

Андреас Мюллер, Сара Гвидо

# Введение в машинное обучение с помощью Python

*Руководство для специалистов  
по работе с данными*



Москва  
2016-2017

# СОДЕРЖАНИЕ

<b>ПРЕДИСЛОВИЕ .....</b>	<b>7</b>
Кому стоит прочитать эту книгу .....	7
Почему мы написали эту книгу .....	8
Структура книги .....	8
Онлайн-ресурсы .....	9
Типографские соглашения .....	9
Использование примеров программного кода .....	10
Благодарности .....	11
<b>ГЛАВА 1. ВВЕДЕНИЕ .....</b>	<b>13</b>
Зачем нужно использовать машинное обучение? .....	13
Задачи, которые можно решить с помощью машинного обучения .....	14
Постановка задач и знакомство с данными .....	17
Почему нужно использовать Python? .....	18
scikit-learn .....	18
Установка scikit-learn .....	19
Основные библиотеки и инструменты .....	20
Jupyter Notebook .....	20
NumPy .....	20
SciPy .....	21
matplotlib .....	22
pandas .....	23
mglearn .....	24
Сравнение Python 2 и Python 3 .....	26
Версии библиотек, используемые в этой книге .....	26
Первый пример: классификация сортов ириса .....	27
Загружаем данные .....	28
Метрики эффективности: обучающий и тестовый наборы .....	31
Сперва посмотрите на Ваши данные .....	32
Построение вашей первой модели: метод $k$ ближайших соседей .....	34
Получение прогнозов .....	36
Оценка качества модели .....	37
Выводы и перспективы .....	37
<b>ГЛАВА 2. МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ С УЧИТЕЛЕМ .....</b>	<b>40</b>

Классификация и регрессия.....	40
Обобщающая способность, переобучение и недообучение .....	42
Взаимосвязь между сложностью модели и размером набора данных.....	45
Алгоритмы машинного обучения с учителем.....	45
Некоторые наборы данных .....	46
Метод $k$ ближайших соседей .....	50
Линейные модели .....	59
Наивные байесовские классификаторы .....	83
Деревья решений .....	85
Ансамбли деревьев решений.....	100
Ядерный метод опорных векторов.....	110
Нейронные сети (глубокое обучение) .....	122
Оценки неопределенности для классификаторов .....	136
Решающая функция.....	137
Прогнозирование вероятностей .....	140
Неопределенность в мультиклассовой классификации .....	142
Выводы и перспективы .....	144
<b>ГЛАВА 3. МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ БЕЗ УЧИТЕЛЯ И</b>	
<b>ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ДАННЫХ .....</b>	<b>147</b>
Типы машинного обучения без учителя.....	147
Проблемы машинного обучения без учителя .....	148
Предварительная обработка и масштабирование.....	149
Различные виды предварительной обработки .....	149
Применение преобразований данных .....	150
Масштабирование обучающего и тестового наборов одинаковым образом .....	152
Влияние предварительной обработки на машинное обучение с учителем.....	155
Снижение размерности, выделение признаков и множественное обучение .....	156
Анализ главных компонент (PCA).....	157
Факторизация неотрицательных матриц (NMF) .....	173
Множественное обучение с помощью алгоритма t-SNE .....	181
Кластеризация .....	185
Кластеризация $k$ -средних .....	185
Агломеративная кластеризация.....	198
DBSCAN .....	204
Сравнение и оценка качества алгоритмов кластеризации.....	208

Выводы по методам кластеризации.....	225
Выводы и перспективы .....	225
<b>ГЛАВА 4. ТИПЫ ДАННЫХ И КОНСТРУИРОВАНИЕ ПРИЗНАКОВ.....</b>	<b>228</b>
Категориальные переменные .....	229
Прямое кодирование (дамми-переменные) .....	230
Числа можно закодировать в виде категорий.....	235
Биннинг, дискретизация, линейные модели и деревья .....	237
Взаимодействия и полиномы.....	241
Одномерные нелинейные преобразования.....	248
Автоматический отбор признаков .....	252
Одномерные статистики .....	253
Отбор признаков на основе модели.....	255
Итеративный отбор признаков .....	257
Применение экспертных знаний .....	259
Выводы и перспективы .....	267
<b>ГЛАВА 5. ОЦЕНКА И УЛУЧШЕНИЕ КАЧЕСТВА МОДЕЛИ .....</b>	<b>268</b>
Перекрестная проверка .....	269
Перекрестная проверка в scikit-learn.....	270
Преимущества перекрестной проверки.....	271
Стратифицированная k-блочная перекрестная проверка и другие стратегии .....	272
Решетчатый поиск .....	278
Простой решетчатый поиск .....	279
Опасность переобучения параметров и проверочный набор данных.....	279
Решетчатый поиск с перекрестной проверкой.....	281
Метрики качества модели и их вычисление.....	295
Помните о конечной цели .....	295
Метрики для бинарной классификации.....	296
Метрики для мультиклассовой классификации .....	320
Метрики регрессии .....	323
Использование метрик оценки для отбора модели .....	323
Выводы и перспективы .....	325
<b>ГЛАВА 6. ОБЪЕДИНЕНИЕ АЛГОРИТМОВ В ЦЕПОЧКИ И КОНВЕЙЕРЫ .....</b>	<b>328</b>
Отбор параметров с использованием предварительной обработки .....	329
Построение конвейеров .....	331
Использование конвейера, помещенного в объект GridSearchCV .....	332



Общий интерфейс конвейера .....	335
Удобный способ построения конвейеров с помощью функции <code>make_pipeline</code> .....	336
Работа с атрибутами этапов .....	337
Работа с атрибутами конвейера, помещенного в объект <code>GridSearchCV</code> .....	338
Находим оптимальные параметры этапов конвейера с помощью решетчатого поиска .....	340
Выбор оптимальной модели с помощью решетчатого поиска.....	342
Выводы и перспективы .....	343
<b>ГЛАВА 7. РАБОТА С ТЕКСТОВЫМИ ДАННЫМИ .....</b>	<b>345</b>
Строковые типы данных .....	345
Пример применения: анализ тональности киноотзывов .....	347
Представление текстовых данных в виде «мешка слов» .....	350
Применение модели «мешка слов» к синтетическому набору данных .....	351
Модель «мешка слов» для киноотзывов .....	353
Стоп-слова .....	357
Масштабирование данных с помощью <code>tf-idf</code> .....	358
Исследование коэффициентов модели.....	361
Модель «мешка слов» для последовательностей из нескольких слов (n-грамм) .....	362
Продвинутая токенизация, стемминг и лемматизация .....	366
Моделирование тем и кластеризация документов .....	370
Латентное размещение Дирихле .....	370
Выводы и перспективы .....	377
<b>ГЛАВА 8. ПОДВЕДЕНИЕ ИТОГОВ.....</b>	<b>379</b>
Общий подход к решению задач машинного обучения.....	379
Вмешательство человека в работу модели .....	380
От прототипа к производству .....	381
Тестирование производственных систем .....	382
Создание своего собственного класса <code>Estimator</code> .....	383
Куда двигаться дальше .....	384
Теория.....	384
Другие фреймворки и пакеты машинного обучения .....	385
Ранжирование, рекомендательные системы и другие виды обучения.....	386
Вероятностное моделирование, теория статистического вывода и вероятностное программирование .....	387
Нейронные сети.....	388
Масштабирование на больших наборах данных.....	388

Оттачивание навыков.....	390
Заключение .....	390

# ПРЕДИСЛОВИЕ

Машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, начиная от постановки медицинского диагноза с последующим лечением и заканчивая поиском друзей в социальных сетях. Многие полагают, что машинное обучение могут использовать только крупные компании, обладающие мощными командами аналитиков. В этой книге мы хотим показать вам, с какой легкостью можно самостоятельно построить модели машинного обучения, и рассказать, как это сделать. Прочитав эту книгу, вы сможете построить свою собственную систему машинного обучения, которая позволит выяснить настроения пользователей Твиттера или получить прогнозы по поводу глобального потепления. Область применения машинного обучения безгранична и, учитывая все многообразие данных, имеющихся на сегодняшний день, ограничивается лишь вашим воображением.

## Кому стоит прочитать эту книгу

Данная книга адресована действующим и начинающим специалистам по машинному обучению, решающим реальные задачи. Эта книга является вводной и не требует предварительных знаний в области машинного обучения или искусственного интеллекта. Мы сосредоточимся на использовании языка Python и библиотеки `scikit-learn`, рассмотрим все этапы создания успешного проекта по машинному обучению. Методы, которые мы затронем, пригодятся ученым и исследователям, а также специалистам по анализу данных, работающим в различных коммерческих сферах. Вы получите максимальную отдачу от книги, если хотя бы немного знакомы с языком Python и библиотеками NumPy и matplotlib.

Мы приложили сознательные усилия, направленные на то, чтобы вместо изложения математических подробностей сосредоточиться в большей степени на практических аспектах использования алгоритмов машинного обучения. Поскольку математика (в частности, теория вероятностей) является той основой, на которой строится машинное обучение, мы не будем детально вдаваться в подробности алгоритмов. Если вас интересует математический аппарат алгоритмов машинного обучения, мы рекомендуем книгу издательства Springer *The Elements of Statistical Learning* за авторством Тренора Хаста, Роберта Тибширани и Джерома Фридмана, которая свободно доступна на [сайте авторов](#). Кроме того, мы не будем рассказывать о том, как написать тот или иной алгоритм машинного обучения с нуля, и вместо этого сосредоточимся на

применении большого спектра моделей, уже реализованных в библиотеке `scikit-learn` и других библиотеках.

## Почему мы написали эту книгу

Существует масса книг по машинному обучению и искусственному интеллекту. Однако все они предназначены для студентов старших курсов и аспирантов, обучающихся по специальности «компьютерные науки», и полны математических подробностей. Это резко контрастирует с тем фактом, что машинное обучение в настоящее время используется в качестве прикладного инструмента в научных и коммерческих проектах. Сегодня применение машинного обучения не требует наличия научной степени. Однако существует очень мало ресурсов, в которых все важные аспекты применения машинного обучения на практике освещались бы доступно, без необходимости посещения сложных математических курсов. Мы надеемся, что эта книга поможет людям, которые хотят использовать машинное обучение здесь и сейчас, не тратя годы на изучение математики, линейной алгебры и теории вероятностей.

## Структура книги

Эта книга организована примерно следующим образом:

- В главе 1 кратко рассказывается об основных понятиях машинного обучения и сферах его применения, а также описана установка основных библиотек, которые мы будем использовать на протяжении всей книги.
- В главах 2 и 3 освещаются актуальные алгоритмы машинного обучения, которые широко используются на практике, и обсуждаются их преимущества и недостатки.
- В главе 4 обсуждается важность определенного представления данных, которое можно получить с помощью алгоритмов машинного обучения, а также рассказывается о том, какие аспекты данных требуют внимания.
- В главе 5 освещаются передовые методы, предназначенные для оценки качества модели и настройки параметров, при этом особое внимание уделено перекрестной проверке и решетчатому поиску.
- В главе 6 излагаются принципы построения конвейеров для связывания моделей в единую цепочку и инкапсуляции рабочего потока.
- В главе 7 рассказывается о том, как применять методы, описанные в предыдущих главах, к текстовым данным, а также кратко описываются некоторые методы обработки текста.
- В главе 8 дается общий обзор различных аспектов машинного обучения.

Несмотря на то, что в главах 2 и 3 дается описание наиболее популярных алгоритмов, вполне возможно, что начинающему специалисту совсем не

обязательно знать их все. Если вам необходимо в сжатые сроки построить систему машинного обучения, мы предлагаем начать чтение книги с главы 1 и начальных разделов главы 2, в которых кратко рассказывается об основных принципах машинного обучения. Затем вы можете перейти к разделу «Выводы и перспективы» в главе 2, который включает в себя обзор всех моделей машинного обучения с учителем, освещаемых в этой книге. Вы можете выбрать модель, которая наилучшим образом соответствует вашим задачам, и вернуться в раздел, посвященный этой модели, чтобы ознакомиться с деталями. Затем вы можете воспользоваться методами, описанными в главе 5, чтобы оценить качество модели и настроить ее параметры.

## Онлайн-ресурсы

Изучая материал этой книги, обязательно пользуйтесь [сайтом библиотеки `scikit-learn`](#), на котором найдете подробную документацию о классах и функциях Python, а также массу примеров. Кроме того, существует видеокурс Андреаса Мюллера «Advanced Machine Learning with `scikit-learn`», дополняющий эту книгу. Вы можете найти его по адресу [http://bit.ly/advanced\\_machine\\_learning\\_scikit-learn](http://bit.ly/advanced_machine_learning_scikit-learn).

## Типографские соглашения

В этой книге применяются следующие типографские соглашения:

### *Курсив*

Используется для обозначения новых терминов, URL-адресов, адресов электронной почты, имен файлов и расширений файлов.

### **Моноширинный шрифт**

Используется для листингов программ, а также внутри параграфов для обозначения элементов программ (названий переменных или функций, баз данных, типов данных, переменных среды, операторов и ключевых слов).

### **Моноширинный жирный шрифт**

Обозначает команды или другой текст, который должен вводиться пользователем.

### ***Моноширинный курсив***

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.



Этот элемент означает совет или подсказку



Этот элемент означает примечание



Этот элемент означает предупреждение или предостережение

## Использование примеров программного кода

Все примеры программного кода и упражнения, что приводятся в этой книге, доступны для скачивания по адресу [https://github.com/amueller/introduction\\_to\\_ml\\_with\\_python](https://github.com/amueller/introduction_to_ml_with_python).

Данная книга призвана оказать вам помощь в решении задач, связанных с машинным обучением. Вы можете свободно использовать примеры программного кода из этой книги в своих программах и документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько фрагментов программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получения разрешения не требуется. Но при включении значительного объема программного кода из этой книги в Вашу документацию необходимо получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: An Introduction to Machine Learning with Python (O'Reilly) by Andreas C. Mueller and Sarah Guido. Copyright 2017 Sarah Guido and Andreas Mueller, 978-1-449-36941-5.

Если вы считаете, что использование вами примеров программного кода выходит за разрешенные рамки, присылайте свои вопросы на нашу электронную почту [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Благодарности

### От Андреаса

Без помощи и поддержки большой группы людей эта книга никогда не вышла бы в свет.

Я хотел бы поблагодарить редакторов Меган Бланшетт, Брайана МакДональда и в особенности Дона Шенефельта, который помог мне и Саре воплотить эту книгу в реальность.

Я хочу поблагодарить моих рецензентов Томас Кэсвелла, Оливье Гризеля, Стефана ван дер Уолта и Джона Майлса Уайта, которые помимо того, что заложили основы экосистемы ПО с открытым исходным кодом, еще и нашли время, чтобы прочитать ранние версии этой книги и дали мне неоценимую обратную связь. Я бесконечно благодарен за теплый прием в научном сообществе Python, особенно в среде разработчиков библиотеки `scikit-learn`. Без поддержки и помощи со стороны этого сообщества (в частности, со стороны Гейла Вероко, Алекса Гремфорта и Оливье Гризеля), я никогда не стал бы одним из разработчиков библиотеки `scikit-learn` и не смог бы работать в нем так хорошо, как сейчас. Кроме того, я выражаю благодарность другим разработчикам, которые посвящают свое время улучшению и поддержке этой библиотеки.

Кроме того, я благодарен многочисленным моим коллегам за дискуссии, которые помогли мне лучше понять задачи машинного обучения и подсказали мне идеи для структурирования учебника. Если говорить о людях, с которыми я обсуждал вопросы машинного обучения, я хотел бы отдельно поблагодарить Брайана МакФи, Даниэлу Хуттенкоппен, Джоэла Нотмена, Жилия Люпе, Хьюго Боуни-Андерсона, Свена Крейса, Уго Боун-Андерсона, Свена Крайса, Элис Чжан, Киунхьун Чо, Пабло Бабераса и Дэна Червоне.

Я также выражаю благодарность Рейчел Раков, которая была внимательным бета-тестером и корректором ранних версий этой книги и всячески помогала мне оформить ее.

От себя лично я хочу поблагодарить моих родителей Харальда и Марго, а также мою сестру Мириам за их постоянную поддержку и содействие. Еще я хочу поблагодарить тех многочисленных людей, чья любовь и дружба придали мне силы и энергии для реализации столь сложной задачи.

От Сары

Я хотела бы поблагодарить Мег Бланшетт, без помощи и поддержки которой этого проекта не было бы. Благодарю Целию Ла и Брайана Карлсона за вычитку ранних версий книги. Благодарю команду издательства O'Reilly за их бесконечное терпение. И, наконец, благодарю DTS за их постоянную поддержку.



# ГЛАВА 1. ВВЕДЕНИЕ

Машинное обучение заключается в извлечении знаний из данных. Это научная область, находящаяся на пересечении статистики, искусственного интеллекта и компьютерных наук и также известная как прогнозная аналитика или статистическое обучение. В последние годы применение методов машинного обучения в повседневной жизни стало обыденным явлением. Многие современные веб-сайты и устройства используют алгоритмы машинного обучения, начиная с автоматических рекомендаций по просмотру фильмов, заказа еды или покупки продуктов, и заканчивая персонализированными онлайн-радиотрансляциями и распознаванием друзей на фотографиях. Когда вы видите сложный сайт типа Facebook, Amazon или Netflix, то весьма вероятно, что каждый раздел сайта содержит несколько моделей машинного обучения.

Выйдя за пределы коммерческих приложений, машинное обучение уже оказало огромное влияние на научные исследования, управляемые данными. Инструменты, представленные в этой книге, использовались для решения различных научных задач (исследование звезд, поиск далеких планет, открытие новых частиц, анализ последовательностей ДНК, а также разработка персонализированных методов лечения рака).

Для извлечения прибыли с помощью машинного обучения совсем необязательно, чтобы ваши задачи были столь же крупномасштабными или меняющими мир, как представленные примеры. В этой главе мы объясним, почему машинное обучение стало таким популярным, и обсудим, какие задачи могут быть решены с помощью него. Затем мы покажем вам, как построить свою первую модель машинного обучения, попутно знакомя вас с важными принципами машинного обучения.

## Зачем нужно использовать машинное обучение?

На заре появления «интеллектуальных» приложений многие системы использовали жесткие правила «if» и «else» для обработки данных или корректировки информации, введенной пользователем. Вспомните о спам-филтре, чья работа состоит в том, чтобы переместить соответствующие входящие сообщения электронной почты в папку «Спам». Вы можете составить черный список слов, которые будут идентифицировать письмо как спам. Это пример использования системы экспертных правил для разработки «интеллектуального» приложения. Разработка правил принятия решений в ручном режиме допустимо в некоторых задачах, особенно в тех, где люди четко понимают процесс

моделирования. Однако, использование жестких решающих правил имеет два основных недостатка:

- Логика, необходимая для принятия решения, относится исключительно к одной конкретной области и задачи. Даже незначительное изменение задачи может повлечь за собой переписывание всей системы.
- Разработка правил требует глубокого понимания процесса принятия решения.

Один из примеров, где этот жесткий подход потерпит неудачу – это распознавание лиц на изображениях. На сегодняшний день каждый смартфон может распознать лицо на изображении. Тем не менее, распознавание лиц была нерешенной проблемой, по крайней мере, до 2001 года. Основная проблема заключается в том, что способ, с помощью которого компьютер «воспринимает» пиксели, формирующие изображение на компьютере, очень сильно отличается от человеческого восприятия лица. Эта разница в принципе не позволяет человеку сформулировать подходящий набор правил, описывающих лицо с точки зрения цифрового изображения.

Однако, благодаря машинному обучению, простого предъявления большого количества изображений с лицами будет достаточно для того, чтобы алгоритм определил, какие признаки необходимы для идентификации лица.

## Задачи, которые можно решить с помощью машинного обучения

Наиболее успешные алгоритмы машинного обучения – это те, которые автоматизируют процессы принятия решений путем обобщения известных примеров. В этих методах, известных как *обучение с учителем* или *контролируемое обучение* (*supervised learning*), пользователь предоставляет алгоритму пары объект-ответ, а алгоритм находит способ получения ответа по объекту. В частности, алгоритм способен выдать ответ для объекта, которого он никогда не видел раньше, без какой-либо помощи человека. Если вернуться к примеру классификации спама с использованием машинного обучения, пользователь предъявляет алгоритму большое количество писем (объекты) вместе с информацией о том, является ли письмо спамом или нет (ответы). Для нового электронного письма алгоритм вычислит вероятность, с которой это письмо можно отнести к спаму.

Алгоритмы машинного обучения, которые учатся на парах объект-ответ, называются алгоритмами обучения с учителем, так как «учитель» показывает алгоритму ответ в каждом наблюдении, по которому

происходит обучение. Несмотря на то, что создание набора с объектами и ответами – это часто трудоемкий процесс, осуществляемый вручную, алгоритмы обучения с учителем интерпретируемы и качество их работы легко измерить. Если вашу задачу можно сформулировать в виде задачи обучения с учителем, и вы можете создать набор данных, который включает в себя ответы, вероятно, машинное обучение решит вашу проблему.

Примеры задач машинного обучения с учителем:

*Определение почтового индекса по рукописным цифрам на конверте*

Здесь объектом будет сканированное изображение почерка, а ответ – фактические цифры почтового индекса. Чтобы создать набор данных для построения модели машинного обучения, вам нужно собрать большое количество конвертов. Затем вы можете самостоятельно прочитать почтовые индексы и сохранить цифры в виде ответов.

*Определение доброкачественности опухоли на основе медицинских изображений*

Здесь объектом будет изображение, а ответом – диагноз, является ли опухоль доброкачественной или нет. Чтобы создать набор данных для построения модели, вам нужна база медицинских изображений. Кроме того, необходимо мнение эксперта, поэтому врач должен просмотреть все изображения и решить, какие опухоли являются доброкачественными, а какие – нет. Помимо анализа изображения может понадобиться дополнительная диагностика для определения доброкачественности опухоли.

*Обнаружение мошеннической деятельности в сделках по кредитным картам*

Здесь объект – запись о транзакции по кредитной карте, а ответ – информация о том, является ли транзакция мошеннической или нет. Предположим, вы – учреждение, выдающее кредитные карты, сбор данных подразумевает сохранение всех транзакций и запись сообщений клиентов о мошеннических транзакциях.

Приведя эти примеры, интересно отметить что, хотя объекты и ответы выглядят достаточно просто, процесс сбора данных для этих трех задач существенно отличается. Несмотря на то что чтение конвертов является трудоемким занятием, этот процесс прост и дешев. Получение медицинских изображений и проведение диагностики требует не только дорогостоящего оборудования, но и редких, высокооплачиваемых экспертных знаний, не говоря уже об этических проблемах и вопросах конфиденциальности. В примере обнаружения мошенничества с кредитными картами, сбор данных осуществляется намного проще. Ваши

клиенты сами предоставят вам ответы, сообщая о мошенничестве. Все, что вам нужно сделать для получения объектов и ответов, связанных с мошеннической активностью, – это подождать.

*Алгоритмы обучения без учителя или неконтролируемого обучения (unsupervised algorithms)* – это еще один вид алгоритмов, который мы рассмотрим в этой книге. В алгоритмах обучения без учителя известны только объекты, а ответов нет. Хотя есть много успешных сфер применения этих методов, их, как правило, труднее интерпретировать и оценить.

Примеры задач машинного обучения без учителя:

#### *Определение тем в наборе постов*

Если у вас есть большая коллекция текстовых данных, вы можете агрегировать их и найти распространенные темы. У вас нет предварительной информации о том, какие темы там затрагиваются и сколько их. Таким образом, нет никаких известных ответов.

#### *Сегментирование клиентов на группы с похожими предпочтениями*

Имея набор записей о клиентах, вы можете определить группы клиентов со схожими предпочтениями. Для торгового сайта такими группами могут быть «родители», «книгочеи» или «геймеры». Поскольку вы не знаете заранее о существовании этих групп и их количестве, у вас нет ответов.

#### *Обнаружение паттернов аномального поведения на веб-сайте*

Чтобы выявить злоупотребления или ошибки, часто бывает полезно найти паттерны поведения, которые отличаются от нормы. Паттерны аномального поведения могут быть разными, и, возможно, у вас не будет зарегистрированных случаев аномального поведения. Поскольку в этом примере вы наблюдаете лишь трафик, и вы не знаете, что представляет собой нормальное и ненормальное поведение, речь идет о задаче обучения без учителя.

Решая задачи машинного обучения с учителем и без, важно представить ваши входные данные в формате, понятном компьютеру. Часто данные представляют в виде таблицы. Каждая точка данных, которую вы хотите исследовать (каждое электронное письмо, каждый клиент, каждая транзакция) является строкой, а каждое свойство, которое описывает эту точку данных (скажем, возраст клиента, сумма или место совершения транзакции), является столбцом. Вы можете описать пользователей по возрасту, полу, дате создания учетной записи и частоте покупок в вашем интернет-магазине. Вы можете описать изображение опухоли с помощью градаций серого цвета для каждого пикселя или с помощью размера, формы и цвета опухоли.

В машинном обучении каждый объект или строка называются *примером (sample)* или *точкой данных (data point)*, а столбцы-свойства, которые описывают эти примеры, называются *характеристиками* или *признаками (features)*.

Позже в этой книге мы более детально остановимся на теме подготовки данных, которая называется *выделение признаков (feature extraction)* или *конструирование признаков (feature engineering)*. Однако, вы должны иметь в виду, что ни один алгоритм машинного обучения не сможет сделать прогноз по данным, которые не содержат никакой полезной информации. Например, если единственный признак пациента – это его фамилия, алгоритм не сможет предсказать его пол. Этой информации просто нет в данных. Если добавить еще один признак – имя пациента, то дело уже будет обстоять лучше, поскольку часто, зная имя человека, можно судить о его поле.

## Постановка задач и знакомство с данными

Вполне возможно, что самая важная часть процесса машинного обучения – это интерпретация данных, с которыми вы работаете, и применимость этих данных к задаче, которую вы хотите решить. Выбрать случайным образом алгоритм и скормить ему свои данные – неэффективное решение. Прежде чем приступить к построению модели, необходимо понять, что представляет собой ваш набор данных. Каждый алгоритм отличается с точки зрения типа обрабатываемых данных и вида решаемых задач. Создавая модель машинного обучения, вы должны ответить, или, по крайней мере, задуматься над следующими вопросами:

- На какой вопрос(ы) я пытаюсь ответить? Собранные данные могут ответить на этот вопрос?
- Как лучше всего сформулировать свой вопрос(ы) с точки зрения задач машинного обучения?
- У меня собрано достаточно данных, чтобы составить представление о задаче, которую я хочу решить?
- Какие признаки я извлек и помогут ли они мне получить правильные прогнозы?
- Как я буду измерять эффективность решения задачи?
- Как решение, полученное с помощью машинного обучения, будет взаимодействовать с другими компонентами моего исследования или бизнес-продукта?

В более широком контексте, алгоритмы и методы машинного обучения являются лишь этапом более крупного процесса, призванного решить конкретную задачу, и поэтому необходимо всегда держать схему

этого процесса в голове. Многие тратят массу времени, создавая сложные модели машинного обучения решения, затем узнавая, что решают задачу неправильно.

Углубляясь в технические аспекты машинного обучения (что мы и будем делать этой книге), легко упустить из виду конечные цели. Несмотря на то, что мы не будем подробно обсуждать вопросы, перечисленные здесь, мы все же рекомендуем вам вспомнить о них, когда вы начнете строить модели машинного обучения.

## Почему нужно использовать Python?

Python стал общепринятым языком для многих сфер применения науки о данных (data science). Он сочетает в себе мощь языков программирования с простотой использования предметно-ориентированных скриптовых языков типа MATLAB или R. В Python есть библиотеки для загрузки данных, визуализации, статистических вычислений, обработки естественного языка, обработки изображений и многого другого. Этот обширный набор инструментов предлагает специалистам по работе с данными (data scientists) большой набор инструментов общего и специального назначения. Одним из основных преимуществ использования Python является возможность напрямую работать с программным кодом с помощью терминала или других инструментов типа Jupyter Notebook, который мы рассмотрим ниже. Машинное обучение и анализ данных – это в основном итерационные процессы, в которых данные задают ход анализа. Крайне важно для этих процессов иметь инструменты, которые позволяют оперативно и легко работать.

В качестве языка программирования общего назначения Python позволяет создавать сложные графические пользовательские интерфейсы (GUI) и веб-сервисы, а также легко интегрироваться в уже существующие системы.

## scikit-learn

**scikit-learn** – проект с открытым исходным кодом, это означает, что его можно свободно использовать и распространять, и любой человек может легко получить исходный код, чтобы увидеть, что происходит «за кулисами». Проект **scikit-learn** постоянно развивается и совершенствуется, и у него очень активное сообщество пользователей. Он содержит ряд современных алгоритмов машинного обучения, а также полную документацию по каждому алгоритму. **scikit-learn** – очень популярный инструмент и самая известная питоновская библиотека для

машинного обучения. Она широко используется в промышленности и науке, а в интернете имеется богатый выбор обучающих материалов и примеров программного кода. `scikit-learn` прекрасно работает с рядом других научных инструментов Python, которые мы обсудим позже в этой главе.

По мере чтения книги мы рекомендуем также ознакомиться с [руководством пользователя](#) по `scikit-learn` и документацией по API для получения дополнительной информации о многочисленных параметрах каждого алгоритма. Онлайн-документация является очень подробной, и эта книга познакомит вас со всеми необходимыми основами машинного обучения, чтобы вы научились детально разбираться в нем.

## Установка `scikit-learn`

`scikit-learn` требует наличия еще двух пакетов Python – NumPy и SciPy. Для построения графиков и интерактивной работы необходимо также установить `matplotlib`, IPython и Jupyter Notebook. Мы рекомендуем использовать один из нижеперечисленных дистрибутивов Python, которые уже включают все необходимые пакеты:

### [\*Anaconda\*](#)

Дистрибутив Python, предназначенный для крупномасштабной обработки данных, прогнозной аналитики и научных вычислений. Anaconda уже включает NumPy, SciPy, `matplotlib`, `pandas`, IPython, Jupyter Notebook и `scikit-learn`. Есть версии для Mac OS, Windows и Linux. Это очень удобное решение и это тот дистрибутив, который мы рекомендуем пользователям, у которых еще не установлены пакеты Python для научных вычислений. Кроме того, сейчас Anaconda включает в себя коммерческую библиотеку Intel MKL, которой можно пользоваться бесплатно. Использование MKL (это происходит автоматически при установке Anaconda) может дать значительный прирост скорости при выполнении различных алгоритмов в `scikit-learn`.

### [\*Enthought Canopy\*](#)

Еще один дистрибутив Python для научных вычислений. Он уже содержит NumPy, SciPy, `matplotlib`, `pandas` и IPython, но бесплатная версия не включает `scikit-learn`. Если вы являетесь учебным заведением, вы можете запросить учебную лицензию и получить свободный доступ к платной версии Enthought Canopy. Enthought Canopy доступен для Python 2.7.x и работает на Mac OS, Windows и Linux.



## [\*Python \(x, y\)\*](#)

Свободный дистрибутив Python для научных вычислений, специально предназначенный для Windows. Python (x, y) включает NumPy, SciPy, matplotlib, pandas, IPython и scikit-learn.

Если у вас уже стоит Python, вы можете использовать `pip` для установки всех этих пакетов:

```
$ pip install numpy scipy matplotlib ipython scikit-learn pandas
```

## Основные библиотеки и инструменты

Понимание принципов работы и использования `scikit-learn` – важно, но есть несколько других библиотек, которые расширят ваш опыт. `scikit-learn` базируется на двух питоновских библиотеках для научных вычислений NumPy и SciPy. Помимо NumPy и SciPy мы будем использовать `pandas` и `matplotlib`. Кроме того, мы познакомимся с Jupyter Notebook, который представляет собой интерактивную среду программирования на основе браузера. Если коротко, то ниже приводится информация о перечисленных инструментах, которыми вы должны овладеть, чтобы получить максимальную отдачу от `scikit-learn`.<sup>1</sup>

### Jupyter Notebook

Jupyter Notebook представляет собой интерактивную среду для запуска программного кода в браузере. Это отличный инструмент для разведочного анализа данных и широко используется специалистами по анализу данных. Несмотря на то что Jupyter Notebook поддерживает множество языков программирования, нам нужна лишь поддержка Python. Jupyter Notebook позволяет легко интегрировать программный код, текст и изображения, и вся эта книга была фактически написана в формате Jupyter Notebook. Все примеры программного кода, приведенные в этой книге, можно загрузить с [GitHub](#).

### NumPy

NumPy – это один из основных пакетов для научных вычислений в Python. Он содержит функциональные возможности для работы с многомерными массивами, высокоуровневыми математическими

---

<sup>1</sup> Если вы не знакомы с NumPy или matplotlib, мы рекомендуем прочитать первую главу [SciPy Lectures Notes](#).



функциями (операции линейной алгебры, преобразование Фурье, генератор псевдослучайных чисел).

В `scikit-learn` массив NumPy – это основная структура данных. `scikit-learn` принимает данные в виде массивов NumPy. Любые данные, которые вы используете, должны быть преобразованы в массив NumPy. Базовый функционал NumPy – это класс `ndarray`, многомерный (n-мерный) массив. Все элементы массива должны быть одного и того же типа. Массив NumPy выглядит следующим образом:

```
In[2]:
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))

Out[2]:
x:
[[1 2 3]
 [4 5 6]]
```

Мы будем *очень много* использовать NumPy в этой книге, и будем называть объекты класса `ndarray` «массивами NumPy» или просто «массивами».

## SciPy

SciPy – это набор функций для научных вычислений в Python. Помимо всего прочего он предлагает продвинутые процедуры линейной алгебры, математическую оптимизацию функций, обработку сигналов, специальные математические функции и статистические функции. `scikit-learn` использует набор функций SciPy для реализации своих алгоритмов. Для нас наиболее важной частью SciPy является пакет `scipy.sparse`: с помощью него мы получаем *разреженные матрицы* (*sparse matrices*), которые представляют собой еще один формат данных, который используется в `scikit-learn`. Разреженные матрицы используются всякий раз, когда нам нужно сохранить 2D массив, который содержит в основном нули:

```
In[3]:
from scipy import sparse

# Создаем 2D массив NumPy с единицами по главной диагонали и нулями в остальных ячейках
eye = np.eye(4)
print("массив NumPy:\n{}".format(eye))

Out[3]:
массив NumPy:
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]
```

```
In[4]:
# Преобразовываем массив NumPy в разреженную матрицу SciPy в формате CSR
# Сохраняем лишь ненулевые элементы
sparse_matrix = sparse.csr_matrix(eye)
print("\nразреженная матрица SciPy в формате CSR:\n{}".format(sparse_matrix))
```

```
Out[4]:
разреженная матрица SciPy в формате CSR:
(0, 0) 1.0
(1, 1) 1.0
(2, 2) 1.0
(3, 3) 1.0
```

Обычно невозможно плотно записать разреженные данные (поскольку они не уместились бы в памяти), поэтому нам нужно непосредственно создать разреженные матрицы. Ниже приводится способ, который позволяет создать такую же разреженную матрицу, что была приведена выше, но этот раз с использованием формата COO<sup>2</sup>:

```
In[5]:
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("формат COO:\n{}".format(eye_coo))
```

```
Out[5]:
формат COO:
(0, 0) 1.0
(1, 1) 1.0
(2, 2) 1.0
(3, 3) 1.0
```

Более подробную информацию о разреженных матрицах SciPy можно найти в [SciPy Lecture Notes](#).

## matplotlib

**matplotlib** – это основная библиотека для построения научных графиков в Python. Она включает функции для создания высококачественных визуализаций типа линейных диаграмм, гистограмм, диаграмм разброса и т.д. Визуализация данных и различных аспектов вашего анализа может дать вам важную информацию, и мы будем использовать **matplotlib** для всех наших визуализаций. При работе в Jupyter Notebook, вы можете вывести рисунок прямо в браузере с помощью встроенных команд `%matplotlib notebook` и `%matplotlib inline`. Мы рекомендуем использовать `%matplotlib notebook`, который предлагает интерактивное окружение (хотя при написании этой книги мы использовали `%matplotlib inline`). Например, нижеприведенный программный код строит график, изображенный на рис. 1.1:

---

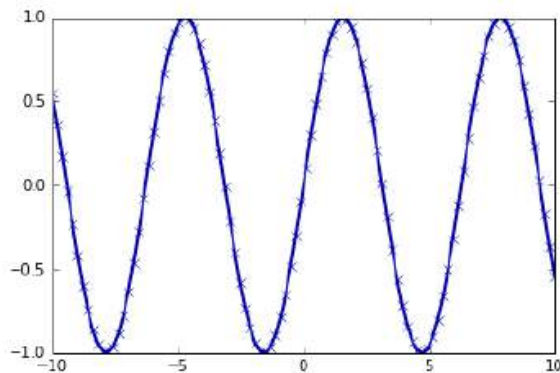
<sup>2</sup> COO (coordinate format) – координатный формат хранения разреженных матриц: хранятся только ненулевые элементы матрицы и их координаты (номера строк и столбцов). – Прим. пер.

```

In[6]:
%matplotlib inline
import matplotlib.pyplot as plt

# Генерируем последовательность чисел от -10 до 10 с 100 шагами
x = np.linspace(-10, 10, 100)
# Создаем второй массив с помощью синуса
y = np.sin(x)
# Функция создает линейный график на основе двух массивов
plt.plot(x, y, marker="x")

```



**Рис. 1.1** Простой линейный график синусоидальной функции с использованием matplotlib

## pandas

**pandas** – библиотека Python для обработки и анализа данных. Она построена на основе структуры данных, называемой **DataFrame** и смоделированной по принципу датафреймов среды статистического программирования R. Проще говоря, **DataFrame** библиотеки **pandas** представляет собой таблицу, похожую на электронную таблицу Excel. Библиотека **pandas** предлагает большой спектр методов по работе с этой таблицей, в частности, она позволяет выполнять SQL-подобные запросы и присоединения таблиц. В отличие от NumPy, который требует, чтобы все записи в массиве были одного и того же типа, в **pandas** каждый столбец может иметь отдельный тип (например, целые числа, даты, числа с плавающей точкой и строки). Еще одним преимуществом библиотеки **pandas** является ее способность работать с различными форматами файлов и баз данных, например, с файлами SQL, Excel и CSV. Детальное рассмотрение функционала **pandas** выходит за рамки этой книги. Тем не менее, [Python for Data Analysis](#) Уэса МакКинни (O'Reilly, 2012) является замечательным руководством. Ниже приводится небольшой пример создания **DataFrame** с помощью словаря:

```
In[7]:
import pandas as pd

# создаем простой набор данных с характеристиками пользователей
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location': ["New York", "Paris", "Berlin", "London"],
        'Age': [24, 13, 53, 33]}

data_pandas = pd.DataFrame(data)
# IPython.display позволяет "красиво напечатать" датафреймы
# в Jupyter notebook
display(data_pandas)
```

Приведенный код генерирует следующий вывод:

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

Существует несколько способов осуществить запрос к таблице. Например:

```
In[8]:
# Выбрать все строки, в которых значение столбца age больше 30
display(data_pandas[data_pandas.Age > 30])
```

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

## mglearn

Эта книга содержит сопутствующий программный код, который вы можете найти на [GitHub](#). Программный код включает в себя не только все примеры, приведенные в этой книге, но и библиотеку **mglearn**. Она представляет собой библиотеку, включающую разные полезные функции. Мы написали ее для этой книги, чтобы не перегружать листинги подробной информацией о построении графиков и загрузке данных. Если вам интересно, вы можете посмотреть все эти функции в репозитории, но детали **mglearn** не очень важны для понимания материала этой книги. Если вы видите вызов **mglearn** в программном коде, то речь, как правило,

идет о быстром способе построить красивую картинку или загрузить некоторые интересные данные.<sup>3</sup>



На протяжении всей книги мы будем достаточно много использовать NumPy, matplotlib и pandas. Поэтому убедитесь в импорте следующих библиотек:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
```

Кроме того, мы полагаем, что вы будете запускать программный код в Jupyter Notebook, используя замечательные возможности `%matplotlib notebook` или `%matplotlib inline` для построения графиков. Если вы не используете notebook или эти замечательные команды, вам придется вызвать `plt.show`, чтобы вывести эти графики.

Некоторые примеры используют функцию `display` оболочки IPython, поэтому если при выполнении программного кода вы получаете ошибку, в которой так или иначе упоминается `display`, запустите следующую строку:

```
from IPython.display import display
```

Для решения проблемы корректного отображения русских надписей в графиках matplotlib воспользуйтесь следующим программным кодом:

```
plt.rc('font', family='Verdana')
```

Таким образом, первый блок программного кода каждой главы должен выглядеть следующим образом:

```
[In1]
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import mglearn
from IPython.display import display
plt.rc('font', family='Verdana')
```

---

<sup>3</sup> Самый простой способ воспользоваться библиотекой mglearn, скачать папку mglearn с [GitHub](#) и в переменной окружения PATH прописать полный путь к ней. В Windows 7 для этого нажмите кнопку **Пуск**, выберите **Панель управления**. Дважды нажмите на **Система**, затем выберите **Дополнительные параметры системы**. Во вкладке **Дополнительно** нажмите на **Переменные среды**. Выберите **Path** и нажмите на **Изменить**. В поле **Значение переменной** введите путь к папке mglearn (например, C:\Anaconda3\mglearn). – *Прим. пер.*

## Сравнение Python 2 и Python 3

Существуют две основные версии Python, которые широко используются на данный момент: Python 2 (точнее, 2.7) и Python 3 (последняя версия 3.5 на момент написания книги). Иногда это приводит к некоторой путанице. Python 2 уже активно не развивается, а поскольку Python 3 содержит существенные изменения, код, написанный для Python 2, как правило, не запустится в Python 3. Если вы новичок в Python или запускаете новый проект с нуля, мы настоятельно рекомендуем использовать последнюю версию Python 3. Если у вас есть большой фрагмент программного кода, написанный для Python 2, вам не нужно что-либо менять. Однако вы должны попытаться перейти на Python 3 как можно скорее. Вообще, при написании нового программного кода, как правило, довольно легко написать код, который будет работать и под Python 2 и под Python 3.<sup>4</sup> Если у вас нет необходимости использовать устаревшее программное обеспечение, вы должны обязательно использовать Python 3. Весь программный код в этой книге написан таким образом, что работает в обеих версиях. Однако некоторые детали вывода могут отличаться в этих версиях.

## Версии библиотек, используемые в этой книге

В этой книге мы используем следующие версии ранее упомянутых библиотек:

```
In[9]:
import sys
print("версия Python: {}".format(sys.version))

import pandas as pd
print("версия pandas: {}".format(pd.__version__))

import matplotlib
print("версия matplotlib: {}".format(matplotlib.__version__))

import numpy as np
print("версия NumPy: {}".format(np.__version__))

import scipy as sp
print("версия SciPy: {}".format(sp.__version__))

import IPython
print("версия IPython: {}".format(IPython.__version__))

import sklearn
print("версия scikit-learn: {}".format(sklearn.__version__))

версия Python: 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
версия pandas: 0.18.1
версия matplotlib: 1.5.1
версия NumPy: 1.11.1
версия SciPy: 0.17.1
```

---

<sup>4</sup> Для решения этой задачи очень удобен пакет [six](#).

Несмотря на то, что точное соответствие вышеприведенным версиям неважно, у вас должна быть установлена версия **scikit-learn**, которая была последней по крайней мере на момент написания книги.

Теперь, когда мы все установили, давайте в первый раз применим машинное обучение.



Эта книга предполагает, что у вас установлена **scikit-learn** версии 0.18 или более свежая. Модуль **model\_selection** появился в версии 0.18, и если вы используете более раннюю версию **scikit-learn**, вам нужно обновить **scikit-learn**, чтобы воспользоваться этим модулем.<sup>5</sup>

## Первый пример: классификация сортов ириса

В этом разделе мы рассмотрим простой пример применения машинного обучения и построим нашу первую модель. В процессе изложения материала мы познакомим вас с некоторыми основными принципами и терминами.

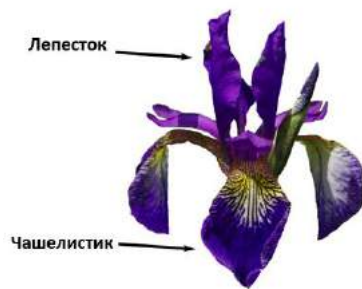
Предположим, что ботаник-любитель хочет классифицировать сорта ирисов, которые он собрал. Он измерил в сантиметрах некоторые характеристики ирисов: длину и ширину лепестков, а также длину и ширину чашелистиков (см. рис. 1.2).

Кроме того, у него есть измерения этих же характеристик ирисов, которые ранее позволили опытному эксперту отнести их к сортам *setosa*, *versicolor* и *virginica*. Относительно этих ирисов ботаник-любитель уверенно может сказать, к какому сорту принадлежит каждый ирис. Давайте предположим, что перечисленные сорта являются единственными сортами, которые ботаник-любитель может встретить в дикой природе.

Наша цель заключается в построении модели машинного обучения, которая сможет обучиться на основе характеристик ирисов, уже классифицированных по сортам, и затем предскажет сорт для нового цветка ириса.

---

<sup>5</sup> Например, если вы установили пакет Anaconda для Windows, воспользуйтесь менеджером **conda**:  
`conda install -c anaconda scikit-learn=0.18.1`



**Рис. 1.2** Структура цветка ириса

Поскольку у нас есть примеры, по которым мы уже знаем правильные сорта ириса, решаемая задача является задачей обучения с учителем. В этой задаче нам нужно спрогнозировать один из сортов ириса. Это пример задачи *классификации* (*classification*). Возможные ответы (различные сорта ириса) называются *классами* (*classes*). Каждый ирис в наборе данных принадлежит к одному из трех классов, таким образом решаемая задача является задачей трехклассовой классификации.

Ответом для отдельной точки данных (ириса) является тот или иной сорт этого цветка. Сорт, к которому принадлежит цветок (конкретная точка данных), называется *меткой* (*label*).

## Загружаем данные

Данные, которые мы будем использовать для этого примера, – это набор данных Iris, классический набор данных в машинном обучении и статистике. Он уже включен в модуль `datasets` библиотеки `scikit-learn`. Мы можем загрузить его, вызвав функцию `load_iris`:

```
In[10]:  
from sklearn.datasets import load_iris  
iris_dataset = load_iris()
```

Объект `iris`, возвращаемый `load_iris`, является объектом `Bunch`, который очень похож на словарь. Он содержит ключи и значения:

```
In[11]:  
print("Ключи iris_dataset: \n{}".format(iris_dataset.keys()))  
  
Out[11]:  
Ключи iris_dataset:  
dict_keys(['target_names', 'feature_names', 'DESCR', 'data', 'target'])
```

Значение ключа `DESCR` – это краткое описание набора данных. Здесь мы покажем начало описания (оставшуюся часть описания вы можете посмотреть самостоятельно):



```
In[12]:
print(iris_dataset['DESCR'][:193] + "\n...")
```

```
Out[12]:
Iris Plants Database
=====
Notes
-----
Data Set Characteristics:
: Number of Instances: 150 (50 in each of three classes)
: Number of Attributes: 4 numeric, predictive att
...
-----
```

Значение ключа `target_names` – это массив строк, содержащий сорта цветов, которые мы хотим предсказать:

```
In[13]:
print("Названия ответов: {}".format(iris_dataset['target_names']))

Out[13]:
Названия ответов: ['setosa' 'versicolor' 'virginica']
```

Значение `feature_names` – это список строк с описанием каждого признака:

```
In[14]:
print("Названия признаков: {}".format(iris_dataset['feature_names']))

Out[14]:
Названия признаков:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
 'petal width (cm)']
```

Сами данные записаны в массивах `target` и `data`. `data` – массив NumPy, который содержит количественные измерения длины чашелистиков, ширины чашелистиков, длины лепестков и ширины лепестков:

```
In[15]:
print("Тип массива data: {}".format(type(iris_dataset['data'])))

Out[15]:
Тип массива data: <class 'numpy.ndarray'>
```

Строки в массиве `data` соответствуют цветам ириса, а столбцы представляют собой четыре признака, которые были измерены для каждого цветка:

```
In[16]:
print("Форма массива data: {}".format(iris_dataset['data'].shape))

Out[16]:
Форма массива data: (150, 4)
```

Мы видим, что массив содержит измерения для 150 различных цветов по 4 признакам. Вспомним, что в машинном обучении отдельные элементы называются *примерами* (*samples*), а их свойства –

характеристиками или признаками (*feature*). Форма (*shape*) массива данных определяется количеством примеров, умноженным на количество признаков. Это является общепринятым соглашением в `scikit-learn`, и ваши данные всегда будут представлены в этой форме. Ниже приведены значения признаков для первых пяти примеров:

```
In[17]:
print("Первые пять строк массива data:\n{}".format(iris_dataset['data'][:5]))
```

```
Out[17]:
Первые пять строк массива data:
[[ 5.1  3.5  1.4  0.2]
 [ 4.9  3.   1.4  0.2]
 [ 4.7  3.2  1.3  0.2]
 [ 4.6  3.1  1.5  0.2]
 [ 5.   3.6  1.4  0.2]]
```

Взглянув на эти данные, мы видим, что все пять цветов имеют ширину лепестка 0.2 см и первый цветок имеет самую большую длину чашелистика, 5.1 см.

Массив `target` содержит сорта уже измеренных цветов, тоже записанные в виде массива NumPy:

```
In[18]:
print("Тип массива target: {}".format(type(iris_dataset['target'])))
```

Out[18]:  
Тип массива target: <class 'numpy.ndarray'>

**target** представляет собой одномерный массив, по одному элементу для каждого цветка:

```
In[19]:
print("Форма массива target: {}".format(iris_dataset['target'].shape))
```

**Out[19]:**  
Форма массива target: (150,)

Сорта кодируются как целые числа от 0 до 2:

```
In[20]:
print("Ответы:\n{}".format(iris_dataset['target']))
```

[illegible]

Значения чисел задаются массивом `iris['target_names']`: 0 – *setosa*, 1 – *versicolor*, а 2 – *virginica*.

## Метрики эффективности: обучающий и тестовый наборы

На основе этих данных нам нужно построить модель машинного обучения, которая предскажет сорта ириса для нового набора измерений. Но прежде, чем мы применим нашу модель к новому набору, мы должны убедиться в том, что модель на самом деле работает и ее прогнозам можно доверять.

К сожалению, для оценки качества модели мы не можем использовать данные, которые были взяты нами для построения модели. Это обусловлено тем, что наша модель просто запомнит весь обучающий набор и поэтому она всегда будет предсказывать правильную метку для любой точки данных в обучающем наборе. Это «запоминание» ничего не говорит нам об обобщающей способности модели (другими словами, мы не знаем, будет ли эта модель так же хорошо работать на новых данных).

Для оценки эффективности модели, мы предъявляем ей новые размеченные данные (размеченные данные, которые она не видела раньше). Обычно это делается путем разбиения собранных размеченных данных (в данном случае 150 цветов) на две части. Одна часть данных используется для построения нашей модели машинного обучения и называется *обучающими данными* (*training data*) или *обучающим набором* (*training set*). Остальные данные будут использованы для оценки качества модели, их называют *тестовыми данными* (*test data*), *тестовым набором* (*test set*) или *контрольным набором* (*hold-out set*).

В библиотеке `scikit-learn` есть функция `train_test_split`, которая перемешивает набор данных и разбивает его на две части. Эта функция отбирает в обучающий набор 75% строк данных с соответствующими метками. Оставшиеся 25% данных с метками объявляются тестовым набором. Вопрос о том, сколько данных отбирать в обучающий и тестовый наборы, является дискуссионным, однако использование тестового набора, содержащего 25% данных, является хорошим правилом.

В `scikit-learn` данные, как правило, обозначаются заглавной  $X$ , тогда как метки обозначаются строчной  $y$ . Это навеяно стандартной математической формулой  $f(x)=y$ , где  $x$  является аргументом функции, а  $y$  – выводом. В соответствии с некоторыми математическими соглашениями мы используем заглавную  $X$ , потому что данные представляют собой двумерный массив (матрицу) и строчную  $y$ , потому что целевая переменная – это одномерный массив (вектор).

Давайте вызовем функцию `train_test_split` для наших данных и зададим обучающие данные, обучающие метки, тестовые данные, тестовые метки, используя вышеупомянутые буквы:

```
In[21]:
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

Перед разбиением функция `train_test_split` перемешивает набор данных с помощью генератора псевдослучайных чисел. Если мы просто возьмем последние 25% наблюдений в качестве тестового набора, все точки данных будет иметь метку 2, поскольку все точки данных отсортированы по меткам (смотрите вывод для `iris['target']`, показанный ранее). Используя тестовый набор, содержащий только один из трех классов, вы не сможете объективно судить об обобщающей способности модели, таким образом, мы перемешиваем наши данные, чтобы тестовые данные содержали все три класса.

