Во время оценки специфического тензора в графе библиотека TensorFlow должна выполнить все предшествующие узлы в графе, пока не достигнет интересующего узла. При наличии одного или большего числа переменных-заполнителей нам также необходимо предоставить для них значения, как будет показано позже в главе.

После определения статического графа в предыдущем фрагменте кода мы можем выполнить этот граф в сеансе TensorFlow и оценить тензор `z`:

```
## Стил TF v1.x
>>> with tf.compat.v1.Session(graph=g) as sess:
...     print('Результат: z =', sess.run(z))
Результат: z = 1
```

## Перенос графа в TensorFlow v2

Далее давайте посмотрим, как имеющийся код можно перенести в TensorFlow v2. В версии TensorFlow v2 по умолчанию используются динамические (в противоположность статическим) графы (в TensorFlow это также называется энергичным выполнением), которые позволяют оценивать операцию на лету. Следовательно, мы не обязаны явно создавать граф и сеанс, что делает поток разработки более удобным:

```
## Стиль TF v2
>>> a = tf.constant(1, name='a')
>>> b = tf.constant(2, name='b')
>>> c = tf.constant(3, name='c')
>>> z = 2*(a - b) + c
>>> tf.print('Результат: z= ', z)
Результат: z = 1
```

## Загрузка входных данных в модель: стиль TensorFlow v1.x

Еще одно важное усовершенствование версии TensorFlow v2 по сравнению с TensorFlow v1.x касается того, каким образом можно загружать данные в модели. В TensorFlow v2 мы можем напрямую подавать данные в форме переменных Python или массивов NumPy. Тем не менее, когда применяется низкоуровневый API-интерфейс TensorFlow v1.x, мы должны создавать переменные-заполнители для снабжения модели входными данными. Продолжая приведенный ранее пример простого вычислительного графа,  $z = 2 \times (a - b) + c$ , давайте предположим, что  $a$ ,  $b$  и  $c$  — входные тензоры ранга 0. Затем мы можем определить три заполнителя, которые будем использовать для “подачи” данных модели через так называемый словарь `feed_dict`:

```
## Стиль TF-v1.x
>>> g = tf.Graph()
>>> with g.as_default():
...     a = tf.compat.v1.placeholder(shape=None,
...                                   dtype=tf.int32, name='tf_a')
...     b = tf.compat.v1.placeholder(shape=None,
...                                   dtype=tf.int32, name='tf_b')
...     c = tf.compat.v1.placeholder(shape=None,
...                                   dtype=tf.int32, name='tf_c')
...     z = 2*(a-b) + c
>>> with tf.compat.v1.Session(graph=g) as sess:
...     feed_dict={a:1, b:2, c:3}
...     print('Результат: z =', sess.run(z, feed_dict=feed_dict))
Результат: z = 1
```

## Загрузка входных данных в модель: стиль TensorFlow v2

В TensorFlow v2 все это можно сделать, просто определяя *обычную функцию Python* с входными аргументами  $a$ ,  $b$  и  $c$ , например:



```
## Стилль TF-v2
>>> def compute_z(a, b, c):
...     r1 = tf.subtract(a, b)
...     r2 = tf.multiply(2, r1)
...     z = tf.add(r2, c)
...     return z
```

Чтобы выполнить вычисление, мы можем вызывать функцию `compute_z` с объектами `Tensor` в качестве аргументов. Обратите внимание, что функции TensorFlow вроде `add`, `subtract` и `multiply` также позволяют предоставлять входные данные более высоких рангов в форме объекта `Tensor` библиотеки TensorFlow, массива NumPy или возможно других объектов Python, таких как списки и кортежи. В следующем примере кода мы предоставляем скалярные входные данные (ранг 0), а также входные данные ранга 1 и 2 в виде списков:

```
>>> tf.print('Скалярные входные данные:', compute_z(1, 2, 3))
Скалярные входные данные: 1
>>> tf.print('Входные данные ранга 1:', compute_z([1], [2], [3]))
Входные данные ранга 1: [1]
>>> tf.print('Входные данные ранга 2:', compute_z([[1]], [[2]], [[3]]))
Входные данные ранга 2: [[1]]
```

В текущем разделе вы узнали, что переход в TensorFlow v2 делает стиль программирования простым и эффективным за счет устранения шагов, связанных с явным созданием графа и сеанса. Теперь, когда вы видели, как соотносятся версии TensorFlow v1.x и TensorFlow v2, в оставшихся материалах книги мы сосредоточимся только на TensorFlow v2. Далее мы более подробно обсудим декорирование функций Python внутри графа, позволяющее ускорять вычисление.

## Увеличение вычислительной мощности с помощью декораторов функций

В предыдущем разделе вы видели, что мы можем с легкостью написать обычную функцию Python и задействовать операции TensorFlow. Однако вычисления в режиме энергичного выполнения (через динамические графы) не настолько эффективны, как выполнение статических графов в TensorFlow v1.x.

Таким образом, в TensorFlow v2 предлагается инструмент под названием AutoGraph, который способен автоматически трансформировать код Python в код графов TensorFlow для более быстрого выполнения. Вдобавок в TensorFlow предоставляется простой механизм для компиляции обычной функции Python в статический граф TensorFlow с тем, чтобы сделать вычисления более эффективными.

Чтобы посмотреть, как все работает на практике, мы возьмем предыдущую функцию `compute_z` и аннотируем ее для компиляции графа с применением декоратора `@tf.function`:

```
>>> @tf.function
... def compute_z(a, b, c):
...     r1 = tf.subtract(a, b)
...     r2 = tf.multiply(2, r1)
...     z = tf.add(r2, c)
...     return z
```

Обратите внимание, что декорированную функцию мы можем использовать и вызывать таким же способом, как прежде, но теперь TensorFlow будет создавать статический граф на основе входных аргументов. Язык Python поддерживает динамическую типизацию и полиморфизм, поэтому мы в состоянии определить функцию наподобие `def f(a, b): return a+b` и затем вызывать ее с применением входных данных целочисленного типа, типа с плавающей точкой, спискового или строкового типа (вспомните, что `a+b` является допустимой операцией для списков и строк). Несмотря на то что графы TensorFlow требуют статических типов и форм, `tf.function` поддерживает возможность динамической типизации. Например, давайте вызовем функцию `compute_z` со следующими входными данными:

```
>>> tf.print('Скалярные входные данные:', compute_z(1, 2, 3))
>>> tf.print('Входные данные ранга 1:', compute_z([1], [2], [3]))
>>> tf.print('Входные данные ранга 2:', compute_z([[1]], [[2]],
...         [[3]]))
```

Вывод будет таким же, как ранее. Здесь для конструирования графа на основе входных аргументов библиотека TensorFlow использует механизм трассировки. Для такого механизма трассировки TensorFlow генерирует кортеж ключей на базе входных сигнатур, указанных при вызове функции.

Вот какие ключи генерируются:

- для аргументов `tf.Tensor` ключ основан на их формах и типах данных;
- в случае типов Python, таких как списки, их `id()` применяется для генерации ключей кеша;
- для простых значений Python ключи кеша базируются на входных величинах.

После вызова декорированной функции TensorFlow проверит, не был ли граф с соответствующим ключом сгенерирован ранее. Если граф такого рода не существует, тогда TensorFlow сгенерирует новый граф и сохранит новый ключ. С другой стороны, если мы хотим ввести ограничения на способ вызова функции, то при ее определении можем указать входную сигнатуру посредством кортежа из объектов `tf.TensorSpec`. Скажем, давайте переопределим предыдущую функцию `compute_z` и укажем, что разрешены только тензоры ранга 1 типа `tf.int32`:

```
>>> @tf.function(input_signature=(tf.TensorSpec(shape=[None],
...                                           dtype=tf.int32),
...                               tf.TensorSpec(shape=[None],
...                                           dtype=tf.int32),
...                               tf.TensorSpec(shape=[None],
...                                           dtype=tf.int32)),)
... def compute_z(a, b, c):
...     r1 = tf.subtract(a, b)
...     r2 = tf.multiply(2, r1)
...     z = tf.add(r2, c)
...     return z
```

Теперь функцию можно вызывать с использованием тензоров ранга 1 (или списков, допускающих преобразование в тензоры ранга 1):

```
>>> tf.print('Входные данные ранга 1:', compute_z([1], [2], [3]))
>>> tf.print('Входные данные ранга 1:', compute_z([1, 2], [2, 4],
...     [3, 6]))
```

Тем не менее, вызов функции с применением тензоров ранга, отличающегося от 1, приведет к ошибке, т.к. ранг не будет совпадать с указанной входной сигнатурой:

```
>>> tf.print('Входные данные ранга 0:', compute_z(1, 2, 3)
### приведет к ошибке
>>> tf.print('Входные данные ранга 2:', compute_z([[1], [2]],
...                                              [[2], [4]],
...                                              [[3], [6]]))
### приведет к ошибке
```

В этом разделе вы узнали, как аннотировать обычную функцию Python, чтобы библиотека TensorFlow скомпилировала ее в виде графа для более быстрого выполнения. Далее мы рассмотрим переменные TensorFlow: способы их создания и использования.

## Объекты Variable библиотеки TensorFlow для хранения и обновления параметров модели

В главе 13 мы раскрыли объекты Tensor. В контексте TensorFlow объект Variable представляет собой особый объект Tensor, который позволяет сохранять и обновлять параметры моделей во время обучения. Объект Variable можно создать путем вызова класса `tf.Variable` с начальными значениями, указанными пользователем. В следующем коде мы генерируем объекты Variable типов `float32`, `int32`, `bool` и `string`:

```
>>> a = tf.Variable(initial_value=3.14, name='var_a')
>>> print(a)
<tf.Variable 'var_a:0' shape=() dtype=float32, numpy=3.14>

>>> b = tf.Variable(initial_value=[1, 2, 3], name='var_b')
>>> print(b)
<tf.Variable 'var_b:0' shape=(3,) dtype=int32, numpy=array([1, 2, 3],
dtype=int32)>

>>> c = tf.Variable(initial_value=[True, False], dtype=tf.bool)
>>> print(c)
<tf.Variable 'Variable:0' shape=(2,) dtype=bool,
numpy=array([ True, False])>

>>> d = tf.Variable(initial_value=['abc'], dtype=tf.string)
>>> print(d)
<tf.Variable 'Variable:0' shape=(1,) dtype=string,
numpy=array([b'abc'], dtype=object)>
```

Обратите внимание, что при создании объекта Variable мы всегда обязаны предоставлять начальные значения. Переменные имеют ат-

рибут по имени `trainable`, который по умолчанию установлен в `True`. Высокоуровневые API-интерфейсы вроде Keras будут применять упомянутый атрибут для управления обучаемыми и необучаемыми переменными. Вот как определить необучаемую переменную `Variable`:

```
>>> w = tf.Variable([1, 2, 3], trainable=False)
>>> print(w.trainable)
False
```

Значения объекта `Variable` можно эффективно модифицировать, выполняя операции, такие как `.assign()`, `.assign_add()` и связанные методы. Рассмотрим несколько примеров:

```
>>> print(w.assign([3, 1, 4], read_value=True))
<tf.Variable 'UnreadVariable' shape=(3,) dtype=int32,
  numpy=array([3, 1, 4], dtype=int32)>
>>> w.assign_add([2, -1, 2], read_value=False)
>>> print(w.value())
tf.Tensor([5 0 6], shape=(3,), dtype=int32)
```

Когда аргумент `read_value` установлен в `True` (что принято по умолчанию), эти операции будут автоматически возвращать новые значения после обновления текущих значений объекта `Variable`. Установка `read_value` в `False` подавляет автоматический возврат обновленного значения (но объект `Variable` по-прежнему будет обновляться на месте). Вызов `w.value()` будет возвращать значения в формате тензора. Имейте в виду, что изменять форму или тип объекта `Variable` во время присваивания нельзя.

Вспомните, что для нейросетевых моделей необходима инициализация их параметров случайными весами, чтобы разрушить симметрию во время обратного распространения, иначе многослойная нейронная сеть окажется не полезнее однослойной нейронной сети наподобие логистической регрессии. При создании объекта `Variable` из TensorFlow мы также можем использовать схему случайной инициализации. Библиотека TensorFlow способна генерировать случайные числа, основанные на разнообразных распределениях с помощью `tf.random` (см. [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/random](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/random)). В приведенном ниже примере мы взглянем на ряд стандартных методов инициализации, которые доступны в Keras (см. [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/initializers](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/initializers)).

Итак, давайте посмотрим, как можно создать объект `Variable` с инициализацией Глоро, которая является классической схемой случайной инициализации, предложенной Ксавье Глоро и Йошуа Бенджи. Для этого мы создадим операцию по имени `init` как объект класса `GlorotNormal`. Затем мы вызовем созданную операцию и предоставим желательную форму выходного тензора:

```
>>> tf.random.set_seed(1)
>>> init = tf.keras.initializers.GlorotNormal()
>>> tf.print(init(shape=(3,)))
[-0.722795904 1.01456821 0.251808226]
```

Теперь мы можем задействовать эту операцию для инициализации объекта `Variable` формы  $2 \times 3$ :

```
>>> v = tf.Variable(init(shape=(2, 3)))
>>> tf.print(v)
[[0.28982234 -0.782292783 -0.0453658961]
 [0.960991383 -0.120003454 0.708528221]]
```



На  
заметку!

### Инициализация Ксавье (или Глоро)

На ранней стадии развития ГО путем наблюдений выяснилось, что инициализация весов со случайным равномерным или случайным нормальным распределением часто могла приводить к низкой эффективности модели во время обучения.

В 2010 году Глоро и Бенджи исследовали влияние инициализации и предложили новаторскую более надежную схему инициализации, облегчающую обучение глубоких нейронных сетей. Основная идея инициализации Ксавье заключается в том, чтобы приблизительно уравновесить дисперсию градиентов между разными слоями. В противном случае во время обучения некоторые слои могут получить слишком много внимания, тогда как другие слои будут отставать.

Согласно научной статье Глоро и Бенджи, если мы хотим инициализировать веса значениями с равномерным распределением, то должны выбирать интервал равномерного распределения следующим образом:

$$\mathbf{W} \sim \text{Равномерное} \left( -\frac{\sqrt{6}}{\sqrt{n_{\text{вх}} + n_{\text{вых}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{вх}} + n_{\text{вых}}}} \right)$$

Здесь  $n_{ax}$  — количество входных нейронов, умноженных на веса, а  $n_{вых}$  — количество выходных нейронов, которые передают значения следующему слою. Для инициализации весов значениями с гауссовым (нормальным) распределением рекомендуется выбирать стандартное отклонение  $\sigma = \frac{\sqrt{2}}{\sqrt{n_{ax} + n_{вых}}}$ .

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{ax} + n_{вых}}}$$

Библиотека TensorFlow поддерживает инициализацию Ксавье с равномерным и нормальным распределением весов.

Дополнительную информацию о схеме инициализации Глоро и Бенджи, включая математический вывод и доказательство, можно найти в их исходной статье (“Understanding the difficulty of training deep feedforward neural networks” (Осмысление сложности обучения глубоких нейронных сетей прямого распространения), Ксавье Глоро и Йошуа Бенджи (2010 год)), которая свободно доступна по ссылке <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.

А теперь, чтобы поместить это в контекст более практичного сценария применения, мы посмотрим, как можно определить объект `Variable` внутри базового класса `tf.Module`. Мы определим две переменные — одну обучаемую и одну необучаемую:

```
>>> class MyModule(tf.Module):
...     def __init__(self):
...         init = tf.keras.initializers.GlorotNormal()
...         self.w1 = tf.Variable(init(shape=(2, 3)),
...                                trainable=True)
...         self.w2 = tf.Variable(init(shape=(1, 2)),
...                                trainable=False)
>>> m = MyModule()
>>> print('Все переменные модуля:', [v.shape for v in m.variables])
Все переменные модуля: [TensorShape([2, 3]), TensorShape([1, 2])]
>>> print('Обучаемая переменная:', [v.shape for v in
...                                   m.trainable_variables])
Обучаемая переменная: [TensorShape([2, 3])]
```

Как несложно заметить в примере кода, создание подкласса класса `tf.Module` дает нам прямой доступ ко всем переменным, определенным

в имеющемся объекте (экземпляре специального класса `MyModule`) через атрибут `.variables`.

В заключение давайте взглянем на использование переменных внутри функции, декорированной с помощью `tf.function`. Определяя объект `Variable` из TensorFlow внутри обычной (не декорированной) функции, мы могли бы ожидать, что новый объект `Variable` будет создаваться и инициализироваться каждый раз, когда функция вызывается. Однако `tf.function` будет стараться повторно использовать объект `Variable`, основываясь на трассировке и создании графа. Следовательно, TensorFlow не разрешает создание объекта `Variable` внутри декорированной функции, в результате чего показанный ниже код приведет к возникновению ошибки:

```
>>> @tf.function
... def f(x):
...     w = tf.Variable([1, 2, 3])
>>> f([1])
ValueError: tf.function-decorated function tried to create
variables on non-first call.
Ошибка значения: функция, декорированная tf.function, попыталась
создать переменные при не первом своем вызове.
```

Один из способов избежать проблемы предусматривает определение объекта `Variable` за пределами декорированной функции и его потребление внутри этой функции:

```
>>> w = tf.Variable(tf.random.uniform((3, 3)))
>>> @tf.function
... def compute_z(x):
...     return tf.matmul(w, x)
>>> x = tf.constant([[1], [2], [3]], dtype=tf.float32)
>>> tf.print(compute_z(x))
```

## Расчет градиентов посредством автоматического дифференцирования и `GradientTape`

Вы уже знаете, что оптимизация нейронных сетей требует расчета градиентов издержек относительно весов нейросетевых моделей, для чего нужны алгоритмы оптимизации, такие как *стохастический градиентный спуск* (SGD). Кроме того, градиенты имеют другие приложения наподобие диагностиро-



вания сети с целью выяснения, почему нейросетевая модель вырабатывает определенный прогноз для испытательного образца. Таким образом, в настоящем разделе мы покажем, как рассчитывать градиенты вычисления по отношению к ряду переменных.

## Расчет градиентов потери по отношению к обучаемым переменным

Библиотека TensorFlow поддерживает *автоматическое дифференцирование*, которое можно трактовать как реализацию *цепного правила* для расчета градиентов вложенных функций. Когда мы определяем последовательность операций, которая в результате дает какой-то выход или даже промежуточные тензоры, TensorFlow обеспечивает контекст для расчета градиентов таких вычисленных тензоров относительно их зависимых углов в вычислительном графе. Чтобы рассчитать эти градиенты, мы должны “зарегистрировать” вычисления посредством `tf.GradientTape`.

Давайте проработаем простой пример, в котором вычислим  $z = wx + b$  и определим потерю как квадратичную ошибку между целью и прогнозом,  $Потеря = (y - z)^2$ . В более общем случае, где мы можем иметь множество прогнозов и целей, мы вычисляем потерю как сумму квадратичных ошибок, т.е.  $Потеря = \sum_i (y_i - z_i)^2$ . Для реализации такого вычисления в TensorFlow мы определим параметры модели  $w$  и  $b$  как переменные, а вход  $x$  и  $y$  как тензоры. Вычисление  $z$  и потери мы поместим внутрь контекста `tf.GradientTape`:

```
>>> w = tf.Variable(1.0)
>>> b = tf.Variable(0.5)
>>> print(w.trainable, b.trainable)
True True

>>> x = tf.convert_to_tensor([1.4])
>>> y = tf.convert_to_tensor([2.1])
>>> with tf.GradientTape() as tape:
...     z = tf.add(tf.multiply(w, x), b)
...     loss = tf.reduce_sum(tf.square(y - z))

>>> dloss_dw = tape.gradient(loss, w)
>>> tf.print('dL/dw:', dloss_dw)
dL/dw: -0.559999764
```

При вычислении значения  $z$  мы могли бы думать об обязательных операциях, записываемых на “ленту градиентов”, как о прямом проходе в ней-

ронной сети. Мы применяем `tape.gradient` для вычисления  $\frac{\partial \text{Потеря}}{\partial w}$ . Поскольку пример очень прост, мы можем получить производные  $\frac{\partial \text{Потеря}}{\partial w} = 2x(wx + b - y)$  в символьной форме, чтобы убедиться в том, что рассчитанные градиенты совпадают с результатами, которые были получены в предыдущем фрагменте кода:

```
# проверка рассчитанного градиента
>>> tf.print(2*x*(w*x+b-y))
[-0.559999764]
```



### Понятие автоматического дифференцирования

Автоматическое дифференцирование представляет собой набор вычислительных приемов для расчета производных или градиентов произвольных арифметических операций. В течение этого процесса градиенты вычисления (выраженные как последовательность операций) получают за счет накопления градиентов посредством многократного применения цепного правила. Чтобы лучше понять концепцию, лежащую в основе автоматического дифференцирования, давайте обсудим последовательность вычислений  $y = f(g(h(x)))$  с входом  $x$  и выходом  $y$ . Ее можно разбить на ряд шагов:

- $u_0 = x$
- $u_1 = h(x)$
- $u_2 = g(u_1)$
- $u_3 = f(u_2) = y$

Производную  $\frac{\partial y}{\partial x}$  можно рассчитать двумя разными способами: прямое накопление, которое начинается с  $\frac{\partial u_3}{\partial x} = \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1}$ , и обратное накопление, начинающееся с  $\frac{\partial y}{\partial u_0} = \frac{\partial y}{\partial u_1} \frac{\partial u_1}{\partial u_0}$ . Обратите внимание, что TensorFlow использует обратное накопление.

## Расчет градиентов по отношению к необучаемым тензорам

Класс `tf.GradientTape` автоматически поддерживает градиенты для обучаемых переменных. Тем не менее, для отслеживания необучаемых переменных и других объектов `Tensor` нам необходимо добавить к

GradientTape дополнительную модификацию вида `tape.watch()`. Скажем, если нас интересует расчет  $\frac{\partial \text{Потеря}}{\partial x}$ , то код будет выглядеть следующим образом:

```
>>> with tf.GradientTape() as tape:
...     tape.watch(x)
...     z = tf.add(tf.multiply(w, x), b)
...     loss = tf.reduce_sum(tf.square(y - z))
>>> dloss_dx = tape.gradient(loss, x)
>>> tf.print('dL/dx:', dloss_dx)
dL/dx: [-0.399999857]
```



Совет

### Состязательные образцы

Расчет градиентов потери по отношению к входному образцу применяется для генерирования *состязательных образцов* (или *состязательных атак* (*adversarial attack*)). В компьютерном зрении состязательные образцы — это такие образцы, которые генерируются путем добавления к входному образцу небольшого и незначительного шума (или возмущений), что в результате приводит к его неправильной классификации нейронной сетью. Раскрытие состязательных образцов выходит за рамки тематики настоящей книги, но если вам интересно, тогда можете ознакомиться с первоначальной работой Кристиана Сегеди и др. “Intriguing properties of neural networks” (Занимательные свойства нейронных сетей), свободно доступной по ссылке <https://arxiv.org/pdf/1312.6199.pdf>.

## Сохранение ресурсов для множества вычислений градиентов

Когда мы отслеживаем вычисления в контексте `tf.GradientTape`, лента по умолчанию будет сохранять ресурсы только для одиночного расчета градиентов. Например, после однократного вызова `tape.gradient()` ресурсы будут освобождены и лента очистится. Следовательно, если мы хотим рассчитать более одного градиента, например,  $\frac{\partial \text{Потеря}}{\partial w}$  и  $\frac{\partial \text{Потеря}}{\partial b}$ , тогда понадобится сделать ленту постоянной:

```
>>> with tf.GradientTape(persistent=True) as tape:
...     z = tf.add(tf.multiply(w, x), b)
...     loss = tf.reduce_sum(tf.square(y - z))
```

```
>>> dloss_dw = tape.gradient(loss, w)
>>> tf.print('dL/dw:', dloss_dw)
dL/dw: -0.559999764

>>> dloss_db = tape.gradient(loss, b)
>>> tf.print('dL/db:', dloss_db)
dL/db: -0.399999857
```

Однако имейте в виду, что это необходимо только при желании рассчитывать более одного градиента, т.к. регистрация и хранение ленты градиентов менее эффективна в плане расхода памяти, чем освобождение памяти после расчета одиночного градиента. Именно потому стандартной настройкой является `persistent=False`.

Наконец, если мы вычисляем градиенты члена потери относительно параметров модели, то можем определить оптимизатор и применить градиенты для оптимизации параметров модели с использованием API-интерфейса `tf.keras`:

```
>>> optimizer = tf.keras.optimizers.SGD()
>>> optimizer.apply_gradients(zip([dloss_dw, dloss_db], [w, b]))
>>> tf.print('Обновленный вес:', w)
Обновленный вес: 1.0056

>>> tf.print('Обновленное смещение:', b)
Обновленное смещение: 0.504
```

Как вы должны вспомнить, первоначальный вес и смещение составляли  $w = 1.0$  и  $b = 0.5$ , а применение градиентов потери относительно параметров модели изменило их на  $w = 1.0056$  и  $b = 0.504$ .

## Упрощение реализаций распространенных архитектур посредством API-интерфейса Keras

Вы уже встречали несколько примеров построения нейросетевой модели прямого распространения (скажем, многослойного персептрона) и определения последовательности слоев с использованием класса `Sequential` библиотеки Keras. Прежде чем рассматривать различные подходы к конфигурированию таких слоев, давайте кратко повторим базовые шаги, построив модель с двумя плотными (полносвязными) слоями:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(units=16, activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=32, activation='relu'))
>>> ## позднее создание переменных
>>> model.build(input_shape=(None, 4))
>>> model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	multiple	80
dense_1 (Dense)	multiple	544

Total params: 624

Trainable params: 624

Non-trainable params: 0

Модель: "sequential"

Слой (тип)	Форма выхода	Кол-во параметров
dense (Dense)	множественная	80
dense_1 (Dense)	множественная	544

Всего параметров: 624

Обучаемых параметров: 624

Необучаемых параметров: 0

В вызове `model.build()` мы указали форму входа, создавая переменные после определения модели для этой конкретной формы, а также отобразили количество параметров каждого слоя:  $16 \times 4 + 16 = 80$  для первого слоя и  $16 \times 32 + 32 = 544$  для второго слоя. После того как переменные (или параметры модели) созданы, мы можем получать доступ к обучаемым и необучаемым переменным:

```
>>> ## вывод переменных модели
>>> for v in model.variables:
...     print('{:20s}'.format(v.name), v.trainable, v.shape)
```

```
dense/kernel:0      True (4, 16)
dense/bias:0        True (16,)
dense_1/kernel:0    True (16, 32)
dense_1/bias:0      True (32,)
```

В нашем случае каждый слой имеет весовую матрицу по имени `kernel` и вектор смещений. Далее мы сконфигурируем созданные слои, например, за счет применения к параметрам разных функций активации, инициализаторов переменных или методов регуляризации. Полный список доступных вариантов для указанных категорий можно найти в официальной документации:

- выбор функций активации посредством `tf.keras.activations` — [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/activations](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/activations);
- инициализация параметров слоя посредством `tf.keras.initializers` — [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/initializers](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/initializers);
- применение регуляризации к параметрам слоя (во избежание переобучения) посредством `tf.keras.regularizers` — [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/regularizers](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/regularizers).

В показанном ниже примере кода мы конфигурируем первый слой, указывая инициализаторы для переменных ядра и смещения. Затем мы конфигурируем второй слой, указывая регуляризатор L1 для ядра (весовой матрицы):

```
>>> model = tf.keras.Sequential()
>>> model.add(
...     tf.keras.layers.Dense(
...         units=16,
...         activation=tf.keras.activations.relu,
...         kernel_initializer= \
...             tf.keras.initializers.glorot_uniform(),
...         bias_initializer=tf.keras.initializers.Constant(2.0)
...     )
>>> model.add(
...     tf.keras.layers.Dense(
...         units=32,
...         activation=tf.keras.activations.sigmoid,
...         kernel_regularizer=tf.keras.regularizers.l1
...     )
... )
```

Кроме того, в дополнение к конфигурированию индивидуальных слоев мы также можем сконфигурировать модель во время ее компиляции. Мы можем указать тип оптимизатора и функцию потерь для обучения, равно как и метрики для использования при составлении отчетности об эффективности на обучающем, проверочном и испытательном наборах данных. Исчерпывающий список всех доступных вариантов приведен в официальной документации:

- оптимизаторы посредством `tf.keras.optimizers` — [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/optimizers](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/optimizers);
- функции потерь посредством `tf.keras.losses` — [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/losses](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/losses);
- метрики эффективности посредством `tf.keras.metrics` — [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/metrics](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/metrics).

**Совет**

### Выбор функции потерь

Что касается выбора алгоритма оптимизации, то самыми широко применяемыми методами являются SGD и Adam. Выбор функции потерь зависит от задачи; скажем, для задачи регрессии вы можете использовать потерю в форме среднеквадратической ошибки.

Семейство функций потери перекрестной энтропии предлагает возможные варианты для задач классификации, которые подробно обсуждались в главе 15.

Вдобавок вы можете применять методики, о которых узнали в предшествующих главах (например, методики оценки моделей из главы 6) в сочетании с надлежащими метриками для задачи. Скажем, для оценки классификационных моделей подходящими метриками будут точность и полнота, правильность, площадь под кривой (AUC), а также доли ложноотрицательных и ложноположительных классификаций.

В рассматриваемом примере мы скомпилируем модель с использованием оптимизатора SGD, потери перекрестной энтропии для двоичной классификации и характерного списка метрик, включающего правильность, точность и полноту:

```
>>> model.compile(
...     optimizer=tf.keras.optimizers.SGD(learning_rate=0.001),
...     loss=tf.keras.losses.BinaryCrossentropy(),
...     metrics=[tf.keras.metrics.Accuracy(),
...               tf.keras.metrics.Precision(),
...               tf.keras.metrics.Recall(),])
```

При обучении этой модели с помощью вызова метода `model.fit(...)` будет возвращаться хронология потери и указанных метрик для оценки эффективности обучения и проверки (если применяется проверочный набор данных), которую можно использовать для диагностирования поведения во время обучения.

Далее мы выполним более практичный пример: решим задачу классификации XOR с применением API-интерфейса Keras. Сначала мы будем использовать класс `tf.keras.Sequential`, чтобы построить модель. Попутно вы узнаете о емкости модели в плане обработки нелинейных границ решений. Затем мы раскроем другие способы построения моделей, которые обеспечат нам большую гибкость и контроль над слоями сети.

## Решение задачи классификации XOR

Задача классификации XOR — это классическая задача анализа емкости модели в отношении захватывания нелинейной границы решений между двумя классами. Мы сгенерируем игрушечный набор данных из 200 обучающих экземпляров с двумя признаками ( $x_0$ ,  $x_1$ ), взятыми из равномерного распределения с диапазоном  $[-1, 1)$ . Далее мы назначаем достоверную метку для обучающего образца  $i$  согласно следующему правилу:

$$y^{(i)} = \begin{cases} 0, & \text{если } x_0^{(i)} \times x_1^{(i)} < 0 \\ 1 & \text{в противном случае} \end{cases}$$

Мы задействуем половину данных (100 обучающих образцов) для обучения и другую половину для проверки. Вот как выглядит код для генерирования данных и их расщепления на обучающий и проверочный наборы:



```

>>> import tensorflow as tf
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> tf.random.set_seed(1)
>>> np.random.seed(1)

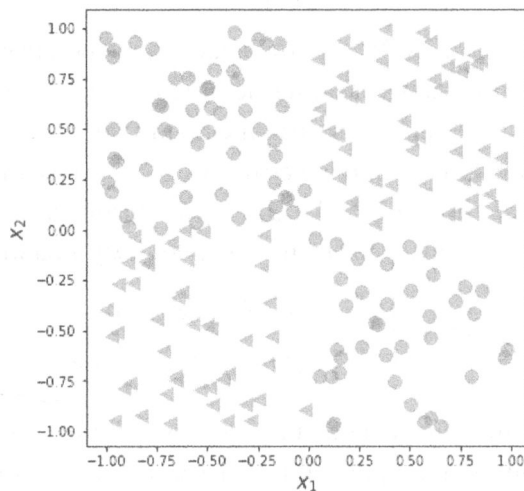
>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1] < 0] = 0

>>> x_train = x[:100, :]
>>> y_train = y[:100]
>>> x_valid = x[100:, :]
>>> y_valid = y[100:]

>>> fig = plt.figure(figsize=(6, 6))
>>> plt.plot(x[y==0, 0],
...         x[y==0, 1], 'o', alpha=0.75, markersize=10)
>>> plt.plot(x[y==1, 0],
...         x[y==1, 1], '<', alpha=0.75, markersize=10)
>>> plt.xlabel(r'$x_1$', size=15)
>>> plt.ylabel(r'$x_2$', size=15)
>>> plt.show()

```

Результатом выполнения кода будет представленный на рис. 14.2 график рассеяния обучающих и проверочных образцов, показанных с разными маркерами в зависимости от их меток классов.



**Рис. 14.2.** График рассеяния обучающих и проверочных образцов

В предыдущем подразделе мы обсудили важные инструменты, которые необходимы для реализации классификатора в TensorFlow. Теперь нам нужно решить, какую архитектуру мы должны выбрать для этой задачи и набора данных. Запомните в качестве эмпирического правила: чем больше имеется слоев и больше нейронов в каждом слое, тем более высокой будет емкость модели. Здесь емкость модели следует воспринимать как меру того, насколько легко модель может аппроксимировать сложные функции. В то время как наличие большего количества параметров означает, что сеть в состоянии подгоняться к более сложным функциям, крупные модели обычно труднее обучать (и они предрасположены к переобучению). На практике всегда полезно начинать с простой модели в качестве базовой линии, например, однослойной нейронной сети вроде логистической регрессии:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(units=1,
...                                   input_shape=(2,),
...                                   activation='sigmoid'))
>>> model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 1)	3
Total params: 3		
Trainable params: 3		
Non-trainable params: 0		

Общий размер параметров для такой простой логистической регрессионной модели составляет 3: весовая матрица (или ядро) размера  $2 \times 1$  и вектор смещений размера 1. После определения модели мы скомпилируем и обучим ее на протяжении 200 эпох с применением размера пакета 2:

```
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])
>>> hist = model.fit(x_train, y_train,
...                 validation_data=(x_valid, y_valid),
...                 epochs=200, batch_size=2, verbose=0)
```

Обратите внимание, что `model.fit()` возвращает хронологию эпох обучения, которая полезна для визуального контроля после обучения. В приведенном ниже коде мы вычертим кривые обучения, включая потерю при обучении и потерю при проверке, а также их правильности.

Мы также будем использовать библиотеку `MLxtend` для визуализации проверочных данных и границы решений.

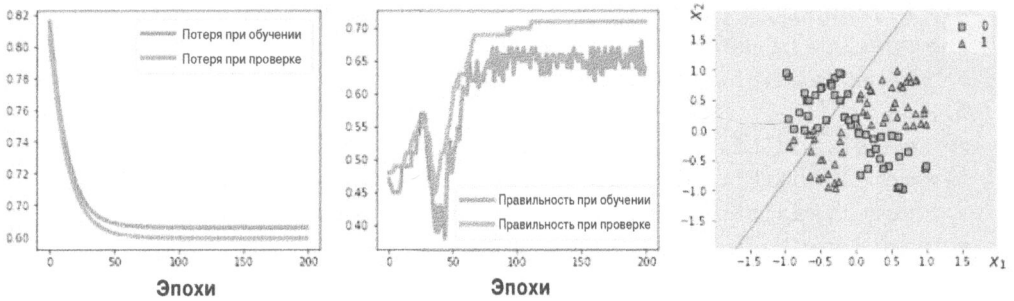
Библиотеку `MLxtend` можно установить посредством `conda` или `pip`:

```
conda install mlxtend -c conda-forge
pip install mlxtend
```

Следующий код вычерчивает график, отражающий эффективность при обучении и смещение области решений:

```
>>> from mlxtend.plotting import plot_decision_regions
>>> history = hist.history
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history['loss'], lw=4)
>>> plt.plot(history['val_loss'], lw=4)
>>> plt.legend(['Потеря при обучении', 'Потеря при проверке'],
...            fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history['binary_accuracy'], lw=4)
>>> plt.plot(history['val_binary_accuracy'], lw=4)
>>> plt.legend(['Правильность при обучении',
...            'Правильность при проверке'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid, y=y_valid.astype(np.integer),
...                      clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

Результат показан на рис. 14.3; он содержит три части — график потерь, график правильности и график рассеяния проверочных образцов вместе с границей решений.



*Рис. 14.3. Эффективность при обучении и смещение области решений*

Как видите, простая модель без скрытых слоев может получить только линейную границу решений, которая не способна решить задачу XOR. Как следствие, мы наблюдаем, что член потери для обучающего и проверочного наборов данных очень высок, а правильность классификации крайне низка.

Чтобы вывести нелинейную границу решений, мы можем добавить один и более скрытых слоев, связанных с помощью нелинейных функций активации. Теорема об универсальной аппроксимации утверждает, что нейронная сеть прямого распространения с единственным скрытым слоем и относительно большим количеством скрытых элементов способна относительно хорошо аппроксимировать произвольные непрерывные функции. Таким образом, один из подходов к получению более приемлемого решения задачи XOR предусматривает добавление скрытого слоя и сравнение различных количеств скрытых элементов до тех пор, пока мы не будем наблюдать удовлетворительные результаты на проверочном наборе данных. Добавление дополнительных скрытых элементов соответствует увеличению ширины слоя.

Альтернативно мы также можем добавить дополнительные скрытые слои, что сделает модель глубже. Преимущество создания более глубокой, а не более широкой нейронной сети заключается в том, что для достижения сопоставимой емкости модели требуется меньше параметров. Тем не менее, недостаток глубоких моделей (в сравнении с широкими моделями) связан с тем, что глубокие модели предрасположены к исчезновению и взрывному росту градиентов, из-за чего их обучение становится труднее.

В качестве упражнения попробуйте добавить один, два, три и четыре скрытых слоя, каждый с четырьмя скрытыми элементами. В следующем примере кода мы посмотрим на результаты обучения нейронной сети прямого распространения с тремя слоями:

```

>>> tf.random.set_seed(1)
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Dense(units=4, input_shape=(2,),
...                                 activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=4, activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=4, activation='relu'))
>>> model.add(tf.keras.layers.Dense(units=1,
activation='sigmoid'))

>>> model.summary()
Model: "sequential_4"

```

Layer (type)	Output Shape	Param #
dense_11 (Dense)	(None, 4)	12
dense_12 (Dense)	(None, 4)	20
dense_13 (Dense)	(None, 4)	20
dense_14 (Dense)	(None, 1)	5

```

Total params: 57
Trainable params: 57
Non-trainable params: 0

```

```

>>> ## КОМПИЛЯЦИЯ:
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])

>>> ## ОБУЧЕНИЕ:
>>> hist = model.fit(x_train, y_train,
...                 validation_data=(x_valid, y_valid),
...                 epochs=200, batch_size=2, verbose=0)

```

Повторив приведенный ранее код для визуализации, мы получим графики, представленные на рис. 14.4.

Теперь мы видим, что модель в состоянии вывести нелинейную границу решений для имеющихся данных, достигая 100%-ной правильности на обучающем наборе. Правильность на проверочном наборе составляет 95%, что говорит о небольшой степени переобучения модели.

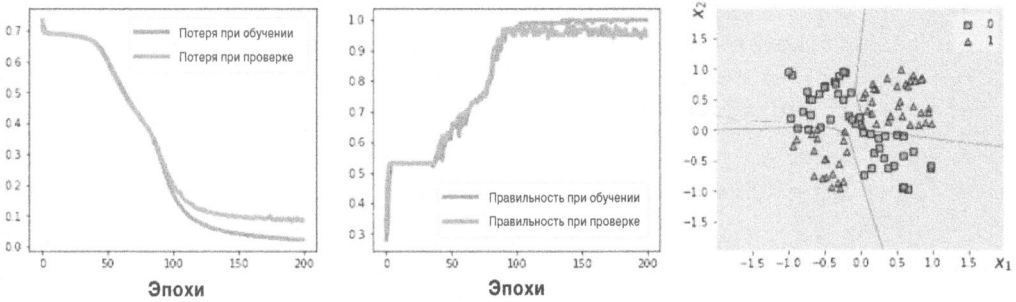


Рис. 14.4. Получение нелинейной границы решений

## Увеличение гибкости построения моделей с помощью функционального API-интерфейса Keras

В предыдущем примере для создания полносвязной нейронной сети с множеством слоев мы применяли класс `Sequential` библиотеки Keras. Он является очень распространенным и удобным способом построения моделей. Однако, к сожалению, такой прием не позволяет создавать более сложные модели, которые имеют множественный вход, выход или промежуточные ветви. Именно здесь пригодится так называемый функциональный API-интерфейс Keras.

Чтобы продемонстрировать, как можно использовать функциональный API-интерфейс, мы реализуем ту же самую архитектуру, которую строили с применением объектно-ориентированного (`Sequential`) подхода в предыдущем разделе; тем не менее, на этот раз мы будем использовать функциональный подход. При таком подходе мы сначала указываем вход. Затем конструируются скрытые слои с их выходами, именованными как `h1`, `h2` и `h3`. При решении данной задачи мы делаем выход каждого слоя входом в последующий слой (обратите внимание, что при построении более сложных моделей, которые имеют много промежуточных ветвей, ситуация может оказаться иной, но все по-прежнему делается посредством функционального API-интерфейса). Наконец, мы указываем выход как финальный плотный слой, который получает `h3` в качестве входа. Вот как выглядит код:

```
>>> tf.random.set_seed(1)
>>> ## входной слой:
>>> inputs = tf.keras.Input(shape=(2,))
```

```

>>> ## скрытые слои
>>> h1 = tf.keras.layers.Dense(units=4, activation='relu')(inputs)
>>> h2 = tf.keras.layers.Dense(units=4, activation='relu')(h1)
>>> h3 = tf.keras.layers.Dense(units=4, activation='relu')(h2)
>>> ## выход:
>>> outputs = tf.keras.layers.Dense(units=1, activation='sigmoid')
... (h3)
>>> ## конструирование модели:
>>> model = tf.keras.Model(inputs=inputs, outputs=outputs)
>>> model.summary()

```

Модель компилируется и обучается аналогично тому, как мы поступали ранее:

```

>>> ## компиляция:
>>> model.compile(
...     optimizer=tf.keras.optimizers.SGD(),
...     loss=tf.keras.losses.BinaryCrossentropy(),
...     metrics=[tf.keras.metrics.BinaryAccuracy()])
>>> ## обучение:
>>> hist = model.fit(
...     x_train, y_train,
...     validation_data=(x_valid, y_valid),
...     epochs=200, batch_size=2, verbose=0)

```

## Реализация моделей на основе класса `Model` библиотеки Keras

Альтернативный способ построения сложных моделей предусматривает создание подклассов класса `tf.keras.Model`. При таком подходе мы создаем новый класс, производный от `tf.keras.Model`, и определяем функцию `__init__()` как конструктор. Для определения обратного прохода применяется метод `call()`. В функции конструктора, `__init__()`, мы определяем слои как атрибуты класса, так что к ним можно обращаться через ссылочный атрибут `self`. Затем в методе `call()` мы указываем, каким образом эти слои должны использоваться в прямом проходе нейронной сети. Ниже приведен код для определения нового класса, который реализует только что описанную модель:

```
>>> class MyModel(tf.keras.Model):
...     def __init__(self):
...         super(MyModel, self).__init__()
...         self.hidden_1 = tf.keras.layers.Dense(
...             units=4, activation='relu')
...         self.hidden_2 = tf.keras.layers.Dense(
...             units=4, activation='relu')
...         self.hidden_3 = tf.keras.layers.Dense(
...             units=4, activation='relu')
...         self.output_layer = tf.keras.layers.Dense(
...             units=1, activation='sigmoid')
...
...     def call(self, inputs):
...         h = self.hidden_1(inputs)
...         h = self.hidden_2(h)
...         h = self.hidden_3(h)
...         return self.output_layer(h)
```

Обратите внимание, что для всех скрытых слоев мы применяем одно и то же имя `h`, тем самым делая код более читабельным и легким в отслеживании.

Класс модели, производный от `tf.keras.Model` с помощью создания подклассов, наследует основные атрибуты модели, такие как `build()`, `compile()` и `fit()`. Следовательно, после определения этого нового класса мы можем компилировать и обучать модель подобно любой другой модели, построенной посредством Keras:

```
>>> tf.random.set_seed(1)
>>> model = MyModel()
>>> model.build(input_shape=(None, 2))
>>> model.summary()
>>> ## КОМПИЛЯЦИЯ:
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])
>>> ## ОБУЧЕНИЕ:
>>> hist = model.fit(x_train, y_train,
...                 validation_data=(x_valid, y_valid),
...                 epochs=200, batch_size=2, verbose=0)
```



## Реализация специальных слоев Keras

В ситуациях, когда мы хотим создать новый слой, который еще не поддерживается библиотекой Keras, то можем определить новый класс, производный от класса `tf.keras.layers.Layer`. Поступать так особенно удобно при проектировании нового слоя или настройке существующего слоя.

Чтобы проиллюстрировать концепцию реализации специальных слоев, мы рассмотрим простой пример. Пусть нам нужно определить новый линейный слой, который вычисляет  $w(x + \varepsilon) + b$ , где  $\varepsilon$  ссылается на случайную переменную в качестве переменной шума. Для реализации такого вычисления мы определим новый класс как подкласс класса `tf.keras.layers.Layer`. В новом классе мы должны определить метод конструктора `__init__()` и метод `call()`. В конструкторе мы определим для нашего специального слоя необходимые переменные и другие обязательные тензоры. У нас есть возможность создавать переменные и инициализировать их в конструкторе, если ему предоставляется `input_shape`. Альтернативно мы можем отложить инициализацию переменных (скажем, если заранее не знаем точную форму входа) и делегировать ее методу `build()` с целью позднего создания переменных. Вдобавок мы можем определить метод `get_config()` для сериализации, т.е. модель, использующая наш специальный слой, будет способна эффективно сохраняться с применением средств сохранения и загрузки библиотеки TensorFlow.

Давайте обратимся к конкретному примеру и определим новый слой по имени `NoisyLinear`, который реализует упоминаемое выше вычисление  $w(x + \varepsilon) + b$ :

```
>>> class NoisyLinear(tf.keras.layers.Layer):
...     def __init__(self, output_dim, noise_stddev=0.1, **kwargs):
...         self.output_dim = output_dim
...         self.noise_stddev = noise_stddev
...         super(NoisyLinear, self).__init__(**kwargs)
...
...     def build(self, input_shape):
...         self.w = self.add_weight(name='weights',
...                                   shape=(input_shape[1],
...                                           self.output_dim),
...                                   initializer='random_normal',
...                                   trainable=True)
... 
```

```

...     self.b = self.add_weight(shape=(self.output_dim,),
...                               initializer='zeros',
...                               trainable=True)
...
...     def call(self, inputs, training=False):
...         if training:
...             batch = tf.shape(inputs)[0]
...             dim = tf.shape(inputs)[1]
...             noise = tf.random.normal(shape=(batch, dim),
...                                         mean=0.0,
...                                         stddev=self.noise_stddev)
...
...             noisy_inputs = tf.add(inputs, noise)
...         else:
...             noisy_inputs = inputs
...         z = tf.matmul(noisy_inputs, self.w) + self.b
...         return tf.keras.activations.relu(z)
...
...     def get_config(self):
...         config = super(NoisyLinear, self).get_config()
...         config.update({'output_dim': self.output_dim,
...                         'noise_stddev': self.noise_stddev})
...         return config

```

Мы добавили к конструктору аргумент `noise_stddev`, позволяющий указывать стандартное отклонение для распределения,  $\epsilon$ , которое выбирается из гауссова распределения. Кроме того, обратите внимание, что в методе `call()` мы используем дополнительный аргумент, `training=False`. В контексте Keras аргумент `training` является особым булевским аргументом, который проводит различие между случаями, когда модель или слой применяется во время обучения (скажем, через `fit()`) либо только для прогнозирования (например, через `predict()`; иногда это также называется “выведением” или оценкой). Одно из основных отличий между обучением и прогнозированием заключается в том, что во время прогнозирования нам не требуются градиенты. Вдобавок определенные методы ведут себя по-разному в режимах обучения и прогнозирования. В предстоящих главах вы встретите пример такого метода — Dropout. В предыдущем фрагменте кода мы также указали, что случайный вектор  $\epsilon$  должен генерироваться и добавляться к входу только во время обучения и не использоваться для вывода или оценки.

Прежде чем двигаться дальше и задействовать наш специальный слой `NoisyLinear` в модели, давайте протестируем его в рамках простого примера.

В показанном ниже коде мы определим новый экземпляр слоя `NoisyLinear`, инициализируем его вызовом `.build()` и запустим на входном тензоре. Затем мы сериализируем его с помощью `.get_config()` и восстановим сериализированный объект посредством `.from_config()`:

```
>>> tf.random.set_seed(1)
>>> noisy_layer = NoisyLinear(4)
>>> noisy_layer.build(input_shape=(None, 4))
>>> x = tf.zeros(shape=(1, 4))
>>> tf.print(noisy_layer(x, training=True))
[[0 0.00821428 0 0]]
>>> ## реконструкция из config:
>>> config = noisy_layer.get_config()
>>> new_layer = NoisyLinear.from_config(config)
>>> tf.print(new_layer(x, training=True))
[[0 0.0108502861 0 0]]
```

В предыдущем фрагменте кода мы вызывали слой два раза на том же самом входном тензоре. Однако обратите внимание, что выходы отличаются, поскольку слой `NoisyLinear` добавляет к входному тензору случайный шум.

А теперь давайте создадим новую модель, похожую на предшествующую модель для решения задачи классификации XOR. Как и ранее, мы будем применять класс `Sequential` из `Keras`, но на этот раз использовать в качестве первого скрытого слоя многослойного персептрона наш слой `NoisyLinear`. Вот необходимый код:

```
>>> tf.random.set_seed(1)
>>> model = tf.keras.Sequential([
...     NoisyLinear(4, noise_stddev=0.1),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=1, activation='sigmoid')])
>>> model.build(input_shape=(None, 2))
>>> model.summary()
```

```

>>> ## КОМПИЛЯЦИЯ:
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])
>>> ## обучение:
>>> hist = model.fit(x_train, y_train,
...                 validation_data=(x_valid, y_valid),
...                 epochs=200, batch_size=2,
...                 verbose=0)
>>> ## вычерчивание графиков:
>>> history = hist.history
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history['loss'], lw=4)
>>> plt.plot(history['val_loss'], lw=4)
>>> plt.legend(['Потеря при обучении', 'Потеря при проверке'],
...           fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history['binary_accuracy'], lw=4)
>>> plt.plot(history['val_binary_accuracy'], lw=4)
>>> plt.legend(['Правильность при обучении',
...           'Правильность при проверке'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid, y=y_valid.astype(np.integer),
...                     clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()

```

На рис. 14.5 представлены результирующие графики.

Наша цель заключалась в том, чтобы выяснить, каким образом определять новый специальный слой, созданный в форме подкласса класса `tf.keras.layers.Layer`, и применять его подобно любому другому стандартному слою Keras. Хотя в рассмотренном конкретном примере слой `NoisyLinear` не помог повысить эффективность, имейте в виду, что мы большей частью преследовали цель научиться реализовывать специальный слой с нуля.

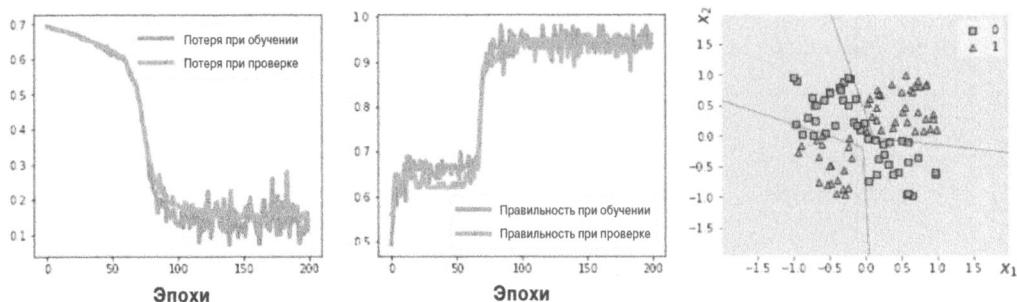


Рис. 14.5. Многослойный перцептрон со слоем NoisyLinear

В целом реализация нового специального слоя может быть полезной в других приложениях, скажем, при разработке нового алгоритма, который полагается на новый слой, расположенный поверх существующих слоев.

## Оценщики TensorFlow

До сих пор в главе мы были сосредоточены главным образом на низкоуровневом API-интерфейсе TensorFlow. Мы использовали декораторы для модификации функций, чтобы явно компилировать вычислительные графы с целью повышения эффективности расчетов. Затем мы работали с API-интерфейсом Keras и реализовывали нейронные сети прямого распространения, к которым добавляли специальные слои. В текущем разделе мы переключим внимание на оценщики TensorFlow. В API-интерфейсе `tf.estimator` инкапсулируются внутренние шаги задач МО, такие как обучение, прогнозирование (выведение) и оценка. Оценщики более инкапсулированы (т.е. защищены от внешнего вмешательства), но и более масштабируемы в сравнении с предшествующими подходами, раскрытыми в главе. К тому же API-интерфейс `tf.estimator` добавляет поддержку для прогона моделей на множестве платформ, не требуя внесения в код крупных изменений, что делает их более подходящими для так называемой “производственной стадии” в промышленных приложениях. Кроме того, библиотека TensorFlow поступает с подборкой готовых оценщиков для распространенных архитектур МО и ГО, которые удобны в сравнительных исследованиях, скажем, с целью быстрой оценки применимости определенного подхода к отдельному набору данных или задаче.

В оставшихся разделах главы вы узнаете, как использовать такие готовые оценщики и создавать оценщик из существующей модели Keras. Одним из важнейших элементов оценщиков является определение столбцов признаков как механизм для импортирования данных в модель, основанную на оценщиках, что мы раскроем в следующем разделе.

## Работа со столбцами признаков

В приложениях МО и ГО мы можем столкнуться с разнообразными типами признаков: непрерывными, неупорядоченными категориальными (именными) и упорядоченными категориальными (порядковыми). Вспомните, что в главе 4 мы рассматривали различные типы признаков и выясняли, каким образом обрабатывать каждый тип. Обратите внимание, что хотя числовые данные могут быть либо непрерывными, либо дискретными, в контексте API-интерфейса TensorFlow “числовые” данные конкретно имеют отношение к непрерывным данным типа с плавающей точкой.

Иногда наборы признаков состоят из смеси признаков разных типов. В то время как оценщики TensorFlow проектировались для поддержки всех типов признаков, мы должны указывать, как каждый признак будет интерпретироваться оценщиком. Например, рассмотрим сценарий с набором из семи признаков, изображенный на рис. 14.6.

Признаки, показанные на рис. 14.6 (год выпуска модели, количество цилиндров, рабочий объем, количество лошадиных сил, вес, разгон и происхождение), были получены из набора данных Auto MPG, который является общим эталонным набором данных МО для прогнозирования эффективности расхода топлива автомобилем в *милях на галлон (miles per gallon — MPG)*. Полный набор данных вместе с описанием доступен в хранилище UCI для МО по ссылке <https://archive.ics.uci.edu/ml/datasets/auto+mpg>.

Мы собираемся трактовать пять признаков из набора данных Auto MPG (количество цилиндров, рабочий объем, количество лошадиных сил, вес и разгон) как “числовые” (здесь непрерывные) признаки. Год выпуска модели можно рассматривать как упорядоченный категориальный (порядковый) признак. Наконец, происхождение можно считать неупорядоченным категориальным (именным) признаком с тремя возможными дискретными значениями, 1, 2 и 3, которые соответствуют США, Европе и Японии.

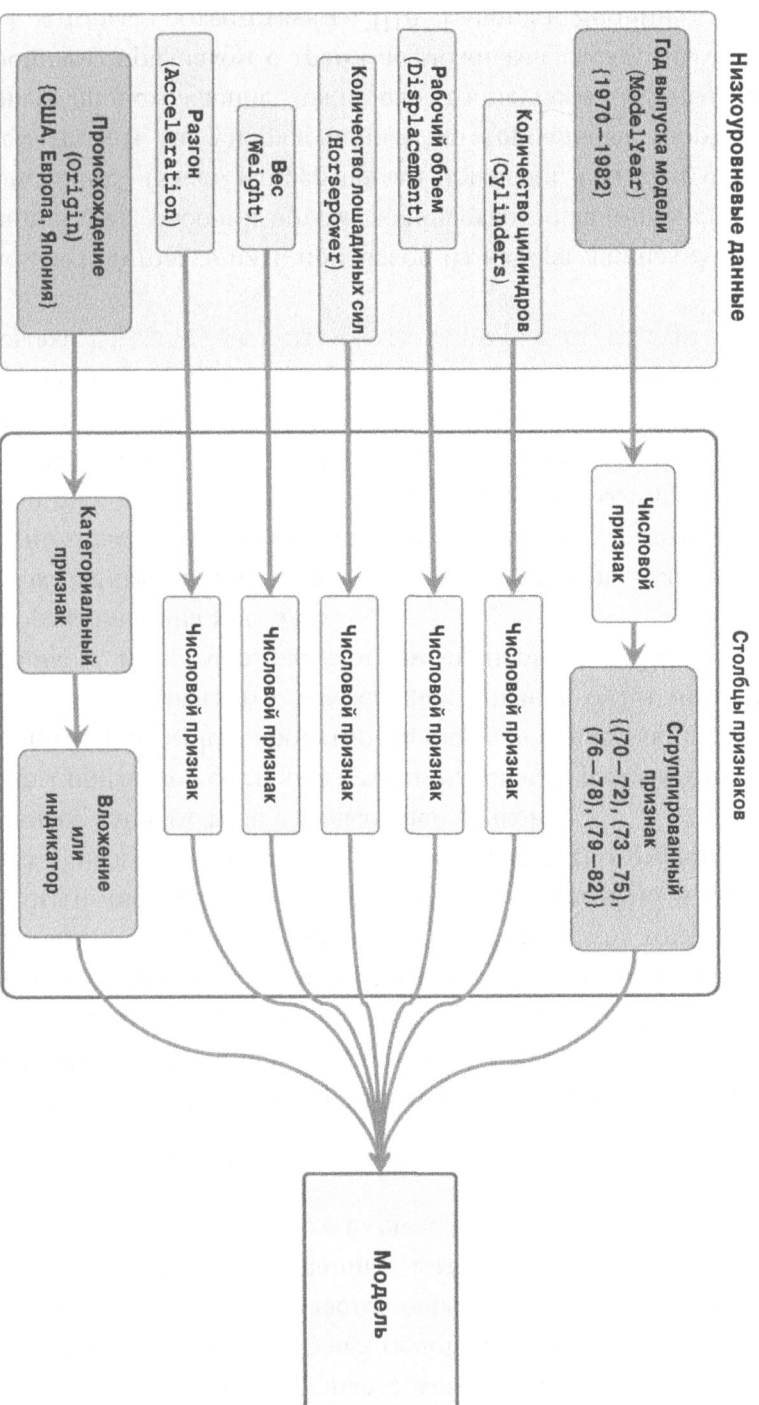


Рис. 14.6. Набор из семи признаков

Давайте сначала загрузим данные и применим необходимые шаги предварительной обработки, такие как расщепление набора данных на обучающий и испытательный наборы плюс стандартизация непрерывных признаков:

```
>>> import pandas as pd

>>> dataset_path = tf.keras.utils.get_file(
...     "auto-mpg.data",
...     ("http://archive.ics.uci.edu/ml/machine-learning"
...      "-databases/auto-mpg/auto-mpg.data"))

>>> column_names = [
...     'MPG', 'Cylinders', 'Displacement',
...     'Horsepower', 'Weight', 'Acceleration',
...     'ModelYear', 'Origin']

>>> df = pd.read_csv(dataset_path, names=column_names,
...                   na_values = '?', comment='\t',
...                   sep=' ', skipinitialspace=True)
>>> ## отбросить строки с отсутствующими (NA) значениями
>>> df = df.dropna()
>>> df = df.reset_index(drop=True)

>>> ## расщепление на обучающий и испытательный наборы:
>>> import sklearn
>>> import sklearn.model_selection

>>> df_train, df_test = sklearn.model_selection.train_test_split(
...     df, train_size=0.8)
>>> train_stats = df_train.describe().transpose()

>>> numeric_column_names = [
...     'Cylinders', 'Displacement',
...     'Horsepower', 'Weight',
...     'Acceleration']

>>> df_train_norm, df_test_norm = df_train.copy(), df_test.copy()

>>> for col_name in numeric_column_names:
...     mean = train_stats.loc[col_name, 'mean']
...     std = train_stats.loc[col_name, 'std']
...     df_train_norm.loc[:, col_name] = (
...         df_train_norm.loc[:, col_name] - mean)/std
...     df_test_norm.loc[:, col_name] = (
...         df_test_norm.loc[:, col_name] - mean)/std

>>> df_train_norm.tail()
```



Результат представлен на рис. 14.7.

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	ModelYear	Origin
<b>203</b>	28.0	-0.824303	-0.901020	-0.736562	-0.950031	0.255202	76	3
<b>255</b>	19.4	0.351127	0.413800	-0.340982	0.293190	0.548737	78	1
<b>72</b>	13.0	1.526556	1.144256	0.713897	1.339617	-0.625403	72	1
<b>235</b>	30.5	-0.824303	-0.891280	-1.053025	-1.072585	0.475353	77	1
<b>37</b>	14.0	1.526556	1.563051	1.636916	1.470420	-1.359240	71	1

*Рис. 14.7. Результирующий кадр данных*

Созданный предыдущим фрагментом кода объект `DataFrame` из `pandas` содержит пять столбцов со значениями типа `float`. Эти столбцы будут образовывать непрерывные признаки. В следующем коде мы будем использовать функцию `numeric_column` из модуля `feature_column` библиотеки TensorFlow для трансформации непрерывных признаков в структуру данных под названием столбец признаков, с которой могут работать оценщики TensorFlow:

```
>>> numeric_features = []
>>> for col_name in numeric_column_names:
...     numeric_features.append(
...         tf.feature_column.numeric_column(key=col_name))
```

Далее мы сгруппируем довольно мелкозернистую информацию о годах выпуска в участки, чтобы упростить задачу обучения модели. Выражаясь более конкретно, мы назначим каждый автомобиль одному из четырех участков “годов”:

$$\text{участок} = \begin{cases} 0, & \text{если год} < 73 \\ 1, & \text{если } 73 \leq \text{год} < 76 \\ 2, & \text{если } 76 \leq \text{год} < 79 \\ 3, & \text{если год} \geq 79 \end{cases}$$

Обратите внимание, что интервалы были выбраны произвольным образом с целью иллюстрации концепций “группирования в участки”. Чтобы сгруппировать автомобили в такие участки, мы сначала определяем числовой признак на основе каждого исходного года выпуска. Затем эти числовые признаки передаются функции `bucketized_column` вместе с тремя

граничными значениями интервалов: [73, 76, 79]. Указанные значения представляют собой значения отсечения справа и служат для определения полузамкнутых интервалов, например,  $(-\infty, 73)$ ,  $[73, 76)$ ,  $[76, 79)$  и  $[79, \infty)$ . Ниже приведен код:

```
>>> feature_year = tf.feature_column.numeric_column(key='ModelYear')
>>> bucketized_features = []
>>> bucketized_features.append(
...     tf.feature_column.bucketized_column(
...         source_column=feature_year,
...         boundaries=[73, 76, 79]))
```

Ради согласованности мы добавили такой сгруппированный признак в список Python, хотя сам список состоит только из одного элемента. В последующих шагах мы объединим этот список со списками, созданными из других признаков, и передадим результат в качестве входа модели, основанной на оценщиках TensorFlow.

Далее мы продолжим определением списка для неупорядоченного категориального признака `Origin`. В TensorFlow предусмотрены разные способы создания столбцов категориальных признаков. Если данные содержат названия категорий (скажем, в строковом формате вроде “US” (США), “Europe” (Европа) и “Japan” (Япония)), тогда мы можем задействовать функцию `tf.feature_column.categorical_column_with_vocabulary_list`, передав ей список возможных уникальных имен категорий. Если список возможных категорий слишком велик, как бывает, например, в типичном контексте анализа текста, тогда взамен мы можем использовать функцию `tf.feature_column.categorical_column_with_vocabulary_file`. При вызове указанной функции мы просто предоставляем файл, который содержит все категории/слова, так что нам не придется хранить список всех возможных слов в памяти. Кроме того, если признаки уже ассоциированы с индексами категорий из диапазона  $[0, \text{num\_categories})$ , то мы можем применять функцию `tf.feature_column.categorical_column_with_identity`. Тем не менее, в таком случае признак `Origin` задается как целочисленные значения 1, 2, 3 (в противоположность 0, 1, 2), которые не соответствуют требованию категориальной индексации, т.к. ожидается, что индексы начинаются с 0.

В приведенном ниже фрагменте кода мы создаем словарный список:

```
>>> feature_origin =  
...   tf.feature_column.categorical_column_with_vocabulary_list(  
...     key='Origin',  
...     vocabulary_list=[1, 2, 3])
```

Определенные оценщики, такие как `DNNClassifier` и `DNNRegressor`, принимают только то, что называется “плотными столбцами”. По этой причине следующим шагом будет преобразование существующего столбца категориального признака в плотный столбец подобного рода, для чего существуют два способа: использование столбца вложений посредством `embedding_column` или столбца индикаторов через `indicator_column`. Столбец индикаторов преобразует категориальные индексы в векторы в унитарном коде, т.е. индекс 0 будет закодирован как `[1, 0, 0]`, индекс 1 — как `[0, 1, 0]` и т.д. С другой стороны, столбец вложений отображает каждый индекс на вектор случайных чисел типа `float`, который может быть обучен.

Когда количество категорий велико, применение столбца вложений с числом измерений, меньшим количества категорий, может улучшить производительность. В следующем фрагменте кода мы будем использовать подход со столбцом индикаторов для категориального признака, преобразуя его в плотный столбец:

```
>>> categorical_indicator_features = []  
>>> categorical_indicator_features.append(  
...   tf.feature_column.indicator_column(feature_origin))
```

В этом разделе мы рассмотрели самые распространенные подходы к созданию столбцов признаков, которые могут применяться с оценщиками TensorFlow. Однако существует несколько дополнительных столбцов признаков, не обсуждаемых здесь, в том числе хешированные и перекрестные столбцы. Исчерпывающие сведения обо всех столбцах признаков доступны в официальной документации TensorFlow по ссылке [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/feature\\_column](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/feature_column).

## Машинное обучение с использованием готовых оценщиков

После конструирования обязательных столбцов признаков мы можем, наконец, задействовать оценщики TensorFlow. Работа с готовыми оценщиками может быть подытожена в виде четырех шагов.

1. Определить входную функцию для загрузки данных.
2. Преобразовать набор данных в столбцы признаков.
3. Создать объект оценщика (применяя готовый оценщик или создавая новый путем преобразования модели Keras в оценщик).
4. Использовать методы оценщика `train()`, `evaluate()` и `predict()`.

Продолжая пример с набором данных Auto MPG из предыдущего раздела, мы применим описанные четыре шага с целью иллюстрации практического использования оценщиков. Для первого шага нам необходимо определить функцию, которая обработает данные и возвратит набор данных TensorFlow, состоящий из кортежа с входными признаками и метками (достоверные значения MPG). Обратите внимание, что признаки должны находиться в формате словаря, ключи которого обязаны соответствовать именам столбцов признаков.

Начав с первого шага, мы определим входную функцию для обучающих данных следующим образом:

```
>>> def train_input_fn(df_train, batch_size=8):
...     df = df_train.copy()
...     train_x, train_y = df, df.pop('MPG')
...     dataset = tf.data.Dataset.from_tensor_slices(
...         (dict(train_x), train_y))
...
...     # тасование, повторение и разбиение на пакеты образцов.
...     return dataset.shuffle(1000).repeat().batch(batch_size)
```

Как видите, в этой функции мы применяем `dict(train_x)` для преобразования объекта DataFrame библиотеки pandas в словарь Python. Давайте загрузим пакет из этого набора данных, чтобы посмотреть, на что он похож:

```
>>> ds = train_input_fn(df_train_norm)
>>> batch = next(iter(ds))
>>> print('Ключи:', batch[0].keys())
Ключи: dict_keys(['Cylinders', 'Displacement', 'Horsepower',
'Weight', 'Acceleration', 'ModelYear', 'Origin'])
>>> print('Пакет с годами выпуска:', batch[0]['ModelYear'])
Пакет с годами выпуска: tf.Tensor([74 71 81 72 82 81 70 74],
shape=(8,), dtype=int32)
```

Также нам понадобится определить входную функцию для испытательного набора данных, который будет использоваться при оценке модели после обучения:

```
>>> def eval_input_fn(df_test, batch_size=8):  
...     df = df_test.copy()  
...     test_x, test_y = df, df.pop('MPG')  
...     dataset = tf.data.Dataset.from_tensor_slices(  
...         (dict(test_x), test_y))  
...     return dataset.batch(batch_size)
```

С переходом ко второму шагу нам нужно определить столбцы признаков. Мы уже определили список, содержащий непрерывные признаки, список для столбца сгруппированного признака и список для столбца категориального признака. Теперь мы можем объединить упомянутые индивидуальные списки в единственный список, содержащий все столбцы признаков:

```
>>> all_feature_columns = (  
...     numeric_features +  
...     bucketized_features +  
...     categorical_indicator_features)
```

Для третьего шага нам необходимо создать новый объект оценщика. Поскольку прогнозирование значений MPG является типичной задачей регрессии, мы будем применять класс `tf.estimator.DNNRegressor`. При создании объекта регрессионного оценщика мы предоставим список столбцов признаков и укажем количество скрытых элементов, которые желаем иметь в каждом скрытом слое, используя аргумент `hidden_units`. В примере мы будем применять два скрытых слоя, первый из которых имеет 32 элемента, а второй — 10 элементов:

```
>>> regressor = tf.estimator.DNNRegressor(  
...     feature_columns=all_feature_columns,  
...     hidden_units=[32, 10],  
...     model_dir='models/autompg-dnnregressor/')
```

Дополнительно предоставленный аргумент `model_dir` указывает каталог для сохранения параметров модели. Одно из преимуществ оценщиков заключается в том, что во время обучения они автоматически сохраняют контрольные точки модели, поэтому в случае аварийного отказа процесса обучения по непредвиденной причине (вроде неисправности сети питания)

мы легко можем загрузить последнюю сохраненную контрольную точку и продолжить с нее обучение. Контрольные точки также будут сохраняться в каталоге, указанном посредством `model_dir`. Если мы опускаем аргумент `model_dir`, то оценщик создаст случайный временный каталог (например, в среде Linux будет создан случайный подкаталог в каталоге `/tmp/`), предназначенный для такой цели.

После пройденных трех базовых шагов мы можем задействовать оценщик для обучения, оценки и в итоге прогнозирования. Регрессор можно обучить вызовом метода `train()`, для которого требовалась ранее определенная входная функция:

```
>>> EPOCHS = 1000
>>> BATCH_SIZE = 8
>>> total_steps = EPOCHS * int(np.ceil(len(df_train) / BATCH_SIZE))
>>> print('Шаги обучения:', total_steps)
Шаги обучения: 40000

>>> regressor.train(
...     input_fn=lambda:train_input_fn(
...         df_train_norm, batch_size=BATCH_SIZE),
...     steps=total_steps)
```

Вызов `.train()` будет автоматически сохранять контрольные точки во время обучения модели. Затем мы можем загрузить последнюю контрольную точку:

```
>>> reloaded_regressor = tf.estimator.DNNRegressor(
...     feature_columns=all_feature_columns,
...     hidden_units=[32, 10],
...     warm_start_from='models/autompg-dnnregressor/',
...     model_dir='models/autompg-dnnregressor/')
```

Далее для оценки прогнозирующей эффективности обученной модели мы можем использовать метод `evaluate()`:

```
>>> eval_results = reloaded_regressor.evaluate(
...     input_fn=lambda:eval_input_fn(df_test_norm, batch_size=8))
>>> print('Средняя потеря {:.4f}'.format(
...     eval_results['average_loss']))
Средняя потеря 15.1866
```

Наконец, для прогнозирования целевых значений на новых точках данных мы можем применять метод `predict()`. Для целей рассматриваемого примера мы предполагаем, что в реалистичном приложении испытательный набор представляет собой набор из новых непомеченных точек данных.

Обратите внимание, что в реальной задаче прогнозирования при условии недоступности меток входная функция должна будет возвращать только набор данных, состоящий из признаков. Для получения прогнозов по всем образцам мы просто используем ту же самую входную функцию, которая применялась при оценке:

```
>>> pred_res = regressor.predict(
...     input_fn=lambda: eval_input_fn(
...         df_test_norm, batch_size=8))
>>> print(next(iter(pred_res)))
{'predictions': array([23.747658], dtype=float32)}
```

Несмотря на то что предшествующие фрагменты кода завершили иллюстрацию четырех шагов, требующихся для использования готовых оценщиков, давайте ради практики рассмотрим еще один готовый оценщик: регрессор на основе деревьев с градиентным бустингом, `tf.estimator.BoostedTreeRegressor`. Так как входные функции и столбцы признаков уже созданы, нам понадобится лишь повторить шаги 3 и 4. Для шага 3 мы создадим экземпляр `BoostedTreeRegressor` и сконфигурируем его так, чтобы он имел 200 деревьев.



Совет

### Бустинг деревьев принятия решений

Ансамблевые алгоритмы, включая бустинг, были раскрыты в главе 7. Алгоритм на основе деревьев с градиентным бустингом представляет собой особое семейство алгоритмов бустинга, которые базируются на оптимизации произвольной функции потерь. Дополнительные сведения о градиентном бустинге доступны по ссылке <https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d>.