Чтобы в точности повторно воспроизвести полученный результат, мы воспользуемся генератором псевдослучайных чисел с фиксированным стартовым значением, которое задается с помощью параметра `random_state`. Это позволит сделать результат воспроизводим, поэтому вышеприведенный программный код будет генерировать один и тот же результат. Мы всегда будем задавать `random_state` при использовании рандомизированных процедур в этой книге.

Выводом функции `train_test_split` являются `X_train`, `X_test`, `y_train` и `y_test`, которые все являются массивами Numpy. `X_train` содержит 75% строк набора данных, а `X_test` содержит оставшиеся 25%:

```
In[22]:
print("форма массива X_train: {}".format(X_train.shape))
print("форма массива y_train: {}".format(y_train.shape))
```

```
Out[22]:
форма массива X_train: (112, 4)
форма массива y_train: (112,)
```

```
In[23]:
print("форма массива X_test: {}".format(X_test.shape))
print("форма массива y_test: {}".format(y_test.shape))
```

```
Out[23]:
форма массива X_test: (38, 4)
форма массива y_test: (38,)
```

## Сперва посмотрите на Ваши данные

Перед тем как строить модель машинного обучения, неплохо было бы исследовать данные, чтобы понять, можно ли легко решить поставленную задачу без машинного обучения или содержится ли нужная информация в данных.

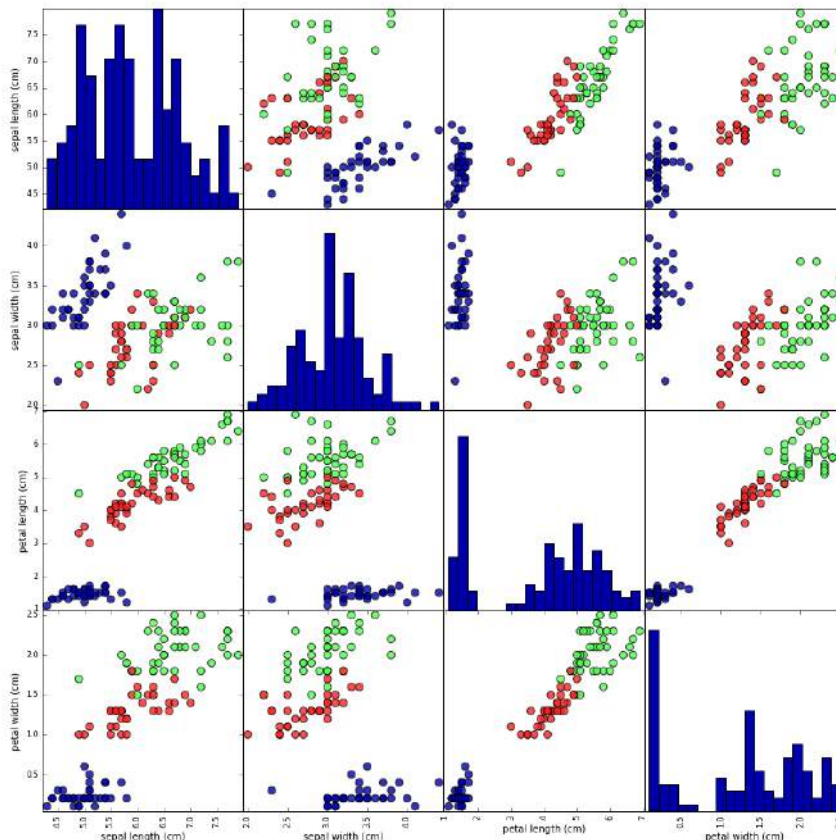
Кроме того, исследование данных – это хороший способ обнаружить аномалии и особенности. Например, вполне возможно, что некоторые из ваших ирисов измерены в дюймах, а не в сантиметрах. В реальном мире нестыковки в данных и неожиданности очень распространены.

Один из лучших способов исследовать данные – визуализировать их. Это можно сделать, используя *диаграмму рассеяния* (*scatter plot*). В диаграмме рассеяния один признак откладывается по оси x, а другой признак – по оси y, каждому наблюдению соответствует точка. К сожалению, экран компьютера имеют только два измерения, что позволяет разместить на графике только два (или, возможно, три) признака одновременно. Таким образом, трудно разместить на графике наборы данных с более чем тремя признаками. Один из способов решения этой проблемы – построить *матрицу диаграмм рассеяния* (*scatterplot matrix*) или *парные диаграммы рассеяния* (*pair plots*), на которых будут изображены все возможные пары признаков. Если у вас есть небольшое число признаков, например, четыре, как здесь, то использование матрицы диаграмм рассеяния будет вполне разумным. Однако, вы должны помнить, что матрица диаграмм рассеяния не показывает взаимодействие между всеми признаками сразу, поэтому некоторые интересные аспекты данных не будут выявлены с помощью этих графиков.

Рис. 1.3 представляет собой матрицу диаграмм рассеяния для признаков обучающего набора. Точки данных окрашены в соответствии с сортами ириса, к которым они относятся. Чтобы построить диаграммы, мы сначала преобразовываем массив NumPy в **DataFrame** (основный тип данных в библиотеке **pandas**). В **pandas** есть функция для создания парных диаграмм рассеяния под названием **scatter\_matrix**. По диагонали этой матрицы располагаются гистограммы каждого признака:

**In[24]:**

```
# создаем dataframe из данных в массиве X_train
# маркируем столбцы, используя строки в iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# создаем матрицу рассеяния из dataframe, цвет точек задаем с помощью y_train
grr = pd.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15), marker='o',
                        hist_kws={'bins': 20}, s=60, alpha=.8, cmap=mglearn.cm3)
```



**Рис. 1.3** Матрица диаграмм рассеяния для набора данных Iris, цвет точек данных определяется метками классов

Взглянув на график, мы можем увидеть, что, похоже, измерения чашелистиков и лепестков позволяют относительно хорошо разделить три класса. Это означает, что модель машинного обучения, вероятно, сможет научиться разделять их.

## Построение вашей первой модели: метод $k$ ближайших соседей

Теперь мы можем начать строить реальную модель машинного обучения. В библиотеке `scikit-learn` имеется довольно много алгоритмов классификации, которые мы могли бы использовать для построения модели. В данном примере мы будем использовать классификатор на основе метода  $k$  ближайших соседей, который легко интерпретировать. Построение этой модели заключается лишь в запоминании обучающего набора. Для того, чтобы сделать прогноз для новой точки данных, алгоритм находит точку в обучающем наборе, которая находится ближе всего к новой точке. Затем он присваивает метку, принадлежащую этой точке обучающего набора, новой точке данных.



$k$  в методе  $k$  ближайших соседей означает, что вместо того, чтобы использовать лишь ближайшего соседа новой точки данных, мы в ходе обучения можем рассмотреть любое фиксированное число ( $k$ ) соседей (например, рассмотреть ближайшие три или пять соседей). Тогда мы можем сделать прогноз для точки данных, используя класс, которому принадлежит большинство ее соседей. Подробнее мы поговорим об этом в главе 2, а в данный момент мы будем использовать только одного соседа.

В `scikit-learn` все модели машинного обучения реализованы в собственных классах, называемых классами `Estimator`. Алгоритм классификации на основе метода  $k$  ближайших соседей реализован в классификаторе `KNeighborsClassifier` модуля `neighbors`. Прежде чем использовать эту модель, нам нужно создать объект-экземпляр класса. Это произойдет, когда мы зададим параметры модели. Самым важным параметром `KNeighborsClassifier` является количество соседей, которые мы установим равным 1:

```
In[25]:
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

Объект `knn` включает в себя алгоритм, который будет использоваться для построения модели на обучающих данных, а также алгоритм, который сгенерирует прогнозы для новых точек данных. Он также будет содержать информацию, которую алгоритм извлек из обучающих данных. В случае с `KNeighborsClassifier` он будет просто хранить обучающий набор.

Для построения модели на обучающем наборе, мы вызываем метод `fit` объекта `knn`, который принимает в качестве аргументов массив NumPy `X_train`, содержащий обучающие данные, и массив NumPy `y_train`, соответствующий обучающим меткам:

```
In[26]:
knn.fit(X_train, y_train)

Out[26]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=1, p=2,
                    weights='uniform')
```

Метод `fit` возвращает сам объект `knn` (и изменяет его), таким образом, мы получаем строковое представление нашего классификатора. Оно показывает нам, какие параметры были использованы при создании модели. Почти все параметры имеют значения по умолчанию, но вы также можете обнаружить параметр `n_neighbors=1`, заданный нами. Большинство моделей в `scikit-learn` имеют массу параметров, но большая часть из них связана с оптимизацией скорости вычислений или

предназначена для особых случаев использования. Вам не нужно беспокоиться о других параметрах, приведенных здесь. Вывод модели в `scikit-learn` может быть очень длинным, но не нужно пугаться его. Мы рассмотрим все важные параметры в главе 2. В оставшейся части этой книги мы не будем приводить вывод метода `fit`, поскольку он не содержит никакой новой информации.

## Получение прогнозов

Теперь мы можем получить прогнозы, применив эту модель к новым данным, по которым мы еще не знаем правильные метки. Представьте, что мы нашли в дикой природе ирис с длиной чашелистика 5 см, шириной чашелистика 2.9 см, длиной лепестка 1 см и шириной лепестка 0.2 см. К какому сорту ириса нужно отнести этот цветок? Мы можем поместить эти данные в массив NumPy, снова вычисляя форму массива, т.е. количество примеров (1), умноженное на количество признаков (4):

```
In[27]:
X_new = np.array([[5, 2.9, 1, 0.2]])
print("форма массива X_new: {}".format(X_new.shape))
```

```
Out[27]:
форма массива X_new: (1, 4)
```

Обратите внимание, что мы записали измерения по одному цветку в двумерный массив NumPy, поскольку `scikit-learn` работает с двумерными массивами данных.

Чтобы сделать прогноз, мы вызываем метод `predict` объекта `knn`:

```
In[28]:
prediction = knn.predict(X_new)
print("Прогноз: {}".format(prediction))
print("Спрогнозированная метка: {}".format(
    iris_dataset['target_names'][prediction]))
```

```
Out[28]:
Прогноз: [0]
Спрогнозированная метка: ['setosa']
```

Наша модель предсказывает, что этот новый цветок ириса принадлежит к классу 0, что означает сорт *setosa*. Но как узнать, можем ли мы доверять нашей модели? Правильный сорт ириса для этого примера нам неизвестен, а ведь именно получение правильных прогнозов и является главной задачей построения модели!



## Оценка качества модели

Это тот самый момент, когда нам понадобится созданный ранее тестовый набор. Эти данные не использовались для построения модели, но мы знаем правильные сорта для каждого ириса в тестовом наборе.

Таким образом, мы можем сделать прогноз для каждого ириса в тестовом наборе и сравнить его с фактической меткой (уже известным сортом). Мы можем оценить качество модели, вычислив *правильность* (*accuracy*) – процент цветов, для которых модель правильно спрогнозировала сорта:

```
In[29]:
y_pred = knn.predict(X_test)
print("Прогнозы для тестового набора:\n {}".format(y_pred))

Out[29]:
Прогнозы для тестового набора:
[2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0 2]

In[30]:
print("Правильность на тестовом наборе: {:.2f}".format(np.mean(y_pred == y_test)))

Out[30]:
Правильность на тестовом наборе: 0.97
```

Кроме того, мы можем использовать метод `score` объекта `knn`, который вычисляет правильность модели для тестового набора:

```
In[31]:
print("Правильность на тестовом наборе: {:.2f}".format(knn.score(X_test, y_test)))

Out[31]:
Правильность на тестовом наборе: 0.97
```

Правильность этой модели для тестового набора составляет около 0.97, что означает, что мы дали правильный прогноз для 97% ирисов в тестовом наборе. При некоторых математических допущениях, это означает, что мы можем ожидать, что наша модель в 97% случаев даст правильный прогноз для новых ирисов. Для нашего ботаника-любителя этот высокий уровень правильности означает, что наша модель может быть достаточно надежной в использовании. В следующих главах мы обсудим, как можно улучшить эффективность модели, и с какими подводными камнями можно столкнуться при настройке модели.

## Выводы и перспективы

Давайте подытожим то, чему мы научились в этой главе. Мы начали с краткого введения в машинного обучение и сфер его применения, затем обсудили различие между обучением с учителем и обучением без учителя и дали краткий обзор инструментов, которые мы будем

использовать в этой книге. Затем мы сформулировали задачу классификации ирисов на основе проведенных измерений характеристик цветов. Мы использовали набор данных, в котором эксперт уже предварительно классифицировал ирисы для построения модели, таким образом, мы решали задачу обучения с учителем. Было три возможных сорта ирисов – *setosa*, *versicolor* и *virginica*, что делало нашу задачу задачей 3-классовой классификации. В задаче классификации возможные сорта ирисов называются *классами* (*classes*) а сами названия сортов – *метками* (*labels*).

Набор данных Iris состоит из двух массивов NumPy: один содержит данные и в **scikit-learn** обозначается как *X*, другой содержит правильные или нужные ответы и обозначается как *y*. Массив *X* представляет собой двумерный массив признаков, в котором одна строка соответствует одной точке данных, а один столбец – одному признаку. Массив *y* представляет собой одномерный массив, который для каждого примера содержит метку класса, целое число от 0 до 2.

Мы разделили наш набор данных на *обучающий набор* (*training set*), чтобы построить нашу модель, а также *тестовый набор* (*test set*), чтобы оценить, насколько хорошо наша модель будет классифицировать новые, ранее неизвестные ей данные.

Мы выбрали алгоритм классификации *k* ближайших соседей, который генерирует прогноз для новой точки данных, рассматривая ее ближайшего соседа(ей) в обучающем наборе. Все это реализовано в классе **KNeighborsClassifier**, который содержит алгоритм, строящий модель, а также алгоритм, который дает прогнозы, используя построенную модель. Мы создали объект-экземпляр класса, задав параметры. Затем мы построили модель, вызвав метод **fit** и передав обучающие данные (*X\_train*) и обучающие ответы (*y\_train*) в качестве параметров. Мы оценили качество модели с использованием метода **score**, который вычисляет правильность модели. Мы применили метод **score** к тестовым данным и тестовым ответам и обнаружили, что наша модель демонстрирует правильность около 97%. Это означает, что модель выдает правильные прогнозы для 97% наблюдений тестового набора.

Это убедило нас в том, что модель можно применить к новым данным (в нашем примере это измерения характеристик новых цветов), и мы надеемся, что эта модель даст правильные прогнозы в 97% случаев.

Ниже приводится краткое изложение программного кода, необходимого для всей процедуры обучения и оценки модели:

```
In[32]:
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)

knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
```

```
print("Правильность на тестовом наборе: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[32]:

Правильность на тестовом наборе: 0.97

Этот фрагмент содержит базовый код, необходимый для применения любого алгоритма машинного обучения с помощью **scikit-learn**. Методы **fit**, **predict** и **score** являются общими для моделей контролируемого обучения в **scikit-learn** и, используя принципы, приведенные в этой главе, вы можете применить эти модели для решения различных задач машинного обучения. В следующей главе мы подробнее рассмотрим различные модели машинного обучения с учителем, имеющиеся в **scikit-learn**, и расскажем, как успешно применять их.

## ГЛАВА 2. МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ С УЧИТЕЛЕМ

Как мы уже говорили ранее, машинное обучение с учителем является одним из наиболее часто используемых и успешных видов машинного обучения. В этой главе мы более подробно расскажем о машинном обучении с учителем и объясним работу нескольких популярных алгоритмов. Мы уже разбирали применение машинного обучения с учителем в главе 1: классификацию ирисов по нескольким сортам с использованием измерений физических характеристик цветов.

Вспомним, что обучение с учителем используется всякий раз, когда мы хотим предсказать определенный результат (ответ) по данному объекту, и у нас есть пары объект-ответ. Мы строим модель машинного обучения на основе этих пар объект-ответ, которые составляют наш обучающий набор данных. Наша цель состоит в том, чтобы получить точные прогнозы для новых, никогда ранее не встречавшихся данных. Машинное обучение с учителем часто требует вмешательства человека, чтобы получить обучающий набор данных, но потом оно автоматизирует и часто ускоряет решение трудоемких или неосуществимых задач.

### Классификация и регрессия

Есть два основные задачи машинного обучения с учителем: *классификация (classification)* и *регрессия (regression)*.

Цель классификации состоит в том, чтобы спрогнозировать *метку класса (class label)*, которая представляет собой выбор из заранее определенного списка возможных вариантов. В главе 1 мы использовали пример классификации ирисов, когда относили цветок к одному из трех возможных сортов. Классификация иногда разделяется на *бинарную классификацию (binary classification)*, которая является частным случаем разделения на два класса, и *мультиклассовую классификацию (multiclass classification)*, когда в классификации участвует более двух классов. Бинарную классификацию можно представить как попытку ответить на поставленный вопрос в формате «да/нет». Классификация электронных писем на спам и не-спам является примером бинарной классификации. В данной задаче бинарной классификации ответ «да/нет» дается на вопрос «является ли это электронное письмо спамом?»



В бинарной классификации мы часто говорим о том, что один класс является *положительным* (*positive*) классом, а другой класс является *отрицательным* (*negative*) классом. При этом «положительный» означает здесь не наличие выгоды (ценности), а объект исследования. Таким образом, при поиске спама, положительным классом может быть класс «спам». Вопрос о том, какой из этих двух классов будет положительным, часто субъективен и зависит от предметной области исследования.

С другой стороны, пример классификации ирисов является примером мультиклассовой классификации. Еще один пример – прогнозирование языка веб-сайта. Классами здесь будет заранее определенный список возможных языков.

Цель регрессии состоит в том, чтобы спрогнозировать непрерывное число или *число с плавающей точкой* (*floating-point number*), если использовать термины программирования, или *вещественное число* (*real number*), если говорить языком математических терминов. Прогнозирование годового дохода человека в зависимости от его образования, возраста и места жительства является примером регрессионной задачи. Прогнозируемое значение дохода представляет собой *сумму* (*amount*) и может быть любым числом в заданном диапазоне. Другой пример регрессионной задачи – прогнозирование объема урожая зерна на ферме в зависимости от таких атрибутов, как объем предыдущего урожая, погода, и количество сотрудников, работающих на ферме. И снова объем урожая может быть любым числом.

Самый простой способ отличить классификацию от регрессии – спросить, заложена ли в полученном ответе определенная непрерывность (преемственность). Если полученные результаты непрерывно связаны друг с другом, то решаемая задача является задачей регрессии. Возьмем прогнозирование годового дохода. Здесь ясно видна непрерывность ответа. Разница между годовым доходом в 40000\$ или 40001\$ не существенна, хотя речь идет о разных денежных суммах. Если наш алгоритм предсказывает 39999\$ или 40001\$, в то время как он должен предсказать 40000\$ (реальное значение годового дохода), мы не будем настаивать на том, что разница существенна. Наоборот, в задаче распознавании языка веб-сайта (задаче классификации) ответы четко определены. Контент сайта может быть написан либо на одном конкретном языке, либо на другом. Между языками нет непрерывной связи, не существует языка, находящегося между английским и французским.<sup>6</sup>

---

<sup>6</sup> Мы просим прощения у лингвистов за упрощенное представление о языках как отдельных и фиксированных объектах.

## Обобщающая способность, переобучение и недообучение

В машинном обучении с учителем нам нужно построить модель на обучающих данных, а затем получить точные прогнозы для новых, еще не встречавшихся нам данных, которые имеют те же самые характеристики, что и использованный нами обучающий набор. Если модель может выдавать точные прогнозы на ранее не встречавшихся данных, мы говорим, что модель обладает способностью *обобщать* (*generalize*) результат на тестовые данные. Нам необходимо построить модель, которая будет обладать максимальной обобщающей способностью.

Обычно мы строим модель таким образом, чтобы она давала точные прогнозы на обучающем наборе. Если обучающий и тестовый наборы имеют много общего между собой, можно ожидать, что модель будет точной и на тестовом наборе. Однако в некоторых случаях этого не происходит. Например, если мы строим очень сложные модели, необходимо помнить, что на обучающей выборке можно получить произвольную правильность.

Давайте взглянем на выдуманный пример, чтобы проиллюстрировать этот тезис. Скажем, начинающий специалист по анализу данных хочет спрогнозировать покупку клиентом лодки на основе записей о клиентах, которые ранее приобрели лодку, и клиентах, которые не заинтересованы в покупке лодки.<sup>7</sup> Цель состоит в том, чтобы отправить рекламные письма клиентам, которые, вероятно, хотят совершить покупку, и не беспокоить клиентов, не заинтересованных в покупке.

Предположим, у нас есть записи о клиентах, приведенные в таблице 2.1.

---

<sup>7</sup> В действительности это сложная проблема. Хотя мы знаем, что остальные клиенты еще не купили у нас лодку, они могли купить ее у кого-то еще, или они могут откладывать сбережения и планировать покупку лодки в будущем.

Возраст	Количество автомобилей в собственности	Есть собственный дом	Количество детей	Семейное положение	Есть собака	Купил лодку
66	1	да	2	вдовец	нет	да
52	2	да	3	женат	нет	да
22	0	нет	0	женат	да	нет
25	1	нет	1	холост	нет	нет
44	0	нет	2	разведен	да	нет
39	1	да	2	женат	да	нет
26	1	нет	2	холост	нет	нет
40	3	да	1	женат	да	нет
53	2	да	2	разведен	нет	да
64	2	да	3	разведен	нет	нет
58	2	да	2	женат	да	да
33	1	нет	1	холост	нет	нет

**Таблица 2.1** Пример данных о клиентах

Поработав с данными некоторое время, наш начинающий специалист формулирует следующее правило «если клиент старше 45 лет, у него менее трех детей, либо у него трое и он женат, то он скорее всего купит лодку». Если спросить, насколько хорошо работает это правило, наш специалист воскликнет «оно дает 100%-ную правильность!» И в самом деле, для данных, приведенных в таблице, это правило демонстрирует идеальную правильность. Мы могли бы сформулировать массу правил, объясняющих покупку лодки. Значения возраста появляются в наборе данных лишь один раз, таким образом, мы могли бы сказать, что люди в возрасте 66, 52, 53 и 58 лет хотят купить лодку, тогда как все остальные не собираются ее покупать. Несмотря на то что можно сформулировать множество правил, которые хорошо работают для этих наблюдений, следует помнить о том, что нам не интересны прогнозы для этого набора данных, мы уже знаем ответы для этих клиентов. Мы хотим знать, могут ли новые клиенты купить лодку. Поэтому нам нужно правило, которое будет хорошо работать для новых клиентов, и достижение 100%-ной правильности на обучающей выборке не поможет нам в этом. Нельзя ожидать, что правило, сформулированное нашим специалистом, будет так же хорошо работать и для новых клиентов. Похоже, с этим у нас сложность, ведь у нас мало данных. Например, часть правила «либо трое и он женат» сформулировано по одному клиенту.

Единственный показатель качества работы алгоритма на новых данных – это использование тестового набора. Однако интуитивно<sup>8</sup> мы ожидаем, что простые модели должны лучше обобщать результат на новые данные. Если бы правило звучало «люди старше 50 лет хотят купить лодку» и оно объясняло бы поведение всех клиентов, мы

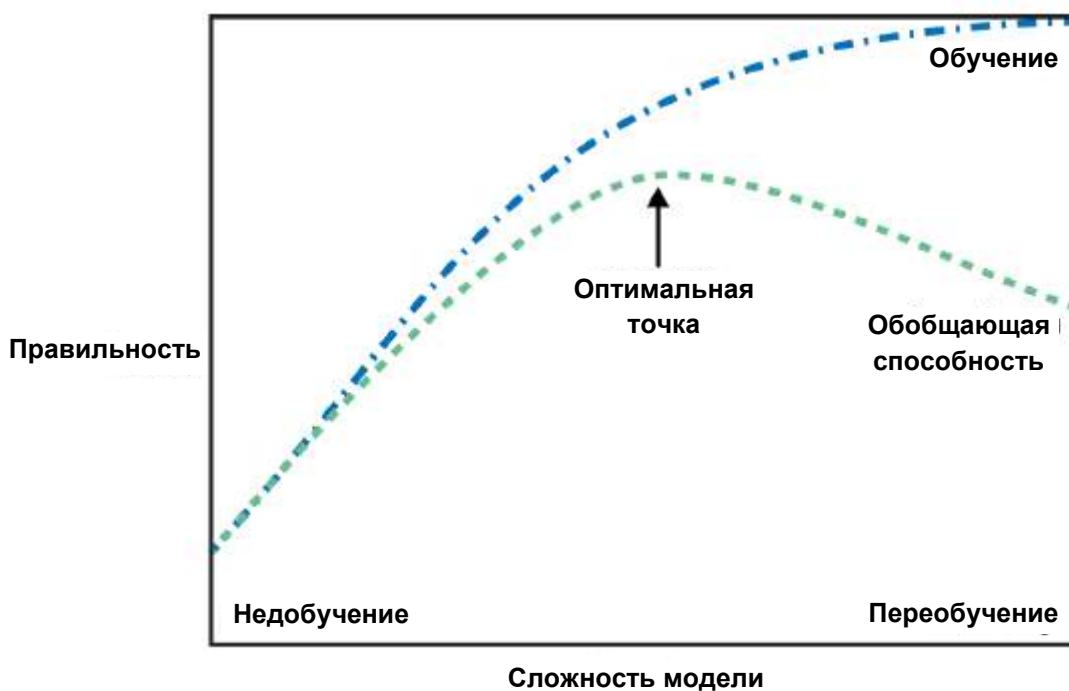
<sup>8</sup> И совершенно обоснованно с точки зрения математики.

доверяли бы ему больше, чем правилу, которое помимо возраста включало бы количество детей и семейное положение. Поэтому нам всегда нужно искать самую простую модель. Построение модели, которая слишком сложна для имеющегося у нас объема информации (что и сделал наш начинающий специалист по анализу данных), называется *переобучением* (*overfitting*). Переобучение происходит, когда ваша модель слишком точно подстраивается под особенности обучающего набора и вы получаете модель, которая хорошо работает на обучающем наборе, но не умеет обобщать результат на новые данные. С другой стороны, если ваша модель слишком проста, скажем, вы сформулировали правило «все, у кого есть собственный дом, покупает лодку», вы, возможно, не смогли охватить все многообразие и изменчивость данных, и ваша модель будет плохо работать даже на обучающем наборе. Выбор слишком простой модели называется *недообучением* (*underfitting*).

Чем сложнее модель, тем лучше она будет работать на обучающих данных. Однако, если наша модель становится слишком сложной, мы начинаем уделять слишком много внимания каждой отдельной точке данных в нашем обучающем наборе, и эта модель не будет хорошо обобщать результат на новые данные.

Существует оптимальная точка, которая позволяет получить наилучшую обобщающую способность. Собственно это и есть модель, которую нам нужно найти.

Компромисс между переобучением и недообучением показан на рис. 2.1.



**Рис. 2.1** Компромисс между сложностью модели и правильностью на обучающей и тестовой выборках



## Взаимосвязь между сложностью модели и размером набора данных

Важно отметить, что сложность модели тесно связана с изменчивостью входных данных, содержащихся в вашем обучающем наборе: чем больше разнообразие точек данных в вашем наборе, тем более сложную модель можно использовать, не беспокоясь о переобучении. Обычно больший объем данных дает большее разнообразие, таким образом, большие наборы данных позволяют строить более сложные модели. Однако простое дублирование одних и тех же точек данных или сбор очень похожих данных здесь не поможет.

Возвращаясь к продажам лодок, можно сказать, что если бы у нас было более 10000 строк данных о клиентах и все они подчинялись бы правилу «если клиент старше 45 лет, у него менее трех детей, либо трое и он женат, то он скорее всего купит лодку», мы бы с гораздо большей вероятностью поверили в это правило, чем если бы оно было сформулировано лишь по 12 строкам таблицы 2.1.

Увеличение объема данных и построение более сложных моделей часто творят чудеса при решении задач машинного обучения с учителем. В этой книге мы сосредоточимся на работе с данными фиксированного размера. В действительности вы, как правило, сами можете определить объем собираемых данных, и это может оказаться более полезным, чем корректировка и настройка вашей модели. Никогда не стоит недооценивать преимущества увеличения объема данных.

## Алгоритмы машинного обучения с учителем

Теперь мы рассмотрим наиболее популярные алгоритмы машинного обучения и объясним, как они обучаются на основе данных и как вычисляют прогнозы. Кроме того, мы расскажем о том, как принцип сложности реализуется для каждой из этих моделей, и покажем, как тот или иной алгоритм строит модель. Мы рассмотрим преимущества и недостатки каждого алгоритма, а также расскажем о том, применительно к каким данным лучше всего использовать тот или иной алгоритм. Мы также объясним значение наиболее важных параметров и опций. Многие алгоритмы имеют опции классификации и регрессии, поэтому мы опишем обе опции.

Необязательно детально вчитываться в описание каждого алгоритма, но понимание модели даст вам лучшее представление о различных способах работы алгоритмов машинного обучения. Кроме того, эту главу можно использовать в качестве справочного руководства, и вы можете вернуться к ней, если не знаете, как работает тот или иной алгоритм.

## Некоторые наборы данных

Для иллюстрации различных алгоритмов мы будем использовать несколько наборов данных. Некоторые наборы данных будут небольшими и синтетическими (то есть выдуманными), призванными подчеркнуть отдельные аспекты алгоритмов. Другие наборы данных будут большими, реальными примерами.

Примером синтетического набора данных для двухклассовой классификации является набор данных **forge**, который содержит два признака. Программный код, приведенный ниже, создает диаграмму рассеяния (рис. 2.2), визуализируя все точки данных в этом наборе. На графике первый признак отложен на оси x, а второй – по оси y. Как это всегда бывает в диаграммах рассеяния, каждая точка данных представлена в виде одного маркера. Цвет и форма маркера указывает на класс, к которому принадлежит точка:

```
In[2]:  
# генерируем набор данных  
X, y = mglearn.datasets.make_forge()  
# строим график для набора данных  
%matplotlib inline  
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)  
plt.legend(["Класс 0", "Класс 1"], loc=4)  
plt.xlabel("Первый признак")  
plt.ylabel("Второй признак")  
print("форма массива X: {}".format(X.shape))
```

```
Out[2]:  
форма массива X: (26, 2)
```

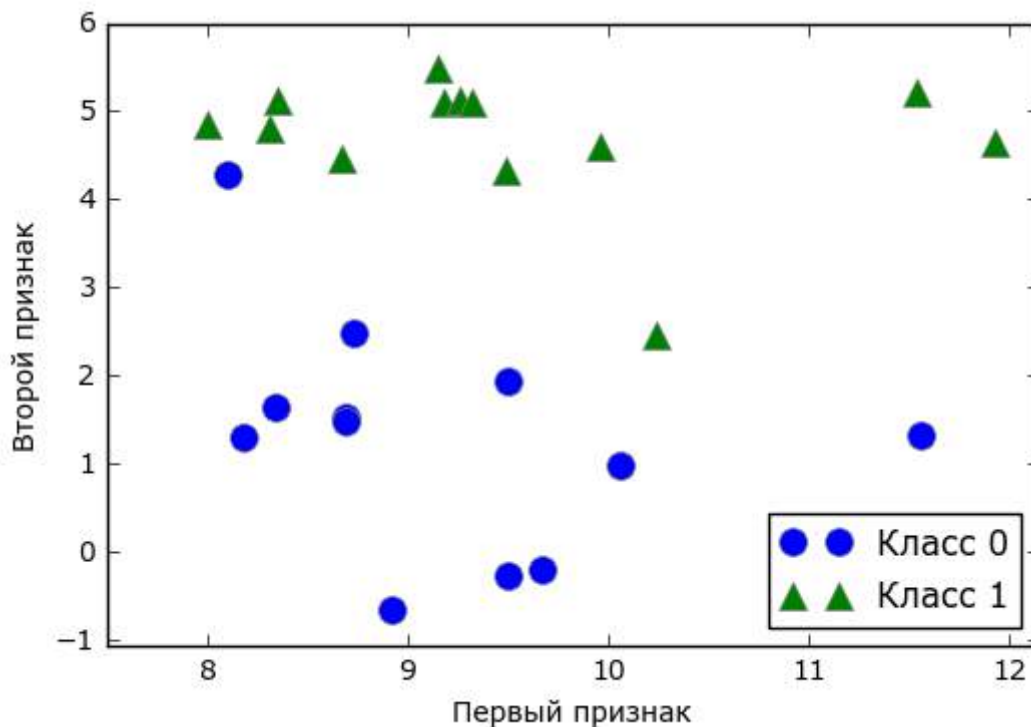
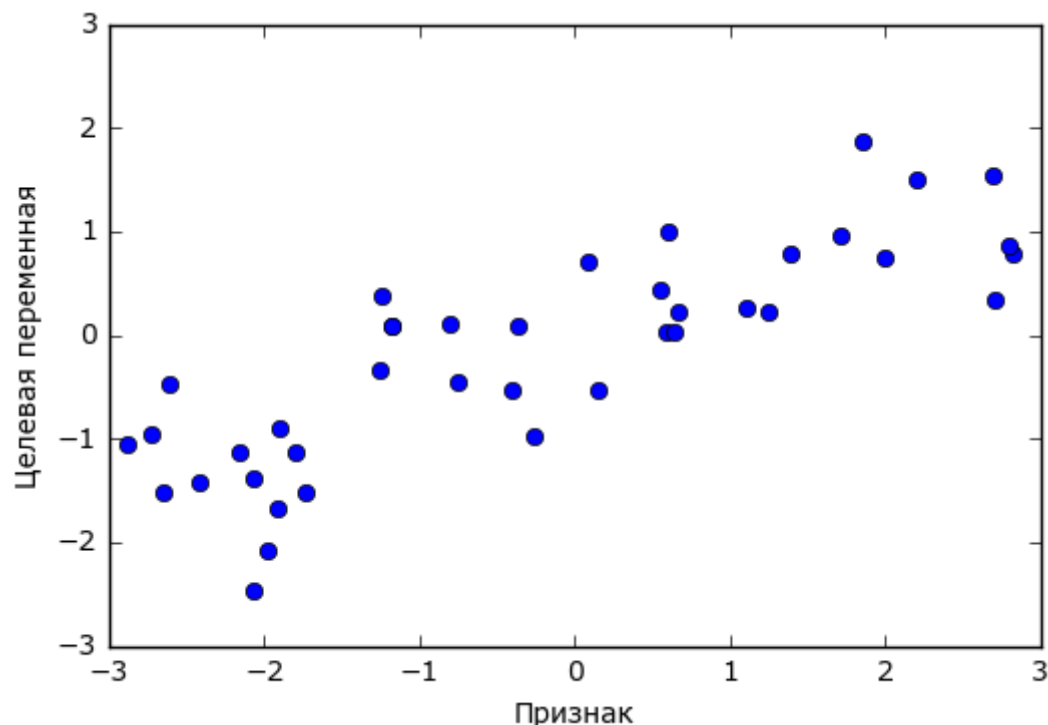


Рис. 2.2 Диаграмма рассеяния для набора данных **forge**

Как видно из сводки по массиву `X`, этот набор состоит из 26 точек данных и 2 признаков. Для иллюстрации алгоритмов регрессии, мы воспользуемся синтетическим набором `wave`. Набор данных имеет единственный входной признак и непрерывную целевую переменную или *отклик* (*response*), который мы хотим смоделировать. На рисунке, построенном здесь (рис. 2.3), по оси `x` располагается единственный признак, а по оси `y` – целевая переменная (ответ).

```
In[3]:
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Признак")
plt.ylabel("Целевая переменная")
```



**Рис. 2.3** График для набора данных `wave`, по оси `x` отложен признак, по оси `y` – целевая переменная

Мы используем эти очень простые, низкоразмерные наборы данных, потому что их легко визуализировать – печатная страница имеет два измерения, и данные, которые содержат более двух признаков, графически представить трудно. Вывод, полученный для набора с небольшим числом признаков или *низкоразмерном* (*low-dimensional*) наборе, возможно, не подтвердится для набора данных с большим количеством признаков или *высокоразмерного* (*high-dimensional*) набора. Если вы помните об этом, проверка алгоритма на низкоразмерном наборе данных может оказаться очень полезной.

Мы дополним эти небольшие синтетические наборы данных двумя реальными наборами, которые включены в `scikit-learn`. Один из них – набор данных по раку молочной железы Университета Висконсин (`cancer` для краткости), в котором записаны клинические измерения опухолей молочной железы. Каждая опухоль обозначается как «benign» («доброкачественная», для неагрессивных опухолей) или `malignant` («злокачественная», для раковых опухолей), и задача состоит в том, чтобы на основании измерений ткани дать прогноз, является ли опухоль злокачественной.

Данные можно загрузить из `scikit-learn` с помощью функции `load_breast_cancer`:

```
In[4]:
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("Ключи cancer(): \n{}".format(cancer.keys()))

Out[4]:
Ключи cancer():
dict_keys(['feature_names', 'data', 'DESCR', 'target', 'target_names'])
```



Наборы данных, которые включены в `scikit-learn`, обычно хранятся в виде объектов `Bunch`, которые содержат некоторую информацию о наборе данных, а также фактические данные. Все, что вам нужно знать об объектах `Bunch` – это то, что они похожи на словари, с тем преимуществом, что вы можете прочитать значения, используя точку (`bunch.key` вместо `bunch['key']`)

Набор данных включает 569 точек данных и 30 признаков.

```
In[5]:
print("Форма массива data для набора cancer: {}".format(cancer.data.shape))

Out[5]:
Форма массива data для набора cancer: (569, 30)
```

Из 569 точек данных 212 помечены как злокачественные, а 357 как доброкачественные.

```
In[6]:
print("Количество примеров для каждого класса:\n{}".format(
    {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))

Out[6]:
Количество примеров для каждого класса:
{'benign': 357, 'malignant': 212}
```

Чтобы получить содержательное описание каждого признака, взглянем на атрибут `feature_names`:

```
In[7]:
print("Имена признаков:\n{}".format(cancer.feature_names))
```

Out[7]:

Имена признаков:

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

Если вам интересно, то более подробную информацию о данных можно получить, прочитав `cancer.DESCR`.

Кроме того, для задач регрессии мы будем использовать реальный набор данных – набор данных Boston Housing. Задача, связанная с этим набором данных, заключается в том, чтобы спрогнозировать медианную стоимость домов в нескольких районах Бостона в 1970-е годы на основе такой информации, как уровень преступности, близость к Charles River, удаленность от радиальных магистралей и т.д. Набор данных содержит 506 точек данных и 13 признаков:

In[8]:

```
from sklearn.datasets import load_boston
boston = load_boston()
print("форма массива data для набора boston: {}".format(boston.data.shape))
```

Out[8]:

форма массива data для набора boston: (506, 13)

Опять же, вы можете получить более подробную информацию о наборе данных, прочитав атрибут `boston.DESCR`. В данном случае мы более детально проанализируем набор данных, учтя не только 13 измерений в качестве входных признаков, но и приняв во внимание все *взаимодействия* (*interactions*) между признаками. Иными словами, мы будем учитывать в качестве признаков не только уровень преступности и удаленность от радиальных магистралей по отдельности, но и взаимодействие уровень преступности–удаленность от радиальных магистралей. Включение производных признаков называется *конструированием признаков* (*feature engineering*), которое мы рассмотрим более подробно в главе 4. Набор данных с производными признаками можно загрузить с помощью функции `load_extended_boston`:

In[9]:

```
X, y = mglearn.datasets.load_extended_boston()
print("форма массива X: {}".format(X.shape))
```

Out[9]:

форма массива X: (506, 104)

Полученные 104 признака – 13 исходных признаков плюс 91 производный признак.

Мы будем использовать эти наборы данных, чтобы объяснить и проиллюстрировать свойства различных алгоритмов машинного обучения. Однако сейчас давайте перейдем к самим алгоритмам. Во-первых, мы вернемся к алгоритму  $k$  ближайших соседей, который рассматривали в предыдущей главе.

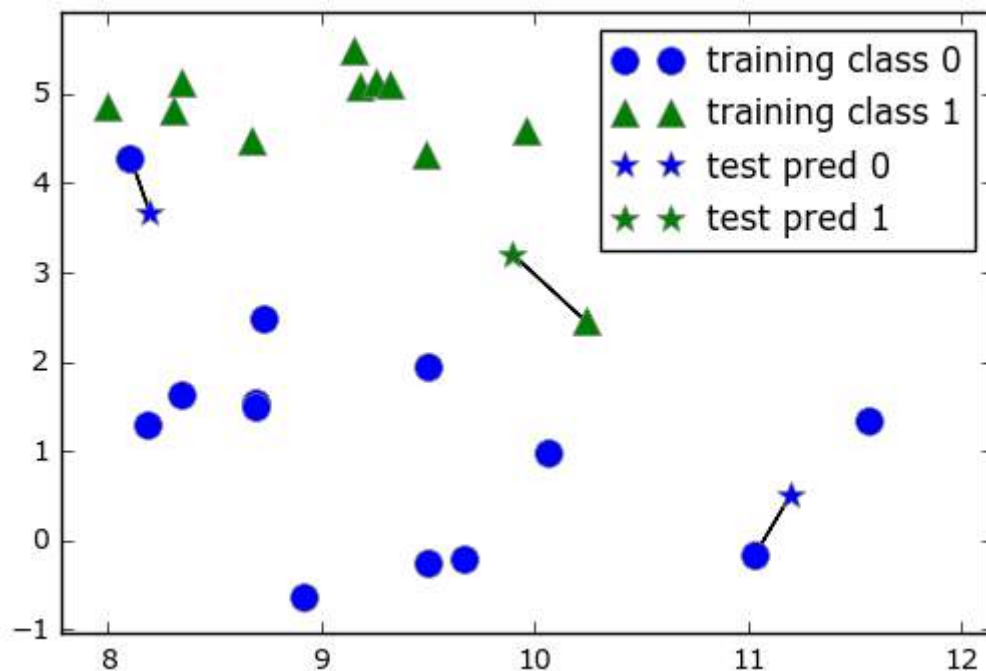
## Метод $k$ ближайших соседей

Алгоритм  $k$  ближайших соседей, возможно, является самым простым алгоритмом машинного обучения. Построение модели заключается в запоминании обучающего набора данных. Для того, чтобы сделать прогноз для новой точки данных, алгоритм находит ближайшие к ней точки обучающего набора, то есть находит «ближайших соседей».

### Классификация с помощью $k$ соседей

В простейшем варианте алгоритм  $k$  ближайших соседей рассматривает лишь одного ближайшего соседа – точку обучающего набора, ближе всего расположенную к точке, для которой мы хотим получить прогноз. Прогнозом является ответ, уже известный для данной точки обучающего набора. На рис. 2.4 показано решение задачи классификации для набора данных `forge`:

```
In[10]:  
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

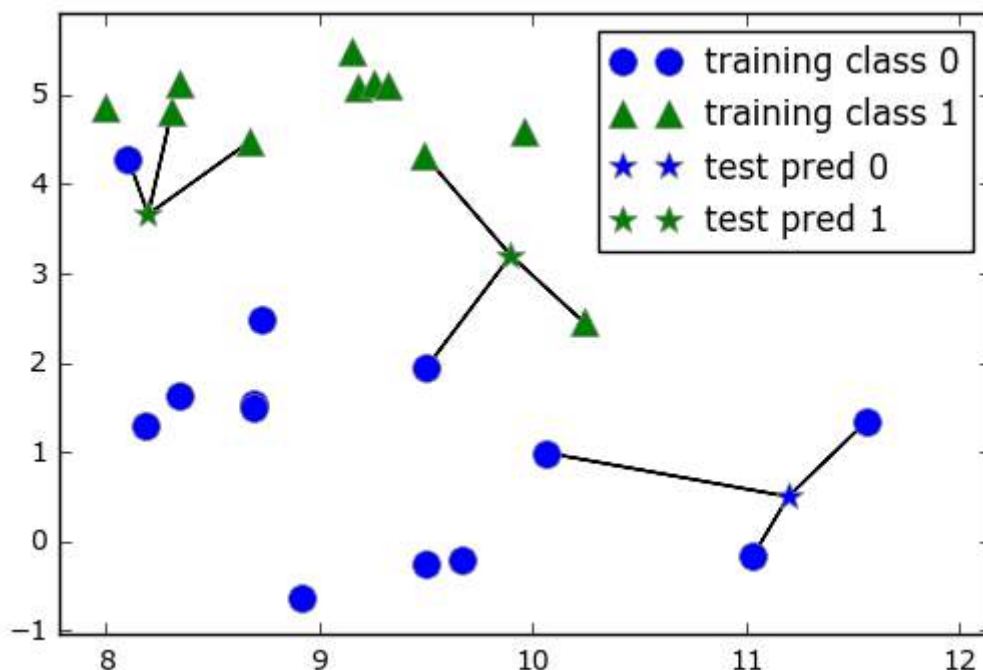


**Рис. 2.4** Прогнозы, полученные для набора данных `forge` с помощью модели одного ближайшего соседа

Здесь мы добавили три новые точки данных, показанные в виде звездочек. Для каждой мы отметили ближайшую точку обучающего набора. Прогноз, который дает алгоритм одного ближайшего соседа – метка этой точки (показана цветом маркера).

Вместо того, чтобы учитывать лишь одного ближайшего соседа, мы можем рассмотреть произвольное количество ( $k$ ) соседей. Отсюда и происходит название алгоритма  $k$  ближайших соседей. Когда мы рассматриваем более одного соседа, для присвоения метки используется *голосование* (*voting*). Это означает, что для каждой точки тестового набора мы подсчитываем количество соседей, относящихся к классу 0, и количество соседей, относящихся к классу 1. Затем мы присваиваем точке тестового набора наиболее часто встречающийся класс: другими словами, мы выбираем класс, набравший большинство среди  $k$  ближайших соседей. В примере, приведенном ниже (рис. 2.5), используются три ближайших соседа:

```
In[11]:  
mglearn.plots.plot_knn_classification(n_neighbors=3)
```



**Рис. 2.5** Прогнозы, полученные для набора данных *forge* с помощью модели трех ближайших соседей

И снова прогнозы переданы цветом маркера. Видно, что прогноз для новой точки данных в верхнем левом углу отличается от прогноза, полученного при использовании одного ближайшего соседа.

Хотя данный рисунок иллюстрирует задачу бинарной классификации, этот метод можно применить к наборам данных с любым количеством классов. В случае мультиклассовой классификации мы подсчитываем



количество соседей, принадлежащих к каждому классу, и снова прогнозируем наиболее часто встречающийся класс.

Теперь давайте посмотрим, как можно применить алгоритм  $k$  ближайших соседей, используя `scikit-learn`. Во-первых, мы разделим наши данные на обучающий и тестовый наборы, чтобы оценить обобщающую способность модели, рассмотренную в главе 1:

```
In[12]:
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Далее выполняем импорт и создаем объект-экземпляр класса, задавая параметры, например, количество соседей, которое будем использовать для классификации. В данном случае мы устанавливаем его равным 3:

```
In[13]:
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Затем подгоняем классификатор, используя обучающий набор. Для `KNeighborsClassifier` это означает запоминание набора данных, таким образом, мы можем вычислить соседей в ходе прогнозирования:

```
In[14]:
clf.fit(X_train, y_train)
```

Чтобы получить прогнозы для тестовых данных, мы вызываем метод `predict`. Для каждой точки тестового набора он вычисляет ее ближайших соседей в обучающем наборе и находит среди них наиболее часто встречающийся класс:

```
In[15]:
print("Прогнозы на тестовом наборе: {}".format(clf.predict(X_test)))

Out[15]:
Прогнозы на тестовом наборе: [1 0 1 0 1 0 0]
```

Для оценки обобщающей способности модели мы вызываем метод `score` с тестовыми данными и тестовыми метками:

```
In[16]:
print("Правильность на тестовом наборе: {:.2f}".format(clf.score(X_test, y_test)))

Out[16]:
Правильность на тестовом наборе: 0.86
```

Мы видим, что наша модель имеет правильность 86%, то есть модель правильно предсказала класс для 86% примеров тестового набора.



## Анализ KNeighborsClassifier

Кроме того, для двумерных массивов данных мы можем показать прогнозы для всех возможных точек тестового набора, разместив в плоскости  $x, y$ . Мы зададим цвет плоскости в соответствии с тем классом, который будет присвоен точке в этой области. Это позволит нам сформировать *границу принятия решений* (*decision boundary*), которая разбивает плоскость на две области: область, где алгоритм присваивает класс 0, и область, где алгоритм присваивает класс 1.

Программный код, приведенный ниже, визуализирует границы принятия решений для одного, трех и девяти соседей (показаны на рис. 2.6):

```
In[17]:
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # создаем объект-классификатор и подгоняем в одной строке
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("количество соседей:{}".format(n_neighbors))
    ax.set_xlabel("признак 0")
    ax.set_ylabel("признак 1")
axes[0].legend(loc=3)
```



**Рис. 2.6** Границы принятия решений, созданные моделью ближайших соседей для различных значений  $n\_neighbors$

На рисунке слева можно увидеть, что использование модели одного ближайшего соседа дает границу принятия решений, которая очень хорошо согласуется с обучающими данными. Увеличение числа соседей приводит к сглаживанию границы принятия решений. Более гладкая граница соответствует более простой модели. Другими словами, использование нескольких соседей соответствует высокой сложности модели (как показано в правой части рис. 2.1), а использование большого количества соседей соответствует низкой сложности модели (как показано в левой части рис. 2.1). Если взять крайний случай, когда количество соседей будет равно количеству точек данных обучающего

набора, каждая точка тестового набора будет иметь одних и тех же соседей (соседями будет все точки обучающего набора) и все прогнозы будут одинаковыми: будет выбран класс, который является наиболее часто встречающимся в обучающем наборе.

Давайте выясним, существует ли взаимосвязь между сложностью модели и обобщающей способностью, о которой мы говорили ранее. Мы сделаем это с помощью реального набора данных Breast Cancer. Начнем с того, что разобьем данные на обучающий и тестовый наборы. Затем мы оценим качество работы модели на обучающем и тестовом наборах с использованием разного количества соседей. Результаты показаны на рис. 2.7:

```
In[18]:
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

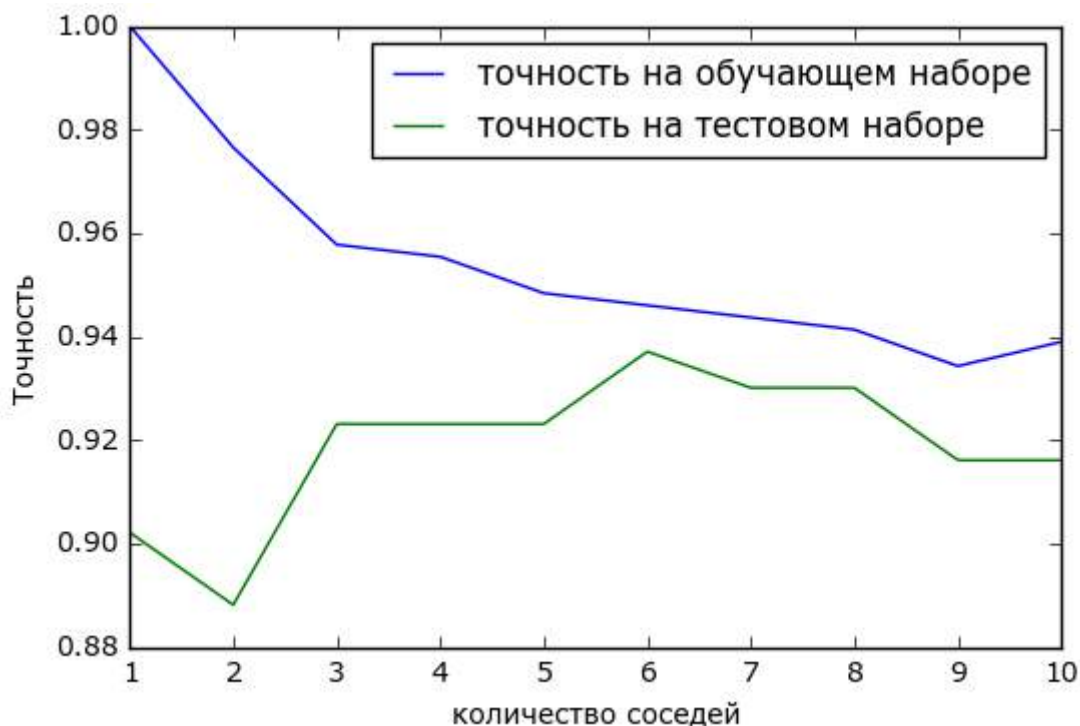
training_accuracy = []
test_accuracy = []
# пробуем n_neighbors от 1 до 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # строим модель
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # записываем правильность на обучающем наборе
    training_accuracy.append(clf.score(X_train, y_train))
    # записываем правильность на тестовом наборе
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="правильность на обучающем наборе")
plt.plot(neighbors_settings, test_accuracy, label="правильность на тестовом наборе")
plt.ylabel("Правильность")
plt.xlabel("количество соседей")
plt.legend()
```

На этом графике по оси y отложена правильность на обучающем наборе и правильность на тестовом наборе, а по оси x – количество соседей. В реальности подобные графики редко бывают гладкими, мы по-прежнему можем увидеть некоторые признаки переобучения и недообучения (обратите внимание, что поскольку использование небольшого количества соседей соответствует более сложной модели, график представляет собой изображение рис. 2.1, зеркально отраженное по горизонтали). При использовании модели одного ближайшего соседа правильность на обучающем наборе идеальна. Однако при использовании большего количества соседей модель становится все проще и правильность на обучающем наборе падает. Правильность на тестовом наборе в случае использования одного соседа ниже, чем при использовании нескольких соседей. Это указывает на то, что

использование одного ближайшего соседа приводит к построению слишком сложной модели. С другой стороны, когда используются 10 соседей, модель становится слишком простой и она работает еще хуже. Оптимальное качество работы модели наблюдается где-то посередине, когда используются шесть соседей. Однако посмотрим на шкалу у. Худшая по качеству модель дает правильность на тестовом наборе около 88%, что по-прежнему может быть приемлемым результатом.

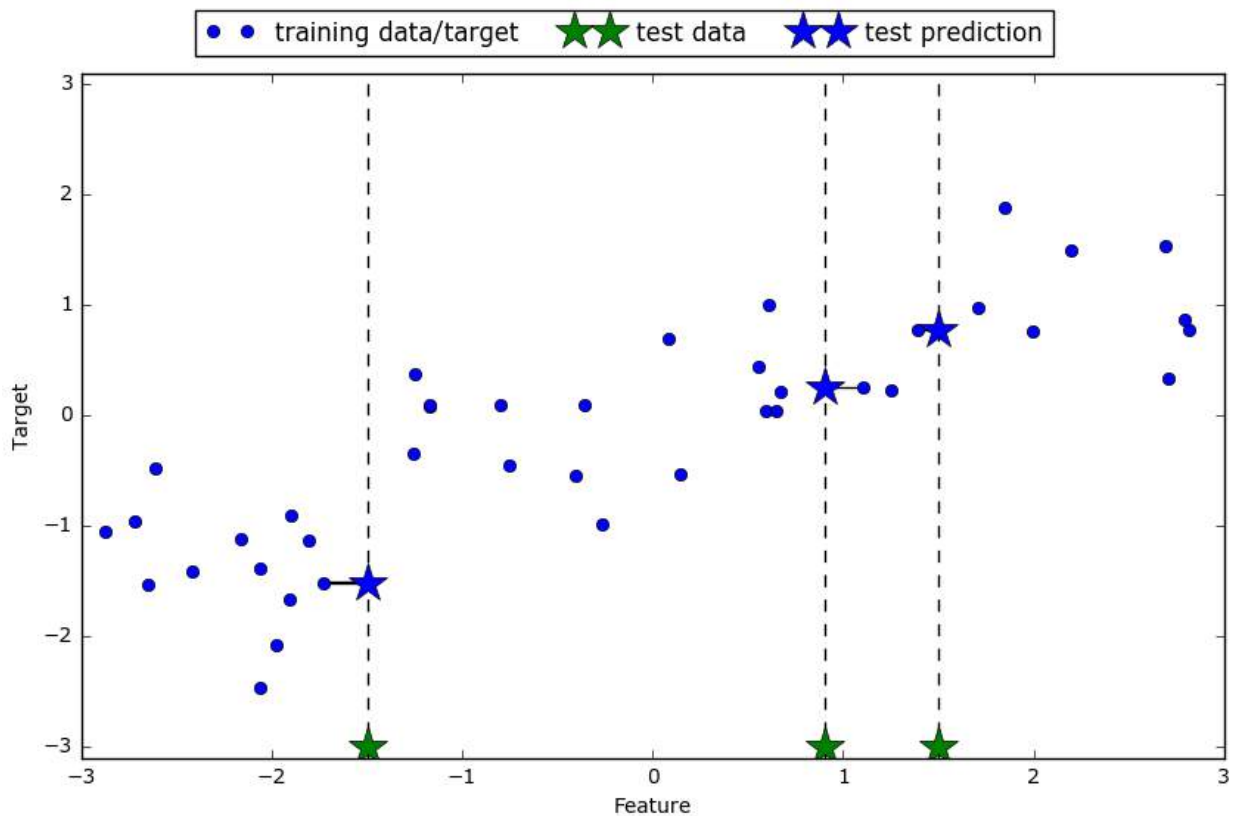


**Рис. 2.7** Сравнение правильности на обучающем и тестовом наборах как функции от количества соседей

### Регрессия k ближайших соседей

Существует также регрессионный вариант алгоритма  $k$  ближайших соседей. Опять же, давайте начнем с рассмотрения одного ближайшего соседа, на этот раз воспользуемся набором данных `wave`. Мы добавили три точки тестового набора в виде зеленых звездочек по оси  $x$ . Прогноз с использованием одного соседа — это целевое значение ближайшего соседа. На рис. 2.8 прогнозы показаны в виде синих звездочек:

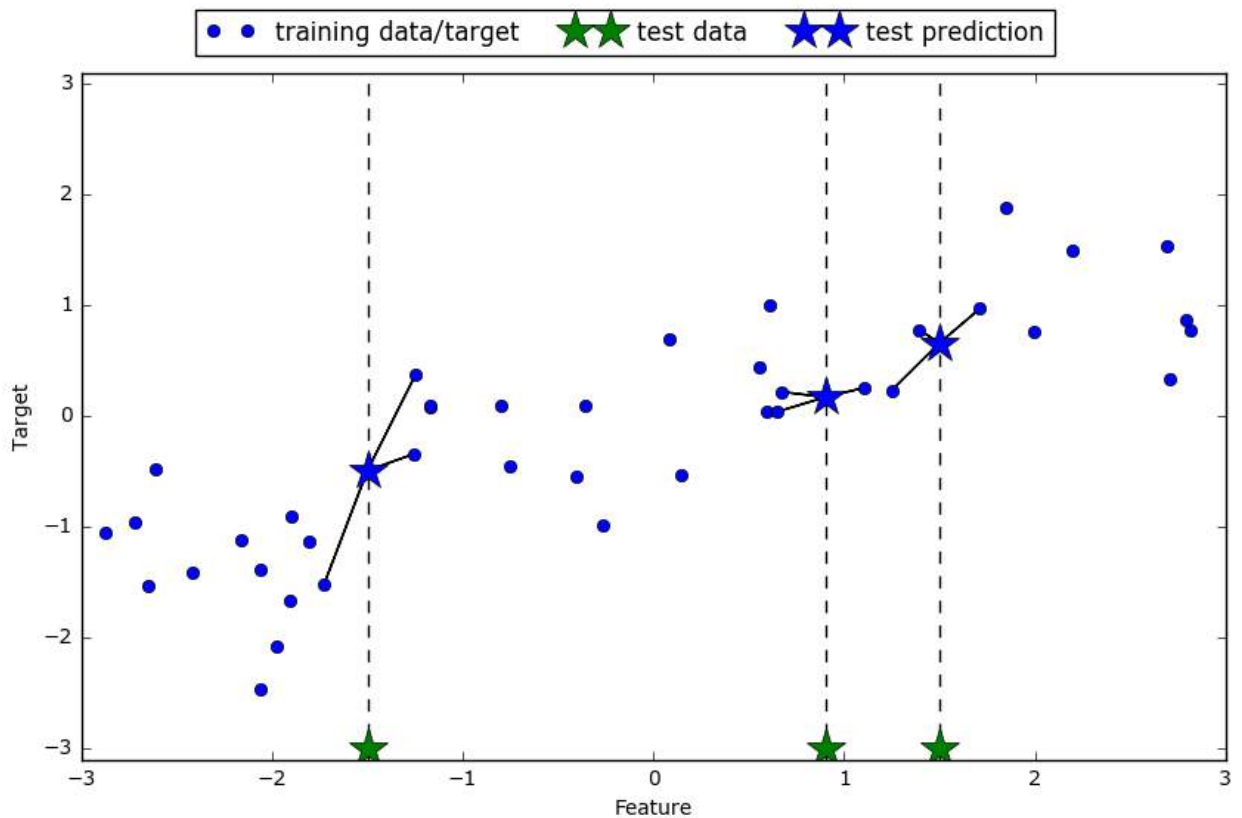
```
In[19]:
mglearn.plots.plot_knn_regression(n_neighbors=1)
```



**Рис. 2.8** Прогнозы, полученные с помощью регрессионной модели одного ближайшего соседа для набора данных wave

И снова для регрессии мы можем использовать большее количество ближайших соседей. При использовании нескольких ближайших соседей прогнозом становится среднее значение соответствующих соседей (рис. 2.9):

**In[20]:**  
`mglearn.plots.plot_knn_regression(n_neighbors=3)`



**Рис. 2.8** Прогнозы, полученные с помощью регрессионной модели трех ближайших соседей для набора данных wave

Алгоритм регрессии  $k$  ближайших соседей реализован в классе `KNeighborsRegressor`. Он используется точно так же, как `KNeighborsClassifier`:

```
In[21]:
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples=40)

# разбиваем набор данных wave на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# создаем экземпляр модели и устанавливаем количество соседей равным 3
reg = KNeighborsRegressor(n_neighbors=3)
# подгоняем модель с использованием обучающих данных и обучающих ответов
reg.fit(X_train, y_train)
```

А теперь получим прогнозы для тестового набора.

```
In[22]:
print("Прогнозы для тестового набора:\n{}".format(reg.predict(X_test)))

Out[22]:
Прогнозы для тестового набора:
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]
```

Кроме того, мы можем оценить качество модели с помощью метода `score`, который для регрессионных моделей возвращает значение  $R^2$ .  $R^2$ , также известный как коэффициент детерминации, является показателем

качества регрессионной модели и принимает значения от 0 до 1. Значение 1 соответствует идеальной прогнозирующей способности, а значение 0 соответствует константе модели, которая лишь предсказывает среднее значение ответов в обучающем наборе, `y_train`:

```
In[23]:
print("R^2 на тестовом наборе: {:.2f}".format(reg.score(X_test, y_test)))
```

```
Out[23]:
R^2 на тестовом наборе: 0.83
```

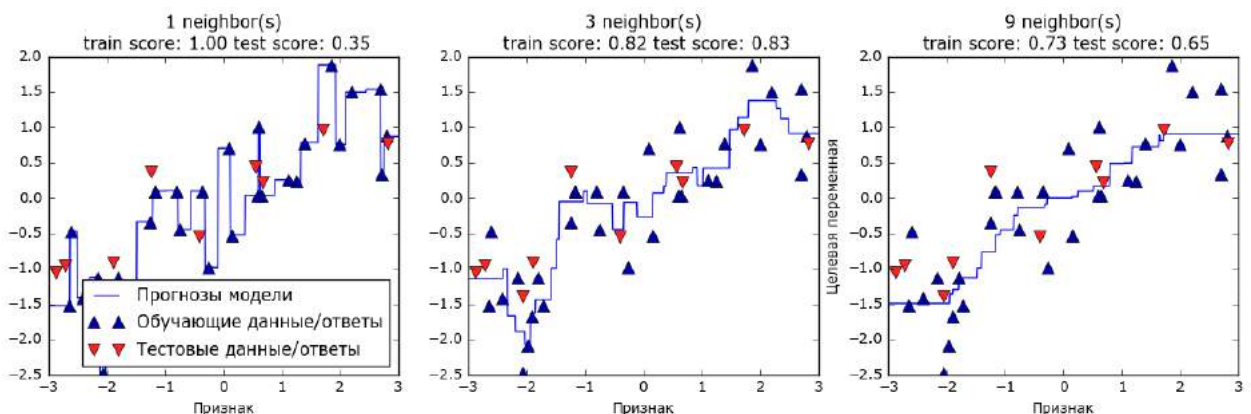
В данном случае значение  $R^2$  составляет 0.83, что указывает на относительно хорошее качество подгонки модели.

## Анализ модели KNeighborsRegressor

Применительно к нашему одномерному массиву данных мы можем увидеть прогнозы для всех возможных значений признаков (рис. 2.10). Для этого мы создаем тестовый набор данных и визуализируем полученные линии прогнозов:

```
In[24]:
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
# создаем 1000 точек данных, равномерно распределенных между -3 и 3
line = np.linspace(-3, 3, 1000).reshape(-1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    # получаем прогнозы, используя 1, 3, и 9 соседей
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)
    reg.fit(X_train, y_train)
    ax.plot(line, reg.predict(line))
    ax.plot(X_train, y_train, '^', c=mglearn.cm2(0), markersize=8)
    ax.plot(X_test, y_test, 'v', c=mglearn.cm2(1), markersize=8)

    ax.set_title(
        "{} neighbor(s)\n train score: {:.2f} test score: {:.2f}".format(
            n_neighbors, reg.score(X_train, y_train),
            reg.score(X_test, y_test)))
    ax.set_xlabel("Признак")
    ax.set_ylabel("Целевая переменная")
axes[0].legend(["Прогнозы модели", "Обучающие данные/ответы",
               "Тестовые данные/ответы"], loc="best")
```



**Рис. 2.10** Сравнение прогнозов, полученных с помощью регрессии ближайших соседей для различных значений `n_neighbors`

Как видно на графике, при использовании лишь одного соседа каждая точка обучающего набора имеет очевидное влияние на прогнозы, и предсказанные значения проходят через все точки данных. Это приводит к очень неустойчивым прогнозам. Увеличение числа соседей приводит к получению более сглаженных прогнозов, но при этом снижается правильность подгонки к обучающим данным.

### Преимущества, недостатки и параметры

В принципе, в классификаторе `KNeighbors` есть два важных параметра: количество соседей и мера расстояния между точками данных. На практике использование небольшого числа соседей (например, 3-5) часто работает хорошо, но вы, конечно, можете самостоятельно настроить этот параметр. Вопрос, связанный с выбором правильной меры расстояния, выходит за рамки этой книги. По умолчанию используется евклидово расстояние, которое хорошо работает во многих ситуациях.

Одним из преимуществ метода ближайших соседей является то, что эту модель очень легко интерпретировать и, как правило, этот метод дает приемлемое качество без необходимости использования большого количества настроек. Он является хорошим базовым алгоритмом, который нужно попробовать в первую очередь, прежде чем рассматривать более сложные методы. Как правило, построение модели ближайших соседей происходит очень быстро, но, когда ваш обучающий набор очень большой (с точки зрения количества характеристик или количества наблюдений) получение прогнозов может занять некоторое время. При использовании алгоритма ближайших соседей важно выполнить предварительную обработку данных (смотрите главу 3). Данный метод не так хорошо работает, когда речь идет о наборах данных с большим количеством признаков (сотни и более), и особенно плохо работает в ситуации, когда подавляющее число признаков в большей части наблюдений имеют нулевые значения (так называемые *разреженные наборы данных* или *sparse datasets*).

Таким образом, несмотря на то что алгоритм ближайших соседей легко интерпретировать, на практике он не часто используется из-за скорости вычислений и его неспособности обрабатывать большое количество признаков. Метод, который мы обсудим ниже, лишен этих недостатков.

## Линейные модели

Линейные модели представляют собой класс моделей, которые широко используются на практике и были предметом детального изучения в течение последних нескольких десятилетий, а их история насчитывает



более ста лет. Линейные модели дают прогноз, используя *линейную функцию* (*linear function*) входных признаков, о которой мы расскажем ниже.

### Линейные модели для регрессии

Для регрессии общая прогнозная формула линейной модели выглядит следующим образом:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Здесь  $x[0]$  по  $x[p]$  обозначают признаки (в данном примере число характеристик равно  $p+1$ ) для отдельной точки данных,  $w$  и  $b$  – параметры модели, оцениваемые в ходе обучения, и  $\hat{y}$  – прогноз, выдаваемый моделью. Для набора данных с одним признаком эта формула имеет вид:

$$\hat{y} = w[0] * x[0] + b$$

Возможно из школьного курса математики вы вспомните, что эта формула – уравнение прямой. Здесь  $x[0]$  является наклоном, а  $b$  – сдвигом по оси  $y$ .<sup>9</sup> Когда используется несколько признаков, регрессионное уравнение содержит параметры наклона для каждого признака. Как вариант, прогнозируемый ответ можно представить в виде взвешенной суммы входных признаков, где веса (которые могут быть отрицательными) задаются элементами  $w$ .

Попробуем вычислить параметры  $w[0]$  и  $b$  для нашего одномерного набора данных `wave` с помощью следующей строки программного кода (см. рис. 2.11):

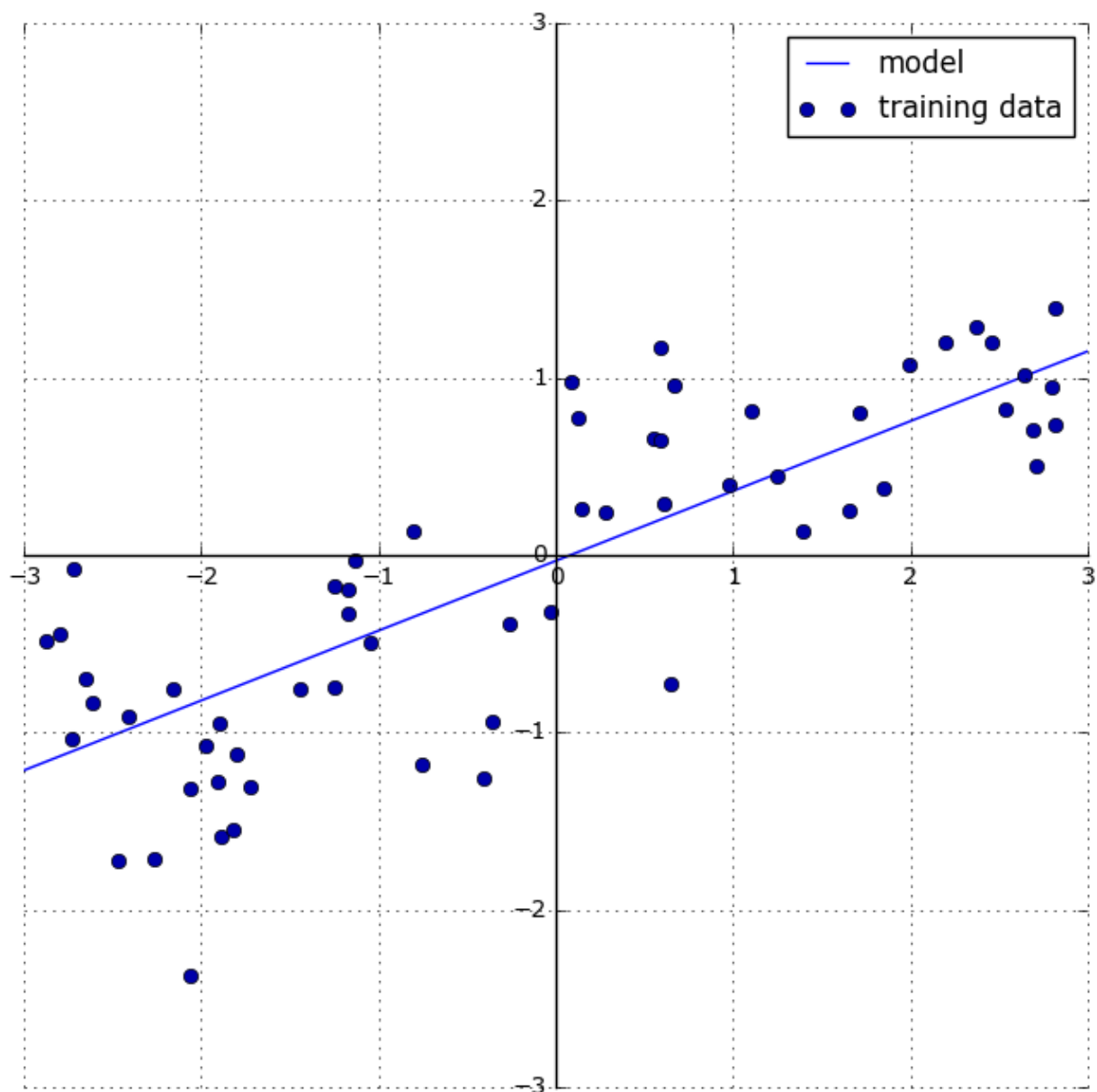
```
In[25]:  
mglearn.plots.plot_linear_regression_wave()
```

```
Out[25]:  
w[0]: 0.393906 b: -0.031804
```

---

<sup>9</sup> Более точно наклон представляет собой тангенс угла наклона линии регрессии и называется регрессионным коэффициентом, а сдвиг определяет точку пересечения линии регрессии с осью ординат и называется свободным членом или константой. – *Прим. пер.*





**Рис. 2.11** Прогнозы линейной модели для набора данных wave

Мы добавили координатный крест на график, чтобы прямую было проще интерпретировать. Взглянув на значение  $w[0]$ , мы видим, что наклон должен быть около 0.4 и это визуально подтверждается на графике. Константа (место пересечения линии прогнозов с осью ординат) чуть меньше нуля, что также подтверждается графиком.

Линейные модели для регрессии можно охарактеризовать как регрессионные модели, в которых прогнозом является прямая линия для одного признака, плоскость, когда используем два признака, или гиперплоскость для большего количества измерений (то есть, когда используем много признаков).

Если прогнозы, полученные с помощью прямой линии, сравнить с прогнозами `KNeighborsRegressor` (рис. 2.10), использование линии регрессии для получения прогнозов кажется очень строгим. Похоже, что

все мелкие детали данных не учитываются. В некотором смысле это верно. Мыдвигаем сильное (и в некоторой степени нереальное) предположение, что наша целевая переменная  $y$  является линейной комбинацией признаков. Однако анализ одномерных данных дает несколько искаженную картину. Для наборов данных с большим количеством признаков линейные модели могут быть очень полезны. В частности, если у вас количество признаков превышает количество точек данных для обучения, любую целевую переменную  $y$  можно прекрасно смоделировать (на обучающей выборке) в виде линейной функции.<sup>10</sup>

Существует различные виды линейных моделей для регрессии. Различие между этими моделями заключается в способе оценивания параметров модели  $w$  и  $b$  по обучающим данным и контроле сложности модели. Теперь мы рассмотрим наиболее популярные линейные модели для регрессии.

### Линейная регрессия (обычный метод наименьших квадратов)

Линейная регрессия или *обычный метод наименьших квадратов* (*ordinary least squares, OLS*) – это самый простой и наиболее традиционный метод регрессии. Линейная регрессия находит параметры  $w$  и  $b$ , которые минимизируют *среднеквадратическую ошибку* (*mean squared error*) между спрогнозированными и фактическими ответами  $y$  в обучающем наборе. Среднеквадратичная ошибка равна сумме квадратов разностей между спрогнозированными и фактическими значениями. Линейная регрессия проста, что является преимуществом, но в то же время у нее нет инструментов, позволяющих контролировать сложность модели.

Ниже приводится программный код, который строит модель, приведенную на рис. 2.11:

```
In[26]:
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

Параметры «наклона» ( $w$ ), также называемые весами или *коэффициентами* (*coefficients*), хранятся в атрибуте `coef_`, тогда как *сдвиг* (*offset*) или *константа* (*intercept*), обозначаемая как  $b$ , хранится в атрибуте `intercept_`:

```
In[27]:
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

---

<sup>10</sup> Это легко увидеть, если вы немного знакомы с линейной алгеброй.

```
Out[27]:
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```



Вы можете заметить странный символ подчеркивания в конце названий атрибутов `coef_` и `intercept_`. Библиотека `scikit-learn` всегда хранит все, что является производным от обучающих данных, в атрибутах, которые заканчиваются символом подчеркивания. Это делается для того, чтобы не спутать их с пользовательскими параметрами.

Атрибут `intercept_` - это всегда отдельное число с плавающей точкой, тогда как атрибут `coef_` - это массив `NumPy`, в котором каждому элементу соответствует входной признак. Поскольку в наборе данных `wave` используется только один входной признак, `lr.coef_` содержит только один элемент.

Давайте посмотрим правильность модели на обучающем и тестовом наборах:

```
In[28]:
print("Правильность на обучающем наборе: {:.2f}".format(lr.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(lr.score(X_test, y_test)))
```

```
Out[28]:
Правильность на обучающем наборе: 0.67
Правильность на тестовом наборе: 0.66
```

Значение  $R^2$  в районе 0.66 указывает на не очень хорошее качество модели, однако можно увидеть, что результаты на обучающем и тестовом наборах очень схожи между собой. Возможно, это указывает на недообучение, а не переобучение. Для этого одномерного массива данных опасность переобучения невелика, поскольку модель очень проста (или строга). Однако для высокоразмерных наборов данных (наборов данных с большим количеством признаков) линейные модели становятся более сложными и существует более высокая вероятность переобучения. Давайте посмотрим, как `LinearRegression` работает на более сложном наборе данных, например, на наборе `Boston Housing`. Вспомним, что этот набор данных имеет 506 примеров (наблюдений) и 105 производных признаков. Во-первых, мы загрузим набор данных и разобьем его на обучающий и тестовый наборы. Затем построим модель линейной регрессии:

```
In[29]:
X, y = mglearn.datasets.load_extended_boston()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

При сравнении правильности на обучающем и тестовом наборах выясняется, что мы очень точно предсказываем на обучающем наборе, однако  $R^2$  на тестовом наборе имеет довольно низкое значение:

```
In[30]:
print("Правильность на обучающем наборе: {:.2f}".format(lr.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(lr.score(X_test, y_test)))
```

```
Out[30]:
Правильность на обучающем наборе: 0.95
Правильность на тестовом наборе: 0.61
```

Это несоответствие между правильностью на обучающем наборе и правильностью на тестовом наборе является явным признаком переобучения и поэтому мы должны попытаться найти модель, которая позволит нам контролировать сложность. Одна из наиболее часто используемых альтернатив стандартной линейной регрессии – гребневая регрессия, которую мы рассмотрим ниже.

### Гребневая регрессия

Гребневая регрессия<sup>11</sup> также является линейной моделью регрессии, поэтому ее формула аналогична той, что используется в обычном методе наименьших квадратов. В гребневой регрессии коэффициенты ( $w$ ) выбираются не только с точки зрения того, насколько хорошо они позволяют предсказывать на обучающих данных, они еще подгоняются в соответствии с дополнительным ограничением. Нам нужно, чтобы величина коэффициентов была как можно меньше. Другими словами, все элементы  $w$  должны быть близки к нулю. Это означает, что каждый признак должен иметь как можно меньшее влияние на результат (то есть каждый признак должен иметь небольшой регрессионный коэффициент) и в то же время он должен по-прежнему обладать хорошей прогнозной силой. Это ограничение является примером *регуляризации* (*regularization*). Регуляризация означает явное ограничение модели для предотвращения переобучения. Регуляризация, используемая в гребневой регрессии, известна как L2 регуляризация.<sup>12</sup>

Гребневая регрессия реализована в классе `linear_model.Ridge`. Давайте посмотрим, насколько хорошо она работает на расширенном наборе данных Boston Housing:

```
In[31]:
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(ridge.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(ridge.score(X_test, y_test)))
```

---

<sup>11</sup> В качестве синонима использует термин «ридж-регрессия». – Прим. пер.

<sup>12</sup> С математической точки зрения `Ridge` штрафует L2 норму коэффициентов или евклидову длину  $w$ .

Out[31]:

Правильность на обучающем наборе: 0.89  
Правильность на тестовом наборе: 0.75

Здесь мы видим, что на обучающем наборе модель **Ridge** дает *меньшую* правильность, чем модель **LinearRegression**, тогда как правильность на тестовом наборе в случае применения гребневой регрессии *выше*. Это согласуется с нашими ожиданиями. При использовании линейной регрессии мы получили переобучение. **Ridge** – модель с более строгим ограничением, поэтому меньше вероятность переобучения. Менее сложная модель означает меньшую правильность на обучающем наборе, но лучшую обобщающую способность. Поскольку нас интересует только обобщающая способность, мы должны выбрать модель **Ridge** вместо модели **LinearRegression**.

Модель **Ridge** позволяет найти компромисс между простотой модели (получением коэффициентов, близких к нулю) и качеством ее работы на обучающем наборе. Компромисс между простотой модели и качеством работы на обучающем наборе может быть задан пользователем при помощи параметра **alpha**. В предыдущем примере мы использовали значение параметра по умолчанию **alpha=1.0**. Впрочем, нет никаких причин считать, что это даст нам оптимальный компромиссный вариант. Оптимальное значение **alpha** зависит от конкретного используемого набора данных. Увеличение **alpha** заставляет коэффициенты сжиматься до близких к нулю значений, что снижает качество работы модели на обучающем наборе, но может улучшить ее обобщающую способность. Например:

In[32]:

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(ridge10.score(X_test, y_test)))
```

Out[32]:

Правильность на обучающем наборе: 0.79  
Правильность на тестовом наборе: 0.64

Уменьшая **alpha**, мы сжимаем коэффициенты в меньшей степени, что означает движение вправо на рис. 2.1. При очень малых значениях **alpha**, ограничение на коэффициенты практически не накладывается и мы в конечном итоге получаем модель, напоминающую линейную регрессию:

In[33]:

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(ridge01.score(X_test, y_test)))
```

Out[33]:

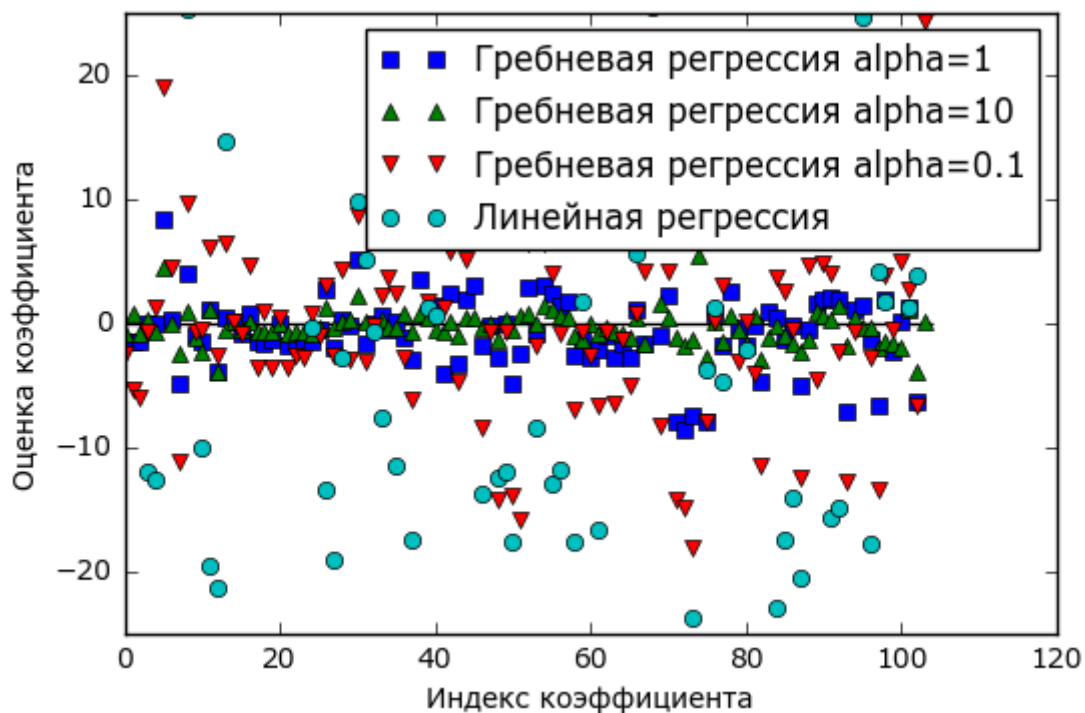
Правильность на обучающем наборе: 0.93  
Правильность на тестовом наборе: 0.77

Похоже, что здесь параметр `alpha=0.1` сработал хорошо. Мы могли бы попробовать уменьшить `alpha` еще больше, чтобы улучшить обобщающую способность. Сейчас обратите внимание на то, как параметр `alpha` соотносится со сложностью модели (рис. 2.1). В главе 5 мы рассмотрим методы правильного подбора параметров.

Кроме того, мы можем лучше понять, как параметр `alpha` меняет модель, используя атрибут `coef_` с разными значениями `alpha`. Чем выше `alpha`, тем более жесткое ограничение накладывается на коэффициенты, поэтому следует ожидать меньшие значения элементов `coef_` для высокого значения `alpha`. Это подтверждается графиком на рис. 2.12:

```
In[34]:
plt.plot(ridge.coef_, 's', label="Гребневая регрессия alpha=1")
plt.plot(ridge10.coef_, '^', label="Гребневая регрессия alpha=10")
plt.plot(ridge01.coef_, 'v', label="Гребневая регрессия alpha=0.1")

plt.plot(lr.coef_, 'o', label="Линейная регрессия")
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```



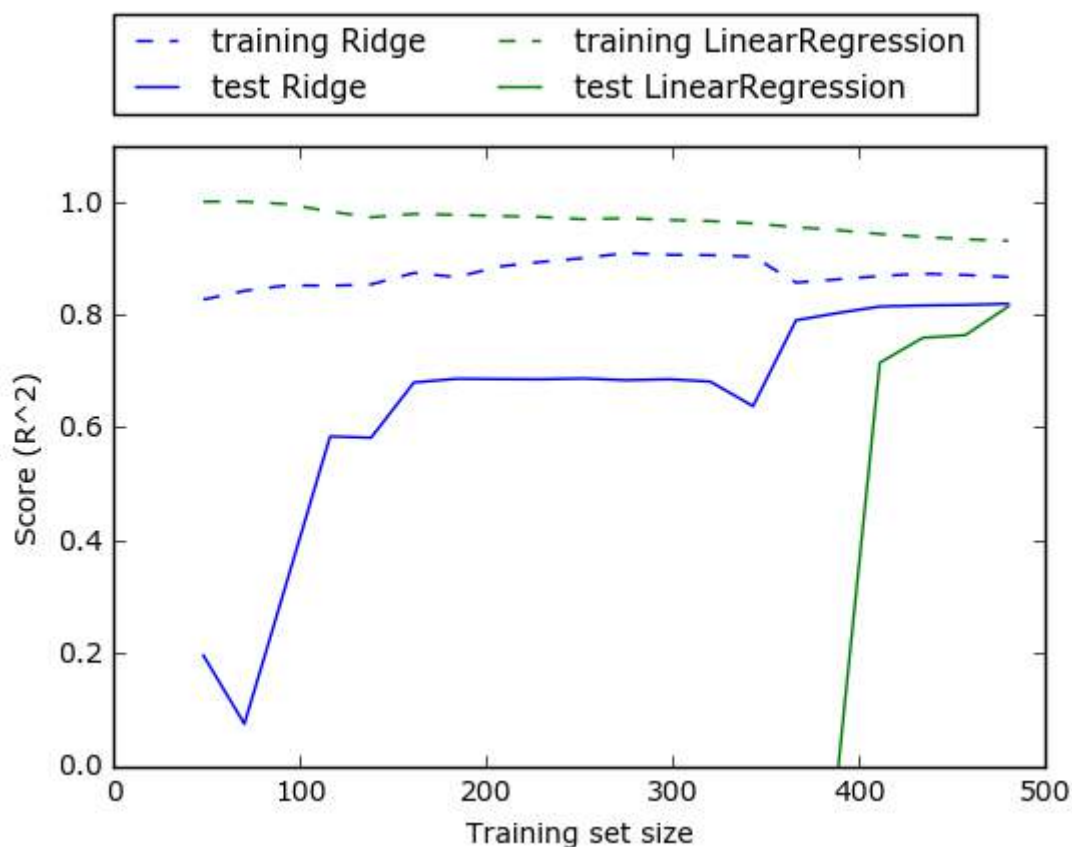
**Рис. 2.12** Сравнение оценок коэффициентов гребневой регрессии с разными значениями `alpha` и линейной регрессии

Здесь ось `x` соответствует элементам атрибута `coef_`: `x=0` показывает коэффициент, связанный с первым признаком, `x=1` – коэффициент, связанный со вторым признаком и так далее, вплоть до `x=100`. Ось `y`

показывает числовые значения соответствующих коэффициентов. Ключевой информацией здесь является то, что для  $\alpha=10$  коэффициенты главным образом расположены в диапазоне от -3 до 3. Коэффициенты для модели Ridge с  $\alpha=1$  несколько больше. Точки, соответствующие  $\alpha=0.1$ , имеют более высокие значения, а большинство точек, соответствующих линейной регрессии без регуляризации (что соответствует  $\alpha=0$ ), настолько велики, что находятся за пределами диаграммы.