```
>>> boosted_tree = tf.estimator.BoostedTreesRegressor(
...     feature_columns=all_feature_columns,
...     n_batches_per_layer=20,
...     n_trees=200)
```

```
>>> boosted_tree.train(
...     input_fn=lambda:train_input_fn(
...         df_train_norm, batch_size=BATCH_SIZE))
>>> eval_results = boosted_tree.evaluate(
...     input_fn=lambda:eval_input_fn(
...         df_test_norm, batch_size=8))
>>> print('Средняя потеря {:.4f}'.format(
...     eval_results['average_loss']))
Средняя потеря 11.2609
```

Как видите, регрессор на основе деревьев с градиентным бустингом добился меньшей средней потери, чем DNNRegressor. Для небольших наборов данных подобного рода результат вполне ожидаем.

В этом разделе мы исследовали важные шаги по применению оценщиков TensorFlow для регрессии. В следующем подразделе мы рассмотрим типичный пример классификации с использованием оценщиков.

## Использование оценщиков для классификации рукописных цифр MNIST

Для решения такой задачи классификации мы собираемся применять оценщик DNNClassifier из библиотеки TensorFlow, который позволяет очень удобно реализовывать многослойный персептрон. В предыдущем разделе мы детально раскрыли четыре важных шага по использованию готовых оценщиков, а в этом разделе мы их повторим. Первым делом мы импортируем подмодуль `tensorflow_datasets` (`tfds`), который можно задействовать для загрузки набора данных MNIST и указания гиперпараметров модели.



Совет

### API-интерфейс оценщиков и проблема с графом

Поскольку в ряде частей TensorFlow 2.0 все еще присутствуют шероховатости, при выполнении приведенного далее кода вы можете столкнуться с ошибкой `RuntimeError: Graph is finalized and cannot be modified` (Ошибка времени выполнения: граф финализирован и не может быть изменен). На время написания книги хорошее решение проблемы отсутствовало, а обходной путь предусматривал перезапуск сеанса Python, IPython или Jupyter Notebook перед выполнением следующего блока кода.



Шаг настройки включает загрузку набора данных и указание гиперпараметров (`BUFFER_SIZE` для тасования набора данных, `BATCH_SIZE` для размера мини-пакета и `NUM_EPOCHS` для количества эпох обучения):

```
>>> import tensorflow_datasets as tfds
>>> import tensorflow as tf
>>> import numpy as np

>>> BUFFER_SIZE = 10000
>>> BATCH_SIZE = 64
>>> NUM_EPOCHS = 20
>>> steps_per_epoch = np.ceil(60000 / BATCH_SIZE)
```

Обратите внимание, что `steps_per_epoch` определяет количество итераций в каждой эпохе, что необходимо для бесконечно повторенных наборов данных (как обсуждалось в главе 13). Далее мы определим вспомогательную функцию, которая будет выполнять предварительную обработку входного изображения и его метки. Так как входное изображение изначально имеет тип `'uint8'` (в диапазоне `[0, 255]`), мы будем применять функцию `tf.image.convert_image_dtype` для преобразования его типа в `tf.float32` (и таким образом в диапазон `[0, 1]`):

```
>>> def preprocess(item):
...     image = item['image']
...     label = item['label']
...     image = tf.image.convert_image_dtype(
...         image, tf.float32)
...     image = tf.reshape(image, (-1,))
...
...     return {'image-pixels':image}, label[..., tf.newaxis]
```

**Шаг 1.** Определение двух входных функций (первой для обучения и второй для оценки):

```
>>> ## Шаг 1: определение двух входных функций
>>> def train_input_fn():
...     datasets = tfds.load(name='mnist')
...     mnist_train = datasets['train']
...
...     dataset = mnist_train.map(preprocess)
...     dataset = dataset.shuffle(BUFFER_SIZE)
...     dataset = dataset.batch(BATCH_SIZE)
...     return dataset.repeat()
```

```
>>> def eval_input_fn():
...     datasets = tfds.load(name='mnist')
...     mnist_test = datasets['test']
...     dataset = mnist_test.map(preprocess).batch(BATCH_SIZE)
...     return dataset
```

Отметим, что словарь признаков имеет только один ключ, 'image-pixels'. Мы будем использовать этот ключ в следующем шаге.

### Шаг 2. Определение столбцов признаков:

```
>>> ## Шаг 2: столбцы признаков
>>> image_feature_column = tf.feature_column.numeric_column(
...     key='image-pixels', shape=(28*28))
```

Обратите внимание, что мы определяем столбцы признаков размера 784 (т.е.  $28 \times 28$ ), который представляет собой размер входных изображений MNIST после того, как они разглажены.

### Шаг 3. Создание нового объекта оценщика. Здесь мы определяем два скрытых слоя: 32 элемента в первом и 16 элементов во втором.

Мы также указываем количество классов (вспомните, что набор данных MNIST состоит из изображений 10 разных цифр, 0–9) с применением аргумента `n_classes`:

```
>>> ## Шаг 3: создание объекта оценщика
>>> dnn_classifier = tf.estimator.DNNClassifier(
...     feature_columns=[image_feature_column],
...     hidden_units=[32, 16],
...     n_classes=10,
...     model_dir='models/mnist-dnn/')

```

### Шаг 4. Использование оценщика для обучения, оценки и прогнозирования:

```
>>> ## Шаг 4: обучение и оценка
>>> dnn_classifier.train(
...     input_fn=train_input_fn,
...     steps=NUM_EPOCHS * steps_per_epoch)
>>> eval_result = dnn_classifier.evaluate(
...     input_fn=eval_input_fn)
>>> print(eval_result)
{'accuracy': 0.8957, 'average_loss': 0.3876346,
 'loss': 0.38815108, 'global_step': 18760}
```

К настоящему моменту вы научились использовать готовые оценщики и применять их для предварительной оценки с целью выяснения, например, подходит ли существующая модель к имеющейся задаче. Помимо использования готовых оценщиков мы также можем создать оценщик, преобразовав в него модель Keras, чем и займемся в следующем подразделе.

## Создание специального оценщика из существующей модели Keras

Преобразование модели Keras в оценщик полезно как в научном сообществе, так и в производственной среде в ситуациях, когда вы разработали модель и хотите запустить ее в обращение или поделиться с остальными сотрудниками внутри вашей организации. Такое преобразование позволяет нам получить доступ к сильным сторонам оценщиков, среди которых распределенное обучение и автоматическое сохранение контрольных точек. Вдобавок другим станет легче использовать эту модель, в частности, будет устранена путаница при интерпретировании входных признаков за счет указания столбцов признаков и входной функции.

Чтобы выяснить, как можно создать собственный оценщик из модели Keras, мы займемся предыдущей задачей классификации XOR. Для начала мы восстановим данные и расщепим их на обучающий и проверочный наборы:

```
>>> tf.random.set_seed(1)
>>> np.random.seed(1)

>>> ## Создание данных
>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1] < 0] = 0

>>> x_train = x[:100, :]
>>> y_train = y[:100]
>>> x_valid = x[100:, :]
>>> y_valid = y[100:]
```

Давайте теперь построим модель Keras, которая позже будет преобразована в оценщик. Как и ранее, мы определим модель с применением класса `Sequential`. На этот раз мы также добавим входной слой, определенный как `tf.keras.layers.Input`, чтобы назначить имя входу данной модели:

```
>>> model = tf.keras.Sequential([
...     tf.keras.layers.Input(shape=(2,), name='input-features'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(units=4, activation='relu'),
...     tf.keras.layers.Dense(1, activation='sigmoid')
... ])
```

Далее мы пройдем через четыре шага, описанные в предыдущем подразделе. Шаги 1, 2 и 4 будут такими же, как те, что мы использовали с готовыми оценщиками. Важно отметить, что ключевое имя для входных признаков, применяемое при выполнении шагов 1 и 2, обязательно должно совпадать с именем, которое мы определили во входном слое модели. Код выглядит следующим образом:

```
>>> ## Шаг 1: определение входных функций
>>> def train_input_fn(x_train, y_train, batch_size=8):
...     dataset = tf.data.Dataset.from_tensor_slices(
...         ({'input-features':x_train}, y_train.reshape(-1, 1)))
...     # тасование, повторение и разбиение на пакеты образцов.
...     return dataset.shuffle(100).repeat().batch(batch_size)

>>> def eval_input_fn(x_test, y_test=None, batch_size=8):
...     if y_test is None:
...         dataset = tf.data.Dataset.from_tensor_slices(
...             {'input-features':x_test})
...     else:
...         dataset = tf.data.Dataset.from_tensor_slices(
...             ({'input-features':x_test}, y_test.reshape(-1, 1)))
...     # тасование, повторение и разбиение на пакеты образцов.
...     return dataset.batch(batch_size)

>>> ## Шаг 2: определение столбцов признаков
>>> features = [
...     tf.feature_column.numeric_column(
...         key='input-features:', shape=(2,))
... ]
```

На шаге 3 вместо того, чтобы создать объект одного из готовых оценщиков, мы преобразуем модель в оценщик с использованием функции `tf.keras.estimator.model_to_estimator`. Перед преобразованием модель необходимо скомпилировать:

```
>>> model.compile(optimizer=tf.keras.optimizers.SGD(),
...               loss=tf.keras.losses.BinaryCrossentropy(),
...               metrics=[tf.keras.metrics.BinaryAccuracy()])
>>> my_estimator = tf.keras.estimator.model_to_estimator(
...     keras_model=model,
...     model_dir='models/estimator-for-XOR/')

```

В заключение на шаге 4 мы можем обучить модель с применением нашего оценщика и оценить ее на проверочном наборе данных:

```
>>> ## Шаг 4: использование оценщика
>>> num_epochs = 200
>>> batch_size = 2
>>> steps_per_epoch = np.ceil(len(x_train) / batch_size)
>>> my_estimator.train(
...     input_fn=lambda: train_input_fn(x_train, y_train, batch_size),
...     steps=num_epochs * steps_per_epoch)
>>> my_estimator.evaluate(
...     input_fn=lambda: eval_input_fn(x_valid, y_valid, batch_size))
{'binary_accuracy': 0.96, 'loss': 0.081909806, 'global_step': 10000}

```

Как видите, преобразовывать модель Keras в оценщик очень легко. Это позволяет нам легко воспользоваться сильными сторонами оценщиков, такими как распределенное обучение и автоматическое сохранение контрольных точек во время обучения.

## Резюме

В главе мы раскрыли наиболее важные и полезные возможности библиотеки TensorFlow. Мы начали с обсуждения перехода от версии TensorFlow v1.x к версии TensorFlow v2. В частности, мы применяли подход с динамическими вычислительными графами TensorFlow (так называемый режим энергичного выполнения), который делает реализацию вычислений более удобной в сравнении с использованием статических графов. Мы также исследовали семантику определения объектов `Variable` из TensorFlow как параметров модели, аннотируя функции Python с помощью декоратора `tf.function` для повышения вычислительной эффективности посредством компиляции графа.

После рассмотрения концепции расчета частных производных и градиентов произвольных функций мы более подробно раскрыли API-интерфейс Keras. Он снабжает нас дружественным к пользователю интерфейсом для построения более сложных глубоких нейросетевых моделей. Наконец, мы задействовали API-интерфейс `tf.estimator` библиотеки TensorFlow, чтобы предоставить согласованный интерфейс, которому обычно отдают предпочтение в производственной среде. В заключение мы выяснили, как преобразовывать модель Keras в специальный оценщик.

После раскрытия главных механизмов библиотеки TensorFlow в следующей главе мы представим концепцию, лежащую в основе архитектур *сверточных нейронных сетей* (*convolutional neural network* — *CNN*) для ГО. Сети CNN являются мощными моделями, которые характеризуются высокой эффективностью в области компьютерного зрения.



# КЛАССИФИКАЦИЯ ИЗОБРАЖЕНИЙ С ПОМОЩЬЮ ГЛУБОКИХ СВЕРТОЧНЫХ НЕЙРОННЫХ СЕТЕЙ

В предыдущей главе мы подробно рассматривали разнообразные аспекты API-интерфейса TensorFlow. Вы ознакомились с тензорами и декорированными функциями, а также научились работать с оценщиками TensorFlow. В текущей главе вы узнаете о *сверточных нейронных сетях* (*convolutional neural network* — *CNN*) для классификации изображений. Мы начнем с обсуждения базовых строительных блоков сетей CNN, используя восходящий подход. Затем мы глубже погрузимся в архитектуру CNN и выясним, как реализовывать сети CNN в TensorFlow. В главе будут раскрыты следующие темы:

- операции свертки в одном или двух измерениях;
- строительные блоки архитектур CNN;
- реализация глубоких сверточных нейронных сетей в TensorFlow;
- методики дополнения данных с целью увеличения эффективности обобщения;
- реализация классификатора изображений лиц на основе сети CNN для прогнозирования пола человека.



## Строительные блоки сверточных нейронных сетей

Сверточные нейронные сети (CNN) представляют собой семейство моделей, появление которых было вдохновлено способом работы зрительной коры головного мозга при опознавании объектов. Разработка сетей CNN началась в 1990-х годах, когда Ян Лекун и его коллеги предложили новаторскую архитектуру нейронных сетей для классификации рукописных цифр по их изображениям (“Handwritten Digit Recognition with a Back-Propagation Network” (Распознавание рукописных цифр с помощью сети обратного распространения), Я. Лекун и др., конференция по нейронным системам обработки информации (NeurIPS), 1989 г.).



### Зрительная кора головного мозга человека

Первоначальное открытие того, как функционирует зрительная кора нашего головного мозга, сделали Дэвид Х. Хьюбел и Торстен Визель в 1959 году, когда ввели микроэлектрод в первичную зрительную кору анестезированной кошки. Затем они заметили, что нейроны головного мозга по-разному реагируют после проецирования перед кошкой различных световых шаблонов. В конечном итоге это привело к открытию разных слоев зрительной коры. Хотя первичный слой выявляет преимущественно грани и прямые линии, слои более высоких порядков больше сфокусированы на выделении сложных форм и образов.

Благодаря выдающейся эффективности сетей CNN при решении задач классификации изображений к этому конкретному типу нейронных сетей прямого распространения было приковано большое внимание, что привело к крупным улучшениям в МО для компьютерного зрения. Спустя несколько лет, в 2019 году, Ян Лекун вместе с остальными двумя исследователями, Йошуа Бенджи и Джефффри Хинтоном, имена которых уже упоминались в предшествующих главах, получили премию Тьюринга (самую престижную премию в информатике) за свой вклад в область искусственного интеллекта.

В последующих разделах мы обсудим более широкую концепцию сетей CNN и посмотрим, почему сверточные архитектуры часто описываются как “слои выделения признаков”. Затем мы углубимся в теоретическое определение вида операции свертки, которая обычно применяется в сетях CNN, и проработаем примеры для вычисления сверток в одном и двух измерениях.

## Понятие сетей CNN и иерархий признаков

Успешное извлечение *заметных (значимых) признаков* — ключевой аспект эффективности любого алгоритма МО, и традиционные модели МО опираются на входные признаки, которые могут поступать от эксперта в предметной области или основываться на вычислительных методиках выделения признаков. Определенные типы нейронных сетей, такие как CNN, способны автоматически выявлять в сырых данных признаки, которые наиболее полезны для отдельно взятой задачи. По этой причине слои CNN принято считать средствами выделения признаков: начальные слои (находящиеся сразу после входного слоя) извлекают *низкоуровневые признаки* из сырых данных, а более поздние слои (зачастую *полносвязные слои* вроде имеющихся в многослойном персептроне) используют такие признаки для прогнозирования непрерывного целевого значения или метки класса.

Определенные типы многослойных нейронных сетей, в особенности глубокие сверточные нейронные сети, создают так называемую *иерархию признаков*, объединяя низкоуровневые признаки в послойной манере для формирования высокоуровневых признаков. Например, если бы мы имели дело с изображениями, тогда начальные слои извлекали бы низкоуровневые признаки наподобие граней и пятен, которые объединялись бы вместе с целью образования высокоуровневых признаков. Высокоуровневые признаки способны создавать более сложные формы, такие как общие контуры объектов вроде зданий, котов и собак. На рис. 15.1 видно, что сеть CNN вычисляет из входного изображения *карты признаков*, где каждый элемент порождается на основе участка пикселей во входном изображении.

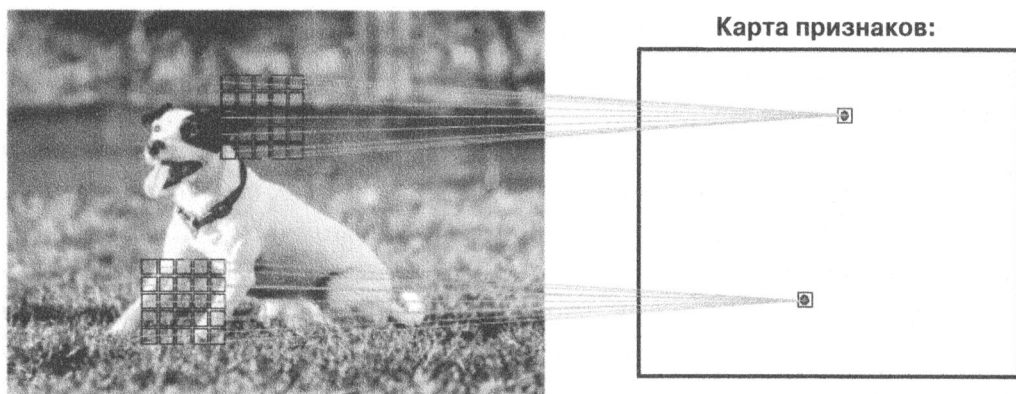


Рис. 15.1. Карты признаков (фотография Александра Даммера из веб-сайта Unsplash)

На такой локальный участок пикселей ссылаются как на *локальное рецепторное поле*. Сети CNN обычно будут очень хорошо работать при решении задач, связанных с изображениями, что в значительной степени объясняется двумя важными идеями.

- *Разреженная связность*. Одиночный элемент в карте признаков связывается только с небольшим участком пикселей. (Это сильно отличается от связывания с целым входным изображением, как в случае перцептронов. Вы можете счесть полезным еще раз взглянуть на реализацию в главе 12 полносвязной сети, которая связана с целым изображением.)
- *Совместное использование параметров*. Для разных участков входного изображения применяются те же самые веса.

В качестве прямого следствия указанных двух идей замена традиционного полносвязного многослойного перцептрона сверточным слоем существенно уменьшает количество весов (параметров) в сети, и мы будем наблюдать улучшение способности захвата *заметных* признаков. В контексте данных изображения рационально предполагать, что близлежащие пиксели имеют большее отношение друг к другу, чем пиксели, находящиеся далеко друг от друга.

Сети CNN обычно состоят из нескольких *сверточных слоев* и слоев подвыборки, за которыми следует один или большее число полносвязных слоев в конце. Полносвязные слои по существу представляют собой многослойный перцептрон, где каждый входной элемент  $i$  связан с каждым выходным элементом  $j$  с весом  $w_{ij}$  (см. главу 12).

Важно отметить, что слои подвыборки, также называемые *объединяющими слоями*, не имеют каких-либо обучаемых параметров; например, в объединяющих слоях отсутствуют веса или элементы смещения. Однако сверточные и полносвязные слои располагают весами и смещениями, которые оптимизируются во время обучения.

В последующих разделах мы более подробно исследуем сверточные и полносвязные слои, ознакомившись с особенностями их функционирования. Чтобы разобраться в работе операций свертки, мы начнем со свертки в одном измерении, которая временами используется для обработки определенных видов последовательных данных, таких как текст. После обсуждения одномерных сверток мы займемся типовыми двумерными свертками, обычно применяемыми к двумерным изображениям.

## Выполнение дискретных сверток

*Дискретная свертка* (или просто *свертка*) является фундаментальной операцией в сети CNN и потому важно понимать, как она работает. В этом разделе мы представим математическое определение и обсудим ряд *наивных* алгоритмов для вычисления сверток одномерных тензоров (векторов) и двумерных тензоров (матриц).

Обратите внимание, что формулы и описания, приводимые в настоящем разделе, предназначены только для понимания, каким образом работают операции свертки в сетях CNN. Позже в главе будет показано, что в таких пакетах, как TensorFlow, доступны более эффективные реализации сверточных операций.



На заметку!

### Математические обозначения

Для обозначения размера многомерного массива (тензора) в главе мы будем использовать подстрочные индексы; скажем,  $A_{n_1 \times n_2}$  — двумерный массив размера  $n_1 \times n_2$ . Мы применяем квадратные скобки [ ] для обозначения индексации в многомерном массиве.

Например,  $A[i, j]$  означает элемент по индексу  $i, j$  матрицы  $A$ . Кроме того, мы используем специальный символ  $*$  для обозначения операции свертки между двумя векторами или матрицами, который не следует путать с символом операции умножения  $*$  в языке Python.

### Дискретные свертки в одном измерении

Давайте начнем с нескольких базовых определений и обозначений, которые мы собираемся применять. Дискретная свертка для двух одномерных векторов  $x$  и  $w$  обозначается как  $y = x * w$ , где вектор  $x$  — наш вход (иногда называемый *сигналом*), а  $w$  — *фильтр* или *ядро*. Математически дискретная свертка определяется следующим образом:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]$$

Как упоминалось ранее, квадратные скобки [ ] используются для обозначения индексации элементов в векторе. Индекс  $i$  проходит по всем элементам выходного вектора  $y$ . В предыдущем уравнении есть два странных аспекта, которые нуждаются в пояснении: индексы от  $-\infty$  до  $+\infty$  и отрицательная индексация для  $x$ .

Факт суммирования по индексам от  $-\infty$  до  $+\infty$  выглядит странным в основном потому, что в приложениях МО мы всегда имеем дело с конечными векторами признаков. Например, если  $x$  содержит 10 признаков с индексами 0, 1, 2, ..., 8, 9, тогда индексы  $-\infty:-1$  и  $10:+\infty$  выходят за допустимые границы вектора  $x$ . Следовательно, для корректного вычисления суммы в предыдущем уравнении предполагается, что векторы  $x$  и  $w$  заполнены нулями. В результате выходной вектор  $y$  также будет иметь бесконечный размер и много нулей. Поскольку на практике это не особенно удобно, вектор  $x$  заполняется лишь конечным числом нулей.

Такой процесс называется *дополнением нулями* и просто *дополнением*. Количество нулей, дополняемых с каждой стороны, обозначается как  $p$ . На рис. 15.2 приведен пример дополнения одномерного вектора  $x$ .



Рис. 15.2. Дополнение нулями одномерного вектора  $x$

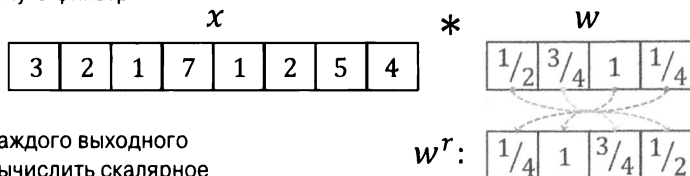
Пусть изначально входной вектор  $x$  и вектор фильтра  $w$  содержат соответственно  $n$  и  $m$  элементов, где  $m \leq n$ . Таким образом, дополненный вектор  $x^p$  имеет размер  $n + 2p$ . Практическое уравнение для вычисления дискретной свертки изменится следующим образом:

$$y = x * w \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i + m - k] w[k]$$

Теперь, когда проблема с бесконечными индексами решена, мы займемся второй проблемой — индексацией вектора  $x$  посредством  $i + m - k$ . Здесь важно отметить, что при суммировании векторы  $x$  и  $w$  индексируются в разных направлениях. Вычисление суммы с одним индексом, идущим в обратном направлении, эквивалентно вычислению суммы с обоими индексами в прямом направлении после зеркального обращения одного из векторов,  $x$  или  $w$ , как только они будут дополнены. Затем мы можем просто вычис-

лить их скалярное произведение. Предположим, что мы зеркально обратили (повернули) фильтр  $w$ , чтобы получить повернутый фильтр  $w^r$ . Далее мы вычисляем скалярное произведение  $x[i : i + m] \cdot w^r$  для получения одного элемента  $y[i]$ , где  $x[i : i + m]$  — участок  $x$  размера  $m$ . Такая операция повторяется подобно тому, как делается в подходе со скользящим окном, чтобы получить все выходные элементы. На рис. 15.3 показан пример вычисления первых трех выходных элементов для  $x = [3 \ 2 \ 1 \ 7 \ 1 \ 2 \ 5 \ 4]$  и  $w = \left(\frac{1}{2}, \frac{3}{4}, 1, \frac{1}{4}\right)$ .

**Шаг 1:** повернуть фильтр



**Шаг 2:** для каждого выходного элемента  $i$  вычислить скалярное произведение  $x[i : i + 4] \cdot w^r$  (сместить фильтр на 2 ячейки)

$$\begin{aligned}
 y[0] &= 3 \times \frac{1}{4} + 2 \times 1 + 1 \times \frac{3}{4} + 7 \times \frac{1}{2} \\
 \rightarrow y[0] &= 7
 \end{aligned}
 \left\{
 \begin{array}{|c|c|c|c|c|c|c|c|}
 \hline
 3 & 2 & 1 & 7 & 1 & 2 & 5 & 4 \\
 \hline
 \frac{1}{4} & 1 & \frac{3}{4} & \frac{1}{2} & & & & \\
 \hline
 \end{array}
 \right.$$

$$\begin{aligned}
 y[1] &= 1 \times \frac{1}{4} + 7 \times 1 + 1 \times \frac{3}{4} + 2 \times \frac{1}{2} \\
 \rightarrow y[1] &= 9
 \end{aligned}
 \left\{
 \begin{array}{|c|c|c|c|c|c|c|c|}
 \hline
 3 & 2 & 1 & 7 & 1 & 2 & 5 & 4 \\
 \hline
 & & \frac{1}{4} & 1 & \frac{3}{4} & \frac{1}{2} & & \\
 \hline
 \end{array}
 \right.$$

$$\begin{aligned}
 y[2] &= 1 \times \frac{1}{4} + 2 \times 1 + 5 \times \frac{3}{4} + 4 \times \frac{1}{2} \\
 \rightarrow y[2] &= 8
 \end{aligned}
 \left\{
 \begin{array}{|c|c|c|c|c|c|c|c|}
 \hline
 3 & 2 & 1 & 7 & 1 & 2 & 5 & 4 \\
 \hline
 & & & & \frac{1}{4} & 1 & \frac{3}{4} & \frac{1}{2} \\
 \hline
 \end{array}
 \right.$$

**Рис. 15.3.** Пример вычисления первых трех выходных элементов скалярного произведения  $x[i : i + m] \cdot w^r$

В предыдущем примере видно, что размер дополнения равен нулю ( $p = 0$ ). Обратите внимание, что повернутый фильтр  $w^r$  каждый раз *смещается* на две ячейки. Такое смещение является еще одним гиперпараметром свертки — *страйдом*  $s$ . В представленном выше примере страйд равен двум ( $s = 2$ ). Важно отметить, что страйд должен быть положительным числом, которое меньше размера входного вектора. Более подробно дополнение и страйды обсуждаются в следующем разделе.



## Взаимная корреляция

Взаимная корреляция (или просто корреляция) между входным вектором и фильтром обозначается как  $y = x * w$  и очень похожа на родственницу свертки, но с небольшим отличием — во взаимной корреляции умножение выполняется в одном и том же направлении. Следовательно, поворачивать фильтр  $w$  в каждом измерении не требуется. Математически взаимная корреляция определяется так:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i + k] w[k]$$

Те же самые правила для дополнения и страйда могут также применяться к взаимной корреляции. Обратите внимание, что большинство фреймворков для ГО (включая TensorFlow) реализуют взаимную корреляцию, но ссылаются на нее как на свертку, что является распространенным соглашением в области ГО.

## Дополнение входов для контроля размера выходных карт признаков

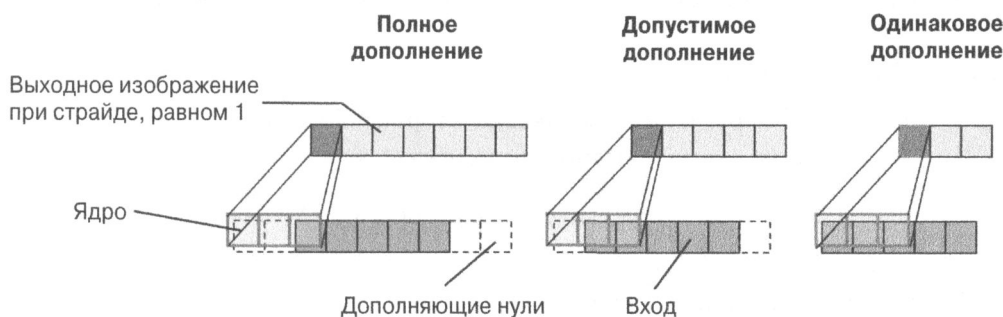
До сих пор мы использовали дополнение нулями в свертках для вычисления выходных векторов с конечными размерами. Формально дополнение может применяться с любым  $p \geq 0$ . В зависимости от выбранного значения  $p$  граничные ячейки могут трактоваться иначе, чем ячейки, расположенные в середине  $x$ .

Теперь давайте рассмотрим пример при  $n = 5$  и  $m = 3$ . Тогда при  $p = 0$  элемент  $x[0]$  используется только в вычислении одного выходного элемента (скажем,  $y[0]$ ), в то время как  $x[1]$  применяется в вычислении двух выходных элементов (например,  $y[0]$  и  $y[1]$ ). Таким образом, можно заметить, что это отличающееся обращение с элементами  $x$  способно искусственно придать больший вес среднему элементу,  $x[2]$ , поскольку он появляется в большинстве вычислений. Мы можем устранить проблему, выбрав  $p = 2$ , и в таком случае каждый элемент  $x$  будет вовлечен в вычисление трех элементов  $y$ .

Кроме того, размер выхода  $y$  также зависит от выбора используемой стратегии дополнения. Существуют три режима дополнения, которые широко применяются на практике: *полный* (*full*), *одинаковый* (*same*) и *допустимый* (*valid*).

- В полном режиме параметр дополнения  $p$  устанавливается в  $p = m - 1$ . Полное дополнение увеличивает количество измерений вывода и поэтому редко используется в архитектурах сверточных нейронных сетей.
- Одинаковое дополнение обычно применяется, если нужно гарантировать, что выходной вектор имеет такой же размер, как входной вектор  $x$ . В этом случае параметр дополнения  $p$  вычисляется в соответствии с размером фильтра вместе с требованием, чтобы размер входа и размер выхода совпадали.
- Наконец, вычисление свертки в допустимом режиме относится к случаю, когда  $p = 0$  (дополнение отсутствует).

На рис. 15.4 иллюстрируются три режима дополнения для простого входа  $5 \times 5$  пикселей с размером ядра  $3 \times 3$  и страйдом 1.



**Рис. 15.4.** Три режима дополнения

В сверточных нейронных сетях наиболее распространено *одинаковое* дополнение. Одно из преимуществ данного режима по сравнению с другими заключается в том, что одинаковое дополнение предохраняет размер вектора (или высоту и ширину входных изображений при обработке задач компьютерного зрения), повышая удобство проектирования архитектуры сети.

Крупный недостаток допустимого дополнения в сравнении с полным и одинаковым дополнением, состоит в том, что в нейронных сетях с множеством слоев объем тензоров значительно уменьшается, и это может пагубно сказаться на эффективности сети.

На практике рекомендуется предохранять пространственный размер, используя одинаковое дополнение для сверточных слоев, и взамен уменьшать пространственный размер посредством объединяющих слоев. Что касается



полного дополнения, то его размер приводит к тому, что выход оказывается больше входа. Полное дополнение обычно применяется в приложениях обработки сигналов, где важно свести к минимуму граничные эффекты. Тем не менее, в контексте ГО граничный эффект, как правило, не является проблемой и потому полное дополнение на практике встречается редко.

### Определение размера выхода свертки

Размер выхода свертки определяется общим количеством сдвигов фильтра  $w$  вдоль входного вектора. Пусть входной вектор имеет размер  $n$ , а фильтр — размер  $m$ . Тогда размер выхода из  $y = x * w$  с дополнением  $p$  и страйдом  $s$  определяется следующим образом:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

Здесь  $\lfloor \cdot \rfloor$  обозначает операцию округления в меньшую сторону (*floor*).



Совет

### Операция округления в меньшую сторону

Операция округления в меньшую сторону возвращает наибольшее целое число, которое равно или меньше входа, например:

$$\text{floor}(1.77) = \lfloor 1.77 \rfloor = 1$$

Рассмотрим следующие два случая.

- Вычислить размер выхода для входного вектора размера 10 с ядром свертки размера 5, дополнением 2 и страйдом 1:

$$n = 10, m = 5, p = 2, s = 1 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 5}{1} \right\rfloor + 1 = 10$$

(Обратите внимание, что в данном случае размер выхода оказывается таким же, как размер входа, поэтому мы выбираем режим одинакового дополнения.)

- Как изменится размер выхода для того же самого входного вектора, когда мы имеем ядро размера 3 и страйд 2?

$$n = 10, m = 3, p = 2, s = 2 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 3}{2} \right\rfloor + 1 = 6$$

Если вас интересуют дополнительные сведения о размере выхода свертки, то рекомендуем ознакомиться со статьей Винсента Дюмулена и Франческо Визина “A guide to convolution arithmetic for deep learning” (Руководство по арифметике свертки для глубокого обучения), которая свободно доступна по ссылке <https://arxiv.org/abs/1603.07285>.

В заключение для демонстрации вычисления сверток в одном измерении ниже показана наивная реализация, результаты которой сравниваются с функцией `numpy.convolve`:

```
>>> import numpy as np
>>> def conv1d(x, w, p=0, s=1):
...     w_rot = np.array(w[::-1])
...     x_padded = np.array(x)
...     if p > 0:
...         zero_pad = np.zeros(shape=p)
...         x_padded = np.concatenate([zero_pad,
...                                     x_padded,
...                                     zero_pad])
...     res = []
...     for i in range(0, int(len(x)/s), s):
...         res.append(np.sum(x_padded[i:i+w_rot.shape[0]] *
...                           w_rot))
...     return np.array(res)
>>> ## Проверка:
>>> x = [1, 3, 2, 4, 5, 6, 1, 3]
>>> w = [1, 0, 3, 1, 2]
>>> print('Реализация Conv1d:',
...       conv1d(x, w, p=2, s=1))
Реализация Conv1d: [ 5. 14. 16. 26. 24. 34. 19. 22.]
>>> print('Результаты NumPy:',
...       np.convolve(x, w, mode='same'))
Результаты NumPy: [ 5 14 16 26 24 34 19 22]
```

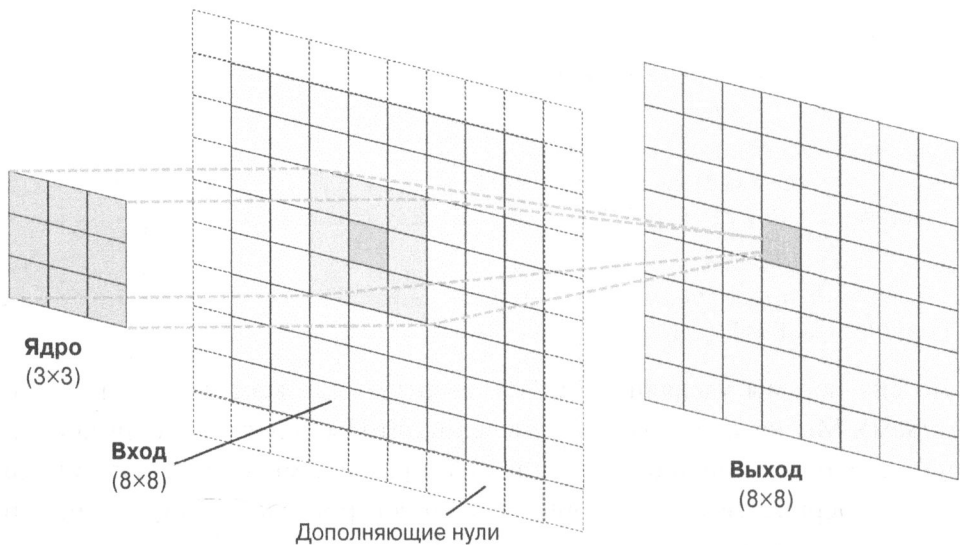
До сих пор мы уделяли внимание сверткам для векторов (одномерным сверткам). Мы начали с одного измерения, чтобы облегчить понимание лежащих в основе концепций. В следующем разделе мы более подробно рассмотрим двумерные свертки, которые являются строительными блоками сетей CNN, ориентированных на решение задач с изображениями.

## Выполнение дискретной свертки в двух измерениях

Концепции, которые объяснялись в предшествующих разделах, легко расширяются на два измерения. Когда мы имеем дело с двумерными входами, такими как матрица  $\mathbf{X}_{n_1 \times n_2}$  и матрицей фильтра  $\mathbf{W}_{m_1 \times m_2}$ , где  $m_1 \leq n_1$  и  $m_2 \leq n_2$ , тогда результатом двумерной свертки между  $\mathbf{X}$  и  $\mathbf{W}$  будет матрица  $\mathbf{Y} = \mathbf{X} * \mathbf{W}$ . Вот математическое определение:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2]$$

Обратите внимание, что если мы опустим одно из измерений, то получим в точности то же самое уравнение, которое использовалось ранее для вычисления свертки в одном измерении. Фактически все упомянутые выше приемы вроде дополнения нулями, поворота матрицы фильтра и выбора страйдов также применимы к двумерным сверткам при условии, что они расширяются на оба измерения независимо. На рис. 15.5 демонстрируется двумерная свертка входной матрицы размером  $8 \times 8$  с использованием ядра размером  $3 \times 3$ . Входная матрица дополняется нулями с  $p = 1$ . В результате выход двумерной свертки будет иметь размер  $8 \times 8$ .



**Рис. 15.5.** Двумерная свертка входной матрицы размером  $8 \times 8$  с применением ядра размером  $3 \times 3$

Следующий пример иллюстрирует вычисление двумерной свертки между входной матрицей  $X_{3 \times 3}$  и матрицей ядра  $W_{3 \times 3}$  с дополнением  $p = (1, 1)$  и страйдом  $s = (2, 2)$ . Согласно указанному дополнению к каждой стороне входной матрицы добавляется один слой нулей, давая в результате дополненную матрицу  $X_{5 \times 5}^{\text{дополненная}}$  (рис. 15.6).

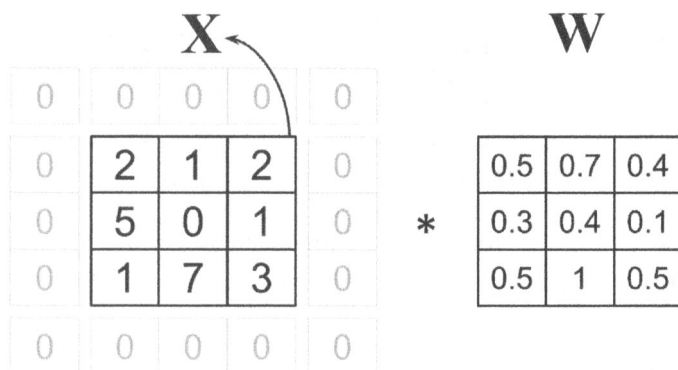


Рис. 15.6. Результирующая дополненная матрица

Повернутым фильтром для предыдущего фильтра будет:

$$W^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Обратите внимание, что поворот — это не транспонирование матрицы. Чтобы получить повернутый фильтр в NumPy, мы записываем  $W_{\text{rot}} = W[:, :, -1, :, -1]$ . Далее мы сдвигаем матрицу фильтра вдоль дополненной входной матрицы  $X^{\text{дополненная}}$  подобно скользящему окну и вычисляем сумму поэлементных произведений, которая на рис. 15.7 обозначается операцией  $\odot$ .

Результатом будет матрица  $Y$  размера  $2 \times 2$ .

Давайте реализуем также и двумерную свертку в соответствии с описанным *наивным* алгоритмом.

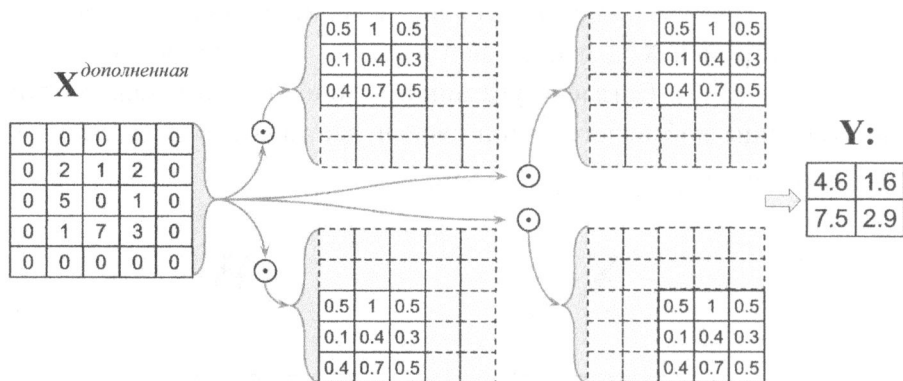


Рис. 15.7. Пример получения двумерной свертки

Пакет `scipy.signal` предлагает способ вычисления двумерной свертки через функцию `scipy.signal.convolve2d`:

```
>>> import numpy as np
>>> import scipy.signal

>>> def conv2d(X, W, p=(0, 0), s=(1, 1)):
...     W_rot = np.array(W)[:,::-1,::-1]
...     X_orig = np.array(X)
...     n1 = X_orig.shape[0] + 2*p[0]
...     n2 = X_orig.shape[1] + 2*p[1]
...     X_padded = np.zeros(shape=(n1, n2))
...     X_padded[p[0]:p[0]+X_orig.shape[0],
...               p[1]:p[1]+X_orig.shape[1]] = X_orig
...
...     res = []
...     for i in range(0, int((X_padded.shape[0] - \
...                             W_rot.shape[0])/s[0])+1, s[0]):
...         res.append([])
...         for j in range(0, int((X_padded.shape[1] - \
...                                 W_rot.shape[1])/s[1])+1, s[1]):
...             X_sub = X_padded[i:i+W_rot.shape[0],
...                               j:j+W_rot.shape[1]]
...             res[-1].append(np.sum(X_sub * W_rot))
...     return(np.array(res))
>>> X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
>>> W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]

>>> print('Реализация Conv2d:\n',
...       conv2d(X, W, p=(1, 1), s=(1, 1)))
```

Реализация Conv2d:

```
[ [ 11. 25. 32. 13.]
  [ 19. 25. 24. 13.]
  [ 13. 28. 25. 17.]
  [ 11. 17. 14.  9.]]
```

```
>>> print('Результаты SciPy:\n',
...       scipy.signal.convolve2d(X, W, mode='same'))
```

Результаты SciPy:

```
[ [11 25 32 13]
  [19 25 24 13]
  [13 28 25 17]
  [11 17 14  9]]
```



На  
заметку!

### Эффективные алгоритмы для вычисления сверток

Мы предоставили наивную реализацию для вычисления двумерной свертки с целью лучшего понимания концепций. Однако такая реализация крайне неэффективна в плане требований к памяти и вычислительной сложности. Следовательно, она не должна использоваться в реальных приложениях нейронных сетей.

Один из аспектов заключается в том, что матрица фильтра в действительности не поворачивается в большинстве инструментов, подобных TensorFlow. Кроме того, в последние годы были разработаны намного более эффективные алгоритмы, которые для вычисления сверток применяют преобразование Фурье. Также важно отметить, что в контексте нейронных сетей размер ядра свертки обычно гораздо меньше размера входного изображения.

Например, современные сети CNN обычно используют размеры ядер  $1 \times 1$ ,  $3 \times 3$  или  $5 \times 5$ , для которых были спроектированы эффективные алгоритмы, способные выполнять сверточные операции намного более рационально, например, алгоритм *минимальной фильтрации Винограда*. Исследование алгоритмов подобного рода выходит за рамки настоящей книги, но если вы заинтересовались, тогда можете почитать статью Эндрю Лэвина и Скотта Грея “Fast Algorithms for Convolutional Neural Networks” (Быстрые алгоритмы для сверточных нейронных сетей), свободно доступную по ссылке <https://arxiv.org/abs/1509.09308>.

В следующем разделе мы обсудим подвыборку, которая является еще одной важной операцией, часто встречающейся в сетях CNN.

## Слои подвыборки

В сверточных нейронных сетях подвыборка, как правило, применяется в двух формах операций объединения: *объединение по максимуму* (*max-pooling*) и *объединение по среднему* (*mean-pooling* или *average-pooling*). Объединяющий слой обычно обозначается с помощью  $P_{n_1 \times n_2}$ . Здесь подстрочный индекс определяет размер близлежащей области (количество смежных пикселей в каждом измерении), где выполняется операция получения максимума или среднего. Мы ссылаемся на такую близлежащую область как на *размер объединения*.

Работа подвыборки демонстрируется на рис. 15.8. Операция объединения по максимуму получает максимальное значение из близлежащей области пикселей, а операция объединения по среднему вычисляет их среднее значение.



Рис. 15.8. Работа подвыборки

Преимущество объединения двояко.

- Объединение (объединение по максимуму) привносит локальную инвариантность. Это означает, что небольшие изменения в локальной близлежащей области не изменяют результат объединения по максимуму. Следовательно, инвариантность помогает генерировать признаки, более устойчивые к шуму во входных данных. Ниже представлен при-

мер, который показывает, что объединение по максимуму двух входных матриц  $X_1$  и  $X_2$  дает в итоге тот же самый выход:

$$\begin{array}{l}
 X_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix} \\
 X_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix}
 \end{array}
 \xrightarrow{\text{объединение по максимуму } P_{2 \times 2}}
 \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

- Объединение уменьшает размер признаков, что в результате приводит к более высокой вычислительной эффективности. Вдобавок сокращение количества признаков может также уменьшить степень переобучения.



На заметку!

### Пересекающееся или непересекающееся объединение

Традиционно предполагается, что объединение должно быть непересекающимся. Объединение обычно выполняется на непересекающихся близлежащих областях, которые можно сделать, устанавливая параметр страйда равным размеру объединения. Например, непересекающийся объединяющий слой  $P_{n_1 \times n_2}$  требует параметра страйда  $s = (n_1, n_2)$ . С другой стороны, пересекающееся объединение случается, когда страйд меньше размера объединения. Пример использования пересекающегося объединения в сверточной сети описан в работе А. Крижевски, И. Сацкевера и Д. Хинтона “ImageNet Classification with Deep Convolutional Neural Networks” (Классификация ImageNet с помощью глубоких сверточных нейронных сетей), которая свободно доступна по ссылке <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>.



Хотя объединение по-прежнему является важной частью многих архитектур сетей CNN, также было разработано несколько архитектур CNN, в которых объединяющие слои не используются. Вместо применения объединяющих слоев для сокращения размера признаков исследователи используют сверточные слои со страйдом 2.

В известной мере вы можете представлять себе сверточный слой со страйдом 2 как объединяющий слой с обучаемыми весами. Если вас интересует эмпирическое сравнение разных архитектур сетей CNN, спроектированных с и без объединяющих слоев, тогда мы рекомендуем ознакомиться с исследовательской работой “Striving for Simplicity: The All Convolutional Net” (Стремление к простоте: все сверточные сети), написанной Йостом Тобиасом Спрингербергом, Алексеем Досовицким, Томасом Броксом и Мартином Ридмиллером, которая свободно доступна по ссылке <https://arxiv.org/abs/1412.6806>.

## Группирование всего вместе — реализация сверточной нейронной сети

К настоящему моменту вы уже знаете о базовых строительных блоках сверточных нейронных сетей. Проиллюстрированные в главе концепции на самом деле не сложнее концепций, которые лежат в основе традиционных многослойных нейронных сетей. Мы можем сказать, что самой важной операцией в традиционной нейронной сети является перемножение матриц. Например, мы применяем перемножение матриц для расчета предварительных активаций (или общих входов) как в  $z = \mathbf{W}x + b$ . Здесь  $x$  — вектор-столбец (матрица  $\mathbb{R}^{n \times 1}$ ), представляющий пиксели, а  $\mathbf{W}$  — весовая матрица, связывающая входы пикселей с каждым скрытым элементом.

В сети CNN такая операция заменяется операцией свертки как в  $\mathbf{A} = \mathbf{W} * \mathbf{X} + b$ , где  $\mathbf{X}$  — матрица, представляющая пиксели с расположением *высота*×*ширина*. В обоих случаях предварительные активации передаются функции активации, чтобы получить активацию скрытого элемента  $\mathbf{A} = \phi(\mathbf{Z})$ , где  $\phi$  — функция активации. Кроме того, вспомните, что еще одним строительным блоком сети CNN является подвыборка, которая может встречаться в форме объединения, как было описано в предыдущем разделе.

## Работа с множественными входными или цветовыми каналами

Вход сверточного слоя может содержать один и более двумерных массивов или матриц с измерениями  $N_1 \times N_2$  (скажем, высотой и шириной изображения в пикселях). Такие матрицы  $N_1 \times N_2$  называются *каналами*. Традиционные реализации сверточных слоев ожидают на входе представления тензора ранга 3, например, трехмерного массива  $\mathbf{X}_{N_1 \times N_2 \times C_{\text{ex}}}$ , где  $C_{\text{ex}}$  — количество входных каналов. Скажем, давайте рассмотрим изображения в качестве входа в первый слой сети CNN. Если изображение цветное и использует цветовой режим RGB, тогда  $C_{\text{ex}} = 3$  (для красного, зеленого и синего цветовых каналов в RGB). Тем не менее, если изображение представлено в оттенках серого, то мы имеем  $C_{\text{ex}} = 1$ , потому что есть только один канал со значениями интенсивности пикселей в оттенках серого.



Совет

### Чтение файла изображения

При работе с изображениями мы можем читать их в массивы NumPy с применением типа данных `uint8` (8-битное целое число без знака), чтобы сократить расход памяти в сравнении, например, с 16-, 32- или 64-битными целочисленными типами. Беззнаковые 8-битные целые числа имеют значения в диапазоне  $[0, 255]$ , которых достаточно для хранения информации о пикселях изображений RGB, также принимающие значения в том же самом диапазоне.

В главе 13 вы узнали, что библиотека TensorFlow предлагает модуль для загрузки/сохранения и манипулирования изображениями через подмодули `tf.io` и `tf.image`. Давайте кратко повторим, как прочитать изображение (это изображение RGB находится в подкаталоге `images` каталога примеров для текущей главы <https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch15>):

```
>>> import tensorflow as tf
>>> img_raw = tf.io.read_file('example-image.png')
>>> img = tf.image.decode_image(img_raw)
>>> print('Форма изображения:', img.shape)
Форма изображения: (252, 221, 3)
```

При построении моделей и загрузчиков данных в TensorFlow для чтения входных изображений также рекомендуется использовать подмодуль `tf.image`.

Давайте посмотрим, как можно прочитать изображение в сеансе Python с применением пакета `imageio`, который следует установить посредством `conda` или `pip` в командной строке:

```
> conda install imageio
```

или

```
> pip install imageio
```

После установки `imageio` мы можем вызвать функцию `imread` и прочитать то же самое изображение, которое использовалось ранее, с помощью пакета `imageio`:

```
>>> import imageio
>>> img = imageio.imread('example-image.png')
>>> print('Форма изображения:', img.shape)
Форма изображения: (252, 221, 3)
>>> print('Количество каналов:', img.shape[2])
Количество каналов: 3
>>> print('Тип данных изображения:', img.dtype)
Тип данных изображения: uint8
>>> print(img[100:102, 100:102, :])
[[[179 134 110]
  [182 136 112]]
 [[180 135 11]
  [182 137 113]]]
```

Теперь, когда вы знакомы со структурой входных данных, возникает вопрос: как охватить множественные входные каналы операцией свертки, которая обсуждалась в предшествующих разделах? Ответ очень прост: мы выполняем операцию свертки для каждого канала отдельно и затем складываем результаты вместе, используя суммирование матриц. Свертка, связанная с каждым каналом ( $c$ ), имеет собственную матрицу ядра  $\mathbf{W}[:, :, c]$ . Общий результат предварительной активации вычисляется посредством следующего уравнения:

$$\begin{array}{l} \text{Для заданного образца } \mathbf{X}_{n_1 \times n_2 \times C_{\text{вх}}}, \\ \text{матрицы ядра } \mathbf{W}_{m_1 \times m_2 \times C_{\text{вх}}} \\ \text{и значения смещения } b \end{array} \Rightarrow \begin{cases} \mathbf{Z}^{\text{Свертки}} = \sum_{c=1}^{C_{\text{вх}}} \mathbf{W}[:, :, c] * \mathbf{X}[:, :, c] \\ \text{Предварительная активация: } \mathbf{Z} = \mathbf{Z}^{\text{Свертки}} + b_c \\ \text{Карта признаков: } \mathbf{A} = \phi(\mathbf{Z}) \end{cases}$$

Финальный результат,  $A$ , является картой признаков. Обычно сверточный слой сети CNN имеет более одной карты признаков. Если мы применяем несколько карт признаков, тогда тензор ядра становится четырехмерным:  $\text{ширина} \times \text{высота} \times C_{\text{вх}} \times C_{\text{вых}}$ . Здесь  $\text{ширина} \times \text{высота}$  — размер ядра,  $C_{\text{вх}}$  — количество входных каналов и  $C_{\text{вых}}$  — количество выходных карт признаков. Итак, давайте включим количество выходных карт признаков в предыдущее уравнение, обновив его, как показано ниже:

$$\begin{aligned} \text{Для заданного образца } X_{n_1 \times n_2 \times C_{\text{вх}}}, \\ \text{матрицы ядра } W_{m_1 \times m_2 \times C_{\text{вх}} \times C_{\text{вых}}} \\ \text{и значения смещения } b_{C_{\text{вых}}} \end{aligned} \Rightarrow \begin{cases} Z^{\text{Свертки}}[:, :, k] = \sum_{c=1}^{C_{\text{вх}}} W[:, :, c, k] * X[:, :, c] \\ Z[:, :, k] = Z^{\text{Свертки}}[:, :, k] + b[k] \\ A[:, :, k] = \phi(Z[:, :, k]) \end{cases}$$

В завершение нашего обсуждения вычислений сверток в контексте нейронных сетей мы рассмотрим пример, представленный на рис. 15.9, где показан сверточный слой, за которым следует объединяющий слой. В этом примере есть три входных канала. Тензор ядра является четырехмерным. Каждая матрица ядра обозначается как  $m_1 \times m_2$ , и таких матриц три, по одной на входной канал. Вдобавок имеется пять ядер, которые приходятся на пять выходных карт признаков. Наконец, предусмотрен объединяющий слой для подвыборки карт признаков.

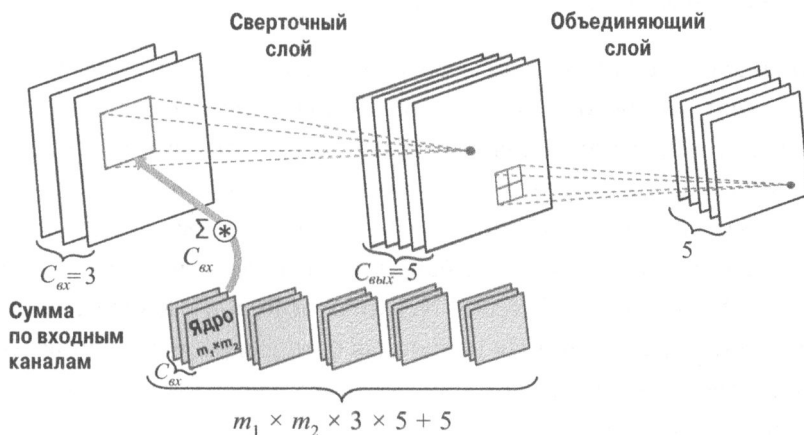


Рис. 15.9. Вычисление свертки в нейронной сети



### Сколько обучаемых параметров существует в предыдущем примере?

Чтобы проиллюстрировать преимущества свертки, *совместного использования параметров и разреженной связности*, давайте проработаем пример. Сверточный слой в сети на рис. 15.9 представляет собой четырехмерный тензор. Следовательно, есть  $m_1 \times m_2 \times 3 \times 5$  параметров, ассоциированных с ядром. Кроме того, для каждой выходной карты признаков сверточного слоя имеется вектор смещения. Таким образом, размер вектора смещения равен 5. Объединяющие слои не имеют каких-либо (обучаемых) параметров, поэтому мы можем записать так:

$$m_1 \times m_2 \times 3 \times 5 + 5$$

Если входной тензор имеет размер  $n_1 \times n_2 \times 3$  и предполагается, что свертка выполняется в режиме одинакового дополнения, тогда выходные карты признаков будут иметь размер  $n_1 \times n_2 \times 5$ .

Обратите внимание, что если вместо сверточного слоя мы применяем полносвязный слой, то это число будет намного больше. Для достижения того же самого количества выходных элементов в случае использования полносвязного слоя число параметров для весовой матрицы выражалось бы так:

$$(n_1 \times n_2 \times 3) \times (n_1 \times n_2 \times 5) = (n_1 \times n_2)^2 \times 3 \times 5$$

Вдобавок размер вектора смещения составляет  $n_1 \times n_2 \times 5$  (по одному элементу смещения для каждого выходного элемента). При условии, что  $m_1 < n_1$  и  $m_2 < n_2$ , разница в количествах обучаемых параметров оказывается огромной.

Наконец, как уже упоминалось, обычно операции свертки выполняются за счет обработки входного изображения с множеством цветовых каналов как стопки матриц; т.е. мы выполняем свертку на каждой матрице отдельно и затем складываем результаты, как было проиллюстрировано на рис. 15.9. Однако свертки могут быть расширены и на трехмерные объемы, если вы работаете с трехмерными наборами данных, например, как показано в работе 2015 года “VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition” (VoxNet: трехмерная сверточная нейронная сеть для распознавания объектов в реальном времени), написанной Дэниэлом Матурана и Себастьяном Шерером, которая свободно доступна по ссылке [https://www.ri.cmu.edu/pub\\_files/2015/9/voxnet\\_maturana\\_scherer\\_iros15.pdf](https://www.ri.cmu.edu/pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf).

В следующем разделе речь пойдет о регуляризации нейронной сети.

## Регуляризация нейронной сети с помощью отключения

Выбор размера сети, как традиционной (полносвязной) нейронной сети, так и CNN, всегда был сложной задачей. Например, для достижения приемлемой эффективности должны подстраиваться размер весовой матрицы и количество слоев.

В главе 14 вы узнали о том, что простая сеть без скрытых слоев может захватывать только линейную границу решений, которой не будет достаточно для решения задачи XOR или какой-то аналогичной. Под *емкостью* сети понимается уровень сложности функции, которую сеть способна научиться аппроксимировать. Небольшие сети, или сети, имеющие относительно малое число параметров, обладают низкой емкостью, а потому вероятно будут *недообучаться*, в результате демонстрируя плохую эффективность, поскольку они не могут выявлять внутреннюю структуру сложных наборов данных. Тем не менее, очень крупные сети подвержены *переобучению*, т.е. сеть просто запоминает обучающие данные и работает исключительно хорошо на обучающем наборе, но показывает низкую эффективность на удерживаемом испытательном наборе. Имея дело с реальными задачами МО, мы не знаем *заранее*, насколько крупной должна быть сеть.

Один из способов решения проблемы предусматривает построение сети с относительно высокой емкостью (на практике мы хотим выбрать емкость, которая немного выше, чем необходимо) для хорошей работы на обучающем наборе данных. Затем, чтобы предотвратить переобучение, мы можем применить одну или несколько схем регуляризации для достижения приемлемой эффективности обобщения на новых данных, таких как удерживаемый испытательный набор.

В главе 3 мы раскрывали регуляризацию L1 и L2. Там в разделе “Решение проблемы переобучения с помощью регуляризации” вы узнали, что обе методики, L1 и L2, могут предотвратить или ослабить эффект переобучения за счет добавления к потере штрафа, который приводит к сокращению параметров весов во время обучения. Хотя регуляризации L1 и L2 могут использоваться также для нейронных сетей, причем L2 оказывается более частым выбором из двух, существуют другие методы регуляризации нейронных сетей наподобие отключения, которое мы обсудим в текущем раз-

деле. Прежде чем переходить к обсуждению отключения, следует отметить, что для применения регуляризации L2 внутри сверточной или полносвязной (плотной) сети вы можете добавить к функции потерь штраф L2. Для этого при использовании API-интерфейса Keras просто установите аргумент `kernel_regularizer` отдельного слоя (затем он автоматически модифицирует функцию потерь надлежащим образом):

```
>>> from tensorflow import keras
>>> conv_layer = keras.layers.Conv2D(
...     filters=16,
...     kernel_size=(3,3),
...     kernel_regularizer=keras.regularizers.l2(0.001))
>>> fc_layer = keras.layers.Dense(
...     units=16,
...     kernel_regularizer=keras.regularizers.l2(0.001))
```

В последние годы появилась еще одна популярная методика для регуляризации (глубоких) нейронных сетей во избежание их переобучения, которая называется *отключением* (*dropout*). Она была представлена в статье Нитиша Шривастава, Джеффри Хинтона, Алекса Крижевского, Ильи Сатскевера и Руслана Слахутдинова “Dropout: a simple way to prevent neural networks from overfitting” (Отключение: простой способ предохранения нейронных сетей от переобучения), Journal of Machine Learning Research 15.1, стр. 1929–1958 (2014 г.), свободно доступной по ссылке <http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>. Отключение обычно применяется к скрытым элементам более высоких слоев и работает следующим образом: в течение стадии обучения нейронной сети на каждой итерации некоторая доля скрытых элементов случайным образом отбрасывается с вероятностью  $p_{\text{отбрасывания}}$  (или сохраняется с вероятностью  $p_{\text{сохранения}} = 1 - p_{\text{отбрасывания}}$ ). Эта вероятность отключения задается пользователем и часто выбираемым вариантом является  $p = 0.5$ , как обсуждалось в упомянутой выше статье Нитиша Шривастава и др. Когда определенная доля входных нейронов отбрасывается, ассоциированные с оставшимися нейронами веса заново масштабируются, чтобы учесть недостающие (отброшенные) нейроны.

Случайное отключение вынуждает сеть изучать избыточное представление данных. Следовательно, сеть не может полагаться на активацию любого

набора скрытых элементов, поскольку они могут быть отключены в любой момент во время обучения, и вынуждена искать более общие и надежные шаблоны в данных.

Такое случайное отключение в состоянии эффективно препятствовать переобучению. На рис. 15.10 иллюстрируется применение отключения с вероятностью  $p = 0.5$  во время стадии обучения, благодаря чему половина нейронов случайным образом становится неактивной (отбрасываемые элементы выбираются на случайной основе при каждом прямом проходе процесса обучения). Однако во время выработки прогноза в вычислении предварительных активаций следующего слоя будут участвовать все нейроны.



Рис. 15.10. Пример отключения с вероятностью  $p = 0.5$

Важно запомнить, что элементы могут отключаться случайным образом только во время обучения, тогда как на стадии оценки (выведения) все скрытые элементы должны быть активными (например,  $p_{\text{отбрасывания}} = 0$  или  $p_{\text{сохранения}} = 1$ ). Для гарантирования того, что все активации имеют тот же самый масштаб во время обучения и прогнозирования, активации активных нейронов должны быть надлежащим образом масштабированы (скажем, за счет деления активации пополам, если вероятность отключения была установлена в  $p = 0.5$ ).

Однако поскольку постоянно масштабировать активации при выработке прогнозов неудобно, TensorFlow и другие инструменты масштабируют активации во время обучения (например, удваивая активации, если вероятность отключения была установлена в  $p = 0.5$ ). На такой подход обычно ссылаются как на инверсное отключение.



Несмотря на то что связь становится очевидной не сразу, отбрасывание может интерпретироваться как соглашение (усреднение) ансамбля моделей. В главе 7 было показано, что при ансамблевом обучении мы независимо обучаем несколько моделей. На стадии прогнозирования мы используем соглашение между всеми обученными моделями. Вам уже известно, что ансамбли моделей работают лучше одиночных моделей. Тем не менее, при ГО обучение нескольких моделей, а также сбор и усреднение выхода множества моделей будет затратным в вычислительном плане. Отключение предлагает обходной путь с эффективным способом обучения множества моделей за раз и вычисления их средних прогнозов во время испытания или прогнозирования.

Как упоминалось ранее, связь между ансамблями моделей и отбрасыванием не сразу будет очевидной. Однако имейте в виду, что при отбрасывании мы имеем отличающуюся модель для каждого мини-пакета (из-за установки весов в ноль случайным образом во время каждого прямого прохода).

Далее посредством итерации по мини-пакетам мы по существу выбираем свыше  $M = 2^h$  моделей, где  $h$  — количество скрытых элементов.

Тем не менее, ограничение и аспект, который проводит различие между отбрасыванием и обычным объединением в ансамбли, заключаются в том, что мы разделяем веса по таким “разным моделям”, что можно рассматривать как форму регуляризации. Затем во время “выведения” (скажем, прогнозирования меток в испытательном наборе) мы можем усреднить все разные модели, которые были выбраны на стадии обучения. Однако такой процесс крайне затратный.

Усреднение моделей, т.е. расчет среднего геометрического вероятности членства в классах, возвращенной моделью  $i$ , может быть представлен так:

$$p_{\text{Ансамбля}} = \left[ \prod_{j=1}^M p^{\{i\}} \right]^{\frac{1}{M}}$$

Трюк, лежащий в основе отбрасывания, связан с тем, что среднее геометрическое ансамблей моделей (здесь  $M$  моделей) может быть приближенно рассчитано путем масштабирования прогнозов последней (или финальной) модели, выбранной во время обучения, с коэффициентом  $1/(1 - p)$ . Это будет гораздо менее затратным, чем явное вычисление среднего геометрического

с применением предыдущего уравнения. (На самом деле, если мы возьмем линейные модели, то приближение в точности эквивалентно настоящему среднему геометрическому.)

## Функции потерь для классификации

В главе 13 мы исследовали различные функции активации, такие как ReLU, сигмоидальная функция и гиперболический тангенс ( $\tanh$ ). Некоторые из этих функций активации вроде ReLU используются главным образом в промежуточных (скрытых) слоях нейронной сети для добавления к модели нелинейностей. Но другие функции наподобие сигмоидальной (для двоичной классификации) и многопеременной логистической (для многоклассовой классификации) добавляются в последний (выходной) слой, давая на выходе модели вероятности членства в классах. Если сигмоидальная или многопеременная логистическая функция активации не включена в выходной слой, тогда вместо вероятностей членства в классах модель будет рассчитывать логиты.

Сосредоточив внимание на задачах классификации, в зависимости от типа задачи (двоичная или многоклассовая) и типа выхода (логиты или вероятности) мы должны выбрать подходящую функцию потерь для обучения модели. Функцией потерь для двоичной классификации (с единственным выходным элементом) является *двоичная перекрестная энтропия*, а для многоклассовой классификации — *категориальная перекрестная энтропия*. В API-интерфейсе Keras API для потери в виде категориальной перекрестной энтропии предлагаются два варианта, соответствующие формату представления достоверных меток — в унитарном коде (например,  $[0, 0, 1, 0]$ ) или в целочисленном виде (скажем,  $y=2$ ), что в контексте Keras также называется “разреженным” представлением.

В таблице на рис. 15.11 описаны три функции потерь, доступные в Keras для работы во всех трех сценариях: двоичная классификация, многоклассовая классификация с достоверными метками в унитарном коде и многоклассовая классификация с целочисленными (разреженными) метками. Каждая из этих трех функций потерь также обладает возможностью получения прогнозов в форме логитов или вероятностей членства в классах.

Обратите внимание, что вычисление потери в виде перекрестной энтропии с указанием логитов, а не вероятностей членства в классах, обычно предпочтительнее по причинам численной устойчивости.

Функция потерь	Использование	Примеры	
		С использованием вероятностей <i>from_logits=False</i>	С использованием логитов <i>from_logits=True</i>
BinaryCrossentropy	Двоичная классификация	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
CategoricalCrossentropy	Многоклассовая классификация	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
Sparse CategoricalCrossentropy	Многоклассовая классификация	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

Рис. 15.11. Функции потерь, доступные в Keras для задач классификации

Если мы предоставим функции потерь логиты в качестве входа и установим `from_logits=True`, то при расчете потери и производной относительно весов соответствующая функция TensorFlow будет применять более эффективную реализацию. Подобное возможно, т.к. в случае передачи на входе логитов определенные математические члены сокращаются и потому не требуют явного вычисления.

Ниже в коде показано, как использовать три функции потерь с двумя форматами, где функциям потерь на входе предоставляются либо логиты, либо вероятности членства в классах:

```
>>> import tensorflow_datasets as tfds
>>> ##### Двоичная перекрестная энтропия (ДПЭ)
>>> bce_probas =
...     tf.keras.losses.BinaryCrossentropy(from_logits=False)
>>> bce_logits =
...     tf.keras.losses.BinaryCrossentropy(from_logits=True)
>>> logits = tf.constant([0.8])
>>> probas = tf.keras.activations.sigmoid(logits)
>>> tf.print(
...     'ДПЭ (с вероятностями): {:.4f}'.format(
...         bce_probas(y_true=[1], y_pred=probas)),
...     ' (с логитами): {:.4f}'.format(
...         bce_logits(y_true=[1], y_pred=logits)))
ДПЭ (с вероятностями): 0.3711 (с логитами): 0.3711
```

```

>>> ##### Категориальная перекрестная энтропия (КПЭ)
>>> cce_probas = tf.keras.losses.CategoricalCrossentropy(
...     from_logits=False)
>>> cce_logits = tf.keras.losses.CategoricalCrossentropy(
...     from_logits=True)
>>> logits = tf.constant([[1.5, 0.8, 2.1]])
>>> probas = tf.keras.activations.softmax(logits)
>>> tf.print(
...     'КПЭ (с вероятностями): {:.4f}'.format(
...         cce_probas(y_true=[0, 0, 1], y_pred=probas)),
...     ' (с логитами): {:.4f}'.format(
...         cce_logits(y_true=[0, 0, 1], y_pred=logits)))
КПЭ (с вероятностями): 0.5996 (с логитами): 0.5996

>>> ##### Разреженная категориальная перекрестная
...     энтропия (разреженная КПЭ)
>>> sp_cce_probas = tf.keras.losses.
SparseCategoricalCrossentropy(
...     from_logits=False)
>>> sp_cce_logits = tf.keras.losses.
SparseCategoricalCrossentropy(
...     from_logits=True)
>>> tf.print(
...     'Разреженная КПЭ (с вероятностями): {:.4f}'.format(
...         sp_cce_probas(y_true=[2], y_pred=probas)),
...     ' (с логитами): {:.4f}'.format(
...         sp_cce_logits(y_true=[2], y_pred=logits)))
Разреженная КПЭ (с вероятностями): 0.5996 (с логитами): 0.5996

```

Следует отметить, что временами можно встречать реализации, в которых потеря в виде категориальной перекрестной энтропии применяется для двоичной классификации. Обычно когда у нас есть задача двоичной классификации, модель возвращает одиночное выходное значение для каждого образца. Мы интерпретируем такой одиночный выход модели как вероятность положительного класса (например, класса 1),  $P[\text{класс} = 1]$ . В задаче двоичной классификации подразумевается, что  $P[\text{класс} = 0] = 1 - P[\text{класс} = 1]$ ; таким образом, для получения вероятности отрицательного класса нам не нужен второй выходной элемент. Тем не менее, иногда специалисты-практики решают возвращать для каждого обучающего образца два выхода и интерпретируют их как вероятности каждого класса:  $P[\text{класс} = 0]$  против  $P[\text{класс} = 1]$ .

В такой ситуации для нормализации выходов (чтобы они давали в сумме 1) рекомендуется использовать многопеременную функцию (вместо логистической сигмоидальной функции) и категориальная перекрестная энтропия оказывается подходящей функцией потерь.

## Реализация глубокой сверточной нейронной сети с использованием TensorFlow

В главе 14 мы применяли оценщики TensorFlow для решения задач распознавания рукописных цифр с использованием API-интерфейсов TensorFlow разных уровней. Несложно вспомнить, что мы достигли почти 89%-ной правильности, применяя оценщик `DNNClassifier` с двумя скрытыми слоями.

Теперь давайте реализуем сеть CNN и посмотрим, сумеет ли она достичь лучшей прогнозирующей эффективности по сравнению с многослойным перцептроном (`DNNClassifier`) при классификации рукописных цифр. Обратите внимание, что полносвязные слои в главе 14 оказались способными хорошо справляться с данной задачей. Однако в ряде приложений, таких как чтение номеров банковских расчетных счетов, представленных рукописными цифрами, даже крошечные погрешности могут обходиться очень дорого. По этой причине критически важно максимально возможно сократить такую ошибку.

### Архитектура многослойной сверточной нейронной сети

Архитектура сети, которую мы собираемся реализовать, показана на рис. 15.12. Входами будут изображения  $28 \times 28$  пикселей в оттенках серого. Учитывая количество каналов (1 для изображений в оттенках серого) и пакет входных изображений, размерность входного тензора составит *размер пакета*  $\times 28 \times 28 \times 1$ .

Входные данные проходят через два сверточных слоя с размером ядра  $5 \times 5$ . Первый сверточный слой имеет 32 выходных карты признаков, а второй — 64 выходных карты признаков. За каждым сверточным слоем следует слой подвыборки в форме операции объединения по максимуму,  $P_{2 \times 2}$ . Затем полносвязный слой передает выход второму полносвязному слою, который действует как финальный *многопеременный* (*softmax*) выходной слой.



Рис. 15.12. Архитектура реализуемой сети

Ниже перечислены измерения тензоров в каждом слое:

- входной слой — [размер пакета  $\times 28 \times 28 \times 1$ ]
- первый сверточный слой (conv\_1) — [размер пакета  $\times 28 \times 28 \times 32$ ]
- первый объединяющий слой (pool\_1) — [размер пакета  $\times 14 \times 14 \times 32$ ]
- второй сверточный слой (conv\_2) — [размер пакета  $\times 14 \times 14 \times 64$ ]
- второй объединяющий слой (pool\_2) — [размер пакета  $\times 7 \times 7 \times 64$ ]
- первый полносвязный слой (fc\_1) — [размер пакета  $\times 1024$ ]
- второй полносвязный слой (fc\_2) и многопеременный слой — [размер пакета  $\times 10$ ]

Для сверточных ядер мы используем `strides=1`, так что размерность входа сохраняется в результирующих картах признаков. Для объединяющих слоев мы применяем `strides=2`, чтобы делать подвыборку изображения и сокращать размер выходных карт признаков. Мы реализуем эту сеть с применением API-интерфейса Keras библиотеки TensorFlow и API-интерфейса.

## Загрузка и предварительная обработка данных

В главе 13 вы ознакомились с двумя способами загрузки доступных наборов данных из модуля `tensorflow_datasets`. Один подход основан на трехшаговом процессе, а более простой метод предусматривает использование функции по имени `load`, которая является оболочкой для трех шагов. Здесь мы будем применять первый метод. Вот как выглядят три шага для загрузки набора данных MNIST:

```
>>> import tensorflow_datasets as tfds
>>> ## Загрузка данных
>>> mnist_bldr = tfds.builder('mnist')
>>> mnist_bldr.download_and_prepare()
>>> datasets = mnist_bldr.as_dataset(shuffle_files=False)
>>> mnist_train_orig = datasets['train']
>>> mnist_test_orig = datasets['test']
```

Набор данных MNIST поступает с заранее определенной схемой расщепления на обучающий и испытательный наборы, но мы также хотим создать проверочную часть из обучающей части. Обратите внимание, что в третьем шаге при вызове метода `.as_dataset()` мы используем необязательный аргумент `shuffle_files=False`. Такая установка препятствует начальному тасованию, что нам необходимо, поскольку мы желаем расщепить обучающий набор на две части: обучающий набор меньшего размера и проверочный набор. (Примечание: если начальное тасование не отключено, то это повлечет за собой повторное тасование набора данных при каждом извлечении мини-пакета данных. Можете проверить сказанное самостоятельно: когда начальное тасование включено, количество меток в проверочных наборах изменяется из-за повторного тасования расщеплений на обучающую и проверочную части. Результатом может оказаться ложная оценка эффективности модели, т.к. обучающий и проверочный наборы на самом деле будут смешанными.) Мы можем расщепить на обучающий и проверочный наборы следующим образом:

```
>>> BUFFER_SIZE = 10000
>>> BATCH_SIZE = 64
>>> NUM_EPOCHS = 20

>>> mnist_train = mnist_train_orig.map(
...     lambda item: (tf.cast(item['image'], tf.float32)/255.0,
...                     tf.cast(item['label'], tf.int32)))
>>> mnist_test = mnist_test_orig.map(
...     lambda item: (tf.cast(item['image'], tf.float32)/255.0,
...                     tf.cast(item['label'], tf.int32)))

>>> tf.random.set_seed(1)
>>> mnist_train = mnist_train.shuffle(buffer_size=BUFFER_SIZE,
...                                   reshuffle_each_iteration=False)

>>> mnist_valid = mnist_train.take(10000).batch(BATCH_SIZE)
>>> mnist_train = mnist_train.skip(10000).batch(BATCH_SIZE)
```

После подготовки набора данных мы готовы к реализации только что описанной сети CNN.

## Реализация сверточной нейронной сети с использованием API-интерфейса Keras библиотеки TensorFlow

Для реализации сети CNN в TensorFlow мы применяем `Sequential` из Keras, который позволит уложить стопкой различные слои — сверточный, объединяющий и отключения, а также полносвязные (плотные) слои. API-интерфейс `layers` в Keras предлагает классы для каждого слоя: `tf.keras.layers.Conv2D` для двумерного сверточного слоя, `tf.keras.layers.MaxPool2D` и `tf.keras.layers.AvgPool2D` для подвыборки (объединения по максимуму и объединения по среднему) плюс `tf.keras.layers.Dropout` для регуляризации с использованием отключения. Ниже мы детально исследуем каждый из перечисленных классов.

### Конфигурирование слоев сверточной нейронной сети в Keras

Конструирование слоя с помощью класса `Conv2D` требует указания количества выходных фильтров (которое эквивалентно числу выходных карт признаков) и размеров ядер.

Кроме того, есть необязательные параметры, которые можно применять для конфигурирования сверточного слоя. Самыми часто используемыми параметрами являются `strides` (со стандартным значением 1 по измерениям  $x$  и  $y$ ) и дополнение, которое может быть одинаковым или допустимым. Список добавочных параметров конфигурации приведен в официальной документации: [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/layers/Conv2D](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Conv2D).

Полезно упомянуть о том, что обычно при чтении изображения стандартным измерением для каналов будет последнее измерение тензорного массива. Формат называется NHWC, где  $N$  обозначает количество (number) изображений внутри пакета,  $H$  и  $W$  — высоту (height) и ширину (width), а  $C$  — каналы (channels).

Обратите внимание, что по умолчанию класс `Conv2D` допускает представление входов в формате NHWC. (Другие инструменты вроде PyTorch принимают формат NCHW.) Тем не менее, если вы столкнетесь с какими-то данными, каналы которых размещены в первом измерении (первое измерение после измерения пакета или второе измерение с учетом измерения пакета), тогда



вам понадобится поменять местами оси в данных, чтобы переместить каналы в последнее измерение. Альтернативный способ работы с входом в формате NCHW предусматривает установку `data_format="channels_first"`. После создания к слою можно обращаться, предоставляя четырехмерный тензор, первое измерение которого зарезервировано для пакета образцов; в зависимости от аргумента `data_format` второе или четвертое измерение соответствует каналу, а два оставшихся измерения являются пространственными.

Как показано в архитектуре модели CNN, которую мы хотим построить, за каждым сверточным слоем следует объединяющий слой для подвыборки (сокращения размера карт признаков). Классы `MaxPool2D` и `AvgPool2D` конструируют слои объединения по максимуму и по среднему. Аргумент `pool_size` определяет размер окна (или близлежащей области), которое будет использоваться для вычисления операций максимума или среднего. Кроме того, параметр `strides` может применяться для конфигурирования объединяющего слоя, как обсуждалось ранее.

Наконец, класс `Dropout` будет конструировать слой отключения для регуляризации с аргументом `rate`, в котором определяется вероятность отключения входных элементов во время обучения. Поведение при вызове слоя управляется посредством аргумента по имени `training`, в котором указывается, сделан ли вызов во время обучения или во время вывода.

## Конструирование сверточной нейронной сети в Keras

Теперь, когда вы ознакомились с этими классами, мы можем сконструировать модель CNN, представленную на рис. 15.12. В следующем коде мы будем использовать класс `Sequential` и добавим сверточные и объединяющие слои:

```
>>> model = tf.keras.Sequential()
>>> model.add(tf.keras.layers.Conv2D(
...     filters=32, kernel_size=(5, 5),
...     strides=(1, 1), padding='same',
...     data_format='channels_last',
...     name='conv_1', activation='relu'))
>>> model.add(tf.keras.layers.MaxPool2D(
...     pool_size=(2, 2), name='pool_1'))
>>> model.add(tf.keras.layers.Conv2D(
...     filters=64, kernel_size=(5, 5),
```

```

...         strides=(1, 1), padding='same',
...         name='conv_2', activation='relu'))
>>> model.add(tf.keras.layers.MaxPool2D(
...         pool_size=(2, 2), name='pool_2'))

```

Пока что мы добавили к модели два сверточных слоя, для каждого из которых указали ядро с размером  $5 \times 5$  и дополнением 'same'. Как обсуждалось ранее, применение `padding='same'` предохраняет пространственные измерения (вертикальное и горизонтальное) карт признаков, так что входы и выходы имеют те же самые высоту и ширину (может отличаться только число каналов в плане количества используемых фильтров). Слои объединения по максимуму с размером объединения  $2 \times 2$  и страйдами 2 вдвое сократят пространственные измерения. (Следует отметить, что если параметр `strides` в `MaxPool2D` не указан, то по умолчанию он устанавливается равным размеру объединения.)

Хотя на этой стадии мы можем рассчитать размер карт признаков вручную, API-интерфейс Keras предлагает для такой цели удобный метод:

```

>>> model.compute_output_shape(input_shape=(16, 28, 28, 1))
...                               TensorShape([16, 7, 7, 64])

```

После предоставления формы входа в виде кортежа метод `compute_output_shape` определяет, что выход должен иметь форму (16, 7, 7, 64), указывающую на карты признаков с 64 каналами и пространственным размером  $7 \times 7$ . Первое измерение соответствует измерению пакета, для которого мы произвольно выбрали 16. Взамен мы могли бы применить `None`, т.е. `input_shape=(None, 28, 28, 1)`.

Следующим мы добавим плотный (или полносвязный) слой для реализации классификатора поверх сверточных и объединяющих слоев. Его вход должен иметь ранг 2, т.е. форму *[размер пакета  $\times$  количество входных элементов]*. Таким образом, чтобы удовлетворить это требование для плотного слоя, нам необходимо выровнять выход предшествующих слоев:

```

>>> model.add(tf.keras.layers.Flatten())
>>> model.compute_output_shape(input_shape=(16, 28, 28, 1))
TensorShape([16, 3136])

```

Как показывает результат `compute_output_shape`, измерения входа для плотного слоя настроены корректно.

Далее мы добавляем два плотных слоя со слоем отключения между ними:

```
>>> model.add(tf.keras.layers.Dense(
...             units=1024, name='fc_1',
...             activation='relu'))
>>> model.add(tf.keras.layers.Dropout(
...             rate=0.5))
>>> model.add(tf.keras.layers.Dense(
...             units=10, name='fc_2',
...             activation='softmax'))
```

Последний полносвязный слой по имени 'fc\_2' имеет 10 выходных элементов для 10 меток классов в наборе данных MNIST. Кроме того мы используем многопеременную активацию, чтобы получить вероятности членства в классах каждого входного образца, исходя из предположения о том, что классы являются взаимоисключающими и потому вероятности для каждого образца в сумме дают 1. (Это означает, что обучающий образец может принадлежать только одному классу.) Исходя из того, что мы обсуждали в разделе “Функции потерь для классификации”, какая функция потерь здесь должна применяться? Вспомните, что для многоклассовой классификации с целочисленными (разреженными) метками (как противоположность меткам в унитарном коде) мы используем класс `SparseCategoricalCrossentropy`. В следующем коде вызывается метод `build()` для позднего создания переменных и компиляции модели:

```
>>> tf.random.set_seed(1)
>>> model.build(input_shape=(None, 28, 28, 1))
>>> model.compile(
...     optimizer=tf.keras.optimizers.Adam(),
...     loss=tf.keras.losses.SparseCategoricalCrossentropy(),
...     metrics=['accuracy'])
```



На заметку!

### Оптимизатор Adam

Обратите внимание, что в приведенной реализации для обучения модели CNN мы применяем класс `tf.keras.optimizers.Adam`. Оптимизатор Adam (adaptive moment estimation — адаптивная оценка момента) представляет собой надежный метод оптимизации на основе градиентов, подходящий для невыпуклых функций и задач МО.

Появлению Adam способствовали два популярных метода оптимизации: RMSProp и AdaGrad.

Ключевое преимущество Adam связано с выбором размера шага обновления, выводимого из скользящего среднего моментов градиента. Оптимизатор Adam более подробно описан в работе Дидерика Кингма и Джимми Ба “Adam: A Method for Stochastic Optimization” (Adam: метод стохастической оптимизации), которая свободно доступна по ссылке <https://arxiv.org/abs/1412.6980>.

Как вы уже знаете, мы можем обучить модель, вызвав метод `fit()`. Важно отметить, что при использовании назначенных методов для обучения и оценки (наподобие `evaluate()` и `predict()`) автоматически устанавливается режим для слоя отключения, а скрытые элементы надлежащим образом масштабируются, не заставляя нас беспокоиться об этом. Затем мы обучим полученную модель CNN и задействуем созданный проверочный набор для наблюдения за процессом обучения:

```
>>> history = model.fit(mnist_train, epochs=NUM_EPOCHS,
...                     validation_data=mnist_valid,
...                     shuffle=True)
Epoch 1/20
782/782 [=====] - 35s 45ms/step - loss:
0.1450 - accuracy: 0.8882 - val_loss: 0.0000e+00 - val_accuracy:
0.0000e+00
Epoch 2/20
782/782 [=====] - 34s 43ms/step - loss:
0.0472 - accuracy: 0.9833 - val_loss: 0.0507 - val_accuracy: 0.9839
..
Epoch 20/20
782/782 [=====] - 34s 44ms/step - loss:
0.0047 - accuracy: 0.9985 - val_loss: 0.0488 - val_accuracy: 0.9920
```

После окончания 20 эпох обучения мы можем визуализировать кривые обучения (рис. 15.13):

```
>>> import matplotlib.pyplot as plt
>>> hist = history.history
>>> x_arr = np.arange(len(hist['loss'])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
```

```

>>> ax.plot(x_arr, hist['loss'], '-o', label='Потеря при обучении')
>>> ax.plot(x_arr, hist['val_loss'], '--<',
...         label='Потеря при проверке')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Эпоха', size=15)
>>> ax.set_ylabel('Потеря', size=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist['accuracy'], '-o',
...         label='Правильность при обучении')
>>> ax.plot(x_arr, hist['val_accuracy'], '--<',
...         label='Правильность при проверке')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Эпоха', size=15)
>>> ax.set_ylabel('Правильность', size=15)
>>> plt.show()

```

Из двух предшествующих глав вам уже известно, что оценка обученной модели на испытательном наборе может делаться вызовом метода `.evaluate()`:

```

>>> test_results = model.evaluate(mnist_test.batch(20))
>>> print('Правильность при испытании: {:.2f}%'.
...       format(test_results[1]*100))
Правильность при испытании: 99.39%

```

Модель CNN достигает правильности 99.39%. Вспомните, что в главе 14 мы добились приблизительно 90%-ной правильности с применением оценщика `DNNClassifier`.

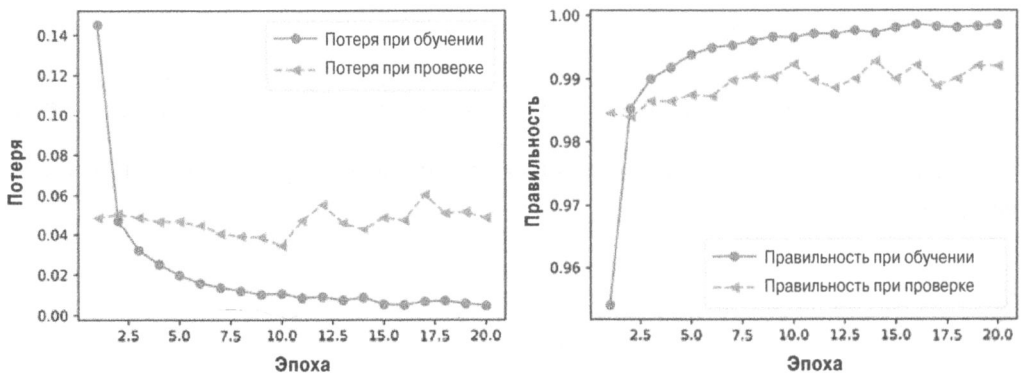


Рис. 15.13. Кривые обучения после окончания 20 эпох

Наконец, мы можем получить результаты прогнозирования в форме вероятностей членства в классах и преобразовать их в спрогнозированные метки, используя функцию `tf.argmax` для нахождения элемента с максимальной вероятностью. Мы сделаем это для пакета из 12 образцов и визуализируем входы и спрогнозированные метки:

```
>>> batch_test = next(iter(mnist_test.batch(12)))
>>> preds = model(batch_test[0])
>>> tf.print(preds.shape)
TensorShape([12, 10])
>>> preds = tf.argmax(preds, axis=1)
>>> print(preds)
tf.Tensor([6 2 3 7 2 2 3 4 7 6 6 9], shape=(12,), dtype=int64)
>>> fig = plt.figure(figsize=(12, 4))
>>> for i in range(12):
...     ax = fig.add_subplot(2, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     img = batch_test[0][i, :, :, 0]
...     ax.imshow(img, cmap='gray_r')
...     ax.text(0.9, 0.1, '{}'.format(preds[i]),
...             size=15, color='blue',
...             horizontalalignment='center',
...             verticalalignment='center',
...             transform=ax.transAxes)
>>> plt.show()
```

На рис. 15.14 показаны входы в виде рукописных цифр и спрогнозированные метки.

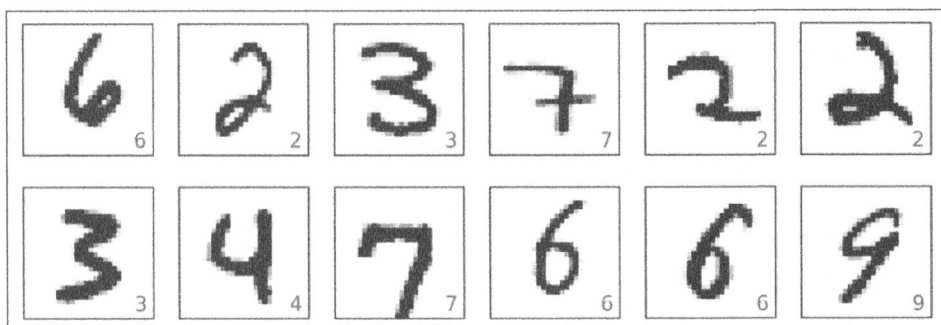


Рис. 15.14. Входы и спрогнозированные метки

В представленных на рис. 15.14 образцах все спрогнозированные метки корректны. Задача отображения неправильно классифицированных цифр, как мы поступали в главе 12, оставлена в качестве упражнения для самостоятельного выполнения.

## Классификация полов по изображениям лиц с использованием сверточной нейронной сети

В текущем разделе мы собираемся реализовать сеть CNN для классификации полов по изображениям лиц с применением набора данных CelebA. Как было показано в главе 13, набор данных CelebA содержит 202 599 изображений лиц знаменитостей. Вдобавок для каждого изображения доступны 40 двоичных атрибутов лица, включая пол (мужской или женский) и возраст (молодой или старый).

Базируясь на том, что вы узнали до сих пор, цель этого раздела заключается в том, чтобы построить и обучить модель CNN для прогнозирования атрибута пола по изображениям лиц. Ради простоты мы будем использовать небольшую порцию обучающих данных (16 000 обучающих образцов) для ускорения процесса обучения. Однако для повышения эффективности обобщения и снижения вероятности переобучения на таком небольшом наборе данных мы применим прием под названием *дополнение данных*.

### Загрузка набора данных CelebA

Первым делом давайте загрузим данные аналогично тому, как поступали в предыдущем разделе для набора данных MNIST. Данные CelebA состоят из трех частей: обучающего набора, проверочного набора и испытательного набора. Затем мы реализуем простую функцию для подсчета количества образцов в каждой части:

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> celeba_bldr = tfds.builder('celeb_a')
>>> celeba_bldr.download_and_prepare()
>>> celeba = celeba_bldr.as_dataset(shuffle_files=False)

>>> celeba_train = celeba['train']
>>> celeba_valid = celeba['validation']
>>> celeba_test = celeba['test']
>>>
```

```
>>> def count_items(ds):  
...     n = 0  
...     for _ in ds:  
...         n += 1  
...     return n  
  
>>> print('Обучающий набор: {}'.format(count_items(celeba_train)))  
Обучающий набор: 162770  
  
>>> print('Проверочный набор: {}'.format(count_items(celeba_valid)))  
Проверочный набор: 19867  
  
>>> print('Испытательный набор: {}'.format(count_items(celeba_test)))  
Испытательный набор: 19962
```

Таким образом, вместо использования всех доступных обучающих и проверочных данных мы возьмем поднабор из 16 000 обучающих образцов и 1 000 образцов для проверки:

```
>>> celeba_train = celeba_train.take(16000)  
>>> celeba_valid = celeba_valid.take(1000)  
>>> print('Обучающий набор: {}'.format(count_items(celeba_train)))  
Обучающий набор: 16000  
  
>>> print('Проверочный набор: {}'.format(count_items(celeba_valid)))  
Проверочный набор: 1000
```

Важно отметить, что если аргумент `shuffle_files` в вызове метода `celeba_bldr.as_dataset()` не был установлен в `False`, то мы по-прежнему будем видеть 16 000 образцов в обучающем наборе и 1 000 образцов в проверочном наборе. Тем не менее, в таком случае на каждой итерации обучающие данные тасуются заново, после чего берется новый набор из 16 000 образцов. В итоге это свело бы на нет нашу цель, которая заключается в том, чтобы намеренно обучить модель с помощью небольшого набора данных. Далее мы обсудим дополнение данных как методику подъема эффективности глубоких нейронных сетей.

## Трансформация изображений и дополнение данных

Дополнение данных объединяет в себе широкий спектр приемов для обработки сценариев, где обучающие данные ограничены. Например, некоторые приемы дополнения данных позволяют нам модифицировать или даже искусственно синтезировать добавочные данные и тем самым повышать эффективность работы машины либо модели ГО за счет сокращения



степени переобучения. Хотя дополнение данных предназначено не только для данных изображений, существует набор трансформаций, единственно применимых к данным изображений, в число которых входят обрезка частей изображения, переворачивание, изменение контрастности, яркости и насыщенности. Давайте взглянем на ряд таких трансформаций, доступных посредством модуля `tf.image`. В приведенном ниже блоке кода мы извлекаем пять образцов из набора данных `celeba_train` и применяем к ним пять типов трансформации: 1) обрезка изображения до граничной рамки, 2) переворачивание изображения по горизонтали, 3) подстройка контрастности, 4) подстройка яркости и 5) обрезка изображения относительно центра и приведение результирующего изображения к исходному размеру (218, 178). Результаты визуализируются в отдельных колонках для сравнения.

```
>>> import matplotlib.pyplot as plt
>>> # извлечь 5 образцов
>>> examples = []
>>> for example in celeba_train.take(5):
...     examples.append(example['image'])
>>> fig = plt.figure(figsize=(16, 8.5))

>>> ## Колонка 1: обрезка до граничной рамки
>>> ax = fig.add_subplot(2, 5, 1)
>>> ax.set_title('Обрезка\ndo граничной рамки', size=15)
>>> ax.imshow(examples[0])
>>> ax = fig.add_subplot(2, 5, 6)
>>> img_cropped = tf.image.crop_to_bounding_box(
...     examples[0], 50, 20, 128, 128)
>>> ax.imshow(img_cropped)

>>> ## Колонка 2: переворачивание (по горизонтали)
>>> ax = fig.add_subplot(2, 5, 2)
>>> ax.set_title('Переворачивание (по горизонтали)', size=15)
>>> ax.imshow(examples[1])
>>> ax = fig.add_subplot(2, 5, 7)
>>> img_flipped = tf.image.flip_left_right(examples[1])
>>> ax.imshow(img_flipped)

>>> ## Колонка 3: подстройка контрастности
>>> ax = fig.add_subplot(2, 5, 3)
>>> ax.set_title('Подстройка контрастности', size=15)
>>> ax.imshow(examples[2])
>>> ax = fig.add_subplot(2, 5, 8)
```

```

>>> img_adj_contrast = tf.image.adjust_contrast(
...     examples[2], contrast_factor=2)
>>> ax.imshow(img_adj_contrast)

>>> ## Колонка 4: подстройка яркости
>>> ax = fig.add_subplot(2, 5, 4)
>>> ax.set_title('Подстройка яркости', size=15)
>>> ax.imshow(examples[3])
>>> ax = fig.add_subplot(2, 5, 9)
>>> img_adj_brightness = tf.image.adjust_brightness(
...     examples[3], delta=0.3)
>>> ax.imshow(img_adj_brightness)

>>> ## Колонка 5: обрезка относительно центра
>>> ax = fig.add_subplot(2, 5, 5)
>>> ax.set_title('Обрезка относительно центра\н изменение размера',
...             size=15)
>>> ax.imshow(examples[4])
>>> ax = fig.add_subplot(2, 5, 10)
>>> img_center_crop = tf.image.central_crop(
...     examples[4], 0.7)
>>> img_resized = tf.image.resize(
...     img_center_crop, size=(218, 178))
>>> ax.imshow(img_resized.numpy().astype('uint8'))
>>> plt.show()

```

Результаты представлены на рис. 15.15.



Рис. 15.15. Результаты применения пяти разных трансформаций

На рис. 15.15 в первой строке показаны исходные изображения, а во второй — их трансформированные версии. Обратите внимание, что для первой трансформации (крайняя слева колонка) граничная рамка задана четырьмя числами: координаты верхнего левого угла рамки (здесь  $x = 20$ ,  $y = 50$ ), а также ширина и высота рамки (ширина = 128, высота = 128). Кроме того, началом отсчета (координатами в позиции, обозначенной как  $(0, 0)$ ) для изображений, загруженных с помощью TensorFlow (равно как и другими пакетами вроде `imageio`), является верхний левый угол изображения.

Трансформации в предыдущем блоке кода детерминированы. Однако все трансформации подобного рода также можно рандомизировать, что рекомендуется делать для дополнения данных во время обучения модели. Скажем, изображение можно обрезать до случайной граничной рамки (когда координаты верхнего левого угла выбираются случайно). Изображение можно случайным образом переворачивать по горизонтали или вертикали с вероятностью 0.5. Контрастность изображения можно случайно изменять, где `contrast_factor` выбирается случайным образом из диапазона значений с равномерным распределением. Вдобавок можно создать конвейер этих трансформаций.

Например, мы можем сначала случайным образом обрезать изображение, затем случайно его перевернуть и в заключение привести к желательному размеру. Вот как выглядит код (поскольку мы имеем дело со случайными элементами, то в целях воспроизводимости устанавливаем начальное случайное значение):

```
>>> tf.random.set_seed(1)
>>> fig = plt.figure(figsize=(14, 12))
>>> for i, example in enumerate(celeba_train.take(3)):
...     image = example['image']
...
...     ax = fig.add_subplot(3, 4, i*4+1)
...     ax.imshow(image)
...     if i == 0:
...         ax.set_title('Оригинал', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+2)
...     img_crop = tf.image.random_crop(image, size=(178, 178, 3))
...     ax.imshow(img_crop)
...     if i == 0:
...         ax.set_title('Шаг 1: случайная обрезка', size=15)
...
... 
```

```

... ax = fig.add_subplot(3, 4, i*4+3)
... img_flip = tf.image.random_flip_left_right(img_crop)
... ax.imshow(tf.cast(img_flip, tf.uint8))
... if i == 0:
...     ax.set_title('Шаг 2: случайное переворачивание', size=15)
...
... ax = fig.add_subplot(3, 4, i*4+4)
... img_resize = tf.image.resize(img_flip, size=(128, 128))
... ax.imshow(tf.cast(img_resize, tf.uint8))
... if i == 0:
...     ax.set_title('Шаг 3: изменение размера', size=15)
>>> plt.show()

```

Результаты применения случайных трансформаций можно видеть на рис. 15.16.

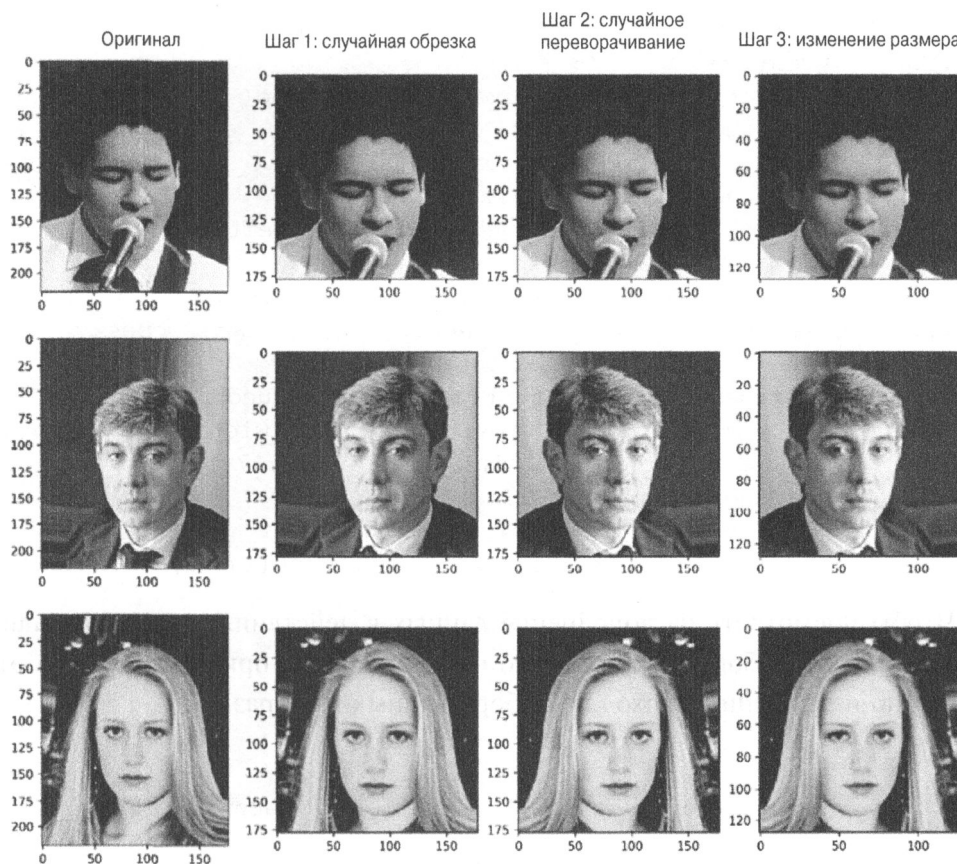


Рис. 15.16. Результаты применения случайных трансформаций

Обратите внимание, что из-за случайных трансформаций при каждом проходе по трем образцам мы получаем слегка отличающиеся изображения.

Ради удобства мы можем определить функцию-оболочку для использования такого конвейера с целью дополнения данных во время обучения модели. В показанном ниже коде мы определим функцию `preprocess()`, которая будет принимать словарь, содержащий ключи `'image'` и `'attributes'`, а возвращать кортеж с трансформированным изображением и меткой, извлеченной из словаря атрибутов.

Тем не менее, мы будем применять дополнение данных только к обучающим образцам, но не к проверочным или испытательным. Вот необходимый код:

```
>>> def preprocess(example, size=(64, 64), mode='train'):
...     image = example['image']
...     label = example['attributes']['Male']
...     if mode == 'train':
...         image_cropped = tf.image.random_crop(
...             image, size=(178, 178, 3))
...         image_resized = tf.image.resize(
...             image_cropped, size=size)
...         image_flip = tf.image.random_flip_left_right(
...             image_resized)
...         return image_flip/255.0, tf.cast(label, tf.int32)
...     else: # для не обучающих данных использовать обрезку
...         # по центру, а не случайную обрезку
...         image_cropped = tf.image.crop_to_bounding_box(
...             image, offset_height=20, offset_width=0,
...             target_height=178, target_width=178)
...         image_resized = tf.image.resize(
...             image_cropped, size=size)
...         return image_resized/255.0, tf.cast(label, tf.int32)
```

Чтобы посмотреть на дополнение данных в действии, давайте создадим небольшой поднабор из обучающего набора данных, применим к нему эту функцию и выполним проход по набору данных пять раз:

```
>>> tf.random.set_seed(1)
>>> ds = celeba_train.shuffle(1000, reshuffle_each_iteration=False)
>>> ds = ds.take(2).repeat(5)
>>> ds = ds.map(lambda x:preprocess(x, size=(178, 178), mode='train'))
>>> fig = plt.figure(figsize=(15, 6))
```

```
>>> for j, example in enumerate(ds):
...     ax = fig.add_subplot(2, 5, j//2+(j%2)*5+1)
...     ax.set_xticks([])
...     ax.set_yticks([])
...     ax.imshow(example[0])
>>> plt.show()
```

На рис. 15.17 представлены пять результирующих трансформаций для дополнения данных на двух примерах изображений.



*Рис. 15.17. Дополнение данных в действии*

Далее мы применим к обучающему и проверочному наборам созданную функцию предварительной обработки. Мы будем использовать размер изображения (64, 64). Кроме того, мы укажем `mode='train'` при работе с обучающими данными и `mode='eval'` при работе с проверочными данными, чтобы случайные элементы конвейера дополнения данных применялись только к обучающим данным:

```
>>> import numpy as np
>>> BATCH_SIZE = 32
>>> BUFFER_SIZE = 1000
>>> IMAGE_SIZE = (64, 64)
>>> steps_per_epoch = np.ceil(16000/BATCH_SIZE)
>>> ds_train = celeba_train.map(
...     lambda x: preprocess(x, size=IMAGE_SIZE, mode='train'))
>>> ds_train = ds_train.shuffle(buffer_size=BUFFER_SIZE).repeat()
>>> ds_train = ds_train.batch(BATCH_SIZE)
>>> ds_valid = celeba_valid.map(
...     lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval'))
>>> ds_valid = ds_valid.batch(BATCH_SIZE)
```

## Обучение классификатора полов, основанного на сверточной нейронной сети

К настоящему времени построение модели с помощью API-интерфейса Keras библиотеки TensorFlow и ее обучение не должны вызывать вопросов. По замыслу наша модель CNN получает входные изображения размера  $64 \times 64 \times 3$  (изображения имеют три цветовых канала и формат 'channels\_last').

Входные данные проходят через четыре сверточных слоя для создания 32, 64, 128 и 256 карт признаков с использованием фильтров с ядром размера  $3 \times 3$ . За первыми тремя сверточными слоями следует слой объединения по максимуму,  $P_{2 \times 2}$ . Вдобавок предусмотрено два слоя отключения для регуляризации:

```
>>> model = tf.keras.Sequential([
...     tf.keras.layers.Conv2D(
...         32, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...     tf.keras.layers.Dropout(rate=0.5),
...
...     tf.keras.layers.Conv2D(
...         64, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...     tf.keras.layers.Dropout(rate=0.5),
...
...     tf.keras.layers.Conv2D(
...         128, (3, 3), padding='same', activation='relu'),
...     tf.keras.layers.MaxPooling2D((2, 2)),
...
...     tf.keras.layers.Conv2D(
...         256, (3, 3), padding='same', activation='relu')
... ])
```

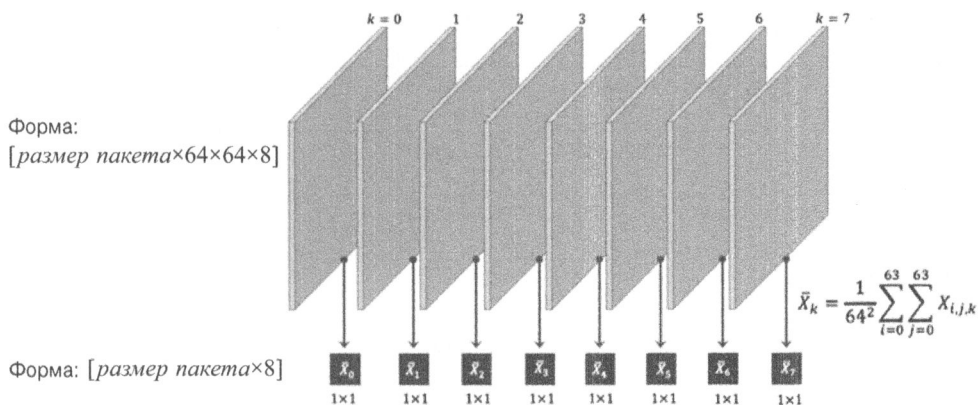
Давайте выясним форму выходных карт признаков после применения указанных слоев:

```
>>> model.compute_output_shape(input_shape=(None, 64, 64, 3))
TensorShape([None, 8, 8, 256])
```

Существуют 256 карт признаков (или каналов) размера  $8 \times 8$ . Теперь мы можем добавить полносвязный слой, чтобы получить выходной слой с единственным элементом. Если мы изменим форму карт признаков (выров-

нием их), тогда количество входных элементов в этом полносвязном слое составит  $8 \times 8 \times 256 = 16\,384$ . В качестве альтернативы мы возьмем новый слой под названием *слой объединения по глобальному среднему*, который рассчитывает среднее каждой карты признаков по отдельности, тем самым сокращая количество скрытых элементов до 256. Затем мы можем добавить полносвязный слой. Несмотря на то что мы явно не обсуждали слои объединения по глобальному среднему, концептуально они очень похожи на другие объединяющие слои. Фактически объединение по глобальному среднему можно рассматривать как особый случай объединения по среднему, когда размер объединения равен размеру входных карт признаков.

Для лучшего понимания на рис. 15.18 показан пример входных карт признаков формы  $[\text{размер пакета} \times 64 \times 64 \times 8]$ . Каналы пронумерованы как  $k = 0, 1, \dots, 7$ . Операция объединения по глобальному среднему рассчитывает среднее каждого канала, так что выход будет иметь форму  $[\text{размер пакета} \times 8]$ . (Примечание: класс `GlobalAveragePooling2D` в API-интерфейсе Keras автоматически сжимает выход. Без сжатия выхода форма выглядела бы как  $[\text{размер пакета} \times 1 \times 1 \times 8]$ , потому что объединение по глобальному среднему сократит пространственное измерение  $64 \times 64$  до  $1 \times 1$ .)



**Рис. 15.18.** Пример входных карт признаков формы  $[\text{размер пакета} \times 64 \times 64 \times 8]$

Следовательно, с учетом того, что в нашем случае форма карт признаков перед этим слоем имеет вид  $[\text{размер пакета} \times 8 \times 8 \times 256]$ , мы ожидаем получить в качестве выхода 256 элементов, т.е. формой выхода будет  $[\text{размер пакета} \times 256]$ . Давайте добавим такой слой и заново рассчитаем форму выхода, удостоверившись в том, что утверждение верно:



```
>>> model.add(tf.keras.layers.GlobalAveragePooling2D())
>>> model.compute_output_shape(input_shape=(None, 64, 64, 3))
TensorShape([None, 256])
```

Наконец, мы можем добавить полносвязный (плотный) слой для получения единственного выходного элемента. В данном случае мы можем указать функцию активации 'sigmoid' или просто использовать `activation=None`, так что модель будет выдавать логиты (вместо вероятностей членства в классах), которые предпочтительнее при обучении модели в TensorFlow и Keras из-за своей численной устойчивости, как обсуждалось ранее:

```
>>> model.add(tf.keras.layers.Dense(1, activation=None))
>>> tf.random.set_seed(1)
>>> model.build(input_shape=(None, 64, 64, 3))
>>> model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
=====		
conv2d (Conv2D)	multiple	896
max_pooling2d (MaxPooling2D)	multiple	0
dropout (Dropout)	multiple	0
conv2d_1 (Conv2D)	multiple	18496
max_pooling2d_1 (MaxPooling2	multiple	0
dropout_1 (Dropout)	multiple	0
conv2d_2 (Conv2D)	multiple	73856
max_pooling2d_2 (MaxPooling2	multiple	0
conv2d_3 (Conv2D)	multiple	295168
global_average_pooling2d (Gl	multiple	0
dense (Dense)	multiple	257
=====		
Total params: 388,673		
Trainable params: 388,673		
Non-trainable params: 0		

Следующий шаг — компиляция модели, но теперь мы должны решить, какую функцию потерь применять. Мы имеем двоичную классификацию с одиночным выходным элементом, что означает необходимость в использовании класса `BinaryCrossentropy`. Кроме того, поскольку наш последний слой не применяет сигмоидальную активацию (мы используем `activation=None`), выходами модели являются логиты, не вероятности. Таким образом, мы будем указывать в `BinaryCrossentropy` также и `from_logits=True`, чтобы функция потерь внутренне применяла сигмоидальную функцию, которая благодаря своему лежащему в основе коду эффективнее, чем делать это вручную. Вот код для компиляции и обучения модели:

```
>>> model.compile(optimizer=tf.keras.optimizers.Adam(),
...               loss=
...               tf.keras.losses.BinaryCrossentropy(from_logits=True),
...               metrics=['accuracy'])
>>> history = model.fit(ds_train, validation_data=ds_valid,
...                     epochs=20,
...                     steps_per_epoch=steps_per_epoch)
```

Давайте визуализируем кривую обучения и сравним потерю и правильность при обучении и при проверке после каждой эпохи:

```
>>> hist = history.history
>>> x_arr = np.arange(len(hist['loss'])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist['loss'], '-o', label='Потеря при обучении')
>>> ax.plot(x_arr, hist['val_loss'], '--<', label='Потеря при проверке')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Эпоха', size=15)
>>> ax.set_ylabel('Потеря', size=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist['accuracy'], '-o',
...         label='Правильность при обучении')
>>> ax.plot(x_arr, hist['val_accuracy'], '--<',
...         label='Правильность при проверке')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Эпоха', size=15)
>>> ax.set_ylabel('Правильность', size=15)
>>> plt.show()
```

На рис. 15.19 показаны графики потери и правильности при обучении и при проверке.

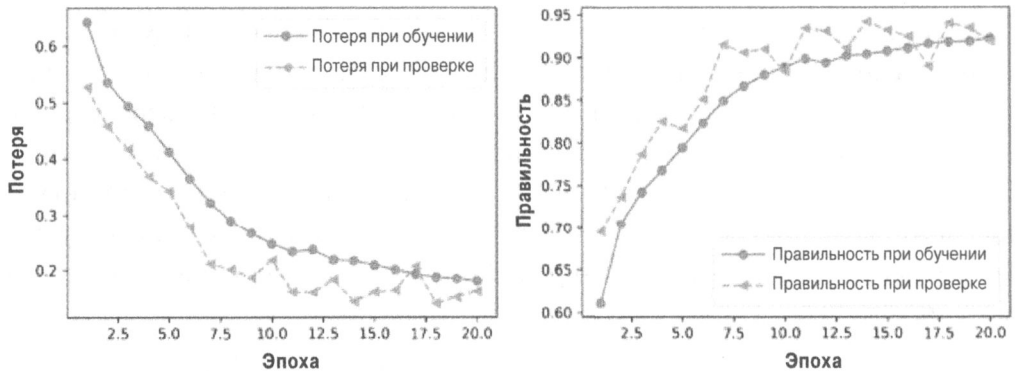


Рис. 15.19. Потеря и правильность при обучении и при проверке

По кривым обучения несложно заметить, что потери при обучении и при проверке не сходятся в области плато. Базируясь на таком результате, мы могли бы продолжить обучение в течение еще нескольких эпох. Используя метод `fit()`, мы можем продолжить обучение для дополнительных 10 эпох:

```
>>> history = model.fit(ds_train, validation_data=ds_valid,
...                       epochs=30, initial_epoch=20,
...                       steps_per_epoch=steps_per_epoch)
```

После того, как кривые обучения начнут нас устраивать, мы можем оценить модель на удерживаемом испытательном наборе:

```
>>> ds_test = celeba_test.map(
...     lambda x: preprocess(x, size=IMAGE_SIZE, mode='eval')).batch(32)
>>> test_results = model.evaluate(ds_test)
>>> print('Правильность при испытании: {:.2f}%')
...     .format(test_results[1]*100))
Правильность при испытании: 94.75%
```

Итак, мы готовы получить результаты прогнозирования на ряде испытательных образцов с применением метода `model.predict()`. Однако вспомните, что модель выдает логиты, не вероятности. Если нас интересуют вероятности членства в классах для этой двоичной задачи с единственным выходным элементом, тогда мы можем использовать функцию `tf.sigmoid` для вычисления вероятности класса 1. (В случае многоклассовой задачи мы применяли функцию `tf.math.softmax`.) В приведенном

ниже коде мы будем брать небольшой поднабор из 10 образцов из предварительно обработанного испытательного набора данных (`ds_test`) и запускать `model.predict()` для получения логитов. Далее для каждого образца мы рассчитаем вероятность его принадлежности к классу 1 (который соответствует мужскому полу, основываясь на метках в наборе данных CelebA) и визуализируем образцы вместе с *достоверными метками* (“ДМ”) и *спрогнозированными вероятностями* (“Вер”). Обратите внимание, что перед извлечением 10 образцов мы применяем `unbatch()` к набору данных `ds_test`, иначе метод `take()` возвратил бы 10 пакетов размера 32, а не 10 индивидуальных образцов:

```
>>> ds = ds_test.unbatch().take(10)
>>> pred_logits = model.predict(ds.batch(10))
>>> probas = tf.sigmoid(pred_logits)
>>> probas = probas.numpy().flatten()*100
>>> fig = plt.figure(figsize=(15, 7))
>>> for j, example in enumerate(ds):
...     ax = fig.add_subplot(2, 5, j+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(example[0])
...     if example[1].numpy() == 1:
...         label='мужской'
...     else:
...         label = 'женский'
...     ax.text(
...         0.5, -0.15, 'ДМ: {:s}\nВер (мужской)={:.0f}%'
...         ''.format(label, probas[j]),
...         size=16,
...         horizontalalignment='center',
...         verticalalignment='center',
...         transform=ax.transAxes)
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 15.20 показаны 10 примеров изображений вместе с достоверными метками и вероятностями их принадлежности к классу 1 (мужской).

Под каждым изображением выводятся вероятности принадлежности к классу 1 (т.е. мужскому полу в соответствии с набором данных CelebA). Как видите, наша обученная модель совершила только одну ошибку в выбранном наборе из 10 испытательных образцов.



**Рис. 15.20.** Десять примеров изображений с их достоверными метками (“ДМ”) и спрогнозированными вероятностями (“Вер”)

В качестве дополнительного упражнения мы рекомендуем попытаться задействовать полный обучающий набор данных, а не созданный ранее небольшой поднабор. Кроме того, вы можете заменить или подкорректировать архитектуру CNN. Скажем, вы можете изменить вероятности отключения и количество фильтров в различных сверточных слоях. Вы также могли бы заменить слой объединения по глобальному среднему плотным слоем. Если вы будете использовать полный обучающий набор данных с моделью на основе CNN, обученной в этой главе, то должны достичь примерно 97–99%-ной точности.

## Резюме

В настоящей главе вы узнали о сетях CNN и их главных компонентах. Мы начали с операции свертки и взглянули на одномерные и двумерные реализации. Затем мы рассмотрели еще один тип слоев, который можно обнаружить в ряде распространенных архитектур CNN: слои подвыборки, также называемые объединяющими слоями. Мы в основном уделяли внимание двум самым часто применяемым формам объединения: по максимуму и по среднему.

Далее, собрав все индивидуальные концепции вместе, мы реализовали глубокие сверточные нейронные сети, используя API-интерфейс Keras библиотеки TensorFlow. Первая сеть применялась к уже знакомой задаче распознавания рукописных цифр из набора данных MNIST.

Затем мы реализовали вторую сеть CNN на более сложном наборе данных, содержащем изображения лиц, и обучили ее для классификации полов. Попутно мы также исследовали дополнение данных и трансформации, которые можно применять к изображениям лиц с использованием класса Dataset из TensorFlow.

В следующей главе мы перейдем к рассмотрению *рекуррентных нейронных сетей* (*recurrent neural network* — RNN). Сети RNN применяются для выяснения структуры последовательных данных и с ними связано несколько восхитительных приложений, включая перевод с одного языка на другой и подписание изображений.



# МОДЕЛИРОВАНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ РЕКУРРЕНТНЫХ НЕЙРОННЫХ СЕТЕЙ

В предыдущей главе наше внимание было сосредоточено на *сверточных нейронных сетях (CNN)*. Мы рассмотрели строительные блоки архитектур CNN и выяснили, каким образом реализовывать глубокие сверточные нейронные сети в TensorFlow. Наконец, мы показали, как использовать сети CNN для классификации изображений. В настоящей главе мы будем исследовать *рекуррентные нейронные сети (recurrent neural network — RNN)*, а также их применение в моделировании последовательных данных.

Мы раскроем в главе следующие темы:

- понятие последовательных данных;
- сети RNN для моделирования последовательностей;
- долгая краткосрочная память (*long short-term memory — LSTM*);
- укороченное обратное распространение во времени (*truncated back-propagation through time — T-BPTT*);



- реализация многослойной сети RNN для моделирования последовательностей в TensorFlow;
- проект номер один — сеть RNN для смыслового анализа набора данных с рецензиями на фильмы IMDb;
- проект номер два — сеть RNN для моделирования языка на уровне символов с ячейками LSTM, использующая текстовые данные из романа Жюль Верна “Таинственный остров”;
- применение *отсечения градиентов* (*gradient clipping*) во избежание проблемы взрывного роста градиентов;
- понятие модели *преобразователей* (*Transformer*) и механизма *самовнимания* (*self-attention mechanism*).

## Понятие последовательных данных

Давайте начнем обсуждение сетей RNN с выяснения природы последовательных данных, которые больше известны как *последовательности*. Мы взглянем на уникальные характеристики последовательностей, отличающие их от данных других видов. Затем мы рассмотрим, каким образом можно представлять последовательные данные, и исследуем разнообразные категории моделей для последовательных данных, которые основаны на входе и выходе модели. Такие сведения помогут нам выяснить взаимосвязь между сетями RNN и последовательностями позже в главе.

## Моделирование последовательных данных — вопросы порядка

Характерная черта последовательностей, которая делает их уникальными, отличающимися от остальных типов данных, связана с тем, что элементы в последовательности расположены в определенном порядке и не являются независимыми друг от друга. Типичные алгоритмы МО с учителем предполагают, что входные данные *независимы и одинаково распределены*, т.е. обучающие образцы *взаимно независимы* и имеют одно и то же внутреннее распределение. В этом отношении, исходя из предположения взаимной независимости, порядок предоставления обучающих образцов модели не играет роли. Например, при наличии выборки из  $n$  обучающих образцов,  $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(n)}$ , порядок их использования для обучения алгоритма МО не имеет значения. Примером сценария подобного рода может служить набор данных Iris, с ко-

торым мы работали ранее. В наборе данных Iris каждый цветок измерялся независимо, а измерения одного цветка не влияют на измерения другого цветка.

Однако такое допущение перестает быть действительным, когда мы имеем дело с последовательностями — по определению порядок в них имеет значение. Примером такого сценария может быть прогнозирование рыночной стоимости конкретной акции. Скажем, пусть у нас есть выборка из  $n$  обучающих образцов, причем каждый обучающий образец представляет рыночную стоимость определенной акции в специфический день. Если наша задача заключается в прогнозировании биржевой стоимости акции на следующие три дня, тогда имеет смысл рассмотреть предыдущие курсы акции в отсортированном по дате порядке, чтобы вывести тенденцию, а не потреблять имеющиеся обучающие образцы в случайном порядке.



### Последовательные данные или данные временных рядов

Данные временных рядов являются особым типом последовательных данных, где каждый образец ассоциирован с измерением для времени. В данных временных рядов выборки производятся в следующие друг за другом отметки времени, а потому измерение времени определяет порядок в точках данных. Например, курсы акций и записи голоса или речи представляют собой данные временных рядов.

С другой стороны, не все последовательные данные имеют измерение времени, скажем, текстовые данные или цепочки ДНК, в которых образцы упорядочены, но они не квалифицируются как данные временных рядов. Далее в главе мы приведем несколько примеров *обработки естественного языка (natural language processing — NLP)* и моделирования текста, которые не являются данными временных рядов, но имейте в виду, что сети RNN могут применяться также для данных временных рядов.

## Представление последовательностей

Мы признали, что в последовательных данных порядок в точках данных важен, поэтому далее нам нужно найти способ использования в своих интересах информации упорядочивания в модели МО. Повсюду в главе мы будем представлять последовательности как  $\langle \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)} \rangle$ . Надстрочные индексы указывают порядок следования образцов, а  $T$  — это длина после-

довательности. В рамках практического примера последовательностей рассмотрим данные временного ряда, где каждая точка данных  $x^{(t)}$  относится к определенному моменту времени  $t$ . На рис. 16.1 показан пример данных временного ряда, в котором входные признаки ( $x$ ) и целевые метки ( $y$ ) естественным образом соблюдают порядок согласно оси времени; таким образом,  $x$  и  $y$  являются последовательностями.

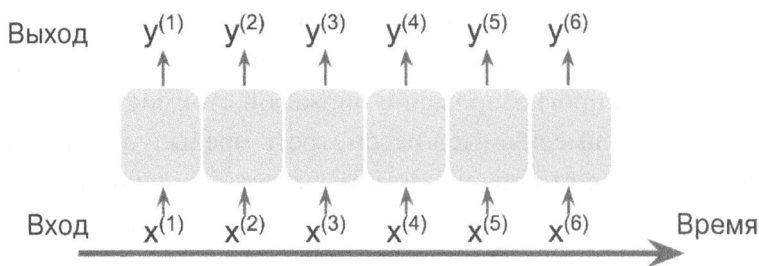


Рис. 16.1. Пример данных временного ряда

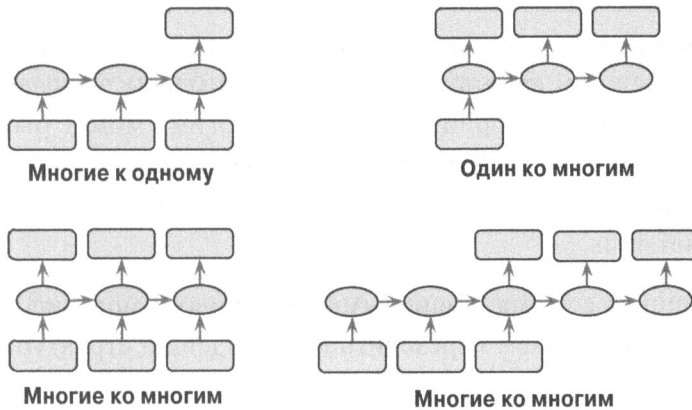
Как уже упоминалось, рассмотренные до сих пор стандартные модели на основе нейронных сетей, подобные многослойному персептрону (MLP) и сверточным нейронным сетям (CNN) для данных изображений, делают допущение о независимости обучающих образцов друг от друга и потому не содержат в себе *информацию упорядочивания*. Можно сказать, что модели подобного рода не имеют *памяти* для ранее встречавшихся обучающих образцов. Например, образцы проходят через шаги прямого и обратного распространения, а веса обновляются независимо от порядка, в котором обрабатывались обучающие образцы.

Напротив, сети RNN спроектированы для моделирования последовательностей и способны запоминать прошлую информацию, а также соответствующим образом обрабатывать новые события, что считается очевидным преимуществом при работе с последовательными данными.

## Категории моделирования последовательностей

Моделирование последовательностей имеет много замечательных приложений, включая перевод с одного языка на другой (скажем, с английского на немецкий язык), подписание изображений и порождение текста. Тем не менее, для выбора подходящей архитектуры и подхода мы должны освоить и быть в состоянии проводить различие между типами задач моделирова-

ния последовательностей. На рис. 16.2 продемонстрированы наиболее распространенные задачи моделирования последовательностей, зависящие от категорий отношений между входными и выходными данными. Диаграммы основаны на объяснениях из великолепной статьи “The Unreasonable Effectiveness of Recurrent Neural Networks” (Необоснованная эффективность рекуррентных нейронных сетей), которая написана Андреем Карпатом и свободно доступна по ссылке <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.



**Рис. 16.2.** Наиболее распространенные задачи моделирования последовательностей

Давайте подробнее обсудим различные категории отношений между входными и выходными данными, изображенные на рис. 16.2. Если ни входные, ни выходные данные не представляют последовательности, тогда мы имеем дело со стандартными данными и для их моделирования можем просто применять многослойный персептрон (или другую классификационную модель из числа рассмотренных в книге ранее). Но если либо вход, либо выход является последовательностью, то задача моделирования данных, вероятно, попадет в одну из описанных ниже трех категорий.

- **Многие к одному.** Входные данные представляют собой последовательность, но выход является вектором фиксированного размера или скаляром, а не последовательностью. Скажем, в смысловом анализе вход основан на тексте (например, рецензия на фильм) и выходом будет метка класса (например, метка, обозначающая то, понравился ли фильм рецензенту).

- **Один ко многим.** Входные данные имеют стандартный формат и не являются последовательностью, но выход представляет собой последовательность. Примером такой категории служит подписание изображений — на входе подается изображение, а на выходе получается фраза на английском языке, подводящая итог по содержимому изображения.
- **Многие ко многим.** Входные и выходные массивы являются последовательностями. Категория “многие ко многим” может дополнительно разделяться на основе того, синхронизированы ли вход и выход. В качестве примера синхронизированной задачи моделирования вида “многие ко многим” можно назвать классификацию видеороликов, когда помечается каждый кадр в видеоролике. Примером *отсроченной* задачи моделирования категории “многие ко многим” может быть перевод с одного языка на другой. Скажем, машина должна прочитать и обработать полную фразу на английском, прежде чем выдать ее перевод на немецкий язык.

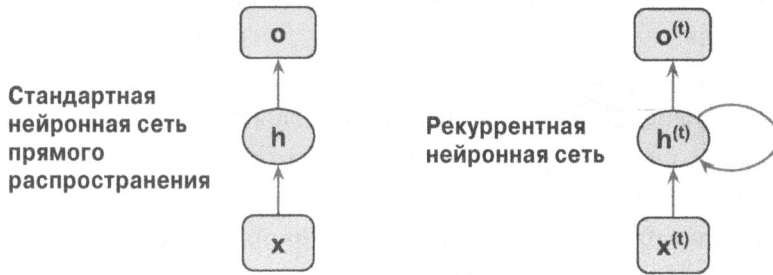
Ознакомившись с тремя обширными категориями моделирования последовательностей, мы можем переходить к обсуждению структуры сети RNN.

## Рекуррентные нейронные сети для моделирования последовательностей

Прежде чем заняться реализацией сетей RNN в TensorFlow, в этом разделе мы рассмотрим основы сетей RNN. Мы начнем с представления типовой структуры RNN, которая включает рекурсивный компонент для моделирования данных последовательностей. Затем мы исследуем вычисление активации нейронов в сети RNN. В результате будет создан контекст для обсуждения распространенных проблем при обучении сетей RNN и их решений, таких как *долгая краткосрочная память (LSTM)* и *управляемый рекуррентный блок (gated recurrent unit — GRU)*.

### Механизм организации циклов рекуррентной нейронной сети

Первым делом давайте взглянем на архитектуру сети RNN. На рис. 16.3 показаны стандартная нейронная сеть прямого распространения и сеть RNN рядом друг с другом в целях сравнения.



**Рис. 16.3.** Стандартная нейронная сеть прямого распространения и сеть RNN

Обе сети имеют только один скрытый слой. В представлении на рис. 16.3 элементы не показаны, но мы предполагаем, что входной слой ( $x$ ), скрытый слой ( $h$ ) и выходной слой ( $o$ ) являются векторами, содержащими множество элементов.



На заметку!

### Определение типа выхода из сети RNN

Такая обобщенная архитектура RNN могла бы соответствовать двум категориям моделирования последовательностей, где входом является последовательность. Обычно рекуррентный слой может возвращать на выходе последовательность  $\langle o^{(0)}, o^{(1)}, \dots, o^{(T)} \rangle$  или просто возвращать последний выход (при  $t = T$ , т.е.  $o^{(T)}$ ). Таким образом, он мог бы относиться к одной из категорий “многие ко многим” или к категории “многие к одному”, например, если в качестве финального выхода мы использовали только последний элемент  $o^{(T)}$ .

Как вы увидите позже, в API-интерфейсе Keras библиотеки TensorFlow поведение рекуррентного слоя в плане возвращения на выходе последовательности или просто применения последнего выхода может быть указано путем установки аргумента `return_sequences` соответственно в `True` или в `False`.

В стандартной нейронной сети прямого распространения информация проходит от входа до скрытого слоя и затем от скрытого слоя до выходного слоя. С другой стороны, в сети RNN скрытый слой получает информацию от входного слоя текущего временного шага и от скрытого слоя предыдущего временного шага.

Поток информации в смежных временных шагах внутри скрытого слоя позволяет сети иметь память о прошедших событиях. Такой поток информа-

ции обычно отображается в виде цикла, известного в графической системе обозначений как *рекуррентное ребро* (*recurrent edge*), которое и дало название общей архитектуре RNN.

Подобно многослойным персептронам сети RNN могут содержать множество скрытых слоев. Обратите внимание, что сеть RNN с одним скрытым слоем принято называть *однослойной рекуррентной нейронной сетью*, которую не следует путать с однослойными нейронными сетями без скрытого слоя, такими как Adaline или логистическая регрессия. На рис. 16.4 показана сеть RNN с одним скрытым слоем (вверху) и сеть RNN с двумя скрытыми слоями (внизу).

Для исследования архитектуры сетей RNN и потока информации компактное представление рекуррентного ребра может быть развернуто, как проиллюстрировано на рис. 16.4.