Еще один способ понять влияние регуляризации заключается в том, чтобы зафиксировать значение  $\alpha$  и при этом менять доступный объем обучающих данных. Мы сформировали выборки разного объема на основе набора данных Boston Housing и затем построили `LinearRegression` и `Ridge(alpha=1)` на полученных подмножествах, увеличивая объем. На рис. 2.13 приводятся графики, которые показывают качество работы модели в виде функции от объема набора данных, их еще называют *кривыми обучения* (*learning curves*):

```
In[35]:  
mglearn.plots.plot_ridge_n_samples()
```



**Рис. 2.13** Кривые обучения гребневой регрессии и линейной регрессии для набора данных Boston Housing



Как и следовало ожидать, независимо от объема данных правильность на обучающем наборе всегда выше правильности на тестовом наборе, как в случае использования гребневой регрессии, так и в случае использования линейной регрессии. Поскольку гребневая регрессия – регуляризированная модель, во всех случаях на обучающем наборе правильность гребневой регрессии ниже правильности линейной регрессии. Однако правильность на тестовом наборе у гребневой регрессии выше, особенно для небольших подмножеств данных. При объеме данных менее 400 наблюдений линейная регрессия не способна обучиться чему-либо. По мере возрастания объема данных, доступного для моделирования, обе модели становятся лучше и в итоге линейная регрессия догоняет гребневую регрессию. Урок здесь состоит в том, что при достаточном объеме обучающих данных регуляризация становится менее важной и при удовлетворительном объеме данных гребневая и линейная регрессии будут демонстрировать одинаковое качество работы (тот факт, что в данном случае это происходит при использовании полного набора данных, является просто случайностью). Еще одна интересная деталь рис. 2.13 – это снижение правильности линейной регрессии на обучающем наборе. С возрастанием объема данных модели становится все сложнее переобучиться или запомнить данные.

## Лассо

Альтернативой Ridge как метода регуляризации линейной регрессии является Lasso. Как и гребневая регрессия, лассо также сжимает коэффициенты до близких к нулю значений, но несколько иным способом, называемым L1 регуляризацией.<sup>13</sup> Результат L1 регуляризации заключается в том, что при использовании лассо некоторые коэффициенты становятся равны *точно нулю*. Получается, что некоторые признаки полностью исключаются из модели. Это можно рассматривать как один из видов автоматического отбора признаков. Получение нулевых значений для некоторых коэффициентов часто упрощает интерпретацию модели и может выявить наиболее важные признаки вашей модели.

Давайте применим метод лассо к расширенному набору данных Boston Housing:

```
In[36]:
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.2f}".format(lasso.score(X_train, y_train)))
print("Правильность на контрольном наборе: {:.2f}".format(lasso.score(X_test, y_test)))
print("Количество использованных признаков: {}".format(np.sum(lasso.coef_ != 0)))
```

---

<sup>13</sup> Лассо штрафует L1 норму вектора коэффициентов или, другими словами, сумму абсолютных значений коэффициентов.



Out[36]:

Правильность на обучающем наборе: 0.29  
Правильность на контрольном наборе: 0.21  
Количество использованных признаков: 4

Как видно из сводки, **Lasso** дает низкую правильность как на обучающем, так и на тестовом наборе. Это указывает на недообучение и мы видим, что из 105 признаков используются только 4. Как и **Ridge**, **Lasso** также имеет параметр регуляризации **alpha**, который определяет степень сжатия коэффициентов до нулевых значений. В предыдущем примере мы использовали значение по умолчанию **alpha=1.0**. Чтобы снизить недообучение, давайте попробуем уменьшить **alpha**. При этом нам нужно увеличить значение **max\_iter** (максимальное количество итераций):

In[37]:

```
# мы увеличиваем значение "max_iter",  
# иначе модель выдаст предупреждение, что нужно увеличить max_iter.  
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)  
print("Правильность на обучающем наборе: {:.2f}".format(lasso001.score(X_train, y_train)))  
print("Правильность на тестовом наборе: {:.2f}".format(lasso001.score(X_test, y_test)))  
print("Количество использованных признаков: {}".format(np.sum(lasso001.coef_ != 0)))
```

Out[37]:

Правильность на обучающем наборе: 0.90  
Правильность на контрольном наборе: 0.77  
Количество использованных признаков: 33

Более низкое значение **alpha** позволило нам получить более сложную модель, которая продемонстрировала более высокую правильность на обучающем и тестовом наборах. Лассо работает немного лучше, чем гребневая регрессия, и мы используем лишь 33 признака из 105. Это делает данную модель более легкой с точки зрения интерпретации.

Однако, если мы установим слишком низкое значение **alpha**, мы снова нивелируем эффект регуляризации и получим в конечном итоге переобучение, придя к результатам, аналогичным результатам линейной регрессии:

In[38]:

```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)  
print("Правильность на обучающем наборе: {:.2f}".format(lasso00001.score(X_train, y_train)))  
print("Правильность на тестовом наборе: {:.2f}".format(lasso00001.score(X_test, y_test)))  
print("Количество использованных признаков: {}".format(np.sum(lasso00001.coef_ != 0)))
```

Out[38]:

Правильность на обучающем наборе: 0.95  
Правильность на контрольном наборе: 0.64  
Количество использованных признаков: 94

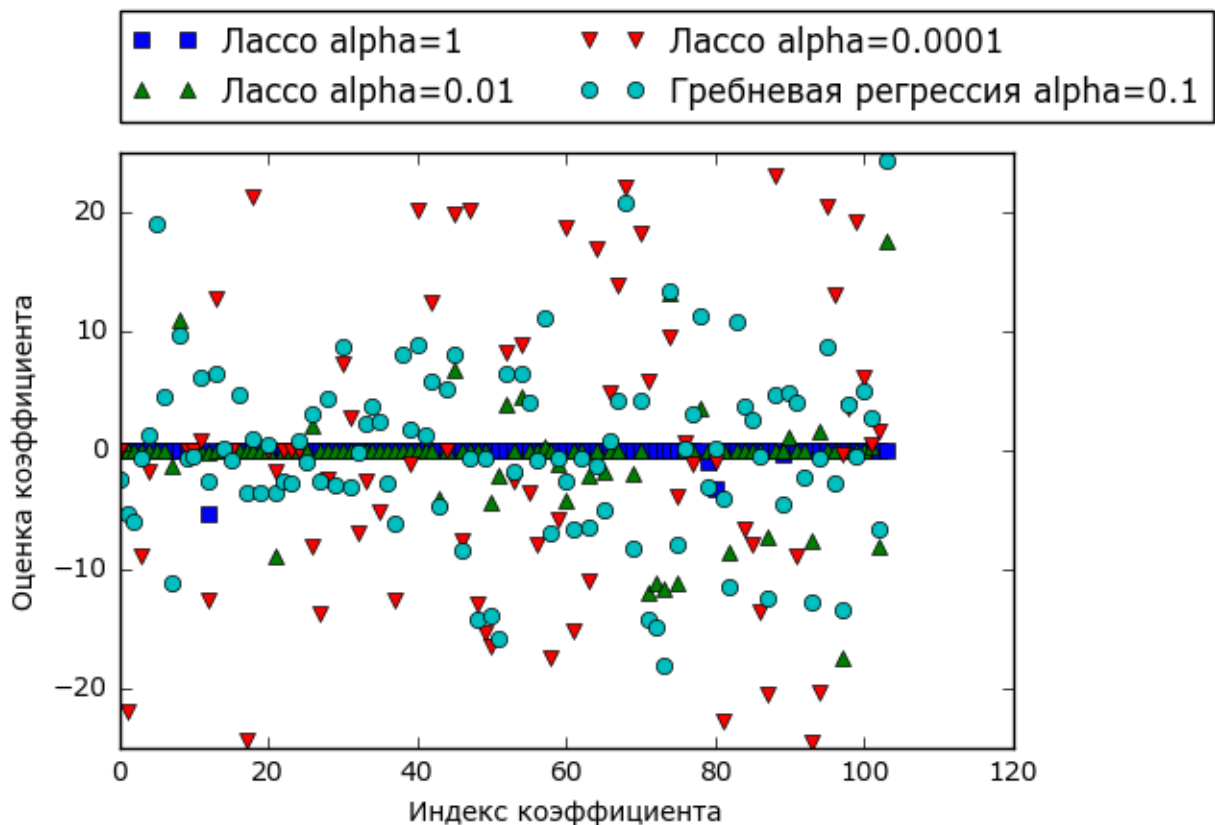
Опять же, мы можем построить графики для коэффициентов различных моделей, аналогичные рис. 2.12. Результат приведен на рис. 2.14:

```

In[39]:
plt.plot(lasso.coef_, 's', label="Лассо alpha=1")
plt.plot(lasso001.coef_, '^', label="Лассо alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Лассо alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Гребневая регрессия alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")

```



**Рис. 2.14** Сравнение коэффициентов для лассо-регрессии с разными значениями  $\alpha$  и гребневой регрессии

Для  $\alpha=1$  мы видим, что не только большинство коэффициентов равны нулю (что мы уже знали), но и остальные коэффициенты также малы по величине. Уменьшив  $\alpha$  до  $0.01$ , получаем решение, показанное в виде зеленых треугольников, большая часть коэффициентов для признаков становятся в точности равными нулю. При  $\alpha=0.0001$  мы получаем практически нерегуляризованную модель, у которой большинство коэффициентов отличны от нуля и имеют большие значения. Для сравнения приводится наилучшее решение, полученное с помощью гребневой регрессии. Модель Ridge с  $\alpha=0.1$  имеет такую же прогностическую способность, что и модель лассо с  $\alpha=0.01$ , однако при использовании гребневой регрессии все коэффициенты отличны от нуля.

На практике, когда стоит выбор между гребневой регрессией и лассо, предпочтение, как правило, отдается гребневой регрессии. Однако, если у вас есть большое количество признаков и есть основания считать, что лишь некоторые из них важны, **Lasso** может быть оптимальным выбором. Аналогично, если вам нужна легко интерпретируемая модель, **Lasso** поможет получить такую модель, так как она выберет лишь подмножество входных признаков. В библиотеке **scikit-learn** также имеется класс **ElasticNet**, который сочетает в себе штрафы **Lasso** и **Ridge**. На практике эта комбинация работает лучше всего, впрочем, это достигается за счет двух корректируемых параметров: один для L1 регуляризации, а другой – для L2 регуляризации.

### Линейные модели для классификации

Линейные модели также широко используются в задачах классификации. Давайте посмотрим сначала на бинарную классификацию. В этом случае прогноз получают с помощью следующей формулы:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b > 0$$

Формула очень похожа на формулу линейной регрессии, но теперь вместо того, чтобы просто возратить взвешенную сумму признаков, мы задаем для прогнозируемого значения порог, равный нулю. Если функция меньше нуля, мы прогнозируем класс  $-1$ , если она больше нуля, мы прогнозируем класс  $+1$ . Это прогнозное правило является общим для всех линейных моделей классификации. Опять же, есть много различных способов найти коэффициенты ( $w$ ) и константу ( $b$ ).

Для линейных моделей регрессии выход  $\hat{y}$  является линейной функцией признаков: линией, плоскостью или гиперплоскостью (для большого количества измерений). Для линейных моделей классификации *граница принятия решений* (*decision boundary*) является линейной функцией аргумента. Другими словами, (бинарный) линейный классификатор – это классификатор, который разделяет два класса с помощью линии, плоскости или гиперплоскости. В этом разделе мы приведем конкретные примеры.

Существует масса алгоритмов обучения линейных моделей. Два критерия задают различия между алгоритмами:

- Измеряемые метрики качества подгонки обучающих данных;
- Факт использования регуляризации и вид регуляризации, если она используется.

Различные алгоритмы по-разному определяют, что значит «хорошая подгонка обучающих данных». В силу технико-математических причин

невозможно скорректировать  $w$  и  $b$ , чтобы минимизировать количество неверно классифицированных случаев, выдаваемое алгоритмами, как можно было бы надеяться. С точки зрения поставленных нами целей и различных сфер применения различные варианты метрик качества подгонки (так называемые *функции потерь* или *loss functions*) не имеют большого значения.

Двумя наиболее распространенными алгоритмами линейной классификации являются *логистическая регрессия* (*logistic regression*), реализованная в классе `linear_model.LogisticRegression`, и *линейный метод опорных векторов* (*linear support vector machines*) или линейный SVM, реализованный в классе `svm.LinearSVC` (SVC расшифровывается как *support vector classifier* – *классификатор опорных векторов*). Несмотря на свое название, логистическая регрессия является алгоритмом классификации, а не алгоритмом регрессии, и его не следует путать с линейной регрессией.

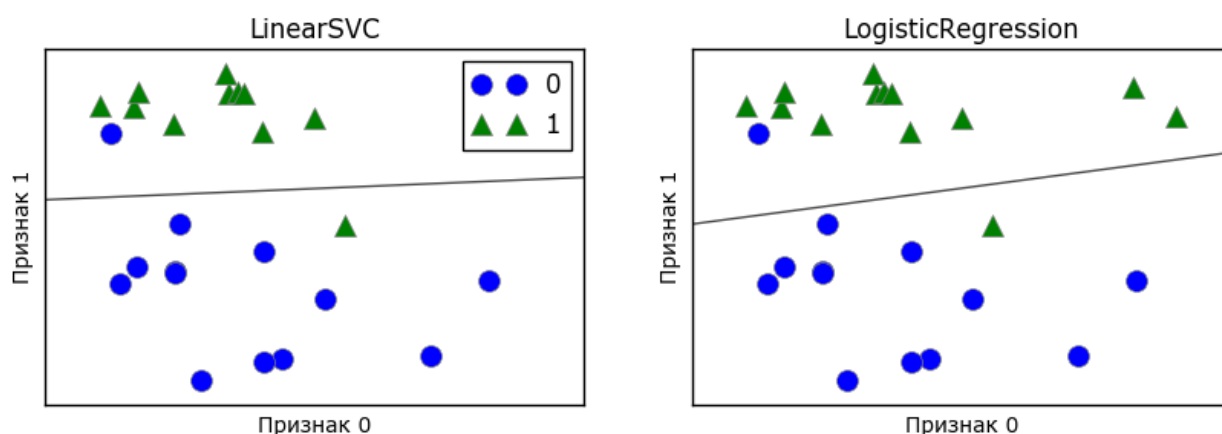
Мы можем применить модели `LogisticRegression` и `LinearSVC` к набору данных `forge` и визуализировать границу принятия решений, найденную линейными моделями (рис. 2.15):

```
In[40]:
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
                                   ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} {}".format(clf.__class__.__name__, model.__name__))
    ax.set_xlabel("Признак 0")
    ax.set_ylabel("Признак 1")
axes[0].legend()
```



**Рис. 2.15** Границы принятия решений линейного SVM и логистической регрессии для набора данных `forge` (использовались значения параметров по умолчанию)

На этом рисунке, как и раньше, первый признак набора данных `forge` отложен по оси  $x$ , а второй признак – по оси  $y$ . Здесь показаны границы принятия решений, найденные `LinearSVC` и `LogisticRegression` соответственно. Они представлены в виде прямых линий, отделяющих область значений, классифицированных как класс 1 (в верхней части графика) от области значений, классифицированных как класс 0 (в нижней части графика). Другими словами, любая новая точка данных, которая лежит выше черной линии будет отнесена соответствующей моделью к классу 1, тогда как любая точка, лежащая ниже черной линии, будет отнесена к классу 0.

Обе модели имеют схожие границы принятия решений. Обратите внимание, что обе модели неправильно классифицировали две точки. По умолчанию обе модели используют L2 регуляризацию, тот же самый метод, который используется в гребневой регрессии.

Для `LogisticRegression` и `LinearSVC` компромиссный параметр, который определяет степень регуляризации, называется  $C$ , и более высокие значения  $C$  соответствуют *меньшей* регуляризации. Другими словами, когда вы используете высокое значение параметра  $C$ , `LogisticRegression` и `LinearSVC` пытаются подогнать модель к обучающим данным как можно лучше, тогда как при низких значениях параметра  $C$  модели делают больший акцент на поиске вектора коэффициентов ( $w$ ), близкого к нулю.

Существует еще одна интересная деталь, связанная с работой параметра  $C$ . Использование низких значений  $C$  приводит к тому, что алгоритмы пытаются подстроиться под «большинство» точек данных, тогда как использование более высоких значений  $C$  подчеркивает важность того, чтобы каждая отдельная точка данных была классифицирована правильно. Ниже приводится иллюстрация использования `LinearSVC` (рис. 2.16):

```
In[41]:  
mglearn.plots.plot_linear_svc_regularization()
```

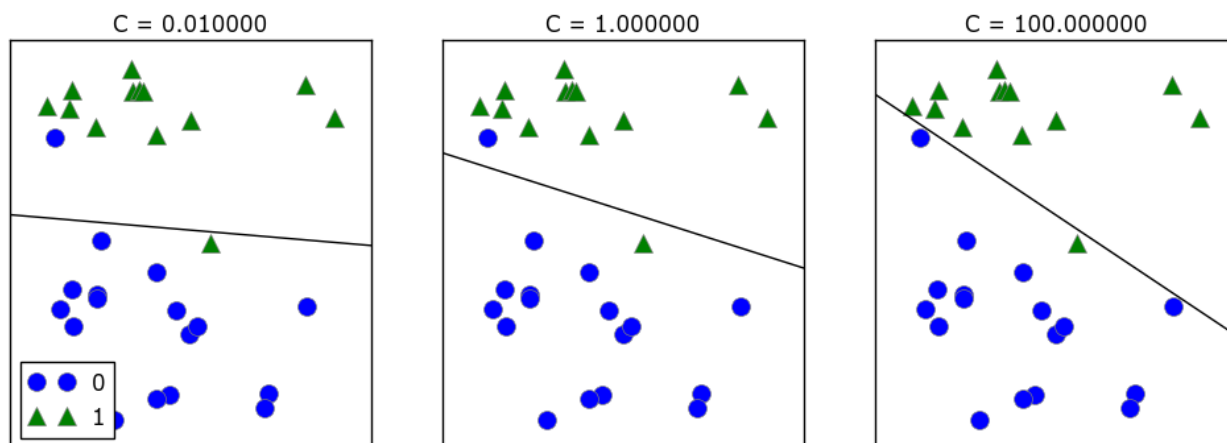


Рис. 2.16 Границы принятия решений линейного SVM с различными значениями  $C$  для набора данных *forge*

На графике слева показана модель с очень маленьким значением  $C$ , соответствующим большой степени регуляризации. Большая часть точек класса 0 находятся в нижней части графика, а большинство точек класса 1 находятся в верхней части. Сильно регуляризованная модель дает относительно горизонтальную линию, неправильно классифицируя две точки. На центральном графике значение  $C$  немного выше и модель в большей степени фокусируется на двух неправильно классифицированных примерах, наклоняя границу принятия решений. Наконец, на графике справа очень высокое значение  $C$  модели наклоняет границу принятия решений еще сильнее, теперь правильно классифицируя все точки класса 0. Одна из точек класса 1 по-прежнему неправильно классифицирована, поскольку невозможно правильно классифицировать все наблюдения этого набора данных с помощью прямой линии. Модель на графике справа старается изо всех сил правильно классифицировать все точки, но не может дать хорошего обобщения сразу для обоих классов. Другими словами, эта модель скорее всего переобучена.

Как и в случае с регрессией, линейные модели классификации могут показаться слишком строгими в условиях низкоразмерного пространства, предлагая границы принятия решений в виде прямых линий или плоскостей. Опять же, при наличии большого числа измерений линейные модели классификации приобретают высокую прогнозную силу и с увеличением числа признаков защита от переобучения становится все более важной.

Давайте более подробно разберем работу `LogisticRegression` на наборе данных *Breast Cancer*:

```

In[42]:
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(logreg.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(logreg.score(X_test, y_test)))

```

```

Out[42]:
Правильность на обучающем наборе: 0.953
Правильность на тестовом наборе: 0.958

```

Значение по умолчанию  $C=1$  обеспечивает неплохое качество модели, правильность на обучающем и тестовом наборах составляет 95%. Однако поскольку качество модели на обучающем и тестовом наборах примерно одинаково, вполне вероятно, что мы недообучили модель. Давайте попробуем увеличить  $C$ , чтобы подогнать более гибкую модель:

```

In[43]:
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(logreg100.score(X_test, y_test)))

```

```

Out[43]:
Правильность на обучающем наборе: 0.972
Правильность на тестовом наборе: 0.965

```

Использование  $C=100$  привело к более высокой правильности на обучающей выборке, а также немного увеличилась правильность на тестовой выборке, что подтверждает наш довод о том, что более сложная модель должна работать лучше.

Кроме того, мы можем выяснить, что произойдет, если мы воспользуемся более регуляризованной моделью (установив  $C=0.01$  вместо значения по умолчанию  $C=1$ ):

```

In[44]:
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(logreg001.score(X_test, y_test)))

```

```

Out[44]:
Правильность на обучающем наборе: 0.934
Правильность на тестовом наборе: 0.930

```

Как и следовало ожидать, когда мы получили недообученную модель и переместились влево по шкале, показанной на рис. 2.1, правильность как на обучающем, так и на тестовом наборах снизилась по сравнению с правильностью, которую мы получили, используя параметры по умолчанию.

И, наконец, давайте посмотрим на коэффициенты логистической регрессии, полученные с использованием трех различных значений параметра регуляризации  $C$  (рис. 2.17):

```

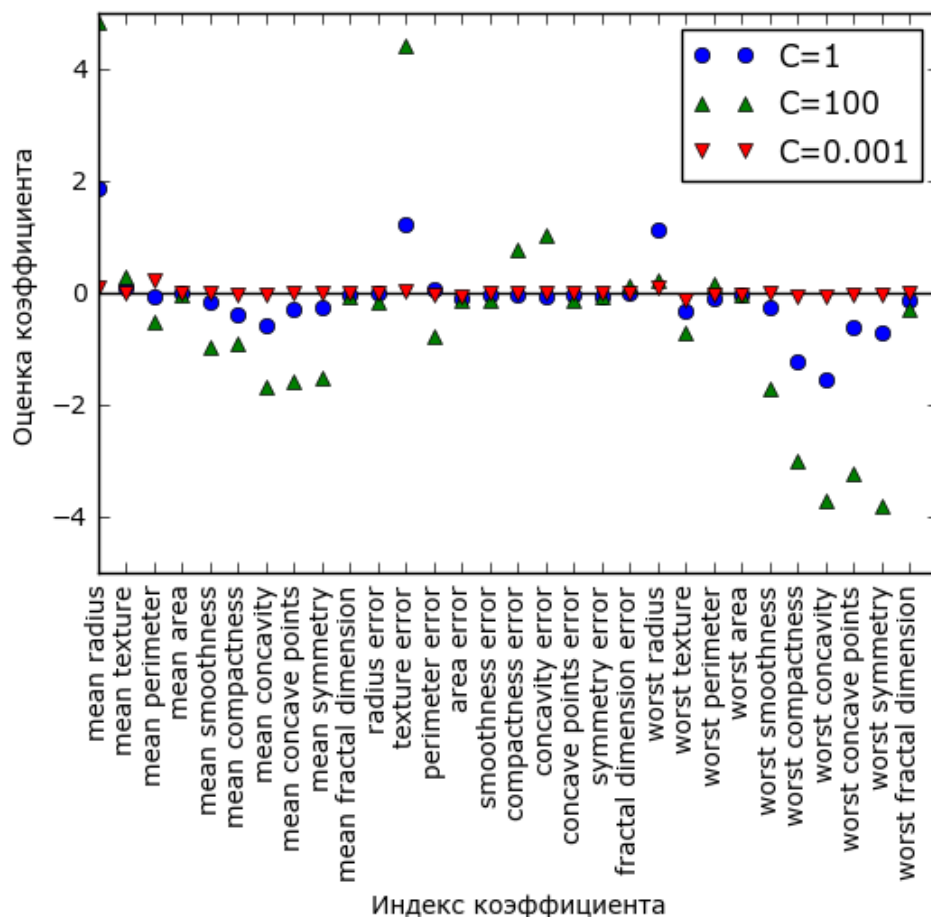
In[45]:
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")
plt.legend()

```



Поскольку `LogisticRegression` по умолчанию использует L2 регуляризацию, результат похож на результат, полученный при использовании модели `Ridge` (рис. 2.12). Большая степень регуляризации сильнее сжимает коэффициенты к нулю, хотя коэффициенты никогда не станут в точности равными нулю. Изучив график более внимательно, можно увидеть интересный эффект, произошедший с третьим коэффициентом, коэффициентом признака «mean perimeter». При  $C=100$  и  $C=1$  коэффициент отрицателен, тогда как при  $C=0.001$  коэффициент положителен, при этом его оценка больше, чем оценка коэффициента при  $C=1$ . Когда мы интерпретируем данную модель, коэффициент говорит нам, какой класс связан с этим признаком. Возможно, что высокое значение признака «texture error» связано с примером, классифицированным как «злокачественный». Однако изменение знака коэффициента для признака «mean perimeter» означает, что в зависимости от рассматриваемой модели высокое значение «mean perimeter» может указывать либо на доброкачественную, либо на злокачественную опухоль. Приведенный пример показывает, что интерпретировать коэффициенты линейных моделей всегда нужно с осторожностью и скептицизмом.





**Рис. 2.17** Коэффициенты, полученные с помощью логистической регрессии с разными значениями  $C$  для набора данных Breast Cancer

Если мы хотим получить более интерпретабельную модель, нам может помочь L1 регуляризация, поскольку она ограничивает модель использованием лишь нескольких признаков. Ниже приводится график с коэффициентами и оценками правильности для L1 регуляризации (рис. 2.18):

```
In[46]:
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("Правильность на обучении для логрегрессии l1 с C={:.3f}: {:.2f}".format(
        C, lr_l1.score(X_train, y_train)))
    print("Правильность на тесте для логрегрессии l1 с C={:.3f}: {:.2f}".format(
        C, lr_l1.score(X_test, y_test)))

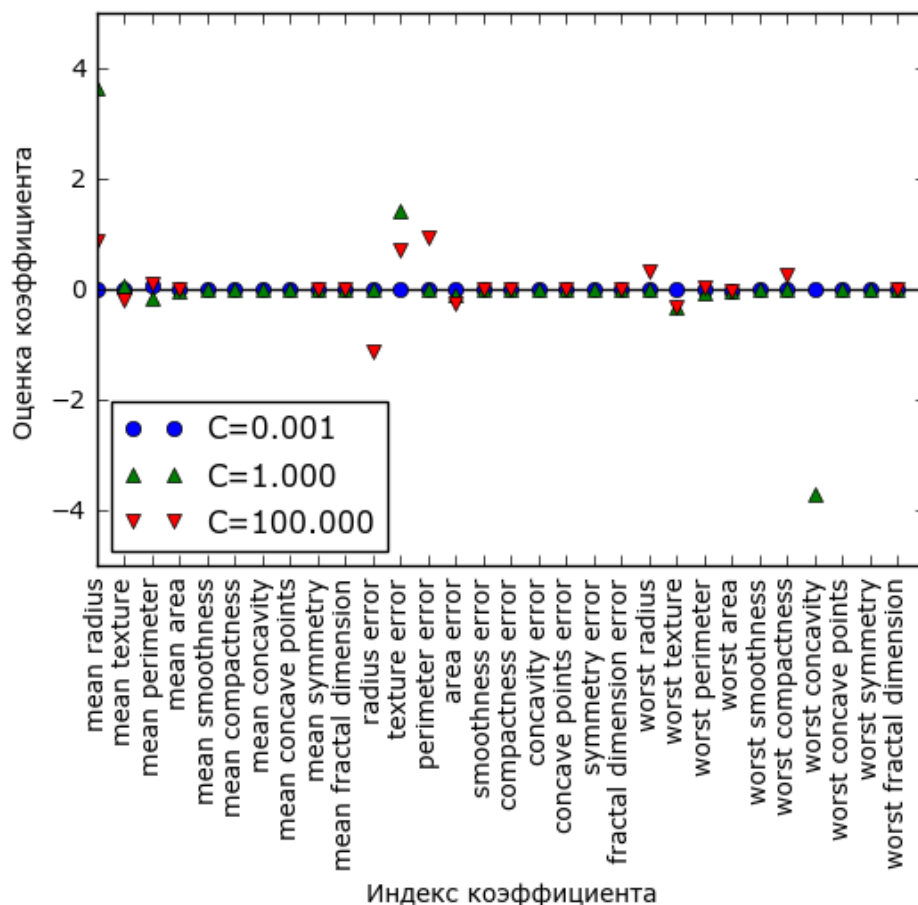
plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Индекс коэффициента")
plt.ylabel("Оценка коэффициента")

plt.ylim(-5, 5)
plt.legend(loc=3)
```

```
Out[46]:
Правильность на обучении для логрегрессии l1 с C=0.001: 0.91
Правильность на тесте для логрегрессии l1 с C=0.001: 0.92
Правильность на обучении для логрегрессии l1 с C=1.000: 0.96
Правильность на тесте для логрегрессии l1 с C=1.000: 0.96
```

Правильность на обучении для логрегрессии l1 с C=100.000: 0.99  
 Правильность на тесте для логрегрессии l1 с C=100.000: 0.98

Видно, что существует много параллелей между линейными моделями для бинарной классификации и линейными моделями для регрессии. Как и в регрессии, основное различие между моделями – в параметре **penalty**, который влияет на регуляризацию и определяет, будет ли модель использовать все доступные признаки или выберет лишь подмножество признаков.



**Рис. 2.18** Коэффициенты логистической регрессии с L1 штрафом для набора данных Breast Cancer (использовались различные значения C)

### Линейные модели для мультиклассовой классификации

Многие линейные модели классификации предназначены лишь для бинарной классификации и не распространяются на случай мультиклассовой классификации (за исключением логистической регрессии). Общераспространенный подход, позволяющий распространить алгоритм бинарной классификации на случай мультиклассовой классификации называется подходом «один против остальных» (*one-vs.-rest*).<sup>14</sup> В подходе «один против остальных» для каждого класса строится бинарная модель, которая пытается отделить

<sup>14</sup> Также используется название «один против всех» (*one-vs.-all*). – Прим. пер.

этот класс от всех остальных, в результате чего количество моделей определяется количеством классов. Для получения прогноза точка тестового набора подается на все бинарные классификаторы. Классификатор, который выдает по своему классу наибольшее значение, «побеждает» и метка этого класса возвращается в качестве прогноза.

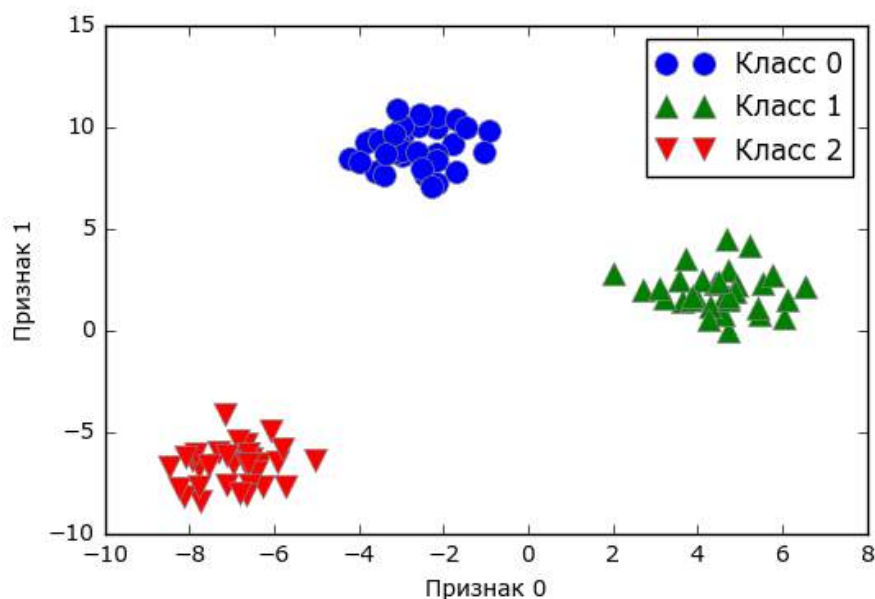
Используя бинарный классификатор для каждого класса, мы получаем один вектор коэффициентов ( $w$ ) и одну константу ( $b$ ) по каждому классу. Класс, который получает наибольшее значение согласно нижеприведенной формуле, становится присвоенной меткой класса:

$$w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Математический аппарат мультиклассовой логистической регрессии несколько отличается от подхода «один против остальных», однако он также дает один вектор коэффициентов и константу для каждого класса и использует тот же самый способ получения прогнозов.

Давайте применим метод «один против остальных» к простому набору данных с 3-классовой классификацией. Мы используем двумерный массив данных, где каждый класс задается данными, полученными из гауссовского распределения (см. рис. 2.19):

```
In[47]:  
from sklearn.datasets import make_blobs  
  
X, y = make_blobs(random_state=42)  
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)  
plt.xlabel("Признак 0")  
plt.ylabel("Признак 1")  
plt.legend(["Класс 0", "Класс 1", "Класс 2"])
```



**Рис. 2.19** Двумерный синтетический набор данных, содержащий три класса

Теперь обучаем классификатор `LinearSVC` на этом наборе данных:

```
In[48]:
linear_svm = LinearSVC().fit(X, y)
print("Форма коэффициента: ", linear_svm.coef_.shape)
print("Форма константы: ", linear_svm.intercept_.shape)
```

```
Out[48]:
Форма коэффициента: (3, 2)
Форма константы: (3,)
```

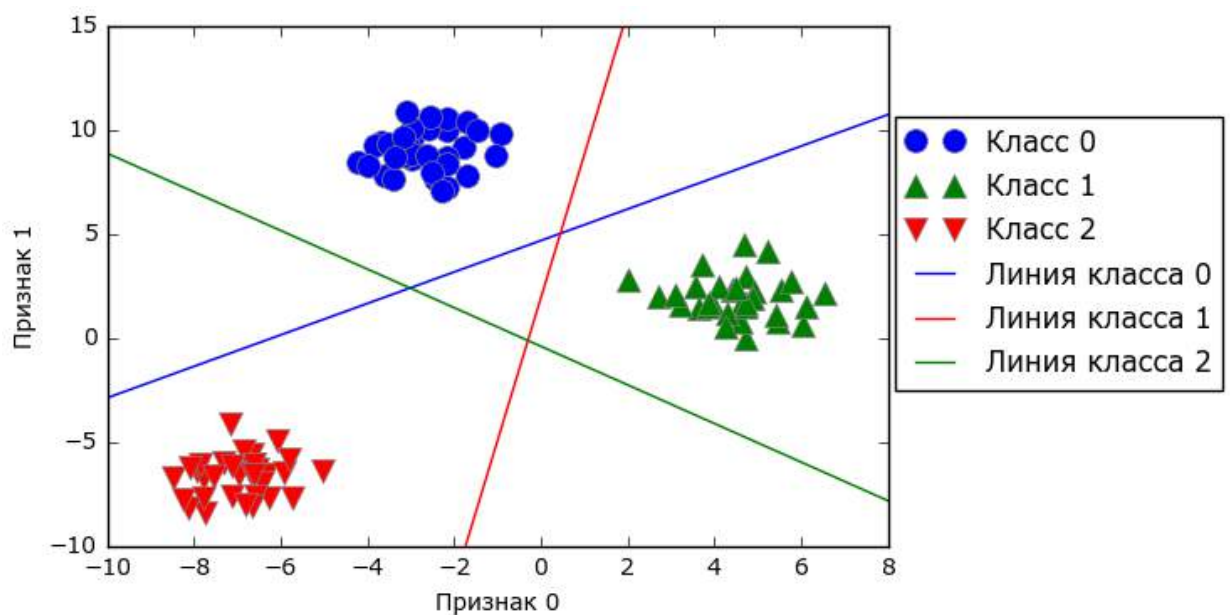
Мы видим, что атрибут `coef_` имеет форму (3, 2), это означает, что каждая строка `coef_` содержит вектор коэффициентов для каждого из трех классов, а каждый столбец содержит значение коэффициента для конкретного признака (в этом наборе данных их два). Атрибут `intercept_` теперь является одномерным массивом, в котором записаны константы классов.

Давайте визуализируем линии (границы принятия решений), полученные с помощью трех бинарных классификаторов (рис. 2.20):

```
In[49]:
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                   ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
plt.legend(['Класс 0', 'Класс 1', 'Класс 2', 'Линия класса 0', 'Линия класса 1',
            'Линия класса 2'], loc=(1.01, 0.3))
```

Видно, что все точки, принадлежащие классу 0 в обучающих данных, находятся выше линии, соответствующей классу 0. Это означает, что они отнесены к «классу 0» данного бинарного классификатора. Точки класса 0 находятся выше линии, соответствующей классу 2. Это означает, что они классифицируются бинарным классификатором для класса 2 как «остальные». Точки, принадлежащие классу 0, находятся слева от линии, соответствующей классу 1. Это означает, что бинарный классификатор для класса 1 также классифицирует их как «остальные». Таким образом, в итоге любая точка в этой области будет отнесена к классу 0 (результат, получаемый по формуле для классификатора 0, больше нуля, тогда как для двух остальных классов он меньше нуля).

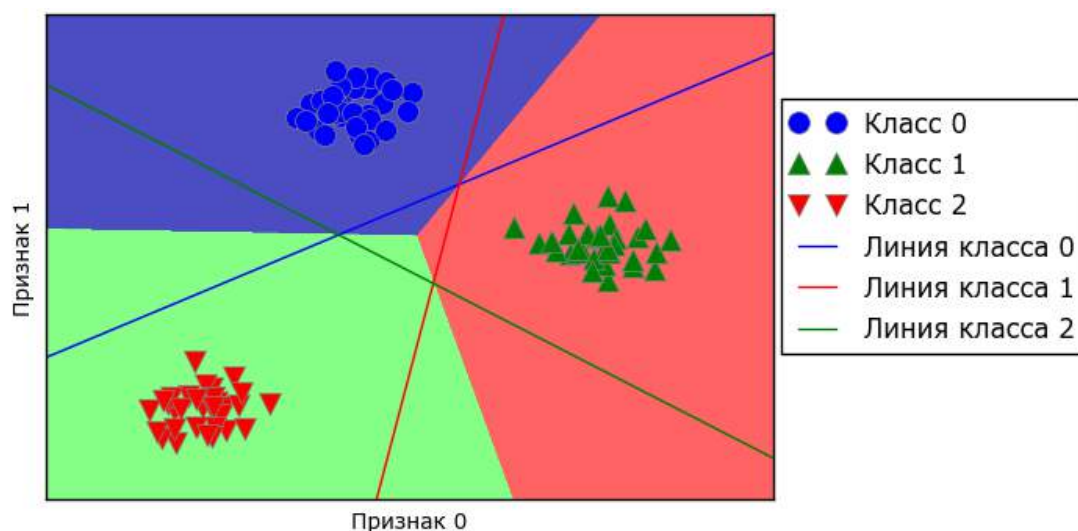
Однако что насчет треугольника в середине графика? Все три бинарных классификатора относят точки, расположенные там, к «остальным». Какой класс будет присвоен точке, расположенной в треугольнике? Ответ – класс, получивший наибольшее значение по формуле классификации, то есть класс ближайшей линии.