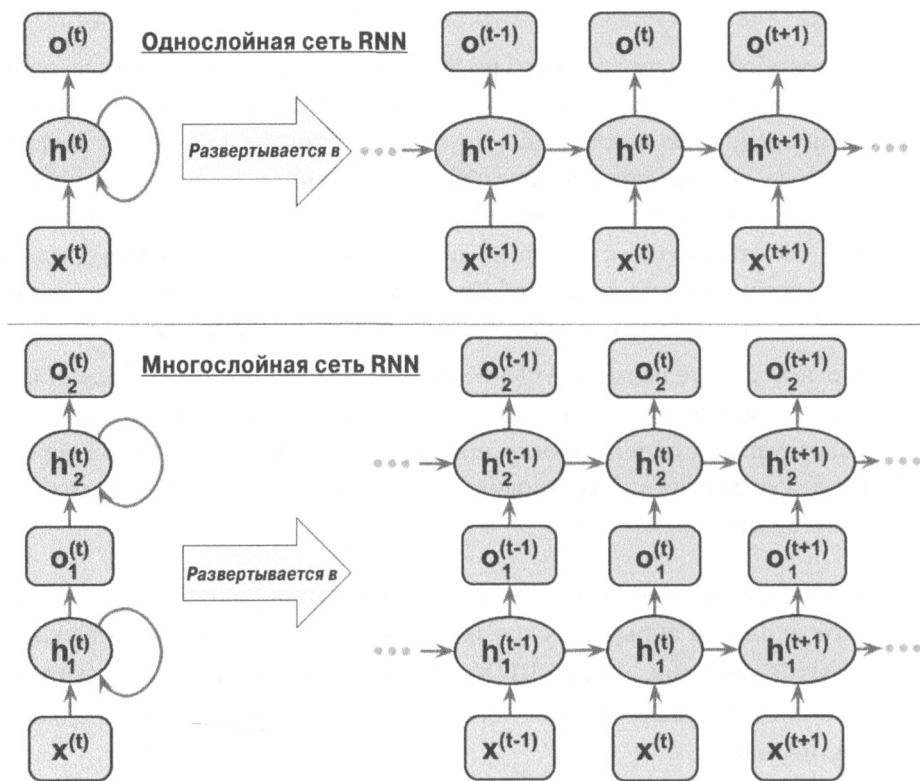


Рис. 16.4. Сети RNN с одним и двумя скрытыми слоями

Ранее мы уже указывали, что каждый скрытый слой в стандартной нейронной сети принимает только один вход — общую предварительную активацию, ассоциированную с входным слоем. Напротив, скрытый слой в сети RNN принимает два *отдельных* набора входов — предварительную активацию из входного слоя и активацию этого же скрытого слоя из предыдущего временного шага  $t - 1$ .

На первом временном шаге,  $t = 0$ , скрытые элементы инициализируются нулями или небольшими случайными значениями. Затем на временном шаге при  $t > 0$  скрытые элементы получают свой вход от точки данных в текущий момент времени  $x^{(t)}$  и предыдущие значения скрытых элементов из шага  $t - 1$ , обозначаемые как  $h^{(t-1)}$ .

Аналогичным образом мы можем подытожить поток информации в случае многослойной сети RNN:

- слой 1 — скрытый слой, представленный как  $h_1^{(t)}$ , получает свой вход от точки данных  $x^{(t)}$  и значения собственных скрытых элементов из предыдущего временного шага  $h_1^{(t-1)}$ ;
- слой 2 — второй скрытый слой, представленный как  $h_2^{(t)}$ , получает свой вход от скрытых элементов нижележащего слоя на текущем временном шаге ( $h_1^{(t)}$ ) и значения собственных скрытых элементов из предыдущего временного шага  $h_2^{(t-1)}$ .

Поскольку в этом случае каждый рекуррентный слой в качестве входа должен получать последовательность, все рекуррентные слои кроме последнего обязаны *возвращать на выходе последовательность* (т.е. `return_sequences=True`). Поведение последнего рекуррентного слоя зависит от типа задачи.

## Вычисление активаций в сети RNN

Теперь, когда вы понимаете структуру и общий поток информации в сети RNN, давайте займемся более специфичными вопросами и рассчитаем действительные активации скрытых слоев и выходного слоя. Ради простоты мы предположим, что имеется только один скрытый слой, но та же самая концепция применима к многослойным сетям RNN.

С каждым ориентированным ребром (связью между блоками) в показанном на рис. 16.4 представлении сети RNN ассоциируется матрица весов. Веса не зависят от времени  $t$ ; следовательно, они разделяются по временной оси.



В однослойной сети RNN различают следующие матрицы весов:

- $\mathbf{W}_{xh}$  — матрица весов между входом  $\mathbf{x}^{(t)}$  и скрытым слоем  $\mathbf{h}$ ;
- $\mathbf{W}_{hh}$  — матрица весов, ассоциированная с рекуррентным ребром;
- $\mathbf{W}_{ho}$  — матрица весов между скрытым слоем и выходным слоем.

Перечисленные матрицы весов изображены на рис. 16.5.

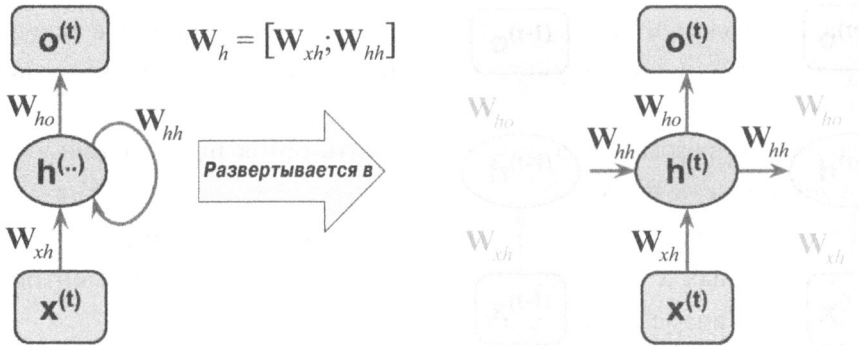


Рис. 16.5. Различные матрицы весов

В определенных реализациях можно заметить, что матрицы весов  $\mathbf{W}_{xh}$  и  $\mathbf{W}_{hh}$  объединены в комбинированную матрицу  $\mathbf{W}_h = [\mathbf{W}_{xh}; \mathbf{W}_{hh}]$ . Позже в этом разделе мы задействуем и такую запись.

Расчет активаций в сети RNN очень похож на вычисление активаций в стандартном многослойном персептроне и других типах нейронных сетей прямого распространения. Общий вход  $\mathbf{z}_h$  (предварительная активация) для скрытого слоя рассчитывается посредством линейной комбинации, т.е. мы вычисляем сумму произведений матриц весов и соответствующих векторов и добавляем член смещения:

$$\mathbf{z}_h^{(t)} = \mathbf{W}_{xh} \mathbf{x}^{(t)} + \mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{b}_h$$

Затем вычисляются активации скрытых элементов на временном шаге  $t$ :

$$\mathbf{h}^{(t)} = \phi_h \left( \mathbf{z}_h^{(t)} \right) = \phi_h \left( \mathbf{W}_{xh} \mathbf{x}^{(t)} + \mathbf{W}_{hh} \mathbf{h}^{(t-1)} + \mathbf{b}_h \right)$$

Здесь  $\mathbf{b}_h$  — вектор смещений для скрытых элементов, а  $\phi_h(\cdot)$  — функция активации скрытого слоя.

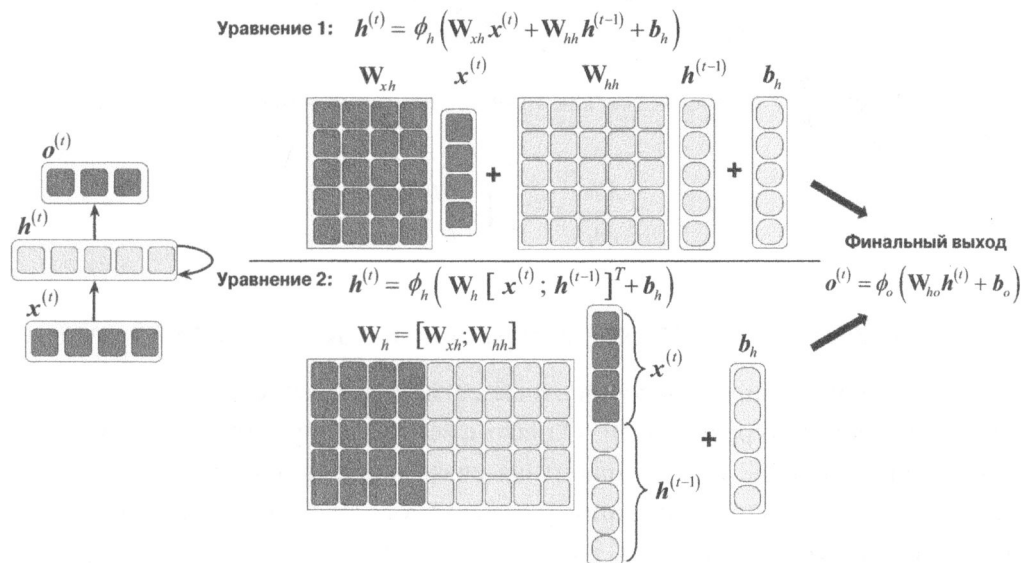
Если желательно использовать объединенную матрицу весов  $\mathbf{W}_h = [\mathbf{W}_{xh}; \mathbf{W}_{hh}]$ , тогда уравнение для вычисления активаций скрытых элементов изменится следующим образом:

$$\mathbf{h}^{(t)} = \phi_h \left( [\mathbf{W}_{xh}; \mathbf{W}_{hh}] \begin{bmatrix} \mathbf{x}^{(t)} \\ \mathbf{h}^{(t-1)} \end{bmatrix} + \mathbf{b}_h \right)$$

После расчета активаций скрытых элементов для текущего временного шага можно вычислить активации выходных элементов:

$$\mathbf{o}^{(t)} = \phi_o \left( \mathbf{W}_{ho} \mathbf{h}^{(t)} + \mathbf{b}_o \right)$$

На рис. 16.6 приведена иллюстрация процесса расчета активаций по двум уравнениям, которая поможет его прояснить.



**Рис. 16.6.** Вычисление активаций скрытых элементов с помощью двух уравнений



## Обучение сетей RNN с применением обратного распространения во времени (backpropagation through time — BPTT)

Алгоритм обучения для сетей RNN был представлен в 1990-х годах: “Backpropagation Through Time: What It Does and How to Do It” (Обратное распространение во времени: что оно делает и как его делать), Пол Вербос, *Proceedings of IEEE*, 78(10): стр. 1550–1560).

Выведение градиентов может показаться несколько сложным, но основная идея в том, что общая потеря  $L$  представляет собой сумму всех функций потерь на временных шагах от  $t = 1$  до  $t = T$ :

$$L = \sum_{t=1}^T L^{(t)}$$

Поскольку потери за время  $1:t$  зависят от скрытых элементов на всех предшествующих временных шагах  $1:t$ , градиент будет вычисляться следующим образом:

$$\frac{\partial L^{(t)}}{\partial \mathbf{W}_{hh}} = \frac{\partial L^{(t)}}{\partial \mathbf{y}^{(t)}} \times \frac{\partial \mathbf{y}^{(t)}}{\partial \mathbf{h}^{(t)}} \times \left( \sum_{k=1}^t \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} \times \frac{\partial \mathbf{h}^{(k)}}{\partial \mathbf{W}_{hh}} \right)$$

Здесь  $\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}}$  вычисляется как произведение смежных временных шагов:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{i=k+1}^t \frac{\partial \mathbf{h}^{(i)}}{\partial \mathbf{h}^{(i-1)}}$$

## Рекуррентность скрытого слоя или рекуррентность выходного слоя

До сих пор вы видели рекуррентные сети, в которых скрытый слой обладал свойством рекуррентности. Однако имейте в виду, что существует альтернативная модель, где рекуррентная связь поступает из выходного слоя. В таком случае общие активации из выходного слоя на предыдущем временном шаге,  $\mathbf{o}^{t-1}$ , могут быть добавлены одним из двух способов:

- к скрытому слою на текущем временном шаге,  $\mathbf{h}^t$  (рекуррентность “выходной к скрытому” на рис. 16.7);
- к выходному слою на текущем временном шаге,  $\mathbf{o}^t$  (рекуррентность “выходной к выходному” на рис. 16.7).

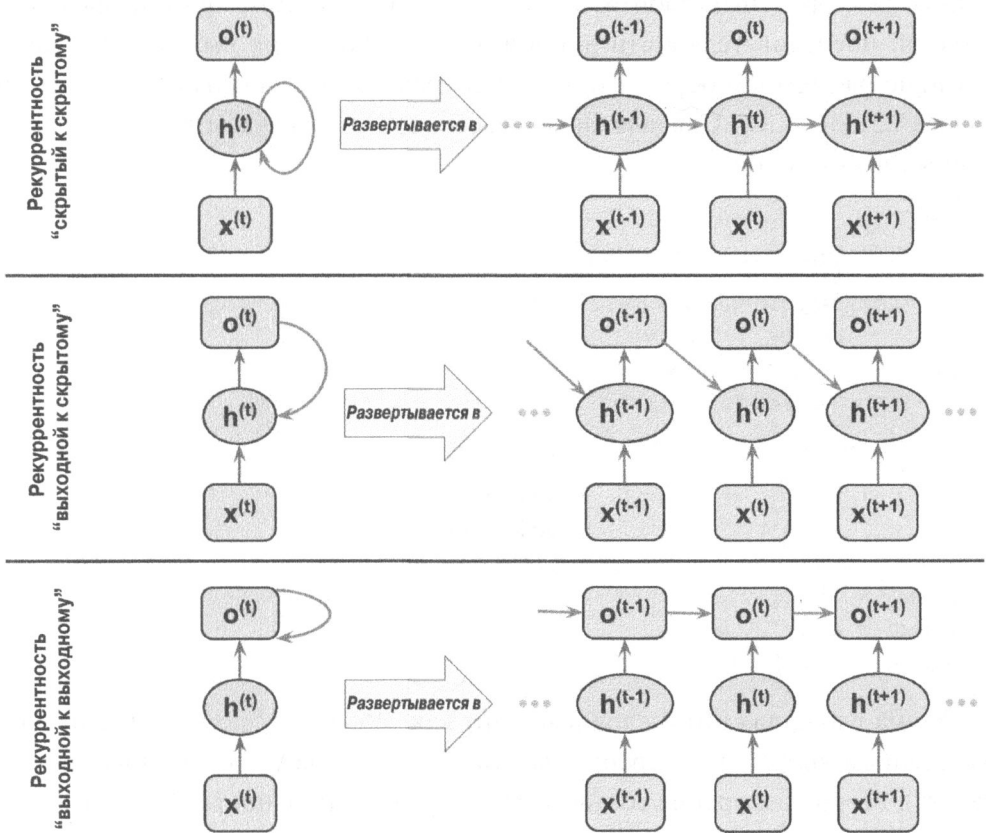


Рис. 16.7. Виды рекуррентности

Как демонстрируется на рис. 16.7, отличия между архитектурами хорошо видны в рекуррентных связях. В соответствии с нашей системой обозначений веса, ассоциированные с рекуррентной связью, будут обозначаться с помощью  $W_{hh}$  для рекуррентности "скрытый к скрытому",  $W_{oh}$  для рекуррентности "выходной к скрытому" и  $W_{oo}$  для рекуррентности "выходной к выходному". В ряде статей веса, ассоциированные с рекуррентными связями, также обозначаются как  $W_{rec}$ .

Чтобы посмотреть, как все работает на практике, давайте вручную рассчитаем прямой проход для одного из указанных типов рекуррентности. В рамках API-интерфейса Keras библиотеки TensorFlow рекуррентный слой может быть определен посредством класса SimpleRNN, который похож на рекуррентность "выходной к выходному". В показанном ниже коде мы со-

сделаем рекуррентный слой из SimpleRNN и выполним прямой проход на входной последовательности длиной 3 для вычисления выхода. Мы также вручную рассчитаем обратный проход и сравним результаты с результатами класса SimpleRNN. Сначала мы создаем слой и назначаем веса для наших ручных вычислений:

```
>>> import tensorflow as tf
>>> tf.random.set_seed(1)

>>> rnn_layer = tf.keras.layers.SimpleRNN(
...     units=2, use_bias=True,
...     return_sequences=True)
>>> rnn_layer.build(input_shape=(None, None, 5))

>>> w_xh, w_oo, b_h = rnn_layer.weights

>>> print('Форма W_xh:', w_xh.shape)
>>> print('Форма W_oo:', w_oo.shape)
>>> print('Форма b_h:', b_h.shape)
Форма W_xh: (5, 2)
Форма W_oo: (2, 2)
Форма b_h: (2,)
```

Форма входа для этого слоя выглядит как  $(None, None, 5)$ , где первым измерением является измерение пакета ( $None$  означает переменный размер пакета), второе измерение соответствует последовательности ( $None$  означает переменную длину последовательности) и последнее измерение относится к признакам. Обратите внимание, что мы устанавливаем `return_sequences=True`, что для входной последовательности длиной 3 в результате даст выходную последовательность  $\langle o^{(0)}, o^{(1)}, o^{(2)} \rangle$ . Иначе возвращался бы только финальный выход,  $o^{(2)}$ .

Теперь мы иницилируем прямой проход слоя `rnn_layer` и вручную рассчитаем выходы на каждом временном шаге, после чего сравним их:

```
>>> x_seq = tf.convert_to_tensor(
...     [[1.0]*5, [2.0]*5, [3.0]*5],
...     dtype=tf.float32)

>>> ## выход SimpleRNN:
>>> output = rnn_layer(tf.reshape(x_seq, shape=(1, 3, 5)))

>>> ## ручной расчет выхода:
>>> out_man = []
>>> for t in range(len(x_seq)):
```

```

...     xt = tf.reshape(x_seq[t], (1, 5))
...     print('Временной шаг {} =>'.format(t))
...     print('    Вход          : ', xt.numpy())
...
...     ht = tf.matmul(xt, w_xh) + b_h
...     print('    Скрытый слой   : ', ht.numpy())
...
...     if t>0:
...         prev_o = out_man[t-1]
...     else:
...         prev_o = tf.zeros(shape=(ht.shape))
...     ot = ht + tf.matmul(prev_o, w_oo)
...     ot = tf.math.tanh(ot)
...     out_man.append(ot)
...     print('    Выход (вручную): ', ot.numpy())
...     print('    Выход SimpleRNN: '.format(t),
...           output[0][t].numpy())
...     print()

```

Временной шаг 0 =>

```

Вход          : [[1. 1. 1. 1. 1.]]
Скрытый слой   : [[0.41464037 0.96012145]]
Выход (вручную): [[0.39240566 0.74433106]]
Выход SimpleRNN: [0.39240566 0.74433106]

```

Временной шаг 1 =>

```

Вход          : [[2. 2. 2. 2. 2.]]
Скрытый слой   : [[0.82928073 1.9202429 ]]
Выход (вручную): [[0.80116504 0.9912947 ]]
Выход SimpleRNN: [0.80116504 0.9912947 ]

```

Временной шаг 2 =>

```

Вход          : [[3. 3. 3. 3. 3.]]
Скрытый слой   : [[1.243921  2.8803642]]
Выход (вручную): [[0.95468265 0.9993069 ]]
Выход SimpleRNN: [0.95468265 0.9993069 ]

```

При ручном расчете прямого прохода мы использовали функцию активации в виде гиперболического тангенса ( $\tanh$ ), т.к. она же применяется в SimpleRNN (стандартная активация). По выведенным результатам видно, что выходы из ручного расчета прямого прохода в точности совпадают с выходом слоя SimpleRNN на каждом временном шаге. Надеемся, рассмотренная практическая задача пролила свет на тайны рекуррентных сетей.

## Сложности изучения долгосрочных взаимодействий

Кратко упомянутое выше обратное распространение во времени (BPTT) привносит ряд новых сложностей.

Из-за наличия множителя  $\frac{\partial h^{(t)}}{\partial h^{(k)}}$  в расчете градиентов функции потерь возникает так называемая проблемы *исчезновения* (*vanishing*) и *взрывного роста* (*exploding*) градиентов. Указанные проблемы объясняются с помощью примеров на рис. 16.8, где представлена сеть RNN, для простоты имеющая только один скрытый элемент.

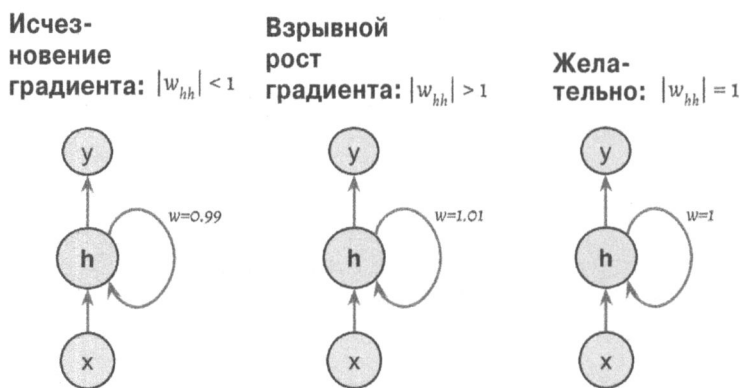


Рис. 16.8. Проблемы исчезновения и взрывного роста градиентов

По существу  $\frac{\partial h^{(t)}}{\partial h^{(k)}}$  имеет  $t - k$  умножений, вследствие чего умножение  $t - k$  раз веса  $w$  дает множитель  $w^{t-k}$ . Если  $|w| < 1$ , то множитель  $w^{t-k}$  становится очень малым при большом  $t - k$ . С другой стороны, если вес рекуррентного ребра  $|w| > 1$ , тогда  $w^{t-k}$  становится очень большим при большом  $t - k$ . Следует отметить, что большое  $t - k$  относится к долгосрочным зависимостям. Несложно заметить, что наивного решения, позволяющего избежать исчезновения или взрывного роста градиента, можно было бы достичь за счет обеспечения  $|w| = 1$ . В случае заинтересованности в более детальном исследовании данного вопроса почитайте статью Р. Паскану, Т. Миколова и Й. Бенджи “On the difficulty of training recurrent neural networks” (О сложности обучения рекуррентных нейронных сетей), которая свободно доступна по ссылке <https://arxiv.org/pdf/1211.5063.pdf>.

На практике существуют, по меньшей мере, три решения проблем исчезновения и взрывного роста градиентов:

- отсечение градиентов;
- укороченное обратное распространение во времени (*T-BPTT*);
- долгая краткосрочная память (*long short-term memory* — *LSTM*).

При отсечении градиентов мы указываем отсекающее или пороговое значение для градиентов и присваиваем это отсекающее значение вместо значений градиентов, которые превышают отсекающее значение. Напротив, метод *TBPTT* просто ограничивает количество временных шагов, в течение которых сигнал может обратно распространяться после каждого прямого прохода. Например, даже если последовательность имеет 100 элементов или шагов, то мы можем обратно распространять лишь самые последние 20 временных шагов.

Хотя методы отсечения градиентов и *TBPTT* способны решить проблему взрывного роста градиентов, усечение ограничивает количество шагов, через которые градиент может фактически протекать обратно и надлежащим образом обновлять веса. С другой стороны, метод *LSTM*, разработанный в 1997 году Сеппом Хохрайтером и Юргеном Шмидхубером, был более успешен в решении проблем исчезновения и взрывного роста градиентов наряду с тем, что моделировал долгосрочные последовательности за счет использования ячеек памяти. Давайте обсудим метод *LSTM* более подробно.

## Ячейки долгой краткосрочной памяти

Первоначально элементы *LSTM* были предложены в качестве способа преодоления проблемы с исчезновением градиентов (“Long Short-Term Memory” (Долгая краткосрочная память), С. Хохрайтер и Ю. Шмидхубер, *Neural Computation*, 9(8): с. 1735–1780 (1997 г.)). Строительным блоком элемента *LSTM* является *ячейка памяти*, которая по существу представляет или заменяет скрытый слой стандартных сетей *RNN*.

В каждой ячейке памяти имеется рекуррентное ребро с упомянутым ранее желательным весом  $w = 1$  для преодоления проблем исчезновения и взрывного роста градиентов. Значения, ассоциированные с этим рекуррентным ребром, вместе называются *состоянием ячейки*. Развернутая структура современной ячейки *LSTM* показана на рис. 16.9.



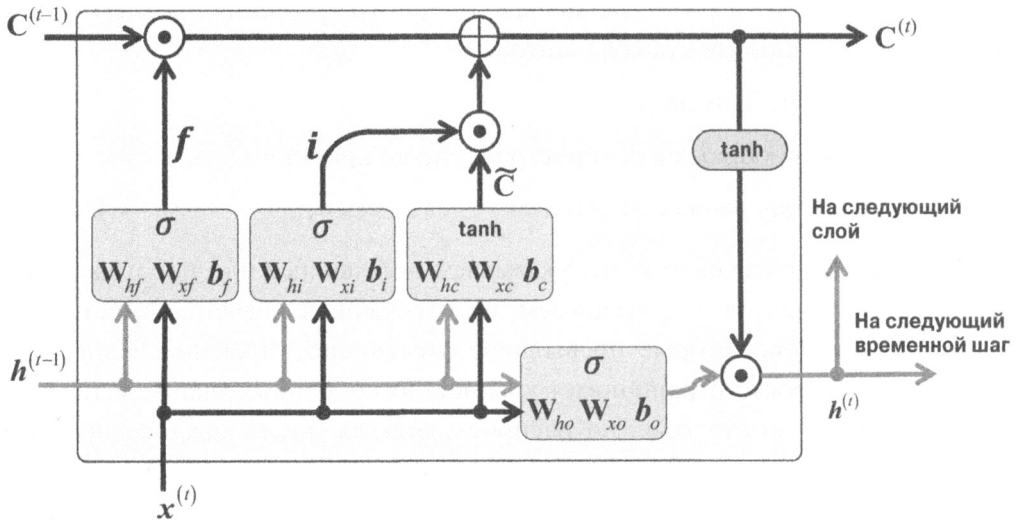


Рис. 16.9. Структура современной ячейки LSTM

Обратите внимание, что состояние ячейки из предыдущего временного шага,  $C^{(t-1)}$ , модифицируется для получения состояния ячейки на текущем временном шаге,  $C^{(t)}$ , без прямого перемножения с каким-либо весовым множителем. Поток информации в этой ячейке памяти управляется несколькими вычислительными элементами (часто называемыми *шлюзами* (*gate*)), которые мы опишем далее. На рис. 16.9 знаком  $\odot$  обозначено *поэлементное произведение* (поэлементное умножение), а знаком  $\oplus$  — *поэлементное суммирование* (поэлементное сложение). Кроме того,  $x^{(t)}$  относится к входным данным в момент времени  $t$ , а  $h^{(t-1)}$  означает скрытые элементы в момент времени  $t-1$ . Четыре прямоугольника указываются с функцией активации, либо сигмоидальной функцией ( $\sigma$ ), либо функцией гиперболического тангенса ( $\tanh$ ), и набором весов; такие прямоугольники применяют линейную комбинацию, выполняя на своих входах умножения матриц на векторы ( $h^{(t-1)}$  и  $x^{(t)}$ ). Вычислительные элементы с сигмоидальными функциями активации, чьи выходные элементы проходят через  $\odot$ , называются шлюзами.

В ячейке LSTM различают три типа шлюзов, известные как *шлюз забывания* (*forget gate*), *входной шлюз* (*input gate*) и *выходной шлюз* (*output gate*).

- *Шлюз забывания* ( $f$ ) позволяет ячейке памяти сбрасывать свое состояние, не возрастая до бесконечности. Фактически шлюз забывания решает, какой информации проходить разрешено, а какой запрещено.

Ниже приведено уравнение для вычисления  $f_t$ :

$$f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f)$$

Важно отметить, что шлюз забывания не был частью первоначальной ячейки LSTM; его добавили несколько лет спустя, чтобы усовершенствовать исходную модель (“Learning to Forget: Continual Prediction with LSTM”) (Изучить, чтобы забыть: непрерывное прогнозирование с помощью LSTM), Ф. Герс, Ю. Шмидхубер и Ф. Камминс, *Neural Computation* 12, с. 2451–2471 (2000 г.)).

- *Входной шлюз* ( $i_t$ ) и *значение-кандидат* ( $\tilde{C}$ ) отвечают за обновление состояния ячейки. Они вычисляются так:

$$i_t = \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i)$$

$$\tilde{C}_t = \tanh(W_{xg}x^{(t)} + W_{hg}h^{(t-1)} + b_c)$$

А вот как вычисляется состояние ячейки в момент времени  $t$ :

$$C^{(t)} = (C^{(t-1)} \odot f_t) \oplus (i_t \odot \tilde{C}_t)$$

- *Выходной шлюз* ( $o_t$ ) решает, как обновлять значения скрытых элементов:

$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o)$$

При таких определениях скрытые элементы на текущем временном шаге вычисляются следующим образом:

$$h^{(t)} = o_t \odot \tanh(C^{(t)})$$

Структура ячейки LSTM и лежащие в основе вычисления могут показаться очень сложными и трудными в реализации. Однако есть и хорошая новость: в библиотеке TensorFlow уже реализованы оптимизированные функции-оболочки, которые позволяют легко и эффективно определять ячейки LSTM. Мы будем применять сети RNN и ячейки LSTM в реальных наборах данных позже в главе.



## Другие расширенные модели RNN

Ячейки LSTM предлагают базовый подход к моделированию долгосрочных зависимостей в последовательностях. Вдобавок важно отметить, что в литературе описано много вариаций LSTM (“An Empirical Exploration of Recurrent Network Architectures” (Эмпирические исследования архитектур рекуррентных сетей), Рафаль Йожефович, Войцех Заремба и Илья Сацкевер, *Proceedings of ICML*, стр. 2342–2350 (2015 г.)). Кроме того, полезно упомянуть о недавно появившемся подходе, который был предложен в 2014 году и называется управляемым рекуррентным блоком (GRU). Архитектура блоков GRU проще, чем ячеек LSTM, из-за чего они более эффективны в вычислительном плане наряду с тем, что их эффективность в задачах вроде моделирования полифонической музыки сравнима с ячейками LSTM. Узнать больше о современных архитектурах RNN можно в статье Цзюнь Янг Чуна и др. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling” (Эмпирическая оценка управляемых рекуррентных нейронных сетей при моделировании последовательностей), которая свободно доступна по ссылке <https://arxiv.org/pdf/1412.3555v1.pdf>.

## Реализация многослойных рекуррентных нейронных сетей для моделирования последовательностей в TensorFlow

После ознакомления с теорией, на которой основаны сети RNN, вы готовы переходить к практической части, связанной с реализацией сетей RNN в TensorFlow. В оставшемся материале главы мы будем применять сети RNN для решения двух распространенных задач:

- 1) смысловой анализ;
- 2) моделирование языка.

Оба проекта, которые мы вместе разберем далее в главе, весьма интересны, но также довольно сложны. Таким образом, вместо предоставления всего кода сразу мы разобьем реализацию на несколько шагов и подробно обсудим код. Если до погружения в обсуждение вы хотите получить более полное представление и увидеть весь код, тогда просмотрите реализацию по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch16>.

## Проект номер один — прогнозирование отношения в рецензиях на фильмы IMDb

Вы можете вспомнить из главы 8, что смысловой анализ касается исследований мнения, выраженного в предложении или текстовом документе. В этом разделе и последующих подразделах мы реализуем многослойную рекуррентную нейронную сеть для смыслового анализа с применением архитектуры “многие к одному”.

В следующем разделе мы займемся реализацией сети RNN с архитектурой “многие ко многим” для приложения моделирования языка. Хотя мы намеренно выбрали простые примеры для представления основных концепций сетей RNN, моделирование языка имеет широкий диапазон интересных приложений, таких как построение чатботов, т.е. наделение компьютеров способностью напрямую вести беседу и взаимодействовать с людьми.

### Подготовка данных с рецензиями на фильмы

Выполняя шаги предварительной обработки в главе 8, мы создали очищенный набор данных по имени `movie_data.csv`, который будем использовать снова. Прежде всего, мы импортируем необходимые модули и прочитаем данные в объект `DataFrame` из `pandas`:

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
```

Вспомните, что этот кадр данных состоит из двух столбцов, `'review'` и `'sentiment'`, где `'review'` содержит тексты рецензий на фильмы (входные признаки), а `'sentiment'` представляет целевые метки, которые мы хотим прогнозировать (0 обозначает отношение, а 1 — положительное отношение). Текстовый компонент рецензии на фильм является последовательностями слов, и модель RNN классифицирует каждую последовательность как положительную (1) или отрицательную (0) рецензию.

Тем не менее, прежде чем мы сможем подать данные модели RNN, нам необходимо предпринять несколько шагов предварительной обработки.

1. Создать объект набора данных TensorFlow и расщепить его на обучающую, испытательную и проверочную части.
2. Идентифицировать уникальные слова в обучающем наборе.
3. Отобразить каждое уникальное слово на уникальное целое число и закодировать текст рецензии с помощью этих целых чисел (индексов уникальных слов).
4. Разделить набор данных на мини-пакеты, которые будут служить входом для модели.

Давайте займемся первым шагом: создание набора данных TensorFlow из имеющегося кадра данных:

```
>>> ## Шаг 1: создание набора данных
>>> target = df.pop('sentiment')
>>> ds_raw = tf.data.Dataset.from_tensor_slices(
...     (df.values, target.values))
>>> ## инспектирование:
>>> for ex in ds_raw.take(3):
...     tf.print(ex[0].numpy()[0][:50], ex[1])
b'In 1974, the teenager Martha Moxley (Maggie Grace)' 1
b'OK... so... I really like Kris Kristofferson and h' 0
b'***SPOILER*** Do not read this, if you think about' 0
```

Теперь мы можем расщепить набор данных на обучающий, испытательный и проверочный наборы. Полный набор данных содержит 50 000 образцов. Мы сохраним первые 25 000 образцов для оценки (удерживаемый испытательный набор), 20 000 образцов будут применяться для обучения и 5 000 для проверки. Вот как выглядит код:

```
>>> tf.random.set_seed(1)
>>> ds_raw = ds_raw.shuffle(
...     50000, reshuffle_each_iteration=False)
>>> ds_raw_test = ds_raw.take(25000)
>>> ds_raw_train_valid = ds_raw.skip(25000)
>>> ds_raw_train = ds_raw_train_valid.take(20000)
>>> ds_raw_valid = ds_raw_train_valid.skip(20000)
```

Чтобы подготовить данные для подачи на вход нейронной сети, нам понадобится закодировать их в виде числовых значений, как упоминалось в

шагах 2 и 3. Сначала мы найдем уникальные слова (лексемы) в обучающем наборе. Хотя поиск уникальных лексем является процессом, для которого допускается использовать наборы данных Python, может быть эффективнее применить класс `Counter` из пакета `collections`, входящего в состав стандартной библиотеки Python.

В приведенном ниже коде мы создадим новый объект `Counter` (`token_counts`), который будет накапливать частоты вхождений каждого уникального слова. Обратите внимание, что в этом конкретном приложении (и в противоположность модели суммирования (мешка) слов) нас интересует только набор уникальных слов, а счетчики слов не требуются и создаются как побочный продукт. Для расщепления текста на слова (или лексемы) пакет `tensorflow_datasets` предлагает класс `Tokenizer`.

Ниже показан код для сбора уникальных лексем:

```
>>> ## Шаг 2: нахождение уникальных лексем (слов)
>>> from collections import Counter

>>> tokenizer = tfds.features.text.Tokenizer()
>>> token_counts = Counter()

>>> for example in ds_raw_train:
...     tokens = tokenizer.tokenize(example[0].numpy()[0])
...     token_counts.update(tokens)

>>> print('Размер словаря:', len(token_counts))
Размер словаря: 87007
```

Если вы хотите получить дополнительные сведения о классе `Counter`, тогда обратитесь в документацию по ссылке <https://docs.python.org/3/library/collections.html#collections.Counter>.

Далее мы собираемся отобразить каждое уникальное слово на уникальное целое число, что можно сделать вручную с использованием словаря Python, где ключи будут уникальными лексемами (словами), а значения — уникальными целыми числами, ассоциированными с ключами. Однако в пакете `tensorflow_datasets` предлагается класс `TokenTextEncoder`, который мы можем применять для создания такого отображения и кодирования всего набора данных. Сначала мы создадим объект `encoder` класса `TokenTextEncoder`, передавая уникальные лексемы (`token_counts` содержит лексемы и их счетчики, хотя здесь счетчики не нужны, так что они игнорируются).

Последующий вызов метода `encoder.encode()` преобразует входной текст в список целочисленных значений:

```
>>> ## Шаг 3: кодирование уникальных лексем в виде целых чисел
>>> encoder = tfds.features.text.TokenTextEncoder(token_counts)
>>> example_str = 'This is an example!'
>>> print(encoder.encode(example_str))
[232, 9, 270, 1123]
```

Обратите внимание, что в проверочных или испытательных данных могут встречаться лексемы, которые отсутствуют в обучающих данных и поэтому не включены в отображение. Если мы имеем  $q$  лексем (т.е. размер словаря `token_counts`, который передавался `TokenTextEncoder` и в нашем случае равен 87 007), тогда всем лексемам, не встречавшимся ранее и соответственно не включенным в `token_counts`, будет назначено целое число  $q + 1$  (в нашем случае 87 008). Иначе говоря, индекс  $q + 1$  резервируется для неизвестных слов. Еще одним зарезервированным значением является целое число 0, которое служит заполнителем для регулировки длины последовательности. Позже при построении модели на основе сети RNN в TensorFlow мы рассмотрим указанные два заполнителя, 0 и  $q + 1$ , более подробно.

Мы можем использовать метод `map()` объектов наборов данных для надлежащей трансформации каждого текста в наборе данных подобно тому, как применялась бы любая другая трансформация к набору данных. Тем не менее, есть небольшая проблема: текстовые данные здесь вложены в объекты тензоров, доступ к которым возможен посредством вызова метода `numpy()` на тензоре в режиме энергичного выполнения. Но во время трансформаций с помощью метода `map()` режим энергичного выполнения отключен. Чтобы решить проблему, мы можем определить две функции. Первая функция будет обращаться с входными тензорами, как если бы режим энергичного выполнения был включен:

```
>>> ## Шаг 3-A: определение функций для трансформации
>>> def encode(text_tensor, label):
...     text = text_tensor.numpy()[0]
...     encoded_text = encoder.encode(text)
...     return encoded_text, label
```

Во второй функции мы поместим первую функцию внутрь вызова `tf.py_function` для ее преобразования в операцию TensorFlow, которую

затем можно использовать через метод `map()`. Процесс кодирования текста в виде списка целых чисел можно обеспечить с применением следующего кода:

```
>>> ## Шаг 3-Б: помещение функции encode внутрь операции TensorFlow.
>>> def encode_map_fn(text, label):
...     return tf.py_function(encode, inp=[text, label],
...                             Tout=(tf.int64, tf.int64))

>>> ds_train = ds_raw_train.map(encode_map_fn)
>>> ds_valid = ds_raw_valid.map(encode_map_fn)
>>> ds_test = ds_raw_test.map(encode_map_fn)

>>> # просмотр формы ряда образцов:
>>> tf.random.set_seed(1)
>>> for example in ds_train.shuffle(1000).take(5):
...     print('Длина последовательности:', example[0].shape)
Длина последовательности: (24,)
Длина последовательности: (179,)
Длина последовательности: (262,)
Длина последовательности: (535,)
Длина последовательности: (130,)
```

Итак, мы преобразовали последовательности слов в последовательности целых чисел. Однако нам осталось решить еще одну проблему — в текущий момент последовательности имеют разные длины (как видно в результате выполнения предыдущего кода для пяти произвольно выбранных образцов). Хотя в общем случае сети RNN способны обрабатывать последовательности с разной длиной, нам придется обеспечить одинаковую длину у всех последовательностей в мини-пакете с целью их эффективного хранения в тензоре.

Для разделения на мини-пакеты набора данных, содержащего элементы с отличающимися формами, библиотека TensorFlow предлагает метод `padded_batch()` (взамен `batch()`), который будет автоматически дополнять следующие друг за другом элементы, объединяемые в пакет, значениями заполнителя (нулями), так что все последовательности внутри пакета будут иметь одну и ту же форму. Чтобы проиллюстрировать это на практическом примере, давайте извлечем небольшой поднабор размера 8 из обучающего набора данных `ds_train` и применим к нему метод `padded_batch()` с `batch_size=4`. Вдобавок мы выведем размеры индивидуальных элементов до их объединения в мини-пакеты, а также размерности результирующих мини-пакетов:



```

>>> ## Извлечение небольшого поднабора
>>> ds_subset = ds_train.take(8)
>>> for example in ds_subset:
...     print('Размер индивидуального элемента:', example[0].shape)
Размер индивидуального элемента: (119,)
Размер индивидуального элемента: (688,)
Размер индивидуального элемента: (308,)
Размер индивидуального элемента: (204,)
Размер индивидуального элемента: (326,)
Размер индивидуального элемента: (240,)
Размер индивидуального элемента: (127,)
Размер индивидуального элемента: (453,)

>>> ## Разделение поднабора на пакеты
>>> ds_batched = ds_subset.padded_batch(
...     4, padded_shapes=([-1], []))
>>> for batch in ds_batched:
...     print('Размерность пакета:', batch[0].shape)
Размерность пакета: (4, 688)
Размерность пакета: (4, 453)

```

Как видно в выведенных формах тензоров, количество столбцов (т.е. `.shape[1]`) в первом пакете составляет 688, что стало результатом объединения первых четырех образцов в пакет и использования максимального размера из них. Оставшиеся три образца в первом пакете дополняются до этого размера. Аналогично второй пакет имеет максимальный размер из своих четырех индивидуальных образцов, равный 453, а остальные образцы соответствующим образом дополняются. Давайте разделим все три набора данных на мини-пакеты с размером пакета 32:

```

>>> train_data = ds_train.padded_batch(
...     32, padded_shapes=([-1], []))
>>> valid_data = ds_valid.padded_batch(
...     32, padded_shapes=([-1], []))
>>> test_data = ds_test.padded_batch(
...     32, padded_shapes=([-1], []))

```

Теперь данные имеют формат, подходящий для модели на основе сети RNN, которую мы собираемся реализовать в последующих подразделах. Тем не менее, ниже мы обсудим *вложение (embedding)* признаков, которое представляет

собой необязательный, но настоятельно рекомендуемый шаг предварительной обработки, направленный на сокращение размерности векторов слов.

## Слой вложений для кодирования предложений

При подготовке данных на предыдущем шаге мы порождали последовательности одинаковой длины. Элементами последовательностей были целые числа, соответствующие *индексам* уникальных слов. Такие индексы слов могут быть преобразованы во входные признаки несколькими способами. Наивный способ предусматривает применение унитарного кодирования для преобразования индексов в векторы нулей и единиц. Затем каждое слово будет сопоставляться с вектором, размер которого равен количеству уникальных слов в полном наборе данных. Учитывая, что число уникальных слов (размер глоссария) может составлять порядка  $10^4$ – $10^5$ , которое будет также количеством входных признаков, обучаемая на таких признаках модель может страдать от “*проклятия размерности*”. К тому же признаки являются крайне разреженными, т.к. все они нулевые кроме одного.

Более элегантный подход заключается в том, чтобы сопоставить каждое слово с вектором фиксированного размера, содержащим вещественные значения (не обязательно целые числа). В противоположность унитарно закодированным векторам мы можем использовать векторы конечных размеров для представления бесконечного количества вещественных чисел. (Теоретически из заданного интервала, скажем,  $[-1, 1]$ , можно извлечь бесконечное количество вещественных чисел.)

Такая идея положена в основу вложения — методики изучения признаков, которую мы можем задействовать здесь для автоматического выявления заметных признаков, чтобы представлять слова в нашем наборе данных. При заданном количестве уникальных слов  $n_{\text{слов}}$  мы можем выбрать размер векторов вложений (он же размерность вложения), который намного меньше количества уникальных слов (*размерность вложения*  $\ll n_{\text{слов}}$ ), для представления полного глоссария как входных признаков.

Вот преимущества вложения перед унитарным кодированием:

- понижение размерности пространства признаков с целью сокращения эффекта “проклятия размерности”;
- выделение заметных признаков из-за того, что слой вложения в нейронной сети может оптимизироваться (или обучаться).

Схематическое представление на рис. 16.10 показывает, как работает вложение за счет отображения индексов лексем на обучаемую матрицу вложений.



Рис. 16.10. Работа вложения

При наличии набора лексем размера  $n + 2$  ( $n$  — размер набора лексем плюс индекс 0 резервируется для заполнителя при дополнении и  $n + 1$  резервируется для слов, отсутствующих в наборе лексем) будет создана матрица вложений размера  $(n + 2) \times \text{размер вложения}$ , каждая строка которой представляет числовые признаки, ассоциированные с лексемой. Следовательно, когда вложению в качестве входа передается целочисленный индекс  $i$ , оно будет находить в матрице соответствующую строку с индексом  $i$  и возвращать числовые признаки. Матрица вложений служит входным слоем наших нейросетевых моделей. На практике слой вложений создается просто с применением класса `tf.keras.layers.Embedding`. Давайте рассмотрим пример, где создадим модель и добавим слой вложений:

```
>>> from tensorflow.keras.layers import Embedding
>>> model = tf.keras.Sequential()
```

```
>>> model.add(Embedding(input_dim=100,
...                       output_dim=6,
...                       input_length=20,
...                       name='embed-layer'))
```

```
>>> model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embed-layer (Embedding)	(None, 20, 6)	600
Total params: 6,00		
Trainable params: 6,00		
Non-trainable params: 0		

Вход этой модели (слой вложений) обязан иметь ранг 2 с размерностью *размер пакета × длина входа*, где *длина входа* представляет собой длину последовательностей (установленную здесь в 20 посредством аргумента `input_length`). Скажем, входной последовательностью в мини-пакете может быть `<14,43,52,61,8,19,67,83,10,7,42,87,56,18,94,17,67,90,6,39>`, каждый элемент которой является индексом уникального слова. Выход будет иметь размерность *размер пакета × длина входа × размерность вложения*, где *размерность вложения* — размер признаков вложения (установленный здесь в 6 через аргумент `output_dim`). Еще один аргумент, предоставляемый слою вложений, `input_dim`, соответствует уникальным целочисленным значениям, которые модель будет получать в качестве входа (например,  $n + 2$ , установленное здесь в 100). Таким образом, матрица вложений в нашем случае имеет размер  $100 \times 6$ .



На  
заметку!

### Работа с переменными длинами последовательностей

Обратите внимание, что аргумент `input_length` не является обязательным, и мы можем использовать `None` в сценариях, когда длины входных последовательностей варьируются. Дополнительные сведения ищите в официальной документации по ссылке [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/layers/Embedding](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding).

## Построение модели на основе рекуррентной нейронной сети

Итак, мы готовы к построению модели на основе RNN. С помощью класса `Sequential` библиотеки Keras мы можем объединить слой вложений, рекуррентные слои сети RNN и полносвязные нерекуррентные слои. Для рекуррентных слоев мы можем применять любую из следующих реализаций:

- `SimpleRNN` — обыкновенный слой RNN, т.е. полносвязный рекуррентный слой;
- `LSTM` — слой RNN с долгой краткосрочной памятью, которая полезна для выявления долгосрочных зависимостей;
- `GRU` — рекуррентный слой с управляемым рекуррентным блоком, предложенный в качестве альтернативы ячейкам LSTM в статье “Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation” (Изучение представлений фраз с использованием рекуррентной нейронной сети “кодировщик–декодировщик” для статистического машинного перевода), которая свободно доступна по ссылке <https://arxiv.org/abs/1406.1078v3>.

Чтобы взглянуть, как можно построить модель на основе многослойной рекуррентной нейронной сети с применением одного из перечисленных выше рекуррентных слоев, в приведенном далее примере мы создадим модель RNN, начав со слоя вложений с `input_dim=1000` и `output_dim=32`. Затем мы добавим два рекуррентных слоя типа `SimpleRNN`. В заключение мы добавим полносвязный нерекуррентный слой в качестве выходного слоя, который будет возвращать одиночное выходное значение, представляющее собой прогноз:

```
>>> from tensorflow.keras import Sequential
>>> from tensorflow.keras.layers import Embedding
>>> from tensorflow.keras.layers import SimpleRNN
>>> from tensorflow.keras.layers import Dense

>>> model = Sequential()
>>> model.add(Embedding(input_dim=1000, output_dim=32))
>>> model.add(SimpleRNN(32, return_sequences=True))
>>> model.add(SimpleRNN(32))
>>> model.add(Dense(1))
>>> model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 32)	32000
simple_rnn (SimpleRNN)	(None, None, 32)	2080
simple_rnn_1 (SimpleRNN)	(None, 32)	2080
dense (Dense)	(None, 1)	33
Total params: 36,193		
Trainable params: 36,193		
Non-trainable params: 0		

Как видите, построение модели на основе сети RNN с использованием таких рекуррентных слоев, выглядит достаточно прямолинейно. В следующем подразделе мы вернемся к нашей задаче смыслового анализа и построим модель RNN для ее решения.

### Построение модели на основе рекуррентной нейронной сети для решения задачи смыслового анализа

Поскольку мы имеем очень длинные последовательности, то собираемся применять слой LSTM для учета долгосрочных эффектов. Вдобавок мы поместим слой LSTM внутрь оболочки Bidirectional, которая заставит рекуррентные слои проходить по входным последовательностям в обоих направлениях — от начала до конца и обратно:

```
>>> embedding_dim = 20
>>> vocab_size = len(token_counts) + 2
>>> tf.random.set_seed(1)
>>> ## построение модели
>>> bi_lstm_model = tf.keras.Sequential([
...     tf.keras.layers.Embedding(
...         input_dim=vocab_size,
...         output_dim=embedding_dim,
...         name='embed-layer'),
...     ])
```

```

...     tf.keras.layers.Bidirectional(
...         tf.keras.layers.LSTM(64, name='lstm-layer'),
...         name='bidir-lstm'),
...
...     tf.keras.layers.Dense(64, activation='relu'),
...
...     tf.keras.layers.Dense(1, activation='sigmoid')
>>> ])

>>> bi_lstm_model.summary()

>>> ## КОМПИЛЯЦИЯ И ОБУЧЕНИЕ:
>>> bi_lstm_model.compile(
...     optimizer=tf.keras.optimizers.Adam(1e-3),
...     loss=tf.keras.losses.BinaryCrossentropy(from_logits=False),
...     metrics=['accuracy'])

>>> history = bi_lstm_model.fit(
...     train_data,
...     validation_data=valid_data,
...     epochs=10)

>>> ## ОЦЕНКА НА ИСПЫТАТЕЛЬНЫХ ДАННЫХ
>>> test_results = bi_lstm_model.evaluate(test_data)
>>> print('Правильность при испытании: {:.2f}%'.
...       format(test_results[1]*100))

Epoch 1/10
625/625 [=====] - 96s 154ms/step - loss:
0.4410 - accuracy: 0.7782 - val_loss: 0.0000e+00 - val_accuracy:
0.0000e+00
Epoch 2/10
625/625 [=====] - 95s 152ms/step - loss:
0.1799 - accuracy: 0.9326 - val_loss: 0.4833 - val_accuracy:
0.8414
. . .

Правильность при испытании: 85.15%

```

После обучения модели в течение 10 эпох оценка на испытательных данных показывает 85%-ную правильность. (Отметим, что этот результат не является наилучшим в сравнении с современными методами, используемыми на наборе данных IMDB. Цель заключалась в том, чтобы просто продемонстрировать работу сети RNN.)



## Дополнительные сведения о двунаправленной сети RNN

На заметку!

Оболочка `Bidirectional` делает два прохода по каждой входной последовательности: прямой проход и противоположный или обратный проход (важно не путать их с прямым и обратным проходами в контексте обратного распространения). Результаты таких прямых и обратных проходов по умолчанию будут объединяться. Но если вы хотите изменить это поведение, тогда можете установить аргумент `merge_mode` в `'sum'` (для сложения), `'mul'` (для умножения результатов двух проходов), `'ave'` (для взятия среднего из двух), `'concat'` (для объединения; стандартное значение) или `None`, что приводит к возвращению двух тензоров в списке. Дополнительную информацию об оболочке `Bidirectional` ищите в официальной документации по ссылке [https://www.tensorflow.org/versions/r2.0/api\\_docs/python/tf/keras/layers/Bidirectional](https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Bidirectional).

Мы также можем опробовать рекуррентные слои других типов, такие как `SimpleRNN`. Однако оказывается, что модель, построенная с обыкновенными рекуррентными слоями, не будет способна достичь хорошей эффективности прогнозирования (даже на обучающих данных). Скажем, если вы попытаетесь заменить двунаправленный слой `LSTM` в предыдущем коде однонаправленным слоем `SimpleRNN` и обучите модель на полноразмерных последовательностях, то можете заметить, что потеря не уменьшается даже во время обучения. Причина в том, что последовательности в этом наборе данных слишком длинные, а потому модель со слоем `SimpleRNN` не в состоянии узнать долгосрочные зависимости и может страдать от проблем исчезновения или взрывного роста градиентов.

Для получения приемлемой эффективности прогнозирования на таком наборе данных с применением слоя `SimpleRNN` мы можем усекать последовательности. Кроме того, используя наше “знание предметной области”, мы можем предположить, что последние абзацы рецензии на фильм содержат большинство информации об отношении к нему. Следовательно, мы можем сосредоточиться только на последней порции каждой рецензии. Для этого мы определим вспомогательную функцию `preprocess_datasets()`, где объединим шаги 2–4. Функция `preprocess_datasets()` будет иметь дополнительный аргумент `max_seq_length`, определяющий количество лексем, которые должны быть задействованы из каждой рецензии. Скажем,



если мы установим `max_seq_length` в 100, а рецензия содержит более 100 лексем, тогда будут применяться только последние 100 лексем. Если `max_seq_length` устанавливается в `None`, то будут использоваться полные последовательности без сокращений. Опробование отличающихся значений для `max_seq_length` позволит лучше понять возможности разных моделей RNN по обработке длинных последовательностей.

Ниже приведен код функции `preprocess_datasets()`:

```
>>> from collections import Counter

>>> def preprocess_datasets(
...     ds_raw_train,
...     ds_raw_valid,
...     ds_raw_test,
...     max_seq_length=None,
...     batch_size=32):
...
...     ## (шаг 1 уже сделан)
...     ## Шаг 2: нахождение уникальных лексем
...     tokenizer = tfds.features.text.Tokenizer()
...     token_counts = Counter()
...
...     for example in ds_raw_train:
...         tokens = tokenizer.tokenize(example[0].numpy()[0])
...         if max_seq_length is not None:
...             tokens = tokens[-max_seq_length:]
...         token_counts.update(tokens)
...
...     print('Размер словаря:', len(token_counts))
...
...     ## Шаг 3: кодирование текста
...     encoder = tfds.features.text.TokenTextEncoder(
...         token_counts)
...     def encode(text_tensor, label):
...         text = text_tensor.numpy()[0]
...         encoded_text = encoder.encode(text)
...         if max_seq_length is not None:
...             encoded_text = encoded_text[-max_seq_length:]
...         return encoded_text, label
...
...     def encode_map_fn(text, label):
...         return tf.py_function(encode, inp=[text, label],
...                                Tout=(tf.int64, tf.int64))
```

```

...
... ds_train = ds_raw_train.map(encode_map_fn)
... ds_valid = ds_raw_valid.map(encode_map_fn)
... ds_test = ds_raw_test.map(encode_map_fn)
...
... ## Шаг 4: создание пакетов для наборов данных
... train_data = ds_train.padded_batch(
...     batch_size, padded_shapes=([-1], []))
...
... valid_data = ds_valid.padded_batch(
...     batch_size, padded_shapes=([-1], []))
...
... test_data = ds_test.padded_batch(
...     batch_size, padded_shapes=([-1], []))
...
... return (train_data, valid_data,
...         test_data, len(token_counts))

```

Далее мы определим еще одну вспомогательную функцию, `build_rnn_model()`, для более удобного построения моделей с разными архитектурами:

```

>>> from tensorflow.keras.layers import Embedding
>>> from tensorflow.keras.layers import Bidirectional
>>> from tensorflow.keras.layers import SimpleRNN
>>> from tensorflow.keras.layers import LSTM
>>> from tensorflow.keras.layers import GRU

>>> def build_rnn_model(embedding_dim, vocab_size,
...                      recurrent_type='SimpleRNN',
...                      n_recurrent_units=64,
...                      n_recurrent_layers=1,
...                      bidirectional=True):
...
...     tf.random.set_seed(1)
...
...     # построение модели
...     model = tf.keras.Sequential()
...
...     model.add(
...         Embedding(
...             input_dim=vocab_size,
...             output_dim=embedding_dim,
...             name='embed-layer')
...     )

```

```

...
...     for i in range(n_recurrent_layers):
...         return_sequences = (i < n_recurrent_layers-1)
...
...         if recurrent_type == 'SimpleRNN':
...             recurrent_layer = SimpleRNN(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='simprnn-layer-{}'.format(i))
...         elif recurrent_type == 'LSTM':
...             recurrent_layer = LSTM(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='lstm-layer-{}'.format(i))
...         elif recurrent_type == 'GRU':
...             recurrent_layer = GRU(
...                 units=n_recurrent_units,
...                 return_sequences=return_sequences,
...                 name='gru-layer-{}'.format(i))
...
...         if bidirectional:
...             recurrent_layer = Bidirectional(
...                 recurrent_layer, name='bidir-' +
...                 recurrent_layer.name)
...
...         model.add(recurrent_layer)
...
...     model.add(tf.keras.layers.Dense(64, activation='relu'))
...     model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
...
...     return model

```

Теперь, располагая этими двумя довольно универсальными, но удобными вспомогательными функциями, мы можем легко сравнивать разные модели RNN с отличающимися длинами входных последовательностей. Например, в следующем коде мы опробуем модель с единственным рекуррентным слоем типа SimpleRNN, усекая последовательности до максимальной длины в 100 лексем:

```

>>> batch_size = 32
>>> embedding_dim = 20
>>> max_seq_length = 100

```

```
>>> train_data, valid_data, test_data, n = preprocess_datasets(
...     ds_raw_train, ds_raw_valid, ds_raw_test,
...     max_seq_length=max_seq_length,
...     batch_size=batch_size
... )
```

```
>>> vocab_size = n + 2
```

```
>>> rnn_model = build_rnn_model(
...     embedding_dim, vocab_size,
...     recurrent_type='SimpleRNN',
...     n_recurrent_units=64,
...     n_recurrent_layers=1,
...     bidirectional=True)
```

```
>>> rnn_model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embed-layer (Embedding)	(None, None, 20)	1161300
bidir-simprnn-layer-0 (Bidir	(None, 128)	10880
Dense (Dense)	(None, 64)	8256
dense_1 (Dense)	(None, 1)	65

```
Total params: 1,180,501
```

```
Trainable params: 1,180,501
```

```
Non-trainable params: 0
```

```
>>> rnn_model.compile(
...     optimizer=tf.keras.optimizers.Adam(1e-3),
...     loss=tf.keras.losses.BinaryCrossentropy(
...         from_logits=False), metrics=['accuracy'])
```

```
>>> history = rnn_model.fit(
...     train_data,
...     validation_data=valid_data,
...     epochs=10)
```

```
Epoch 1/10
```

```
625/625 [=====] - 73s 118ms/step - loss:
0.6996 - accuracy: 0.5074 - val_loss: 0.6880 - val_accuracy: 0.5476
```

Epoch 2/10

```
>>> results = rnn_model.evaluate(test_data)
>>> print('Правильность при испытании: {:.2f}%'.
format(results[1]*100))
Правильность при испытании: 80.70%
```

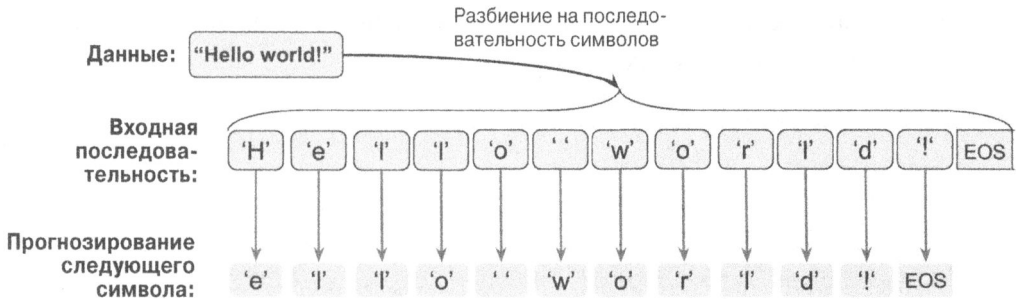
Как выяснилось, усечение последовательностей до 100 лексем и применение двунаправленного слоя SimpleRNN обеспечивает 80%-ную правильность классификации. Хотя правильность чуть ниже, чем у предыдущей модели с двунаправленным слоем LSTM (85.15%-ная правильность на испытательном наборе данных), эффективность на таких усеченных последовательностях гораздо лучше эффективности, которой мы могли бы достичь с помощью слоя SimpleRNN на полных рецензиях. В качестве дополнительного упражнения можете проверить сказанное, используя две вспомогательные функции, которые мы уже определили. Попробуйте установить `max_seq_length` в `None` и аргумент `bidirectional` внутри вспомогательной функции `build_rnn_model()` в `False`. (Полный код доступен в архиве с примерами для книги.)

## Проект номер два — моделирование языка на уровне символов в TensorFlow

Моделирование языка является восхитительным приложением, которое наделяет машины способностью исполнять задачи, связанные с человеческим языком, такие как порождение предложений на английском языке. Одна из интересных научных работ в этой области описана в статье Ильи Сацкевера, Джеймса Мартинса и Джеффри Хинтона “Generating Text with Recurrent Neural Networks” (Генерирование текста с помощью рекуррентных нейронных сетей), *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011 г., свободно доступная по ссылке <https://pdfs.semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf>.

В модели, которую мы построим, входом будет текстовый документ, а наша цель заключается в том, чтобы разработать модель для порождения нового текста, похожего по стилю на текст во входном документе. Примерами такого входа могут служить книги или компьютерные программы на определенном языке программирования.

При моделировании языка на уровне символов вход разбивается на последовательность символов, которая подается в нашу сеть по одному символу за раз. Сеть будет обрабатывать каждый новый символ в сочетании с запомненными ранее встречавшимися символами для прогнозирования следующего символа. Пример моделирования языка на уровне символов демонстрируется на рис. 16.11 (EOS означает “end of sequence” — “конец последовательности”).



*Рис. 16.11. Пример моделирования языка на уровне символов*

Мы можем разделить реализацию на три шага — подготовка данных, построение модели на основе сети RNN и выработка прогноза следующего символа вместе с выборкой для порождения нового текста.

### Предварительная обработка набора данных

В этом разделе мы подготовим данные для моделирования языка на уровне символов.

Чтобы получить входные данные, понадобится зайти на веб-сайт проекта “Гутенберг” (<https://www.gutenberg.org/>), где предлагаются тысячи бесплатных электронных книг. Для нашего примера мы загрузим книгу “The Mysterious Island” (“Таинственный остров”), которая была написана Жюлем Верном и вышла в 1874 году, в формате простого текста по ссылке <http://www.gutenberg.org/files/1268/1268-0.txt>.

Важно отметить, что указанная ссылка напрямую ведет к странице загрузки. В случае работы в среде macOS или Linux загрузить файл можно посредством следующей команды:

```
curl -O http://www.gutenberg.org/files/1268/1268-0.txt
```

На тот случай, если в будущем данный ресурс станет недоступным, копия текста также включена в каталог с кодом для настоящей главы, который находится по ссылке <https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch16>.

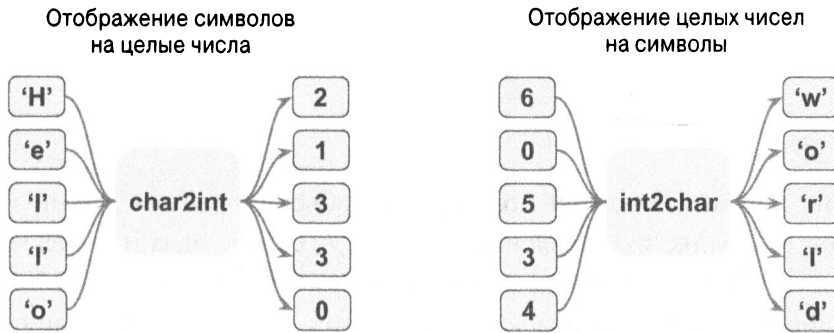
После загрузки набора данных мы можем прочитать его как простой текст в сеансе Python. С помощью показанного ниже кода мы читаем текст непосредственно из загруженного файла и удаляем части в начале и конце (они содержат описание проекта “Гутенберг”). Затем мы создаем переменную Python по имени `char_set`, которая будет представлять набор *уникальных* символов, обнаруженных в тексте:

```
>>> import numpy as np

>>> ## Чтение и обработка текста
>>> with open('1268-0.txt', 'r') as fp:
...     text=fp.read()

>>> start_idx = text.find('THE MYSTERIOUS ISLAND')
>>> end_idx = text.find('End of the Project Gutenberg')
>>> text = text[start_idx:end_idx]
>>> char_set = set(text)
>>> print('Общая длина:', len(text))
Общая длина: 1112350
>>> print('Уникальные символы:', len(char_set))
Уникальные символы: 80
```

После загрузки и предварительной обработки текста мы имеем последовательность, состоящую в общей сложности из 1 112 350 символов, 80 из которых уникальны. Тем не менее, большинство библиотек для нейронных сетей и реализаций сетей RNN не могут работать с входными данными в строковом формате, из-за чего мы должны преобразовать текст в числовой формат. Для такого преобразования мы создадим словарь Python, `char2int`, который отображает каждый символ на целое число. Нам также потребуется обратное отображение, чтобы преобразовывать результаты нашей модели снова в текст. Хотя обратное отображение можно сделать с применением словаря, ассоциирующего целочисленные ключи с символьными значениями, более эффективно использовать массив NumPy, сопоставляя его индексы с уникальными символами. На рис. 16.12 показан пример преобразования символов в целые числа и обратно для слов “Hello” и “world”.



*Рис. 16.12. Преобразование символов в целые числа и обратно для слов "Hello" и "world"*

Вот как построить словарь для отображения символов на целые числа и обратного отображения через индексацию массива NumPy, изображенного на рис. 16.12:

```
>>> chars_sorted = sorted(char_set)
>>> char2int = {ch:i for i,ch in enumerate(chars_sorted)}
>>> char_array = np.array(chars_sorted)

>>> text_encoded = np.array(
...     [char2int[ch] for ch in text],
...     dtype=np.int32)

>>> print('Форма закодированного текста:', text_encoded.shape)
Форма закодированного текста: (1112350,)
>>> print(text[:15], '== Кодирование ==>', text_encoded[:15])
>>> print(text_encoded[15:21], '== Обратное преобразование ==>',
...       ''.join(char_array[text_encoded[15:21]]))
THE MYSTERIOUS == Кодирование ==> [44 32 29  1 37 48 43 44 29 42 33
39 45 43  1]
[33 43 36 25 38 28] == Обратное преобразование ==> ISLAND
```

Массив NumPy по имени `text_encoded` содержит закодированные значения для всех символов в тексте. А теперь создадим из этого массива набор данных TensorFlow:

```
>>> import tensorflow as tf

>>> ds_text_encoded = tf.data.Dataset.from_tensor_slices(
...     text_encoded)
>>> for ex in ds_text_encoded.take(5):
...     print('{} -> {}'.format(ex.numpy(), char_array[ex.numpy()])))
```



44 -> Т  
 32 -> Н  
 29 -> Е  
 1 ->  
 37 -> М

До сих пор мы создавали итерируемый объект Dataset для получения символов в порядке их появления в тексте. А сейчас давайте отвлечемся и взглянем на общую картину того, что мы пытаемся предпринять. Мы можем сформулировать задачу порождения текста как задачу классификации.

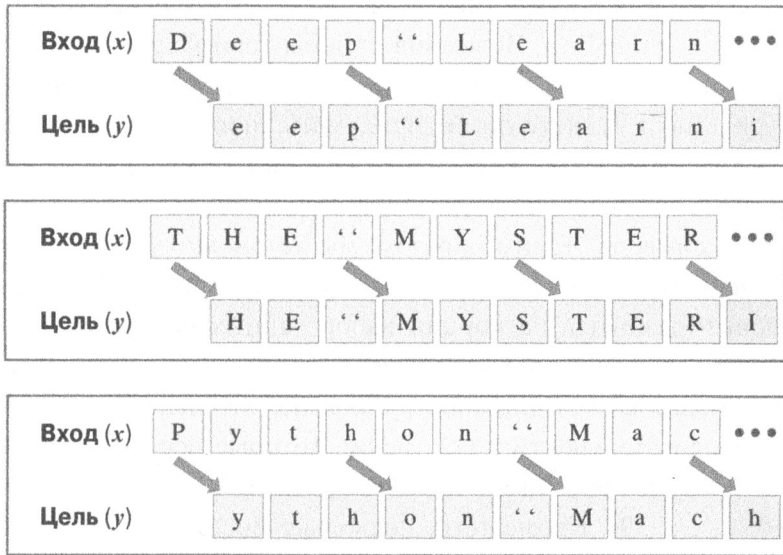
Пусть у нас есть набор последовательностей символов текста, которые являются неполными (рис. 16.13).



**Рис. 16.13.** Неполные последовательности символов текста

Последовательности в левом прямоугольнике на рис. 16.13 мы можем рассматривать как вход. При порождении нового текста наша цель заключается в том, чтобы спроектировать модель, которая сумеет прогнозировать следующий символ заданной входной последовательности, представляющей неполный текст. Например, увидев “Deep Learn”, модель должна спрогнозировать “i” в качестве следующего символа. Учитывая наличие 80 уникальных символов, задача подпадает под категорию многоклассовой классификации.

Начиная с последовательности длиной 1 (т.е. одиночной буквы), мы можем многократно порождать новый текст, основываясь на таком подходе многоклассовой классификации, как демонстрируется на рис. 16.14.



**Рис. 16.14.** Многократное порождение нового текста на основе подхода многоклассовой классификации

Чтобы реализовать задачу порождения текста в TensorFlow, первым делом давайте сократим длину последовательности до 40, т.е. входной тензор  $x$  будет состоять из 40 лексем. На практике длина последовательности влияет на качество порождаемого текста. Более длинные последовательности могут приводить к более осмысленным предложениям. Однако в случае более коротких последовательностей модель способна сосредоточиться на корректном выявлении индивидуальных слов, большей частью игнорируя контекст. Несмотря на то что, как упоминалось, более длинные последовательности обычно дают более осмысленные предложения, для длинных последовательностей модель RNN будет иметь проблемы с выявлением долгосрочных зависимостей. Таким образом, на практике нахождение золотой середины и хорошего значения для длины последовательности является задачей оптимизации гиперпараметров, которую мы должны решить опытным путем. Здесь мы выбрали длину 40, т.к. она обеспечивает хороший компромисс.

Как вы видели на рис. 16.14, входы  $x$  и цели  $y$  смещены на один символ. Следовательно, мы разделим текст на порции с размером 41: первые 40 символом будут формировать входную последовательность  $x$ , а последние 40 элементов образуют целевую последовательность  $y$ .

Мы уже сохранили полный закодированный текст в объекте `Dataset` по имени `ds_text_encoded`. Вспомнив приемы трансформирования наборов данных, раскрытые ранее в главе (в разделе “Подготовка данных с рецензиями на фильмы”), можете ли вы придумать способ получения входа  $x$  и цели  $y$ , как было показано на рис. 16.14? Ответ очень прост: мы сначала применим метод `batch()` для создания порций текста, каждая из которых состоит из 41 символа. Таким образом, мы установим `batch_size` в 41. Далее мы избавимся от последнего пакета, если он короче 41 символа. В результате новый разбитый на порции набор данных по имени `ds_chunks` будет содержать последовательности размера 41. Затем 41-символьные порции будут использоваться для конструирования последовательности  $x$  (т.е. входа) и последовательности  $y$  (т.е. цели), которые обе получают по 40 элементов. Скажем, последовательность  $x$  будет состоять из элементов с индексами `[0, 1, ..., 39]`. Кроме того, поскольку последовательность  $y$  будет смещена на одну позицию относительно последовательности  $x$ , ее индексами будут `[1, 2, ..., 40]`. Потом мы применим функцию трансформации, используя метод `map()`, чтобы надлежащим образом разделить последовательности  $x$  и  $y$ :

```
>>> seq_length = 40
>>> chunk_size = seq_length + 1
>>> ds_chunks = ds_text_encoded.batch(chunk_size,
...                                   drop_remainder=True)
>>> ## определение функции для разделения x и y
>>> def split_input_target(chunk):
...     input_seq = chunk[:-1]
...     target_seq = chunk[1:]
...     return input_seq, target_seq
>>> ds_sequences = ds_chunks.map(split_input_target)
```

Давайте рассмотрим несколько примеров последовательностей из трансформированного набора данных:

```
>>> for example in ds_sequences.take(2):
...     print('Вход (x): ',
...           repr(''.join(char_array[example[0].numpy()])))
...     print('Цель (y): ',
...           repr(''.join(char_array[example[1].numpy()])))
...     print()
```

```

Вход (x): 'THE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced b'
Цель (y): 'HE MYSTERIOUS ISLAND ***\n\n\n\n\nProduced by'

Вход (x): ' Anthony Matonak, and Trevor Carlson\n\n\n\n\n'
Цель (y): 'Anthony Matonak, and Trevor Carlson\n\n\n\n\n'

```

Наконец, последний шаг в подготовке набора данных предусматривает его разделение на мини-пакеты. На первом шаге предварительной обработки для разделения набора данных на пакеты мы создали порции предложений. Каждая порция представляет одно предложение, которое соответствует одному обучающему образцу. Теперь мы перетасуем обучающие образцы и снова разделим входы на мини-пакеты; тем не менее, на этот раз каждый пакет будет содержать множество обучающих образцов:

```

>>> BATCH_SIZE = 64
>>> BUFFER_SIZE = 10000
>>> ds = ds_sequences.shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

```

### **Построение модели на основе рекуррентной нейронной сети для моделирования языка на уровне символов**

Итак, когда набор данных готов, построение модели будет относительно прямолинейным. В целях многократного использования кода мы напишем функцию `build_model`, которая определит модель RNN с применением класса `Sequential` библиотеки Keras. Затем мы можем указать параметры обучения и вызвать функцию `build_model` для получения модели RNN:

```

>>> def build_model(vocab_size, embedding_dim, rnn_units):
...     model = tf.keras.Sequential([
...         tf.keras.layers.Embedding(vocab_size, embedding_dim),
...         tf.keras.layers.LSTM(
...             rnn_units,
...             return_sequences=True),
...         tf.keras.layers.Dense(vocab_size)
...     ])
...     return model

>>> ## Установка параметров обучения
>>> charset_size = len(char_array)
>>> embedding_dim = 256
>>> rnn_units = 512

>>> tf.random.set_seed(1)
>>> model = build_model(

```

```
... vocab_size=charset_size,
... embedding_dim=embedding_dim,
... rnn_units=rnn_units)
>>> model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, None, 256)	20480
lstm (LSTM)	(None, None, 512)	1574912
dense (Dense)	(None, None, 80)	41040
Total params: 1,636,432		
Trainable params: 1,636,432		
Non-trainable params: 0		

Обратите внимание, что слой LSTM в этой модели имеет форму выхода (None, None, 512), т.е. выход слоя LSTM получает ранг 3. Первое измерение обозначает количество пакетов, второе — длину последовательности и третье — число скрытых элементов. Причина наличия выхода с рангом 3 из слоя LSTM связана с тем, что при его определении мы указали `return_sequences=True`. Полносвязный слой (Dense) получает выход из ячейки LSTM и рассчитывает логиты для каждого элемента выходной последовательности. В итоге финальный выход модели также будет тензором ранга 3.

Вдобавок мы указали `activation=None` для последнего полносвязного слоя. Дело в том, что нам необходимо иметь логиты в качестве выходов модели, чтобы мы могли отбирать прогнозы модели для порождения нового текста. Позже мы вернемся к этой части, связанной с выборкой. Пока что давайте обучим модель:

```
>>> model.compile(
...     optimizer='adam',
...     loss=tf.keras.losses.SparseCategoricalCrossentropy(
...         from_logits=True
...     ))
>>> model.fit(ds, epochs=20)
```

```

Epoch 1/20
424/424 [=====] - 80s 189ms/step - loss:
2.3437
Epoch 2/20
424/424 [=====] - 79s 187ms/step - loss:
1.7654
...
Epoch 20/20
424/424 [=====] - 79s 187ms/step - loss:
1.0478

```

Теперь мы можем оценить модель в плане порождения нового текста, начав с заданной короткой строки. В следующем подразделе мы определим функцию для оценки обученной модели.

### Стадия оценки — порождение новых отрывков текста

Модель RNN, обученная в предыдущем подразделе, возвращает логиты размера 80 для каждого уникального символа. Посредством функции `softmax` эти логиты могут быть легко преобразованы в вероятности того, что определенный символ встретится следующим. Чтобы спрогнозировать следующий символ в последовательности, мы можем просто выбрать элемент с максимальным значением логита, что эквивалентно выбору символа с наивысшей вероятностью. Однако вместо постоянного выбора символа с наивысшей вероятностью мы хотим делать (случайную) *выборку* из выходов, иначе модель будет всегда порождать один и тот же текст. Библиотека TensorFlow предоставляет функцию `tf.random.categorical()`, которую мы можем использовать для извлечения образцов из категориального распределения. Чтобы посмотреть, как она работает, давайте сгенерируем ряд случайных образцов из трех категорий [0, 1, 2] с входными логитами [1, 1, 1]:

```

>>> tf.random.set_seed(1)
>>> logits = [[1.0, 1.0, 1.0]]
>>> print('Вероятности:', tf.math.softmax(logits).numpy()[0])
Вероятности: [0.33333334 0.33333334 0.33333334]
>>> samples = tf.random.categorical(
...     logits=logits, num_samples=10)
>>> tf.print(samples.numpy())
array([[0, 0, 1, 2, 0, 0, 0, 1, 0]])

```

Как видите, с заданными логитами категории имеют одинаковые вероятности (т.е. являются равновероятными). Следовательно, если мы применяем большой размер выборки (*количество образцов*  $\rightarrow \infty$ ), то ожидаем, что число вхождений каждой категории достигнет  $\approx 1/3$  размера выборки. Если мы изменим логиты на [1, 1, 3], тогда будем ожидать встретить больше вхождений категории 2 (когда из этого распределения извлекается очень большое количество образцов):

```
>>> tf.random.set_seed(1)
>>> logits = [[1.0, 1.0, 3.0]]
>>> print('Вероятности: ', tf.math.softmax(logits).numpy()[0])
Вероятности: [0.10650698 0.10650698 0.78698605]
>>> samples = tf.random.categorical(
...     logits=logits, num_samples=10)
>>> tf.print(samples.numpy())
array([[2, 0, 2, 2, 2, 0, 1, 2, 2, 0]])
```

С использованием `tf.random.categorical` мы можем генерировать образцы на основе логитов, рассчитанных моделью. Мы определим функцию `sample()`, которая будет получать короткую стартовую строку `starting_str` и порождать новую строку `generated_str`, первоначально установленную во входную строку. Затем из конца `generated_str` берется строка размером `max_input_length` и кодируется в виде последовательности целых чисел `encoded_input`, которая передается модели RNN для расчета логитов. Обратите внимание, что выход из модели RNN является последовательностью логитов с такой же длиной, как у входной последовательности, поскольку мы указали `return_sequences=True` для последнего рекуррентного слоя модели RNN. Таким образом, каждый элемент в выходе модели RNN представляет логиты (здесь вектор размера 80, равного общему количеству символов) для следующего символа после просмотра входной последовательности моделью.

Мы используем только последний элемент выхода логитов (т.е.  $\sigma^{(T)}$ ), который передается функции `tf.random.categorical()` для генерации нового образца. Новый образец преобразуется в символ, который затем присоединяется к концу порожденной строки `generated_text`, увеличивая ее длину на 1. Далее процесс повторяется: с конца `generated_str` берутся последние `max_input_length` символов и применяются для генерации нового символа до тех пор, пока длина порожденной строки не достигнет

желательной величины. Процесс потребления порожденной последовательности в качестве входа для генерирования новых элементов называется *автоматической регрессией*.



### Возвращение последовательностей в виде выхода

Вас может интересовать, зачем мы применяем `return_sequences=True`, если задействуем только последний символ при выборке нового символа и игнорируем остаток выхода. Хотя такой вопрос совершенно обоснован, вы не должны забывать о том, что при обучении мы используем полную выходную последовательность. Потеря вычисляется на базе каждого прогноза в выходе, а не только последнего.

Ниже приведен код функции `sample()`:

```
>>> def sample(model, starting_str,
...             len_generated_text=500,
...             max_input_length=40,
...             scale_factor=1.0):
...     encoded_input = [char2int[s] for s in starting_str]
...     encoded_input = tf.reshape(encoded_input, (1, -1))
...
...     generated_str = starting_str
...
...     model.reset_states()
...     for i in range(len_generated_text):
...         logits = model(encoded_input)
...         logits = tf.squeeze(logits, 0)
...
...         scaled_logits = logits * scale_factor
...         new_char_indx = tf.random.categorical(
...             scaled_logits, num_samples=1)
...
...         new_char_indx = tf.squeeze(new_char_indx)[-1].numpy()
...
...         generated_str += str(char_array[new_char_indx])
...
...         new_char_indx = tf.expand_dims([new_char_indx], 0)
...         encoded_input = tf.concat(
...             [encoded_input, new_char_indx],
...             axis=1)
...         encoded_input = encoded_input[:, -max_input_length:]
...
...     return generated_str
```



Давайте сгенерируем какой-нибудь новый текст:

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island'))
The island is probable that the view of the vegetable discharge on
unexplaint felt, a thore, did not
refrain it existing to the greatest
possing bain and production, for a hundred streamled
established some branches of the
holizontal direction. It was there is all ready, from one things
from contention of the Pacific
acid, and
according to an occurry so
summ on the rooms. When numbered the prud Spilett received an
exceppering from their head, and by went inhabited.

"What are the most abundance a report
```

Как видите, модель порождает главным образом корректные слова и в ряде случаев предложения частично осмысленны. Вы можете дополнительно подстраивать параметры обучения, такие как длина входных последовательностей при обучении, архитектура модели и параметры выборки (вроде `max_input_length`).

Кроме того, чтобы контролировать предсказуемость сгенерированных образцов (т.е. порождать текст, следуя выявленным шаблонам в обучающем тексте, а не добавлять больше случайности), рассчитанные моделью RNN логиты допускается масштабировать перед передачей функции `tf.random.categorical()` для выборки. Масштабный коэффициент  $\alpha$  можно интерпретировать как инверсию температуры в физике. Более высокая температура дает в результате большую случайность по сравнению с более предсказуемым поведением при низких температурах. Благодаря масштабированию логитов с коэффициентом  $\alpha < 1$  вероятности, вычисляемые функцией `softmax`, становятся более равномерными, как показано в следующем коде:

```
>>> logits = np.array([[1.0, 1.0, 3.0]])
>>> print('Вероятности перед масштабированием: ',
...       tf.math.softmax(logits).numpy()[0])
>>> print('Вероятности после масштабирования с коэффициентом 0.5:',
...       tf.math.softmax(0.5*logits).numpy()[0])
```

```
>>> print('Вероятности после масштабирования с коэффициентом 0.1:',
...       tf.math.softmax(0.1*logits).numpy()[0])
Вероятности перед масштабированием:
[0.10650698 0.10650698 0.78698604]
Вероятности после масштабирования с коэффициентом 0.5:
[0.21194156 0.21194156 0.57611688]
Вероятности после масштабирования с коэффициентом 0.1:
[0.31042377 0.31042377 0.37915245]
```

Как видите, масштабирование логитов с коэффициентом  $\alpha = 0.1$  дает вероятности, близкие к равномерным — [0.31, 0.31, 0.38]. А теперь мы можем сравнить порожденный текст при коэффициентах  $\alpha = 2.0$  и  $\alpha = 0.5$ :

- $\alpha = 2.0 \rightarrow$  больше предсказуемости:

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island',
...               scale_factor=2.0))
The island spoke of heavy torn into the island from the sea.

The noise of the inhabitants of the island was to be feared that
the colonists had come a project with a straight be put to the
bank of the island was the surface of the lake and sulphuric
acid, and several supply of her animals. The first stranger
carried a sort of accessible to break these screen barrels to
their distance from the palisade.

"The first huntil," said the reporter, "and his companions the
reporter extended to build a few days a
```

- $\alpha = 0.5 \rightarrow$  больше случайности:

```
>>> tf.random.set_seed(1)
>>> print(sample(model, starting_str='The island',
...               scale_factor=0.5))
The island
glissed in
ascercicedly useful? loigeh, Cyrus,
Spileots," henseporvemented
House to a left
the centlic moment. Tonsense crawl.

Pencrular ed/ of times," tading had coflently often above anzand?"

"Wat;" then:y."
```

Ardivify he acpearly, howcovered--he hassime; however,  
fenquests hen adgents!'.? Let us Neg eqiAl?.

GencNal, my surved thirtyin" ou; is Harding; treuths. Osew  
apartarned. "N,  
the poltuge of about-but durired with purteg.

Chappes wason!

Fears, " returned Spilett; "if  
you tear 8t trung

Результат показывает, что масштабирование логитов с коэффициентом  $\alpha = 0.5$  (увеличение температуры) приводит к порождению более случайного текста. Существует компромисс между новизной порожденного текста и его корректностью.

В этом разделе мы работали с порождением текста на уровне символов, что является задачей моделирования типа “последовательность в последовательность” (sequence-to-sequence — seq2seq). Хотя рассмотренный пример не особенно полезен сам по себе, легко придумать несколько практических приложений для моделей такого типа; скажем, похожую модель RNN можно обучить в качестве чатбота для оказания помощи пользователям с простыми запросами.

## Понимание языка с помощью модели “Преобразователь”

В главе мы решили две задачи моделирования с применением нейронных сетей на основе RNN. Тем не менее, недавно появилась новая архитектура, которая превзошла модели seq2seq, основанные на RNN, в ряде задач обработки естественного языка (NLP).

Архитектура называется “Преобразователь” (*Transformer*). Она способна моделировать глобальные зависимости между входными и выходными последовательностями и была представлена в 2017 году Ашишом Васвани и др. в статье “Attention Is All You Need” (Внимание — это все, что нужно), свободно доступной по ссылке <http://papers.nips.cc/paper/7181-attention-is-all-you-need>. Архитектура “Преобразователь” основана на концепции, которая называется *вниманием* (*attention*), точнее говоря, на *механизме самовнимания* (*self-attention mechanism*). Давайте возьмем задачу смыслового анализа, которая была раскрыта ранее в этой главе. В таком случае использование механизма внимания означает, что у нашей модели появится возможность научиться концентрироваться на частях входной последовательности, которые более существенны для отношения.

## Механизм самовнимания

В настоящем разделе мы объясним *механизм самовнимания* и покажем, как он помогает модели “Преобразователь” сфокусироваться на важных частях последовательности для обработки естественного языка. В первом подразделе мы обсудим самую элементарную форму самовнимания, чтобы пояснить общую идею, лежащую в основе изучения текстовых представлений. Затем мы добавим различные весовые параметры и подойдем к механизму самовнимания, который обычно применяется в моделях “Преобразователь”.

### Базовая версия самовнимания

Для представления базовой идеи, лежащей в основе самовнимания, давайте предположим, что у нас есть входная последовательность с длиной  $T$ ,  $\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(T)}$ , а также выходная последовательность,  $\mathbf{o}^{(0)}, \mathbf{o}^{(1)}, \dots, \mathbf{o}^{(T)}$ . Каждый элемент в этих последовательностях,  $\mathbf{x}^{(i)}$  и  $\mathbf{o}^{(i)}$ , является вектором размера  $d$  (т.е.  $\mathbf{x}^{(i)} \in \mathbb{R}^d$ ). Цель самовнимания для задачи seq2seq — смоделировать зависимости каждого элемента в выходной последовательности от входных элементов. Чтобы достичь указанной цели, механизмы внимания состоят из трех стадий. Во-первых, мы выводим веса важности, основываясь на подобии между текущим элементом и всеми остальными элементами в последовательности. Во-вторых, мы нормализуем веса, что обычно влечет за собой использование уже знакомой функции softmax. В-третьих, мы действуем эти веса в сочетании с соответствующими элементами последовательности для расчета значения внимания.

Выражаясь более формально, выходом самовнимания будет взвешенная сумма всей входной последовательности. Например, для  $i$ -того входного элемента соответствующее выходное значение вычисляется следующим образом:

$$\mathbf{o}^{(i)} = \sum_{j=0}^T \mathbf{W}_{ij} \mathbf{x}^{(j)}$$

Веса  $\mathbf{W}_{ij}$  рассчитываются на основе подобия между текущим входным элементом  $\mathbf{x}^{(i)}$  и всеми остальными элементами во входной последовательности. Говоря конкретнее, подобие вычисляется как скалярное произведение текущего входного элемента,  $\mathbf{x}^{(i)}$ , и другого элемента из входной последовательности,  $\mathbf{x}^{(j)}$ :

$$\omega_{ij} = \mathbf{x}^{(i)\top} \mathbf{x}^{(j)}$$

После расчета весов, основанных на подобии, для  $i$ -того входа и всех входов в последовательности (от  $\mathbf{x}^{(i)}$  до  $\mathbf{x}^{(T)}$ ) “сырые” веса (от  $\omega_{i0}$  до  $\omega_{iT}$ ) нормализуются с применением знакомой многопеременной функции (softmax):

$$\mathbf{W}_{ij} = \frac{\exp(\omega_{ij})}{\sum_{j=0}^T \exp(\omega_{ij})} = \text{softmax}([\omega_{ij}]_{j=0\dots T})$$

Отметим, что вследствие применения многопеременной функции после нормализации веса в сумме дают 1:

$$\sum_{j=0}^T \mathbf{W}_{ij} = 1$$

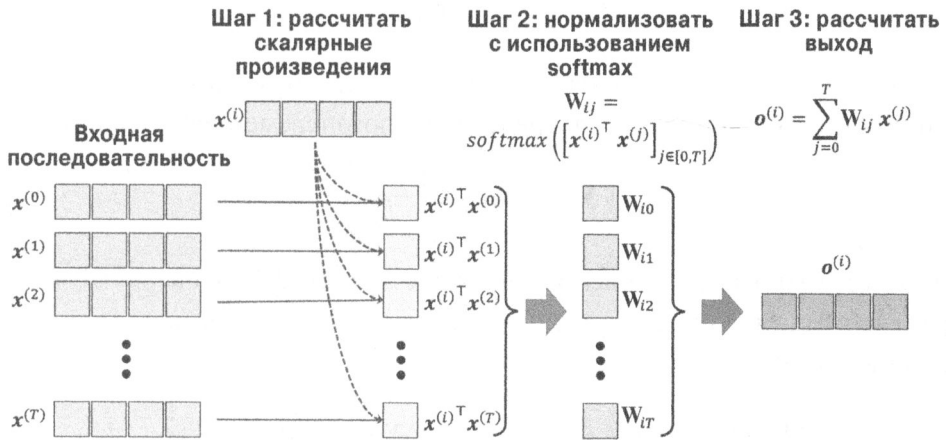
Подведем итоги по трем главным шагам, лежащим в основе операции самовнимания.

1. Для заданного входного элемента  $\mathbf{x}^{(i)}$  и каждого  $j$ -того элемента в диапазоне  $[0, T]$  рассчитать скалярное произведение  $\mathbf{x}^{(i)\top} \mathbf{x}^{(j)}$ .
2. Получить вес  $\mathbf{W}_{ij}$  путем нормализации скалярных произведений с использованием многопеременной функции.
3. Рассчитать выход  $\mathbf{o}^{(i)}$  как взвешенную сумму по всей входной последовательности:  $\mathbf{o}^{(i)} = \sum_{j=0}^T \mathbf{W}_{ij} \mathbf{x}^{(j)}$ .

Дополнительная иллюстрация этих шагов приведена на рис. 16.15.

### **Параметризация механизма самовнимания с помощью весов запросов, ключей и значений**

После ознакомления с базовой концепцией, лежащей в основе самовнимания, в этом подразделе подытоживается более сложный механизм самовнимания, который применяется в модели “Преобразователь”. Важно отметить, что в предыдущем подразделе при расчете выходов мы не задействовали ни одного обучаемого параметра. Следовательно, если мы хотим обучить языковую модель и затем изменить значения внимания для оптимизации цели, такой как минимизация ошибки классификации, тогда нам потребуется изменить вложения слов (т.е. входные векторы), которые лежат в основе каждого входного элемента  $\mathbf{x}^{(i)}$ .



**Рис. 16.15.** Дополнительная иллюстрация главных шагов, лежащих в основе операции самовнимания

Другими словами, при использовании ранее представленного базового механизма самовнимания модель “Преобразователь” довольно ограничена в отношении того, как она может обновлять либо изменять значения внимания во время оптимизации модели для заданной последовательности. Чтобы сделать механизм самовнимания более гибким и поддающимся оптимизации модели, мы введем три дополнительных весовых матрицы, которые можно подгонять в качестве параметров модели во время ее обучения. Мы обозначим эти три весовые матрицы как  $U_q$ ,  $U_k$  и  $U_v$ . Они применяются для проецирования входов на элементы последовательностей *запросов* (*query*), *ключей* (*key*) и *значений* (*value*):

- последовательность запросов —  $q^{(i)} = U_q x^{(i)}$  для  $i \in [0, T]$ ;
- последовательность ключей —  $k^{(i)} = U_k x^{(i)}$  для  $i \in [0, T]$ ;
- последовательность значений —  $v^{(i)} = U_v x^{(i)}$  для  $i \in [0, T]$ .

Здесь  $q^{(i)}$  и  $k^{(i)}$  являются векторами размера  $d_k$ . По этой причине матрицы проекций  $U_q$  и  $U_k$  имеют форму  $d_k \times d$ , тогда как  $U_v$  имеет форму  $d_v \times d$ . Ради простоты мы можем принять, что указанные векторы имеют одинаковую форму, например, используя  $m = d_k = d_v$ . Теперь вместо вычисления ненормализованного веса как попарного скалярного произведения заданного элемента входной последовательности,  $x^{(i)}$ , и  $j$ -того элемента последовательности,  $x^{(j)}$ , мы можем вычислить скалярное произведение запроса и ключа:

$$\omega_{ij} = \mathbf{q}^{(i)\top} \mathbf{k}^{(j)}$$

Далее мы можем применять  $m$ , точнее  $1/\sqrt{m}$ , для масштабирования веса  $\omega_{ij}$  перед его нормализацией посредством многопеременной функции:

$$\mathbf{W}_{ij} = \text{softmax}\left(\frac{\omega_{ij}}{\sqrt{m}}\right)$$

Следует отметить, что масштабирование  $\omega_{ij}$  с коэффициентом  $1/\sqrt{m}$  будет гарантировать, что евклидова длина весовых векторов окажется приблизительно в одном и том же диапазоне.

## Многоголовое внимание и блок “Преобразователь”

Еще одним трюком, значительно улучшающим различительную способность механизма самовнимания, является *многоголовое внимание* (*multi-head attention* — МНА), которое объединяет вместе множество операций самовнимания. В таком случае каждый механизм самовнимания называется *головой* (head) и может рассчитываться параллельно. При использовании  $r$  параллельных голов каждая голова дает вектор  $\mathbf{h}$  размера  $m$ . Затем эти векторы объединяются для получения вектора  $\mathbf{z}$  с формой  $r \times m$ . В заключение объединенный вектор проецируется с применением выходной матрицы  $\mathbf{W}^o$ , чтобы получить финальный выход:

$$\mathbf{o}^{(i)} = \mathbf{W}_{ij}^o \mathbf{z}$$

Архитектура блока “Преобразователь” показана на рис. 16.16.



Рис. 16.16. Архитектура блока “Преобразователь”

Обратите внимание, что в архитектуре блока “Преобразователь”, представленной на рис. 16.16, мы добавили два дополнительных компонента, которые пока еще не обсуждали. Одним из них является *остаточной связью* (*residual connection*), которая складывает выход из слоя (или даже группы слоев) с его входом, т.е.  $x + \text{слой}(x)$ . Блок, состоящий из слоя (или множества слоев) с такой остаточной связью, называется *остаточным блоком*. Блок “Преобразователь”, показанный на рис. 16.16, имеет два остаточных блока.

Еще один новый компонент — *нормализация по слою* (*layer normalization*). Существует семейство слоев нормализации, включающее пакетную нормализацию, которую мы рассмотрим в главе 17. Пока можете думать о нормализации по слою как о причудливом или более развитом способе нормализации либо масштабирования входов и активаций нейронной сети в каждом слое.

Возвращаясь к иллюстрации модели типа “Преобразователь” на рис. 16.16, давайте посмотрим, как она работает. Первым делом входная последовательность передается слоям МНА, которые основаны на обсуждаемом ранее механизме самовнимания. Кроме того, входные последовательности складываются с выходом слоев МНА посредством остаточных связей. Это гарантирует, что более ранние слои получают достаточные градиентные сигналы во время обучения, и является распространенным трюком, который используется для улучшения показателей скорости и сходимости обучения. В случае заинтересованности вы можете узнать больше о концепции, лежащей в основе остаточных связей, из исследовательской статьи “Deep Residual Learning for Image Recognition” (Глубокое остаточное обучение для распознавания изображений), которую написали Кайминг Хе, Сяньгу Чжан, Шаоцин Рен и Цзянь Сунь ([http://openaccess.thecvf.com/content\\_cvpr\\_2016/html/He\\_Deep\\_Residual\\_Learning\\_CVPR\\_2016\\_paper.html](http://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html)).

После добавления входных последовательностей к выходу слоев МНА выходы нормализуются через нормализацию по слою. Такие нормализованные сигналы затем проходят через ряд слоев MLP (т.е. полносвязных), которые также имеют остаточную связь. Наконец, выход из остаточного блока снова нормализуется и возвращается в качестве выходной последовательности, которую можно применять для классификации или порождения последовательностей.

Инструкции по реализации и обучению моделей типа “Преобразователь” были опущены в целях экономии пространства. Однако заинтересованные чи-



татели могут найти великолепную реализацию с объяснениями в официальной документации TensorFlow по ссылке <https://www.tensorflow.org/tutorials/text/transformer>.

## Резюме

В главе вы сначала ознакомились с характеристиками последовательностей, которые отличают их от других типов данных, таких как структурированные данные либо изображения. Затем мы раскрыли основы сетей RNN для моделирования последовательностей. Вы узнали, как работает базовая модель RNN, и выяснили ее ограничения относительно выявления долгосрочных зависимостей в данных последовательности. Далее мы обсудили ячейки LSTM, которые состоят из шлюзового механизма для сокращения эффекта от проблем взрывного роста и исчезновения градиентов, которые часто возникают в моделях RNN.

После рассмотрения главных концепций, лежащих в основе сетей RNN, мы реализовали несколько моделей RNN с различными рекуррентными слоями, используя API-интерфейс Keras. В частности, мы создали модель RNN для смыслового анализа, а также модель RNN для порождения текста. В заключение мы исследовали модель типа “Преобразователь”, которая задействует механизм самовнимания, чтобы сфокусироваться на важных частях последовательности.

В следующей главе вы узнаете о порождающих моделях и в частности о *порождающих состязательных сетях* (*generative adversarial network* — GAN), которые продемонстрировали замечательные результаты для разнообразных задач в области компьютерного зрения.

# ПОРОЖДАЮЩИЕ СОСТЯЗАТЕЛЬНЫЕ СЕТИ ДЛЯ СИНТЕЗА НОВЫХ ДАННЫХ

В предыдущей главе внимание было сосредоточено на *рекуррентных нейронных сетях* для моделирования последовательностей. В настоящей главе мы займемся исследованием *порождающих состязательных сетей* (*generative adversarial network* — GAN) и их применением в синтезе новых образцов данных. Сети GAN считаются самым важным прорывом в области ГО, позволяя компьютерам порождать новые данные (такие как новые изображения). Мы раскроем в главе следующие темы:

- введение в порождающие модели для синтеза новых данных;
- автокодировщики, *вариационные автокодировщики* (*variational autoencoder* — VAE) и их родство с сетями GAN;
- строительные блоки сетей GAN;
- реализация простой модели GAN для порождения изображений рукописных цифр;
- понятие транспонированной свертки и *пакетной нормализации* (*batch normalization* — BatchNorm или BN);
- совершенствование сетей GAN: глубокие сверточные порождающие состязательные сети и порождающие состязательные сети, использующие *расстояние Вассерштейна* (*Wasserstein distance*).

## Понятие порождающих состязательных сетей

Давайте сначала рассмотрим основы моделей GAN. Общая цель сети GAN заключается в синтезе новых данных, которые имеют такое же распределение, как в обучающем наборе. Следовательно, сети GAN в своей первоначальной форме считаются категорией задач МО без учителя, поскольку никаких помеченных данных не требуется. Однако полезно отметить, что расширения, внесенные в первоначальные сети GAN, могут охватывать задачи частичного обучения и обучения с учителем.

Общая концепция сети GAN впервые была предложена в 2014 году Яном Гудфеллоу и его коллегами как метод для синтеза новых изображений с применением глубоких нейронных сетей (Ян Гудфеллоу, Жан Пуже-Абади, Мехди Мирза, Бинг Ксу, Дэвид Уорд-Фарли, Шерджил Озаир, Аарон Курвиль и Йошуа Бенджи, “Generative Adversarial Nets” (Порождающие состязательные сети), *Proceedings of the 27th International Conference on Neural Information Processing Systems 2* (2014 г.): с. 2672–2680). Хотя исходная архитектура GAN, предложенная в указанной статье, подобно архитектурам многослойных персептронов основывалась на полносвязных слоях и обучалась порождению изображений рукописных цифр с низким разрешением вроде MNIST, в большей степени она послужила доказательством осуществимости этого нового подхода.

Тем не менее, после представления архитектуры GAN первоначальные авторы и многочисленные исследователи предложили множество улучшений и различных приложений в разных областях науки и техники. Например, в компьютерном зрении сети GAN используются для трансляции изображений в изображения (выяснения, каким образом отобразить входное изображение на выходное изображение), супер-разрешения изображений (создание изображения с высоким разрешением из версии с низким разрешением), дополнения изображений (выяснения, как воссоздать недостающие части изображения) и решения многих других задач. Скажем, последние достижения в исследованиях архитектуры GAN привели к появлению моделей, которые способны генерировать новые изображения лиц с высоким разрешением. Примеры таких изображений с высоким разрешением можно найти на веб-сайте <https://www.thispersondoesnotexist.com/>, где демонстрируются искусственные изображения лиц, сгенерированные сетью GAN.

## Начало работы с автокодировщиками

Прежде чем приступить к обсуждению работы сетей GAN, мы начнем с автокодировщиков, которые могут сжимать и восстанавливать обучающие данные. Хотя стандартные автокодировщики не способны генерировать новые данные, понимание их функциональности поможет вам ориентироваться в сетях GAN в следующем разделе.

Автокодировщики состоят из двух сетей, соединенных вместе: сети кодировщика и сети декодировщика. Сеть кодировщика получает  $d$ -мерный вектор входных признаков, ассоциированный с образцом  $x$  (т.е.  $x \in R^d$ ), и кодирует его в  $p$ -мерный вектор  $z$  (т.е.  $z \in R^p$ ). Другими словами, роль кодировщика заключается в том, чтобы научиться моделировать функцию  $z = f(x)$ . Закодированный вектор  $z$  также называется латентным вектором или латентным представлением признаков. Обычно размерность латентного вектора меньше размерности входных признаков, т.е.  $p < d$ . Следовательно, мы можем говорить, что кодировщик действует в качестве функции сжатия данных. Затем декодировщик восстанавливает  $\hat{x}$  из латентного вектора меньшей размерности  $z$ , причем мы можем считать декодировщик функцией  $\hat{x} = g(z)$ . Простая архитектура автокодировщика показана на рис. 17.1, где части кодировщика и декодировщика состоят из единственного полносвязного слоя.

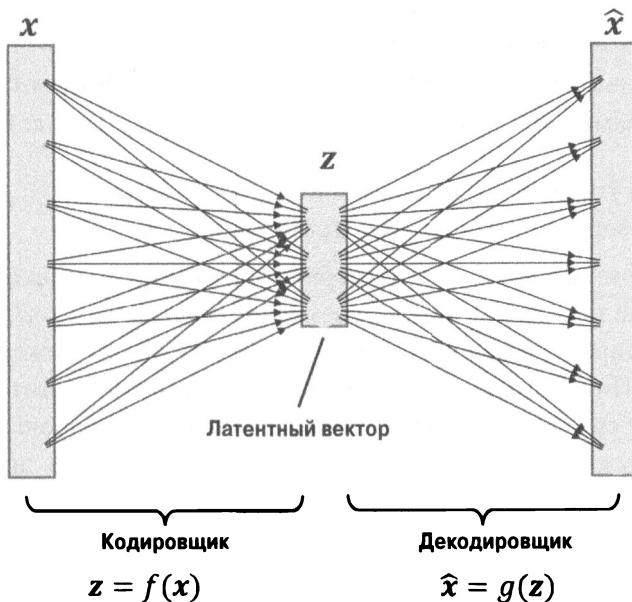


Рис. 17.1. Простая архитектура автокодировщика



### Связь между автокодировщиками и понижение размерности

В главе 5 вы узнали о методиках понижения размерности, таких как анализ главных компонент (PCA) и линейный дискриминантный анализ (LDA). Автокодировщики тоже могут применяться в качестве методики понижения размерности. На самом деле, когда нелинейность в любой из двух подсетей (кодировщика и декодировщика) отсутствует, то подход с автокодировщиком *почти идентичен* PCA.

В данном случае, если мы предположим, что веса однослойного кодировщика (без скрытых слоев и без нелинейных функций активации) обозначаются с помощью матрицы  $U$ , тогда кодировщик моделирует  $z = U^T x$ . Подобным образом однослойный линейный декодировщик моделирует  $\hat{x} = Uz$ . Объединяя вместе указанные два компонента, мы получаем  $\hat{x} = UU^T x$ . Именно это делает PCA за исключением того, что PCA имеет дополнительное ортонормальное ограничение:  $UU^T = I_{n \times n}$ .

Хотя на рис. 17.1 изображен автокодировщик без скрытых слоев внутри кодировщика и декодировщика, конечно же, мы можем добавить множество скрытых слоев с нелинейностями (как в многослойной нейронной сети) для создания автокодировщика, который сумеет изучать более эффективные функции сжатия и восстановления данных. Кроме того, имейте в виду, что обсуждаемый в текущем разделе автокодировщик использует полносвязные слои. Однако, как известно из главы 15, когда мы работаем с изображениями, то можем замещать полносвязные слои сверточными слоями.



### Другие типы автокодировщиков, основанные на размере латентного пространства

Как упоминалось ранее, размерность латентного пространства автокодировщика обычно ниже размерности входов ( $p < d$ ), что делает автокодировщики подходящим средством для понижения размерности. По этой причине на латентный вектор также часто ссылаются как на “сужение”, а такую конфигурацию автокодировщика называют *понижающей* (*undercomplete*). Тем не менее, есть другая категория автокодировщиков, называемая *повышающей* (*overcomplete*), где размерность латентного вектора  $z$  в действительности больше размерности входных образцов ( $p > d$ ).

При обучении повышающего автокодировщика существует тривиальное решение, когда кодировщик и декодировщик могут просто учиться копировать (заучивать) входные признаки в свой выходной слой. Очевидно, такое решение не особенно полезно. Однако благодаря внесению в процедуру обучения ряда модификаций повышающие автокодировщики можно применять для *уменьшения шума*.

В данном случае во время обучения к входным образцам добавляется случайный шум  $\varepsilon$ , а сеть учится восстанавливать чистый образец  $x$  из зашумленного сигнала  $x + \varepsilon$ . Затем во время оценки мы предоставляем новые образцы, которые естественным образом зашумлены (т.е. шум уже присутствует, так что никакой дополнительный искусственный шум  $\varepsilon$  не добавляется), чтобы удалить из них существующий шум. Такая архитектура автокодировщика называется *шумоподавляющей* (*denoising*).

## Порождающие модели для синтеза новых данных

Автокодировщики являются детерминированными моделями, а это значит, что после обучения автокодировщик при заданном входе  $x$  будет способен воссоздать вход из его сжатой версии в пространстве меньшей размерности. Следовательно, он не может генерировать новые данные помимо воссоздания своего входа через трансформацию сжатого представления.

С другой стороны, порождающая модель в состоянии генерировать новый экземпляр  $\tilde{x}$  из случайного вектора  $z$  (соответствующего латентному представлению). На рис. 17.2 показано схематическое представление порождающей модели. Случайный вектор  $z$  поступает из простого распределения с полностью известными характеристиками, так что мы можем легко произвести из него выборку. Например, каждый элемент  $z$  может происходить из равномерного распределения в диапазоне  $[-1, 1]$  (для которого мы записываем  $z_i \sim \text{Uniform}(-1, 1)$ ) либо из стандартного нормального распределения (в таком случае мы записываем  $z_i \sim \text{Normal}(\mu = 0, \sigma^2 = 1)$ ).

Поскольку мы переключили внимание с автокодировщиков на порождающие модели, вы могли заметить, что компонент декодировщика имеет определенные сходные черты с порождающей моделью. В частности, оба получают на входе латентный вектор  $z$  и возвращают выход в том же пространстве, что и  $x$ . (Для автокодировщика  $\hat{x}$  — реконструкция входа  $x$ , а для порождающей модели  $\tilde{x}$  — синтезированный образец.)

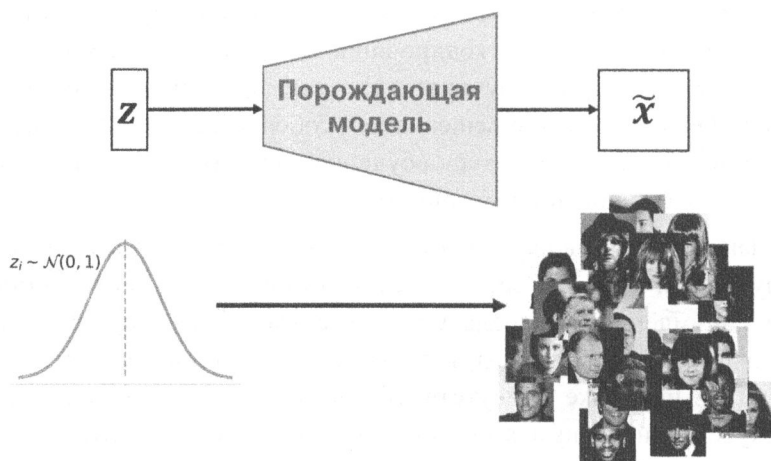


Рис. 17.2. Схематическое представление порождающей модели

Тем не менее, крупное отличие между автокодировщиком и порождающей моделью заключается в том, что в автокодировщике мы не знаем распределение  $z$ , тогда как в порождающей модели все характеристики распределения  $z$  известны. Однако автокодировщик можно обобщить до порождающей модели. Одним из подходов являются *вариационные автокодировщики (VAE)*.

В вариационном автокодировщике, получающем входной образец  $x$ , сеть кодировщика модифицируется так, чтобы вычислять два момента распределения латентного вектора: среднее  $\mu$  и дисперсию  $\sigma^2$ . Во время обучения вариационного автокодировщика сеть вынуждена сопоставлять эти моменты с моментами стандартного нормального распределения (т.е. нулевым средним и единичной дисперсией). Затем после того, как модель VAE обучена, кодировщик отбрасывается, и мы можем использовать сеть декодировщика для генерации новых образцов  $\tilde{x}$ , подавая случайные векторы  $z$  из “выученного” гауссова распределения.

Кроме вариационных автокодировщиков существуют другие типы порождающих моделей, скажем, *автогреессионные модели (autoregressive model)* и *нормализующие потоковые модели (normalizing flow model)*. Тем не менее, в этой главе мы планируем уделять внимание только моделям GAN, которые являются самыми недавними и наиболее популярными типами порождающих моделей в ГО.



## Что такое порождающая модель?

Обратите внимание, что порождающие модели традиционно определяются как алгоритмы, которые моделируют распределение входных данных,  $p(x)$ , или объединенные распределения входных данных и ассоциированных целей,  $p(x, y)$ . По определению такие модели также способны производить выборку из какого-то признака,  $x$ , обусловленную другим признаком,  $y$ , что известно под названием *условное выведение*. Однако в контексте ГО термин “порождающая модель” обычно применяется для ссылки на модели, которые генерируют реалистично выглядящие данные. Это означает, что мы можем осуществлять выборку из распределения входных данных,  $p(x)$ , но не обязательно будем в состоянии выполнять условное выведение.

## Генерирование новых образцов с помощью порождающих состязательных сетей

Чтобы кратко объяснить функционирование сетей GAN, давайте сначала предположим, что у нас есть сеть, которая получает случайный вектор  $z$ , выбранный из известного распределения, и генерирует выходное изображение  $x$ . Мы будем называть такую сеть *генератором* (*generator* —  $G$ ) и использовать обозначение  $\tilde{x} = G(z)$  для ссылки на сгенерированный выход. Пусть наша цель заключается в генерировании ряда изображений, например, изображений лиц, зданий, животных или даже рукописных цифр, как в наборе данных MNIST.

Как всегда, мы будем инициализировать эту сеть случайными весами. Следовательно, первые выходные изображения до регулировки весов будут похожи на белый шум. Теперь представим, что имеется функция, которая способна оценить качество изображений (назовем ее *функцией оценщика* (*assessor function*)).

Если такая функция существует, тогда мы можем задействовать отклик из этой функции для сообщения нашей сети генератора о том, как регулировать ее веса, чтобы улучшить качество генерируемых изображений. Подобным образом мы можем обучать генератор на основе отклика от функции оценщика, так что генератор учится совершенствовать выход, производя реалистично выглядящие изображения.

Наряду с тем, что функция оценщика, как было описано в предыдущем абзаце, очень облегчила бы задачу генерирования изображений, вопрос в



том, существует ли такая универсальная для оценки качества изображений, и если да, то как она определяется. Очевидно, будучи людьми, мы способны легко оценить качество выходных изображений, наблюдая за выходами сети; правда, мы не можем (пока) обеспечить обратное распространение результата из нашего мозга в сеть. Итак, если наш мозг в состоянии оценить качество синтезированных изображений, то можем ли мы спроектировать нейросетевую модель, которая делала бы то же самое? По сути это и есть основная идея сети GAN. Как показано на рис. 17.3, модель GAN состоит из дополнительной нейронной сети под названием *дискриминатор* (*discriminator* —  $D$ ), являющейся классификатором, который учится отличать синтезированное изображение  $\tilde{x}$  от настоящего изображения  $x$ .

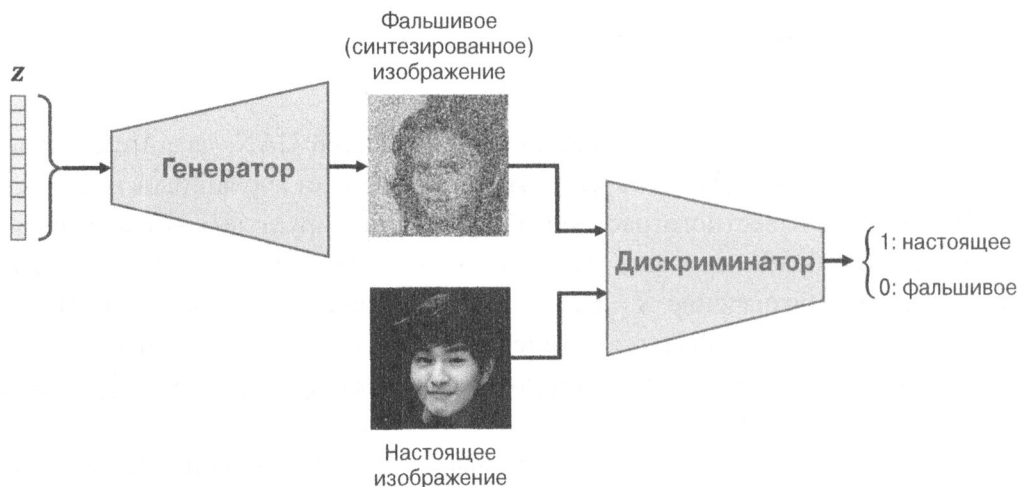


Рис. 17.3. Модель GAN

В модели GAN две сети — генератора и дискриминатора — обучаются вместе. Сначала после инициализации весов модели генератор создает изображения, которые не выглядят реалистичными. Аналогично дискриминатор плохо проводит различие между настоящими изображениями и изображениями, синтезированными генератором. Но с течением времени (т.е. на протяжении обучения) обе сети становятся лучше по мере того, как взаимодействуют друг с другом. Фактически две сети играют в состязательную игру, где генератор учится совершенствовать свой выход, чтобы суметь обмануть дискриминатор. Одновременно дискриминатор становится лучше в выявлении синтезированных изображений.

## Функции потерь сетей генератора и дискриминатора в модели GAN

Целевая функция сетей GAN, как описано в исходной статье “Generative Adversarial Nets” (Порождающие состязательные сети) Яна Гудфеллоу и др. (<https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>), выглядит следующим образом:

$$V(\Theta^{(D)}, \Theta^{(G)}) = E_{x \sim p_{\text{данных}}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

Здесь  $V(\Theta^{(D)}, \Theta^{(G)})$  называется *функцией ценности*, которая может интерпретироваться как вознаграждение: мы хотим довести до максимума ее значение относительно дискриминатора ( $D$ ), одновременно сводя к минимуму ее значение в отношении генератора ( $G$ ), т.е.  $\min_G \max_D V(\Theta^{(D)}, \Theta^{(G)})$ . Далее,  $D(x)$  — вероятность, которая указывает, является ли входной образец  $x$  настоящим или фальшивым (т.е. сгенерированным). Выражение  $E_{x \sim p_{\text{данных}}(x)} [\log D(x)]$  ссылается на ожидаемое значение величины в квадратных скобках относительно образцов из распределения данных (распределения настоящих образцов);  $E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$  ссылается на ожидаемое значение этой величины относительно распределения входных векторов  $z$ .

Один шаг обучения модели GAN с такой функцией ценности требует двух шагов оптимизации: (1) доведение до максимума вознаграждения для дискриминатора и (2) сведение к минимуму вознаграждения для генератора. Практичный способ обучения сетей GAN предусматривает переключение между двумя шагами оптимизации: (1) фиксация (замораживание) параметров одной сети и оптимизация весов другой сети; (2) фиксация параметров второй сети и оптимизация весов первой сети. Такой процесс должен повторяться на каждой итерации процесса обучения. Предположим, что сеть генератора зафиксирована, и мы хотим оптимизировать дискриминатор. Оба члена в функции ценности  $\Theta$  вносят вклад в оптимизацию дискриминатора, где первый член соответствует потере, ассоциированной с настоящими образцами, а второй член представляет собой потерю для фальшивых образцов. Следовательно, когда  $G$  фиксируется, наша цель заключается в *доведении до максимума*  $V(\Theta^{(D)}, \Theta^{(G)})$ , что означает улучшение дискриминатора в плане его возможности проводить различия между настоящими и фальшивыми изображениями.

После оптимизации дискриминатора с применением членов потери для настоящих и фальшивых образцов мы фиксируем дискриминатор и оптимизируем генератор. В этом случае только второй член в  $V(\Theta^{(D)}, \Theta^{(G)})$  вносит вклад в градиенты генератора. В итоге, когда  $D$  фиксируется, наша цель заключается в *сведении к минимуму*  $V(\Theta^{(D)}, \Theta^{(G)})$ , что можно записать в виде  $\min_G E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$ . Как упоминалось в исходной статье Гудфеллоу и др. о сетях GAN, на ранних стадиях обучения данная функция,  $\log(1 - D(G(z)))$ , страдает от проблемы исчезновения градиентов. Причина в том, что на ранних стадиях обучения выходы  $G(z)$  совершенно не похожи на настоящие образцы, а потому функция  $D(G(z))$  с высокой вероятностью будет близка к нулю. Такое явление называется *насыщением*. Чтобы решить проблему, мы можем переформулировать цель минимизации  $\min_G E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$ , записав ее как  $\max_G E_{z \sim p_z(z)} [\log(D(G(z)))]$ .

Приведенная замена означает, что для обучения генератора мы можем поменять местами метки настоящих и фальшивых образцов и выполнить обычную минимизацию функции. Другими словами, хотя образцы, синтезированные генератором, являются фальшивыми и потому помеченными как 0, мы можем переключить метки, назначив этим образцам метку 1, и *минимизировать* потерю двоичной перекрестной энтропии с новыми метками вместо максимизации  $\max_G E_{z \sim p_z(z)} [\log(D(G(z)))]$ .

Теперь, когда мы раскрыли общую процедуру оптимизации для обучения моделей GAN, давайте займемся исследованием разнообразных меток данных, которые можно использовать при обучении сетей GAN. С учетом того, что дискриминатор является двоичным классификатором (с метками классов 0 и 1 соответственно для настоящих и фальшивых изображений), мы можем применять функцию потерь двоичной перекрестной энтропии. Таким образом, мы можем определить достоверные метки для потери дискриминатора следующим образом:

$$\text{Достоверные метки для дискриминатора} = \begin{cases} 1: & \text{для настоящих изображений, т.е. } x \\ 0: & \text{для выходов } G, \text{ т.е. } G(z) \end{cases}$$

А как обстоит дело с метками для обучения генератора? Поскольку мы хотим, чтобы генератор синтезировал реалистичные изображения, то его нужно штрафовать, когда дискриминатор не классифицирует выходы гене-

ратора как настоящие изображения. Таким образом, при вычислении функции потерь генератора мы будем предполагать, что достоверные метки для его выходов равны 1.

На рис. 17.4 собраны вместе все шаги работы простой модели GAN.

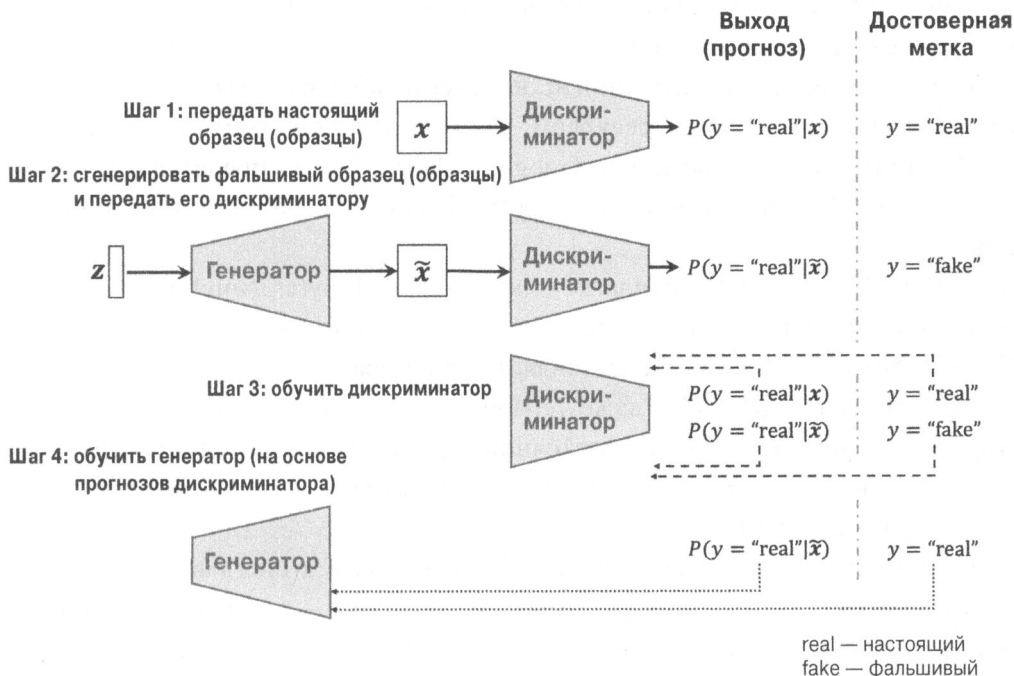


Рис. 17.4. Работа простой модели GAN

В следующем разделе мы реализуем сеть GAN для генерирования новых изображений рукописных цифр.

## Реализация порождающей состязательной сети с нуля

В этом разделе мы покажем, каким образом реализовать и обучить модель GAN для генерирования новых изображений, подобных изображениям цифр в наборе данных MNIST. Так как обучение на обычном центральном процессоре (ЦП) может требовать много времени, в последующих подразделах мы продемонстрируем настройку среды Google Colab, которая позволит запускать вычисления на графических процессорах (ГП).

## Обучение моделей GAN в среде Google Colab

Для некоторых примеров в настоящей главе могут требоваться большие вычислительные ресурсы, выходящие за рамки рядового ноутбука или рабочей станции без ГП. Если вам доступен компьютер с графическим процессором NVIDIA и установленными библиотеками CUDA и cuDNN, то вы можете ускорить вычисления.

Тем не менее, поскольку многие не имеют доступа к высокопроизводительным вычислительным ресурсам, мы будем использовать среду Google Colaboratory (часто называемую Google Colab), которая представляет собой бесплатную облачную вычислительную службу (доступную в большинстве стран).

Среда Google Colab предлагает экземпляры Jupyter Notebook, которые функционируют в облаке; тетради могут храниться на Google Диске или GitHub. Хотя платформа предоставляет разнообразные вычислительные ресурсы, такие как ЦП, ГП и даже тензорные процессоры (ТП), важно отметить, что время выполнения в настоящий момент ограничивается 12 часами. Следовательно, процесс выполнения любой тетради, который длится более 12 часов, будет прерван.

Максимальное время выполнения блоков кода в этой главе составит от двух до трех часов, а потому указанное ограничение — не проблема. Однако если вы решите применять Google Colab для других проектов, требующих более 12 часов, тогда обеспечьте сохранение промежуточных контрольных точек.



### Jupyter Notebook

Jupyter Notebook представляет собой графический пользовательский интерфейс для интерактивного выполнения кода, а также его снабжения текстовой документацией и рисунками. Благодаря своей универсальности и удобству в эксплуатации он стал одним из самых популярных инструментов в науке о данных.

Дополнительные сведения о графическом пользовательском интерфейсе Jupyter Notebook ищите в официальной документации по ссылке <https://jupyter-notebook.readthedocs.io/en/stable/>. Полный код примеров, рассмотренных в книге, также доступен в виде тетрадей Jupyter Notebook, и в каталоге кода для первой главы предлагается краткое введение в Jupyter Notebook:

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/tree/master/ch01#pythonjupyter-notebook>.

В заключение мы настоятельно рекомендуем почитать статью Адама Рула и др. “Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks” (Десять простых правил написания и совместного использования вычислительных анализов в тетрадах Jupyter Notebook), посвященную эффективному использованию Jupyter Notebook в научно-исследовательских проектах, которая свободно доступна по ссылке <https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007007>.

Получить доступ к Google Colab очень легко. Вы можете перейти на веб-сайт <https://colab.research.google.com>, который автоматически отобразит окно подсказки, где будут видны существующие тетради Jupyter Notebook. Щелкните на вкладке GOOGLE DRIVE (Google Диск), т.к. именно на Google Диске вы будете хранить тетрадь.

Чтобы создать новую тетрадь, щелкните на ссылке NEW PYTHON 3 NOTEBOOK (Новая тетрадь Python 3) в нижней части окна подсказки (рис. 17.5).



Рис. 17.5. Создание новой тетради Python 3

Будет создана и открыта новая тетрадь. Все примеры кода, которые вы напишете в этой тетради, будут автоматически сохраняться, и позже вы сможете получить доступ к тетради в каталоге по имени Colab Notebooks на Google Диске.

На следующем шаге будут задействованы ГП для выполнения примеров кода в этой тетради. Щелкните на пункте Change runtime type (Изменить тип исполняющей среды) в меню Runtime (Исполняющая среда) тетради и выберите в раскрывающемся списке Hardware accelerator (Аппаратный ускоритель) вариант GPU (ГП), как показано на рис. 17.6.

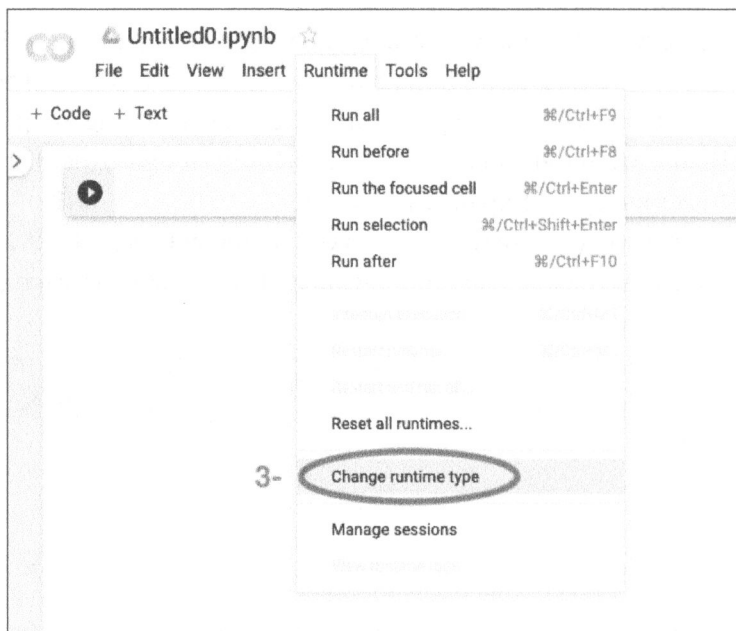


Рис. 17.6. Выбор ГП в качестве аппаратного ускорителя

На последнем шаге нужно лишь установить пакеты Python, которые понадобятся в главе. Среда Colab Notebooks поступает с определенными пакетами, куда входят NumPy, SciPy и последняя стабильная версия TensorFlow. Тем не менее, на момент написания главы последней стабильной версией в Google Colab была TensorFlow 1.15.0, а нас интересует TensorFlow 2.0. Следовательно, необходимо установить TensorFlow 2.0 с поддержкой ГП, выполнив показанную ниже команду в новой ячейке тетради:

```
! pip install -q tensorflow-gpu==2.0.0
```

(Ячейка тетради Jupyter Notebook, начинающаяся с восклицательного знака, интерпретируется как команда оболочки Linux.)

Теперь можно протестировать установку и удостовериться в доступности ГП с помощью такого кода:

```
>>> import tensorflow as tf
>>> print(tf.__version__)
'2.0.0'
>>> print("Доступность ГП:", tf.test.is_gpu_available())
Доступность ГП: True
>>> if tf.test.is_gpu_available():
...     device_name = tf.test.gpu_device_name()
... else:
...     device_name = '/CPU:0'
>>> print(device_name)
'/device:GPU:0'
```

Кроме того, если вы хотите сохранить модель на своем Google Диске, переместить или загрузить другие файлы, то должны смонтировать Google Диск, для чего выполнить приведенный далее код в новой ячейке тетради:

```
>>> from google.colab import drive
>>> drive.mount('/content/drive/')

```

Результатом будет ссылка для аутентификации доступа к вашему Google Дisku. После следования инструкциям по аутентификации появится код, который вы должны скопировать и вставить в указанное поле ввода ниже только что выполненной ячейки. Затем ваш Google Диск будет смонтирован и станет доступным по ссылке `/content/drive/My Drive`.

## Реализация сетей генератора и дискриминатора

Мы начнем реализацию нашей первой модели GAN с генератора и дискриминатора как двух полносвязных сетей с одним или большим числом скрытых слоев (рис. 17.7).

Так выглядит исходная версия GAN, на которую мы будем ссылаться как на *простую сеть GAN*.

В показанной на рис. 17.7 модели для каждого скрытого слоя мы применяем функцию активации на основе ReLU с утечкой.



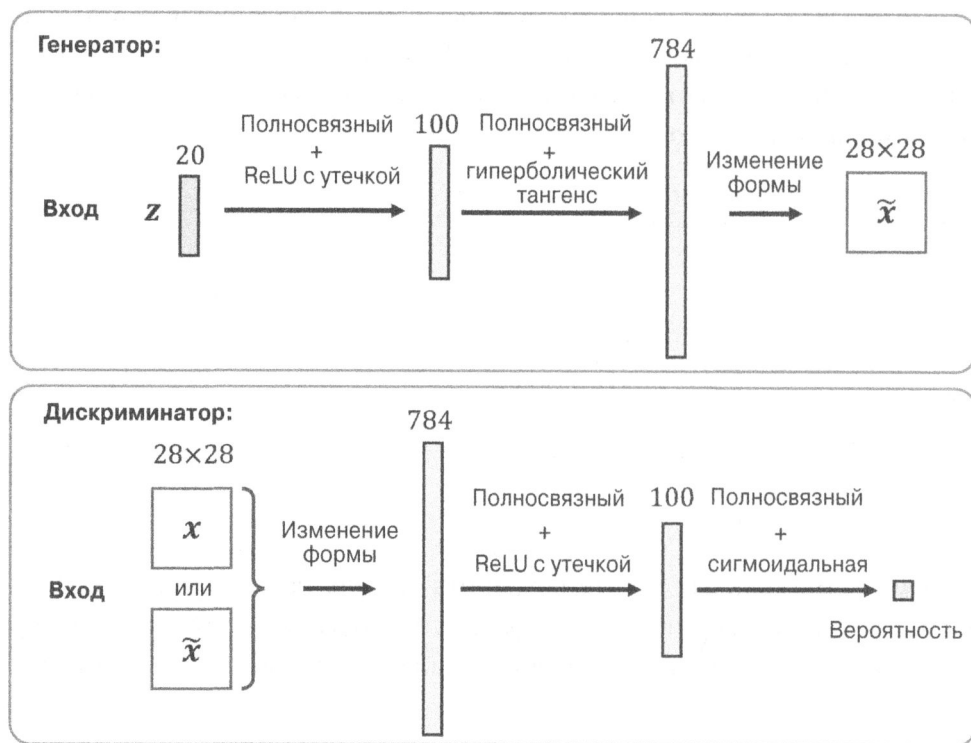


Рис. 17.7. Простая модели GAN

Использование ReLU дает в результате разреженные градиенты, что может оказаться неподходящим, когда желательно иметь градиенты для полного диапазона входных значений. В сети дискриминатора за каждым скрытым слоем следует слой отключения. Вдобавок выходной слой в сети генератора задействует функцию активации в виде гиперболического тангенса ( $\tanh$ ). (Для сети генератора рекомендуется применять функцию активации  $\tanh$ , т.к. она содействует обучению.)

Выходной слой сети дискриминатора не имеет функции активации (т.е. активация линейная), чтобы в итоге получались логиты. В качестве альтернативы мы можем использовать сигмоидальную функцию активации для получения на выходе вероятностей.

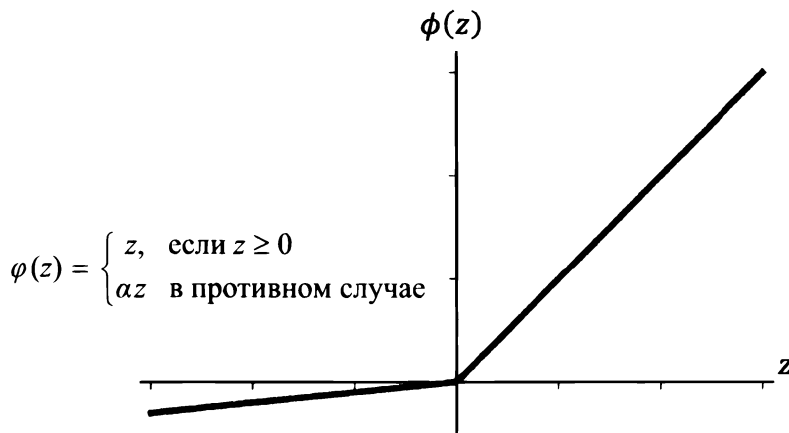


На  
заметку!

### Функция активации в виде выпрямленного линейного элемента (ReLU) с утечкой

В главе 13 были раскрыты различные нелинейные функции активации, которые могут применяться в нейросетевой модели. Вспомните, что функция активации ReLU определялась как  $\phi(z) = \max(0, z)$ , подавляя отрицательные входы (предварительную активацию) за счет их установки в нули. Как следствие, использование функции активации ReLU может привести к получению разреженных градиентов во время обратного распространения. Разреженные градиенты не всегда вредны и могут даже приносить пользу моделям для классификации. Однако в приложениях вроде сетей GAN иногда полезно получать градиенты для полного диапазона входных значений, чего мы можем достичь, внося в функцию ReLU небольшое изменение, чтобы она выдавала небольшие значения при отрицательных входах. Такая модифицированная версия функции ReLU также известна как *ReLU с утечкой* (*leaky ReLU*). Короче говоря, функция активации в виде ReLU с утечкой также допускает ненулевые градиенты для отрицательных входов и в результате делает сети более выразительными в целом.

Функция активации в виде ReLU с утечкой определяется следующим образом:



Здесь  $\alpha$  задает наклон для отрицательных входов (предварительной активации).

Для каждой из двух сетей мы определим две вспомогательные функции, создадим объект модели из класса `Sequential` библиотеки `Keras` и добавим слои, как было описано ранее. Вот как выглядит код:

```
>>> import tensorflow as tf
>>> import tensorflow_datasets as tfds
>>> import numpy as np
>>> import matplotlib.pyplot as plt

>>> ## определение функции для генератора:
>>> def make_generator_network(
...     num_hidden_layers=1,
...     num_hidden_units=100,
...     num_output_units=784):
...
...     model = tf.keras.Sequential()
...     for i in range(num_hidden_layers):
...         model.add(
...             tf.keras.layers.Dense(
...                 units=num_hidden_units, use_bias=False))
...         model.add(tf.keras.layers.LeakyReLU())
...
...     model.add(
...         tf.keras.layers.Dense(
...             units=num_output_units, activation='tanh'))
...     return model

>>> ## определение функции для дискриминатора:
>>> def make_discriminator_network(
...     num_hidden_layers=1,
...     num_hidden_units=100,
...     num_output_units=1):
...
...     model = tf.keras.Sequential()
...     for i in range(num_hidden_layers):
...         model.add(
...             tf.keras.layers.Dense(units=num_hidden_units))
...         model.add(tf.keras.layers.LeakyReLU())
...         model.add(tf.keras.layers.Dropout(rate=0.5))
...
...     model.add(
...         tf.keras.layers.Dense(
...             units=num_output_units, activation=None))
...     return model
```

Затем мы укажем настройки обучения. Как вы помните из предшествующих глав, размер изображения в наборе данных MNIST составляет  $28 \times 28$  пикселей. (Имеется лишь один цветовой канал, потому что MNIST содержит изображения в оттенках серого.) Мы дополнительно укажем размер входного вектора  $z$ , равный 20, а для инициализации весов модели будем применять случайное равномерное распределение. Поскольку мы реализуем очень простую модель GAN только в целях иллюстрации и используем полносвязные слои, в каждой сети будет присутствовать единственный скрытый слой с сотней элементов. В следующем коде мы определим и инициализируем две сети и выведем итоговую информацию о них:

```
>>> image_size = (28, 28)
>>> z_size = 20
>>> mode_z = 'uniform'    # 'uniform' или 'normal'
>>> gen_hidden_layers = 1
>>> gen_hidden_size = 100
>>> disc_hidden_layers = 1
>>> disc_hidden_size = 100
>>> tf.random.set_seed(1)
>>> gen_model = make_generator_network(
...     num_hidden_layers=gen_hidden_layers,
...     num_hidden_units=gen_hidden_size,
...     num_output_units=np.prod(image_size))
>>> gen_model.build(input_shape=(None, z_size))
>>> gen_model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	multiple	2000
leaky_re_lu (LeakyReLU)	multiple	0
dense_1 (Dense)	multiple	79184
Total params: 81,184		
Trainable params: 81,184		
Non-trainable params: 0		

```
>>> disc_model = make_discriminator_network(
...     num_hidden_layers=disc_hidden_layers,
...     num_hidden_units=disc_hidden_size)
>>> disc_model.build(input_shape=(None, np.prod(image_size)))
>>> disc_model.summary()
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
dense_2 (Dense)	multiple	78500
leaky_re_lu_1 (LeakyReLU)	multiple	0
dropout (Dropout)	multiple	0
dense_3 (Dense)	multiple	101
Total params: 78,601		
Trainable params: 78,601		
Non-trainable params: 0		

## Определение обучающего набора данных

Далее мы загрузим набор данных MNIST и применим необходимые шаги предварительной обработки. Так как выходной слой генератора использует функцию активации  $\tanh$ , значения пикселей синтезированных изображений будут попадать в диапазон  $(-1, 1)$ . Тем не менее, входные пиксели изображений MNIST находятся внутри диапазона  $[0, 255]$  (с типом данных `tf.uint8` из TensorFlow). Таким образом, в рамках шагов предварительной подготовки мы будем использовать функцию `tf.image.convert_image_dtype` для преобразования типа `dtype` тензоров входных изображений из `tf.uint8` в `tf.float32`. В результате помимо изменения типа `dtype` вызов этой функции также изменит диапазон интенсивностей входных точек на  $[0, 1]$ . Затем мы можем их масштабировать, умножив на 2, и сдвинуть на  $-1$ , так что интенсивности пикселей попадут в диапазон  $[-1, 1]$ . Кроме того, мы также создадим случайный вектор  $z$ , основанный на желательном случайном распределении (равномерном ('uniform') или нормальном ('normal'), как наиболее распространенные варианты), после чего возвратим в кортеже предварительно обработанное изображение и случайный вектор:

```
>>> mnist_bldr = tfds.builder('mnist')
>>> mnist_bldr.download_and_prepare()
>>> mnist = mnist_bldr.as_dataset(shuffle_files=False)

>>> def preprocess(ex, mode='uniform'):
...     image = ex['image']
...     image = tf.image.convert_image_dtype(image, tf.float32)
...     image = tf.reshape(image, [-1])
...     image = image*2 - 1.0
...     if mode == 'uniform':
...         input_z = tf.random.uniform(
...             shape=(z_size,), minval=-1.0, maxval=1.0)
...     elif mode == 'normal':
...         input_z = tf.random.normal(shape=(z_size,))
...     return input_z, image

>>> mnist_trainset = mnist['train']
>>> mnist_trainset = mnist_trainset.map(preprocess)
```

Обратите внимание, что здесь мы возвращаем входной вектор  $z$  и изображение для удобного извлечения обучающих данных во время подгонки модели. Однако это не подразумевает, что вектор  $z$  как-то связан с изображением — входное изображение поступает из набора данных, тогда как вектор  $z$  генерируется случайным образом. На каждой итерации обучения случайно сгенерированный вектор  $z$  представляет вход, который генератор получает для синтеза нового изображения, а изображения (настоящие и синтезированные) являются входами дискриминатора.

Давайте проинспектируем созданный объект набора данных. В показанном ниже коде мы будем брать один пакет образцов и выводить формы массивов этой выборки входных векторов и изображений. Вдобавок для осмысления общего потока данных модели GAN мы будем обрабатывать прямой проход для генератора и дискриминатора.

Первым делом мы подадим пакет входных векторов  $z$  генератору и получим его выход `g_output`. Он будет пакетом фальшивых образцов, который подается дискриминатору с целью получения логитов для данного пакета, `d_logits_fake`. Кроме того, обработанные изображения, которые мы извлекли из объекта набора данных, будут подаваться дискриминатору, что приведет к получению логитов для настоящих образцов, `d_logits_real`. Вот необходимый код:

```
>>> mnist_trainset = mnist_trainset.batch(32, drop_remainder=True)
>>> input_z, input_real = next(iter(mnist_trainset))
>>> print('input-z -- форма: ', input_z.shape)
>>> print('input-real -- форма:', input_real.shape)
input-z -- форма: (32, 20)
input-real -- форма: (32, 784)

>>> g_output = gen_model(input_z)
>>> print('Выход G -- форма:', g_output.shape)
Выход G -- форма: (32, 784)

>>> d_logits_real = disc_model(input_real)
>>> d_logits_fake = disc_model(g_output)
>>> print('Дискриминатор (настоящие) -- форма:', d_logits_real.shape)
>>> print('Дискриминатор (фальшивые) -- форма:', d_logits_fake.shape)
Дискриминатор (настоящие) -- форма: (32, 1)
Дискриминатор (фальшивые) -- форма: (32, 1)
```

Два логита, `d_logits_fake` и `d_logits_real`, будут применяться для вычисления функций потерь при обучении модели.

## Обучение модели GAN

В качестве следующего шага мы создадим экземпляр класса `Binary Crossentropy`, который будет служить функцией потерь, и используем ее для расчета потерь генератора и дискриминатора, ассоциированных с только что обработанными пакетами. Для такой задачи нам также понадобятся достоверные метки для каждого выхода. В случае генератора мы создадим вектор единиц с такой же формой, как у вектора, содержащего спрогнозированные логиты для сгенерированных изображений, `d_logits_fake`. В потере дискриминатора мы имеем два члена: потеря при обнаружении фальшивых образцов, касающаяся `d_logits_fake`, и потеря при обнаружении настоящих образцов, основанная на `d_logits_real`.

Достоверные метки для члена, связанного с фальшивыми образцами, будут вектором нулей, который мы можем создать посредством функции `tf.zeros()` (или `tf.zeros_like()`). Аналогично мы можем сгенерировать достоверные метки для настоящих изображений с помощью функции `tf.ones()` (или `tf.ones_like()`), которая создает вектор единиц:

```
>>> loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
>>> ## Потеря для генератора
```

```

>>> g_labels_real = tf.ones_like(d_logits_fake)
>>> g_loss = loss_fn(y_true=g_labels_real, y_pred=d_logits_fake)
>>> print('Потеря генератора: {:.4f}'.format(g_loss))
Потеря генератора: 0.7505

>>> ## Потеря для дискриминатора
>>> d_labels_real = tf.ones_like(d_logits_real)
>>> d_labels_fake = tf.zeros_like(d_logits_fake)

>>> d_loss_real = loss_fn(y_true=d_labels_real,
...                       y_pred=d_logits_real)
>>> d_loss_fake = loss_fn(y_true=d_labels_fake,
...                       y_pred=d_logits_fake)
>>> print('Потеря дискриминатора: Настоящие {:.4f} Фальшивые {:.4f}'
...       .format(d_loss_real.numpy(), d_loss_fake.numpy()))
Потеря дискриминатора: Настоящие 1.3683 Фальшивые 0.6434

```

В предыдущем примере кода показан пошаговый расчет различных членов потери в целях понимания общей концепции, лежащей в основе обучения модели GAN. В приведенном ниже коде будет настроена модель GAN и реализован цикл обучения, где мы включим эти вычисления в цикл `for`.

Вдобавок мы будем применять класс `tf.GradientTape()` для расчета градиентов потерь относительно весов модели, а также оптимизировать параметры генератора и дискриминатора, используя два отдельных оптимизатора Adam. В коде вы увидите, что для переключения между обучением генератора и дискриминатора в TensorFlow мы явно предоставляем параметры каждой сети и применяем градиенты каждой сети отдельно к соответствующему оптимизатору:

```

>>> import time
>>> num_epochs = 100
>>> batch_size = 64
>>> image_size = (28, 28)
>>> z_size = 20
>>> mode_z = 'uniform'
>>> gen_hidden_layers = 1
>>> gen_hidden_size = 100
>>> disc_hidden_layers = 1
>>> disc_hidden_size = 100

>>> tf.random.set_seed(1)
>>> np.random.seed(1)

>>> if mode_z == 'uniform':

```



```

...     fixed_z = tf.random.uniform(
...         shape=(batch_size, z_size),
...         minval=-1, maxval=1)
>>> elif mode_z == 'normal':
...     fixed_z = tf.random.normal(
...         shape=(batch_size, z_size))

>>> def create_samples(g_model, input_z):
...     g_output = g_model(input_z, training=False)
...     images = tf.reshape(g_output, (batch_size, *image_size))
...     return (images+1)/2.0

>>> ## Настройка набора данных
>>> mnist_trainset = mnist['train']
>>> mnist_trainset = mnist_trainset.map(
...     lambda ex: preprocess(ex, mode=mode_z))

>>> mnist_trainset = mnist_trainset.shuffle(10000)
>>> mnist_trainset = mnist_trainset.batch(
...     batch_size, drop_remainder=True)

>>> ## Настройка модели
>>> with tf.device(device_name):
...     gen_model = make_generator_network(
...         num_hidden_layers=gen_hidden_layers,
...         num_hidden_units=gen_hidden_size,
...         num_output_units=np.prod(image_size))
...     gen_model.build(input_shape=(None, z_size))
...
...     disc_model = make_discriminator_network(
...         num_hidden_layers=disc_hidden_layers,
...         num_hidden_units=disc_hidden_size)
...     disc_model.build(input_shape=(None, np.prod(image_size)))

>>> ## Функция потерь и оптимизаторы:
>>> loss_fn = tf.keras.losses.BinaryCrossentropy(from_logits=True)
>>> g_optimizer = tf.keras.optimizers.Adam()
>>> d_optimizer = tf.keras.optimizers.Adam()

>>> all_losses = []
>>> all_d_vals = []
>>> epoch_samples = []

>>> start_time = time.time()
>>> for epoch in range(1, num_epochs+1):
...     epoch_losses, epoch_d_vals = [], []
... 
```

```
... for i, (input_z, input_real) in enumerate(mnist_trainset):
...
...     ## Расчет потери генератора
...     with tf.GradientTape() as g_tape:
...         g_output = gen_model(input_z)
...         d_logits_fake = disc_model(g_output,
...                                     training=True)
...         labels_real = tf.ones_like(d_logits_fake)
...         g_loss = loss_fn(y_true=labels_real,
...                           y_pred=d_logits_fake)
...
...     ## Расчет градиентов g_loss
...     g_grads = g_tape.gradient(g_loss,
...                                gen_model.trainable_variables)
...
...     ## Оптимизация: применение градиентов
...     g_optimizer.apply_gradients(
...         grads_and_vars=zip(g_grads,
...                             gen_model.trainable_variables))
...
...     ## Расчет потери дискриминатора
...     with tf.GradientTape() as d_tape:
...         d_logits_real = disc_model(input_real,
...                                     training=True)
...
...         d_labels_real = tf.ones_like(d_logits_real)
...
...         d_loss_real = loss_fn(
...             y_true=d_labels_real, y_pred=d_logits_real)
...
...         d_logits_fake = disc_model(g_output,
...                                     training=True)
...         d_labels_fake = tf.zeros_like(d_logits_fake)
...
...         d_loss_fake = loss_fn(
...             y_true=d_labels_fake, y_pred=d_logits_fake)
...
...         d_loss = d_loss_real + d_loss_fake
...
...     ## Расчет градиентов d_loss
...     d_grads = d_tape.gradient(d_loss,
...                                disc_model.trainable_variables)
...
... 
```

```

...     ## Оптимизация: применение градиентов
...     d_optimizer.apply_gradients(
...         grads_and_vars=zip(d_grads,
...                             disc_model.trainable_variables))
...
...     epoch_losses.append(
...         (g_loss.numpy(), d_loss.numpy(),
...          d_loss_real.numpy(), d_loss_fake.numpy()))
...
...     d_probs_real = tf.reduce_mean(
...         tf.sigmoid(d_logits_real))
...     d_probs_fake = tf.reduce_mean(
...         tf.sigmoid(d_logits_fake))
...     epoch_d_vals.append((d_probs_real.numpy(),
...                           d_probs_fake.numpy()))
...
...     all_losses.append(epoch_losses)
...     all_d_vals.append(epoch_d_vals)
...     print(
...         'Эпоха {:03d} | ET {:.2f} минут | Средние потери >>'
...         ' G/D {:.4f}/{:.4f} [D-Real: {:.4f} D-Fake: {:.4f}]'
...         .format(
...             epoch, (time.time() - start_time)/60,
...             *list(np.mean(all_losses[-1], axis=0)))
...         epoch_samples.append(
...             create_samples(gen_model, fixed_z).numpy())

```

Эпоха 001 | ET 0.88 минут | Средние потери >> G/D 2.9594/0.2843  
[D-Real: 0.0306 D-Fake: 0.2537]

Эпоха 002 | ET 1.77 минут | Средние потери >> G/D 5.2096/0.3193  
[D-Real: 0.1002 D-Fake: 0.2191]

Эпоха ...

Эпоха 100 | ET 88.25 минут | Средние потери >> G/D 0.8909/1.3262  
[D-Real: 0.6655 D-Fake: 0.6607]

В случае использования ГП процесс обучения, реализованный в предыдущем блоке кода, должен завершиться в Google Colab менее чем за час. (При наличии персонального компьютера с современным ЦП и ГП процесс может пройти даже быстрее.) После того, как обучение модели окончено, часто полезно вычертить график потерь дискриминатора и генератора для анализа поведения обеих подсетей и оценки, сходятся ли они.

Кроме того, имеет смысл вычертить график средних вероятностей пакетов настоящих и фальшивых образцов, рассчитанных дискриминатором на каждой итерации. Мы ожидаем, что значения этих вероятностей будут около 0.5, т.е. дискриминатор не способен с уверенностью проводить различие между настоящими и фальшивыми изображениями:

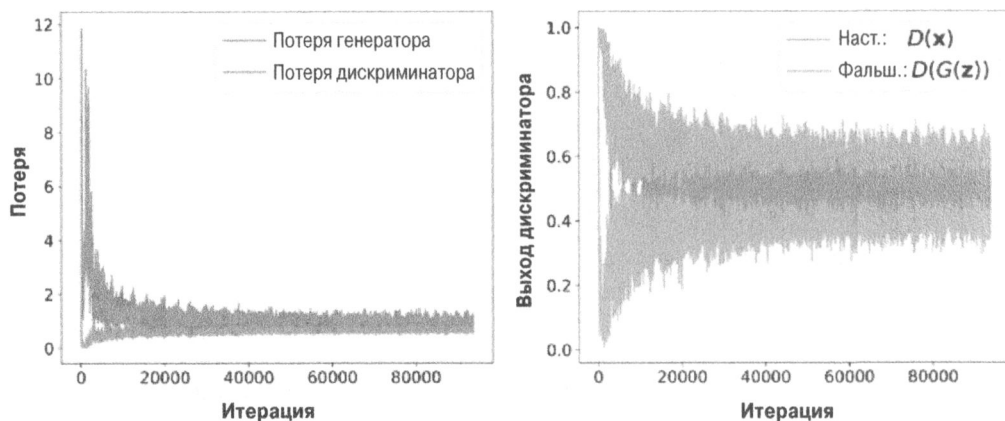
```
>>> import itertools
>>> fig = plt.figure(figsize=(16, 6))
>>> ## Вычерчивание графика потерь
>>> ax = fig.add_subplot(1, 2, 1)
>>> g_losses = [item[0] for item in itertools.chain(*all_losses)]
>>> d_losses = [item[1]/2.0 for item in itertools.chain(
...     *all_losses)]
>>> plt.plot(g_losses, label='Потеря генератора', alpha=0.95)
>>> plt.plot(d_losses, label='Потеря дискриминатора', alpha=0.95)
>>> plt.legend(fontsize=20)
>>> ax.set_xlabel('Итерация', size=15)
>>> ax.set_ylabel('Потеря', size=15)
>>> epochs = np.arange(1, 101)
>>> epoch2iter = lambda e: e*len(all_losses[-1])
>>> epoch_ticks = [1, 20, 40, 60, 80, 100]
>>> newpos = [epoch2iter(e) for e in epoch_ticks]
>>> ax2 = ax.twinx()
>>> ax2.set_xticks(newpos)
>>> ax2.set_xticklabels(epoch_ticks)
>>> ax2.xaxis.set_ticks_position('bottom')
>>> ax2.xaxis.set_label_position('bottom')
>>> ax2.spines['bottom'].set_position(('outward', 60))
>>> ax2.set_xlabel('Эпоха', size=15)
>>> ax2.set_xlim(ax.get_xlim())
>>> ax2.tick_params(axis='both', which='major', labelsize=15)
>>> ax2.tick_params(axis='both', which='major', labelsize=15)
>>> ## Вычерчивание графика выходов дискриминатора
>>> ax = fig.add_subplot(1, 2, 2)
>>> d_vals_real = [item[0] for item in itertools.chain(*all_d_vals)]
>>> d_vals_fake = [item[1] for item in itertools.chain(*all_d_vals)]
>>> plt.plot(d_vals_real, alpha=0.75,
...     label=r'Наст.:  $D(\mathbf{x})$ ')
>>> plt.plot(d_vals_fake, alpha=0.75,
...     label=r'Фальш.:  $D(G(\mathbf{z}))$ ')
```

```

>>> plt.legend(fontsize=20)
>>> ax.set_xlabel('Итерация', size=15)
>>> ax.set_ylabel('Выход дискриминатора', size=15)
>>> ax2 = ax.twinx()
>>> ax2.set_xticks(newpos)
>>> ax2.set_xticklabels(epoch_ticks)
>>> ax2.xaxis.set_ticks_position('bottom')
>>> ax2.xaxis.set_label_position('bottom')
>>> ax2.spines['bottom'].set_position(('outward', 60))
>>> ax2.set_xlabel('Эпоха', size=15)
>>> ax2.set_xlim(ax.get_xlim())
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax2.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()

```

Результаты показаны на рис. 17.8.



**Рис. 17.8.** Графики потерь генератора и дискриминатора и выходов дискриминатора

Обратите внимание, что дискриминатор выдает логиты, но для приведенной визуализации мы сохранили вероятности, вычисленные через сигмоидальную функцию, перед расчетом средних для каждого пакета.

На графике выходов дискриминатора (см. рис. 17.8) можно заметить, что на ранних стадиях обучения дискриминатор смог быстро научиться довольно точно отличать настоящие образцы от фальшивых образцов, т.е. фальшивые образцы имеют вероятности, близкие к 0, а настоящие образцы —

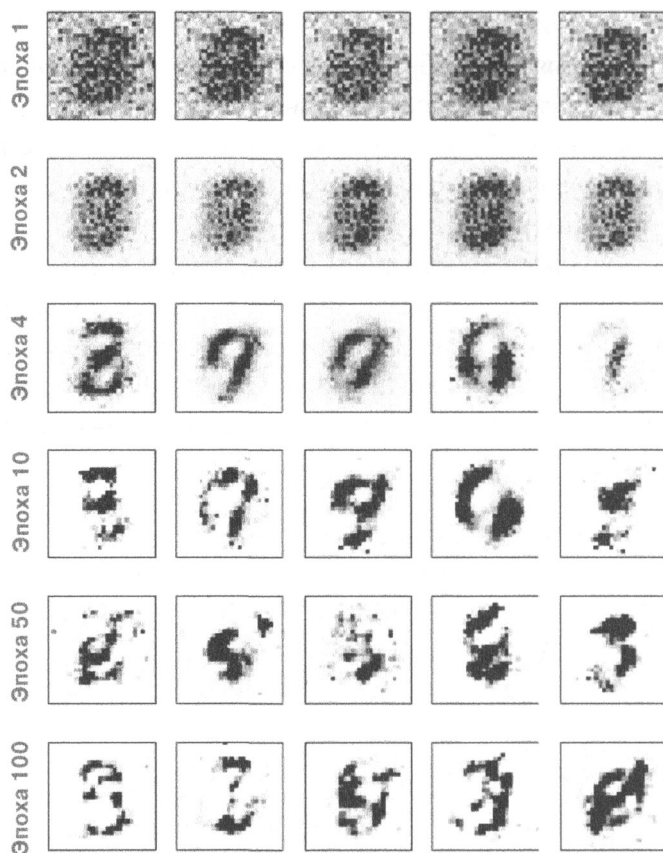
вероятности, близкие к 1. Причина в том, что фальшивые образцы не имели ничего общего с настоящими образцами; следовательно, проводить различие между ними было легко. По мере продвижения процесса обучения генератор улучшает свою способность синтезировать реалистичные изображения, что в результате приводит к получению вероятностей настоящих и фальшивых образцов, близких к 0.5.

Кроме того, мы также можем посмотреть, каким образом выходы генератора (синтезированные изображения) изменяются во время обучения. После каждой эпохи мы генерировали ряд образцов, вызывая функцию `create_samples()`, и сохраняли их в списке Python. В следующем коде мы визуализируем некоторые изображения, выпускаемые генератором, для избранных эпох:

```
>>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         if j == 0:
...             ax.text(
...                 -0.06, 0.5, 'Эпоха {}'.format(e),
...                 rotation=90, size=18, color='red',
...                 horizontalalignment='right',
...                 verticalalignment='center',
...                 transform=ax.transAxes)
...
...         image = epoch_samples[e-1][j]
...         ax.imshow(image, cmap='gray_r')
...
>>> plt.show()
```

Синтезированные генератором изображения представлены на рис. 17.9.

На рис. 17.9 несложно заметить, что по мере продвижения процесса обучения сеть генератора производила все более и более реалистичные изображения. Тем не менее, даже после 100 эпох выпущенные изображения по-прежнему сильно отличаются от изображений рукописных цифр, содержащихся в наборе данных MNIST.



**Рис. 17.9.** Подборка изображений, выпущенных генератором в разных эпохах

В этом разделе мы спроектировали очень простую модель GAN с одиночным полносвязным скрытым слоем в генераторе и дискриминаторе. После обучения модели GAN на наборе данных MNIST мы сумели добиться многообещающих, хотя пока не удовлетворительных результатов с новыми рукописными цифрами. Как объяснялось в главе 15, архитектуры нейронных сетей со сверточными слоями обладают рядом преимуществ по сравнению с полносвязными слоями, когда речь идет о классификации. Аналогичным образом добавление к нашей модели GAN сверточных слоев для работы с данными изображений может улучшить результат. В следующем разделе мы реализуем *глубокую сверточную порождающую состязательную сеть* (deep convolutional GAN — DCGAN), в которой сверточные слои применяются как для сети генератора, так и для сети дискриминатора.

## Повышение качества синтезированных изображений с использованием сверточной сети GAN и сети GAN Вассерштейна

В настоящем разделе мы реализуем сеть DCGAN, что позволит нам повысить эффективность, которой мы достигли в предыдущем примере сети. Вдобавок мы задействуем несколько дополнительных ключевых методик и реализуем *порождающую состязательную сеть Вассерштейна (Wasserstein GAN — WGAN)*.

В разделе будут раскрыты следующие методики:

- транспонированная свертка;
- пакетная нормализация;
- сеть WGAN;
- штраф градиента.

Сеть DCGAN была представлена в 2016 году Алеком Рэдфордом, Люком Метцом и Сумитом Чинтала в их статье “Unsupervised representation learning with deep convolutional generative adversarial networks” (Обучение представлению без учителя с помощью глубоких сверточных порождающих состязательных сетей), которая свободно доступна по ссылке <https://arxiv.org/pdf/1511.06434.pdf>. В указанной статье исследователи предложили применять сверточные слои в обеих сетях, т.е. генератора и дискриминатора. Начиная со случайного вектора  $z$ , сеть DCGAN первым делом использует полносвязный слой для проецирования  $z$  в новый вектор надлежащего размера, чтобы его форму можно было изменить на пространственное представление свертки ( $h \times w \times c$ ), которое меньше размера выходного изображения. Затем набор сверточных слоев, известный как *транспонированная свертка*, применяется для повышения дискретизации карт признаков до желательного размера выходного изображения.

### Транспонированная свертка

В главе 15 вы узнали об операции свертки в одно- и двумерных пространствах. В частности, мы взглянули, каким образом варианты для дополнения и страйдов изменяют выходные карты признаков. Наряду с тем, что операция свертки обычно используется для понижения дискретизации



пространства признаков (например, путем установки страйда в 2 или за счет добавления объединяющего слоя после сверточного слоя), операция *транспонированной свертки*, как правило, применяется для *повышения дискретизации (upsampling)* пространства признаков.

Чтобы понять операцию транспонированной свертки, давайте проведем простой мысленный эксперимент. Предположим, что у нас есть входная карта признаков размера  $n \times n$ . Затем мы применяем к такому входу  $n \times n$  двумерную операцию свертки с определенными параметрами дополнения и страйда, получая в результате выходную карту признаков размера  $m \times m$ . Теперь возникает вопрос: как с помощью другой операции свертки получить из выходной карты признаков  $m \times m$  карту признаков первоначальной размерности  $n \times n$ , одновременно сохраняя шаблоны связности между входом и выходом? Обратите внимание, что восстанавливается только форма входной матрицы  $n \times n$ , но не фактические значения матрицы. Именно это делает транспонированная свертка, как показано на рис. 17.10.

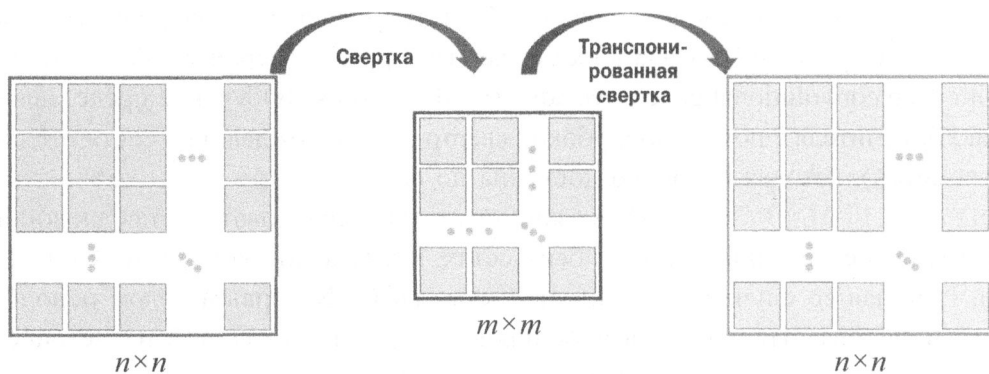


Рис. 17.10. Работа транспонированной свертки



Совет

### Транспонированная свертка или обращение свертки

Транспонированную свертку также называют *частично-страйдинговой сверткой (fractionally strided convolution)*. В литературе по ГО еще одним распространенным термином, который используется для ссылки на транспонированную свертку, является *обращение свертки (deconvolution)*. Однако следует отметить, что обращение свертки первоначально было определено как инверсия операции свертки  $f$  на карте признаков  $x$  с весовыми параметрами  $w$ , производящая кар-

ту признаков  $x'$ ,  $f_w(x) = x'$ . Тогда функция обращения свертки  $f^{-1}$  может быть определена как  $f_w^{-1}(f(x)) = x$ . Тем не менее, имейте в виду, что транспонированная свертка сконцентрирована на восстановлении только размерности пространства признаков, но не фактических значений.

Повышение дискретизации карт признаков с применением транспонированной свертки работает путем вставки нулей между элементами входных карт признаков. На рис. 17.11 демонстрируется пример применения транспонированной свертки к входу размера  $4 \times 4$  со страйдом  $2 \times 2$  и размером ядра  $2 \times 2$ . Матрица размера  $9 \times 9$  в центре показывает результаты после такой вставки нулей во входную карту признаков. Затем выполнение обычной свертки, использующей ядро  $2 \times 2$  со страйдом 1, дает в результате выход размера  $8 \times 8$ . Мы можем проверить обратное направление, выполнив на выходе нормальную свертку со страйдом 2, результатом которой будет выходная карта признаков размера  $4 \times 4$ , что совпадает с первоначальным размером входа.

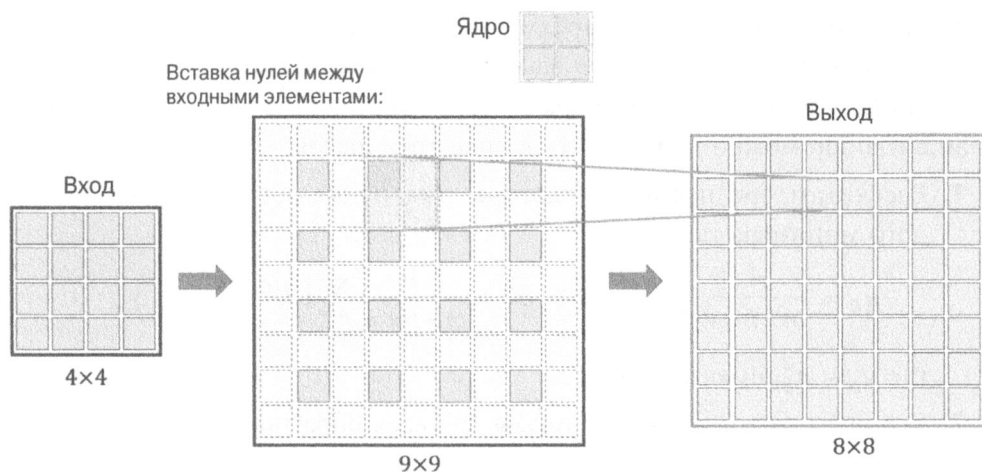


Рис. 17.11. Пример работы транспонированной свертки

На рис. 17.11 показано, как транспонированная свертка работает в целом. Существуют разнообразные сценарии, в которых вариации размера входа, размера ядра, страйдов и дополнения способны изменять выход. Если вы хотите узнать больше о таких сценариях, тогда почитайте руководство “A Guide to Convolution Arithmetic for Deep Learning” (Руководство по ариф-

метике сверток для глубокого обучения), написанное Венсаном Дюмуленом и Франческо Визином, которое свободно доступно по ссылке <https://arxiv.org/pdf/1603.07285.pdf>.

## Пакетная нормализация

*Пакетная нормализация* была представлена в 2015 году Сергеем Иоффе и Кристианом Сегеди в статье “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” (Пакетная нормализация: ускорение обучения глубоких сетей путем сокращения внутреннего ковариационного сдвига), свободно доступной по ссылке <https://arxiv.org/pdf/1502.03167.pdf>. Одна из главных идей, лежащих в основе пакетной нормализации, заключается в том, чтобы нормализовать входы слоев и предотвратить изменения в их распределении во время обучения, делая возможным более быстрое и лучшее сходжение.

Пакетная нормализация трансформирует мини-пакет признаков, базируясь на рассчитанных для него статистических данных. Предположим, что мы имеем карты признаков общей предварительной активации, полученные после сверточного слоя в четырехмерном тензоре  $\mathbf{Z}$  с формой  $[m \times h \times w \times c]$ , где  $m$  — количество образцов в пакете (т.е. размер пакета),  $h \times w$  — размерность пространства карт признаков и  $c$  — количество каналов. Пакетную нормализацию можно подытожить в виде трех шагов.

1. Рассчитать среднее и стандартное отклонение общих входов для каждого мини-пакета:

$$\mu_B = \frac{1}{m \times h \times w} \sum_{i,j,k} \mathbf{Z}^{[i,j,k,\cdot]}, \quad \sigma_B^2 = \frac{1}{m \times h \times w} \sum_{i,j,k} (\mathbf{Z}^{[i,j,k,\cdot]} - \mu_B)^2,$$

где  $\mu_B$  и  $\sigma_B^2$  имеют размер  $c$ .

2. Стандартизировать общие входы для всех образцов в пакете:

$$\mathbf{Z}_{\text{станд.}}^{[i]} = \frac{\mathbf{Z}^{[i]} - \mu_B}{\sigma_B + \varepsilon},$$

где  $\varepsilon$  — небольшое число для обеспечения численной устойчивости (т.е. избегания деления на ноль).

3. Масштабирование и сдвиг нормализованных общих входов с использованием двух векторов обучаемых параметров,  $\gamma$  и  $\beta$ , размера  $c$  (количество каналов):  $\mathbf{A}_{\text{предв.}}^{[i]} = \gamma \mathbf{Z}_{\text{станд.}}^{[i]} + \beta$ .

Процесс проиллюстрирован на рис. 17.12.

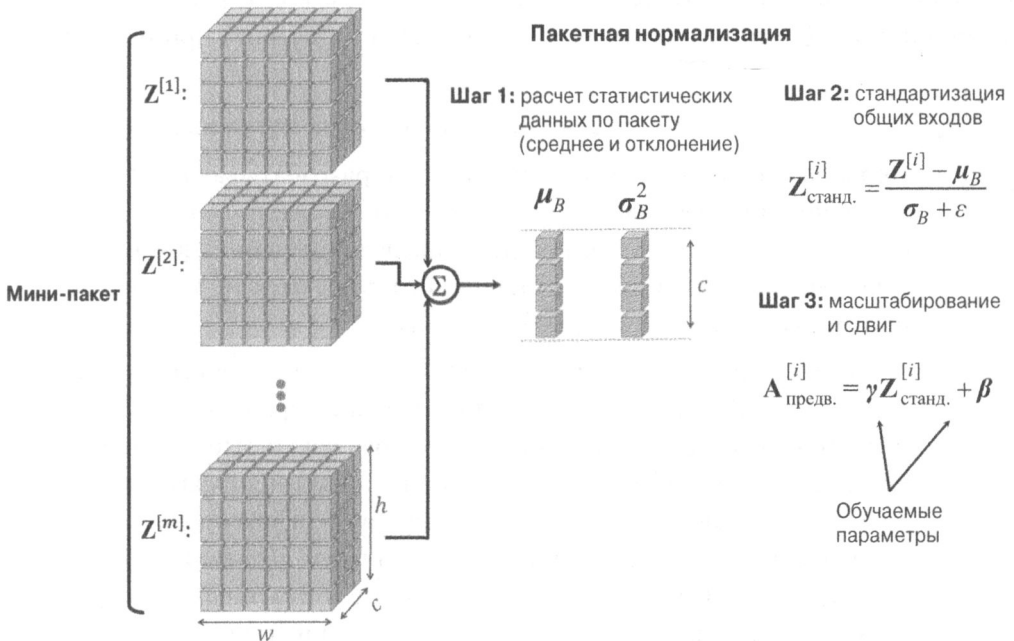


Рис. 17.12. Процесс пакетной нормализации

На первом шаге пакетной нормализации рассчитываются среднее  $\mu_B$  и стандартное отклонение  $\sigma_B$  мини-пакета. Оба они представляют собой векторы размера  $c$  (где  $c$  — количество каналов). Далее эти статистические данные будут задействованы на шаге 2 для масштабирования образцов внутри каждого мини-пакета через нормализацию по z-оценке (стандартизацию), давая в результате стандартизированные общие входы,  $Z_{\text{станд.}}^{[i]}$ . Как следствие, такие общие входы центрированы относительно среднего и имеют *единичную дисперсию*, которая обычно является полезной характеристикой для оптимизации на основе градиентного спуска. С другой стороны, неизменная нормализация общих входов, так чтобы они имели среди разных мини-пакетов одинаковые характеристики, которые могут быть несходными, способна серьезно повлиять на репрезентативную емкость нейронных сетей. Это можно понять, рассмотрев признак  $x \sim N(0, 1)$ , который после сигмоидальной активации  $\sigma(x)$  дает в результате линейную область для значений, близких к 0. Следовательно, на шаге 3 обучаемые параметры  $\beta$  и  $\gamma$ , представляющие собой векторы размера  $c$  (количество каналов), позволяют пакетной нормализации управлять смещением и разбросом нормализованных признаков.

Во время обучения вычисляются скользящее среднее  $\mu_B$  и скользящее отклонение  $\sigma_B^2$ , которые вместе с подстроеными параметрами  $\beta$  и  $\gamma$  используются для нормализации испытательного образца (образцов) при оценке.



### Почему пакетная нормализация содействует оптимизации?

Первоначально пакетная нормализация разрабатывалась для сокращения так называемого *внутреннего ковариационного сдвига* (*internal covariance shift*), который определяется как изменения, которые происходят в распределении активаций слоя из-за обновленных параметров сети во время обучения.

Чтобы объяснить все на простом примере, возьмем фиксированный пакет, который проходит через сеть в течение первой эпохи. Мы записываем активации каждого слоя для этого пакета. После итерации по всему обучающему набору данных и обновления параметров модели мы начинаем вторую эпоху, когда ранее зафиксированный пакет проходит через сеть. Затем мы сравниваем активации слоев из первой и второй эпох. Так как параметры сети изменились, мы замечаем, что активации тоже изменились. Описанное явление называется внутренним ковариационным сдвигом, который, как считалось, замедляет обучение нейронной сети.

Однако в 2018 году Шибани Сантуркар, Димитрис Ципрас, Эндрю Ильяс и Александр Мадрый продолжили выяснять, что же делает пакетную нормализацию настолько эффективной. В ходе работы исследователи заметили, что воздействие пакетной нормализации на внутренний ковариационный сдвиг является незначительным. Основываясь на результатах проведенных экспериментов, они выдвинули гипотезу о том, что эффективность пакетной нормализации взамен базируется на более гладкой поверхности функции потерь, которая увеличивает надежность невыпуклой оптимизации.

Если вам интересно узнать больше о результатах исследований Сантуркара, Ципраса, Ильяса и Мадрого, тогда почитайте исходную статью “How Does Batch Normalization Help Optimization?” (Как пакетная нормализация содействует оптимизации?), свободно доступную по ссылке <http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf>.

В API-интерфейсе Keras библиотеки TensorFlow предлагается класс `tf.keras.layers.BatchNormalization`, который мы можем применять в качестве слоя при определении моделей; он будет выполнять все описанные ранее шаги пакетной нормализации. Обратите внимание, что поведение обновления обучаемых параметров  $\gamma$  и  $\beta$  зависит от установки `training=False` или `training=True`. Это можно использовать для того, чтобы гарантировать, что указанные параметры будут выясняться только во время обучения.

## Реализация генератора и дискриминатора

К настоящему моменту мы раскрыли главные компоненты модели DCGAN, которую теперь реализуем. Архитектуры сетей генератора и дискриминатора показаны на рис. 17.13 и 17.14.

Генератор получает на входе вектор  $z$  размера 20, применяет к нему полносвязный (плотный) слой, чтобы увеличить его размер до 6 272, и придает ему форму тензора ранга 3 с формой  $7 \times 7 \times 128$  (размерность пространства  $7 \times 7$  и 128 каналов). Затем с помощью последовательности транспонированных сверток, использующих класс `tf.keras.layers.Conv2DTranspose`, повышается дискретизация карт признаков, пока размерность пространства результирующих карт признаков не достигнет  $28 \times 28$ . После каждого слоя транспонированной свертки количество каналов уменьшается вдвое за исключением последнего, который задействует только один выходной фильтр для генерации изображения в оттенках серого.

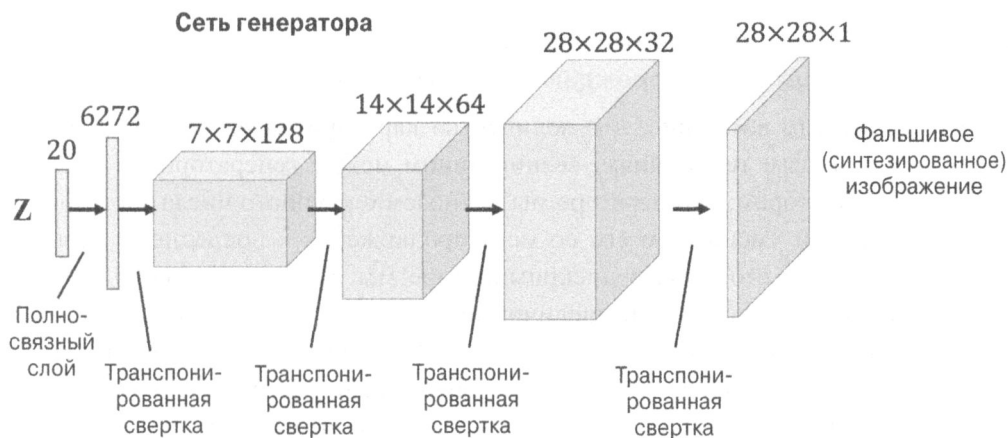


Рис. 17.13. Архитектура сети генератора

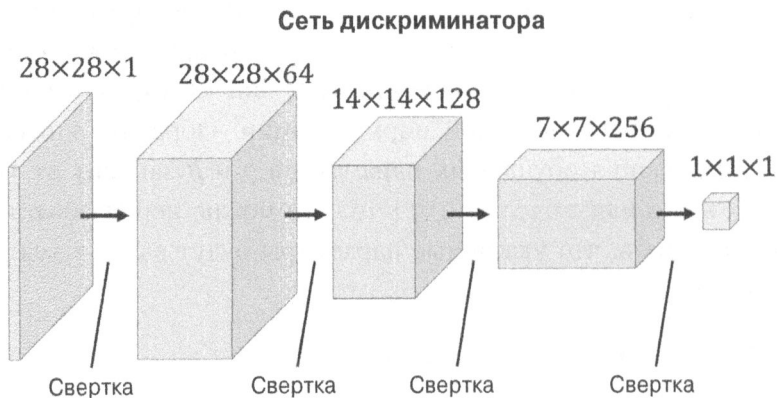


Рис. 17.14. Архитектура сети дискриминатора

За каждым слоем транспонированной свертки следуют пакетная нормализация и функция активации на основе ReLU с утечкой кроме последнего слоя, где применяется функция активации  $\tanh$  (без пакетной нормализации).

Дискриминатор получает изображения размером  $28 \times 28 \times 1$ , которые проходят через четыре сверточных слоя. Первые три сверточных слоя понижают размерность пространства в четыре раза, одновременно увеличивая количество каналов карт признаков. За каждым сверточным слоем находится пакетная нормализация, функция активации на основе ReLU с утечкой и слой отключения с  $\text{rate}=0.3$  (вероятность отключения). Последний сверточный слой использует ядро размером  $7 \times 7$  и единственный фильтр для понижения размерности пространства выхода до  $1 \times 1 \times 1$ .



Совет

### Особенности проектирования архитектуры для сверточных порождающих состязательных сетей

Обратите внимание, что количество карт признаков соответствует различным тенденциям, возникающим между генератором и дискриминатором. В генераторе мы начинаем с крупного числа карт признаков и уменьшаем его по мере продвижения к последнему слою. С другой стороны, в дискриминаторе мы начинаем с небольшого количества каналов и увеличиваем его, направляясь к последнему слою. При проектировании сетей CNN важный момент связан с количеством и пространственным размером карт признаков в обратном порядке. Когда пространственный размер карт признаков увеличивается, количество карт признаков уменьшается — и наоборот.

Вдобавок имейте в виду, что в слое, следующем за слоем пакетной нормализации, применять элементы смещения обычно не рекомендуется. В данном случае элементы смещения были бы избыточными, т.к. слой пакетной нормализации уже имеет параметр сдвига  $\beta$ . Вы можете опустить элементы смещения для отдельно взятого слоя, устанавливая `use_bias=False` в `tf.keras.layers.Dense` или `tf.keras.layers.Conv2D`.

Ниже приведен код двух вспомогательных функций для создания сетей генератора и дискриминатора:

```
>>> def make_dcgan_generator(
...     z_size=20,
...     output_size=(28, 28, 1),
...     n_filters=128,
...     n_blocks=2):
...     size_factor = 2**n_blocks
...     hidden_size = (
...         output_size[0]//size_factor,
...         output_size[1]//size_factor)
...
...     model = tf.keras.Sequential([
...         tf.keras.layers.Input(shape=(z_size,)),
...
...         tf.keras.layers.Dense(
...             units=n_filters*np.prod(hidden_size),
...             use_bias=False),
...         tf.keras.layers.BatchNormalization(),
...         tf.keras.layers.LeakyReLU(),
...         tf.keras.layers.Reshape(
...             (hidden_size[0], hidden_size[1], n_filters)),
...
...         tf.keras.layers.Conv2DTranspose(
...             filters=n_filters, kernel_size=(5, 5),
...             strides=(1, 1), padding='same', use_bias=False),
...         tf.keras.layers.BatchNormalization(),
...         tf.keras.layers.LeakyReLU()
...     ])
...
...     nf = n_filters
...     for i in range(n_blocks):
...         nf = nf // 2
```



```
...     model.add(
...         tf.keras.layers.Conv2DTranspose(
...             filters=nf, kernel_size=(5, 5),
...             strides=(2, 2), padding='same',
...             use_bias=False))
...     model.add(tf.keras.layers.BatchNormalization())
...     model.add(tf.keras.layers.LeakyReLU())
...
...     model.add(
...         tf.keras.layers.Conv2DTranspose(
...             filters=output_size[2], kernel_size=(5, 5),
...             strides=(1, 1), padding='same', use_bias=False,
...             activation='tanh'))
...
...     return model
>>> def make_dcgan_discriminator(
...     input_size=(28, 28, 1),
...     n_filters=64,
...     n_blocks=2):
...     model = tf.keras.Sequential([
...         tf.keras.layers.Input(shape=input_size),
...         tf.keras.layers.Conv2D(
...             filters=n_filters, kernel_size=5,
...             strides=(1, 1), padding='same'),
...         tf.keras.layers.BatchNormalization(),
...         tf.keras.layers.LeakyReLU()
...     ])
...     nf = n_filters
...     for i in range(n_blocks):
...         nf = nf*2
...         model.add(
...             tf.keras.layers.Conv2D(
...                 filters=nf, kernel_size=(5, 5),
...                 strides=(2, 2), padding='same'))
...         model.add(tf.keras.layers.BatchNormalization())
...         model.add(tf.keras.layers.LeakyReLU())
...         model.add(tf.keras.layers.Dropout(0.3))
...     model.add(
...         tf.keras.layers.Conv2D(
...             filters=1, kernel_size=(7, 7),
...             padding='valid'))
...     model.add(tf.keras.layers.Reshape((1,)))
...     return model
```

С помощью этих двух вспомогательных функций можно построить модель DCGAN и обучить ее с использованием того же самого объекта набора данных MNIST, который мы инициализировали в предыдущем сеансе, когда реализовывали простую полносвязную сеть GAN. Кроме того, мы можем применять те же самые функции потерь и процедуру обучения, что и ранее.

В оставшихся разделах главы мы внесем в модель DCGAN несколько дополнительных модификаций. Обратите внимание, что функция `preprocess()`, предназначенная для трансформации набора данных, должна быть изменена, чтобы выдавать тензор изображения, а не сплющивать изображение в вектор. В следующем коде показаны необходимые модификации для построения набора данных, а также продемонстрировано создание новых сетей генератора и дискриминатора:

```
>>> mnist_bldr = tfds.builder('mnist')
>>> mnist_bldr.download_and_prepare()
>>> mnist = mnist_bldr.as_dataset(shuffle_files=False)

>>> def preprocess(ex, mode='uniform'):
...     image = ex['image']
...     image = tf.image.convert_image_dtype(image, tf.float32)
...
...     image = image*2 - 1.0
...     if mode == 'uniform':
...         input_z = tf.random.uniform(
...             shape=(z_size,), minval=-1.0, maxval=1.0)
...     elif mode == 'normal':
...         input_z = tf.random.normal(shape=(z_size,))
...     return input_z, image
```

Мы можем создать сеть генератора, используя вспомогательную функцию `make_dcgan_generator()`, и вывести сведения об ее архитектуре, как показано ниже:

```
>>> gen_model = make_dcgan_generator()
>>> gen_model.summary()
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 6272)	125440
batch_normalization_7 (Batch Normalization)	(None, 6272)	25088

leaky_re_lu_7 (LeakyReLU)	(None, 6272)	0
reshape_2 (Reshape)	(None, 7, 7, 128)	0
conv2d_transpose_4 (Conv2DTr	(None, 7, 7, 128)	409600
batch_normalization_8 (Batch	(None, 7, 7, 128)	512
leaky_re_lu_8 (LeakyReLU)	(None, 7, 7, 128)	0
conv2d_transpose_5 (Conv2DTr	(None, 14, 14, 64)	204800
batch_normalization_9 (Batch	(None, 14, 14, 64)	256
leaky_re_lu_9 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_6 (Conv2DTr	(None, 28, 28, 32)	51200
batch_normalization_10 (Batc	(None, 28, 28, 32)	128
leaky_re_lu_10 (LeakyReLU)	(None, 28, 28, 32)	0
conv2d_transpose_7 (Conv2DTr	(None, 28, 28, 1)	800
=====		
Total params: 817,824		
Trainable params: 804,832		
Non-trainable params: 12,992		

Подобным образом мы можем создать сеть дискриминатора и отобразить информацию о ее архитектуре:

```
>>> disc_model = make_dcgan_discriminator()
>>> disc_model.summary()
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
=====		
conv2d_4 (Conv2D)	(None, 28, 28, 64)	1664
batch_normalization_11 (Batc	(None, 28, 28, 64)	256

leaky_re_lu_11 (LeakyReLU)	(None, 28, 28, 64)	0
conv2d_5 (Conv2D)	(None, 14, 14, 128)	204928
batch_normalization_12 (Batch Normalization)	(None, 14, 14, 128)	512
leaky_re_lu_12 (LeakyReLU)	(None, 14, 14, 128)	0
dropout_2 (Dropout)	(None, 14, 14, 128)	0
conv2d_6 (Conv2D)	(None, 7, 7, 256)	819456
batch_normalization_13 (Batch Normalization)	(None, 7, 7, 256)	1024
leaky_re_lu_13 (LeakyReLU)	(None, 7, 7, 256)	0
dropout_3 (Dropout)	(None, 7, 7, 256)	0
conv2d_7 (Conv2D)	(None, 1, 1, 1)	12545
reshape_3 (Reshape)	(None, 1)	0
=====		
Total params: 1,040,385		
Trainable params: 1,039,489		
Non-trainable params: 896		

Обратите внимание, что количество параметров для слоев пакетной нормализации на самом деле в четыре раза больше числа каналов. Вспомните, что параметры пакетной нормализации  $\mu_B$  и  $\sigma_B$  представляют среднее и стандартное отклонение (необучаемые параметры) для значения каждого признака, выведенные из заданного пакета;  $\gamma$  и  $\beta$  являются обучаемыми параметрами пакетной нормализации.

Отметим, что эта конкретная архитектура не будет очень хорошо работать в случае применения перекрестной энтропии в качестве функции потерь.

В следующем подразделе мы раскроем сеть WGAN, которая использует модифицированную функцию потерь на основе так называемого *расстояния Вассерштейна-1* (или *экскаватора*) между распределениями настоящих и фальшивых изображений для повышения эффективности обучения.

## Меры несходства между двумя распределениями

Сначала мы рассмотрим различные меры для расчета расхождения между двумя распределениями. Затем мы выясним, какая из мер уже встроена в исходную модель GAN. Наконец, переключение этой меры в сетях GAN приведет нас к реализации WGAN.

Как упоминалось в начале главы, цель порождающей модели заключается в том, что научить синтезировать новые образцы, которые имеют такое же распределение, как в обучающем наборе. Пусть  $P(x)$  и  $Q(x)$  представляют распределения случайной переменной  $x$ , как показано на рис. 17.15.

Меры	Формульное выражение
Полная вариация (TV)	$TV(P, Q) = \sup_x  P(x) - Q(x) $
Расстояние Кульбака–Лейблера (KL)	$KL(P  Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Дивергенция Йенсена–Шеннона (JS)	$JS(P, Q) = \frac{1}{2} \left( KL\left(P  \frac{P+Q}{2}\right) + KL\left(Q  \frac{P+Q}{2}\right) \right)$
Расстояние экскаватора (EM)	$EM(P, Q) = \inf_{\gamma \in \Pi(P, Q)} E_{(u, v) \in \gamma} (\ u - v\ )$

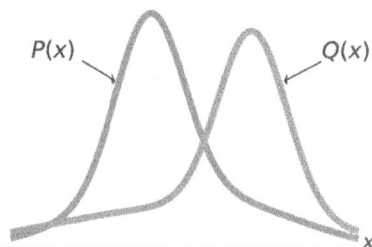


Рис. 17.15. Меры для распределений случайной переменной  $x$

Ниже мы обсудим способы, представленные на рис. 17.15, которые можно применять для измерения несходства между двумя распределениями,  $P$  и  $Q$ .

Функция *супремума* (*supremum*),  $\sup(S)$ , используемая в мере *полной вариации* (*total variation* — TV), означает наименьшую величину, которая больше всех элементов  $S$ . Другими словами,  $\sup(S)$  — точная верхняя грань для  $S$ . Наоборот, функция *инфимума* (*infimum*),  $\inf(S)$ , применяемая в мере *расстояния экскаватора* (*earth mover* — EM), означает наибольшую величину, которая меньше всех элементов  $S$  (точная нижняя грань).

Давайте разберем перечисленные меры, кратко изложив простыми словами то, чего они пытаются достичь.

- Расстояние TV измеряет наибольшее различие между двумя распределениями в каждой точке.
- Расстояние экскаватора, ЕМ, может интерпретироваться как минимальный объем работы, необходимой для трансформирования одного распределения в другое. Функция инфимума в расстоянии ЕМ берется по  $\Pi(P, Q)$  — коллекции всех совместных распределений, маргинальным распределением которых является  $P$  или  $Q$ . Тогда  $\gamma(u, v)$  представляет собой план перемещения, который указывает, каким образом мы перераспределяем землю с места  $u$  в  $v$ , при условии ряда ограничений для поддержания допустимых распределений после таких перемещений. Расчет расстояния ЕМ сам по себе является задачей оптимизации, направленной на нахождение оптимального плана перемещения  $\gamma(u, v)$ .
- Меры расстояния Кульбака–Лейблера (Kullback-Leibler — KL) и дивергенции Йенсена–Шеннона (Jensen-Shannon — JS) происходят из области теории информации. Обратите внимание, что расстояние KL несимметрично, т.е.  $KL(P \| Q) \neq KL(Q \| P)$  в отличие от дивергенции JS.

Уравнения для расчета несходства, приведенные на рис. 17.15, соответствуют непрерывным распределениям, но могут быть расширены на дискретные сценарии. Пример вычисления разных мер несходства с двумя простыми дискретными распределениями иллюстрируется на рис. 17.16.

Обратите внимание, что в случае расстояния ЕМ в таком простом примере мы видим, что  $Q(x)$  в точке  $x = 2$  имеет избыточное значение  $0.5 - 1/3 = 0.166$ , в то время как значение  $Q$  в двух других точках  $x$  меньше  $1/3$ . Следовательно, минимальный объем работы получается, когда мы перемещаем избыточное значение из  $x = 2$  в  $x = 1$  и  $x = 3$ , как было показано на рис. 17.16. В этом простом примере легко заметить, что такие перемещения дадут в результате минимальный объем работы из всех возможных перемещений. Однако в более сложных сценариях поступать так может оказаться нереально.

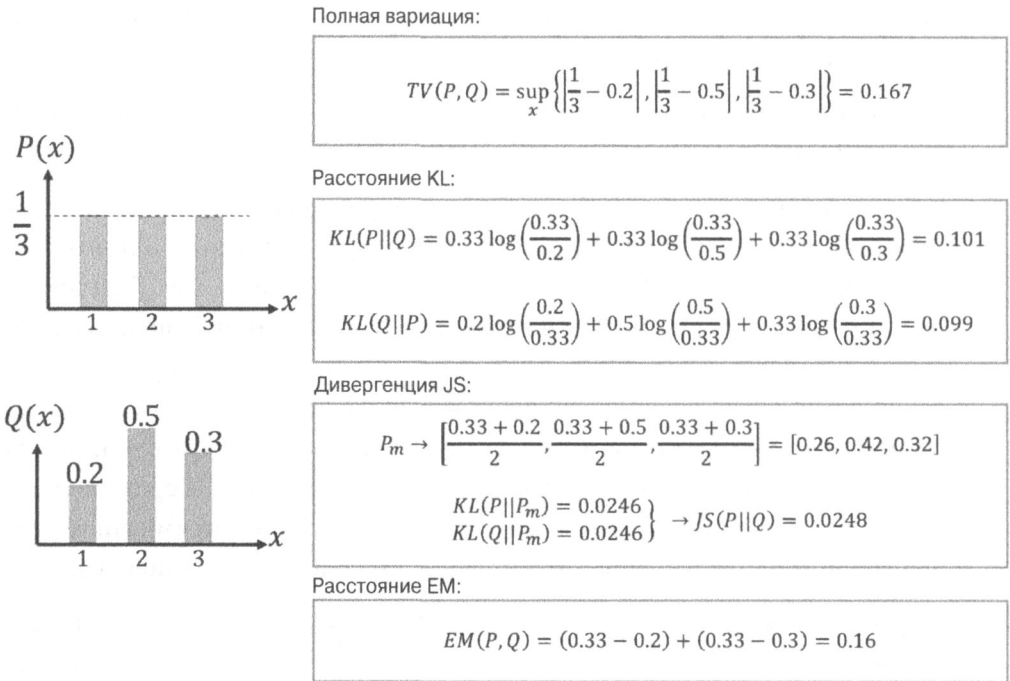


Рис. 17.16. Пример вычисления мер несходства с двумя простыми дискретными распределениями



### Связь между расстоянием KL и перекрестной энтропией

Расстояние KL,  $KL(P||Q)$ , измеряет относительную энтропию распределения  $P$  по отношению к эталонному распределению  $Q$ . Формульное выражение для расстояния KL может быть расширено как

$$KL(P||Q) = -\int P(x) \log(Q(x)) dx - \left( -\int P(x) \log(P(x)) \right)$$

Кроме того, для дискретных распределений расстояние KL можно записать так:

$$KL(P||Q) = -\sum_i P(x_i) \frac{P(x_i)}{Q(x_i)}$$

что аналогичным образом может быть расширено как

$$KL(P||Q) = -\sum_i P(x_i) \log(Q(x_i)) - \left( -\sum_i P(x_i) \log(P(x_i)) \right)$$

Базируясь на расширенном формульном выражении (дискретном или непрерывном), расстояние KL рассматривается как перекрестная энтропия между  $P$  и  $Q$  (первый член в предыдущем уравнении), из которой вычитается (собственная) энтропия  $P$  (второй член), т.е.  $KL(P \parallel Q) = H(P, Q) - H(P)$ .

Теперь, возвращаясь к нашему обсуждению сетей GAN, давайте посмотрим, как различные меры расстояния связаны с функцией потерь для сетей GAN. Математически можно показать, что функция потерь в первоначальной сети GAN на самом деле *минимизирует дивергенцию JS между распределением настоящих и фальшивых образцов*. Но, как объясняется в статье Мартена Аржовски и др. “Wasserstein Generative Adversarial Networks” (Порождающие состязательные сети Вассерштейна; <http://proceedings.mlr.press/v70/arjovsky17a/arjovsky17a.pdf>), у дивергенции JS есть проблемы с обучением модели GAN. По указанной причине для улучшения процесса обучения исследователи предложили использовать в качестве меры несходства между распределениями настоящих и фальшивых образцов расстояние EM.



Совет

### В чем преимущество использования расстояния EM?

Чтобы ответить на этот вопрос, мы можем рассмотреть пример, который был приведен в упомянутой выше статье Мартена Аржовски и др. Давайте предположим, что у нас имеются два распределения,  $P$  и  $Q$ , являющиеся двумя параллельными линиями. Одна линия зафиксирована в точке  $x = 0$ , а другая может перемещаться вдоль оси  $x$ , но первоначально находится в точке  $x = \Theta$ , где  $\Theta > 0$ .

Можно показать, что меры несходства KL, TV и JS выглядят как  $KL(P \parallel Q) = +\infty$ ,  $TV(P, Q) = 1$  и  $JS(P, Q) = 1/2 \log 2$ . Ни одна из этих мер несходства не является функцией параметра  $\Theta$ , а потому их нельзя дифференцировать относительно  $\Theta$ , чтобы распределения  $P$  и  $Q$  стали похожими друг на друга. С другой стороны, расстояние EM выражается как  $EM(P, Q) = |\Theta|$ , градиент которого относительно  $\Theta$  существует и способен двигать  $Q$  в направлении  $P$ .

А теперь давайте сосредоточим наше внимание на том, как расстояние EM можно применять для обучения модели GAN. Пусть  $P_r$  — распределение настоящих образцов и  $P_g$  — распределение фальшивых (сгенерирован-



ных) образцов.  $P_r$  и  $P_g$  заменяют  $P$  и  $Q$  в уравнении расстояния ЕМ. Как упоминалось ранее, расчет расстояния ЕМ сам по себе является задачей оптимизации; таким образом, она становится трудно поддающейся решению в вычислительном плане, особенно если мы хотим повторять расчет на каждой итерации цикла обучения модели GAN. К счастью расчет расстояния ЕМ может быть упрощен с использованием теоремы о двойственности Канторовича–Рубинштейна (Kantorovich-Rubinstein duality):

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} E_{u \in P_r}[f(u)] - E_{v \in P_g}[f(v)]$$

Здесь супремум берется по всем 1-липшицевым непрерывным функциям, обозначаемым посредством  $\|f\|_L \leq 1$ .



Совет

### Липшицева непрерывность

Основываясь на 1-липшицевой непрерывности (1-Lipschitz continuity), функция  $f$  обязана удовлетворять следующему свойству:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

Кроме того, вещественная функция  $f: R \rightarrow R$ , которая удовлетворяет свойству

$$|f(x_1) - f(x_2)| \leq K |x_1 - x_2|$$

называется *K-липшицевой непрерывностью*.

## Практическое использование расстояния Вассерштейна для порождающих состязательных сетей

Вопрос теперь в том, как найти такую 1-липшицевую непрерывную функцию для расчета расстояния Вассерштейна между распределениями настоящих ( $P_r$ ) и фальшивых ( $P_g$ ) выходов в сети GAN? Хотя теоретические концепции, лежащие в основе подхода WGAN, поначалу могут выглядеть сложными, ответ на данный вопрос проще, чем может показаться. Вспомните, что мы считаем глубокие нейронные сети универсальными аппроксиматорами функций. Это значит, что мы можем просто обучить нейросетевую модель для аппроксимации функции расстояния Вассерштейна. В предыдущем разделе было показано, что простая сеть GAN использует дискриминатор в форме классификатора. Для сети WGAN дискриминатор

может быть изменен, чтобы вести себя как *критик*, который возвращает скалярную оценку вместо значения вероятности. Мы можем трактовать такую оценку как то, настолько реалистичны входные изображения (подобно тому, как искусствовед назначает оценки произведениям искусства в галерее).

Чтобы обучить модель GAN с применением расстояния Вассерштейна, потери для дискриминатора  $D$  и генератора  $G$  определяются так, как описано ниже. Критик (т.е. сеть дискриминатора) возвращает свои выходы для пакета настоящих образцов и пакета синтезированных образцов. Мы используем обозначения  $D(x)$  и  $D(G(z))$  соответственно. Затем можно определить следующие члены потерь:

- компонент потери дискриминатора для настоящих образцов:

$$L_{\text{наст.}}^D = -\frac{1}{N} \sum_i D(x_i)$$

- компонент потери дискриминатора для фальшивых образцов:

$$L_{\text{фальш.}}^D = \frac{1}{N} \sum_i D(G(z_i))$$

- потеря генератора —  $L^G = -\frac{1}{N} \sum_i D(G(z_i))$ .

Это все, что нужно для сети WGAN, но понадобится еще обеспечить предохранение 1-липшицева свойства функции критика во время обучения. Для такой цели в статье о сетях WGAN предлагается ограничивать веса небольшой областью, скажем,  $[-0.01, 0.01]$ .

## Штраф градиента

В статье Аржовски и др. предлагается отсечение весов для поддержания 1-липшицева свойства дискриминатора (или критика). Тем не менее, в другой статье под названием “Improved Training of Wasserstein GANs” (Усовершенствованное обучение порождающих состязательных сетей Вассерштейна), которая свободно доступна по ссылке <https://arxiv.org/pdf/1704.00028.pdf>, Ишаан Гулрадхани и др. показали, что отсечение весов может привести к взрывному росту и исчезновению градиентов. Кроме того, отсечение весов также способно стать причиной недоиспользования емкости модели, порождая то, что модель критика будет ограничена выявле-

нием только ряда простых, но не более сложных функций. Таким образом, вместо отсечения весов Ишаан Гулрадхани и др. предложили альтернативное решение — *штраф градиента* (*gradient penalty* — GP). Результатом будет *порождающая состязательная сеть Вассерштейна со штрафом градиента* (WGAN with gradient penalty — WGAN-GP).

Процедуру для GP, которая добавляется в каждой итерации, можно подытожить в виде представленной ниже последовательности шагов.

1. Для каждой пары настоящих и фальшивых образцов ( $\mathbf{x}^{[i]}$ ,  $\tilde{\mathbf{x}}^{[i]}$ ) в заданном пакете выбрать случайное число  $\alpha^{[i]}$ , отобранное из равномерного распределения, т.е.  $\alpha^{[i]} \in U(0, 1)$ .
2. Рассчитать интерполяцию между настоящими и фальшивыми образцами:  $\check{\mathbf{x}}^{[i]} = \alpha \mathbf{x}^{[i]} + (1 - \alpha) \tilde{\mathbf{x}}^{[i]}$ , получив в результате интерполированные образцы.
3. Рассчитать выход дискриминатора (критика) для всех интерполированных образцов,  $D(\check{\mathbf{x}}^{[i]})$ .
4. Рассчитать градиенты выхода критика относительно каждого интерполированного образца, т.е.  $\nabla_{\check{\mathbf{x}}^{[i]}} D(\check{\mathbf{x}}^{[i]})$ .
5. Рассчитать GP как  $L_{gp}^D = \frac{1}{N} \sum_i \left( \left\| \nabla_{\check{\mathbf{x}}^{[i]}} D(\check{\mathbf{x}}^{[i]}) \right\|_2 - 1 \right)^2$ .

Тогда общая потеря для дискриминатора будет следующей:

$$L_{\text{общая}}^D = L_{\text{наст.}}^D + L_{\text{фальш.}}^D + \lambda L_{gp}^D$$

Здесь  $\lambda$  — подстраиваемый гиперпараметр.

## Реализация порождающей состязательной сети Вассерштейна со штрафом градиента для обучения модели DCGAN

Мы уже определили вспомогательные функции, которые создают сети генератора и дискриминатора для модели DCGAN (`make_dcgan_generator()` и `make_dcgan_discriminator()`). Ниже показан код для построения модели DCGAN.