**Рис. 2.20** Границы принятия решений, полученные с помощью трех бинарных классификаторов в рамках подхода «один против остальных»

Следующий пример (рис. 2.21) показывает прогнозы для всех областей двумерного пространства:

```
In[50]:
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Класс 0', 'Класс 1', 'Класс 2', 'Линия класса 0', 'Линия класса 1',
          'Линия класса 2'], loc=(1.01, 0.3))
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```



**Рис. 2.21** Мультиклассовые границы принятия решений, полученные с помощью трех бинарных классификаторов в рамках подхода «один против остальных»

## Преимущества, недостатки и параметры

Основной параметр линейных моделей – параметр регуляризации, называемый **alpha** в моделях регрессии и **C** в **LinearSVC** и **LogisticRegression**. Большие значения **alpha** или маленькие значения **C** означают простые модели. Конкретно для регрессионных моделей настройка этих параметров имеет весьма важное значение. Как правило, поиск **C** и **alpha** осуществляется по логарифмической шкале. Кроме того вы должны решить, какой вид регуляризации нужно использовать: **L1** или **L2**. Если вы полагаете, что на самом деле важны лишь некоторые признаки, следует использовать **L1**. В противном случае используйте установленную по умолчанию **L2** регуляризацию. Еще **L1** регуляризация может быть полезна, если интерпретируемость модели имеет важное значение. Поскольку **L1** регуляризация будет использовать лишь несколько признаков, легче будет объяснить, какие признаки важны для модели и каковы эффекты этих признаков.

Линейные модели очень быстро обучаются, а также быстро прогнозируют. Они масштабируются на очень большие наборы данных, а также хорошо работают с разреженными данными. При работе с данными, состоящими из сотен тысяч или миллионов примеров, вас, возможно, заинтересует опция **solver='sag'** в **LogisticRegression** и **Ridge**, которая позволяет получить результаты быстрее, чем настройки по умолчанию. Еще пара опций – это класс **SGDClassifier** и класс **SGDRegressor**, реализующие более масштабируемые версии описанных здесь линейных моделей.

Еще одно преимущество линейных моделей заключается в том, что они позволяют относительно легко понять, как был получен прогноз, при помощи формул, которые мы видели ранее для регрессии и классификации. К сожалению, часто бывает совершенно не понятно, почему были получены именно такие коэффициенты. Это особенно актуально, если ваш набор данных содержит высоко коррелированные признаки, в таких случаях коэффициенты сложно интерпретировать.

Как правило, линейные модели хорошо работают, когда количество признаков превышает количество наблюдений. Кроме того, они часто используются на очень больших наборах данных, просто потому, что не представляется возможным обучить другие модели. Вместе с тем в низкоразмерном пространстве альтернативные модели могут показать более высокую обобщающую способность. В разделе «Ядерные машины опорных векторов» мы рассмотрим несколько примеров, в которых использование линейных моделей не увенчалось успехом.

## Цепочка методов (method chaining)

Во всех моделях **scikit-learn** метод **fit** возвращает **self**. Это позволяет писать код, приведенный ниже и уже широко использованный нами в этой главе:

**In[51]:**

```
# создаем экземпляр модели и подгоняем его в одной строке
logreg = LogisticRegression().fit(X_train, y_train)
```

Здесь мы использовали значение, возвращаемое методом **fit** (**self**), чтобы присвоить обученную модель переменной **logreg**. Эта конкатенация вызовов методов (в данном случае **\_init\_**, а затем **fit**) известна как *цепочка методов* (*method chaining*). Еще одно общераспространенное применение цепочки методов в **scikit-learn** — это связывание методов **fit** и **predict** в одной строке:

**In[52]:**

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Наконец, вы можете создать экземпляр модели, подогнать модель и получить прогнозы в одной строке:

**In[53]:**

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

Однако этот очень короткий вариант не идеален. В одной строке происходит масса всего, что может сделать код трудночитаемым. Кроме того, подогнанная модель логистической регрессии не сохранена в какой-то определенной переменной, поэтому мы не можем проверить или использовать ее, чтобы получить прогнозы для других данных.

## Наивные байесовские классификаторы

Наивные байесовские классификаторы представляют собой семейство классификаторов, которые очень схожи с линейными моделями, рассмотренными в предыдущем разделе. Однако они имеют тенденцию обучаться быстрее. Цена, которую приходится платить за такую эффективность — немного более низкая обобщающая способность моделей Байеса по сравнению с линейными классификаторами типа **LogisticRegression** и **LinearSVC**.

Причина, по которой наивные байесовские модели столь эффективны, заключается в том, что они оценивают параметры, рассматривая каждый признак отдельно и по каждому признаку собирают простые статистики классов. В **scikit-learn** реализованы три вида наивных байесовских классификаторов: **GaussianNB**, **BernoulliNB** и **MultinomialNB**. **GaussianNB** можно применить к любым непрерывным данным, в то время как **BernoulliNB** принимает бинарные данные, **MultinomialNB** принимает счетные или дискретные данные (то есть каждый признак представляет собой подсчет целочисленных значений какой-то характеристики,



например, речь может идти о частоте встречаемости слова в предложении). `BernoulliNB` и `MultinomialNB` в основном используются для классификации текстовых данных.

Классификатор `BernoulliNB` подсчитывает ненулевые частоты признаков по каждому классу. Это легче всего понять на примере:

```
In[54]:
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Здесь у нас есть четыре точки данных с четырьмя бинарными признаками. Есть два класса 0 и 1. Для класса 0 (первая и третья точки данных) первый признак равен нулю два раза и отличен от нуля ноль раз, второй признак равен нулю один раз и отличен от нуля один раз и так далее. Те же самые частоты затем подсчитываются для точек данных во втором классе. Подсчет ненулевых элементов в каждом классе по сути выглядит следующим образом:

```
In[55]:
counts = {}
for label in np.unique(y):
    # итерируем по каждому классу
    # подсчитываем (суммируем) элементы 1 по признаку
    counts[label] = X[y == label].sum(axis=0)
print("Частоты признаков:\n{}".format(counts))
```

```
Out[55]:
Частоты признаков:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

Две другие наивные байесовские модели `MultinomialNB` и `GaussianNB`, немного отличаются с точки зрения вычисляемых статистик. `MultinomialNB` принимает в расчет среднее значение каждого признака для каждого класса, в то время как `GaussianNB` записывает среднее значение, а также стандартное отклонение каждого признака для каждого класса.

Для получения прогноза точка данных сравнивается со статистиками для каждого класса и прогнозируется наиболее подходящий класс. Интересно отметить, что для `MultinomialNB` и `BernoulliNB` это приводит к прогнозной формуле, которая имеет точно такой же вид, что и формула для линейных моделей (см. «Линейные модели классификации»). К сожалению, `coef_` для наивных байесовских моделей имеет несколько иной смысл, чем `coef_` для линейных моделей, здесь `coef_` не тождественен  $W$ .



## Преимущества, недостатки и параметры

**MultinomialNB** и **BernoulliNB** имеют один параметр **alpha**, который контролирует сложность модели. Параметр **alpha** работает следующим образом: алгоритм добавляет к данным зависящее от **alpha** определенное количество искусственных наблюдений с положительными значениями для всех признаков. Это приводит к «сглаживанию» статистик. Большее значение **alpha** означает более высокую степень сглаживания, что приводит к построению менее сложных моделей. Алгоритм относительно устойчив к разным значениям **alpha**. Это означает, что значение **alpha** не оказывает значительного влияния на хорошую работу модели. Вместе с тем тонкая настройка этого параметра обычно немного увеличивает правильность.

**GaussianNB** в основном используется для данных с очень высокой размерностью, тогда как остальные наивные байесовские модели широко используются для разреженных дискретных данных, например, для текста. **MultinomialNB** обычно работает лучше, чем **BernoulliNB**, особенно на наборах данных с относительно большим количеством признаков, имеющих ненулевые частоты (т.е. на больших документах).

Наивные байесовские модели разделяют многие преимущества и недостатки линейных моделей. Они очень быстро обучаются и прогнозируют, а процесс обучения легко интерпретировать. Модели очень хорошо работают с высокоразмерными разреженными данными и относительно устойчивы к изменениям параметров. Наивные байесовские модели являются замечательными базовыми моделями и часто используются на очень больших наборах данных, где обучение даже линейной модели может занять слишком много времени.

## Деревья решений

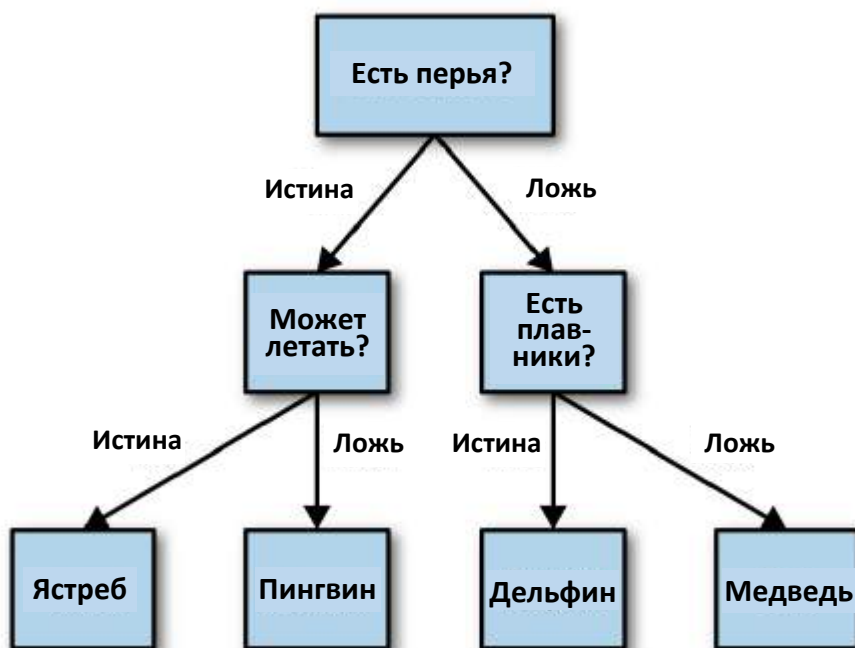
Деревья решений являются моделями, широко используемыми для решения задач классификации и регрессии. По сути они задают вопросы и выстраивают иерархию правил «если... то», приводящую к решению.

Эти вопросы похожи на вопросы, которые вы можете спросить в игре «20 Questions». Представьте, вам нужно научиться отличать друг от друга четыре вида животных: медведей, ястребов, пингвинов и дельфинов. Ваша цель состоит в том, чтобы получить правильный ответ, задав несколько вопросов. Вы могли бы начать с вопроса, есть ли у этих видов животных перья, вопроса, который сужает количество возможных видов животных до двух. Если получен ответ «да», вы можете задать еще один вопрос, который может помочь вам различать ястребов и пингвинов. Например, вы могли бы спросить, может ли данный вид животных летать. Если у этого вида животных нет перьев, ваши возможные

варианты – дельфины и медведи, и вам нужно задать вопрос, чтобы провести различие между этими двумя видами животных, например, спросить, есть ли плавники у этого вида животных.

Эти вопросы можно выразить в виде дерева решений, как это показано на рис. 2.22.

```
In[56]:  
mglearn.plots.plot_animal_tree()
```



**Рис. 2.22** Дерево решений, различающее несколько видов животных

На этом рисунке каждый узел дерева либо представляет собой либо вопрос, либо терминальный узел (его еще называют *листом* или *leaf*), который содержит ответ. Ребра соединяют вышестоящие узлы с нижестоящими.

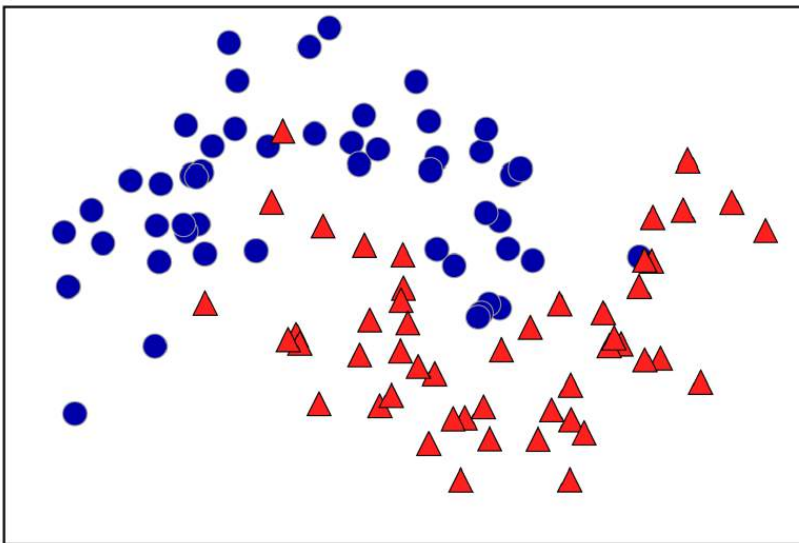
Говоря языком машинного обучения, мы построили модель, различающую четыре класса животных (ястребов, пингвинов, дельфинов и медведей), используя три признака «есть перья», «может летать» и «имеет плавники». Вместо того, чтобы строить эти модели вручную, мы можем построить их с помощью контролируемого обучения.

### Построение деревьев решений

Давайте рассмотрим процесс построения дерева решений для двумерного классификационного набора данных, показанного на рис. 2.23. Набор данных состоит из точек, обозначаемых маркерами двух типов. Каждому типу маркера соответствует свой класс, на каждый класс приходится по 75 точек данных. Назовем этот набор данных `two_moons`.

Построение дерева решений означает построение последовательности правил «если... то...», которая приводит нас к истинному ответу максимально коротким путем. В машинном обучении эти правила называются *тестами* (*tests*). Не путайте их с тестовым набором, который мы используем для проверки обобщающей способности нашей модели. Как правило, данные бывают представлены не только в виде бинарных признаков да/нет, как в примере с животными, но и в виде непрерывных признаков, как в двумерном наборе данных, показанном на рис. 2.23. Тесты, которые используются для непрерывных данных имеют вид «Признак  $i$  больше значения  $a$ »

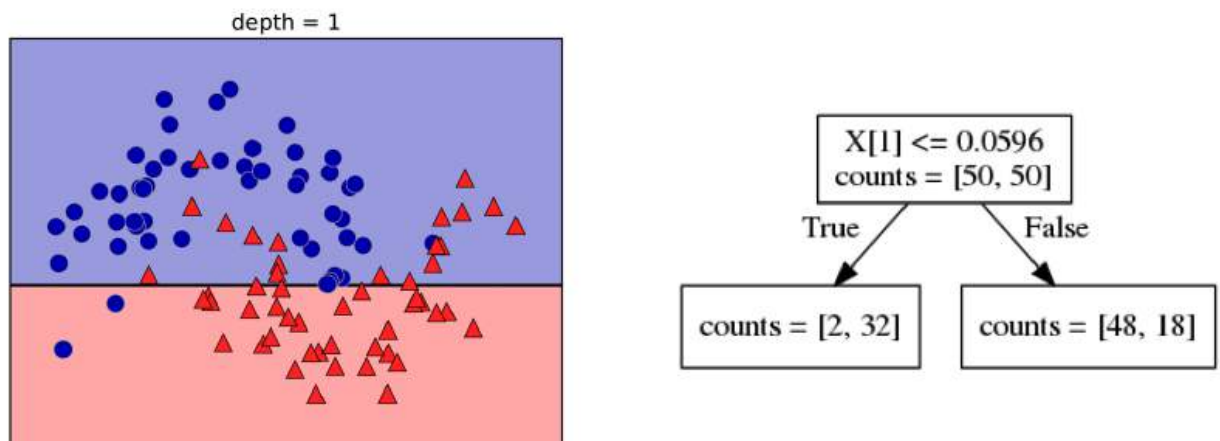
```
In[57]:  
mglearn.plots.plot_tree_progressive()
```



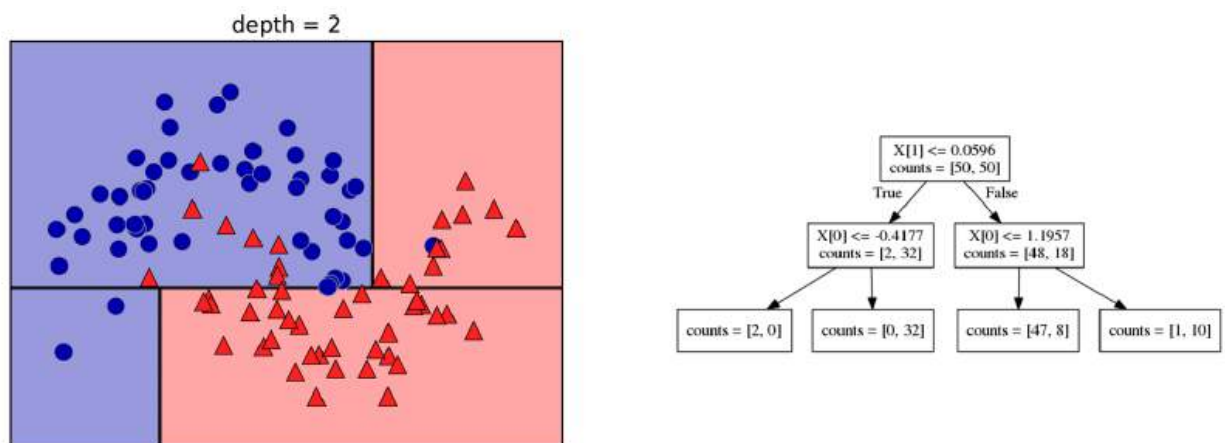
**Рис. 2.23** Набор данных `two_moons`, по которому будет построено дерево решений

Чтобы построить дерево, алгоритм перебирает все возможные тесты и находит тот, который является наиболее информативным с точки зрения прогнозирования значений целевой переменной. Рис. 2.24 показывает первый выбранный тест. Разделение набора данных по горизонтали в точке  $x[1]=0.0596$  дает наиболее полную информацию. Оно лучше всего разделяет точки класса 0 от точек класса 1. Верхний узел, также называемый *корнем* (*root*), представляет собой весь набор данных, состоящий из 50 точек, принадлежащих к классу 0, и 50 точек, принадлежащих к классу 1. Разделение выполняется путем тестирования  $x[1] \leq 0.0596$ , обозначенного черной линией. Если тест верен, точка назначается левому узлу, который содержит 2 точки, принадлежащие классу 0, и 32 точки, принадлежащие классу 1. В противном случае точка будет присвоена правому узлу, который содержит 48 точек, принадлежащих классу 0, и 18 точек, принадлежащих классу 1. Эти два

узла соответствуют верхней и нижней областям, показанным на рис. 2-24. Несмотря на то что первое разбиение довольно хорошо разделило два класса, нижняя область по-прежнему содержит точки, принадлежащие к классу 0, а верхняя область по-прежнему содержит точки, принадлежащие к классу 1. Мы можем построить более точную модель, повторяя процесс поиска наилучшего теста в обеих областях. Рис. 2.25 показывает, что следующее наиболее информативное разбиение для левой и правой областей основывается на  $x[0]$ .



**Рис. 2.24** Граница принятия решений, полученная с помощью дерева глубины 1 (слева) и соответствующее дерево решений (справа)

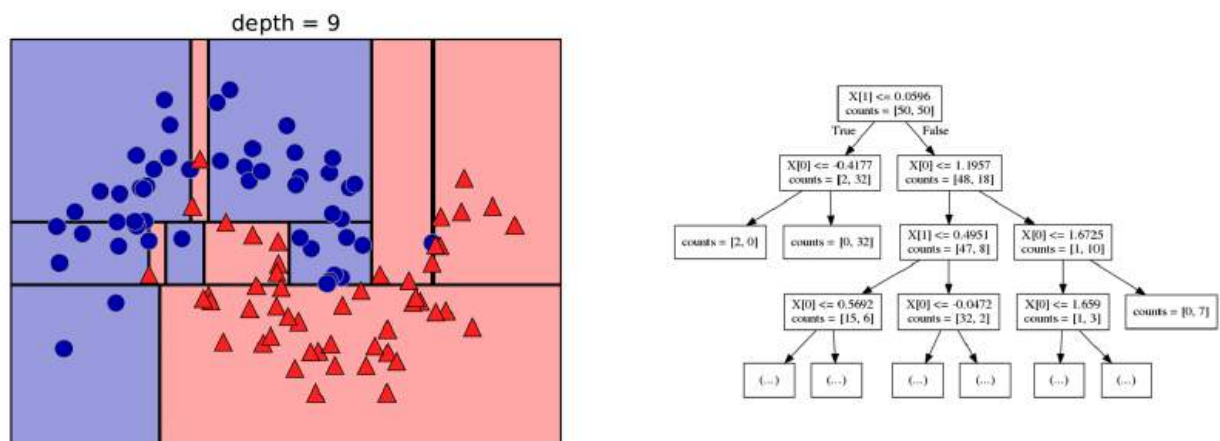


**Рис. 2.25** Граница принятия решений, полученная с помощью дерева глубины 2 (слева) и соответствующее дерево решений (справа)

Этот рекурсивный процесс строит в итоге бинарное дерево решений, в котором каждый узел соответствует определенному тесту. Кроме того, вы можете интерпретировать тест как разбиение части данных, рассматриваемое в данном случае вдоль одной оси. Это позволяет составить представление об алгоритме как способе выстроить иерархию разбиений. Поскольку каждый тест рассматривает только один признак,

области, получающиеся в результате разбиения, всегда имеют границы, параллельные осям.

Рекурсивное разбиение данных повторяется до тех пор, пока все точки данных в каждой области разбиения (каждом листе дерева решений) не будут принадлежать одному и тому же значению целевой переменной (классу или количественному значению). Лист дерева, который содержит точки данных, относящиеся к одному и тому же значению целевой переменной, называется *чистым* (*pure*). Итоговое разбиение для нашего набора данных показано на рис. 2.26.



**Рис. 2.26** Граница принятия решений, полученная с помощью дерева глубиной 9 (слева) и фрагмент соответствующего дерева (справа), полное дерево имеет довольно большой размер и его сложно визуализировать

Прогноз для новой точки данных получают следующим образом: сначала выясняют, в какой области разбиения пространства признаков находится данная точка, а затем определяют класс, к которому относится большинство точек в этой области (либо единственный класс в области, если лист является чистым). Область может быть найдена с помощью обхода дерева, начиная с корневого узла и путем перемещения влево или вправо, в зависимости от того, выполняется ли тест или нет.

Кроме того, можно использовать деревья для решения задач регрессии, используя точно такой же подход. Для получения прогноза мы обходим дерево на основе тестов в каждом узле и находим лист, в который попадает новая точка данных. Выходом для этой точки данных будет значение целевой переменной, усредненное по всем обучающим точкам в этом листе.

### Контроль сложности деревьев решений

Как правило, построение дерева, описанное здесь и продолжающееся до тех пор, пока все листья не станут чистыми, приводит к получению моделей, которые являются очень сложными и характеризуются сильным переобучением на обучающих данных. Наличие чистых листьев

означает, что дерево имеет 100%-ную правильность на обучающей выборке. Каждая точка обучающего набора находится в листе, который имеет правильный мажоритарный класс. Переобучение можно увидеть в левой части рис. 2.26. Видно, что точки, определяемые как точки класса 1, находятся посреди точек, принадлежащих к классу 0. С другой стороны, мы видим ряд точек, спрогнозированных как класс 1, вокруг точки, отнесенной к классу 0. Это не та граница принятия решений, которую мы могли бы себе представить. Здесь граница принятия решений фокусируется больше на отдельных точках-выбросах, которые находятся слишком далеко от остальных точек данного класса.

Есть две общераспространенные стратегии, позволяющие предотвратить переобучение. Первая стратегия – ранняя остановка построения дерева, называемая *предварительной обрезкой (pre-pruning)*. Вторая стратегия – построение дерева с последующим удалением или сокращением малоинформативных узлов, называемое *пост-обрезкой (post-pruning)* или просто *обрезкой (pruning)*. Возможные критерии предварительной обрезки включают в себя ограничение максимальной глубины дерева, ограничение максимального количества листьев или минимальное количество наблюдений в узле, необходимое для разбиения.

В библиотеке `scikit-learn` деревья решений реализованы в классах `DecisionTreeRegressor` и `DecisionTreeClassifier`. Обратите внимание, в `scikit-learn` реализована лишь предварительная обрезка.

Давайте более детально посмотрим, как работает предварительная обрезка на примере набора данных `Breast Cancer`. Как всегда, мы импортируем набор данных и разбиваем его на обучающую и тестовую части. Затем мы строим модель, используя настройки по умолчанию для построения полного дерева (выращиваем дерево до тех пор, пока все листья не станут чистыми). Зафиксируем `random_state` для воспроизводимости результатов:

```
In[58]:
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Правильность на обучающем наборе: {:.3f}".format(tree.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(tree.score(X_test, y_test)))

Out[58]:
Правильность на обучающем наборе: 1.000
Правильность на тестовом наборе: 0.937
```

Как и следовало ожидать, правильность на обучающем наборе составляет 100%, поскольку листья являются чистыми. Дерево имеет



глубину, как раз достаточную для того, чтобы прекрасно запомнить все метки обучающих данных. Правильность на тестовом наборе немного хуже, чем при использовании ранее рассмотренных линейных моделей, правильность которых составляла около 95%.

Если не ограничить глубину, дерево может быть сколь угодно глубоким и сложным. Поэтому необрезанные деревья склонны к переобучению и плохо обобщают результат на новые данные. Теперь давайте применим к дереву предварительную обрезку, которая остановит процесс построения дерева до того, как мы идеально подгоним модель к обучающим данным. Один из вариантов – остановка процесса построения дерева по достижении определенной глубины. Здесь мы установим `max_depth=4`, то есть можно задать только четыре последовательных вопроса (см. рис. 2.24 и 2.26). Ограничение глубины дерева уменьшает переобучение. Это приводит к более низкой правильности на обучающем наборе, но улучшает правильность на тестовом наборе:

```
In[59]:
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(tree.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(tree.score(X_test, y_test)))

Out[59]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.951
```

## Анализ деревьев решений

Мы можем визуализировать дерево, используя функцию `export_graphviz` из модуля `tree`. Она записывает файл в формате `.dot`, который является форматом текстового файла, предназначенным для описания графиков. Мы можем задать цвет узлам, чтобы выделить класс, набравший большинство в каждом узле, и передать имена классов и признаков, чтобы дерево было правильно размечено:

```
In[60]:
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

Мы можем прочитать этот файл и визуализировать его, как показано на рис. 2.27, используя модуль `graphviz`<sup>15</sup> (или любую другую программу, которая может читать файлы с расширением `.dot`):

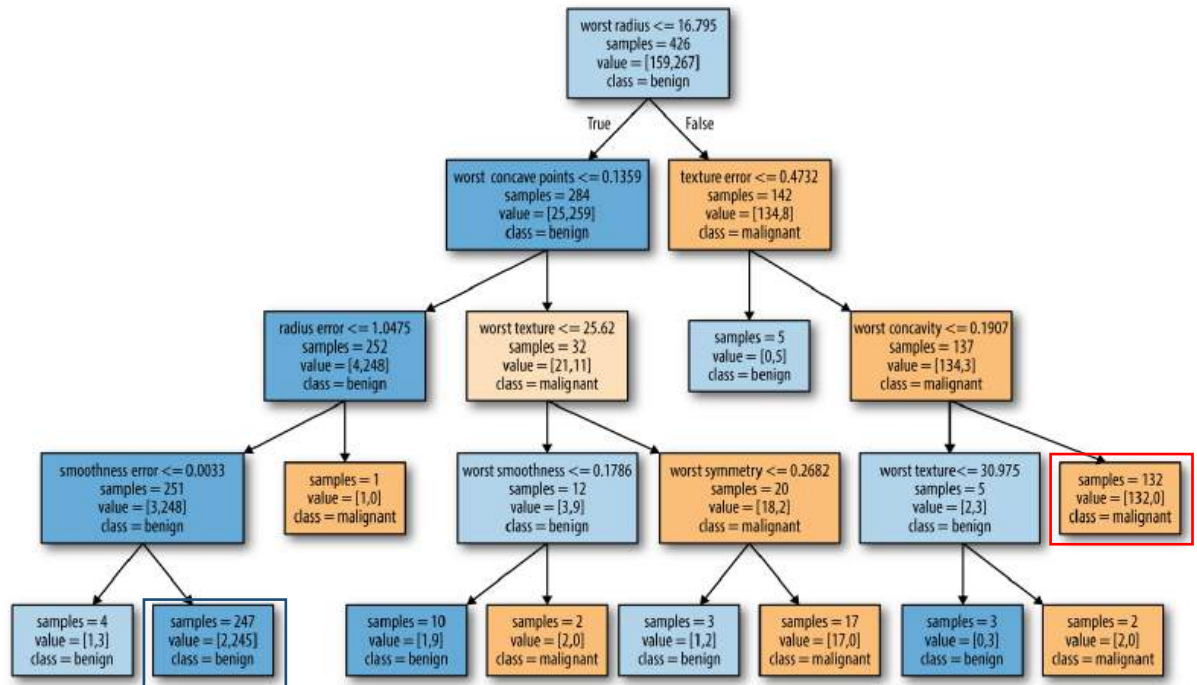
---

<sup>15</sup> Если вы используете Anaconda под Windows, то необходимо установить conda-пакет `graphviz` и pip-пакет `graphviz`:

```
conda install -c anaconda graphviz=2.38.0
```

```
In[61]:
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```



**Рис. 2.27** Визуализация дерева решений, построенного на наборе данных Breast Cancer

Как вариант, можно построить диаграмму дерева и записать ее в файл *.pdf*. Дополнительно нам потребуется модуль `pydotplus`.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn import tree
from sklearn.tree import export_graphviz
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
clf = tree.DecisionTreeClassifier(max_depth=4, random_state=0)
clf = clf.fit(X_train, y_train)

import pydotplus
dot_data = tree.export_graphviz(clf, out_file=None)
```

---

```
pip install graphviz
```

Затем в переменной окружения PATH необходимо прописать полный путь к установленной папке `graphviz`. В Windows 7 для этого нажмите кнопку **Пуск**, выберите **Панель управления**. Дважды нажмите на **Система**, затем выберите **Дополнительные параметры системы**. Во вкладке **Дополнительно** нажмите на **Переменные среды**. Выберите **Path** и нажмите на **Изменить**. В поле **Значение переменной** введите путь к папке `graphviz` (например, `C:\Anaconda3\Library\bin\graphviz`). – *Прим. пер.*



```
graph = pydotplus.graph_from_dot_data(dot_data)
graph.write_pdf("cancer.pdf")
```

Можно построить визуализацию дерева с помощью функции `Image` интерактивной оболочки `IPython`:

```
from IPython.display import Image
dot_data = tree.export_graphviz(clf, out_file=None,
                               feature_names=cancer.feature_names,
                               class_names=cancer.target_names,
                               filled=True, rounded=True,
                               special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```

Визуализация дерева дает более глубокое представление о том, как алгоритм делает прогнозы и является хорошим примером алгоритма машинного обучения, который легко объяснить неспециалистам. Однако, как показано здесь, даже при глубине 4 дерево может стать немного громоздким. Деревья с большим значением глубины (деревья глубиной 10 – не редкость) еще труднее понять. Один из полезных способов исследования дерева заключается в том, чтобы выяснить, какие узлы содержат наибольшее количество данных. Параметр `samples`, выводимый в каждом узле на рис. 2.27, показывает общее количество примеров в узле, тогда как параметр `value` показывает количество примеров в каждом классе. Проследовав по правой ветви, отходящей от корневого узла, мы видим, что правилу `worst radius > 16.795` соответствует узел, который содержит 134 случая злокачественной опухоли и лишь 8 случаев доброкачественной опухоли. Далее дерево выполняет серию более точных разбиений оставшихся 142 случаев. Из 142 случаев, которые при первоначальном разбиении были записаны в правый узел, почти все (132) в конечном итоге попали в правый лист (для удобства выделен красной рамкой).

Проследовав по левой ветви, отходящей от корневого узла, мы видим, что правилу `worst radius <= 16.795` соответствует узел, который содержит 25 случаев злокачественной опухоли и 259 случаев доброкачественной опухоли. Почти все случаи доброкачественной опухоли попадают во второй лист справа (для удобства выделен синей рамкой), остальные случаи распределяются по нескольким листьям, содержащим очень мало наблюдений.

### Важность признаков в деревьях

Вместо того, чтобы просматривать все дерево, что может быть обременительно, есть некоторые полезные параметры, которые мы можем использовать как итоговые показатели работы дерева. Наиболее часто используемым показателем является *важность признаков* (*feature importance*), которая оценивает, насколько важен каждый признак с

точки зрения получения решений. Это число варьирует в диапазоне от 0 до 1 для каждого признака, где 0 означает «не используется вообще», а 1 означает, что «отлично предсказывает целевую переменную». Важности признаков в сумме всегда дают 1:

```
In[62]:
print("Важности признаков:\n{}".format(tree.feature_importances_))

Out[62]:
Важности признаков
[ 0.          0.          0.          0.          0.          0.          0.
  0.          0.          0.          0.01019737  0.04839825  0.          0.
  0.0024156   0.          0.          0.          0.          0.
  0.72682851  0.0458159   0.          0.          0.0141577   0.          0.018188
  0.1221132   0.01188548  0.          ]
```

Приведенная сводка не совсем удобна, поскольку мы не знаем, каким именно признакам соответствуют приведенные важности. Чтобы исправить это, воспользуемся программным кодом, приведенным ниже:

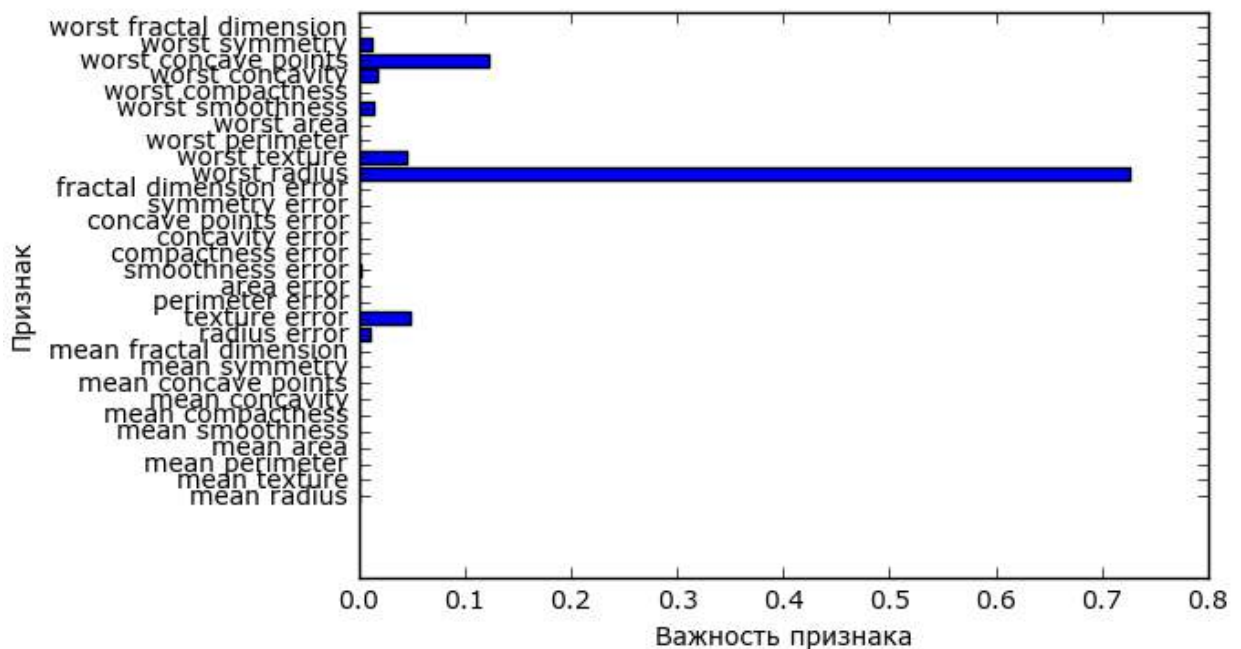
```
for name, score in zip(cancer["feature_names"], tree.feature_importances_):
    print(name, score)

mean radius 0.0
mean texture 0.0
mean perimeter 0.0
mean area 0.0
mean smoothness 0.0
mean compactness 0.0
mean concavity 0.0
mean concave points 0.0
mean symmetry 0.0
mean fractal dimension 0.0
radius error 0.0101973682021
texture error 0.0483982536186
perimeter error 0.0
area error 0.0
smoothness error 0.00241559508532
compactness error 0.0
concavity error 0.0
concave points error 0.0
symmetry error 0.0
fractal dimension error 0.0
worst radius 0.72682850946
worst texture 0.0458158970889
worst perimeter 0.0
worst area 0.0
worst smoothness 0.0141577021047
worst compactness 0.0
worst concavity 0.0181879968645
worst concave points 0.122113199265
worst symmetry 0.0118854783101
worst fractal dimension 0.0
```

Мы можем визуализировать важности признаков аналогично тому, как мы визуализируем коэффициенты линейной модели (рис. 2.28):

```
In[63]:
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Важность признака")
    plt.ylabel("Признак")

plot_feature_importances_cancer(tree)
```



**Рис. 2.28** Важности признаков, вычисленные с помощью дерева решений для набора данных Breast Cancer

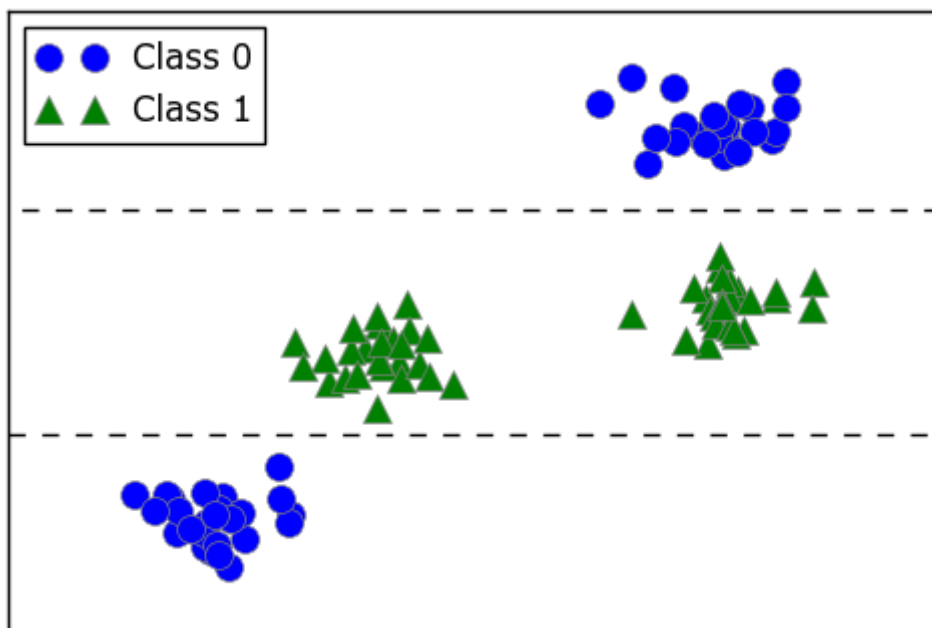
Здесь мы видим, что признак, использованный в самом верхнем разбиении («worst radius»), на данный момент является наиболее важным. Это подтверждает наш вывод о том, что уже на первом уровне два класса достаточно хорошо разделены.

Однако, если признак имеет низкое значение `feature_importance_`, это не значит, что он неинформативен. Это означает только то, что данный признак не был выбран деревом, поскольку, вероятно, другой признак содержит ту же самую информацию.

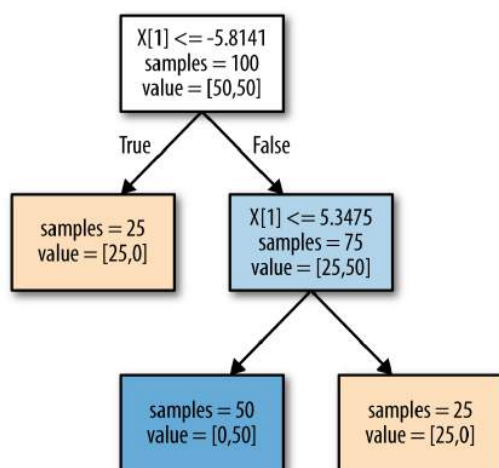
В отличие от коэффициентов линейных моделей важности признаков всегда положительны и они не указывают на взаимосвязь с каким-то конкретным классом. Важности признаков говорят нам, что «worst radius» важен, но мы не знаем, является ли высокое значение радиуса признаком доброкачественной или злокачественной опухоли. На самом деле, найти такую очевидную взаимосвязь между признаками и классом невозможно, что можно проиллюстрировать на следующем примере (рис. 2.29 и 2.30):

```
In[64]:  
tree = mglearn.plots.plot_tree_not_monotone()  
display(tree)
```

```
Out[64]:  
Feature importances: [ 0.  1.]
```



**Рис. 2.29** Двумерный массив данных, в котором признак имеет немонотонную взаимосвязь с меткой класса, и границы принятия решений, найденные с помощью дерева



**Рис. 2.30** Дерево решений для набора данных, показанном на рис. 2.29

График показывает набор данных с двумя признаками и двумя классами. Здесь вся информация содержится в  $X[1]$ , а  $X[0]$  не используется вообще. Но взаимосвязь между  $X[1]$  и целевым классом не является монотонной, то есть мы не можем сказать, что «высокое

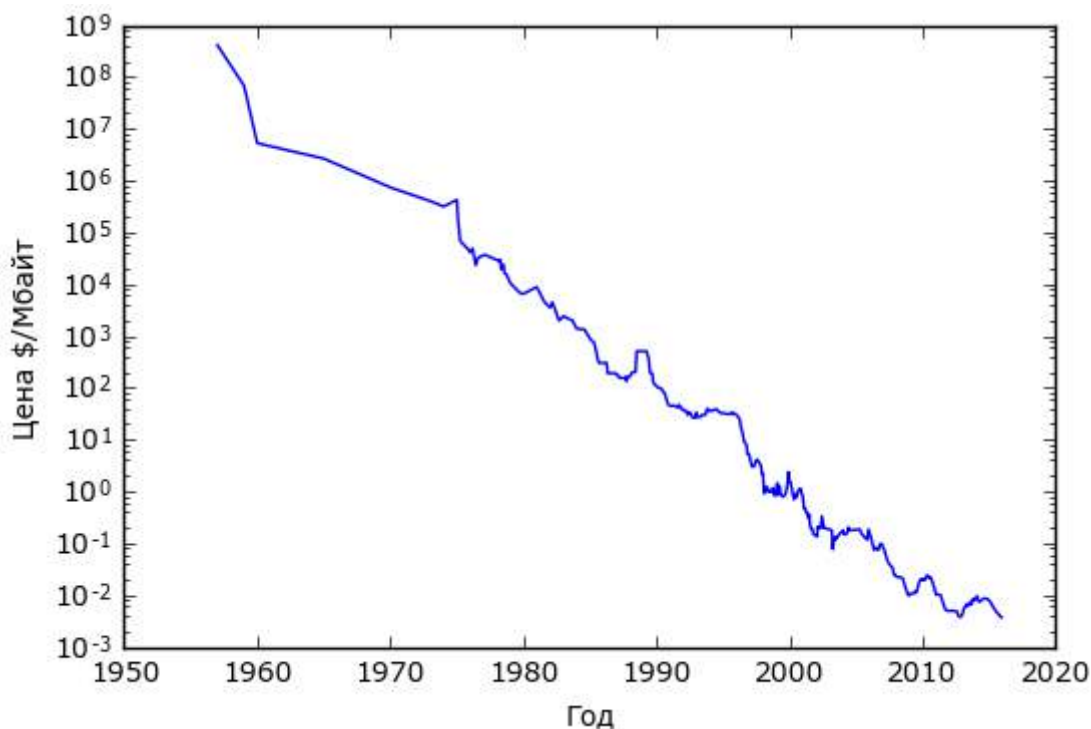
значение  $X[0]$  означает класс 0, а низкое значение означает класс 1» (или наоборот).

Несмотря на то что мы сосредоточились здесь на деревьях классификации, все вышесказанное верно и для деревьев регрессии, которые реализованы в `DecisionTreeRegressor`. Применение и анализ деревьев регрессии очень схожи с применением и анализом деревьев классификации. Однако существует одна особенность использования деревьев регрессии, на которую нужно указать. `DecisionTreeRegressor` (и все остальные регрессионные модели на основе дерева) не умеет *экстраполировать* или делать прогнозы вне диапазона значений обучающих данных.

Давайте детальнее рассмотрим это, воспользовавшись набором данных `RAM Price` (содержит исторические данные о ценах на компьютерную память). Рис. 2.31 визуализирует этот набор данных<sup>16</sup>, дата отложена по оси x, а цена одного мегабайта оперативной памяти в соответствующем году – по оси y:

```
In[65]:
import pandas as pd
ram_prices = pd.read_csv("C:/Data/ram_price.csv")

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Год")
plt.ylabel("Цена $/Мбайт")
```



**Рис. 2.31** Историческое развитие цен на RAM по логарифмической шкале

<sup>16</sup> Скачайте набор данных `ram_price.csv` по [ссылке](#) и перенесите его в папку `Data` на диске `C`. – Прим. пер.

Обратите внимание на логарифмическую шкалу оси y. При логарифмическом преобразовании взаимосвязь выглядит вполне линейной и таким образом становится легко прогнозируемой, за исключением некоторых всплесков.

Мы будем прогнозировать цены на период после 2000 года, используя исторические данные до этого момента, единственным признаком будут даты. Мы сравним две простые модели: `DecisionTreeRegressor` и `LinearRegression`. Мы отмасштабируем цены, используя логарифм, таким образом, взаимосвязь будет относительно линейной. Это несущественно для `DecisionTreeRegressor`, однако существенно для `LinearRegression` (мы рассмотрим ее более подробно в главе 4). После обучения модели и получения прогнозов мы применим экспоненцирование, чтобы обратить логарифмическое преобразование. Мы получим и визуализируем прогнозы для всего набора данных, но для количественной оценки мы будем рассматривать только тестовый набор:

```
In[66]:
from sklearn.tree import DecisionTreeRegressor
# используем исторические данные для прогнозирования цен после 2000 года
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# прогнозируем цены по датам
X_train = data_train.date[:, np.newaxis]
# мы используем логпреобразование, что получить простую взаимосвязь между данными и откликом
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)

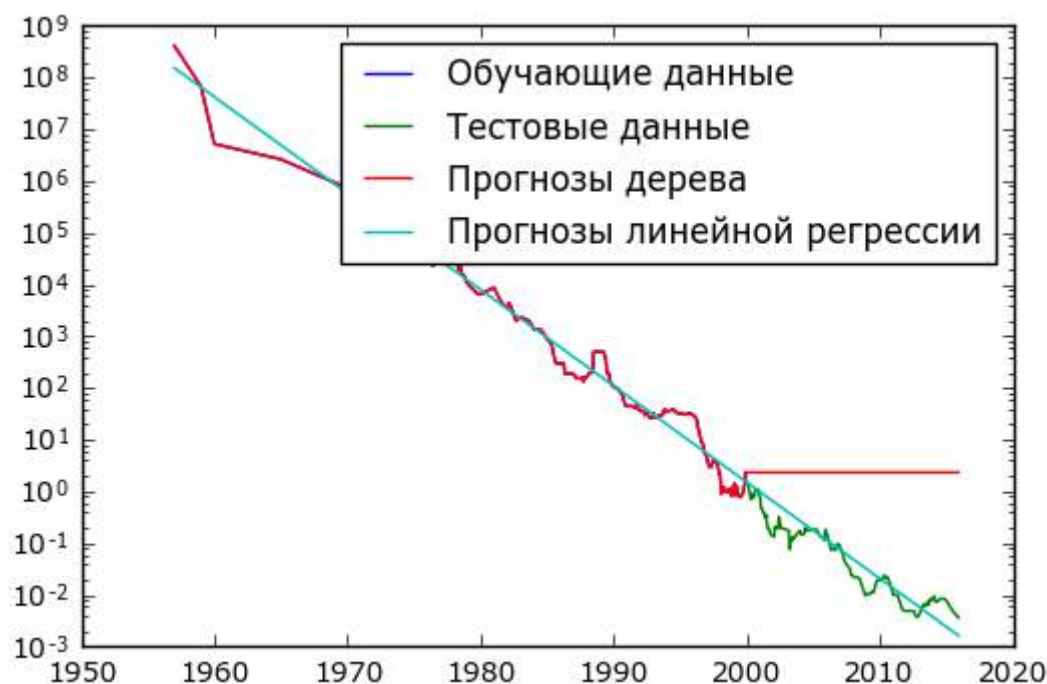
# прогнозируем по всем данным
X_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

# экспоненцируем, чтобы обратить логарифмическое преобразование
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

Рис. 2.32, созданный здесь, сравнивает прогнозы дерева решений и линейной регрессии с реальными.

```
In[67]:
plt.semilogy(data_train.date, data_train.price, label="Обучающие данные")
plt.semilogy(data_test.date, data_test.price, label="Тестовые данные")
plt.semilogy(ram_prices.date, price_tree, label="Прогнозы дерева")
plt.semilogy(ram_prices.date, price_lr, label="Прогнозы линейной регрессии")
plt.legend()
```



**Рис. 2.32** Сравнение прогнозов линейной модели и прогнозов дерева регрессии для набора данных RAM price

Разница между моделями получилась весьма впечатляющая. Линейная модель аппроксимирует данные с помощью уже известной нам прямой линии. Эта линия дает достаточно хороший прогноз для тестовых данных (период после 2000 года), при этом сглаживая некоторые всплески в обучающих и тестовых данных. С другой стороны, модель дерева прекрасно прогнозирует на обучающих данных. Здесь мы не ограничивали сложность дерева, поэтому она полностью запомнила весь набор данных. Однако, как только мы выходим из диапазона значений, известных модели, модель просто продолжает предсказывать последнюю известную точку. Дерево не способно генерировать «новые» ответы, выходящие за пределы значений обучающих данных. Этот недостаток относится ко всем моделям на основе деревьев решений.<sup>17</sup>

### Преимущества, недостатки и параметры

Как уже говорилось выше, параметры, которые контролируют сложность модели в деревьях решений — это параметрами предварительной обрезки дерева, которые останавливают построение дерева, прежде чем оно достигнет максимального размера. Обычно, чтобы предотвратить

<sup>17</sup> На самом деле с помощью деревьев решений можно получать очень точные прогнозы (например, предсказать, будет ли цена повышаться или понижаться). Суть этого примера была не в том, чтобы показать, что деревья являются плохой моделью для временных рядов, а в том, чтобы конкретно показать, как деревья делают прогнозы.



переобучение, достаточно выбрать одну из стратегий предварительной обрезки – настроить `max_depth`, `max_leaf_nodes` или `min_samples_leaf`.

По сравнению со многими алгоритмами, обсуждавшимися до сих пор, деревья решений обладают двумя преимуществами: полученная модель может быть легко визуализирована и понята неспециалистами (по крайней мере это верно для небольших деревьев) и деревья не требуют масштабирования данных. Поскольку каждый признак обрабатывается отдельно, а возможные разбиения данных не зависят от масштабирования, алгоритмы деревьев решений не нуждаются в таких процедурах предварительной обработки, как нормализация или стандартизация признаков. Деревья решений хорошо работают, когда у вас есть признаки, измеренные в совершенно разных шкалах, или когда ваши данные представляют смесь бинарных и непрерывных признаков.

Основным недостатком деревьев решений является то, что даже при использовании предварительной обрезки, они склонны к переобучению и имеют низкую обобщающую способность. Поэтому в большинстве случаев, как правило, вместо одиночного дерева решений используются ансамбли деревьев, которые мы обсудим далее.

## Ансамбли деревьев решений

*Ансамбли (ensembles)* – это методы, которые сочетают в себе множество моделей машинного обучения, чтобы в итоге получить более мощную модель. Существует много моделей машинного обучения, которые принадлежат к этой категории, но есть две ансамблевых модели, которые доказали свою эффективность на самых различных наборах данных для задач классификации и регрессии, обе используют деревья решений в качестве строительных блоков: случайный лес деревьев решений и градиентный бустинг деревьев решений.

### Случайный лес

Как мы только что отметили, основным недостатком деревьев решений является их склонность к переобучению. Случайный лес является одним из способов решения этой проблемы. По сути случайный лес – это набор деревьев решений, где каждое дерево немного отличается от остальных. Идея случайного леса заключается в том, что каждое дерево может довольно хорошо прогнозировать, но скорее всего переобучается на части данных. Если мы построим много деревьев, которые хорошо работают и переобучаются с разной степенью, мы можем уменьшить переобучение путем усреднения их результатов. Уменьшение переобучения при сохранении прогнозной силы деревьев можно проиллюстрировать с помощью строгой математики.



Для реализации вышеизложенной стратегии нам нужно построить большое количество деревьев решений. Каждое дерево должно на приемлемом уровне прогнозировать целевую переменную и должно отличаться от других деревьев. Случайные леса получили свое название из-за того, что в процесс построения деревьев была внесена случайность, призванная обеспечить уникальность каждого дерева. Существует две техники, позволяющие получить рандомизированные деревья в рамках случайного леса: сначала выбираем точки данных (наблюдения), которые будут использоваться для построения дерева, а затем отбираем признаки в каждом разбиении. Давайте разберем этот процесс более подробно.

### Построение случайного леса

Для построения модели случайных лесов необходимо определиться с количеством деревьев (параметр `n_estimators` для `RandomForestRegressor` или `RandomForestClassifier`). Допустим, мы хотим построить 10 деревьев. Эти деревья будут построены совершенно независимо друг от друга, и алгоритм будет случайным образом отбирать признаки для построения каждого дерева, чтобы получить непохожие друг на друга деревья. Для построения дерева мы сначала сформируем *бутстреп-выборку* (*bootstrap sample*) наших данных. То есть из `n_samples` примеров мы случайным образом выбираем пример с возвращением `n_samples` раз (поскольку отбор с возвращением, то один и тот же пример может быть выбран несколько раз). Мы получаем выборку, которая имеет такой же размер, что и исходный набор данных, однако некоторые примеры будут отсутствовать в нем (примерно одна треть), а некоторые попадут в него несколько раз.

Чтобы проиллюстрировать это, предположим, что мы хотим создать бутстреп-выборку списка `['a', 'b', 'c', 'd']`. Возможная бутстреп-выборка может выглядеть как `['b', 'd', 'd', 'c']`. Другой возможной бутстреп-выборкой может быть `['d', 'a', 'd', 'a']`.

Далее на основе этой сформированной бутстреп-выборки строится дерево решений. Однако алгоритм, который мы описывали для дерева решений, теперь слегка изменен. Вместо поиска наилучшего теста для каждого узла, алгоритм для разбиения узла случайным образом отбирает подмножество признаков и затем находит наилучший тест, используя один из этих признаков. Количество отбираемых признаков контролируется параметром `max_features`. Отбор подмножества признаков повторяется отдельно для каждого узла, поэтому в каждом узле дерева может быть принято решение с использованием «своего» подмножества признаков.

Использование бутстрепа приводит к тому, что деревья решений в случайном лесе строятся на немного отличающихся между собой

бутстреп-выборках. Из-за случайного отбора признаков в каждом узле все расщепления в деревьях будут основано на отличающихся подмножествах признаков. Вместе эти два механизма приводят к тому, что все деревья в случайном лесе отличаются друг от друга.

Критическим параметром в этом процессе является `max_features`. Если мы установим `max_features` равным `n_features`, это будет означать, что в каждом разбиении могут участвовать все признаки набора данных, и в отбор признаков не будет привнесена случайность (впрочем, случайность в силу использования бутстрепа остается). Если мы установим `max_features` равным 1, это означает, что при разбиении не будет никакого отбора признаков для тестирования вообще, будет осуществляться поиск с учетом различных пороговых значений для случайно выбранного признака. Таким образом, высокое значение `max_features` означает, что деревья в случайном лесе будут весьма схожи между собой и они смогут легко аппроксимировать данные, используя наиболее дискриминирующие признаки. Низкое значение `max_features` означает, что деревья в случайном лесе будут сильно отличаться друг от друга и, возможно, каждое дерево будет иметь очень большую глубину, чтобы хорошо соответствовать данным.

Чтобы дать прогноз для случайного леса, алгоритм сначала дает прогноз для каждого дерева в лесе. Для регрессии мы можем усреднить эти результаты, чтобы получить наш окончательный прогноз. Для классификации используется стратегия «мягкого голосования». Это означает, что каждый алгоритм дает «мягкий» прогноз, вычисляя вероятности для каждого класса. Эти вероятности усредняются по всем деревьям и прогнозируется класс с наибольшей вероятностью.

## Анализ случайного леса

Давайте применим случайный лес, состоящий из пяти деревьев, к набору данных `two_moons`, который мы изучали ранее:

```
In[68]:
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

Деревья, которые строятся в рамках случайного леса, сохраняются в атрибуте `estimator_`. Давайте визуализируем границы принятия решений, полученные каждым деревом, а затем выведем агрегированный прогноз, выданный лесом (рис. 2.33):

```

In[69]:
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Дерево {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Случайный лес")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)

```

На рисунках отчетливо видно, что границы принятия решений, полученные с помощью пяти деревьев, существенно различаются между собой. Каждое дерево совершает ряд ошибок, поскольку из-за бутстрепа некоторые точки исходного обучающего набора фактически не были включены в обучающие наборы, по которым строились деревья.

В отличие от отдельных деревьев случайный лес переобучается в меньшей степени и дает гораздо более чувствительную (гибкую) границу принятия решений. В реальных примерах используется гораздо большее количество деревьев (часто сотни или тысячи), что приводит к получению еще более чувствительной границы.

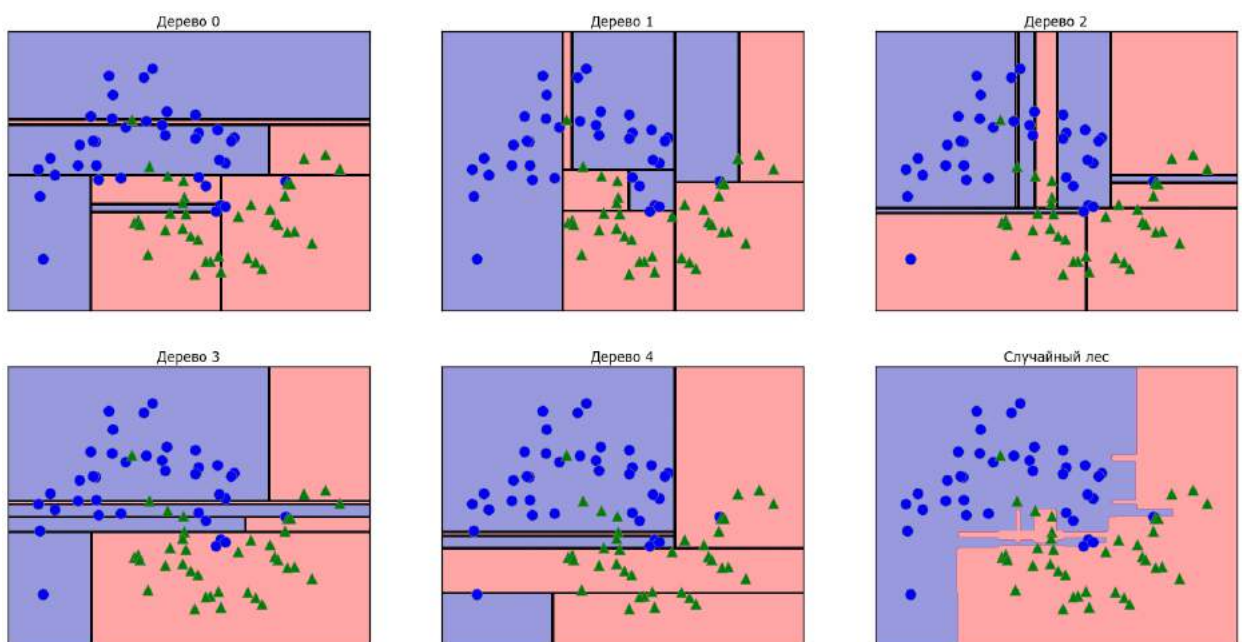


Рис. 2.33 Границы принятия решений, найденные пятью рандомизированными деревьями решений, и граница принятия решений, полученная путем усреднения их спрогнозированных вероятностей

В качестве еще одного примера давайте построим случайный лес, состоящий из 100 деревьев, на наборе данных Breast Cancer:

```

In[70]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(forest.score(X_train, y_train)))

```

```
print("Правильность на тестовом наборе: {:.3f}".format(forest.score(X_test, y_test)))
```

Out[70]:

Правильность на обучающем наборе: 1.000

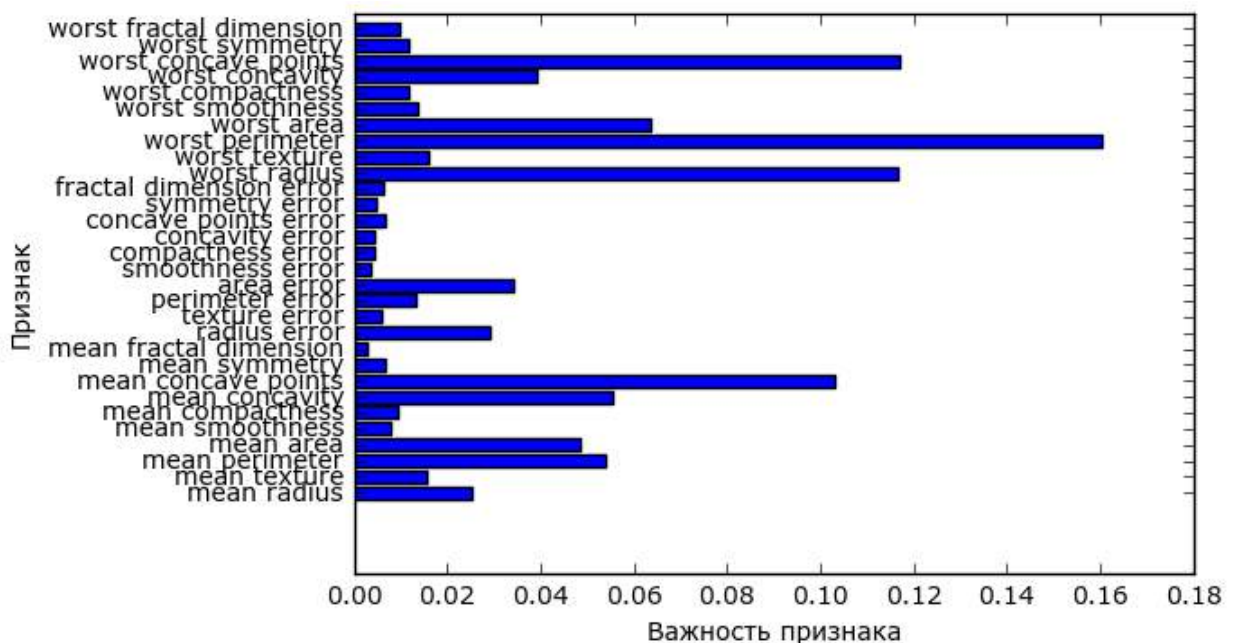
Правильность на тестовом наборе: 0.972

Без настройки каких-либо параметров случайный лес дает нам правильность 97%, это лучше результата линейных моделей или одиночного дерева решений. Мы могли бы отрегулировать настройку `max_features` или применить предварительную обрезку, как это делали для одиночного дерева решений. Однако часто параметры случайного леса, выставленные по умолчанию, работают уже сами по себе достаточно хорошо.

Как и дерево решений, случайный лес позволят вычислить важности признаков, которые рассчитываются путем агрегирования значений важности по всем деревьям леса. Как правило, важности признаков, вычисленные случайным лесом, являются более надежным показателем, чем важности, вычисленные одним деревом. Посмотрите на рис. 2.34.

In[71]:

```
def plot_feature_importances_cancer(model):  
    n_features = cancer.data.shape[1]  
    plt.barh(range(n_features), model.feature_importances_, align='center')  
    plt.yticks(np.arange(n_features), cancer.feature_names)  
    plt.xlabel("Важность признака")  
    plt.ylabel("Признак")  
plot_feature_importances_cancer(forest)
```



**Рис. 2.34** Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

На рисунке видно, что в отличие от одиночного дерева решения случайный лес вычисляет ненулевые значения важностей для гораздо большего числа признаков. Как и дерево решений, случайный лес также

присваивает высокое значение важности признаку «worst radius», однако в качестве наиболее информативного признака выбирает «worst perimeter». Случайность, лежащая в основе случайного леса, заставляет алгоритм рассматривать множество возможных интерпретаций. Это приводит к тому, что случайный лес дает гораздо более широкую картину данных, чем одиночное дерево.

### Преимущества, недостатки и параметры

В настоящее время случайные леса регрессии и классификации являются одним из наиболее широко используемых методов машинного обучения. Они обладают высокой прогнозной силой, часто дают хорошее качество модели без утомительной настройки параметров и не требуют масштабирования данных.

По сути случайные леса обладают всеми преимуществами деревьев решений, хотя и не лишены некоторых их недостатков. Одна из причин, в силу которой деревья решений еще используются до сих пор, – это компактное представление процесса принятия решений. Детальная интерпретация десятков или сотен деревьев невозможна в принципе, и, как правило, деревья в случайном лесу получаются более глубокими по сравнению с одиночными деревьями решений (из-за использования подмножеств признаков). Поэтому, если вам нужно в сжатом виде визуализировать процесс принятия решений для неспециалистов, одиночное дерево решений может быть оптимальным выбором. Несмотря на то, что построение случайных лесов на больших наборах данных может занимать определенное время, его можно легко распараллелить между несколькими ядрами процессора в компьютере. Если ваш компьютер оснащен многоядерным процессором (как почти все современные компьютеры), вы можете использовать параметр `n_jobs` для настройки количества используемых ядер. Использование большего количества процессорных ядер приведет к линейному росту скорости (при использовании двух ядер обучение случайного леса будет осуществляться в два раза быстрее), однако установка значения `n_jobs`, превышающего количество ядер, не поможет. Вы можете установить `n_jobs=-1`, чтобы использовать все ядра вашего процессора.

Вы должны помнить, что случайный лес по своей природе является рандомизированным алгоритмом и установка различных стартовых значений генератора псевдослучайных чисел (или вообще отказ от использования `random_state`) может кардинально изменить построение модели. Чем больше деревьев в лесу, тем более устойчивым он будет к изменению стартового значения. Если вы хотите получить результаты, которые потом нужно будет воспроизвести, важно зафиксировать `random_state`.

Случайный лес плохо работает на данных очень высокой размерности, разреженных данных, например, на текстовых данных. Для подобного рода данных линейные модели подходят больше. Случайный лес, как правило, хорошо работает даже на очень больших наборах данных, и обучение могут легко распараллелить между многочисленными процессорными ядрами в рамках мощного компьютера. Однако случайный лес требует больше памяти и медленнее обучается и прогнозирует, чем линейные модели. Если время и память имеют важное значение, имеет смысл вместо случайного леса использовать линейную модель.

Важными параметрами настройки являются `n_estimators`, `max_features` и опции предварительной обрезки деревьев, например, `max_depth`. Что касается `n_estimators`, большее значение всегда дает лучший результат. Усреднение результатов по большому количеству деревьев позволит получить более устойчивый ансамбль за счет снижения переобучения. Однако обратная сторона увеличения числа деревьев заключается в том, что с ростом количества деревьев требуется больше памяти и больше времени для обучения. Общее правило заключается в том, чтобы построить «столько, сколько позволяет ваше время/память».

Как было описано ранее, `max_features` случайным образом определяет признаки, используемые при разбиении в каждом дереве, а меньшее значение `max_features` уменьшает переобучение. В общем, лучше взять за правило использовать значения, выставленные по умолчанию: `max_features=sqrt(n_features)` для классификации и `max_features=n_features` для регрессии. Увеличение значений `max_features` или `max_leaf_nodes` иногда может повысить качество модели. Кроме того, оно может резко снизить требования к пространству на диске и времени вычислений в ходе обучения и прогнозирования.

### Градиентный бустинг деревьев регрессии (машины градиентного бустинга)

Градиентный бустинг деревьев регрессии – еще один ансамблевый метод, который объединяет в себе множество деревьев для создания более мощной модели. Несмотря на слово «регрессия» в названии, эти модели можно использовать для регрессии и классификации. В отличие от случайного леса, градиентный бустинг строит последовательность деревьев, в которой каждое дерево пытается исправить ошибки предыдущего. По умолчанию в градиентном бустинге деревьев регрессии отсутствует случайность, вместо этого используется строгая предварительная обрезка. В градиентном бустинге деревьев часто используются деревья небольшой глубины, от одного до пяти уровней,

что делает модель меньше с точки зрения памяти и ускоряет вычисление прогнозов.

Основная идея градиентного бустинга заключается в объединении множества простых моделей (в данном контексте известных под названием *слабые ученики* или *weak learners*), деревьев небольшой глубины. Каждое дерево может дать хорошие прогнозы только для части данных и таким образом для итеративного улучшения качества добавляется все большее количество деревьев.

Градиентный бустинг деревьев часто занимает первые строчки в соревнованиях по машинному обучению, а также широко используется в коммерческих сферах. В отличие от случайного леса он, как правило, немного более чувствителен к настройке параметров, однако при правильно заданных параметрах может дать более высокое значение правильности.

Помимо предварительной обрезки и числа деревьев в ансамбле, еще один важный параметр градиентного бустинга – это `learning_rate`, который контролирует, насколько сильно каждое дерево будет пытаться исправить ошибки предыдущих деревьев. Более высокая скорость обучения означает, что каждое дерево может внести более сильные корректировки и это позволяет получить более сложную модель. Добавление большего количества деревьев в ансамбль, осуществляемое за счет увеличения значения `n_estimators`, также увеличивает сложность модели, поскольку модель имеет больше шансов исправить ошибки на обучающем наборе.

Ниже приведен пример использования `GradientBoostingClassifier` на наборе данных Breast Cancer. По умолчанию используются 100 деревьев с максимальной глубиной 3 и скорости обучения 0.1:

```
In[72]:
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Out[72]:
Правильность на обучающем наборе: 1.000
Правильность на тестовом наборе: 0.958
```

Поскольку правильность на обучающем наборе составляет 100%, мы, вероятно, столкнулись с переобучением. Для уменьшения переобучения мы можем либо применить более сильную предварительную обрезку, ограничив максимальную глубину, либо снизить скорость обучения:



```
In[73]:
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))
```

```
Out[73]:
Правильность на обучающем наборе: 0.991
Правильность на тестовом наборе: 0.972
```

```
In[74]:
gbrt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbrt.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(gbrt.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(gbrt.score(X_test, y_test)))
```

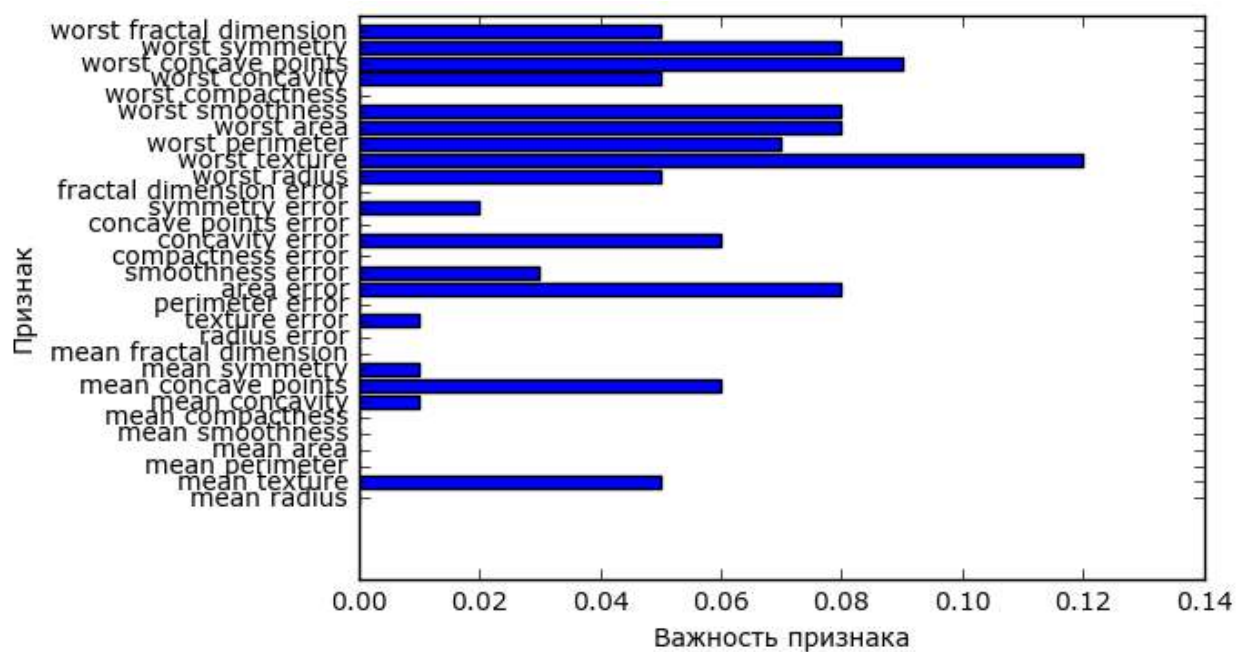
```
Out[74]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.965
```

Как и ожидалось, эти методы, направленные на уменьшение сложности модели, снижают правильность на обучающем наборе. В данном случае снижение максимальной глубины деревьев значительно улучшило модель, тогда как скорость обучения лишь незначительно повысило обобщающую способность.

И вновь, как и в случае с остальными моделями на основе деревьев, мы можем визуализировать важности признаков, чтобы получить более глубокое представление о нашей модели (рис. 2.35). Поскольку мы использовали 100 деревьев, вряд ли целесообразно проверять все деревья, даже если все они имеют глубину 1:

```
In[75]:
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Важность признака")
    plt.ylabel("Признак")
plot_feature_importances_cancer(gbrt)
```



**Рис. 2.35** Важности признаков, вычисленные случайным лесом для набора данных Breast Cancer

На рисунке видно, что важности признаков, вычисленные градиентным бустингом деревьев, в какой-то степени схожи с важностями признаков, полученными с помощью случайного леса, хотя градиентный бустинг полностью проигнорировал некоторые признаки.

Поскольку и градиентный бустинг и случайный лес хорошо работают на одних и тех же данных, общераспространенный подход заключается в том, чтобы сначала попытаться построить случайный лес, который дает вполне устойчивые результаты. Если случайный лес дает хорошее качество модели, однако время, отводимое на прогнозирование, на вес золота или важно выжать из модели максимальное значение правильности, выбор в пользу градиентного бустинга часто помогает решить эти задачи.

Если вы хотите применить градиентный бустинг для решения крупномасштабной задачи, возможно стоит обратиться к пакету `xgboost` и его Python-интерфейсу, который на многих наборах данных работает быстрее (а иногда и проще настраивается), чем реализация градиентного бустинга в `scikit-learn`.

### Преимущества, недостатки и параметры

Градиентный бустинг деревьев решений — одна из самых мощных и широко используемых моделей обучения с учителем. Его основной недостаток заключается в том, что он требует тщательной настройки параметров и для обучения может потребоваться много времени. Как и другие модели на основе дерева, алгоритм хорошо работает на данных, представляющих смесь бинарных и непрерывных признаков, не требуя

масштабирования. Как и остальные модели на основе дерева, он также плохо работает на высокоразмерных разреженных данных.

Основные параметры градиентного бустинга деревьев – это количество деревьев (`n_estimators`) и скорость обучения (`learning_rate`), контролирующая степень вклада каждого дерева в устранение ошибок предыдущих деревьев. Эти два параметра тесно взаимосвязаны между собой, поскольку более низкое значение `learning_rate` означает, что для построения модели аналогичной сложности необходимо большее количество деревьев. В отличие от случайного леса, в котором более высокое значение `n_estimators` всегда дает лучшее качество, увеличение значения `n_estimators` в градиентном бустинге дает более сложную модель, что может привести к переобучению. Общепринятая практика – подгонять `n_estimators` в зависимости от бюджета времени и памяти, а затем подбирать различные значения `learning_rate`.

Другим важным параметром является параметр `max_depth` (или, как альтернатива, `max_leaf_nodes`), направленный на уменьшение сложности каждого дерева. Обычно для моделей градиентного бустинга значение `max_depth` устанавливается очень низким, как правило, не больше пяти уровней.

## Ядерный метод опорных векторов

Следующий тип обучения с учителем, который мы обсудим, – это метод опорных векторов. Мы рассматривали использование линейного метода опорных векторов для задач классификации в разделе «Линейные модели для классификации». Ядерный метод опорных векторов (часто его просто называют SVM) – это расширение метода опорных векторов, оно позволяет получать более сложные модели, которые не сводятся к построению простых гиперплоскостей в пространстве. Несмотря на то что метод опорных векторов можно применять для задач классификации и регрессии, мы ограничимся классификацией, реализованной в `SVC`. Аналогичные принципы применяются в опорных векторах для регрессии и реализованы в `SVR`.

Математический аппарат ядерного метода опорных векторов сложен и выходит за рамки данной книги. Вы можете подробнее прочитать о нем в главе 12 книги Хасты, Тибширани и Фридмана [«Элементы статистического обучения»](#). Однако мы попытаемся дать вам некоторое представление об идеях, лежащих в основе этого метода.

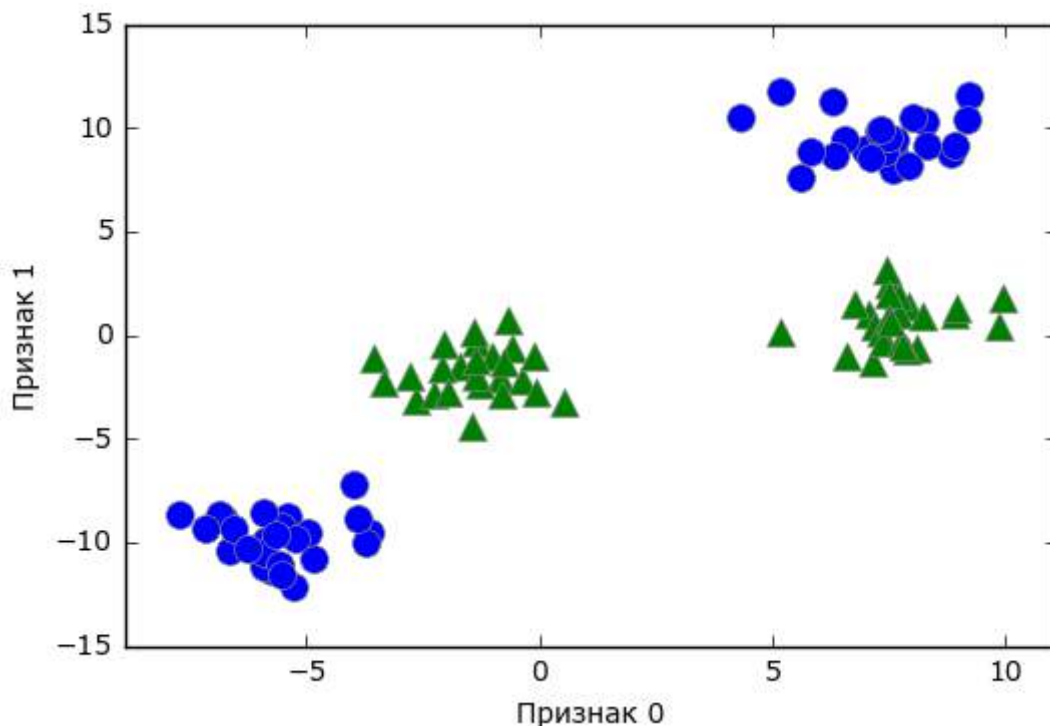
## Линейные модели и нелинейные признаки

На рис. 2.15 было видно, что в низкоразмерных пространствах линейные модели накладывают весьма жесткие ограничения, поскольку линии и гиперплоскости имеют ограниченную гибкость. Один из способов сделать линейную модель более гибкой – добавить новые признаки, например, добавить взаимодействия или полиномы входных признаков.

Давайте взглянем на синтетический набор данных, который мы использовали в разделе «Важность признаков в деревьях» (см. рис. 2.29):

```
In[76]:
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```



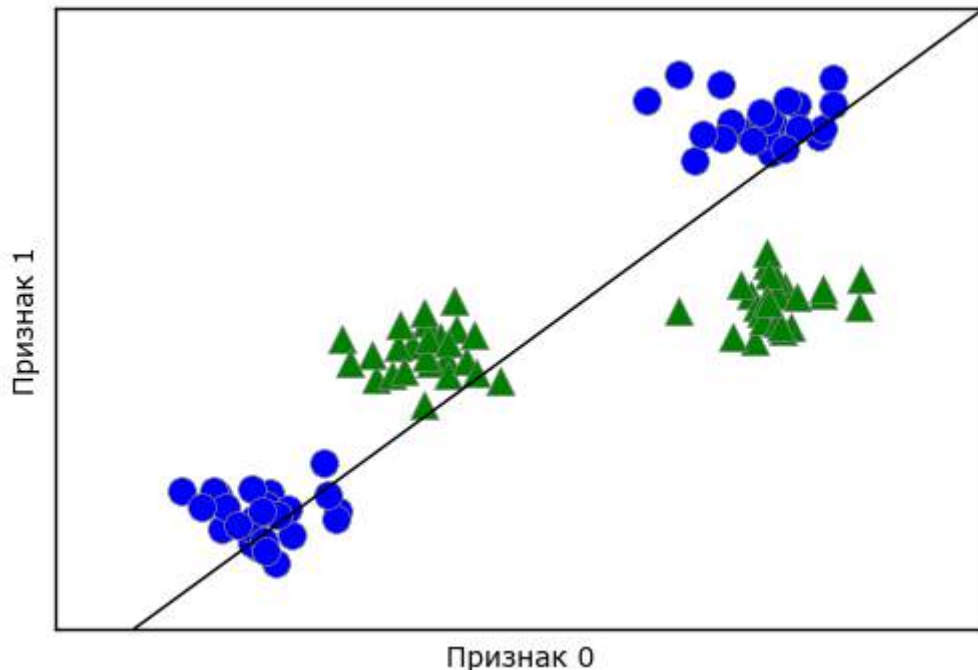
**Рис. 2.36** Набор данных с двухклассовой классификацией, в котором классы линейно неразделимы

Линейная модель классификации может отделить точки только с помощью прямой линии и не может дать хорошее качество для этого набора данных (см. рис. 2.37):

```
In[77]:
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

Теперь давайте расширим набор входных признаков, скажем, добавим в качестве нового признака `feature1 ** 2`, квадрат второго признака. Теперь каждую точку данных мы представим не в виде точки двумерного пространства `(feature0, feature1)`, а в виде точки трехмерного пространства `(feature0, feature1, feature1 ** 2)`.<sup>18</sup> Новое пространство признаков показано на рис. 2.38 в виде трехмерной диаграммы рассеяния:

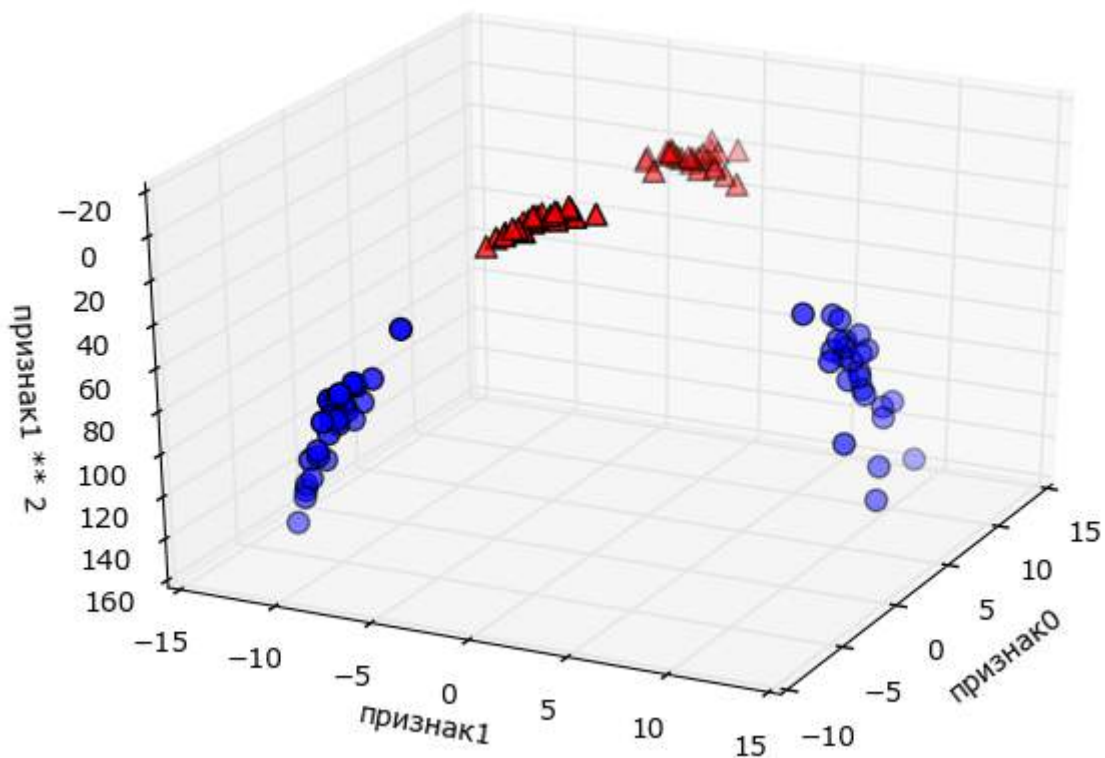


**Рис. 2.37** Граница принятия решений, найденная с помощью линейного SVM

```
In[78]:
# добавляем второй признак, возведенный в квадрат
X_new = np.hstack([X, X[:, 1:] ** 2])

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# визуализируем в 3D
ax = Axes3D(figure, elev=-152, azimuth=-26)
# сначала размещаем на графике все точки с y == 0, затем с y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60)
ax.set_xlabel("признак0")
ax.set_ylabel("признак1")
ax.set_zlabel("признак1 ** 2")
```

<sup>18</sup> Мы добавили этот признак в иллюстративных целях. Этот выбор не является принципиально важным.



**Рис. 2.38** Расширение набора данных, показанного на рис. 2.37, за счет добавления третьего признака, полученного на основе признака 1

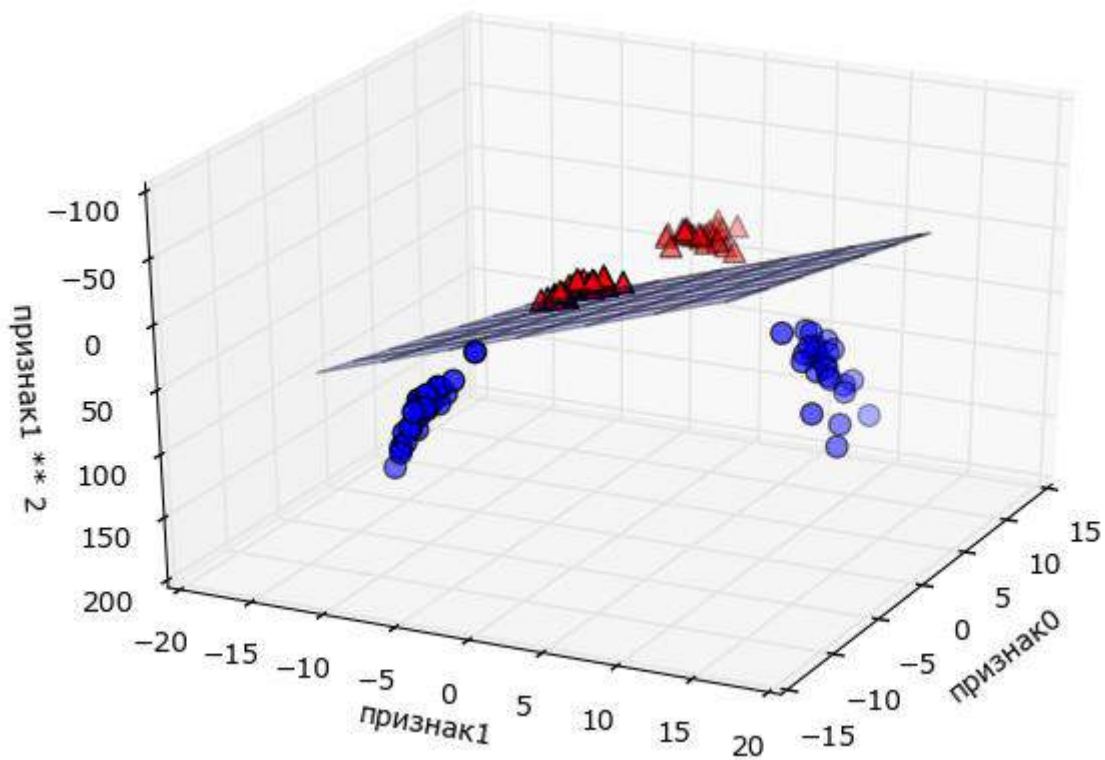
В новом представлении данных уже можно отделить два класса друг от друга, используя линейную модель, плоскость в трехмерном пространстве. Мы можем убедиться в этом, подогнав линейную модель к дополненным данным (см. рис. 2.39):

```
In[79]:
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

# показать границу принятия решений линейной модели
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azimuth=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60)

ax.set_xlabel("признак0")
ax.set_ylabel("признак1")
ax.set_zlabel("признак1 ** 2")
```

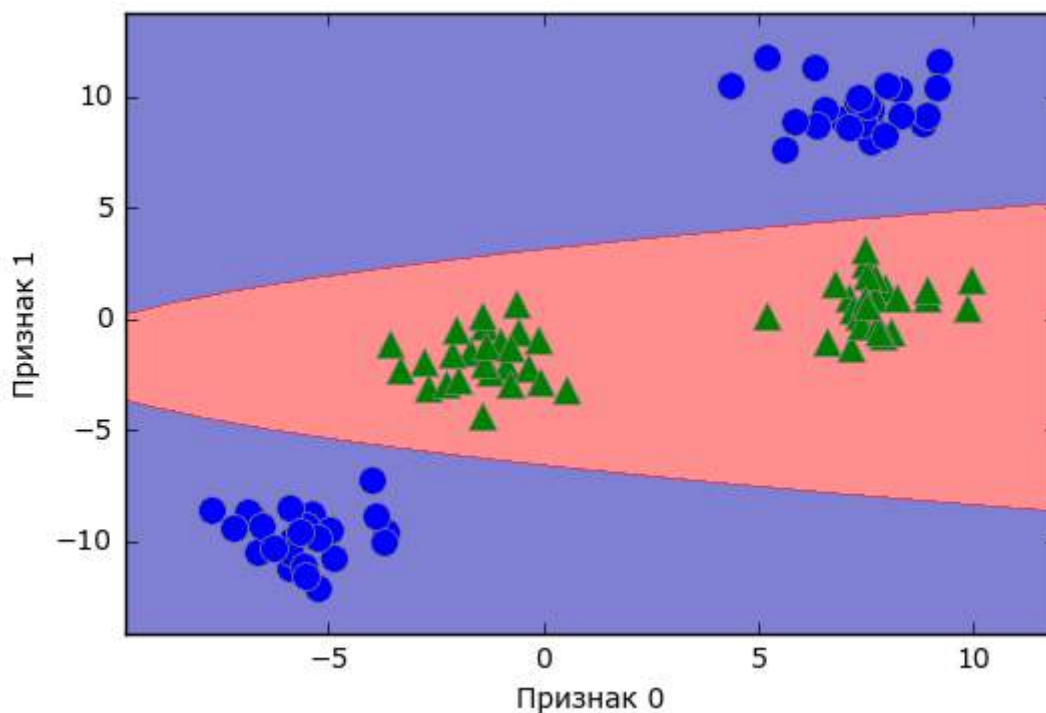


**Рис. 2.39** Граница принятия решений, найденная линейным SVM для расширенного трехмерного набора данных

Фактически модель линейного SVM как функция исходных признаков не является больше линейной. Это не линия, а скорее эллипс, как можно увидеть на графике, построенном ниже (рис. 2.40):

```
In[80]:
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```





**Рис. 2.40** Граница принятия решений для рис. 2.39 как функция от двух исходных признаков

### «Ядерный трюк» (kernel trick)

Из вышесказанного можно сделать вывод, что добавление нелинейных признаков может улучшить прогнозную силу линейной модели. Однако часто мы не знаем, какие признаки необходимо добавить, и добавление большего числа признаков (например, рассмотрение всех возможных взаимодействий в 100-мерном пространстве признаков) может очень сильно увеличить стоимость вычислений. К счастью, есть хитрый математический трюк, который позволяет нам обучить классификатор в многомерном пространстве, фактически не прибегая к вычислению нового, возможно, очень высокоразмерного пространства. Этот трюк известен под названием «ядерный трюк» (*kernel trick*) и он непосредственно вычисляет евклидовы расстояния (более точно, скалярные произведения точек данных), чтобы получить расширенное пространство признаков без фактического добавления новых признаков.

Существуют два способа поместить данные в высокоразмерное пространство, которые чаще всего используются методом опорных векторов: полиномиальное ядро, которое вычисляет все возможные полиномиальные комбинации исходных признаков до определенной степени, и ядро RBF (радиальная базисная функция), также известное как гауссовское ядро. Гауссовское ядро немного сложнее объяснить, поскольку оно соответствует бесконечному пространству признаков. Объяснить гауссовское ядро можно так: оно рассматривает все

возможные полиномы всех степеней, однако важность признаков снижается с возрастанием степени.<sup>19</sup>

И хотя на практике математические детали ядерного SVM не столь важны и можно легко понять, каким образом SVM с помощью ядра RBF делает прогнозы, мы рассмотрим их в следующем разделе.

## Понимание принципов работы SVM

В ходе обучения SVM вычисляет важность каждой точки обучающих данных с точки зрения определения решающей границы между двумя классами. Обычно лишь часть точек обучающего набора важна для определения границы принятия решений: точки, которые лежат на границе между классами. Они называются *опорными векторами* (*support vectors*) и дали свое название машине опорных векторов.

Чтобы получить прогноз для новой точки, измеряется расстояние до каждого опорного вектора. Классификационное решение принимается, исходя из расстояний до опорных векторов, а также важности опорных векторов, полученных в процессе обучения (хранятся в атрибуте `dual_coef_` класса `SVC`).

Расстояние между точками данных измеряется с помощью гауссовского ядра:

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$$

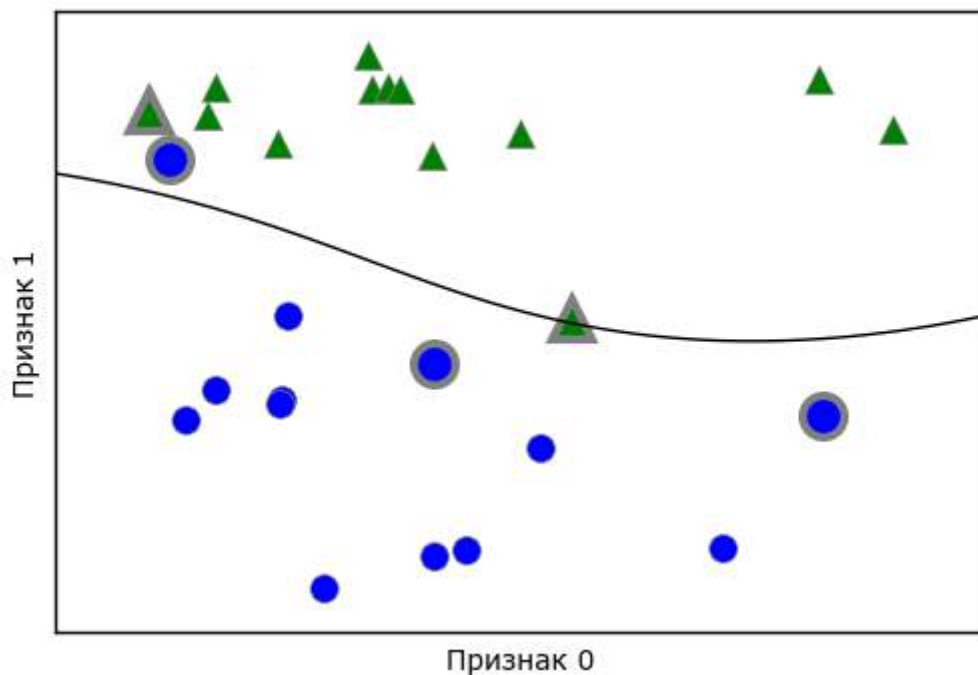
Здесь  $x_1$  и  $x_2$  — точки данных,  $\|x_1 - x_2\|$  обозначает евклидово расстояние, а  $\gamma$  (гамма) — параметр, который регулирует ширину гауссовского ядра.