```
>>> num_epochs = 100
>>> batch_size = 128
>>> image_size = (28, 28)
>>> z_size = 20
```

```

>>> mode_x = 'uniform'
>>> lambda_gp = 10.0
>>> tf.random.set_seed(1)
>>> np.random.seed(1)

>>> ## Настройка набора данных
>>> mnist_trainset = mnist['train']
>>> mnist_trainset = mnist_trainset.map(preprocess)
>>> mnist_trainset = mnist_trainset.shuffle(10000)
>>> mnist_trainset = mnist_trainset.batch(
...     batch_size, drop_remainder=True)
>>> ## Настройка модели
>>> with tf.device(device_name):
...     gen_model = make_dcgan_generator()
...     gen_model.build(input_shape=(None, z_size))
...
...     disc_model = make_dcgan_discriminator()
...     disc_model.build(input_shape=(None, np.prod(image_size)))

```

Теперь мы можем обучить модель. Обратите внимание, что обычно для модели WGAN (без GP) рекомендуется оптимизатор RMSprop, тогда как для WGAN-GP применяется оптимизатор Adam. Вот необходимый код:

```

>>> import time
>>> ## Оптимизаторы:
>>> g_optimizer = tf.keras.optimizers.Adam(0.0002)
>>> d_optimizer = tf.keras.optimizers.Adam(0.0002)
>>> if mode_z == 'uniform':
...     fixed_z = tf.random.uniform(
...         shape=(batch_size, z_size), minval=-1, maxval=1)
... elif mode_z == 'normal':
...     fixed_z = tf.random.normal(shape=(batch_size, z_size))
...
>>> def create_samples(g_model, input_z):
...     g_output = g_model(input_z, training=False)
...     images = tf.reshape(g_output, (batch_size, *image_size))
...     return (images+1)/2.0
>>> all_losses = []
>>> epoch_samples = []
>>> start_time = time.time()

```

```

>>> for epoch in range(1, num_epochs+1):
...     epoch_losses = []
...
...     for i, (input_z, input_real) in enumerate(mnist_trainset):
...
...         with tf.GradientTape() as d_tape, tf.GradientTape() \
...             as g_tape:
...
...             g_output = gen_model(input_z, training=True)
...
...             d_critics_real = disc_model(input_real,
...                 training=True)
...             d_critics_fake = disc_model(g_output,
...                 training=True)
...
...             ## Расчет потери генератора:
...             g_loss = -tf.math.reduce_mean(d_critics_fake)
...
...             ## Расчет потерь дискриминатора:
...             d_loss_real = -tf.math.reduce_mean(d_critics_real)
...             d_loss_fake = tf.math.reduce_mean(d_critics_fake)
...             d_loss = d_loss_real + d_loss_fake
...
...             ## Штраф градиента:
...             with tf.GradientTape() as gp_tape:
...                 alpha = tf.random.uniform(
...                     shape=[d_critics_real.shape[0], 1, 1, 1],
...                     minval=0.0, maxval=1.0)
...                 interpolated = (alpha*input_real +
...                     (1-alpha)*g_output)
...                 gp_tape.watch(interpolated)
...                 d_critics_intp = disc_model(interpolated)
...
...             grads_intp = gp_tape.gradient(
...                 d_critics_intp, [interpolated,])[0]
...             grads_intp_l2 = tf.sqrt(
...                 tf.reduce_sum(tf.square(grads_intp),
...                     axis=[1, 2, 3]))
...             grad_penalty = tf.reduce_mean(tf.square(
...                 grads_intp_l2 - 1.0))
...
...             d_loss = d_loss + lambda_gp*grad_penalty
...

```

```

...     ## Оптимизация: расчет и применение градиентов
...     d_grads = d_tape.gradient(d_loss,
...                               disc_model.trainable_variables)
...     d_optimizer.apply_gradients(
...         grads_and_vars=zip(d_grads,
...                             disc_model.trainable_variables))
...
...     g_grads = g_tape.gradient(g_loss,
...                               gen_model.trainable_variables)
...     g_optimizer.apply_gradients(
...         grads_and_vars=zip(g_grads,
...                             gen_model.trainable_variables))
...
...     epoch_losses.append(
...         (g_loss.numpy(), d_loss.numpy(),
...          d_loss_real.numpy(), d_loss_fake.numpy()))
...
...     all_losses.append(epoch_losses)
...     print(
...         'Эпоха {:03d} | ET {:.2f} минут | Средние потери >>'
...         ' G/D {:.6.2f}/{:.6.2f} [D-Real: {:.6.2f}]'
...         ' D-Fake: {:.6.2f}]'
...         .format(
...             epoch, (time.time() - start_time)/60,
...             *list(np.mean(all_losses[-1], axis=0)))
...     epoch_samples.append(
...         create_samples(gen_model, fixed_z).numpy())

```

В заключение мы визуализируем сохраненные образцы в некоторых эпохах, чтобы посмотреть, как модель обучается, а качество синтезируемых образцов изменяется в процессе обучения:

```

>>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         if j == 0:
...             ax.text(-0.06, 0.5, 'Эпоха {}'.format(e),
...                     rotation=90, size=18, color='red',
...                     horizontalalignment='right',

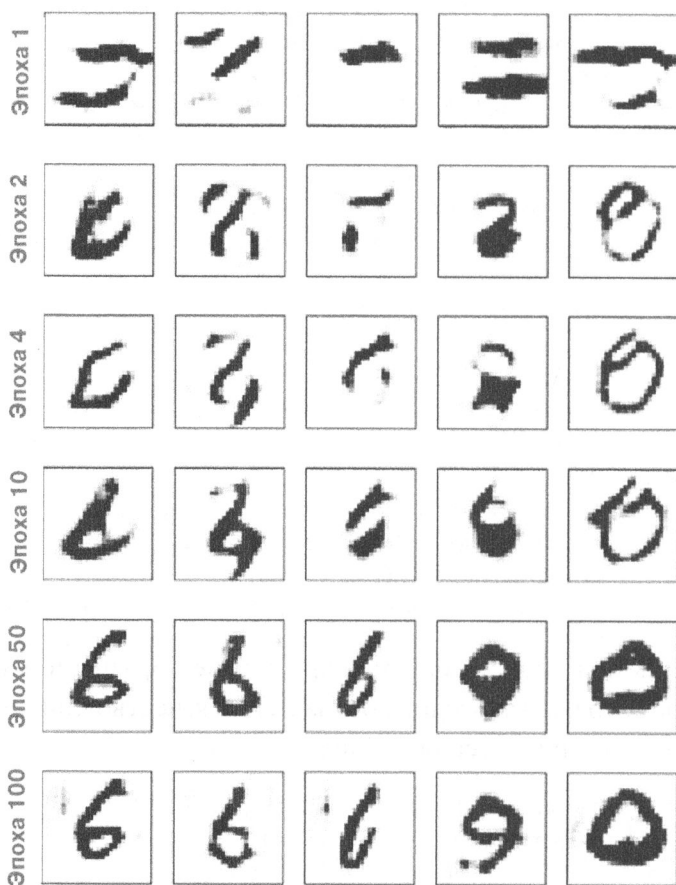
```

```

...             verticalalignment='center',
...             transform=ax.transAxes)
...
...     image = epoch_samples[e-1][j]
...     ax.imshow(image, cmap='gray_r')
>>> plt.show()

```

Результаты приведены на рис. 17.17.



**Рис. 17.17.** Сохраненные образцы в некоторых эпохах

Для визуализации результатов мы использовали тот же самый код, что и в разделе “Реализация сетей генератора и дискриминатора” ранее в главе. Сравнение новых образцов показывает, что модели DCGAN (с расстоянием Вассерштейна и штрафом градиента) способны генерировать изображения гораздо более высокого качества.

## Коллапс мод

Из-за состязательной природы моделей GAN их крайне нелегко обучать. Часто случается так, что во время обучения модель GAN может застрять в небольшом подпространстве и учиться генерировать похожие образцы. Такая ситуация называется *коллапсом мод (mode collapse)* и на рис. 17.18 приведен пример.

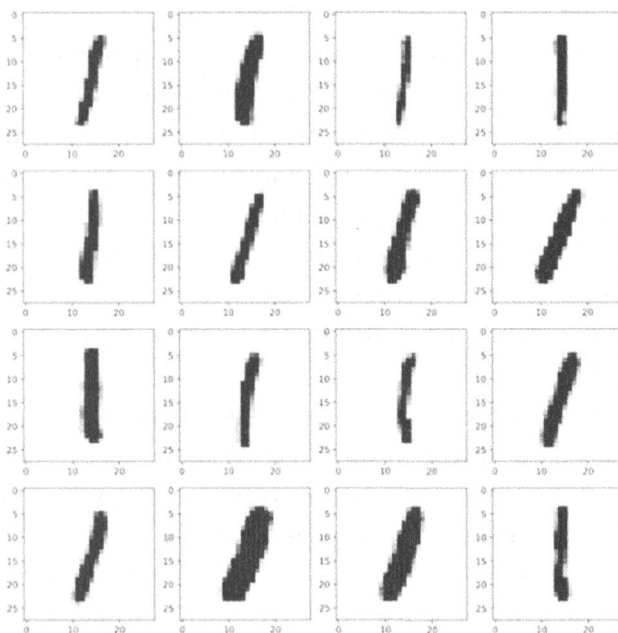


Рис. 17.18. Возникновение коллапса мод

Синтезированные образцы на рис. 17.18 не являются специально подобранными. Они показывают, что генератору не удалось изучить полное распределение данных, а взамен он избрал ленивый подход, сосредоточившись на некотором подпространстве.

Помимо упомянутых ранее проблем исчезновения и взрывного роста градиентов существует ряд дополнительных аспектов, которые также могут усложнить обучение моделей GAN (на самом деле это искусство). Ниже предложено несколько трюков от мастеров в области GAN.

Один из подходов называется *мини-пакетным различием (mini-batch discrimination)* и основан на том факте, что пакеты, которые состоят только из настоящих или фальшивых образцов, подаются дискриминатору по от-



дельности. При мини-пакетном различении мы позволяем дискриминатору сравнивать образцы среди таких пакетов, чтобы выяснить, какие образцы содержит пакет — настоящие или фальшивые. Разнообразие в пакете, состоящем только из настоящих образцов, почти наверняка выше, чем разнообразие в пакете, содержащем только фальшивые образцы, когда модель страдает от коллапса мод.

Еще одна методика, часто применяемая для стабилизации обучения моделей GAN, называется *согласованием признаков* (*feature matching*), когда мы вносим небольшую модификацию в целевую функцию генератора, добавляя дополнительный член, который минимизирует отличие между исходными и синтезированными изображениями, основываясь на промежуточных представлениях (картах признаков) дискриминатора. Мы рекомендуем прочитать статью “High Resolution Image Synthesis and Semantic Manipulation with Conditional GANs” (Синтез изображений с высоким разрешением и семантическое манипулирование с помощью условных порождающих состязательных сетей), написанную Тинь Чун Вонгом и др., которая свободно доступна по ссылке <https://arxiv.org/pdf/1711.11585.pdf>.

Во время обучения модель GAN также способна застревать в нескольких модах и перепрыгивать между ними. Во избежание такого поведения вы можете сохранять ряд старых образцов и подавать их дискриминатору, предотвращая возвращение генератора к предшествующим модам. Такая методика называется *воспроизведением опыта* (*experience replay*). Кроме того, вы можете обучить множество моделей GAN с разными начальными случайными числами, чтобы сочетание их всех покрывало большую часть распределения данных, чем любая из моделей.

## Другие приложения порождающих состязательных сетей

Основное внимание в главе мы уделяли генерированию образцов с использованием сетей GAN и вдобавок рассмотрели несколько трюков и методик повышения качества синтезируемого выхода. Спектр приложений сетей GAN быстро расширяется, включая компьютерное зрение, машинное обучение и даже другие предметные области науки и техники. Хороший список различных моделей GAN и областей их применения находится по ссылке <https://github.com/hindupuravinash/the-gan-zoo>.

Стоит отметить, что мы обсуждали сети GAN в манере без учителя, т.е. в моделях, раскрытых в главе, информация о метках классов не использовалась. Однако подход GAN может быть обобщен также на задачи частичного обучения и обучения с учителем. Например, условные порождающие состязательные сети, предложенные Мехди Мирза и Саймоном Осиндеро в статье “Conditional Generative Adversarial Nets” (Условные порождающие состязательные сети; <https://arxiv.org/pdf/1411.1784.pdf>), задействуют информацию о метках классов и учатся синтезировать новые изображения, обусловленные предоставленной меткой  $\tilde{x} = G(z|y)$  применительно к MNIST. В итоге у нас появляется возможность избирательного генерирования различных цифр в диапазоне 0–9.

Кроме того, условные сети GAN позволяют выполнять трансляцию изображений в изображения, которая предусматривает выяснение способа преобразования заданного изображения из одной предметной области в другую. Интересной работой в этом контексте является алгоритм Pix2Pix, описанный в статье “Image-to-Image Translation with Conditional Adversarial Networks” (Трансляция изображений в изображения с помощью условных состязательных сетей) Филлипа Изоля и др., которая свободно доступна по ссылке <https://arxiv.org/pdf/1611.07004.pdf>. Полезно упомянуть о том, что в алгоритме Pix2Pix вместо единственного прогноза для целого изображения дискриминатор предоставляет прогнозы “настоящий/фальшивый” для множества участков изображения.

Еще одной интересной моделью GAN, построенной поверх условной сети GAN, считается CycleGAN, которая тоже предназначена для трансляции изображений в изображения. Тем не менее, следует отметить, что обучающие образцы из двух предметных областей в CycleGAN не объединяются в пары, т.е. соответствие типа “один к одному” между входами и выходами отсутствует.

Например, с использованием модели CycleGAN мы могли бы поменять время года фотографии, сделанной летом, на зиму. В статье “Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks” (Непарная трансляция изображений в изображения с использованием циклически согласованных состязательных сетей) Цзюнь-Янь Чжу и др. (<https://arxiv.org/pdf/1703.10593.pdf>) демонстрируется впечатляющий пример преобразования лошадей в зебр.

## Резюме

В этой главе вы сначала узнали о порождающих моделях в ГО и их общей цели — синтезе новых данных. Затем мы показали, как модели GAN применяют сети генератора и дискриминатора, которые соперничают друг с другом при состязательном обучении, улучшая друг друга с течением времени. Далее мы реализовали простую модель GAN, используя для генератора и дискриминатора только полносвязные слои.

Мы также рассмотрели способы усовершенствования моделей GAN. Вы увидели архитектуру DCGAN, в которой для генератора и дискриминатора применяются глубокие сверточные сети. Попутно вы также ознакомились с двумя новыми концепциями: транспонированной сверткой (для повышения размерности пространства карт признаков) и пакетной нормализацией (для улучшения сходимости во время обучения).

Затем мы взглянули на сеть WGAN, которая использует расстояние ЕМ для измерения несходства между распределениями настоящих и фальшивых образцов. В заключение мы обсудили сеть WGAN со штрафом градиента для поддержания 1-липшицева свойства вместо отсечения весов.

В следующей главе мы рассмотрим обучение с подкреплением, которое представляет собой совершенно другую категорию МО в сравнении с тем, что было раскрыто в книге до сих пор.

# ОБУЧЕНИЕ С ПОДКРЕПЛЕНИЕМ ДЛЯ ПРИНЯТИЯ РЕШЕНИЙ В СЛОЖНЫХ СРЕДАХ

В предыдущих главах внимание было сосредоточено на машинном обучении с учителем и без учителя. Мы также выяснили, как использовать искусственные нейронные сети и глубокое обучение для решения таких типов задач МО. Вспомните, что обучение с учителем сфокусировано на прогнозировании метки категории или непрерывной величины из заданного вектора входных признаков. Обучение без учителя сконцентрировано на выделении шаблонов в данных, что делает его удобным при сжатии данных (глава 5), кластеризации (глава 11) или аппроксимации распределения обучающего набора для генерирования новых данных (глава 17).

В этой главе мы уделим внимание отдельной категории МО — *обучению с подкреплением* (*reinforcement learning* — *RL*), которая отличается от предшествующих категорий тем, что она фокусируется на изучении *последовательности действий* с целью оптимизации общей награды, скажем, выигрыша шахматной партии.

Мы раскроем в главе следующие темы:

- основы обучения с подкреплением, взаимодействия агента со средой и работа с процессом получения наград, которая помогает принимать решения в сложных средах;

- введение в различные категории задач обучения с подкреплением, задачи обучения на основе модели и без модели, а также алгоритмы обучения методом временных разностей и методом Монте-Карло;
- реализация алгоритма Q-обучения в табличной форме;
- понятие аппроксимации функций для решения задач обучения с подкреплением и сочетание обучения с подкреплением с ГО путем реализации алгоритма *глубокого Q-обучения*.

Обучение с подкреплением — сложная и обширная область исследований, и в главе мы сосредоточимся на его основах. Поскольку настоящая глава служит введением, а также для концентрации внимания на важных методах и алгоритмах мы будем работать главным образом над базовыми примерами, которые иллюстрируют основные концепции. Однако ближе к концу главы мы перейдем к более сложному примеру и задействуем архитектуры ГО для отдельного подхода обучения с подкреплением, который известен под названием *глубокое Q-обучение*.

## Введение — обучение на опыте

В этом разделе мы сначала представим концепцию обучения с подкреплением как ветвь МО и выясним его главные отличия от других задач МО. Затем мы раскроем фундаментальные компоненты системы обучения с подкреплением, после чего покажем математическую формулировку обучения с подкреплением, основанную на марковском процессе принятия решений.

### Понятие обучения с подкреплением

Вплоть до текущего места в книге внимание было сосредоточено главным образом на обучении *с учителем* и *без учителя*. Вспомните, что при обучении *с учителем* мы полагаемся на помеченные обучающие образцы, которые предоставляются диспетчером или экспертом-человеком, а цель заключается в том, чтобы наделить модель способностью хорошо обобщаться на не виденные ранее непомеченные испытательные образцы. Это значит, что модель обучения с учителем должна научиться назначать заданному входному образцу такие же метки или величины, как диспетчер или эксперт-человек. С другой стороны, при обучении *без учителя* целью является нахождение структуры, лежащей в основе набора данных, как в методах

кластеризации и понижения размерности, или выяснение, каким образом генерировать новые синтетические обучающие образцы с похожим внутренним распределением. Обучение с подкреплением существенно отличается от обучения с учителем и без учителя, а потому часто рассматривается как “третья категория машинного обучения”.

Ключевой элемент, который отличает обучение с подкреплением от остальных подзадач МО, таких как обучение с учителем и без учителя, состоит в том, что обучение с подкреплением базируется на концепции *обучения через взаимодействие*, т.е. модель обучается на основе взаимодействий со средой, чтобы довести до максимума *функцию наград*.

Хотя максимизация функции наград связана с минимизацией функции издержек в обучении с учителем, в обучении с подкреплением *корректные метки* для выяснения последовательности действий не известны или заранее не определены. Взамен их необходимо выяснить через взаимодействия со средой с целью достижения желаемого исхода вроде выигрыша в игре. При обучении с подкреплением модель (также называемая *агентом*) взаимодействует со своей средой и за счет этого генерирует последовательность взаимодействий, которые вместе называются *эпизодом*. Посредством таких взаимодействий агент накапливает ряд наград, определяемых средой. Награды могут быть положительными или отрицательными, а временами они не раскрываются агенту вплоть до конца эпизода.

Например, представим, что мы хотим научить компьютер играть в шахматы и выигрывать у игроков-людей. *Метки (награды)* для каждого отдельно взятого шахматного хода, сделанного компьютером, не известны вплоть до конца игры, потому что в течение самой игры мы не знаем, приведет ли конкретный ход в итоге к выигрышу или к проигрышу. Ответная реакция определяется лишь в конце игры. Вероятно, она будет положительной наградой, если компьютер выиграл партию, т.к. агент достиг общей желательной цели, и наоборот — отрицательной, если компьютер проиграл.

Кроме того, в примере с игрой в шахматы входом является текущая конфигурация, скажем, расположение индивидуальных фигур на доске. Учитывая большое количество возможных входов (состояний системы), пометить каждую конфигурацию или состояние как положительное или отрицательное попросту невозможно. Следовательно, для определения процесса обучения мы предоставляем *награды (или штрафы)* в конце каждой игры, когда знаем, достигнута ли желательная цель — выиграли мы игру или нет.

В этом и заключается сущность обучения с подкреплением. При обучении с подкреплением мы не можем или не учим агента, компьютер или робота тому, *как* что-то делать; мы можем только указать то, *что* именно хотим добиться от агента. Затем, основываясь на исходе конкретного испытания, мы можем определить награды в зависимости от успеха или неудачи агента. В результате обучение с подкреплением становится очень привлекательным для принятий решений в сложных средах — особенно когда решение задачи требует последовательности шагов, которые неизвестны, трудны в объяснении или непросты в определении.

Помимо приложений в играх и робототехнике примеры обучения с подкреплением можно обнаружить в живой природе. Например, дрессировка собаки включает в себя обучение с подкреплением — мы выдаем награды (лакомство) собаке, когда она выполняет желательные действия. Или возьмем медицинскую собаку, которая обучена предупреждать своего партнера о надвигающемся приступе. В таком случае мы не знаем точный механизм, посредством которого собака способна выявлять надвигающийся приступ, и мы, безусловно, не имеем возможности определить последовательность шагов, чтобы научить выявлению приступа, даже если бы располагали точным знанием этого механизма. Тем не менее, мы можем награждать собаку с помощью лакомства, если она успешно выявляет приступ, чтобы *подкрепить* такое поведение!

Хотя обучение с подкреплением предлагает мощную базу для изучения произвольных последовательностей действий, направленных на достижение определенной цели, имейте в виду, что обучение с подкреплением является все еще новой и активной областью исследований с многочисленными нерешенными проблемами. Один из аспектов, делающих обучение с подкреплением особенно сложным, связан с тем, что последовательные входы модели зависят от действий, предпринятых ранее. Это может приводить к всевозможным проблемам и обычно результатом оказывается нестабильное поведение при обучении. Кроме того, зависимость от последовательности в обучении с подкреплением создает так называемый *отсроченный эффект*, когда действие, предпринятое на временном шаге  $t$ , может привести к будущей награде, которая появится на какое-то произвольное количество шагов позже.

## Определение интерфейса “агент–среда” системы обучения с подкреплением

Во всех примерах обучения с подкреплением мы можем обнаружить две отдельных сущности: агента и среду. Формально *агент* определяется как сущность, которая учится принимать решения и взаимодействует с окружающей ее средой, предпринимая действия. Взамен в виде последовательности принятия действий агент получает наблюдения и сигнал награды, управляемый средой. *Среда* — это все, что находится снаружи агента. Среда обменивается информацией с агентом и определяет сигнал награды для действия агента, а также его наблюдения.

*Сигнал награды* представляет собой ответную реакцию, которую агент получает при взаимодействии со средой, обычно принимает форму скалярного значения и может быть либо положительным, либо отрицательным. Целью награды является сообщение агенту о том, насколько хорошо он выполнялся. Частота, с которой агент получает награды, зависит от решаемой задачи. Скажем, в шахматах награда будет определяться после окончания игры на основе исхода всех ходов: выигрыш или проигрыш. С другой стороны, мы могли бы создать лабиринт так, чтобы награда определялась после каждого временного шага. Тогда агент в таком лабиринте будет пытаться довести до максимума награды, накопленные за время его существования, где под временем существования понимается длительность эпизода. Диаграмма на рис. 18.1 иллюстрирует взаимодействия и обмен информацией между агентом и средой.



Рис. 18.1. Взаимодействия и обмен информацией между агентом и средой



Как показано на рис. 18.1, состояние агента представляет собой набор из всех его переменных ❶. Например, в случае робота-дрона такие переменные могли бы включать текущее местоположение дрона (долгота, широта и высота), оставшееся время работы батареи дрона, скорость каждого пропеллера и т.д. На каждом временном шаге агент взаимодействует со средой через набор доступных действий  $A_t$  ❷. Но основе предпринятого действия, обозначенного как  $A_t$ , пока агент находится в состоянии  $S_t$ , он получит сигнал награды  $R_{t+1}$  ❸, а его состоянием станет  $S_{t+1}$  ❹.

Во время процесса обучения агент обязан опробовать различные действия (*исследование*), чтобы он мог постепенно узнавать, каким действиям отдавать предпочтение и выполнять более часто (*эксплуатация*) для доведения до максимума общей накопленной награды.

Чтобы разобраться в этой концепции, давайте рассмотрим очень простой пример, где новый выпускник факультета компьютерных наук, специализирующийся на разработке ПО, задается вопросом, начать ли ему работать в какой-то компании (*эксплуатация*) либо продолжить обучение в магистратуре или докторантуре с целью обретения дополнительных знаний в области науки о данных и МО (*исследование*).

В целом эксплуатация будет приводить к выбору действий с большей краткосрочной наградой, тогда как исследование потенциально может обеспечить более высокие награды в долгосрочной перспективе. Компромисс между исследованием и эксплуатацией изучался довольно широко, но до сих пор нет универсального выхода из указанного затруднительного положения при принятии решений.

## Теоретические основы обучения с подкреплением

Прежде чем мы погрузимся в практические примеры и начнем обучение модели, давайте проясним несколько теоретических основ обучения с подкреплением. В последующих разделах мы начнем с исследования математической формулировки *марковских процессов принятия решений*, эпизодических и продолжающихся задач, ряда ключевых терминов обучения с подкреплением и динамического программирования с применением *уравнения Беллмана*. Итак, займемся марковскими процессами принятия решений.

## Марковские процессы принятия решений

В общем случае тип задач, с которыми имеет дело обучение с подкреплением, обычно формулируется как *марковские процессы принятия решений* (*Markov decision process* — MDP). Стандартный подход к решению задач MDP предусматривает использование динамического программирования, но обучение с подкреплением обладает рядом ключевых преимуществ по сравнению с динамическим программированием.



### Динамическое программирование

Под динамическим программированием понимается набор компьютерных алгоритмов и методик программирования, которые были разработаны Ричардом Беллманом в 1950-х годах. В некотором смысле динамическое программирование касается рекурсивного решения задач — решения относительно сложных задач путем их разбиения на меньшие подзадачи.

Основное различие между рекурсией и динамическим программированием заключается в том, что при динамическом программировании сохраняются результаты подзадач (обычно в виде словаря или другой формы таблицы поиска), поэтому к ним можно получать доступ за постоянное время (вместо их повторного расчета), если они снова встретятся в будущем.

В число примеров ряда знаменитых задач в области компьютерных наук, которые решаются с помощью динамического программирования, входят выравнивание последовательностей и вычисление кратчайшего пути из точки А в точку Б.

Однако динамическое программирование не является осуществимым подходом, когда размер состояний (т.е. количество возможных конфигураций) относительно велик. В таких случаях обучение с подкреплением считается гораздо более эффективной и практичной альтернативой для решения задач MDP.

## Математическая формулировка марковских процессов принятия решений

Типы задач, которые требуют изучения интерактивного и последовательного процесса принятия решений, где решение на временном шаге  $t$  влияет на последующие ситуации, математически формализованы как марковские процессы принятия решений (MDP).

В сценарии взаимодействий агент/среда при обучении с подкреплением, если мы обозначим начальное состояние агента как  $S_0$ , то взаимодействия между агентом и средой дадут в результате последовательность такого вида:

$$\{S_0, A_0, R_1\}, \{S_1, A_1, R_2\}, \{S_2, A_2, R_3\}, \dots$$

Обратите внимание, что фигурные скобки служат только в качестве визуальной подсказки. Здесь  $S_t$  и  $A_t$  означают состояние и действие, предпринятое на временном шаге  $t$ . Посредством  $R_{t+1}$  обозначается награда, полученная от среды после выполнения действия  $A_t$ . Следует отметить, что  $S_t$ ,  $R_{t+1}$  и  $A_t$  представляют собой независимые от времени переменные, которые берут значения из предварительно определенных конечных наборов, обозначаемых с помощью  $s \in \hat{S}$ ,  $r \in \hat{R}$  и  $a \in \hat{A}$  соответственно. В марковском процессе принятия решений зависимые от времени переменные  $S_t$  и  $R_{t+1}$  имеют распределения вероятностей, которые зависят только от их значений на предыдущем временном шаге,  $t-1$ . Распределение вероятностей для  $S_{t+1} = s'$  и  $R_{t+1} = r$  может быть записано как условная вероятность от предыдущего состояния ( $S_t$ ) и предпринятого действия ( $A_t$ ):

$$p(s', r | s, a) \triangleq P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a)$$

Такое распределение вероятностей полностью определяет *динамику среды* (или *модели среды*), поскольку на основе этого распределения могут быть рассчитаны вероятности всех переходов среды. Следовательно, динамика среды является центральным критерием для категоризации различных методов обучения с подкреплением. Типы методов обучения с подкреплением, которые требуют модели среды или пытаются изучить модель среды (т.е. выяснить динамику среды), называются методами *на основе модели* в противоположность методам *без модели*.

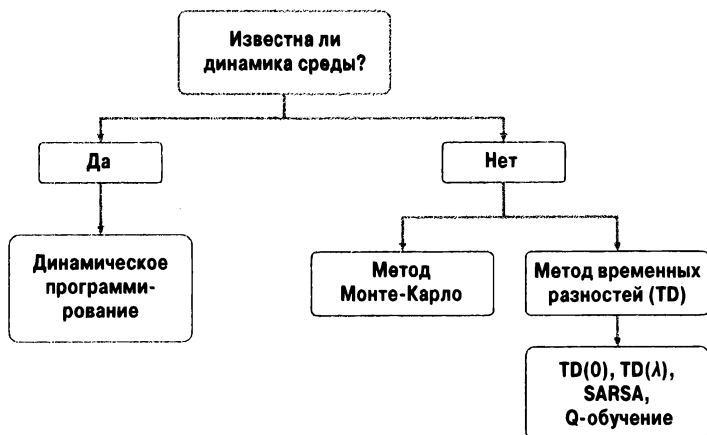


На заметку!

### Обучение с подкреплением без модели и на основе модели

Когда вероятность  $p(s', r | s, a)$  известна, тогда задачу обучения можно решить с помощью динамического программирования. Но при неизвестной динамике среды, как в случае многих реальных задач, вам нужно будет запрашивать большое количество образцов через взаимодействие со средой, чтобы компенсировать отсутствие сведений о динамике среды.

Справиться с проблемой помогут два подхода: метод *Монте-Карло* без модели и метод *временных разностей* (temporal difference — TD). На следующей диаграмме показаны две главных категории и ответвления каждого метода.



Далее в главе мы раскроем эти разные подходы и их ответвления, начиная с теории и заканчивая практическими алгоритмами.

Динамику среды можно считать детерминированной, если определенные действия для заданных состояний предпринимаются всегда или никогда, т.е.  $p(s', r | s, a) \in \{0, 1\}$ . Иначе в более общем случае среда будет обладать стохастическим поведением.

Чтобы понять такое стохастическое поведение, давайте рассмотрим вероятность наблюдения будущего состояния  $S_{t+1} = s'$  при условии, что текущим состоянием является  $S_t = s$  и предпринято действие  $A_t = a$ . Она обозначается как  $p(s' | s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s' | S_t = s, A_t = a)$ .

Ее можно вычислить как безусловную вероятность, выполнив сумму по всем возможным наградам:

$$p(s' | s, a) \stackrel{\text{def}}{=} \sum_{r \in \mathcal{R}} p(s', r | s, a)$$

Такая вероятность называется *вероятностью смены состояния*. Исходя из вероятности смены состояния, если динамика среды детерминирована, то это значит, что когда агент предпринимает действие  $A_t = a$  в состоянии  $S_t = s$ , переход в следующее состояние,  $S_{t+1} = s'$ , будет на 100% достоверным, т.е.  $p(s' | s, a) = 1$ .

## Визуализация марковского процесса

Марковский процесс может быть представлен как ориентированный циклический граф, узлы которого соответствуют разным состояниям среды. Ребра графа (связи между узлами) представляют вероятности переходов между состояниями.

Например, возьмем студента, который делает выбор среди трех состояний: (А) подготовка к сдаче экзамена дома, (В) игра в видеоигры дома или (С) подготовка к сдаче экзамена в библиотеке. Кроме того, имеется заключительное состояние (Т) — пойти спать. Решения принимаются каждый час, и в течение этого конкретного часа студент остается в выбранном состоянии. Далее предположим, что находясь дома (состояние А), студент с вероятностью 50% будет переключаться с учебы на игру в видеоигры. С другой стороны, когда студент пребывает в состоянии В (игра в видеоигры), есть относительно высокий шанс (80%) того, что он продолжит играть в видеоигру в течение последующих часов.

Динамика поведения студента показана в виде марковского процесса на рис. 18.2, включая циклический граф и таблицу переходов.

Значения на ребрах графа представляют вероятности переходов в поведении студента, и они также приведены в таблице справа. Рассматривая строки в таблице, имейте в виду, что вероятности переходов из каждого состояния (узла) всегда в сумме дают 1.

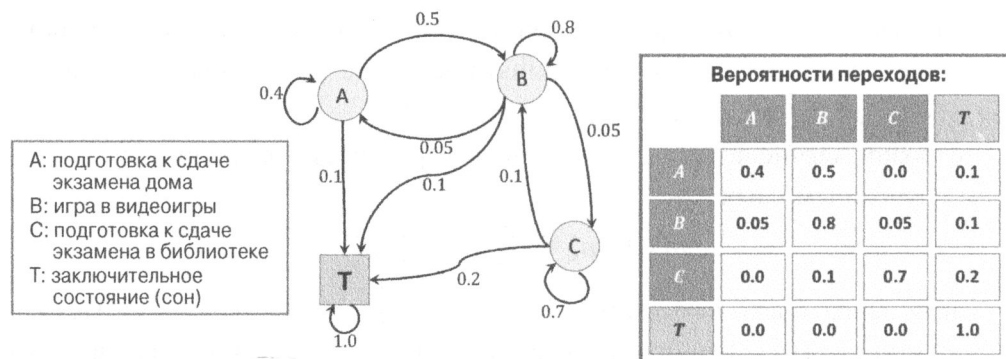


Рис. 18.2. Динамика поведения студента

## Эпизодические или продолжающиеся задачи

По мере взаимодействия агента со средой последовательность наблюдений или состояний формирует траекторию. Траектории бывают двух типов. Если траектория агента может быть разделена на части, так что каждая из них начинается в момент времени  $t = 0$  и заканчивается в заключительном состоянии  $S_T$  (при  $t = T$ ), тогда задача называется *эпизодической*. С другой стороны, если траектория длится бесконечно, не попадая в заключительное состояние, то задача называется *продолжающейся*.

Задача, связанная с агентом обучения для игры в шахматы, является эпизодической, в то время как робот-уборщик, который поддерживает чистоту в доме, обычно выполняет продолжающуюся задачу. В настоящей главе мы будем обсуждать только эпизодические задачи.

В эпизодических задачах *эпизод* представляет собой последовательность или траекторию, которую агент проходит от начального состояния  $S_0$  в заключительное состояние  $S_T$ :

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_T, A_T, R_{T+1}, \dots, S_{T-1}, A_{T-1}, R_T, S_T$$

В приведенном на рис. 18.2 марковском процессе, который изображает задачу студента, готовящегося к сдаче экзамена, мы можем встретить эпизоды вроде следующих трех примеров:

- **Эпизод 1:** BBCCCCBAT  $\rightarrow$  экзамен сдан (финальная награда = +1)
- **Эпизод 2:** ABVBVBVBVBVT  $\rightarrow$  экзамен провален (финальная награда = -1)
- **Эпизод 3:** BCCCCCT  $\rightarrow$  экзамен сдан (финальная награда = +1)

## Терминология обучения с подкреплением: отдача, политика и функция ценности

Давайте определим несколько дополнительных терминов, специфичных для обучения с подкреплением, которые мы будем применять в оставшейся части главы.

### Отдача

Так называемая отдача в момент времени  $t$  представляет собой накопленные награды, полученные на всем протяжении эпизода. Вспомните, что  $R_{t+1} = r$  — это *немедленная награда*, полученная после выполнения действия  $A_t$  в момент времени  $t$ ; более поздними наградами являются  $R_{t+2}$ ,  $R_{t+3}$  и т.д.

Тогда отдача в момент времени  $t$  может быть рассчитана из немедленной награды и более поздних наград следующим образом:

$$G_t \triangleq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Здесь  $\gamma$  — коэффициент дисконтирования с диапазоном  $[0, 1]$ . Параметр  $\gamma$  указывает, насколько будущие награды “ценны” в текущий момент времени ( $t$ ). Обратите внимание, что установка  $\gamma = 0$  подразумевает, что нас не заботят будущие награды. В таком случае отдача будет равна немедленной награде, а более поздние награды после момента времени  $t + 1$  игнорируются, и агент окажется “близоруким”. С другой стороны, если  $\gamma = 1$ , тогда отдача будет невзвешенной суммой всех более поздних наград.

Кроме того, следует отметить, что уравнение для отдачи может быть выражено проще с использованием рекурсии:

$$G_t = R_{t+1} + \gamma G_{t+1} = r + \gamma G_{t+1}$$

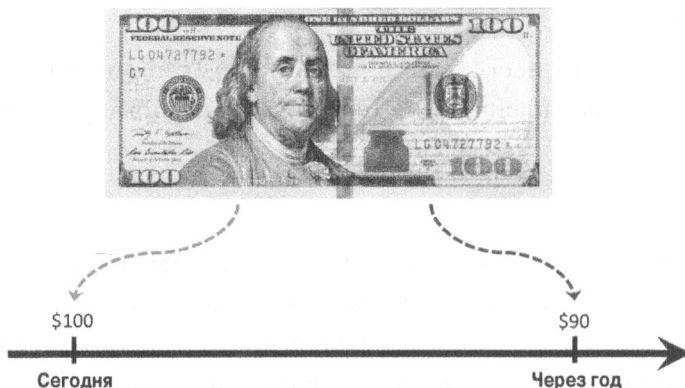
Смысл в том, что отдача в момент времени  $t$  равна немедленной награде  $r$  плюс дисконтированная будущая отдача в момент времени  $t + 1$ . Это очень важное свойство, которое облегчает вычисления отдачи.



На заметку!

### Смысл коэффициента дисконтирования

Чтобы получить представление о коэффициенте дисконтирования, взгляните на показанный ниже рисунок, который иллюстрирует ценность заработка стодолларовой банкноты сегодня в сравнении с ее заработком через год. При определенных экономических условиях, таких как инфляция, заработок стодолларовой банкноты прямо сейчас может оказаться более ценным, нежели ее заработок в будущем.



Таким образом, мы говорим, что если эта банкнота стоит \$100 прямо сейчас, тогда через год она будет стоить \$90 с учетом коэффициента дисконтирования  $\gamma = 0.9$ .

Давайте рассчитаем отдачу на разных временных шагах для эпизодов, взятых из предыдущего примера со студентом. Пусть  $\gamma = 0.9$ , а единственная предоставляемая награда основана на результате экзамена (+1 за его сдачу и -1 за провал). Награды на промежуточных временных шагах равны 0.

- **Эпизод 1:** BBCCCCBAT  $\rightarrow$  экзамен сдан (финальная награда = +1)

$$t = 0 : G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots + \gamma^6 R_7$$

$$\rightarrow G_0 = 0 + 0 \times \gamma + \dots + 1 \times \gamma^6 = 0.9^6 \approx 0.531$$

$$t = 1 : G_1 = 1 \times \gamma^5 = 0.590$$

$$t = 2 : G_2 = 1 \times \gamma^4 = 0.656$$

...

$$t = 6 : G_6 = 1 \times \gamma = 0.9$$

$$t = 7 : G_7 = 1 = 1$$

- **Эпизод 2:** ABBBBBBBBBT  $\rightarrow$  экзамен провален (финальная награда = -1)

$$t = 0 : G_0 = -1 \times \gamma^8 = -0.430$$

$$t = 1 : G_1 = -1 \times \gamma^7 = -0.478$$

...

$$t = 8 : G_8 = -1 \times \gamma = -0.9$$

$$t = 9 : G_9 = -1 = -1$$

Расчет отдачи для третьего эпизода оставлен в качестве упражнения для самостоятельной проработки.

## Политика

*Политика*, обычно обозначаемая как  $\pi(a | s)$ , представляет собой функцию, которая определяет действие, предпринимаемое следующим, и может быть либо детерминированной, либо стохастической (т.е. давать вероятность выбора следующего действия). Стохастическая политика имеет распределение вероятностей по действиям, которые агент может предпринимать в заданном состоянии:

$$\pi(a | s) \stackrel{\text{def}}{=} P[A_t = a | S_t = s]$$



В процессе обучения политика может изменяться по мере того, как агент набирается опыта. Например, агент мог бы начать со случайной политики, где распределение вероятностей всех действий равномерно; в то же время агент благополучно научится оптимизировать свою политику в направлении достижения оптимальной политики. *Оптимальной политикой*  $\pi_*(a | s)$  считается такая политика, которая обеспечивает наивысшую отдачу.

## Функция ценности

*Функция ценности (value function)*, также называемая *функцией ценности состояния (state-value function)*, измеряет *доброкачественность* каждого состояния — другими словами, насколько хорошим или плохим должно быть индивидуальное состояние. Обратите внимание, что критерий доброкачественности базируется на отдаче.

Теперь, основываясь на отдаче  $G_t$ , мы определяем функцию ценности состояния  $s$  как ожидаемую отдачу (среднюю отдачу по всем возможным эпизодам) после *следования политике*  $\pi$ :

$$v_{\pi}(s) \stackrel{\text{def}}{=} E_{\pi} [ G_t | S_t = s ] = E_{\pi} \left[ \sum_{k=0}^{\infty} \gamma^{k+1} R_{t+k+1} | S_t = s \right]$$

В реальной реализации мы обычно оцениваем функцию ценности с применением таблиц поиска, а потому у нас нет нужды повторно вычислять ее много раз. (Это аспект динамического программирования.) Скажем, на практике, когда мы оцениваем функцию ценности, используя такие табличные методы, то сохраняем ценности состояний в таблице, обозначаемой как  $V(s)$ . В реализации на языке Python ею может быть список или массив NumPy, индексы которого соответствуют различным состояниям, либо словарь Python, чьи ключи отображают состояния на надлежащие ценности.

Кроме того, мы также можем определить ценность для каждой пары “состояние–действие”, что называется *функцией ценности действия (action-value function)* и обозначается как  $q_{\pi}(s, a)$ . Функция ценности действия ссылается на ожидаемую отдачу  $G_t$ , когда агент находится в состоянии  $S_t = s$  и предпринимает действие  $A_t = a$ . Расширив определение функции ценности состояния на пары “состояние–действие”, вот что мы получим:

$$q_{\pi}(s, a) \stackrel{\text{def}}{=} E_{\pi} [ G_t | S_t = s, A_t = a ] = E_{\pi} \left[ \sum_{k=0}^{\infty} \gamma^{k+1} R_{t+k+1} | S_t = s, A_t = a \right]$$

Подобно обозначению оптимальной политики как  $\pi_*(a | s)$ ,  $v_*(s)$  и  $q_*(s, a)$  также обозначают оптимальные функции ценности состояния и ценности действия.

Оценка функции ценности является важным компонентом методов обучения с подкреплением. Мы раскроем различные способы вычисления и оценки функций ценности состояния и ценности действия позже в этой главе.



### Отличие между наградой, отдачей и функцией ценности

*Награда* является последствием того, что агент предпринял некоторое действие при заданном текущем состоянии среды. Другими словами, награда представляет собой сигнал, который агент получает, когда выполняет действие для перехода из одного состояния в другое. Тем не менее, не забывайте о том, что не каждое действие выдает положительную или отрицательную награду — вспомните о нашем примере с шахматами, где положительная награда получалась только при выигрыше партии, а награды для всех промежуточных действий были нулевыми.

Само состояние имеет определенную ценность, которую мы ему назначаем для измерения того, насколько хорошим или плохим оно является — именно здесь в игру вступает функция ценности. Обычно состояния с “высокой” или “достойной” ценностью будут теми состояниями, которые имеют высокую ожидаемую *отдачу*, и вероятно будут выдавать высокую награду с учетом конкретной политики.

Например, давайте еще раз обратимся к компьютеру, играющему в шахматы. Положительная награда может предоставляться лишь в конце игры, если компьютер выиграл. В случае проигрыша компьютера нет никакой (положительной) награды. Теперь предположим, что компьютер делает специфический ход, который берет ферзя соперника безо всяких отрицательных последствий для себя. Поскольку компьютер получает награду только в случае выигрыша партии, он не принимает немедленную награду, делая ход, на котором берется ферзь соперника. Однако новое состояние (состояние доски после взятия ферзя) может иметь *высокую ценность*, которая может выдать награду (если позже партия выиграна). По идее высокая ценность, ассоциированная с взятием ферзя соперника, связана с тем фактом, что это часто в итоге обеспечивает выигрыш партии — и соответственно высокую ожидаемую отдачу, или ценность. Тем не менее, обратите

внимание, что взятие ферзя соперника далеко не всегда приводит к выигрышу партии; следовательно, агент, по всей видимости, получит положительную награду, но не гарантированно.

Короче говоря, *отдача* представляет собой взвешенную сумму *наград* для полного эпизода, которая будет равна дисконтированной финальной награде в нашем примере с шахматами (т.к. есть только одна награда). *Функция ценности* является математическим ожиданием по всем возможным эпизодам, которая по существу рассчитывает то, насколько в среднем “ценно” делать определенный ход.

Перед тем, как переходить непосредственно к алгоритмам обучения с подкреплением, давайте кратко пройдемся по выведению уравнения Беллмана, которое мы можем применять для реализации оценки политики.

## Динамическое программирование с использованием уравнения Беллмана

Уравнение Беллмана — один из центральных элементов многих алгоритмов обучения с подкреплением. Уравнение Беллмана упрощает расчет функции ценности, так что вместо суммирования в течение множества временных шагов применяется рекурсия, которая похожа на рекурсию для вычисления отдачи.

Базируясь на рекурсивном уравнении для общей отдачи,  $G_t = r + \gamma G_{t+1}$ , мы можем переписать функцию ценности следующим образом:

$$\begin{aligned} v_{\pi}(s) &\stackrel{\text{def}}{=} E_{\pi}[G_t | S_t = s] \\ &= E_{\pi}[r + \gamma G_{t+1} | S_t = s] \\ &= r + \gamma E_{\pi}[G_{t+1} | S_t = s] \end{aligned}$$

Обратите внимание, что немедленная награда  $r$  вынесена из математического ожидания, поскольку она является константой с известной величиной в момент времени  $t$ .

Аналогично мы можем записать для функции ценности действия:

$$\begin{aligned} q_{\pi}(s, a) &\stackrel{\text{def}}{=} E_{\pi}[G_t | S_t = s, A_t = a] \\ &= E_{\pi}[r + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= r + \gamma E_{\pi}[G_{t+1} | S_t = s, A_t = a] \end{aligned}$$

Для расчета математического ожидания мы можем использовать динамику среды, взяв сумму по всем вероятностям следующего состояния  $s'$  и соответствующих наград  $r$ :

$$v_{\pi}(s) = \sum_{a \in \hat{A}} \pi(a | s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r | s, a) [r + \gamma E_{\pi}[G_{t+1} | S_{t+1} = s']]$$

Теперь мы видим, что математическое ожидание отдачи,  $E_{\pi}[G_{t+1} | S_{t+1} = s']$ , по существу является функцией ценности состояния,  $v_{\pi}(s')$ . Таким образом, мы можем записать  $v_{\pi}(s)$  как функцию от  $v_{\pi}(s')$ :

$$v_{\pi}(s) = \sum_{a \in \hat{A}} \pi(a | s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r | s, a) [r' + \gamma v_{\pi}(s')]$$

Результат называется *уравнением Беллмана*, которое связывает функцию ценности для состояния  $s$  с функцией ценности для последующего состояния  $s'$ . Оно значительно упрощает вычисление функции ценности, потому что устраняет итерационный цикл по оси времени.

## Алгоритмы обучения с подкреплением

В текущем разделе мы рассмотрим комплект алгоритмов обучения и начнем с динамического программирования, которое предполагает, что динамика переходов (или динамика среды, т.е.  $p(s', r | s, a)$ ) известна. Однако в большинстве задач обучения с подкреплением это не так. Чтобы справиться с проблемой неизвестной динамики среды, были разработаны методики, предусматривающие обучение через взаимодействие со средой. В их число входят метод Монте-Карло, метод временных разностей и набирающие все большую популярность приемы Q-обучения и глубокого Q-обучения. На рис. 18.3 описан процесс продвижения алгоритмов обучения с подкреплением от динамического программирования до Q-обучения.

В последующих разделах главы мы пошагово разберем каждый алгоритм обучения с подкреплением, указанный на рис. 18.3. Мы начнем с динамического программирования, затем перейдем к методу Монте-Карло и в заключение обсудим метод временных разностей, а также его ответвления — метод внутри политики, *SARSA* (*state-action-reward-state-action* — состояние–действие–награда–состояние–действие), и метод вне политики, Q-обучение. Во время создания нескольких практических моделей мы также затронем метод глубокого Q-обучения.



Рис. 18.3. Процесс продвижения алгоритмов обучения с подкреплением

## Динамическое программирование

В этом разделе мы сосредоточим внимание на решении задач обучения с подкреплением при следующих допущениях:

- мы обладаем полным знанием динамики среды, т.е. вероятности всех переходов  $p(s', r | s, a)$  известны;
- состояние агента имеет марковское свойство, а потому следующее действие и награда зависят только от текущего состояния и выбора действия, которое мы делаем в данный момент или на текущем временном шаге.

Математическая формулировка для задач обучения с подкреплением, использующая марковский процесс принятия решений (MDP), была представлена в разделе “Математическая формулировка марковских процессов принятия решений” ранее в главе. Там было введено формальное определение функции ценности  $v_\pi(s)$ , следующей политике  $\pi$ , и уравнение Беллмана, которое выводилось с применением динамики среды.

Мы должны подчеркнуть, что динамическое программирование не является практичным подходом к решению задач обучения с подкреплением. Проблема, связанная с использованием динамического программирования, заключается в том, что оно предполагает наличие полного знания динамики среды, которое для большинства реальных приложений обычно необоснованно или нецелесообразно. Тем не менее, с образовательной точки зрения динамическое программирование помогает представить обучение с подкреплением в простой манере и служит мотивом к применению более развитых и сложных алгоритмов.

В задачах, описанных в последующих подразделах, преследуются две цели.

1. Получить точную функцию ценности состояния,  $v_{\pi}(s)$ ; это также известно как задача прогнозирования и выполняется с помощью *оценки политики*.
2. Найти оптимальную функцию ценности,  $v_{*}(s)$ , что достигается посредством *обобщенной итерации политики* (*generalized policy iteration* — GPI).

### Оценка политики — прогнозирование функции ценности с помощью динамического программирования

Основываясь на уравнении Беллмана, мы можем рассчитать функцию ценности для произвольной политики  $\pi$  с помощью динамического программирования, когда динамика среды известна. При расчете этой функции ценности мы можем адаптировать итеративное решение и начать с ценностей  $v^{<0>}(s)$ , инициализированных нулями для всех состояний. Затем на каждой итерации  $i + 1$  мы обновляем ценности всех состояний на базе уравнения Беллмана, которое в свою очередь основано на ценностях состояний из предыдущей итерации  $i$ :

$$v^{<i+1>}(s) = \sum_a \pi(a | s) \sum_{s' \in \hat{S}, r \in \hat{R}} p(s', r' | s, a) [r + \gamma v^{<i>}(s')]$$

Можно показать, что с увеличением количества итераций до бесконечности  $v^{<i>}(s)$  сходится в точную функцию ценности состояния  $v_{\pi}(s)$ .

Также обратите здесь внимание, что нам не нужно взаимодействовать со средой. Причина в том, что мы точно знаем динамику среды. В итоге мы можем задействовать эту информацию и легко оценить функцию ценности.

После расчета функции ценности возникает очевидный вопрос: какую пользу она способна нам принести, если наша политика по-прежнему является случайной? Ответ заключается в том, что на самом деле мы можем использовать рассчитанную функцию ценности  $v_{\pi}(s)$  для улучшения нашей политики, как будет показано далее.

### Улучшение политики с использованием ожидаемой функции ценности

Теперь, когда мы рассчитали функцию ценности  $v_{\pi}(s)$ , следуя существующей политике  $\pi$ , имеет смысл применить  $v_{\pi}(s)$  и улучшить существующую политику  $\pi$ . Это значит, что мы хотим найти новую политику  $\pi'$ , которая для

каждого состояния  $s$ , следующего  $\pi'$ , выдавала бы более высокую или хотя бы равную ценность, чем при использовании существующей политики  $\pi$ . Математически мы можем выразить такую цель для улучшенной политики  $\pi'$  следующим образом:

$$v_{\pi'}(s) \geq v_{\pi}(s) \quad \forall s \in \hat{S}$$

Первым делом вспомните, что политика  $\pi$  определяет вероятность выбора каждого действия  $a$ , пока агент находится в состоянии  $s$ . Чтобы отыскать политику  $\pi'$ , которая всегда имеет лучшую или равную ценность для каждого состояния, мы сначала рассчитываем функцию ценности действия,  $q_{\pi}(s, a)$ , для каждого состояния  $s$  и действия  $a$ , базируясь на вычисленной ценности состояния с применением функции ценности  $v_{\pi}(s)$ . Мы проходим по всем состояниям и для каждого состояния  $s$  сравниваем ценность следующего состояния  $s'$ , куда произошел бы переход в случае выбора действия  $a$ .

После получения наивысшей ценности состояния за счет оценки всех пар “состояние–действие” посредством  $q_{\pi}(s, a)$  мы можем сравнить соответствующее действие с действием, выбранным текущей политикой. Если действие, предлагаемое текущей политикой (т.е.  $\arg \max_a \pi(a | s)$ ), отличается от действия, предлагаемого функцией ценности действия (т.е.  $\arg \max_a q_{\pi}(s, a)$ ), тогда мы можем обновить политику, переназначив вероятности действий с целью соответствия действию, которое дает наивысшую ценность действия,  $q_{\pi}(s, a)$ . Такая процедура называется алгоритмом *улучшения политики*.

### Итерация политики

С использованием алгоритма улучшения политики, описанного в предыдущем подразделе, можно показать, что улучшение политики будет определенно давать лучшую политику, если только текущая политика уже не является оптимальной (т.е.  $v_{\pi}(s) = v_{\pi'}(s) = v_{\star}(s)$  для каждого  $s \in \hat{S}$ ). Следовательно, если мы многократно выполним оценку политики, а за ней улучшение политики, то гарантированно отыщем оптимальную политику.



**Совет**

Обратите внимание, что такая методика называется *обобщенной итерацией политики (GPI)*, которая распространена во многих методах обучения с подкреплением. Мы будем применять GPI далее в главе для методов Монте-Карло и временных разностей.

## Итерация ценности

Мы выяснили, что за счет повторения оценки политики (расчета  $v_{\pi}(s)$  и  $q_{\pi}(s, a)$ ) и улучшения политики (нахождения  $\pi$ , так что  $v_{\pi'}(s) \geq v_{\pi}(s) \forall s \in \hat{S}$ ) можно достичь оптимальной политики. Однако может оказаться эффективнее объединить две задачи — оценку и улучшение политики — в один шаг. В следующем уравнении обновляется функция ценности для итерации  $i+1$  (обозначаемая как  $v^{<i+1>}$ ) на основе действия, которое доводит до максимума взвешенную сумму ценности следующего состояния и его немедленной награды ( $r + \gamma v^{<i>}(s')$ ):

$$v^{<i+1>}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v^{<i>}(s')]$$

В данном случае обновленная ценность для  $v^{<i+1>}(s)$  доводится до максимума путем выбора наилучшего действия из всех возможных действий, тогда как при оценке политики обновленная ценность использовала взвешенную сумму по всем действиям.



На заметку!

### Система обозначений табличных оценок для функций ценности состояния и ценности действия

В большинстве книг и статей, посвященных обучению с подкреплением, строчные  $v_{\pi}$  и  $q_{\pi}$  применяются для ссылки на точные функции ценности состояния и ценности действия как на математические функции.

В то же время для практических реализаций эти функции ценности определяются как таблицы поиска. Табличные оценки данных функций ценности обозначаются как  $V(S_i = s) \approx v_{\pi}(s)$  и  $Q_{\pi}(S_i = s, A_i = a) \approx q_{\pi}(s, a)$ . Мы также будем использовать в главе описанную систему обозначений.

## Обучение с подкреплением с помощью метода Монте-Карло

Как было показано в предыдущем разделе, динамическое программирование полагается на упрощенческое допущение о том, что динамика среды полностью известна. Отойдя от подхода динамического программирования, теперь мы предположим, что не располагаем знаниями о динамике среды.

Таким образом, мы не знаем вероятности переходов между состояниями среды и взамен хотим, чтобы агент учился через *взаимодействие* со средой.



В случае применения метода Монте-Карло процесс обучения основан на так называемом *имитированном опыте* (*simulated experience*).

При обучении с подкреплением на базе метода Монте-Карло мы определяем класс агента, следующего вероятностной политике  $\pi$ , на основе которой агент предпринимает действие на каждом шаге. Результатом будет имитированный эпизод.

Ранее мы определяли функцию ценности состояния, так что ценность состояния указывала ожидаемую отдачу от данного состояния. В динамическом программировании такое вычисление опиралось на знание динамики среды, т.е.  $p(s', r | s, a)$ .

Тем не менее, в дальнейшем мы будем разрабатывать алгоритмы, для которых динамика среды не требуется. Методы на основе Монте-Карло решают эту задачу, генерируя имитированные эпизоды, где агент взаимодействует со средой. Из таких имитированных эпизодов мы сможем рассчитать среднюю отдачу для каждого состояния, посещенного в заданном имитированном эпизоде.

### **Оценка функции ценности состояния с использованием метода Монте-Карло**

После генерации набора эпизодов для каждого состояния  $s$  набор эпизодов, в котором все эпизоды проходят через состояние  $s$ , просматривается с целью вычисления ценности состояния  $s$ . Давайте предположим, что для получения ценности применяется таблица поиска  $V(S_t = s)$ , соответствующая функции ценности. Обновления в методе Монте-Карло для оценки функции ценности базируются на общей отдаче, которая получена в данном эпизоде, начиная с посещения состояния  $s$  в первый раз. Такой алгоритм называется прогнозированием ценности *методом Монте-Карло первого посещения* (*first-visit Monte Carlo*).

### **Оценка функции ценности действия с использованием метода Монте-Карло**

Когда динамика среды известна, мы можем без труда вывести функцию ценности действия из функции ценности состояния, заглядывая на один шаг вперед для нахождения действия, которое дает максимальную ценность, как было показано в разделе “Динамическое программирование” ранее в главе. Однако это неосуществимо, если мы не знаем динамику среды.

Чтобы решить проблему, мы можем расширить алгоритм для оценки прогноза ценности состояния методом Монте-Карло первого посещения. Скажем, мы можем рассчитать *ожидаемую* отдачу для каждой пары “состояние–действие”, применяя функцию ценности действия. Чтобы получить ожидаемую отдачу, мы принимаем во внимание посещения каждой пары “состояние–действие”  $(s, a)$ , что означает пребывание в состоянии  $s$  и принятие действия  $a$ .

Тем не менее, возникает проблема, т.к. некоторые действия могут никогда не выбираться, что приводит к неполному исследованию. Решить проблему можно несколькими способами. Простейший подход называется *исследовательским началом* (*exploratory start*), который предполагает, что каждая пара “состояние–действие” имеет ненулевую вероятность в начале эпизода.

Еще один подход к решению проблемы нехватки исследования называется  *$\epsilon$ -жадной политикой* ( *$\epsilon$ -greedy policy*) и обсуждается в разделе “Улучшение политики — расчет жадной политики из функции ценности действия” далее в главе.

### Нахождение оптимальной политики с использованием контроля Монте-Карло

Под *контролем Монте-Карло* (*MC control*) понимается процедура оптимизации для улучшения политики. Подобно подходу итерации политики из предыдущего раздела (“Динамическое программирование”) мы можем многократно чередовать оценку политики и улучшение политики до тех пор, пока не достигнем оптимальной политики. Таким образом, начиная со случайной политики  $\pi_0$ , процесс чередования оценки политики и улучшения политики может быть проиллюстрирован так, как показано на рис. 18.4.

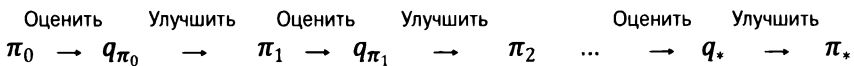


Рис. 18.4. Чередование оценки политики и улучшения политики

### Улучшение политики — расчет жадной политики из функции ценности действия

Имея функцию ценности действия  $q(s, a)$ , вот как можно сгенерировать жадную (детерминированную) политику:

$$\pi(s) \stackrel{\text{def}}{=} \arg \max_a q(s, a)$$

разностей будет идентичным алгоритму TD(0), описанному в предыдущем абзаце. Тем не менее, если  $n \rightarrow \infty$ , то  $n$ -шаговый алгоритм временных разностей будет таким же, как алгоритм Монте-Карло. Ниже приведено правило обновления для  $n$ -шагового алгоритма временных разностей:

$$V(S_t) = V(S_t) + \alpha[G_{t:t+n} - V(S_t)]$$

А  $G_{t:t+n}$  определяется так:

$$G_{t:t+n} \stackrel{\text{def}}{=} \begin{cases} R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}), & \text{если } t+n < T \\ G_{t:T} \end{cases}$$



Совет

### Сравнение метода Монте-Карло и метода временных разностей: какой из них сходится быстрее?

Хотя точный ответ на данный вопрос по-прежнему неизвестен, на практике эмпирически показано, что метод временных разностей способен сходиться быстрее, чем метод Монте-Карло. Дополнительные сведения о сходимости этих методов можно найти в книге “Reinforcement Learning: An Introduction” (Обучение с подкреплением: введение) Ричарда Саттона и Эндрю Барто.

Теперь, когда задача прогнозирования с применением метода временных разностей полностью раскрыта, мы можем переходить к задаче контроля. Мы рассмотрим два алгоритма для контроля временных разностей: контроль *внутри политики* и контроль *вне политики*. В обоих случаях мы используем обобщенную итерацию политики (GPI), которая применялась в динамическом программировании и алгоритмах Монте-Карло. При контроле временных разностей *внутри политики* функция ценности обновляется на основе действий из той же политики, которой следует агент, тогда как в алгоритме *вне политики* функция ценности обновляется на основе действий за пределами текущей политики.

### Контроль временных разностей внутри политики (SARSA)

Для простоты мы обсудим только одношаговый алгоритм TD(0). Однако алгоритм контроля временных разностей *внутри политики* можно легко обобщить до  $n$ -шагового алгоритма временных разностей. Мы начнем с расширения формулы прогнозирования, определив функцию ценности со-

стояния для описания функции ценности действия. Мы будем использовать таблицу поиска, т.е. табличный двумерный массив  $Q(S_t, A_t)$ , который представляет функцию ценности действия для каждой пары “состояние–действие”. В нашем случае вот что получается:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Такой алгоритм часто называют SARSA из-за пятерки  $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ , которая применяется в формуле обновления.

Как было показано в предшествующих разделах, посвященных динамическому программированию и методу Монте-Карло, мы можем использовать GPI и, начав со случайной политики, многократно оценивать функцию ценности действия для текущей политики, после чего оптимизировать политику с применением  $\varepsilon$ -жадной политики, основываясь на текущей функции ценности действия.

### Контроль временных разностей вне политики (Q-обучение)

При использовании предыдущего алгоритма контроля временных разностей внутри политики мы видели, что способ оценки функции ценности действия базируется на политике, которая применяется в имитированном эпизоде. После обновления функции ценности действия выполняется отдельный шаг для улучшения политики путем инициирования действия с наивысшей ценностью.

Альтернативный (и лучший) подход предусматривает объединение этих двух шагов. Другими словами, представьте, что агент следует политике  $\pi$ , генерируя эпизод с пятеркой  $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$  текущего перехода. Вместо обновления функции ценности действия с использованием ценности действия  $A_{t+1}$ , которое предпринято агентом, мы можем отыскивать наилучшее действие, даже если оно фактически не выбрано агентом, следующим текущей политике. (Вот почему он считается алгоритмом *вне политики*.)

Для этого мы можем модифицировать правило обновления, чтобы учитывать максимальную Q-ценность, варьируя различные действия в следующем состоянии. Ниже приведено модифицированное уравнение для обновления Q-ценностей:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Мы рекомендуем вам сравнить показанное правило обновления с правилом обновления алгоритма SARSA. Как видите, мы находим наилучшее действие в следующем состоянии  $S_{t+1}$  и задействуем его в корректирующем члене для обновления оценки  $Q(S_t, A_t)$ .

Чтобы представить все материалы в соответствующем контексте, в следующем разделе мы посмотрим, как реализовать алгоритм Q-обучения для решения задачи с миром сетки (*grid world*).

## Реализация первого алгоритма обучения с подкреплением

В настоящем разделе мы раскроем реализацию алгоритма Q-обучения для решения задачи с миром сетки. Здесь мы задействуем комплект инструментов OpenAI Gym.

### Введение в комплект инструментов OpenAI Gym

OpenAI Gym представляет собой специализированный комплект инструментов для упрощения разработки моделей обучения с подкреплением. Комплект OpenAI Gym поступает с несколькими предварительно определенными средами. В число базовых примеров входят CartPole и MountainCar, задачами которых являются соответственно балансировка дышла телеги и движение автомобиля вверх на возвышенность. Существует также много развитых робототехнических сред для обучения робота выбирать, толкать и добираться до предметов на лабораторном столе либо обучения механической руки определять местонахождение кубиков, шариков или ручек. Вдобавок OpenAI Gym предоставляет удобный унифицированный фреймворк для разработки новых сред. Дополнительные сведения доступны на официальном веб-сайте: <https://gym.openai.com/>.

Для проработки примеров кода OpenAI Gym в последующих разделах вам понадобится установить библиотеку gym, что легко сделать с применением pip:

```
> pip install gym
```

Если нужна помощь по установке, тогда обратитесь к соответствующему руководству по ссылке <https://gym.openai.com/docs/#installation>.

## Работа с существующими средами в OpenAI Gym

Чтобы попрактиковаться со средами Gym, мы создадим среду из `CartPole-v1`, которая входит в состав OpenAI Gym. В этом примере среды имеется дышло, прикрепленное к телеге, которые могут двигаться горизонтально, как показано на рис. 18.5.



Рис. 18.5. Среда `CartPole-v1`

Движение дышла регулируется законами физики. Цель агента обучения с подкреплением заключается в том, чтобы выяснить, каким образом перемещать телегу для стабилизации дышла и предотвращения его опрокидывания в любую сторону.

Давайте взглянем на некоторые свойства среды `CartPole` в контексте обучения с подкреплением, такие как ее пространство состояний (или наблюдений), пространство действий и способ запуска действия:

```
>>> import gym
>>> env = gym.make('CartPole-v1')
>>> env.observation_space
Box(4, )
>>> env.action_space
Discrete(2)
```

В предыдущем коде мы создали среду для задачи с `CartPole`. Пространством наблюдений для этой среды является `Box(4, )`, что представляет четырехмерное пространство, соответствующее четырем вещественным числам: местоположение телеги, скорость телеги, угол отклонения дышла и угловая скорость вершины дышла. Пространство действий — дискретное пространство `Discrete(2)` с двумя вариантами: толкание телеги влево или вправо.

Объект среды `env`, созданный ранее вызовом `gym.make('CartPole-v1')`, имеет метод `reset()`, который мы можем использовать для повторной инициализации.

циализации среды перед каждым эпизодом. Вызов метода `reset()` по существу будет устанавливать начальное состояние дышла ( $S_0$ ):

```
>>> env.reset()
array([-0.03908273, -0.00837535,  0.03277162, -0.0207195 ])
```

Значения в возвращенном из метода `env.reset()` массиве означают, что телега имеет начальное местоположение  $-0.039$  при скорости  $-0.008$ , а угол отклонения дышла составляет  $0.033$  радиана при угловой скорости  $-0.021$ . Когда вызывается метод `reset()`, упомянутые значения инициализируются случайными величинами с равномерным распределением в диапазоне  $[-0.05, 0.05]$ .

После сброса среды мы можем взаимодействовать с ней, выбирая действие и запуская его за счет передачи методу `step()`:

```
>>> env.step(action=0)
(array([-0.03925023, -0.20395158,  0.03235723,  0.28212046]),
 1.0, False, {})
>>> env.step(action=1)
(array([-0.04332927, -0.00930575,  0.03799964, -0.00018409]),
 1.0, False, {})
```

Посредством предыдущих двух команд, `env.step(action=0)` и `env.step(action=1)`, мы толкнули телегу влево (`action=0`) и затем вправо (`action=1`). Базируясь на выбранном действии, телега и ее дышло могут перемещаться в соответствии с законами физики. Каждый раз, когда мы вызываем метод `env.step()`, он возвращает кортеж, состоящий из четырех элементов:

- массив для нового состояния (или наблюдений);
- награда (скалярное значение типа `float`);
- флаг окончания (`True` или `False`);
- словарь Python, содержащий вспомогательную информацию.

Объект `env` также имеет метод `render()`, который мы можем вызывать после каждого шага (или последовательности шагов) для визуализации среды и движения телеги с дышлом во времени.

Эпизод заканчивается, когда угол наклона дышла становится больше  $12$  градусов (в любую сторону) относительно воображаемой вертикальной оси

или местоположение телеги сдвигается больше, чем на 2.4 единицы, от позиции центра. Награда, определенная в этом примере, заключается в доведении до максимума времени, в течение которого телега и дышло стабилизированы внутри допустимых областей. Другими словами общая награда (т.е. отдача) может быть доведена до максимума за счет максимизации длительности эпизода.

### Пример с миром сетки

После представления среды CartPole в качестве разминки для работы с комплектом инструментов OpenAI Gym мы перейдем к другой среде. Мы рассмотрим пример с миром сетки, который является упрощенческой средой, имеющей  $m$  строк  $n$  столбцов. Принимая  $m = 4$  и  $n = 6$ , мы можем изобразить среду, как показано на рис. 18.6.



Рис. 18.6. Среда мира сетки

В среде есть 30 разных возможных состояний. Четыре состояния являются заключительными: горшок с золотом в состоянии 16 и три ловушки в состояниях 10, 15 и 22. Попадание в любое из заключительных состояний заканчивает эпизод, но с отличием между "золотым" состоянием и ловушками. Попадание в "золотое" состояние выдает положительную награду +1, тогда как перемещение агента в одну из ловушек выдает отрицательную награду -1. Все остальные состояния имеют награду 0. Агент всегда начинает с состояния 0. Следовательно, всякий раз, когда мы сбрасываем среду, агент будет возвращаться обратно в состояние 0. Пространство состояний состоит из четырех направлений: перемещения вверх, вниз, влево и вправо. Когда агент находится на внешней границе сетки, выбор действия, которое привело бы к покиданию сетки, не изменяет состояние.



Далее мы посмотрим, как реализовать такую среду на языке Python с применением пакета OpenAI Gym.

### Реализация среды мира сетки в OpenAI Gym

При экспериментировании со средой мира сетки посредством OpenAI Gym настоятельно рекомендуется использовать редактор сценариев или IDE-среду, а не выполнять код интерактивно.

Первым делом мы создаем новый сценарий Python по имени `gridworld_env.py`, после чего импортируем необходимые пакеты и две вспомогательных функции, которые определим для построения и визуализации среды.

Для визуализации среды библиотека OpenAI Gym применяет библиотеку Pyglet и предоставляет удобные классы-оболочки и функции. Мы будем использовать эти классы-оболочки для визуализации среды мира сетки в следующем примере кода. Дополнительные детали о классах-оболочках ищите по ссылке [https://github.com/openai/gym/blob/master/gym/envs/classic\\_control/rendering.py](https://github.com/openai/gym/blob/master/gym/envs/classic_control/rendering.py).

Ниже приведен пример кода, где задействованы классы-оболочки:

```
## Сценарий: gridworld_env.py

import numpy as np
from gym.envs.toy_text import discrete
from collections import defaultdict
import time
import pickle
import os

from gym.envs.classic_control import rendering

CELL_SIZE = 100
MARGIN = 10

def get_coords(row, col, loc='center'):
    xc = (col+1.5) * CELL_SIZE
    yc = (row+1.5) * CELL_SIZE
    if loc == 'center':
        return xc, yc
    elif loc == 'interior_corners':
        half_size = CELL_SIZE//2 - MARGIN
        xl, xr = xc - half_size, xc + half_size
        yt, yb = yc - half_size, yc + half_size
        return [(xl, yt), (xr, yt), (xr, yb), (xl, yb)]
```