Рис. 2-41 показывает результат обучения машины опорных векторов на двумерном 2-классовом наборе данных. Граница принятия решений показана черным цветом, а опорные векторы — это точки большего размера, обведенные широким контуром. Программный код строит этот график, обучая SVM на наборе данных `forge`:

```
In[81]:
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# размещаем на графике опорные векторы
sv = svm.support_vectors_
# метки классов опорных векторов определяются знаком дуальных коэффициентов
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

---

<sup>19</sup> Это следует из ряда Тейлора для экспоненциальной функции.



**Рис. 2.40** Граница принятия решений и опорные векторы, найденные SVM с помощью ядра RBF

В данном случае SVM дает очень гладкую и нелинейную (непрямую) границу. Здесь мы скорректировали параметр  $C$  и параметр  $\gamma$ , которые сейчас подробно обсудим.

### Настройка параметров SVM

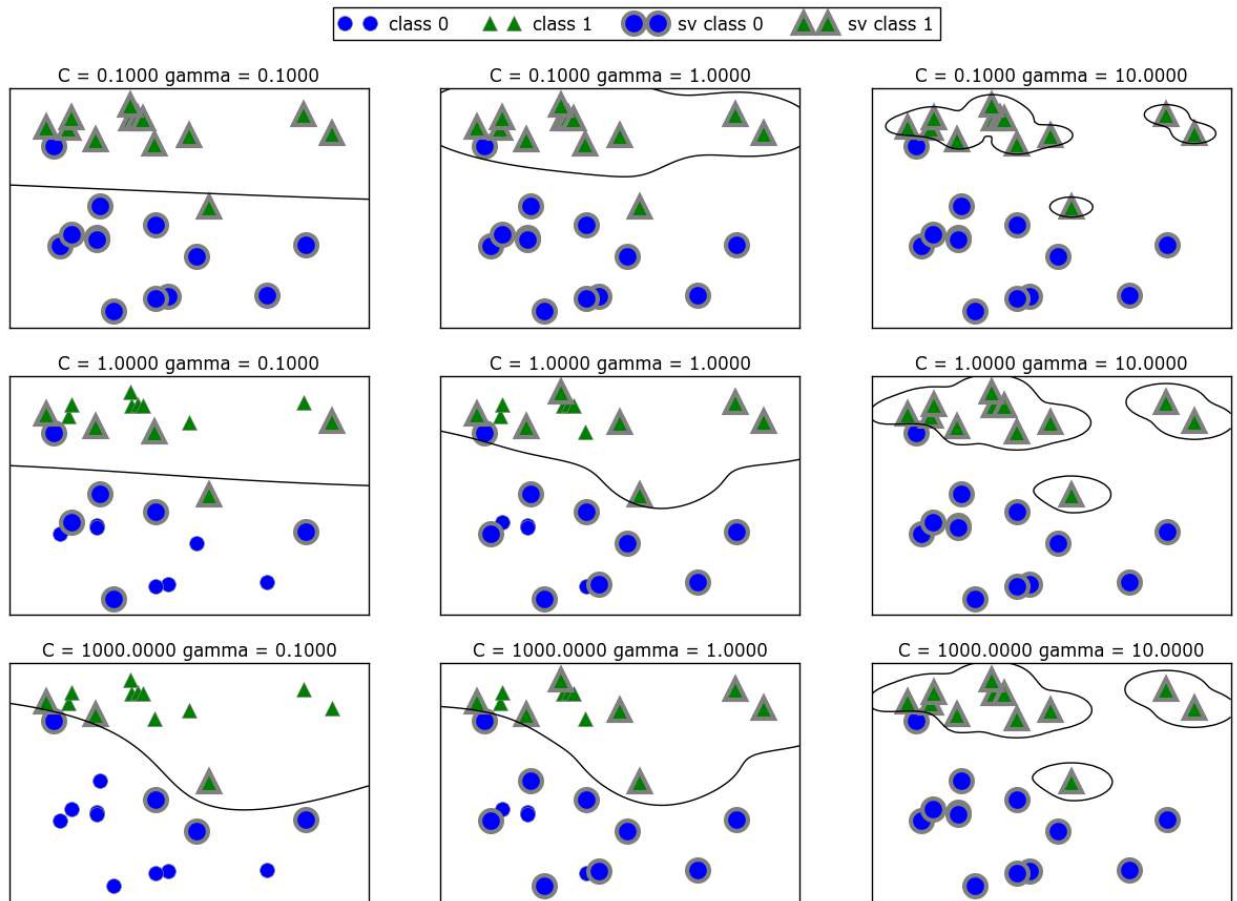
Параметр  $\gamma$  – это параметр формулы, приведенной в предыдущем разделе. Он регулирует ширину гауссовского ядра. Параметр  $\gamma$  задает степень близости расположения точек. Параметр  $C$  представляет собой параметр регуляризации, аналогичный тому, что использовался в линейных моделях. Он ограничивает важность каждой точки (точнее, ее `dual_coef_`).

Давайте посмотрим, что происходит при изменении этих параметров (рис. 2.42):

```
In[82]:
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mglearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=ax)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
                  ncol=4, loc=(.9, 1.2))
```



**Рис. 2.42** Границы принятия решений и опорные векторы для различных значений параметров  $C$  и  $\gamma$

Перемещаясь слева направо, мы увеличиваем значение параметра  $\gamma$  с 0.1 до 10. Небольшое значение  $\gamma$  соответствует большому радиусу гауссовского ядра, это означает, что многие точки рассматриваются как расположенные поблизости. Это приводит к получению очень гладких границ принятия решений, показанных в левой части графика, а границы, которые больше фокусируются на отдельных точках, расположились в правой части графика. Низкое значение  $\gamma$  означает медленное изменение решающей границы, которое дает модель низкой сложности, в то время как высокое значение  $\gamma$  дает более сложную модель.

Перемещаясь сверху вниз, мы увеличиваем параметр  $C$  с 0.1 до 1000. Как и в случае с линейными моделями, небольшое значение  $C$  соответствует модели с весьма жесткими ограничениями, в которой каждая точка данных может иметь лишь очень ограниченное влияние. В левом верхнем углу рис. можно увидеть, что граница принятия решений выглядит как почти линейная, неправильно классифицированные точки почти не влияют на линию. Увеличение значения  $C$ , как показано в левом нижнем углу, позволяет этим точкам оказывать более сильное влияние

на модель и делает решающую границу изогнутой, позволяя правильно классифицировать данные точки.

Давайте применим SVM с RBF-ядром к набору данных Breast Cancer. По умолчанию используются  $C=1$  и  $\gamma=1/n_{\text{features}}$ :

```
In[83]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

svc = SVC()
svc.fit(X_train, y_train)

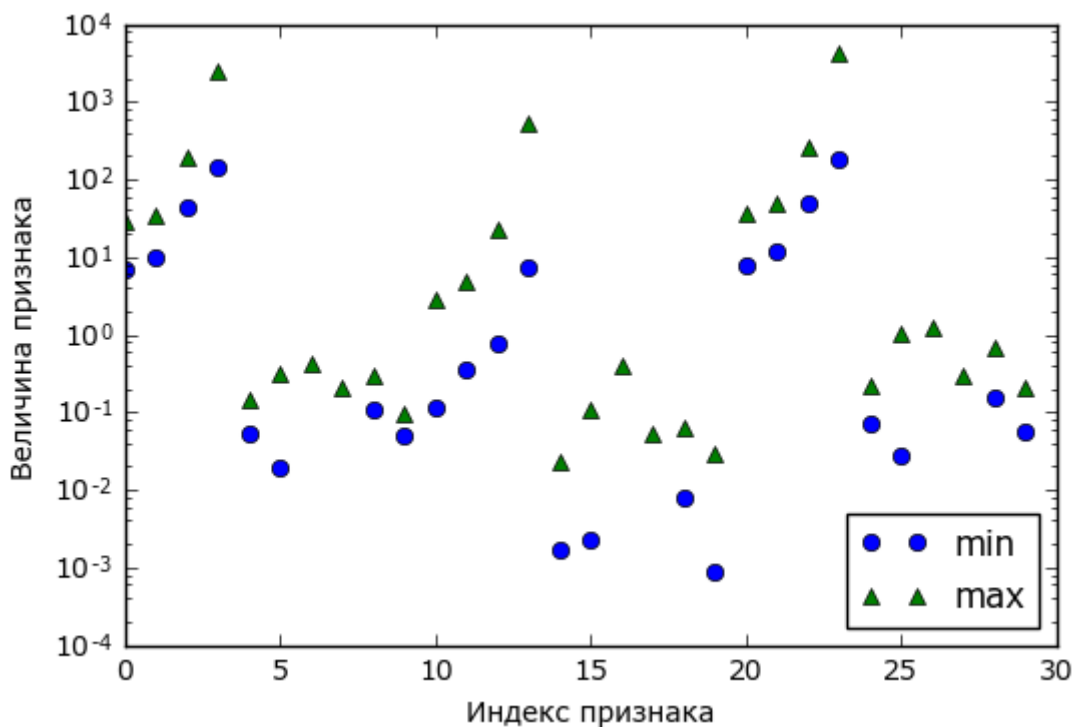
print("Правильность на обучающем наборе: {:.2f}".format(svc.score(X_train, y_train)))
print("Правильность на тестовом наборе: {:.2f}".format(svc.score(X_test, y_test)))
```

```
Out[83]:
Правильность на обучающем наборе: 1.00
Правильность на тестовом наборе: 0.63
```

Переобучение модели весьма существенно: идеальная правильность на обучающем наборе и лишь 63%-ная правильность на тестовом наборе. Хотя SVM часто дает хорошее качество модели, он очень чувствителен к настройкам параметров и масштабированию данных. В частности, SVM требует, чтобы все признаки были измерены в одном и том же масштабе. Давайте посмотрим на минимальное и максимальное значения каждого признака в log-пространстве (рис. 2.43):

```
In[84]:
plt.plot(X_train.min(axis=0), 'o', label="min")
plt.plot(X_train.max(axis=0), '^', label="max")
plt.legend(loc=4)
plt.xlabel("Индекс признака")
plt.ylabel("Величина признака")
plt.yscale("log")
```

Исходя из этого графика, мы можем заключить, что признаки в наборе данных Breast Cancer имеют совершенно различные порядки величин. Для ряда моделей (например, для линейных моделей) данный факт может быть в некоторой степени проблемой, однако для ядерного SVM он будет иметь разрушительные последствия. Давайте рассмотрим некоторые способы решения этой проблемы.



**Рис. 2.43** Диапазоны значений признаков для набора данных Breast Cancer (обратите внимание, что ось y имеет логарифмическую шкалу)

### Предварительная обработка данных для SVM

Один из способов решения этой проблемы – масштабирование всех признаков таким образом, чтобы все они имели примерно один и тот же масштаб. Общераспространенный метод масштабирования для ядерного SVM заключается в масштабировании данных так, чтобы все признаки принимали значения от 0 до 1. Мы увидим, как это делается с помощью метода предварительной обработки `MinMaxScaler` в главе 3, в которой дадим более подробную информацию. А сейчас давайте сделаем это «вручную»:

**In[85]:**

```
# вычисляем минимальное значение для каждого признака обучающего набора
min_on_training = X_train.min(axis=0)
# вычисляем ширину диапазона для каждого признака (max - min) обучающего набора
range_on_training = (X_train - min_on_training).max(axis=0)

# вычитаем минимальное значение и затем делим на ширину диапазона
# min=0 и max=1 для каждого признака
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Минимальное значение для каждого признака\n{}".format(X_train_scaled.min(axis=0)))
print("Максимальное значение для каждого признака\n{}".format(X_train_scaled.max(axis=0)))
```

**Out[85]:**

```
Минимальное значение для каждого признака
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Максимальное значение для каждого признака
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

```

In[86]:
# используем ТО ЖЕ САМОЕ преобразование для тестового набора,
# используя минимум и ширину диапазона из обучающего набора (см. главу 3)
X_test_scaled = (X_test - min_on_training) / range_on_training

In[87]:
svc = SVC()
svc.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(svc.score(X_test_scaled, y_test)))

Out[87]:
Правильность на обучающем наборе: 0.948
Правильность на тестовом наборе: 0.951

```

Масштабирование данных привело к огромной разнице! На самом деле наша модель имеет признаки недообучения, когда качество модели на обучающем и тестовом наборе весьма схоже, но все еще далеко от 100%-ной правильности. Исходя из этого, мы можем попытаться увеличить  $C$  или  $\gamma$ , чтобы подогнать более сложную модель. Например:

```

In[88]:
svc = SVC(C=1000)
svc.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    svc.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(svc.score(X_test_scaled, y_test)))

Out[88]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.972

```

В данном случае увеличение  $C$  позволяет значительно улучшить модель, в результате правильность на тестовом наборе составляет 97.2%.

### Преимущества, недостатки и параметры

Ядерный метод опорных векторов – это модели, обладающие мощной прогнозной силой и хорошо работающие на различных наборах данных. SVM позволяет строить сложные решающие границы, даже если данные содержат лишь несколько признаков. Они хорошо работают на низкоразмерных и высокоразмерных данных (то есть когда у нас мало или, наоборот, много признаков), однако плохо масштабируются с ростом объема данных. Запуск SVM на наборе данных объемом 10000 наблюдений не составляет проблем, однако работа с наборами данных объемом 100000 наблюдений и больше может стать сложной задачей с точки зрения времени вычислений и использования памяти.

Другим недостатком является то, что SVM требует тщательной предварительной обработки данных и настройки параметров. Именно поэтому сейчас многие специалисты в различных сферах вместо SVM



используют модели на основе дерева, например, случайные леса или градиентный бустинг (который практически не требует предварительную обработки данных). Кроме того, модели SVM трудноисследуемы, тяжело понять, почему был сделан именно такой прогноз и довольно сложно объяснить модель неспециалисту.

Однако все же стоит попробовать SVM, особенно в тех случаях, когда все ваши признаки имеют одинаковые единицы измерения (например, все признаки являются интенсивностями пикселей) и измерены в одном и том же масштабе.

Важными параметрами ядерного SVM являются параметр регуляризации *C*, тип ядра, а также параметры, определяемые ядром. Хотя мы в основном сосредоточились на ядре RBF, в `scikit-learn` доступны и другие типы ядер. Ядро RBF имеет лишь один параметр *gamma*, который является обратной величиной ширины гауссовского ядра. *gamma* и *C* регулируют сложность модели, более высокие значения этих параметров дают более сложную модель. Таким образом, оптимальные настройки обоих параметров, как правило, сильно взаимосвязаны между собой и поэтому *C* и *gamma* должны быть отрегулированы вместе.

## Нейронные сети (глубокое обучение)

Семейство алгоритмов, известное как нейронные сети, недавно пережило свое возрождение под названием «глубокое обучение». Несмотря на то что глубокое обучение сулит большие перспективы в различных сферах применения машинного обучения, алгоритмы глубокого обучения, как правило, жестко привязаны к конкретным случаям использования. В данном разделе мы рассмотрим лишь некоторые относительно простые методы, а именно многослойные персептроны для классификации и регрессии, которые могут служить отправной точкой в изучении более сложных методов машинного обучения. Многослойные персептроны (MLP) также называют *простыми (vanilla)* нейронными сетями прямого распространения, а иногда и просто нейронными сетями.

### Модель нейронной сети

MLP можно рассматривать как обобщение линейных моделей, которое прежде чем прийти к решению выполняет несколько этапов обработки данных.

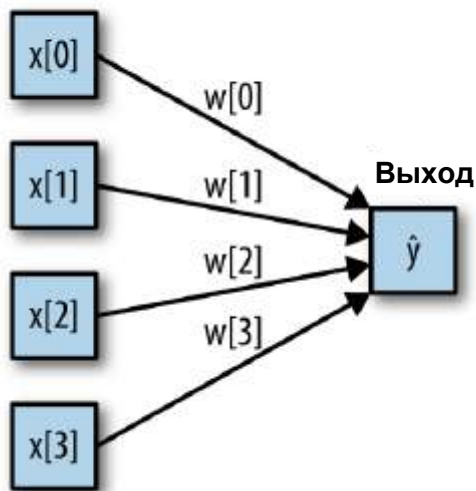
Вспомним, что в линейной регрессии прогноз получают с помощью следующей формулы:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Говоря простым языком,  $\hat{y}$  – это взвешенная сумма входных признаков  $x[0] \dots x[p]$ . Входные признаки взвешены по вычисленным в ходе обучения коэффициентам  $w[0] \dots w[p]$ . Мы можем представить их графически, как показано на рис. 2-44:

```
In[89]:  
display(mglearn.plots.plot_logistic_regression_graph())
```

**Входы**

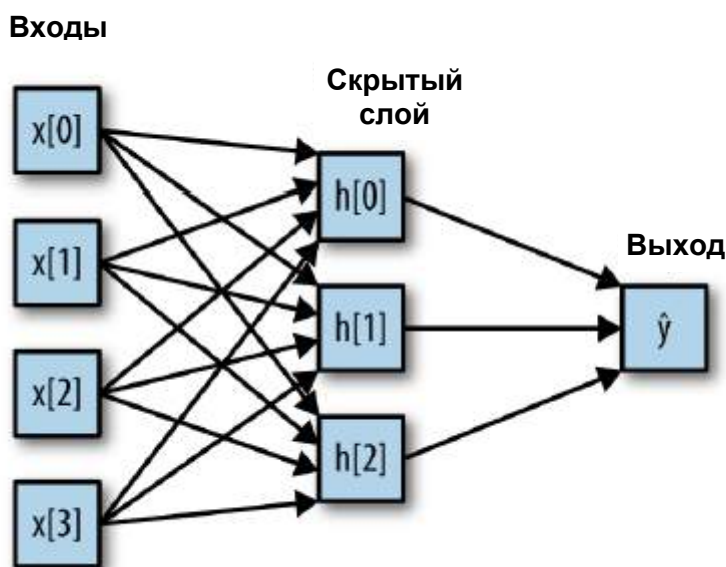


**Рис. 2.44** Визуализация логистической регрессии, в которой входные признаки и прогнозы показаны в виде узлов, а коэффициенты – в виде соединений между узлами

Здесь каждый узел, показанный слева, представляет собой входной признак, соединительные линии – коэффициенты, а узел справа – это выход, который является взвешенной суммой входов.

В MLP процесс вычисления взвешенных сумм повторяется несколько раз. Сначала вычисляются *скрытые элементы* (*hidden units*), которые представляют собой промежуточный этап обработки. Они вновь объединяются с помощью взвешенных сумм для получения конечного результата (рис. 2.45):

```
In[90]:  
display(mglearn.plots.plot_single_hidden_layer_graph())
```



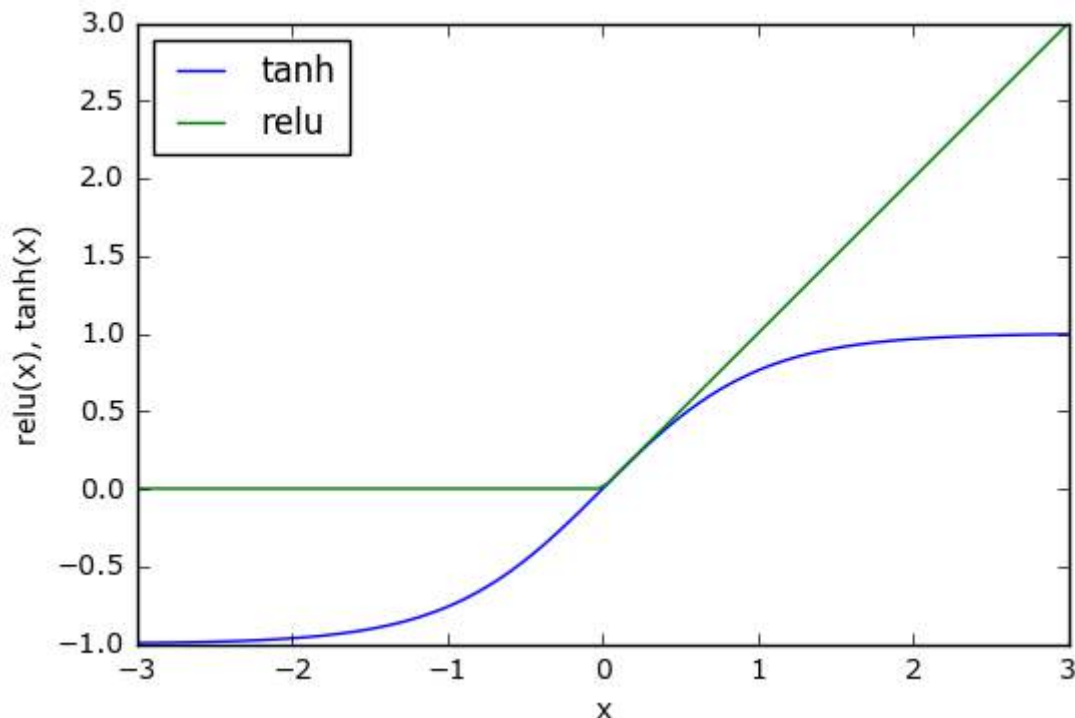
**Рис. 2.45** Иллюстрация многослойного персептрона с одним скрытым слоем

У этой модели гораздо больше вычисляемых коэффициентов (также называемых весами): коэффициент между каждым входом и каждым скрытым элементом (которые образуют *скрытый слой* или *hidden layer*) и коэффициент между каждым элементом скрытого слоя и выходом. С математической точки зрения вычисление серии взвешенных сумм – это то же самое, что вычисление лишь одной взвешенной суммы, таким образом, чтобы эта модель обладала более мощной прогнозной силой, чем линейная модель, нам нужен один дополнительный трюк. Поясним его на примере нейронной сети с одним скрытым слоем. Входной слой просто передает входы скрытому слою сети, либо без преобразования, либо выполнив сначала стандартизацию входов. Затем происходит вычисление взвешенной суммы входов для каждого элемента скрытого слоя, к ней применяется функция активации – обычно используются нелинейные функции *выпрямленный линейный элемент* (*rectified linear unit* или *relu*) или *гиперболический тангенс* (*hyperbolic tangent* или *tanh*). В итоге получаем выходы нейронов скрытого слоя. Эти промежуточные выходы могут считаться нелинейными преобразованиями и комбинациями первоначальных входов. Они становятся входами выходного слоя. Снова вычисляем взвешенную сумму входов, применяем функцию активации и получаем итоговые значения целевой переменной. Функции активации *relu* и *tanh* показаны на рис. 2.46. *Relu* отсекает значения ниже нуля, в то время как *tanh* принимает значения от  $-1$  до  $1$  (соответственно для минимального и максимального значений входов). Любая из этих двух нелинейных функций позволяет нейронной сети в

отличие от линейной модели вычислять гораздо более сложные зависимости.

In[91]:

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```



**Рис. 2.46** Функция активации гиперболический тангенс и функция активации выпрямленного линейного элемента

Для небольшой нейронной сети, изображенной на рис. 2.45, полная формула вычисления  $\hat{y}$  в случае регрессии будет выглядеть так (при использовании  $\tanh$ ):

$$h[0] = \tanh(w[0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3])$$

$$h[1] = \tanh(w[0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3])$$

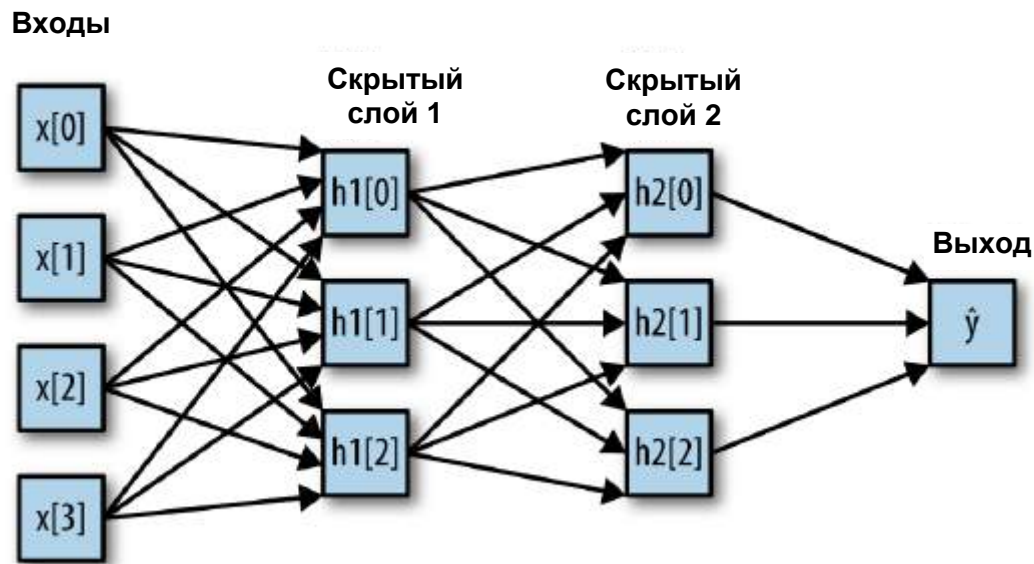
$$h[2] = \tanh(w[0,0] * x[0] + w[1,0] * x[1] + w[2,0] * x[2] + w[3,0] * x[3])$$

$$\hat{y} = v[0] * h[0] + v[1] * h[1] + v[2] * h[2]$$

Здесь  $w$  – веса между входом  $x$  и скрытым слоем  $h$ , а  $v$  – весовые коэффициенты между скрытым слоем  $h$  и выходом  $\hat{y}$ . Веса  $v$  и  $w$  вычисляются по данным,  $x$  являются входными признаками,  $\hat{y}$  – вычисленный выход, а  $h$  – промежуточные вычисления. Важный параметр, который должен задать пользователь – количество узлов в скрытом слое. Его значение может быть маленьким, например, 10 для очень маленьких или простых наборов данных или же большим,

например, 10000 для очень сложных данных. Кроме того, можно добавить дополнительные скрытые слои, как показано на рис. 2.47:

```
In[92]:
mglearn.plots.plot_two_hidden_layer_graph()
```



**Рис. 2.47** Многослойный персептрон с двумя скрытыми слоями

Построение больших нейронных сетей, состоящих из множества слоев вычислений, вдохновило специалистов ввести в обиход термин «глубокое обучение» («deep learning»).

### Настройка нейронных сетей

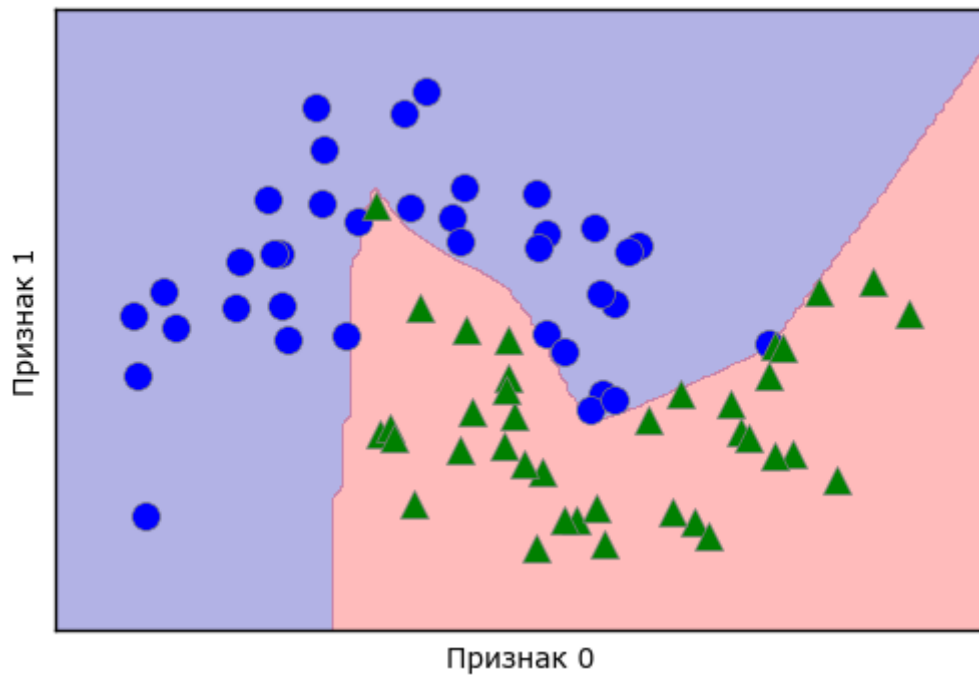
Давайте посмотрим, как работает MLP, применив `MLPClassifier` к набору данных `two_moons`, который мы использовали ранее в этой главе. Результаты показаны на рис. 2.48:

```
In[93]:
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

mlp = MLPClassifier(solver='lbfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

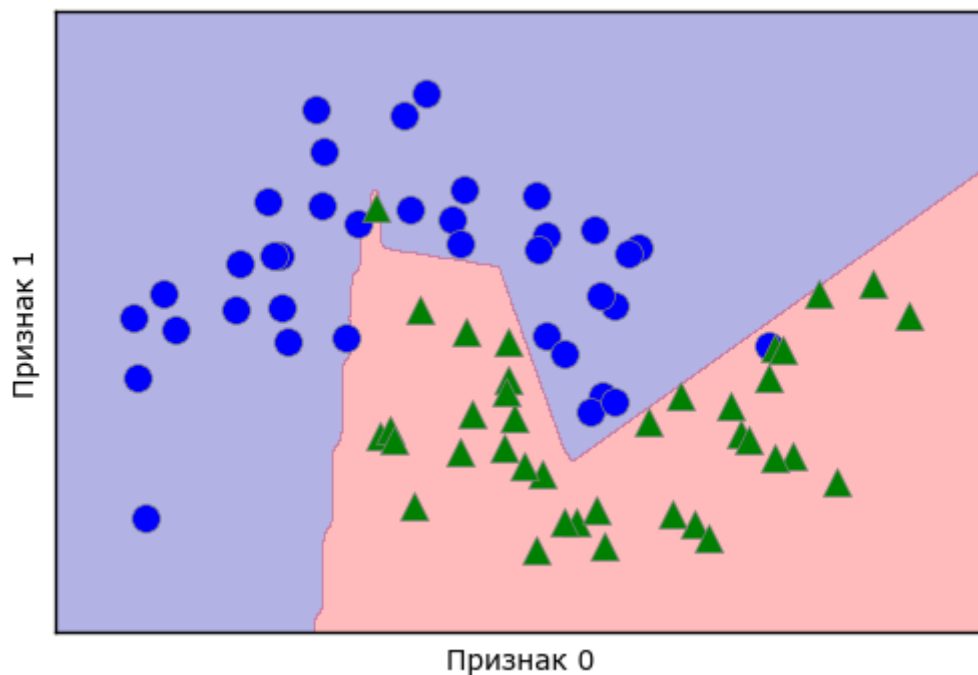


**Рис. 2.48** Граница принятия решений, построенная нейронной сетью со 100 скрытыми элементами на наборе данных two\_moons

Как видно из рис., нейронная сеть построила нелинейную, но относительно гладкую границу принятия решений. Мы использовали `solver='lbfgs'`, который рассмотрим позднее.

По умолчанию MLP использует 100 скрытых узлов, что довольно много для этого небольшого набора данных. Мы можем уменьшить число (что снизит сложность модели) и снова получить хороший результат (рис. 2.49):

```
In[94]:
mlp = MLPClassifier(solver='lbfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```



**Рис. 2.49** Граница принятия решений, построенная нейронной сетью с 10 скрытыми элементами на наборе данных `two_moons`

При использовании лишь 10 скрытых элементов граница принятия решений становится более неровной. По умолчанию используется функция активации `relu`, показанная на рис. 2.46. При использовании одного скрытого слоя решающая функция будет состоять из 10 прямолинейных отрезков. Если необходимо получить более гладкую решающую границу, можно добавить большее количество скрытых элементов (как показано на рис. 2.49), добавить второй скрытый слой (рис. 2.50), или использовать функцию активации `tanh` (рис. 2.51):

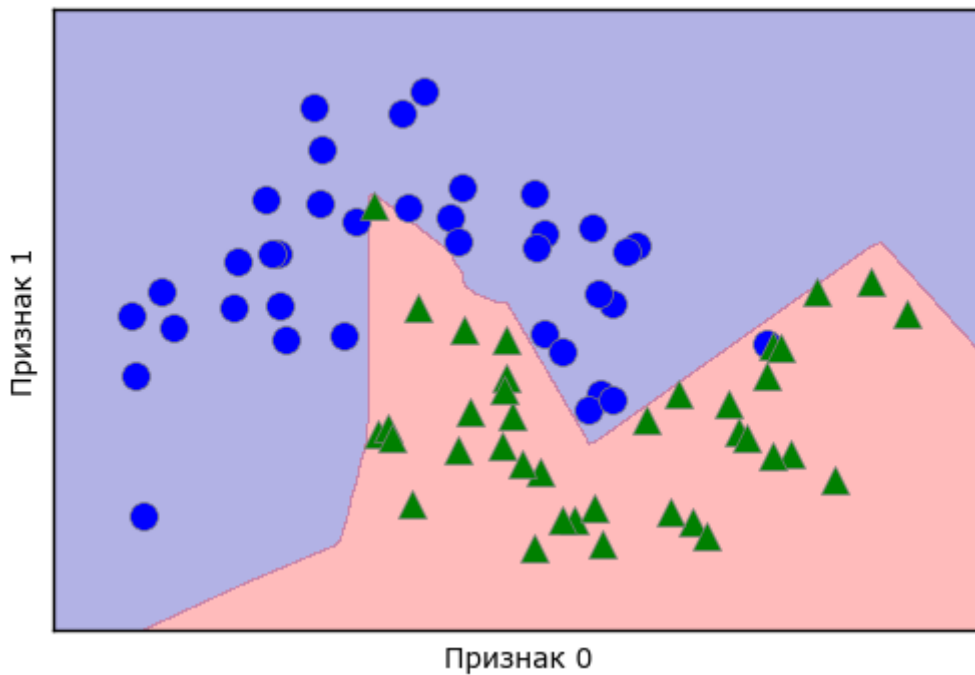
**In[95]:**

```
# использование двух скрытых слоев по 10 элементов в каждом
mlp = MLPClassifier(solver='lbfgs', random_state=0,
                    hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

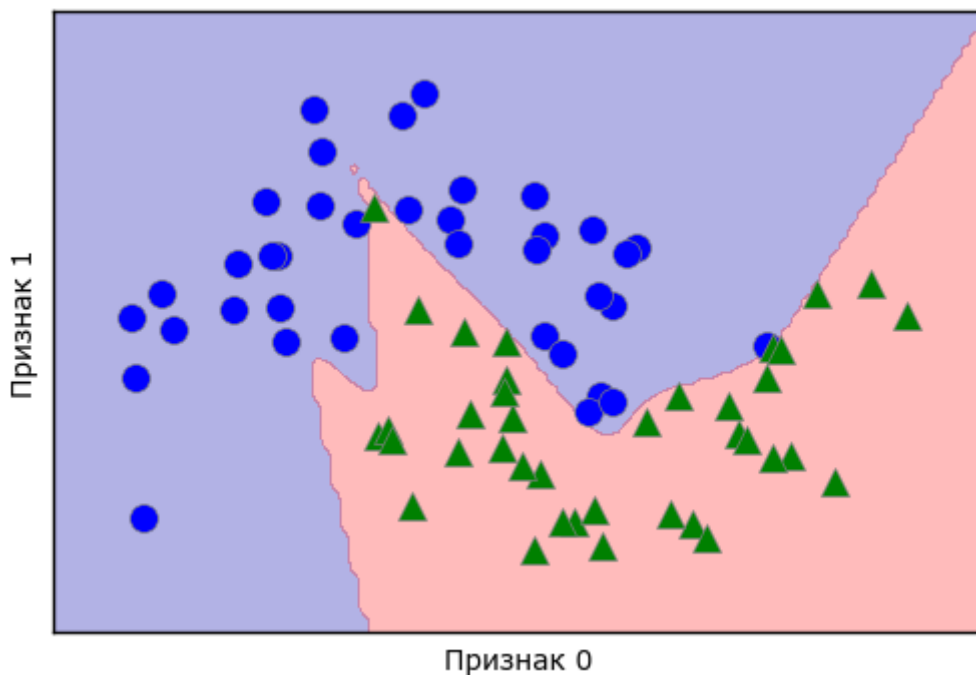
**In[96]:**

```
# использование двух скрытых слоев по 10 элементов в каждом, на этот раз с функцией tanh
mlp = MLPClassifier(solver='lbfgs', activation='tanh',
                    random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```





**Рис. 2.50** Граница принятия решений, построенная нейронной сетью с 2 скрытыми слоями по 10 элементов в каждом и функцией активации выпрямленный линейный элемент

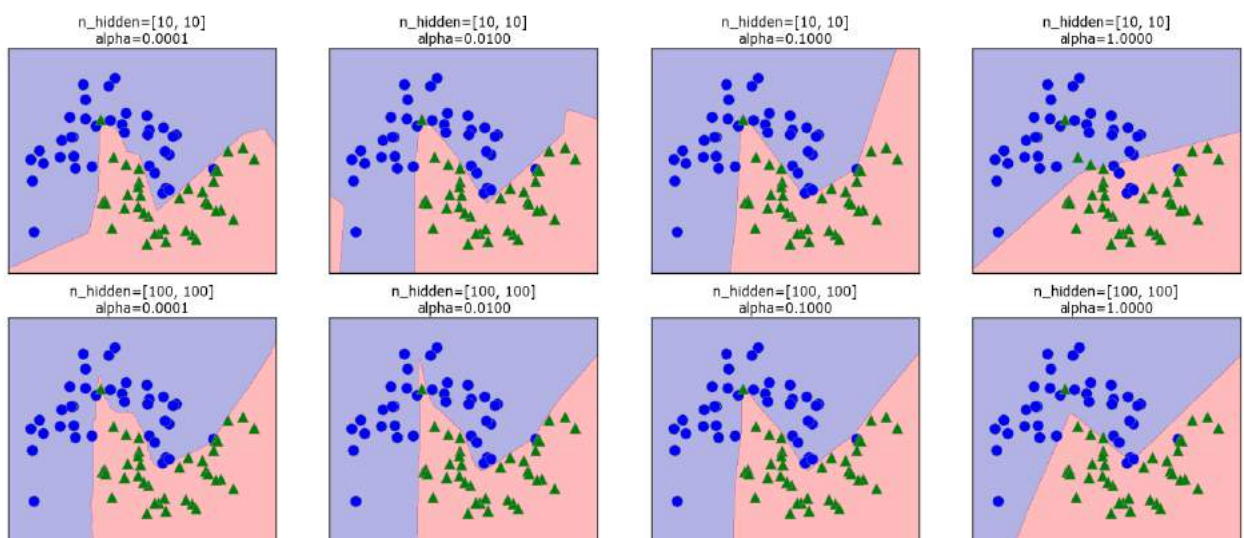


**Рис. 2.51** Граница принятия решений, построенная нейронной сетью с 2 скрытыми слоями по 10 элементов в каждом и функцией активации гиперболический тангенс

И, наконец, мы можем дополнительно настроить сложность нейронной сети с помощью  $l_2$  штрафа, чтобы сжать весовые коэффициенты до близких к нулю значений, как мы это делали в гребневой регрессии и линейных классификаторов. В `MLPClassifier` за это отвечает параметр `alpha` (как и в моделях линейной регрессии), и по

умолчанию установлено очень низкое значение (небольшая регуляризация). На рис. 2.52 показаны результаты применения к набору данных `two_moons` различных значений `alpha` с использованием двух скрытых слоев с 10 или 100 элементами в каждом:

```
In[97]:
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(solver='lbfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
                            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
            n_hidden_nodes, n_hidden_nodes, alpha))
```



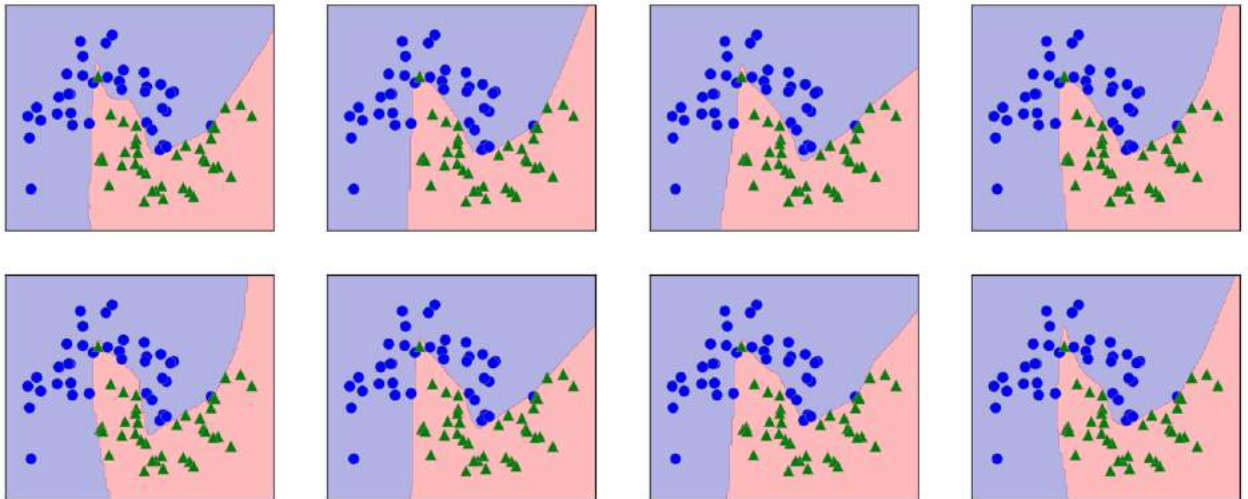
**Рис. 2.52** Границы принятия решений для различного количества скрытых элементов и разных значений параметра `alpha`

Как вы, наверное, уже поняли, существуют различные способы регулировать сложность нейронной сети: количество скрытых слоев, количество элементов в каждом скрытом слое и регуляризация (`alpha`). На самом деле их гораздо больше, но мы не будем здесь вдаваться в подробности.

Важным свойством нейронных сетей является то, что их веса задаются случайным образом перед началом обучения и случайная инициализация влияет на процесс обучения модели. Это означает, что даже при использовании одних и тех же параметров мы можем получить очень разные модели, задавая разные стартовые значения генератора псевдослучайных чисел. При условии, что сеть имеет большой размер и ее сложность настроена правильно, данный факт не должен сильно влиять на правильность, однако о нем стоит помнить (особенно при работе с небольшими сетями). На рис. 2-53 представлены графики

нескольких моделей, обученных с использованием тех же самых значений параметров:

```
In[98]:
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(solver='lbfgs', random_state=i,
                       hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```



**Рис. 2.53** Границы принятия решений, полученные с использованием тех же самых параметров, но разных стартовых значений

Чтобы лучше понять, как нейронная сеть работает на реальных данных, давайте применим `MLPClassifier` к набору данных Breast Cancer. Мы начнем с параметров по умолчанию:

```
In[99]:
print("Максимальные значения характеристик:\n{}".format(cancer.data.max(axis=0)))
```

```
Out[99]:
Максимальные значения характеристик:
[ 28.110  39.280 188.500 2501.000  0.163  0.345  0.427
  0.201  0.304  0.097  2.873  4.885 21.980 542.200
  0.031  0.135  0.396  0.053  0.079  0.030  36.040
 49.540 251.200 4254.000  0.223  1.058  1.252  0.291
 0.664  0.207]
```

```
In[100]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Правильность на обучающем наборе: {:.2f}".format(mlp.score(X_train, y_train)))
print("Правильности на тестовом наборе: {:.2f}".format(mlp.score(X_test, y_test)))
```

```
Out[100]:
Правильность на обучающем наборе: 0.92
Правильность на тестовом наборе: 0.90
```

MLP демонстрирует довольно неплохую правильность, однако не столь хорошую, если сравнивать с другими моделями. Как и в предыдущем примере с SVC, это, вероятно, обусловлено масштабом данных. Нейронные сети также требуют того, чтобы все входные признаки были измерены в одном и том же масштабе, в идеале они должны иметь среднее 0 и дисперсию 1. Мы должны отмасштабировать наши данные так, чтобы они отвечали этим требованиям. Опять же, мы будем делать это вручную, однако в главе 3 расскажем, как это делать автоматически с помощью `StandardScaler`.