```

elif loc == 'interior_triangle':
    x1, y1 = xc, yc + CELL_SIZE//3
    x2, y2 = xc + CELL_SIZE//3, yc - CELL_SIZE//3
    x3, y3 = xc - CELL_SIZE//3, yc - CELL_SIZE//3
    return [(x1, y1), (x2, y2), (x3, y3)]

def draw_object(coords_list):
    if len(coords_list) == 1:          # -> круг
        obj = rendering.make_circle(int(0.45*CELL_SIZE))
        obj_transform = rendering.Transform()
        obj.add_attr(obj_transform)
        obj_transform.set_translation(*coords_list[0])
        obj.set_color(0.2, 0.2, 0.2)  # -> черный
    elif len(coords_list) == 3:        # -> треугольник
        obj = rendering.FilledPolygon(coords_list)
        obj.set_color(0.9, 0.6, 0.2)  # -> желтый
    elif len(coords_list) > 3:         # -> многоугольник
        obj = rendering.FilledPolygon(coords_list)
        obj.set_color(0.4, 0.4, 0.8)  # -> синий
    return obj

```

Первая вспомогательная функция, `get_coords()`, возвращает координаты геометрических фигур, которые мы будем применять для аннотирования среды мира сетки, например, треугольника для отображения золота или кругов для отображения ловушек. Список координат передается функции `draw_object()`, которая на основе длины входного списка координат принимает решение о том, что вычерчивать — круг, треугольник или многоугольник.

Теперь мы можем определить среду мира сетки. В том же самом файле (`gridworld_env.py`) мы определяем класс по имени `GridWorldEnv`, унаследованный от класса `DiscreteEnv` из `OpenAI Gym`. Самой важной функцией в этом классе является метод конструктора `__init__()`, где мы определяем пространство действий, указываем роль каждого действия и задаем заключительные состояния (“золотое” и ловушек):

```

class GridWorldEnv(discrete.DiscreteEnv):
    def __init__(self, num_rows=4, num_cols=6, delay=0.05):
        self.num_rows = num_rows
        self.num_cols = num_cols

        self.delay = delay

        move_up = lambda row, col: (max(row-1, 0), col)

```

```

move_down = lambda row, col: (min(row+1, num_rows-1), col)
move_left = lambda row, col: (row, max(col-1, 0))
move_right = lambda row, col: (
    row, min(col+1, num_cols-1))

self.action_defs={0: move_up, 1: move_right,
                  2: move_down, 3: move_left}

## Количество состояний/действий
nS = num_cols*num_rows
nA = len(self.action_defs)
self.grid2state_dict={ (s//num_cols, s%num_cols):s
                       for s in range(nS) }
self.state2grid_dict={s:(s//num_cols, s%num_cols)
                      for s in range(nS) }

## "Золотое" состояние
gold_cell = (num_rows//2, num_cols-2)

## Состояния ловушек
trap_cells = [(gold_cell[0]+1), gold_cell[1]],
              (gold_cell[0], gold_cell[1]-1),
              ((gold_cell[0]-1), gold_cell[1])]

gold_state = self.grid2state_dict[gold_cell]
trap_states = [self.grid2state_dict[(r, c)]
               for (r, c) in trap_cells]
self.terminal_states = [gold_state] + trap_states
print(self.terminal_states)

## Построение вероятностей переходов
P = defaultdict(dict)
for s in range(nS):
    row, col = self.state2grid_dict[s]
    P[s] = defaultdict(list)
    for a in range(nA):
        action = self.action_defs[a]
        next_s = self.grid2state_dict[action(row, col)]

        ## Заключительное состояние
        if self.is_terminal(next_s):
            _r = (1.0 if next_s == self.terminal_states[0]
                  else -1.0)
        else:
            r = 0.0

```

```

        if self.is_terminal(s):
            done = True
            next_s = s
        else:
            done = False
        P[s][a] = [(1.0, next_s, r, done)]

## Начальное распределение состояний
isd = np.zeros(nS)
isd[0] = 1.0

super(GridWorldEnv, self).__init__(nS, nA, P, isd)

self.viewer = None
self._build_display(gold_cell, trap_cells)

def is_terminal(self, state):
    return state in self.terminal_states

def _build_display(self, gold_cell, trap_cells):
    screen_width = (self.num_cols+2) * CELL_SIZE
    screen_height = (self.num_rows+2) * CELL_SIZE
    self.viewer = rendering.Viewer(screen_width,
                                    screen_height)

    all_objects = []

    ## Список координат граничных точек
    bp_list = [
        (CELL_SIZE-MARGIN, CELL_SIZE-MARGIN),
        (screen_width-CELL_SIZE+MARGIN, CELL_SIZE-MARGIN),
        (screen_width-CELL_SIZE+MARGIN,
         screen_height-CELL_SIZE+MARGIN),
        (CELL_SIZE-MARGIN, screen_height-CELL_SIZE+MARGIN)
    ]
    border = rendering.PolyLine(bp_list, True)
    border.set_linewidth(5)
    all_objects.append(border)

    ## Вертикальные линии
    for col in range(self.num_cols+1):
        x1, y1 = (col+1)*CELL_SIZE, CELL_SIZE
        x2, y2 = (col+1)*CELL_SIZE, \
            (self.num_rows+1)*CELL_SIZE
        line = rendering.PolyLine([(x1, y1), (x2, y2)], False)
        all_objects.append(line)

```

```
## Горизонтальные линии
for row in range(self.num_rows+1):
    x1, y1 = CELL_SIZE, (row+1)*CELL_SIZE
    x2, y2 = (self.num_cols+1)*CELL_SIZE,\
              (row+1)*CELL_SIZE
    line=rendering.PolyLine([(x1, y1), (x2, y2)], False)
    all_objects.append(line)

## Ловушки: --> круги
for cell in trap_cells:
    trap_coords = get_coords(*cell, loc='center')
    all_objects.append(draw_object([trap_coords]))

## Золото: --> треугольник
gold_coords = get_coords(*gold_cell,
                        loc='interior_triangle')
all_objects.append(draw_object(gold_coords))

## Агент: --> квадрат или робот
if (os.path.exists('robot-coordinates.pkl') and
    CELL_SIZE==100):
    agent_coords = pickle.load(
        open('robot-coordinates.pkl', 'rb'))
    starting_coords = get_coords(0, 0, loc='center')
    agent_coords += np.array(starting_coords)
else:
    agent_coords = get_coords(
        0, 0, loc='interior_corners')
    agent = draw_object(agent_coords)
    self.agent_trans = rendering.Transform()
    agent.add_attr(self.agent_trans)
    all_objects.append(agent)

for obj in all_objects:
    self.viewer.add_geom(obj)

def render(self, mode='human', done=False):
    if done:
        sleep_time = 1
    else:
        sleep_time = self.delay
    x_coord = self.s % self.num_cols
    y_coord = self.s // self.num_cols
    x_coord = (x_coord+0) * CELL_SIZE
    y_coord = (y_coord+0) * CELL_SIZE
```

```

self.agent_trans.set_translation(x_coord, y_coord)
rend = self.viewer.render(
    return_rgb_array=(mode=='rgb_array'))
time.sleep(sleep_time)
return rend

def close(self):
    if self.viewer:
        self.viewer.close()
        self.viewer = None

```

В коде реализован класс среды мира сетки, экземпляры которого мы можем создавать. Затем мы можем взаимодействовать с экземпляром в манере, похожей на взаимодействие в примере с CartPole. Класс реализации GridWorldEnv наследует такие методы, как `reset()` для сброса состояния и `step()` для запуска действия. Ниже описаны детали реализации.

- Мы определяем четыре разных действия, используя лямбда-функции: `move_up()`, `move_down()`, `move_left()` и `move_right()`.
- Массив NumPy по имени `isd` содержит вероятности начальных состояний, так что при вызове метода `reset()` (из родительского класса) случайное состояние будет выбираться на основе этого распределения. Поскольку мы всегда начинаем с состояния 0 (левый нижний угол мира сетки), то устанавливаем вероятность состояния 0 в 1.0, а вероятности всех остальных 29 состояний в 0.0.
- Вероятности переходов, определенные в словаре Python по имени `P`, устанавливают вероятности перемещения из одного состояния в другое при выборе некоторого действия. В итоге мы располагаем вероятностной средой, где выбор действия может иметь отличающиеся исходы в зависимости от стохастичности среды. Для простоты мы имеем только один исход, которым является изменение состояния в направлении выбранного действия. Наконец, вероятности переходов будут применяться функцией `env.step()` для определения следующего состояния.
- Кроме того, функция `_build_display()` будет настраивать начальную визуализацию среды, а функция `render()` — показывать перемещения агента.

**Совет**

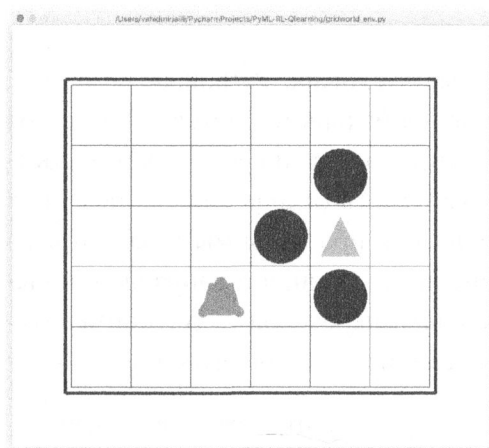
Обратите внимание, что во время процесса обучения мы не знаем вероятности переходов, а наша цель — обучение через взаимодействие со средой. Таким образом, у нас нет доступа к  $P$  за пределами определения класса.

Итак, мы можем протестировать реализацию: создать новую среду и визуализировать случайный эпизод, выбирая в каждом состоянии случайные действия. Поместите следующий код в конец того же сценария (`gridworld_env.py`) и запустите сценарий:

```
if __name__ == '__main__':
    env = GridWorldEnv(5, 6)
    for i in range(1):
        s = env.reset()
        env.render(mode='human', done=False)

        while True:
            action = np.random.choice(env.nA)
            res = env.step(action)
            print('Action ', env.s, action, ' -> ', res)
            env.render(mode='human', done=res[2])
            if res[2]:
                break

    env.close()
```



**Рис. 18.7.** Визуализация среды сетки мира

После выполнения сценария вы должны увидеть визуализацию среды сетки мира, как показано на рис. 18.7.

### Решение задачи с миром сетки с помощью Q-обучения

После рассмотрения теории и процесса разработки алгоритмов обучения с подкреплением, а также настройки среды посредством OpenAI Gym мы займемся реализацией самого популярного в настоящее время алгоритма из данной об-

ласти — Q-обучения. Для этого мы будем использовать пример с сеткой мира, который уже реализован в сценарии `gridworld_env.py`.

## Реализация алгоритма Q-обучения

Далее мы создадим новый сценарий и назовем его `agent.py`. Внутри сценария `agent.py` мы определяем агент для взаимодействия со средой:

```
## Сценарий: agent.py

from collections import defaultdict
import numpy as np

class Agent(object):
    def __init__(
        self, env,
        learning_rate=0.01,
        discount_factor=0.9,
        epsilon_greedy=0.9,
        epsilon_min=0.1,
        epsilon_decay=0.95):
        self.env = env
        self.lr = learning_rate
        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay

        ## Определение q_table
        self.q_table = defaultdict(lambda: np.zeros(self.env.nA))

    def choose_action(self, state):
        if np.random.uniform() < self.epsilon:
            action = np.random.choice(self.env.nA)
        else:
            q_vals = self.q_table[state]
            perm_actions = np.random.permutation(self.env.nA)
            q_vals = [q_vals[a] for a in perm_actions]
            perm_q_argmax = np.argmax(q_vals)
            action = perm_actions[perm_q_argmax]
        return action

    def _learn(self, transition):
        s, a, r, next_s, done = transition
        q_val = self.q_table[s][a]
```



```

    if done:
        q_target = r
    else:
        q_target = r + self.gamma*np.max(self.q_table[next_s])

    ## Обновление q_table
    self.q_table[s][a] += self.lr * (q_target - q_val)

    ## Корректировка эпсилон
    self._adjust_epsilon()

def _adjust_epsilon(self):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

```

Конструктор `__init__()` настраивает разнообразные гиперпараметры, такие как скорость обучения, коэффициент дисконтирования ( $\gamma$ ), а также параметры для  $\varepsilon$ -жадной политики. В исходном положении мы начинаем с высокого значения  $\varepsilon$ , но метод `_adjust_epsilon()` понижает его до тех пор, пока не достигнет минимального значения,  $\varepsilon_{\text{мин}}$ . Метод `choose_action()` выбирает действие на основе  $\varepsilon$ -жадной политики следующим образом. Для определения, должно ли действие выбираться случайно или же иным способом, на основе функции ценности действия, отбирается случайное число с равномерным распределением. Метод `_learn()` реализует правило обновления для алгоритма Q-обучения. Для каждого перехода он получает кортеж, содержащий текущее состояние ( $s$ ), выбранное действие ( $a$ ), наблюдаемую награду ( $r$ ), следующее состояние ( $s'$ ) и флаг, который устанавливает, достигнут ли конец эпизода. Целевая ценность равна наблюдаемой награде ( $r$ ), если переход помечен как конец эпизода; в противном случае целевая ценность вычисляется как  $r + \gamma \max_a Q((s', a))$ .

На следующем шаге мы создаем новый сценарий `qlearning.py`, чтобы собрать все вместе и обучить агента с применением алгоритма Q-обучения.

В показанном ниже коде мы определяем функцию `run_qlearning()`, которая реализует алгоритм Q-обучения, имитируя эпизод с помощью вызова метода `_choose_action()` агента и запуска среды. Затем кортеж перехода передается методу `_learn()` агента для обновления функции ценности действия. Вдобавок для отслеживания процесса обучения мы также сохраняем финальную награду каждого эпизода (она может составлять  $-1$  или  $+1$ ) и длины эпизодов (количество перемещений, предпринятых агентом с начала до конца эпизода).

В заключение посредством функции `plot_learning_history()` вычерчиваются графики для финальных наград и количества перемещений.

```
## Сценарий: qlearning.py

from gridworld_env import GridWorldEnv
from agent import Agent
from collections import namedtuple
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)

Transition = namedtuple(
    'Transition', ('state', 'action', 'reward',
                  'next_state', 'done'))

def run_qlearning(agent, env, num_episodes=50):
    history = []
    for episode in range(num_episodes):
        state = env.reset()
        env.render(mode='human')
        final_reward, n_moves = 0.0, 0
        while True:
            action = agent.choose_action(state)
            next_s, reward, done, _ = env.step(action)
            agent._learn(Transition(state, action, reward,
                                   next_s, done))
            env.render(mode='human', done=done)
            state = next_s
            n_moves += 1
            if done:
                break
            final_reward = reward
        history.append((n_moves, final_reward))
        print('Episode %d: Reward %.1f #Moves %d'
              % (episode, final_reward, n_moves))

    return history

def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 10))
    ax = fig.add_subplot(2, 1, 1)
    episodes = np.arange(len(history))
    moves = np.array([h[0] for h in history])
    plt.plot(episodes, moves, lw=4,
             marker='o', markersize=10)
```

```

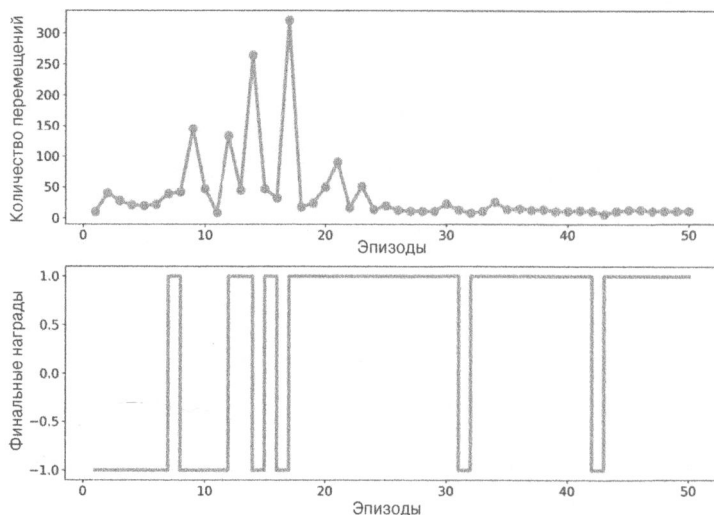
ax.tick_params(axis='both', which='major', labels=15)
plt.xlabel('Эпизоды', size=20)
plt.ylabel('Количество перемещений', size=20)
ax = fig.add_subplot(2, 1, 2)
rewards = np.array([h[1] for h in history])
plt.step(episodes, rewards, lw=4)
ax.tick_params(axis='both', which='major', labels=15)
plt.xlabel('Эпизоды', size=20)
plt.ylabel('Финальные награды', size=20)
plt.savefig('q-learning-history.png', dpi=300)
plt.show()

if __name__ == '__main__':
    env = GridWorldEnv(num_rows=5, num_cols=6)
    agent = Agent(env)
    history = run_qlearning(agent, env)
    env.close()

    plot_learning_history(history)

```

Запуск этого сценария приведет к выполнению программы Q-обучения для 50 эпизодов. Поведение агента будет визуализировано, и вы сможете увидеть, что в начале процесса обучения агент по большей части оказывается в состояниях с ловушками. Но с течением времени он извлекает уроки из своих неудач и в конечном итоге находит “золотое” состояние (скажем, первый раз в эпизоде 7). Количество перемещений и награды агента представлены на рис. 18.8.



**Рис. 18.8.** Количество перемещений и награды агента

Вычерченные графики хронологии обучения, приведенные на рис. 18.8, указывают на то, что после 20 эпизодов агент выяснил кратчайший маршрут к “золотому” состоянию. В результате длины эпизодов после 30-го эпизода оказываются более или менее одинаковыми с небольшими отклонениями из-за  $\epsilon$ -жадной политики.

## Обзор глубокого Q-обучения

В предыдущем коде была представлена реализация популярного алгоритма Q-обучения для примера с миром сетки. Пример включал дискретное пространство состояний с размером 30, где было достаточно хранить Q-ценности в словаре Python.

Тем не менее, мы обязаны отметить, что временами количество состояний может стать очень большим — чуть ли не бесконечно большим. Кроме того, вместо работы с дискретными состояниями мы можем иметь дело с непрерывным пространством состояний. Вдобавок некоторые состояния могут вообще не посещаться во время обучения, что может привести к проблемам при последующем обобщении агента для работы с такими не виденными ранее состояниями.

Чтобы решить указанные проблемы, вместо представления функции ценности в табличном формате вида  $V(S_t)$  или  $Q(S_t, A_t)$  для функции ценности действия мы используем подход с *аппроксимацией функции*. Здесь мы определяем параметрическую функцию  $v_W(x_S)$ , которая способна научиться аппроксимировать настоящую функцию ценности, т.е.  $v_W(x_S) \approx v_\pi(s)$ , где  $x_S$  — набор входных признаков (или “снабженных признаками” состояний).

Когда функция аппроксиматора  $q_W(x_S, a)$  является глубокой нейронной сетью, результирующая модель называется *глубокой Q-сетью* (*deep Q-network* — DQN). Для обучения модели DQN веса обновляются в соответствии с алгоритмом Q-обучения. Пример модели DQN показан на рис. 18.9, где состояния представлены как признаки, переданные первому слою.

А теперь давайте посмотрим, как обучить модель DQN с применением алгоритма *глубокого Q-обучения*. В целом основной подход очень похож на табличный метод Q-обучения. Главное отличие в том, что в данном случае мы располагаем многослойной нейронной сетью, которая рассчитывает ценности действий.

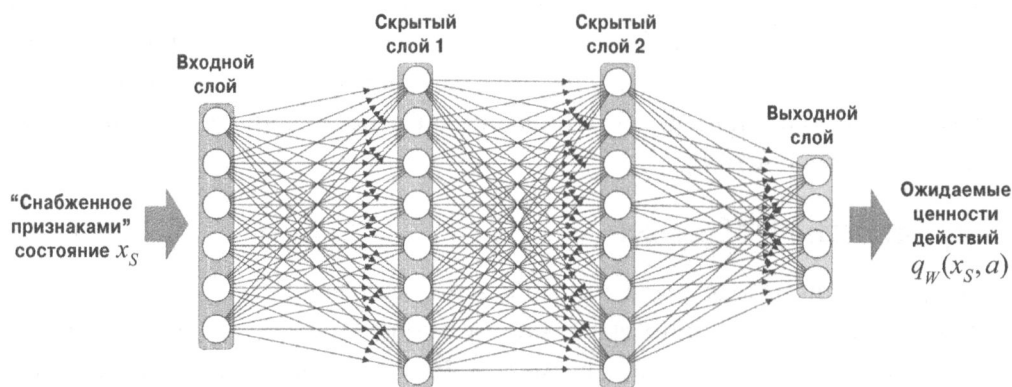


Рис. 18.9. Пример модели DQN

### Обучение модели DQN в соответствии с алгоритмом Q-обучения

В этом разделе мы рассмотрим процедуру обучения модели DQN с использованием алгоритма Q-обучения. Глубокое Q-обучение требует внесения ряда модификаций в ранее реализованный стандартный алгоритм Q-обучения.

Одной такой модификацией является метод `choose_action()` агента, где в предыдущем разделе просто производился доступ к ценностям действий, хранящимся в словаре. Теперь данный метод нужно изменить, чтобы выполнять прямой проход нейросетевой модели для вычисления ценностей действий.

Другие модификации, необходимые для алгоритма глубокого Q-обучения, описаны в последующих двух разделах.

#### Память воспроизведения

С применением табличного метода при Q-обучении мы могли обновлять ценности для индивидуальных пар “состояние–действие”, не влияя на ценности остальных. Однако теперь, когда мы аппроксимируем  $q(s, a)$  с помощью нейросетевой модели, обновление весов для какой-то пары “состояние–действие” вполне вероятно повлияет на выход остальных состояний. При обучении нейронных сетей с использованием стохастического градиентного спуска для задачи с учителем (например, задачи классификации) мы применяем множество эпох, чтобы многократно проходить по обучающим данным вплоть до схождения.

В Q-обучении это неосуществимо, поскольку эпизоды будут изменяться во время обучения и в результате снизится вероятность будущего посещения ряда состояний, которые посещались на ранних стадиях обучения.

Кроме того, еще одна проблема заключается в том, что при обучении нейронной сети мы предполагаем, что обучающие образцы *независимы и идентично распределены*. Тем не менее, образцы, взятые из некоторого эпизода агента, не являются независимыми и идентично распределенными, т.к. они вполне очевидно формируют последовательность переходов.

Чтобы решить упомянутые проблемы, по мере того, как агент взаимодействует со средой и генерирует пятерку перехода  $q_W(x_S, a)$ , мы сохраняем большое (но конечное) количество переходов в буфере памяти, который часто называется *памятью воспроизведения* (*replay memory*). После каждого нового взаимодействия (т.е. агент выбирает действие и запускает его в среде) результирующая пятерка перехода добавляется в эту память.

Для ограничения размера памяти воспроизведения самый старый переход будет удаляться (скажем, если память реализована в виде списка Python, то мы можем использовать метод `pop(0)` с целью удаления первого элемента из списка). Затем из буфера памяти случайным образом выбирается минипакет образцов, который применяется для расчета потери и обновления параметров сети. Процесс проиллюстрирован на рис. 18.10.

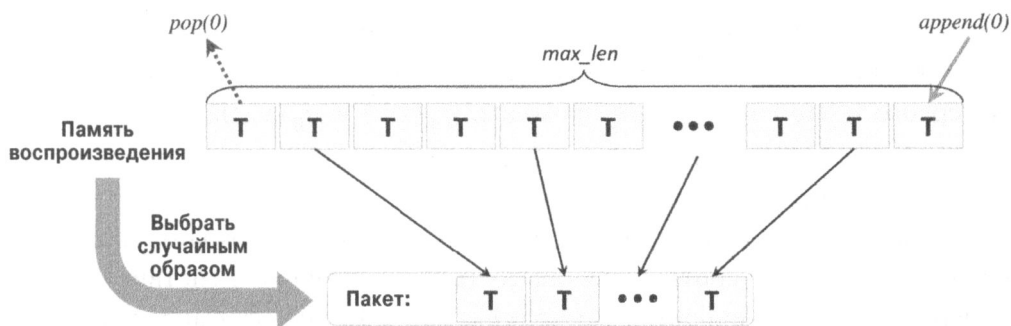


Рис. 18.10. Использование памяти воспроизведения



Совет

### Реализация памяти воспроизведения

Память воспроизведения можно реализовать с использованием списка Python. При добавлении к нему нового элемента нам нужно проверять размер списка и в случае необходимости вызывать метод `pop(0)`.

В качестве альтернативы мы можем применять структуру данных `deque` из Python-библиотеки `collections`, позволяющую указывать дополнительный параметр `max_len`, с помощью которого мы получим ограниченную очередь с двусторонним доступом. Таким образом, когда объект `deque` полон, добавление к нему нового элемента приводит к автоматическому удалению из него старого элемента.

Обратите внимание, что структура данных `deque` более эффективна, чем список Python, т.к. удаление первого элемента из списка посредством `pop(0)` имеет сложность  $O(n)$ , тогда как сложность времени выполнения `deque` составляет  $O(1)$ . Исчерпывающие сведения о реализации структуры данных `deque` доступны в официальной документации по ссылке <https://docs.python.org/3.7/library/collections.html#collections.deque>.

### Определение целевых ценностей для расчета потери

Еще одно обязательное изменение метода табличного Q-обучения касается адаптации правила обновления для узнавания параметров модели DQN. Вспомните, что пятерка транзакции  $T$ , хранящаяся в пакете образцов, содержит  $(x_s, a, r, x_{s'}, done)$ .

Как показано на рис. 18.11, мы выполняем два прямых прохода модели DQN. На первом прямом проходе используются признаки текущего состояния  $(x_s)$ . Затем на втором прямом проходе применяются признаки следующего состояния  $(x_{s'})$ . В результате мы получим ожидаемые ценности действий из первого и второго прямых проходов,  $q_W(x_s, :)$  и  $q_W(x_{s'}, :)$ . (Здесь  $q_W(x_{s'}, :)$  обозначает вектор Q-ценностей для всех действий в  $\hat{A}$ .) Из пятерки перехода нам известно, что агент выбрал действие  $a$ .

Следовательно, согласно алгоритму Q-обучения нам необходимо обновить ценность действия, соответствующего паре “состояние–действие”  $(x_s, a)$ , скалярной целевой ценностью  $r + \gamma \max_{a' \in \hat{A}} q_W(x_{s'}, a')$ . Вместо формирования скалярной целевой ценности мы будем создавать целевой вектор пар “состояние–действие”, который хранит ценности для остальных действий,  $a' \neq a$  (см. рис 18.11).

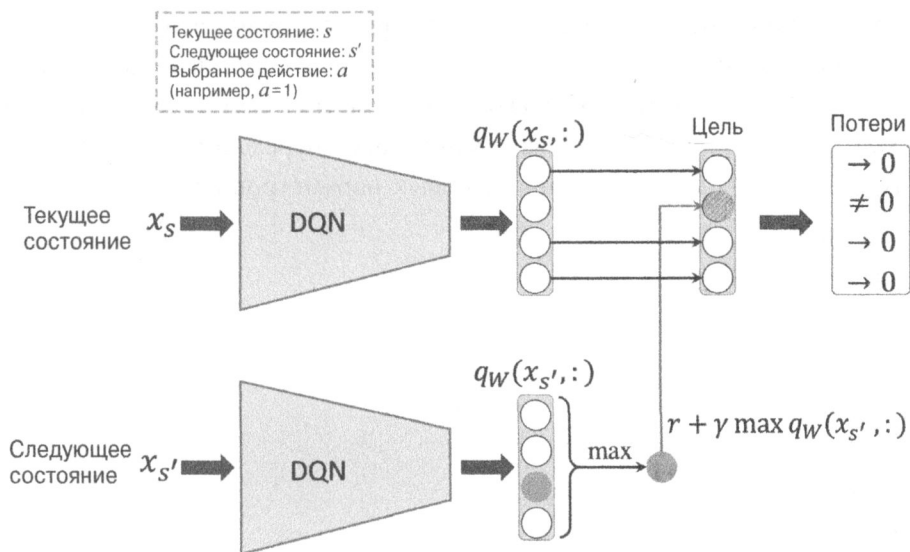


Рис. 18.11. Два прямых прохода модели DQN

Мы трактуем это как задачу регрессии, используя следующие три величины:

- ценности, спрогнозированные в текущий момент,  $q_w(x_s, :)$ ;
- целевой вектор ценностей, описанный выше;
- функция издержек в виде стандартной среднеквадратической ошибки (MSE).

В результате потери будут нулевыми для всех действий кроме  $a$ . Наконец, выполняется обратное распространение рассчитанной потери для обновления параметров сети.

### Реализация алгоритма глубокого Q-обучения

В заключение мы применим все рассмотренные методики для реализации алгоритма глубокого Q-обучения. Мы будем использовать представленную ранее среду CartPole из комплекта инструментов OpenAI Gym. Вспомните, что среда CartPole имеет непрерывное пространство состояний размера 4. В приведенном далее коде мы определяем класс DQNAgent, который строит модель и задает различные гиперпараметры.



В сравнении с предыдущим классом агента, который базировался на табличном Q-обучении, класс `DQNAgent` имеет два дополнительных метода. Метод `remember()` будет добавлять новую пятерку перехода в буфер памяти, а метод `replay()` — создавать мини-пакет переходов и передавать его методу `_learn()` для обновления весовых параметров сети:

```
import gym
import numpy as np
import tensorflow as tf
import random
import matplotlib.pyplot as plt
from collections import namedtuple
from collections import deque

np.random.seed(1)
tf.random.set_seed(1)

Transition = namedtuple(
    'Transition', ('state', 'action', 'reward',
                  'next_state', 'done'))

class DQNAgent:
    def __init__(
        self, env, discount_factor=0.95,
        epsilon_greedy=1.0, epsilon_min=0.01,
        epsilon_decay=0.995, learning_rate=1e-3,
        max_memory_size=2000):
        self.env = env
        self.state_size = env.observation_space.shape[0]
        self.action_size = env.action_space.n

        self.memory = deque(maxlen=max_memory_size)

        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.lr = learning_rate
        self._build_nn_model()

    def _build_nn_model(self, n_layers=3):
        self.model = tf.keras.Sequential()
```

```

## Скрытые слои
for n in range(n_layers-1):
    self.model.add(tf.keras.layers.Dense(
        units=32, activation='relu'))
    self.model.add(tf.keras.layers.Dense(
        units=32, activation='relu'))
## Последний слой
self.model.add(tf.keras.layers.Dense(
    units=self.action_size))
## Построение и компиляция модели
self.model.build(input_shape=(None, self.state_size))
self.model.compile(
    loss='mse',
    optimizer=tf.keras.optimizers.Adam(lr=self.lr))

def remember(self, transition):
    self.memory.append(transition)

def choose_action(self, state):
    if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
    q_values = self.model.predict(state)[0]
    return np.argmax(q_values) # возвращает действие

def _learn(self, batch_samples):
    batch_states, batch_targets = [], []
    for transition in batch_samples:
        s, a, r, next_s, done = transition
        if done:
            target = r
        else:
            target = (r +
                self.gamma * np.amax(
                    self.model.predict(next_s)[0]
                )
            )
        target_all = self.model.predict(s)[0]
        target_all[a] = target
        batch_states.append(s.flatten())
        batch_targets.append(target_all)
    self._adjust_epsilon()
    return self.model.fit(x=np.array(batch_states),
                          y=np.array(batch_targets),
                          epochs=1,
                          verbose=0)

```

```
def _adjust_epsilon(self):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

def replay(self, batch_size):
    samples = random.sample(self.memory, batch_size)
    history = self._learn(samples)
    return history.history['loss'][0]
```

С помощью следующего кода мы обучаем модель для 200 эпизодов и в конце визуализируем хронологию с применением функции `plot_learning_history()`:

```
def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 5))
    ax = fig.add_subplot(1, 1, 1)
    episodes = np.arange(len(history[0]))+1
    plt.plot(episodes, history[0], lw=4,
             marker='o', markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Эпизоды', size=20)
    plt.ylabel('Общие награды', size=20)
    plt.show()

## Общие настройки
EPISODES = 200
batch_size = 32
init_replay_memory_size = 500
if __name__ == '__main__':
    env = gym.make('CartPole-v1')
    agent = DQNAgent(env)
    state = env.reset()
    state = np.reshape(state, [1, agent.state_size])
    ## Заполнение памяти воспроизведения
    for i in range(init_replay_memory_size):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, agent.state_size])
        agent.remember(Transition(state, action, reward,
                                   next_state, done))

    if done: --
        state = env.reset()
        state = np.reshape(state, [1, agent.state_size])
    else:
        state = next_state
```

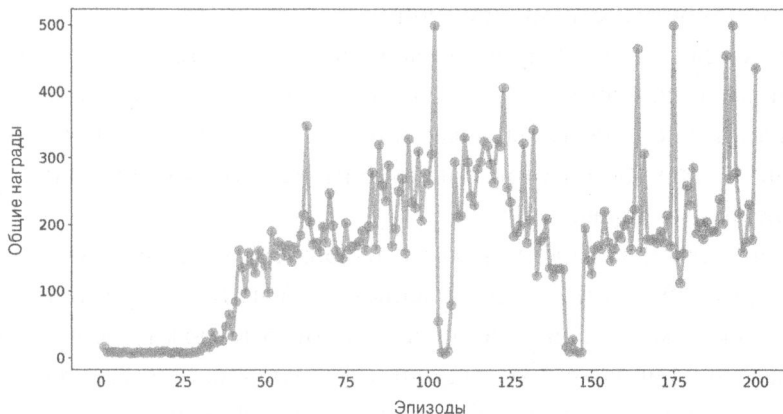
```

total_rewards, losses = [], []
for e in range(EPISODES):
    state = env.reset()
    if e % 10 == 0:
        env.render()
    state = np.reshape(state, [1, agent.state_size])
    for i in range(500):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state,
                                [1, agent.state_size])
        agent.remember(Transition(state, action, reward,
                                next_state, done))

    state = next_state
    if e % 10 == 0:
        env.render()
    if done:
        total_rewards.append(i)
        print('Эпизод: %d/%d, Общая награда: %d'
              % (e, EPISODES, i))
        break
    loss = agent.replay(batch_size)
    losses.append(loss)
plot_learning_history(total_rewards)

```

После обучения агента для 200 эпизодов мы видим, что он действительно научился увеличивать общие награды с течением времени, как иллюстрируется на рис. 18.12.



**Рис. 18.12.** График хронологии обучения

Обратите внимание, что общие награды, полученные в эпизоде, равны количеству времени, на протяжении которого агент способен балансировать дышло. График хронологии обучения на рис. 18.12 показывает, что после примерно 30 эпизодов агент научился балансировать дышло и удерживать его в течение более 200 временных шагов.

## Резюме по главе и по книге

В этой главе мы раскрыли важные концепции обучения с подкреплением, начиная с самых основ и заканчивая его способностью обеспечивать принятие решений в сложных средах.

Вы узнали о взаимодействиях агента со средой и марковских процессах принятия решений (MDP), а также о трех главных подходах к решению задач обучения с подкреплением: динамическом программировании, методе Монте-Карло и методе временных разностей. Мы выяснили, что алгоритм динамического программирования предполагает доступность полного знания динамики среды — допущение, которое обычно не будет верным для большинства реальных задач.

Затем было показано, как алгоритмы на основе методов Монте-Карло и временных разностей учатся, позволяя агенту взаимодействовать со средой и генерировать имитированный опыт. После обсуждения теории мы реализовали алгоритм Q-обучения как подкатегорию вне политики метода временных разностей для решения задачи с миром сетки. В заключение мы раскрыли концепцию аппроксимации функций и в частности глубокое Q-обучение, которое может использоваться для задач с большими или непрерывными пространствами состояний.

Мы надеемся, что вы получили удовольствие от чтения последней главы книги и захватывающего путешествия по МО и ГО, во время которого мы рассмотрели важнейшие темы, касающиеся этой области, так что теперь вы должны быть хорошо подготовлены к применению описанных приемов для решения реальных задач.

Мы начали путешествие с краткого обзора типов задач обучения: обучение с учителем, обучение с подкреплением и обучение без учителя. Затем мы обсудили несколько алгоритмов обучения, которые можно использовать для классификации, начиная с простых однослойных нейронных сетей в главе 2.

Мы продолжили обсуждать алгоритмы классификации в главе 3 и выяснили наиболее важные аспекты конвейера МО в главах 4 и 5.

Вспомните, что даже самый передовой алгоритм ограничен информацией из обучающих данных, на которых он учится. Таким образом, в главе 6 мы представили рекомендуемые приемы для построения и оценки прогнозирующих моделей, которые являются еще одним важным аспектом в приложениях МО.

Если одиночный алгоритм обучения не обеспечивает желаемой эффективности, то временами для вырабатывания прогнозов полезно создавать ансамбли экспертов. Мы исследовали ансамблевое обучение в главе 7.

В главе 8 мы применяли МО для анализа одной из самых популярных и интересных форм данных в современную эпоху, которые доминируют на социальных медиаплощадках в Интернете — текстовых документах.

Поскольку приемы МО не ограничиваются анализом автономных данных, в главе 9 мы показали, как внедрить модель МО в веб-приложение, чтобы сделать ее доступной внешнему миру.

Большей частью мы были сосредоточены на алгоритмах для классификации, которые являются, по всей видимости, наиболее популярным применением МО. Однако на этом путешествие не закончилось. В главе 10 мы исследовали несколько алгоритмов регрессионного анализа для прогнозирования значений непрерывных целевых переменных.

Еще одна подобласть МО — кластерный анализ — может помочь в поиске скрытых структур в данных, даже если обучающие данные не поступают с заведомо правильными ответами. Этому была посвящена глава 11.

Затем мы переместили внимание на, пожалуй, самую захватывающую часть во всей области МО — искусственные нейронные сети. Мы начали с реализации в главе 12 многослойного персептрона с нуля посредством NumPy.

Полезность библиотеки TensorFlow 2 для ГО стала очевидной в главе 13, где мы использовали ее для облегчения процесса построения нейросетевых моделей и работы с объектами Dataset из TensorFlow, а также выяснили, как применять к набору данных шаги предварительной обработки.

В главе 14 мы погрузились глубже в механику библиотеки TensorFlow и обсудили различные аспекты и механизмы TensorFlow, включая переменные, декорирование функций TensorFlow, расчет градиентов и оценщики TensorFlow.

В главе 15 мы занялись сверточными нейронными сетями, которые в данный момент широко используются в компьютерном зрении из-за своей великолепной эффективности при решении задач классификации изображений.

В главе 16 мы исследовали моделирование последовательностей с применением рекуррентных нейронных сетей и раскрыли модель “Преобразователь” — один из самых последних алгоритмов для моделирования seq2seq.

В главе 17 мы показали, как генерировать новые изображения с использованием порождающих состязательных сетей, и попутно также рассмотрели автокодировщики, пакетную нормализацию, транспонированную свертку и порождающие состязательные сети Вассерштейна.

Наконец, в текущей главе мы обсудили совершенно отдельную категорию задач МО и выяснили, как разрабатывать алгоритмы, которые учатся, взаимодействуя со своей средой через процесс получения наград.

Хотя всесторонний анализ глубокого обучения выходит за рамки настоящей книги, мы надеемся, что сумели разжечь у вас достаточный интерес, чтобы вы следили за развитием в данной области.

Если вы планируете заняться машинным обучением или просто хотите знать о текущем состоянии дел в этой области, тогда мы рекомендуем следить за работами ведущих экспертов в сфере машинного обучения, которые перечислены ниже:

- Джеффри Хинтон (<http://www.cs.toronto.edu/~hinton/>)
- Эндрю Ын (<http://www.andrewng.org/>)
- Ян Лекун (<http://yann.lecun.com>)
- Юрген Шмидхубер (<http://people.idsia.ch/~juergen/>)
- Йошуа Бенджи ([http://www.iro.umontreal.ca/~bengioy/yoshua\\_en/](http://www.iro.umontreal.ca/~bengioy/yoshua_en/))

Список далеко не полон!

В заключение отметим, что вы можете найти нас, авторов книги, на следующих веб-сайтах:

<https://sebastianraschka.com>

<http://vahidmirjalili.com>.

Вы всегда можете связаться с нами, если у вас возникли какие-то вопросы по книге или потребность получить подсказки в области машинного обучения.

# Предметный указатель

## A

Accuracy, 260  
Action-value function, 794  
Adaline (ADaptive LInear NEuron), 67;  
77; 83; 459  
Adam (adaptive moment estimation), 644  
Adaptive Boosting (AdaBoost), 302  
Adversarial attack, 572  
Anaconda, 47  
API-интерфейс Keras, 573; 583  
API-интерфейс оценщиков, 601  
API-интерфейс перцептрона, 56  
Artificial intelligence (AI), 456  
Assessor function, 729  
Attention, 716  
AutoGraph, 563  
Autoregressive model, 728  
Average linkage, 438

## B

Backpropagation through time  
(BPTT), 676  
Bagging, 135; 273  
bag-of-words model, 317  
Batch normalization (BatchNorm), 723  
Bias unit, 52  
BLAS (Basic Linear Algebra  
Subprograms), 60  
Boosting, 135; 273

## C

Coefficient of determination, 401  
Comma-separated values (CSV), 146  
Complete linkage, 437  
Conda, 351  
Confusion matrix, 258  
Convolutional neural network (CNN), 607;  
609

Cost function, 41  
CSS (Cascading Style Sheet), 356  
Curse of dimensionality, 142

## D

DB Browser for SQLite, 350  
Deconvolution, 754  
Deep convolutional GAN (DCGAN), 752  
Deep neural network (DNN), 455  
Deep Q-network (DQN), 823  
Density-based spatial clustering  
of applications with noise  
(DBSCAN), 446  
Dimensionality reduction, 37; 173  
Directed acyclic graph (DAG), 558  
Discriminability, 207  
Discriminator (D), 730  
Dropout, 486; 632

## E

Eager execution, 558  
Eigenvalue, 188  
Eigenvector, 188  
Elastic net, 403  
Elbow method, 420  
Embedding, 690  
Ensemble method, 135  
Error, 260  
Error function, 42  
Experience replay, 778  
Exploding, 680  
Exploratory data analysis (EDA), 383

## F

F1 score, 257  
False negative (FN), 258  
False positive (FP), 258  
False positive rate (FPR), 260



FCM (fuzzy C-Means), 423

Feature, 34; 39

Feature extraction, 173

Feature matching, 778

Feature scaling, 162

Feature selection, 173

Feedforward neural network, 461

Flask, 350; 351; 360

Forget gate, 682

Fully connected (FC), 539

Fuzzifier, 429

## G

Gate, 682

Gated recurrent unit (GRU), 670

Generalized policy iteration (GPI), 799

Generative adversarial network (GAN), 722;  
723; 724

Generator (G), 729

Global interpreter lock (GIL), 502

Google Colab, 734

Gradient boosting, 311

Gradient clipping, 666

Gradient descent, 69

Graphviz, 132

Grid search, 252

Gym environments

working with, 809

## H

Hermitian, 191

Holdout cross-validation, 239

HTML, 352

## I

Imputing, 145

Information gain (IG), 125

Input gate, 682

Internal covariance shift, 758

Internet Movie Database (IMDb), 313

## J

Jensen-Shannon (JS), 767

Jinja2, 356

Joblib, 345

Jupyter Notebook, 734

## K

Kantorovich-Rubinstein duality, 770

Kernel function, 121

Kernelized, 118

Kernel principal component analysis  
(KPCA), 185

Kernel SVM, 118

Kernel trick, 121; 213

Key, 719

K-Means, 419

K-nearest neighbor (KNN), 139; 163

Kullback-Leibler (KL), 767

## L

LAPACK (Linear Algebra Package), 60

Layer normalization, 721

Leaky ReLU, 739

Least absolute shrinkage and selection  
operator (LASSO), 403

Leave-one-out cross-validation  
(LOOCV), 243

Linear discriminant analysis (LDA), 185;  
200

Linkage matrix, 440

Logistic regression, 94

Long short-term memory (LSTM), 665;  
670; 681

Loss function, 41

## M

Majority voting, 274

Markov decision process (MDP), 787

Matplotlib, 47

Max-pooling, 624

McCullock-Pitts (MCP), 50  
 Mean imputation, 149  
 Mean-pooling, 624  
 Mean squared error (MSE), 401; 534  
 Median absolute deviation, 397  
 Miles per gallon (MPG), 591  
 Mini-batch discrimination, 777  
 Min-max scaling, 163  
 MNIST (Mixed National Institute of Standards and Technology), 468  
 Mode, 274  
 Mode collapse, 777  
 Monte Carlo, 802  
 Multi-head attention (MHA), 720  
 Multilayer perceptron (MLP), 461

## N

National Institute of Standards and Technology (NIST), 469  
 Natural language processing (NLP), 313; 667  
 n-gram, 319  
 Normalization, 163  
 Normalizing flow model, 728

## O

Odds, 95  
 One-hot encoding, 156  
 One-versus-all (OvA), 61  
 One-versus-rest (OvR), 89  
 Online learning, 79  
 OpenAI Gym, 808  
 Opinion mining, 314  
 Ordinary least squares (OLS), 389  
 Out-of-core learning, 314  
 Output gate, 682  
 Overfitting, 91

## P

Pandas, 47; 152  
 Pickle, 344; 346

Pip, 351  
 Porter stemmer, 325  
 Precision, 257  
 Principal component analysis (PCA), 185; 236  
 Python Progress Indicator (PyPrind), 315

## Q

Query, 719

## R

Radial Basis Function (RBF), 121  
 Random forest, 135  
 Randomized search, 254  
 RANSAC (RANDOM SAMple Consensus), 396  
 Raw term frequency, 319  
 RBF, 215  
 Recall, 257  
 Receiver operating characteristic (ROC), 263  
 Rectified linear unit (ReLU), 551  
 Recurrent edge, 672  
 Recurrent neural network (RNN), 663; 665  
 Recursive backward elimination, 180  
 Reinforcement learning (RL), 30; 781  
 Replay memory, 825  
 Residual connection, 721  
 Ridge regression, 403  
 ROC area under the curve (ROC AUC), 263

## S

SARSA (state-action-reward-state-action), 797  
 Scikit-learn, 254; 308; 321; 393  
 Self-attention mechanism, 666; 716  
 Sentiment analysis, 313  
 Sequential backward selection (SBS), 174  
 SGD (стохастический градиентный спуск), 569

Silhouette coefficient, 431  
Silhouette plot, 420  
SIMD (Single Instruction, Multiple Data), 60  
Similarity function, 122  
Single linkage, 437  
Slack variable, 115  
Softmax, 638  
SQLite, 348  
Stacking, 295  
Standardization, 163  
State-value function, 794  
Statsmodels, 395  
Stop-word removal, 327  
Sum of Squared Errors (SSE), 68; 422  
Supervised learning, 30  
Support Vector Machine (SVM), 113

## T

Target, 41  
Target variable, 34  
Temporal difference (TD), 789  
TensorFlow, 501; 504; 556; 590  
TensorFlow v1.x, 559  
TensorFlow v2, 561  
Term frequency-inverse document frequency (tf-idf), 320  
Training, 41  
Training sample, 41  
Transformer, 150; 666; 716  
True negative (TN), 258  
True positive rate (TPR), 260  
True positive (TP), 258  
Truncated backpropagation through time (T-BPTT), 665

## U

Underfitting, 109  
Unit step function, 51  
Unsupervised learning, 30  
Upsampling, 754

## V

Value, 719  
Value function, 796  
Vanishing, 680  
Variance explained ratio, 192  
Variational autoencoder (VAE), 723

## W

Ward's linkage, 438  
Wasserstein distance, 723  
Wasserstein GAN (WGAN), 753  
Word stemming, 325

## A

Автокодировщик, 725  
    вариационный (VAE), 723; 728  
Агент, 34; 783  
Агрегирование  
    бутстрэп-, 296  
Алгоритм  
    AdaBoost, 304; 308; 311  
    Adaline, 77; 83; 459  
    K-Means, 420  
    K-Means++, 425  
    K-Medoids, 423  
    KNN, 139  
    k ближайших соседей (KNN), 163  
    Q-обучения, 819; 824  
        глубокого, 827  
    RANSAC, 396  
    Бройдена-Флетчера-Гольдфарба-Шанно, 107  
    временных разностей, 805  
    глубокого Q-обучения, 782  
    исчерпывающего поиска, 174  
    минимальной фильтрации  
        Винограда, 623  
    обучения персептрона, 56  
    обучения с подкреплением, 797  
    оптимизации на основе градиентного спуска, 162  
    поглощающий (“жадный”), 174

последовательного выбора признаков, 173  
 стемминга Портера, 325  
 случайного леса, 135  
 улучшения политики, 800

## Анализ

главных компонентов (PCA), 236  
 линейный дискриминантный, 200  
 мнений  
   глубинный, 314  
 регрессионный, 33; 377  
 смысловой, 313

## Ансамбль

деревьев принятия решений, 135  
 построение ансамблей с использованием стекинга, 295

## Архитектура

“Преобразователь”, 716

## Атака

сопоставительная, 572

## Б

### База данных

SQLite, 348

### Библиотека

joblib, 345  
 LIBLINEAR, 117  
 Matplotlib, 47  
 pandas, 47; 381  
 scikit-learn, 87; 155; 237; 321; 393  
 TensorFlow, 504; 556  
   создание тензоров в TensorFlow, 507  
 tensorflow\_datasets, 524

### Биномиальный коэффициент, 276

### Блокировка интерпретатора

глобальная, 502

### Бустинг, 135; 273; 303

адаптивный, 302  
 градиентный, 311

### Бутстрэп

-агрегирование, 296  
 -выборка, 135

### Бэггинг, 135; 273; 295; 296

## В

### Веб-приложение

разработка с помощью Flask, 350

### Веб-фреймворк Flask, 360

### Вектор

опорный, 113

### Векторизация, 60

### Вероятность

смены состояния, 789  
 спрогнозированная, 661

### Взвешивание, 279

### Визуализация

марковского процесса, 790

### Внимание

многоголовое (МНА), 720

### Выбросы, 433

### Выражение

регулярное, 323

### Вычислительная мощность

увеличение вычислительной  
 мощности с помощью декораторов  
 функций, 562

## Г

### Генератор, 729

график потерь генератора, 750

### Гиперболический тангенс, 548

### Гиперпараметр, 239; 252

### Глубокое обучение (ГО), 455; 462

### Голосование

мажоритарное, 274  
 относительным большинством  
 голосов, 274

### Градиент

взрывной рост градиентов, 680  
 отсечение градиентов, 666  
 проблема исчезновения градиентов, 680  
 расчет градиентов по отношению  
   к необучаемым тензорам, 571  
 расчет градиентов посредством автоматического дифференцирования, 569

Градиентный спуск (GD), 389  
 пакетный, 71; 78  
 стохастический, 78; 389; 460; 569

Граф, 601

ациклический  
 ориентированный (DAG), 558  
 вычислительный, 558  
 проблемы с графами, 601

График

искажения для разного числа  
 кластеров, 431  
 для простого двумерного набора  
 данных, 421  
 коэффициента “различимости”, 207  
 объясненной дисперсии, 193  
 остатков, 399  
 рассеяния  
 с обучающими образцами, 532; 578  
 проверочных образцов, 578  
 силуэтов, 420

## Д

Данные

CelebA, 648  
 анализ данных, 383  
 дополнение данных, 649  
 загрузка и предварительная обработка  
 данных, 639  
 набор данных MNIST, 475  
 определение обучающего набора  
 данных, 742  
 понижение размерности для сжатия  
 данных, 37  
 последовательные, 666

Двойственность Канторовича-Рубин-  
 штейна, 770

Декоратор функций, 562

Дендрограмма, 437; 443

прикрепление дендрограмм к тепловой  
 карте, 443

Дерево

принятия решений, 124; 162; 410  
 каталогов, 362

Дивергенция Йенсена-Шеннона, 767

Дискриминант Фишера  
 линейный, 201

Дискриминатор, 730

Дисперсия, 110

внутриузловая, 412  
 график объясненной дисперсии, 193  
 коэффициент объясненной дисперсии, 192  
 понижение дисперсии, 412

Дифференцирование

автоматическое, 492; 571

## Е

Евклидово расстояние

квадратичное, 422

## З

Загрязненность

Джини, 126

мера загрязненности, 126

Задача

классификации, 31  
 XOR, 577  
 двоичной, 33  
 многоклассовой, 33  
 отсроченная, 670  
 продолжающаяся, 791  
 эпизодическая, 791

Зазор, 114

Запрос (query), 719

Значения, 719

отделенные друг от друга запятыми  
 (CSV), 146

## И

Изображения

трансформация изображений, 649

Индикатор выполнения на Python, 315

Инициализация Ксавье (Глоро), 567

Инструмент

AutoGraph, 563

OpenAI Gym, 808

- Интеллект  
искусственный, 30; 456
- Интерпретатор  
блокировка интерпретатора, 502
- Исключающее ИЛИ (XOR), 555
- Исключение  
рекурсивное обратное, 180
- Итерация  
политики, 800  
ценности, 801
- К**
- Канал, 627
- Карта  
признаков, 611  
тепловая, 443; 444
- Квадратичное евклидово  
расстояние, 422
- Класс  
Model, 584  
RANSACRegressor, 396  
Sequential, 573  
дисбаланс классов, 267  
метка класса, 54  
настоящая, 54  
спрогнозированная, 54  
не сепарабельный  
линейно, 55  
сепарабельный  
линейно, 55
- Классификатор  
ансамблевый, 289  
на основе k ближайших соседей, 141
- Классификация, 32; 106  
без учителя, 37  
двоичная, 33  
многозначная (multilabel), 106  
многоклассовая, 33  
с мягким зазором, 115
- Кластер, 37  
инерция кластера, 422  
отделение кластера, 431  
связность кластера, 431
- Кластеризация, 37; 419
- К-Means  
с использованием scikit-learn, 420  
жесткая, 427  
иерархическая, 420; 436  
мягкая, 427  
на основе графов, 451  
на основе плотности, 420  
на основе прототипов, 420  
нечеткая, 427  
плохая, 434  
применение агломеративной иерархи-  
ческой кластеризации с помощью  
scikit-learn, 445  
спектральная, 451  
хорошая, 434
- Ключ, 719
- Кодирование  
меток классов, 154  
унитарное (one-hot encoding), 156
- Коллапс мод, 777
- Коллинеарность, 206
- Команда  
conda, 351  
pip, 351
- Конвейер  
из библиотеки scikit-learn, 237
- Корреляция  
взаимная, 616
- Коэффициент  
биномиальный, 276  
корреляции смешанного момента  
Пирсона, 386  
детерминации, 401  
дисконтирования, 792  
нечеткости, 429  
объясненной дисперсии, 192
- Кривые  
обучения, 245; 542; 646  
проверки, 245  
решение проблем недообучения и  
переобучения с помощью кривых  
проверки, 250

**Л****Лес**

случайный, 162; 180; 410

Липшицева непрерывность, 770

**М**

Мажоритарное голосование, 274; 285

**Марковский процесс**

визуализация марковского процесса, 790

принятия решений (MDP), 786

**Массив**

NumPy, 345

**Масштабирование**

по минимуму, 163

признаков, 162

**Матрица**

весов, 674

графиков рассеяния, 383; 384

неточностей, 258

связей, 440

симметричная, 439

трансформации, 208

эрмитова (Hermitian), 191

Машинное обучение, См. Обучение, 29

без учителя, 36

с подкреплением, 34; 781

с учителем, 30

Медоид, 420

**Мера**

F1, 257; 261

tf-idf, 320

загрязненности, 126

**Метка**

достоверная, 661

класса, 54

настоящая, 54

спрогнозированная, 54

корректная, 783

положительная, 262

**Метод**

K-Means (K-средних), 419

кластеризация K-Means

с использованием scikit-learn, 420

k ближайших соседей, 139

K-медоидов, 423

ансамблевый, 135; 273

без модели, 788

временных разностей, 804

локтя, 420

для нахождения оптимального  
количества кластеров, 430

Монте-Карло, 801

без модели, 789

мягких K-средних, 427

наименьших квадратов, 389

на основе модели, 788

нечетких C-средних, 423

одиночной связи, 437

“один против остальных” (OvR), 89

опорных векторов (SVM), 113; 416

перекрестной проверки

с удержанием, 239

полной связи, 437

связи Уорда, 438

средней связи, 438

ядерный, 120

**Методика**

один против всех (OvA), 61

**Механизм**

самовнимания, 717

шаблонизации Jinja2, 356

Микрофреймворк Flask, 351

Мода, 274

Моделирование нелинейных связей, 407

**Модель**

DQN, 824

GAN, 730; 744

авторегрессионная, 728

выбор прогнозирующей модели, 44

мешка слов, 317

отбор модели, 239

оценка эффективности линейных

регрессионных моделей, 399

персептрона на наборе данных Iris, 60

порождающая, 729

потоковая  
 для синтеза новых данных, 727  
 нормализующая, 728  
 обучение в среде Google Colab, 734  
 “Преобразователь”, 716  
 реализация моделей на основе класса  
 Model библиотеки Keras, 584  
 суммирования, 317  
 униграммная, 319  
 Модуль pickle для Python, 344; 346

## Н

Набор данных  
 Housing, 383  
 MNIST, 475  
 проверочный, 177  
 Награда, 783; 795  
 немедленная, 791  
 поздняя, 791  
 сигнал награды, 785  
 без модели и на основе модели, 788  
 с помощью метода Монте-Карло, 801  
 Насыщение, 732  
 Недообучение (underfitting), 109; 250  
 Нейрон  
 адаптивный линейный (Adaline), 67;  
 389; 459  
 биологический, 50  
 искусственный, 50  
 Мак-Каллока-Питтса, 50; 456  
 Нейронная сеть  
 искусственная  
 глубокая, 461  
 обучение искусственной нейронной  
 сети, 488  
 многослойная  
 прямого распространения, 461  
 сходимость в нейронных сетях, 498  
 Нормализация, 163  
 пакетная, 471; 756  
 по слою, 721

## О

Образец  
 обучающий, 41  
 Обучение (training), 41  
 внешнее, 314  
 глубокое, 455  
 динамическое, 79  
 искусственной нейронной сети, 488  
 мини-пакетное, 79  
 на основе многообразий, 230  
 на основе образцов, 139  
 скорость обучения, 459  
 с подкреплением, 781  
 с учителем, 377  
 через взаимодействие, 783  
 Объединение  
 непересекающееся, 625  
 Объект  
 Variable, 565  
 Оператор  
 размытия, 429  
 Операция  
 объединения, 624  
 по максимуму, 624  
 по среднему, 624  
 округления в меньшую сторону, 618  
 транспонированной свертки, 754  
 Оптимизатор Adam, 644  
 Остаточная связь, 721  
 Отдача, 791; 795; 796  
 Отклонение  
 медианное абсолютное, 397  
 Отключение (dropout), 486  
 Ошибка, 260  
 классификации, 90; 126  
 производная квадратичной ошибки, 70  
 среднеквадратическая (MSE), 401; 534  
 сумма квадратичных ошибок (SSE), 68;  
 389; 422



## П

Пакетная нормализация, 471; 723; 756

Память

воспроизведения, 825

долгая краткосрочная (LSTM), 670; 681

ячейка памяти, 681

Перевес (odds), 95

Перекрестная проверка

по k блокам, 239; 240

по одному, 243

с удержанием, 238

Переменная

создание переменных

позднее, 534

фиктивная, 115

целевая, 34

Переобучение (overfitting), 91; 109; 250

Персептрон, 53

API-интерфейс персептрона, 56

алгоритм обучения персептрона, 56

многослойный, 461; 590

построение, 538

обучение модели персептрона, 60

Розенблатта, 53

сходимость персептронов, 67

Подкаталог, 362

Позднее создание переменных, 540

Поиск

рандомизированный, 254

решетчатый, 252; 253

Политика, 793

ε-жадная, 803

обобщенная итерация политики

(GPI), 799; 800

оптимальная, 794

Полнота, 257; 260

Понижение размерности, 37; 173; 726

без учителя с помощью анализа

главных компонентов, 186

Последовательности, 666

Правдоподобие (likelihood), 99

Правило Видроу-Хоффа, 67

Правильность, 260

Признак (feature), 34; 39; 41

выбор признаков, 173

выделение признаков, 173; 185

согласование признаков, 778

Приложение

DeerFace, 457

DeepSpeech от Baidu, 457

веб-, 350

Прирост информации (IG), 411

Проблема

недообучения, 250

переобучения, 250

Программа

Graphviz, 132

Программирование

динамическое, 787; 798

с использованием уравнения

Беллмана, 796

Производная функции активации, 495

Проклятие размерности, 142; 450; 691

## Р

Различимость (discriminability), 207

график коэффициента

“различимости”, 207

Распространение

обратное, 497

прямое, 494

Расстояние

Вассерштейна, 723

евклидово

квадратичное, 422

Кульбака–Лейблера, 767

Ребро

рекуррентное, 672

Регрессия, 31

гребневая, 403

к среднему, 34

линейная, 35

логистическая, 94

методом наименьшего абсолютного сокращения и выбора (LASSO), 403  
 множественная линейная, 379  
 на основе случайного леса, 413  
 простая линейная, 378  
 регрессионный анализ, 33  
 с помощью метода опорных векторов, 416  
 Регуляризация, 110  
   L1 и L2, 166  
 Регулярные выражения, 323  
 Резервная копия, 374  
 Рекуррентность  
   выходного слоя, 676  
   скрытого слоя, 676  
 Рекурсия, 792

## С

Свертка, 613  
   дискретная, 613  
   обращение свертки, 754  
   операция транспонированной свертки, 754  
   транспонированная, 753; 754  
 Сериализация массивов NumPy  
   с помощью библиотеки joblib, 345  
 Сеть  
   DCGAN, 752  
   GAN, 724  
   Q-сетью  
     глубокая, 823  
   Вассерштейна (WGAN), 753  
   генератора, 729; 737  
   дискриминатора, 737  
   емкость сети, 631  
   недообучение сети, 631  
   нейронная (CNN)  
     глубокая, 455  
     регуляризация нейронной сети  
       с помощью отключения, 631  
     рекуррентная (RNN), 663; 665; 670; 672  
     сверточная, 607; 609; 641  
   отключение элементов сети, 632  
   переобучение сети, 631

состязательная  
   порождающая (GAN), 723  
   с утечкой, 739  
   эластичная, 403  
 Сигнал, 613  
   награды, 785  
 Силуэт, 431  
   коэффициент силуэта, 431  
 Скорость обучения, 54; 459  
 Слой  
   выходной  
     многопеременный (softmax), 638  
   линейный, 539  
   объединения по глобальному  
     среднему, 657  
   полносвязный, 539  
 Смещение, 379  
 Спуск  
   градиентный, 69  
 Среда, 785  
   Google Colab, 734  
 Стандартизация, 163  
 Стекинг, 295  
 Стеммер Портера, 325  
 Стемминг слов, 325  
 Страйд, 615  
 Сумма квадратичных ошибок (SSE), 389;  
   422

## Т

Тензор, 507  
   необучаемый, 571  
   объединение двух тензоров в общий  
     набор данных, 515  
 Теорема  
   об отсутствии бесплатных завтраков, 44  
 Тепловая карта, 443; 444  
 Терм  
   частота терма, 320  
 Точка  
   границная (border point), 446  
   шумовая (noise point), 446

ядерная (core point), 446  
 Точность, 257; 260  
 Трансформация изображений, 649  
 Трансформирование слов в векторы признаков, 318

Трюк  
 ядерный, 121; 212

## У

Удаление стоп-слов, 327  
 Уравнение Беллмана, 786; 796  
 Ученик  
 ленивый, 139  
 слабый, 303

## Ф

Файл  
 CSS, 356  
 Фреймворк Flask, 353  
 Функция активации  
 производная функции активации, 495  
 аппроксимация функции, 823  
 близости, 122  
 декораторы функций, 562  
 единичная ступенчатая, 51  
 издержек, 41  
 инфимума,  $\inf(S)$ , 766  
 логарифмического правдоподобия (log-likelihood), 99  
 логистическая, 545  
 логит-функция (logit), 95  
 наград, 783  
 оценщика, 729  
 ошибки, 42  
 потерь (loss function), 41; 576; 635  
 радиальная базисная (RBF), 121; 215  
 сигмоидальная, 95; 544

супремума,  $\sup(S)$ , 766  
 целевая, 68  
 ценности, 794; 796  
 действия, 794  
 ядерная, 121; 212

## Ц

Цель (target), 41  
 Ценность  
 высокая, 795  
 Центроид, 420

## Ч

Частота терма, 320  
 сырая, 319  
 Член смещения, 52

## Ш

Шлюз (gate), 682  
 входной, 682  
 выходной, 682  
 забывания, 682  
 Штрафы, 783

## Э

Энергичное выполнение, 558  
 Энтропия, 126  
 перекрестная, 768  
 двоичная, 635  
 категориальная, 635  
 Эпизод, 783; 791  
 Эпоха, 55  
 Эффект  
 отсроченный, 784

## Я

Ядерный трюк, 213  
 Ядро  
 гауссово, 121

# ПРИКЛАДНОЕ МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ SCIKIT-LEARN, KERAS И TENSORFLOW 2-е ПОЛНОЦВЕТНОЕ ИЗДАНИЕ

*Орельен Жерон*



[www.williamspublishing.com](http://www.williamspublishing.com)

Благодаря серии выдающихся достижений глубокое обучение значительно усилило всю область машинного обучения. В наше время даже программисты, почти ничего не знающие об этой технологии, могут использовать простые и эффективные инструменты для реализации программ, которые способны обучаться на основе данных.

За счет применения конкретных примеров, минимума теории и фреймворков Python производственного уровня обновленное издание этой ставшей бестселлером книги поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

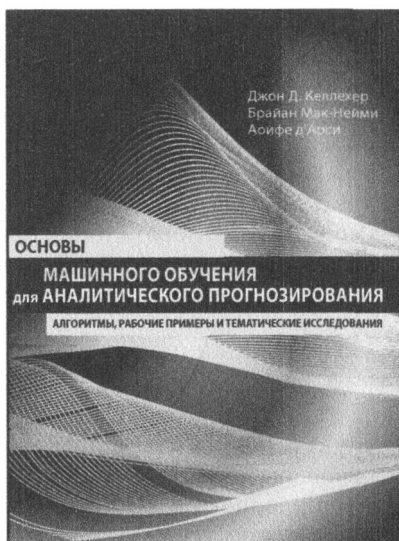
ISBN 978-5-907203-33-4

в продаже

# ОСНОВЫ МАШИННОГО ОБУЧЕНИЯ ДЛЯ АНАЛИТИЧЕСКОГО ПРОГНОЗИРОВАНИЯ

АЛГОРИТМЫ, РАБОЧИЕ ПРИМЕРЫ И ТЕМАТИЧЕСКИЕ  
ИССЛЕДОВАНИЯ

**Джон Д. Келлехер  
Брайан Мак-Нейми  
Аоифе д'Арси**



[www.williamspublishing.com](http://www.williamspublishing.com)

Книга представляет собой учебник по машинному обучению с акцентом на коммерческие приложения. Она предлагает подробное описание наиболее важных подходов к машинному обучению, используемых в интеллектуальном анализе данных, охватывающих как теоретические концепции, так и практические приложения. Формальный математический материал дополняется пояснительными примерами, а примеры исследований иллюстрируют применение этих моделей в более широком контексте бизнеса. В книге рассмотрены информационное обучение, обучение на основе сходства, вероятностное обучение и обучение на основе ошибок. Описанию каждого из этих подходов предшествует объяснение основополагающей концепции, за которой следуют математические модели и алгоритмы, иллюстрированные подробными рабочими примерами.

ISBN 978-5-6040044-9-4

в продаже

# Python и машинное обучение

## 3-е издание

Книга является всеобъемлющим руководством по машинному и глубокому обучению с использованием языка Python. Она служит как пошаговым учебным пособием, так и справочником, к которому вы постоянно будете обращаться в ходе построения систем машинного обучения.

Книга наполнена четкими пояснениями, визуальными представлениями, работающими примерами и детально раскрывает все важные методики машинного обучения. В то время как некоторые книги учат вас следовать инструкциям, Рашка и Мирджалили излагают принципы, лежащие в основе машинного обучения, что позволит вам самостоятельно строить модели и приложения.

Третье издание книги обновлено с целью учета версии библиотеки TensorFlow 2 и последних добавлений в scikit-learn. Оно расширено для охвата двух самых современных методик машинного обучения: обучения с подкреплением и порождающих состязательных сетей.

Эта книга — ваш попутчик в машинном обучении с применением Python, будь вы разработчиком приложений на языке Python, не знакомым с машинным обучением, или разработчиком, желающим углубить свои знания в современных областях.

**Категория:** машинное и глубокое обучение с помощью Python  
**Уровень:** для пользователей средней и высокой квалификации



<http://www.williamspublishing.com>

**Packt**

[www.packtpub.com](http://www.packtpub.com)

### Основные темы книги

- Фреймворки, модели и методики, которые позволяют машинам “учиться” на основе данных
- Использование scikit-learn для машинного обучения и TensorFlow для глубокого обучения
- Применение машинного обучения для классификации изображений, смыслового анализа, создания интеллектуальных веб-приложений и многого другого
- Построение и обучение нейронных сетей, порождающих состязательных сетей и других моделей
- Реализация веб-приложений с искусственным интеллектом
- Выполнение очистки и подготовки данных для машинного обучения
- Классификация изображений с использованием глубоких сверточных нейронных сетей
- Рекомендуемые приемы для оценки и настройки моделей
- Прогнозирование непрерывных целевых результатов с использованием регрессионного анализа
- Обнаружение скрытых шаблонов и структуры в данных с помощью кластеризации
- Углубление в текстовые данные и данные социальных сетей с применением смыслового анализа

Все иллюстрации к книге в цветном варианте доступны по адресу  
<http://go.dialektika.com/pythonml>



ISBN 978-5-907203-57-0



9 785907 203570