```
In[101]:
# вычисляем среднее для каждого признака обучающего набора
mean_on_train = X_train.mean(axis=0)
# вычисляем стандартное отклонение для каждого признака обучающего набора
std_on_train = X_train.std(axis=0)

# вычитаем среднее и затем умножаем на обратную величину стандартного отклонения
# mean=0 и std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# используем ТО ЖЕ САМОЕ преобразование (используем среднее и стандартное отклонение
# обучающего набора) для тестового набора
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(mlp.score(X_test_scaled, y_test)))

Out[101]:
Правильность на обучающем наборе: 0.991
Правильность на тестовом наборе: 0.965

ConvergenceWarning:
  Stochastic Optimizer: Maximum iterations reached and the optimization
  hasn't converged yet.
```

После масштабирования результаты стали намного лучше и теперь уже вполне могут конкурировать с результатами остальных моделей. Впрочем, мы получили предупреждение о том, что достигнуто максимальное число итераций. Оно является неотъемлемой частью алгоритма `adam` и сообщает нам о том, что мы должны увеличить число итераций:

```
In[102]:
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(mlp.score(X_test_scaled, y_test)))

Out[102]:
Правильность на обучающем наборе: 0.995
Правильность на тестовом наборе: 0.965
```

Увеличение количества итераций повысило правильность лишь на обучающем наборе. Тем не менее модель имеет достаточно высокое значение правильности. Поскольку существует определенный разрыв между правильностью на обучающем наборе и правильностью на тестовом наборе, мы можем попытаться уменьшить сложность модели, чтобы улучшить обобщающую способность. В данном случае мы увеличим параметр `alpha` (довольно сильно с `0.0001` до `1`), чтобы применить к весам более строгую регуляризацию:

```
In[103]:
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Правильность на обучающем наборе: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Правильность на тестовом наборе: {:.3f}".format(mlp.score(X_test_scaled, y_test)))

Out[103]:
Правильность на обучающем наборе: 0.988
Правильность на тестовом наборе: 0.972
```

Это дает правильность, сопоставимую с правильностью лучших моделей.<sup>20</sup>

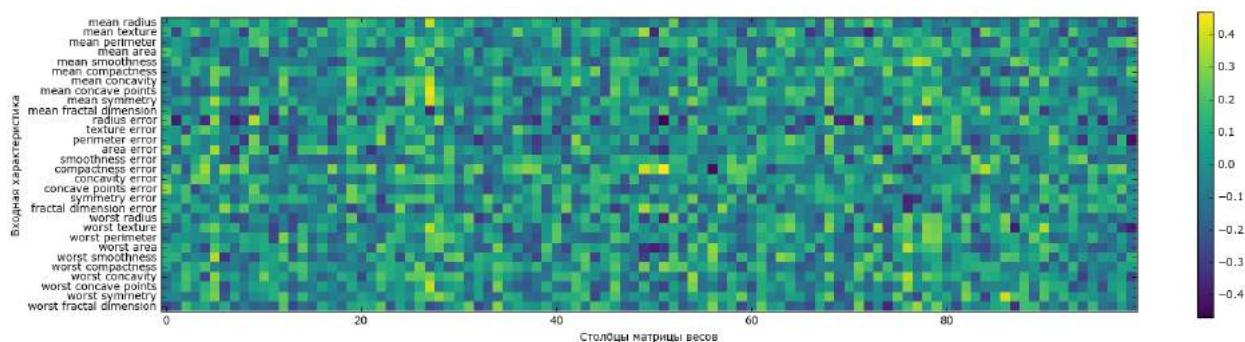
Несмотря на то что анализ нейронной сети возможен, он, как правило, гораздо сложнее анализа линейной модели или модели на основе дерева. Один из способов анализа нейронной сети заключается в том, чтобы исследовать веса модели. Образец такого анализа вы можете увидеть в [галерее примеров scikit-learn](#). Применительно к набору данных Breast Cancer такой анализ может быть немного сложен. Следующий график (рис. 2.54) показывает весовые коэффициенты, которые были вычислены при подключении входного слоя к первому скрытому слою. Строки в этом графике соответствуют 30 входным признакам, а столбцы – 100 скрытым элементам. Светлые цвета соответствуют высоким положительным значениям, в то время как темные цвета соответствуют отрицательным значениям:

```
In[104]:
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Столбцы матрицы весов")
plt.ylabel("Входная характеристика")
plt.colorbar()
```

---

<sup>20</sup> Здесь вы могли заметить, что большинство моделей, давших наилучший прогноз, достигают одинаковой правильности на тестовом наборе 0.972. Это означает, что все эти модели дают одинаковое количество ошибочных прогнозов, равное четырем. Если сравнить полученные прогнозы с фактическими, вы увидите, что все эти модели неправильно прогнозируют одни и те же наблюдения. Это может быть обусловлено очень маленьким размером набора данных или же тем, что эти наблюдения сильно отличаются от остальных.





**Рис. 2.54** Теплокарта для весов первого слоя нейронной сети, обученной на наборе данных Breast Cancer

Один из возможных выводов, который мы можем сделать, заключается в том, что признаки с небольшими весами скрытых элементов «менее важны» в модели. Мы можем увидеть, что «mean smoothness» и «mean compactness» наряду с признаками, расположенными между «smoothness error» и «fractal dimension error», имеют относительно низкие веса по сравнению с другими признаками. Это может означать, что эти признаки являются менее важными или, возможно, мы не преобразовали их таким способом, чтобы их могла использовать нейронная сеть.

Кроме того, мы можем визуализировать веса, соединяющие скрытый слой с выходным слоем, но их еще труднее интерпретировать.

Несмотря на то что для наиболее распространенных архитектур нейронных сетей `MLPClassifier` и `MLPRegressor` предлагают легкий в использовании интерфейс, они представляют лишь небольшой набор возможных средств, позволяющих строить нейронные сети. Если вас интересует работа с более гибкими или более масштабными моделями, мы рекомендуем вам не ограничиваться возможностями библиотеки `scikit-learn` и обратиться к фантастическим по своим возможностям библиотекам глубокого обучения. Для пользователей Python наиболее устоявшимися являются `keras`, `lasagna` и `tensorflow`. `lasagna` построена на основе библиотеки `theano`, тогда как `keras` может использовать либо `tensorflow`, либо `theano`. Эти библиотеки предлагают гораздо более гибкий интерфейс для построения нейронных сетей и обновляются в соответствии с последними достижениями в области глубокого обучения. Кроме того, все популярные библиотеки глубокого обучения позволяют использовать высокопроизводительные графические процессоры (GPU), которые в `scikit-learn` не поддерживаются. Использование графических процессоров позволяет ускорить вычисления от 10 до 100 раз, и они

имеют важное значение для применения методов глубокого обучения для крупномасштабных наборов данных.

### Преимущества, недостатки и параметры

Нейронные сети вновь «вышли на сцену» во многих сферах применения машинного обучения в качестве передовых методов. Одно из их главных преимуществ заключается в том, что они способны обрабатывать информацию, содержащуюся в больших объемах данных, и строить невероятно сложные модели. При наличии достаточного времени вычислений, данных и тщательной настройки параметров нейронные сети часто превосходят другие алгоритмы машинного обучения (для задач классификации и регрессии).

Это дает и свои минусы. Нейронные сети, особенно крупные нейронные сети, как правило, требуют длительного времени обучения. Как мы видели здесь, они также требуют тщательной предварительной обработки данных. Аналогично SVM, нейронные сети лучше всего работают с «однородными» данными, где все признаки измерены в одном и том же масштабе. Что касается данных, в которых признаки имеют разный масштаб, модели на основе дерева могут дать лучший результат. Кроме того, настройка параметров нейронной сети – это само по себе искусство. В наших экспериментах мы едва коснулись возможных способов настройки и обучения нейросетевых моделей.

### Оценка сложности в нейронных сетях

Наиболее важными параметрами являются ряде слоев и число скрытых блоков в одном слое. Вы должны начать с одной или двумя скрытыми слоями, и, возможно, расширить оттуда. Количество узлов на скрытом уровне часто аналогично числу входных функций, но редко выше, чем в низких до средних тысяч.

Полезным показателем, позволяющим судить о сложности нейронной сети, является количество вычисляемых в ходе обучения весов или коэффициентов. Если вы работаете с 2-классовым набором данных, содержащим 100 признаков, и используете нейронную сеть, состоящую из 100 скрытых элементов, то между входным и первым скрытым слоем будет  $100 * 100 = 10000$  весов. Кроме того, между скрытым слоем и выходным слоем будет  $100 * 1 = 100$  весов, что в итоге даст около 10100 весов. Если использовать второй скрытый слой размером 100 скрытых элементов, то  $100 * 100 = 10000$  весов из первого скрытого слоя добавятся ко второму скрытому слою, что в итоге составит 20100 весов. Если вместо этого вы будете использовать один слой из 1000 скрытых элементов, вы вычислите  $100 * 1000 = 100000$  весов на пути от входного слоя к скрытому слою и  $1000 * 1$  весов на пути из скрытого слоя к выходному слою, всего



101000 весов. Если использовать второй скрытый слой, вы добавите еще  $1000 \times 1000 = 1000000$  весов, получив колоссальную цифру 1101000, что в 50 раз больше количества весов, вычисленных для модели с двумя скрытыми слоями по 100 элементов в каждом.

Общераспространенный способ настройки параметров в нейронной сети – сначала построить сеть достаточно большого размера, чтобы она обучилась. Затем, убедившись в том, что сеть может обучаться, сжимаете веса сети или увеличиваете `alpha`, чтобы добавить регуляризацию, которая улучшит обобщающую способность.

В наших экспериментах мы сосредоточились главным образом на спецификации модели: количестве слоев и узлов в слое, регуляризации и нелинейных функциях активации. Эти параметры задают модель, которую мы хотим обучить. Кроме того, встает вопрос о том, *как* обучить модель или алгоритм, который используется для вычисления весов и задается с помощью параметра `solver`. Существует два простых в использовании алгоритма. Алгоритм `'adam'`, выставленный по умолчанию, дает хорошее качество в большинстве ситуаций, но весьма чувствителен к масштабированию данных (поэтому важно отмасштабировать ваши данные так, чтобы каждая характеристика имела среднее 0 и дисперсию 1). Другой алгоритм `'lbfgs'` вполне надежен, но может занять много времени в случае больших моделей или больших массивов данных. Существует также более продвинутая опция `'sgd'`, которая используется многими специалистами по глубокому обучению. Опция `'sgd'` имеет большее количество дополнительных параметров, которые нужно настроить для получения наилучших результатов. Вы можете найти описания всех этих параметров в руководстве пользователя. Начиная работать с MLP, придерживайтесь алгоритмов `'adam'` и `'l-bfgs'`.



Каждый раз метод `fit` строит модель заново

Важное свойство моделей `scikit-learn` заключается в том, что вызов метода `fit` всегда будет сбрасывать все, чему модель обучилась ранее. Таким образом, если вы построите модель на одном наборе данных, а затем вызовете метод `fit` снова для другого набора данных, модель «забудет» все, чему обучилась на первом наборе данных. Вы можете без ограничений вызывать метод `fit` для модели и результат будет таким, как если бы вы вызвали его для «новой» модели.

## Оценки неопределенности для классификаторов

Еще одна полезная деталь интерфейса `scikit-learn`, о которой мы еще не говорили – это возможность вычислить оценки неопределенности

прогнозов. Часто вас интересует не только класс, спрогнозированный моделью для определенной точки тестового набора, но и степень уверенности модели в правильности прогноза. В реальной практике различные виды ошибок приводят к очень разным результатам. Представьте себе медицинский тест для определения рака. Ложно положительный прогноз может привести к проведению дополнительных исследований, тогда как ложно отрицательный прогноз может привести к пропуску серьезной болезни. Мы подробнее разберем эту тему в главе 6.

В `scikit-learn` существует две различные функции, с помощью которых можно оценить неопределенность прогнозов: `decision_function` и `predict_proba`. Большая часть классификаторов (но не все) позволяет использовать по крайней мере одну из этих функций. Давайте применим эти две функции к синтетическому двумерному набору данных, построив классификатор `GradientBoostingClassifier`, который позволяет использовать как метод `decision_function`, так и метод `predict_proba`:

```
In[105]:
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_blobs, make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# мы переименовываем классы в «blue» и «red» для удобства
y_named = np.array(["blue", "red"])[y]

# мы можем вызвать train_test_split с любым количеством массивов,
# все будут разбиты одинаковым образом
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)

# строим модель градиентного бустинга
gbrt = GradientBoostingClassifier(random_state=0)
gbrt.fit(X_train, y_train_named)
```

## Решающая функция

В бинарной классификации возвращаемое значение `decision_function` имеет форму `(n_samples)`:

```
In[106]:
print("Форма массива X_test: {}".format(X_test.shape))
print("Форма решающей функции: {}".format(
    gbrt.decision_function(X_test).shape))
```

```
Out[106]:
Форма массива X_test: (25, 2)
Форма решающей функции: (25,)
```

Возвращаемое значение представляет собой число с плавающей точкой для каждого примера:

```
In[107]:
# выведем несколько первых элементов решающей функции
print("Решающая функция:\n{}".format(gbrt.decision_function(X_test)[:6]))

Out[107]:
Решающая функция:
[ 4.136 -1.683 -3.951 -3.626 4.29 3.662]
```

Значение показывает, насколько сильно модель уверена в том, что точка данных принадлежит «положительному» классу, в данном случае, классу 1. Положительное значение указывает на предпочтение в пользу позиционного класса, а отрицательное значение – на предпочтение в пользу «отрицательного» (другого) класса.

Мы можем судить о прогнозах, лишь взглянув на знак решающей функции.

```
In[108]:
print("Решающая функция с порогом отсеечения:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("Прогнозы:\n{}".format(gbrt.predict(X_test)))

Out[108]:
Решающая функция с порогом отсеечения:
[ True False False False True True False True True True False True
  True False True False False False True True True True True False
  False]
Прогнозы:
['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'
 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'
 'red' 'blue' 'blue']
```

Для бинарной классификации «отрицательный» класс – это всегда первый элемент атрибута `classes_`, а «положительный» класс – второй элемент атрибута `classes_`. Таким образом, если вы хотите полностью просмотреть вывод метода `predict`, вам нужно воспользоваться атрибутом `classes_`:

```
In[109]:
# переделаем булевы значения True/False в 0 и 1
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
# используем 0 и 1 в качестве индексов атрибута classes_
pred = gbrt.classes_[greater_zero]
# pred идентичен выводу gbrt.predict
print("pred идентичен прогнозам: {}".format(
    np.all(pred == gbrt.predict(X_test))))

Out[109]:
pred идентичен прогнозам: True
```

Диапазон значений `decision_function` может быть произвольным и зависит от данных и параметров модели:

```
In[110]:
decision_function = gbrt.decision_function(X_test)
print("Решающая функция минимум: {:.2f} максимум: {:.2f}".format(
    np.min(decision_function), np.max(decision_function)))

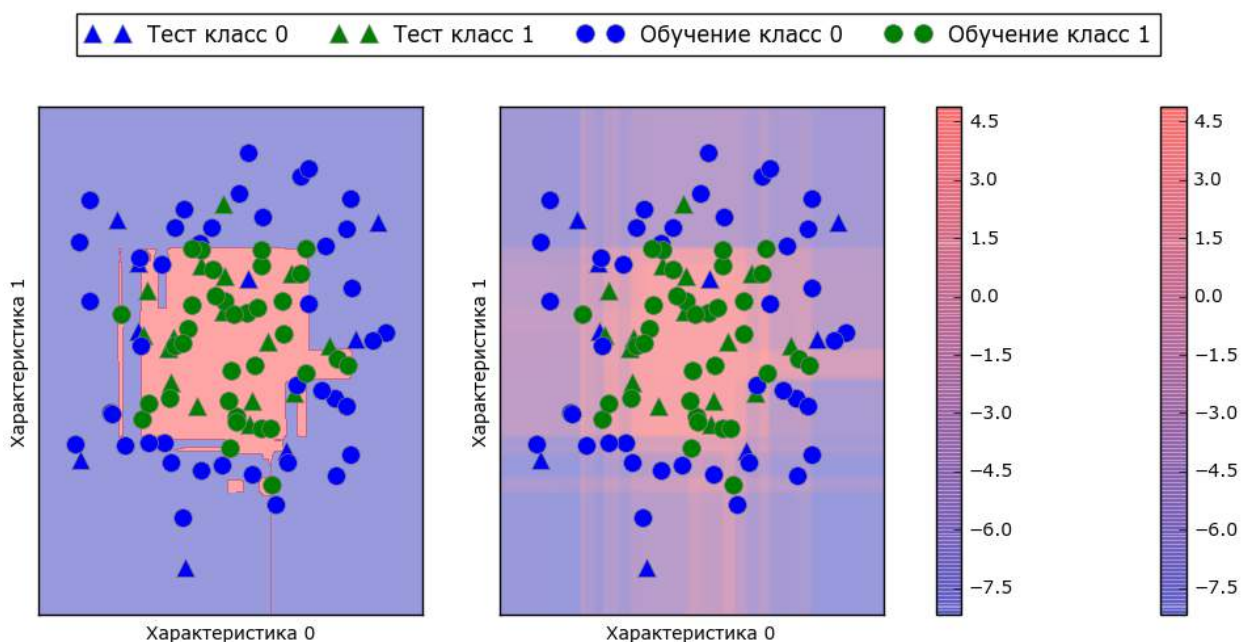
Out[110]:
Решающая функция минимум: -7.69 максимум: 4.29
```

Это произвольное масштабирование часто затрудняет интерпретацию вывода `decision_function`.

В следующем примере мы построим `decision_function` для всех точек в двумерной плоскости, используя цветовую кодировку и уже знакомую визуализацию решающей границы. Мы представим точки обучающего набора в виде кружков, а тестовые данные – в виде треугольников (рис. 2.55):

```
In[111]:
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                               fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                             alpha=.4, cm=mglearn.ReBl)

for ax in axes:
    # размещаем на графике точки обучающего и тестового наборов
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Характеристика 0")
    ax.set_ylabel("Характеристика 1")
    cbar = plt.colorbar(scores_image, ax=axes.tolist())
    axes[0].legend(["Тест класс 0", "Тест класс 1", "Обучение класс 0",
                   "Обучение класс 1"], ncol=4, loc=(.1, 1.1))
```



**Рис. 2.55** Граница принятия решений (слева) и решающая функция (справа) модели градиентного бустинга, построенной на двумерном синтетическом наборе данных

Цветовая кодировка не только спрогнозированного результата, но степени определенности прогноза дает дополнительную информацию.

Однако в этой визуализации трудно разглядеть границу между двумя классами.

## Прогнозирование вероятностей

Вывод метода `predict_proba` – это вероятность каждого класса и часто его легче понять, чем вывод метода `decision_function`. Для бинарной классификации он всегда имеет форму `(n_samples, 2)`:

```
In[112]:  
print("Форма вероятностей: {}".format(gbrt.predict_proba(X_test).shape))
```

```
Out[112]:  
Форма вероятностей: (25, 2)
```

Первый элемент строки – это оценка вероятности первого класса, а второй элемент строки – это оценка вероятности второго класса. Поскольку речь идет о вероятности, то значения в выводе `predict_proba` всегда находятся в диапазоне между 0 и 1, а сумма значений для обоих классов всегда равна 1:

```
In[113]:  
# выведем первые несколько элементов predict_proba  
print("Спрогнозированные вероятности:\n{}".format(  
    gbrt.predict_proba(X_test[:6])))
```

```
Out[113]:  
Спрогнозированные вероятности:  
[[ 0.016 0.984]  
 [ 0.843 0.157]  
 [ 0.981 0.019]  
 [ 0.974 0.026]  
 [ 0.014 0.986]  
 [ 0.025 0.975]]
```

Поскольку вероятности обоих классов в сумме дают 1, один из классов всегда будет иметь определенность, превышающую 50%. Этот класс и будет спрогнозирован.<sup>21</sup>

В предыдущем выводе видно, что большинство точек отнесены к тому или иному классу с высокой долей определенности. Соответствие спрогнозированной неопределенности фактической зависит от модели и параметров. Для переобученной модели характерна высокая доля определенности прогнозов, даже если они и ошибочные. Модель с меньшей сложностью обычно характеризуется высокой долей неопределенности своих прогнозов. Модель называется *калиброванной* (*calibrated*), если вычисленная неопределенность соответствует

---

<sup>21</sup> Поскольку вероятности – это числа с плавающей точкой, то маловероятно, что они обе будут точно равны 0.500. Однако, если это произойдет, то прогноз будет осуществлен случайным образом.

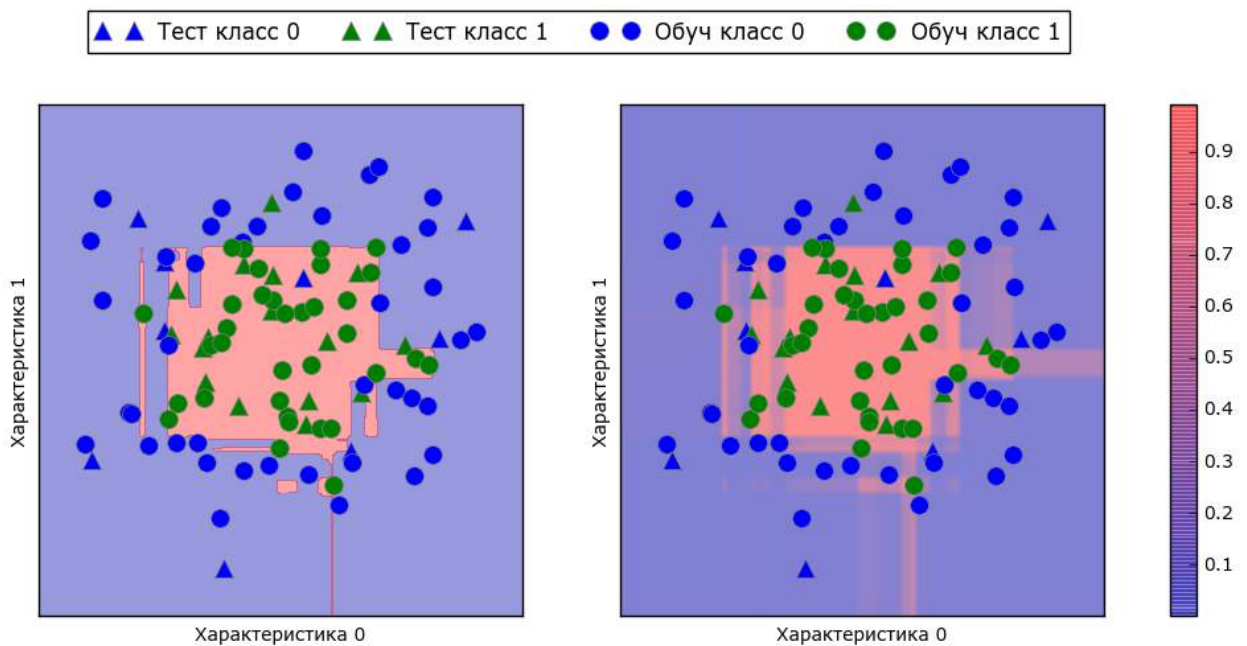
фактической: в калиброванной модели прогноз, полученный с 70%-ной определенностью, будет правильным в 70% случаев.

В следующем примере (рис. 2.56) мы снова покажем границу принятия решения для набора данных, а также вероятности для класса 1:

```
In[114]:
fig, axes = plt.subplots(1, 2, figsize=(13, 5))

mglearn.tools.plot_2d_separator(
    gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(
    gbrt, X, ax=axes[1], alpha=.5, cm=mglearn.ReBl, function='predict_proba')

for ax in axes:
    # размещаем на графике точки обучающего и тестового наборов
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Характеристика 0")
    ax.set_ylabel("Характеристика 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Тест класс 0", "Тест класс 1", "Обуч класс 0",
               "Обуч класс 1"], ncol=4, loc=(.1, 1.1))
```

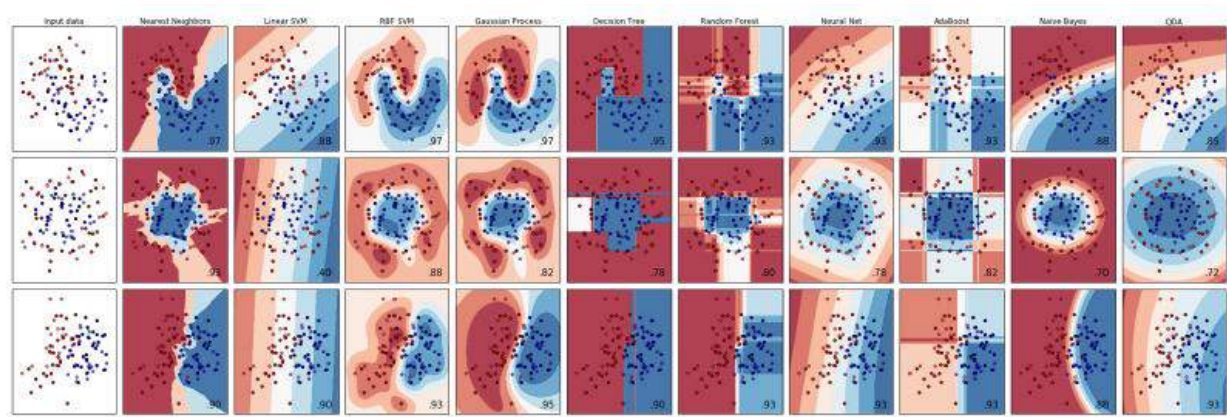


**Рис. 2.56** Граница принятия решений (слева) спрогнозированные вероятности для модели градиентного бустинга, показанной на рис. 2.55

Границы на этом рисунке определены гораздо более четко, а небольшие участки неопределенности отчетливо видны.

На сайте `scikit-learn` дается сравнение различных моделей и визуализации оценок неопределенности для этих моделей. Мы воспроизвели их на рис. 2.57 и рекомендуем ознакомиться с ними.





**Рис. 2.57** Сравнение нескольких классификаторов scikit-learn, построенных на синтетических наборах данных (изображение предоставлено <http://scikit-learn.org>)

## Неопределенность в мультиклассовой классификации

До сих пор мы говорили только об оценках неопределенности в бинарной классификации. Однако методы `decision_function` и `predict_proba` также можно применять в мультиклассовой классификации. Давайте применим их к набору данных Iris, который представляет собой пример 3-классовой классификации:

```
In[115]:
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbdt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbdt.fit(X_train, y_train)

In[116]:
print("Форма решающей функции: {}".format(gbdt.decision_function(X_test).shape))
# выведем первые несколько элементов решающей функции
print("Решающая функция:\n{}".format(gbdt.decision_function(X_test)[:6, :]))

Out[116]:
Форма решающей функции: (38, 3)
Решающая функция:
[[-0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

В мультиклассовой классификации `decision_function` имеет форму (`n_samples`, `n_classes`) и каждый столбец показывает «оценку определенности» для каждого класса, где высокая оценка означает большую вероятность данного класса, а низкая оценка означает меньшую вероятность этого класса. Вы можете получить прогнозы, исходя из этих



оценок, с помощью функции `np.argmax`. Она возвращает индекс максимального элемента массива для каждой точки данных:

```
In[117]:
print("Аргмакс решающей функции:\n{}".format(
    np.argmax(gbrt.decision_function(X_test), axis=1)))
print("Прогнозы:\n{}".format(gbrt.predict(X_test)))

Out[117]:
Аргмакс решающей функции:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 1 0 0 2 1 0]
Прогнозы:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 2 0 0 0 1 0 0 2 1 0]
```

Вывод `predict_proba` имеет точно такую же форму (`n_samples`, `n_classes`). И снова вероятности возможных классов для каждой точки данных дают в сумме 1:

```
In[118]:
# выведем первые несколько элементов predict_proba
print("Спрогнозированные вероятности:\n{}".format(gbrt.predict_proba(X_test)[:6]))
# покажем, что сумма значений в каждой строке равна 1
print("Суммы: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))

Out[118]:
Спрогнозированные вероятности:
[[ 0.107 0.784 0.109]
 [ 0.789 0.106 0.105]
 [ 0.102 0.108 0.789]
 [ 0.107 0.784 0.109]
 [ 0.108 0.663 0.228]
 [ 0.789 0.106 0.105]]
Суммы: [ 1.  1.  1.  1.  1.  1.]
```

Мы вновь можем получить прогнозы, вычислив `argmax` для `predict_proba`:

```
In[119]:
print("Аргмакс спрогнозированных вероятностей:\n{}".format(
    np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Прогнозы:\n{}".format(gbrt.predict(X_test)))

Out[119]:
Аргмакс спрогнозированных вероятностей:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 2 0 0 0 1 0 0 2 1 0]
Прогнозы:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 2 0 0 0 1 0 0 2 1 0]
```

Подводя итог, отметим, что `predict_proba` и `decision_function` всегда имеют форму (`n_samples`, `n_classes`), за исключением `decision_function` в случае бинарной классификации. В бинарной классификации `decision_function` имеет только один столбец, соответствующий «положительному» классу `classes_[1]`.

Для количества столбцов, равного `n_classes`, вы можете получить прогноз, вычислив `argmax` по столбцам. Однако будьте осторожны, если ваши классы – строки или вы используете целые числа, которые не являются последовательными и начинаются не с 0. Если вы хотите

сравнить результаты, полученные с помощью `predict`, с результатами `decision_function` или `predict_proba`, убедитесь, что используете атрибут `classes_` для получения фактических названий классов:

```
In[120]:
logreg = LogisticRegression()

# представим каждое целевое значение названием класса в наборе iris
named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("уникальные классы в обучающем наборе: {}".format(logreg.classes_))
print("прогнозы: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax решающей функции: {}".format(argmax_dec_func[:10]))
print("argmax объединенный с классами: {}".format(
    logreg.classes_[argmax_dec_func[:10]]))

Out[120]:
уникальные классы в обучающем наборе: ['setosa' 'versicolor' 'virginica']
прогнозы: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
argmax решающей функции: [1 0 2 1 1 0 1 2 1 1]
argmax объединенный с классами: ['versicolor' 'setosa' 'virginica' 'versicolor'
'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

## Выводы и перспективы

Мы начали эту главу с обсуждения такого понятия, как сложность модели, а затем рассказали об *обобщающей способности* (*generalization*), то есть о построении такой модели, которая может хорошо работать на новых, ранее неизвестных данных. Это привело нас к понятиям «недообучение», когда модель не может описать изменчивость обучающих данных, и «переобучение», когда модель слишком много внимания уделяет обучающим данным<sup>22</sup> и не способна хорошо обобщить новые данные.

Затем мы рассмотрели широкий спектр моделей машинного обучения для классификации и регрессии, их преимущества и недостатки, настройки сложности для каждой модели. Мы увидели, что для достижения хорошего качества работы во многих алгоритмах важное значение имеет установка правильных параметров. Кроме того, некоторые алгоритмы чувствительны к типу входных данных, и, в частности, к тому, как масштабированы признаки. Поэтому слепое применение алгоритма к данным без понимания исходных предположений модели и принципов работы параметров редко приводит к построению точной модели.

Эта глава содержит много информации об алгоритмах, но вам необязательно помнить все эти детали, чтобы понимать содержание следующих глав. Тем не менее некоторая информация о моделях,

---

<sup>22</sup> То есть модель стремится описать даже незначительные колебания в обучающих данных, которые могут иметь случайный характер (принимает шум за сигнал). – *Прим. пер.*

упомянутых здесь, и контексте использования этих моделей, имеет важное значение для успешного применения машинного обучения на практике. Ниже дается краткий обзор случаев использования той или иной модели:

#### *Ближайшие соседи*

Подходит для небольших наборов данных, хорош в качестве базовой модели, прост в объяснении.

#### *Линейные модели*

Считается первым алгоритмом, который нужно попробовать, хорош для очень больших наборов данных, подходит для данных с очень высокой размерностью.

#### *Наивный байесовский классификатор*

Подходит только для классификации. Работает даже быстрее, чем линейные модели, хорош для очень больших наборов данных и высокоразмерных данных. Часто менее точен, чем линейные модели.

#### *Деревья решений*

Очень быстрый метод, не нужно масштабировать данные, результаты можно визуализировать и легко объяснить.

#### *Случайные леса*

Почти всегда работают лучше, чем одно дерево решений, очень устойчивый и мощный метод. Не нужно масштабировать данные. Плохо работает с данными очень высокой размерности и разреженными данными.

#### *Градиентный бустинг деревьев решений*

Как правило, немного более точен, чем случайный лес. В отличие от случайного леса медленнее обучается, но быстрее предсказывает и требует меньше памяти. По сравнению со случайным лесом требует настройки большего числа параметров.

#### *Машины опорных векторов*

Мощный метод для работы с наборами данных среднего размера и признаками, измеренными в едином масштабе. Требует масштабирования данных, чувствителен к изменению параметров.

## *Нейронные сети*

Можно построить очень сложные модели, особенно для больших наборов данных. Чувствительны к масштабированию данных и выбору параметров. Большим моделям требуется много времени для обучения.

При работе с новым набором данных лучше начать с простой модели, например, с линейной модели, наивного байесовского классификатора или классификатора ближайших соседей, и посмотреть, как далеко можно продвинуться с точки зрения качества модели. Лучше изучив данные, вы можете выбрать алгоритм, который может строить более сложные модели, например, случайный лес, градиентный бустинг деревьев решений, SVM или нейронную сеть.

Теперь у вас уже есть некоторое представление о том, как применять, настраивать и анализировать модели, которые мы здесь рассмотрели. В этой главе мы сосредоточились на бинарной классификации, поскольку ее, как правило, легче всего интерпретировать. Большинство представленных алгоритмов могут решать задачи регрессии и классификации вариантов, при этом все алгоритмы классификации поддерживают как бинарную, так и мультиклассовую классификацию. Попробуйте применить любой из этих алгоритмов к наборам данных, включенным в `scikit-learn`, например, к наборам для регрессии `boston_housing` или `diabetes`, или к набору `digits` для мультиклассовой классификации. Экспериментирование с алгоритмами на различных наборах данных позволит вам лучше понять, насколько быстро обучаются различные алгоритмы, насколько легко анализировать построенные с их помощью модели и насколько эти алгоритмы чувствительны к типу данных.

Несмотря на то что мы проанализировали результаты применения различных параметров для исследованных нами алгоритмов, процесс построения модели, которая будет хорошо обобщать новые данные, выглядит немного сложнее. В главе 6 мы увидим, как правильно настраивать параметры и как автоматически найти оптимальные параметры.

Но для начала в следующей главе мы более детально рассмотрим обучение без учителя и предварительную обработку данных.

## ГЛАВА 3. МЕТОДЫ МАШИННОГО ОБУЧЕНИЯ БЕЗ УЧИТЕЛЯ И ПРЕДВАРИТЕЛЬНАЯ ОБРАБОТКА ДАННЫХ

Вторая группа алгоритмов машинного обучения, которую мы будем рассматривать, – это машинное обучение без учителя. Машинное обучение без учителя включает в себя все виды машинного обучения, когда ответ неизвестен и отсутствует учитель, указывающий ответ алгоритму. В машинном обучении без учителя есть лишь входные данные и алгоритму необходимо извлечь знания из этих данных.

### Типы машинного обучения без учителя

В этой главе мы рассмотрим два вида машинного обучения без учителя: преобразования данных и кластеризацию.

*Неконтролируемые преобразования (unsupervised transformations)* – это алгоритмы, создающие новое представление данных, которое в отличие от исходного представления человеку или алгоритму машинного обучения будет обработать легче. Общераспространенное применение неконтролируемых преобразований – сокращение размерности. Мы берем высокоразмерное представление данных, состоящее из множества признаков, и находим новый способ представления этих данных, обобщая основные характеристики и получая меньшее количество признаков. Общераспространенное применение сокращения размерности – получение двумерного пространства в целях визуализации.

Еще одно применение неконтролируемых преобразований – поиск компонент, из которых «состоят» данные. Примером такого преобразования является выделение тем из коллекций текстовых документов. Здесь задача состоит в том, чтобы найти неизвестные темы, обсуждаемые в коллекции документов, а также выяснить, какие темы встречаются в каждом документе. Это может быть полезно для отслеживания в социальных сетях обсуждений таких тем, как выборы, контроль огнестрельного оружия или жизнь поп-звезд.

С другой стороны, *алгоритмы кластеризации (clustering algorithms)* разбивают данные на отдельные группы схожих между собой элементов. Рассмотрим пример загрузки фотографий в социальной сети. Часто вы формируете запросы типа «покажите мне все фотографии, на которых изображен Иван Петров». Для выполнения подобных запросов, администрация сайта, возможно, захочет сгруппировать фотографии, на которых изображен один и тот же человек. Однако при этом неизвестно,

на каких загружаемых фотографиях кто показан, и неизвестно, какое количество различных пользователей присутствует на ваших фотографиях. Разумный подход заключался бы в том, чтобы извлечь все лица и разделить их на группы лиц, которые схожи между собой. Будем надеяться, что они соответствуют одному и тому же человеку и изображения в сгруппированном виде будут предъявлены вам.

## Проблемы машинного обучения без учителя

Главная проблема машинного обучения без учителя – оценка полезности информации, извлеченной алгоритмом. Алгоритмы машинного обучения без учителя, как правило, применяются к данным, которые не содержат никаких меток, таким образом, мы не знаем, каким должен быть правильный ответ. Поэтому очень трудно судить о качестве работы модели. Например, наш гипотетический алгоритм кластеризации мог бы сгруппировать вместе все фотографии лиц в профиль и все фотографии лиц в анфас. Перед нами, несомненно, один из способов разбить коллекцию фотографий лиц на группы, но это совсем не то, что нам нужно. Тем не менее у нас нет никакой возможности «рассказать» алгоритму, что мы ищем, и часто единственный способ оценить результат работы алгоритма машинного обучения без учителя – ручная проверка этого результата.

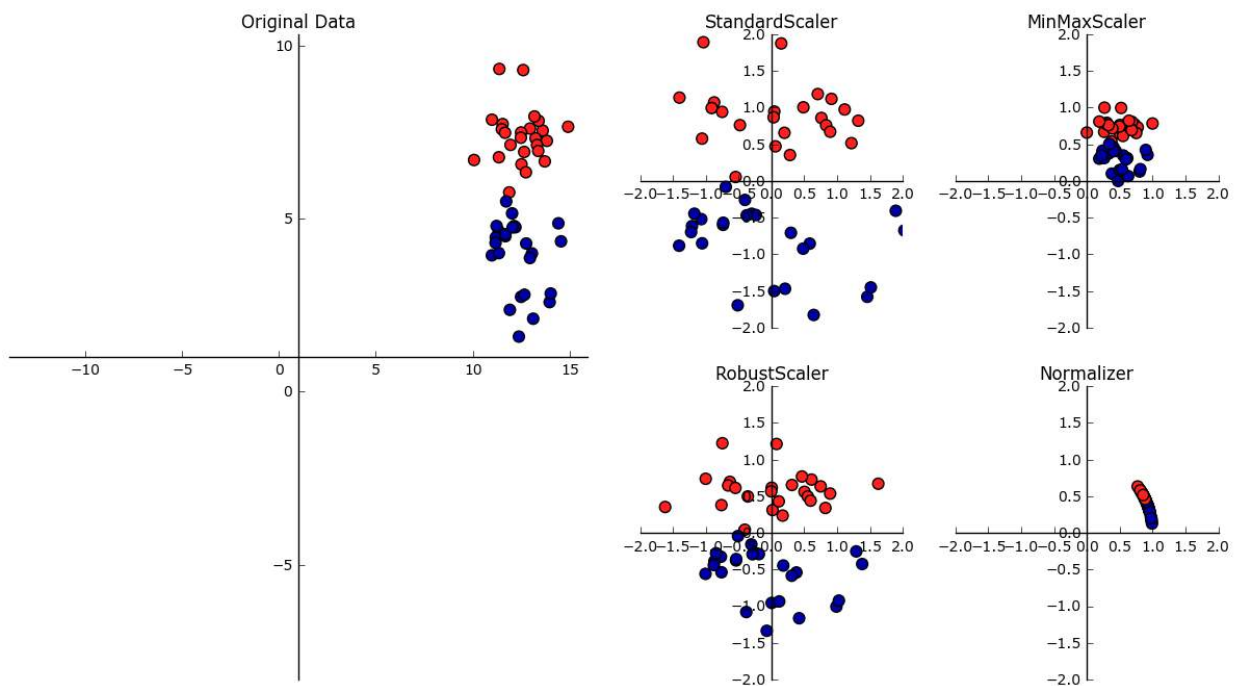
Как следствие, алгоритмы машинного обучения без учителя часто используются в разведочных целях, когда специалист хочет лучше изучить сами данные. Еще одно общераспространенное применение алгоритмов машинного обучения без учителя заключается в том, что они служат этапом предварительной обработки данных для алгоритмов машинного обучения с учителем. Изучение нового представления данных иногда может повысить правильность алгоритмов машинного обучения с учителем или может привести к снижению времени вычислений и потребления объема памяти.

Прежде чем начать знакомство с «реальными» алгоритмами машинного обучения без учителя, мы кратко рассмотрим некоторые простые методы предварительной обработки данных, которые часто могут пригодиться. Хотя предварительная обработка данных и масштабирование часто применяются вместе с алгоритмами контролируемого обучения, методы масштабирования не используют учителя, что делает их методами неконтролируемого обучения.

## Предварительная обработка и масштабирование

В предыдущей главе мы видели, что некоторые алгоритмы, например, нейронные сети и SVM, очень чувствительны к масштабированию данных. Поэтому обычной практикой является преобразование признаков с тем, чтобы итоговое представление данных было более подходящим для использования вышеупомянутых алгоритмов. Часто достаточно простого масштабирования признаков и корректировки данных. Программный код (рис. 3.1) показывает простой пример:

```
In[2]:  
mglearn.plots.plot_scaling()
```



**Рис. 3.1** Различные способы масштабирования и предварительной обработки данных

## Различные виды предварительной обработки

Первый график на рис. 3.1 соответствует синтетическому двуклассовому набору данных с двумя признаками. Первый признак (ось  $x$ ) принимает значения в диапазоне от 10 до 15. Второй признак (ось  $y$ ) принимает значения примерно в диапазоне от 1 до 9.

Следующие четыре графика показывают четыре различных способа преобразования данных, которые дают более стандартные диапазоны значений. Применение **StandardScaler** в **scikit-learn** гарантирует, что для каждого признака среднее будет равно 0, а дисперсия будет равна 1, в результате чего все признаки будут иметь один и тот же масштаб. Однако это масштабирование не гарантирует получение каких-то конкретных минимальных и максимальных значений признаков.



**RobustScaler** аналогичен **StandardScaler** в том плане, что в результате его применения признаки будут иметь один и тот же масштаб. Однако **RobustScaler** вместо среднего и дисперсии использует медиану и квартили<sup>23</sup>. Это позволяет **RobustScaler** игнорировать точки данных, которые сильно отличаются от остальных (например, ошибки измерений). Эти странные точки данных еще называются *выбросами* (*outliers*) и могут стать проблемой для остальных методов масштабирования.

С другой стороны, **MinMaxScaler** сдвигает данные таким образом, что все признаки находились строго в диапазоне от 0 до 1. Для двумерного набора данных это означает, что все данные помещаются в прямоугольник, образованный осью  $x$  с диапазоном значений от 0 и 1 и осью  $y$  с диапазоном значений от 0 и 1.

И, наконец, **Normalizer** осуществляет совершенно иной вид масштабирования. Он масштабирует каждую точку данных таким образом, чтобы вектор признаков имел евклидову длину 1. Другими словами, он проецирует точку данных на окружность с радиусом 1 (или сферу в случае большого числа измерений). Вектор умножается на инверсию своей длины. Подобная нормализация используется тогда, когда важным является направление (но не длина) вектора признаков.

## Применение преобразований данных

Теперь, когда мы увидели, что делают различные виды преобразований, давайте применим их, воспользовавшись **scikit-learn**. Мы будем использовать набор данных **cancer**, известный нам по главе 2. Методы предварительной обработки обычно применяются перед использованием алгоритма машинного обучения с учителем. Допустим, в качестве примера нам нужно применить ядерный SVM (**SVC**) к набору данных **cancer** и использовать **MinMaxScaler** для предварительной обработки данных. Мы начнем с того, что загрузим наш набор данных и разобьем его на тренировочный и тестовый наборы (обучающий и тестовый наборы нам нужны для оценки качества модели, которую мы построим с помощью алгоритма контролируемого обучения после предварительной обработки):

```
In[3]:
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
```

---

<sup>23</sup> Медиана множества чисел – это такое число  $x$ , при котором половина значений множества меньше  $x$ , а другая половина значений больше  $x$ . Нижний квартиль – это число  $x$ , ниже которого находится четверть значений, а верхний квартиль – это число  $x$ , выше которого находится четверть значений.

```
print(X_train.shape)
print(X_test.shape)
```

```
Out[3]:
(426, 30)
(143, 30)
```

Напомним, что набор содержит 569 точек данных, каждая из которых представлена 30 признаками. Мы разбиваем набор данных на 426 примеров в обучающей выборке и 143 примера в тестовой выборке.

Как и в случае с моделями контролируемого обучения, построенными ранее, мы сначала импортируем класс, который осуществляет предварительную обработку, а затем создаем его экземпляр:

```
In[4]:
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

Затем с помощью метода `fit` мы подгоняем `scaler` на обучающих данных. Для `MinMaxScaler` метод `fit` вычисляет минимальное и максимальное значения каждого признака на обучающем наборе. В отличие от классификаторов и регрессоров, описанных в главе 2, при вызове метода `fit` `scaler` работает с данными (`X_train`), а ответы (`y_train`) не используются:

```
In[5]:
scaler.fit(X_train)
```

```
Out[5]:
MinMaxScaler(copy=True, feature_range=(0, 1))
```

Чтобы применить преобразование, которое мы только что подогнали, то есть фактически *отмасштабировать* (*scale*) обучающие данные, мы воспользуемся методом `transform`. Метод `transform` используется в `scikit-learn`, когда модель возвращает новое представление данных:

```
In[6]:
# преобразовываем данные
X_train_scaled = scaler.transform(X_train)
# печатаем значения признаков в обучающем наборе до и после масштабирования
print("форма преобразованного массива: {}".format(X_train_scaled.shape))
print("min значение признака до масштабирования:\n {}".format(X_train.min(axis=0)))
print("max значение признака до масштабирования:\n {}".format(X_train.max(axis=0)))
print("min значение признака после масштабирования:\n {}".format(
    X_train_scaled.min(axis=0)))
print("max значение признака после масштабирования:\n {}".format(
    X_train_scaled.max(axis=0)))
```

```
Out[6]:
форма преобразованного массива: (426, 30)
min значение признака до масштабирования:
[  6.98  9.71 43.79 143.50  0.05  0.02  0.  0.  0.11
   0.05  0.12  0.36  0.76  6.80  0.  0.  0.  0.
   0.01  0.  7.93 12.02 50.41 185.20 0.07 0.03  0.
   0.  0.16  0.06]
max значение признака до масштабирования:
```

```
[ 28.11  39.28 188.5  2501.0    0.16   0.29   0.43   0.2
  0.300  0.100  2.87   4.88   21.98  542.20   0.03   0.14
  0.400  0.050  0.06   0.03   36.04  49.54  251.20  4254.00
  0.220  0.940  1.17   0.29   0.58   0.15]
min значение признака после масштабирования:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
max значение признака после масштабирования:
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

Преобразованные данные имеют такую же форму, что и исходные данные – признаки просто смещены и масштабированы. Видно, что теперь все признаки принимают значения в диапазоне от 0 до 1, как нам и требовалось.

Чтобы применить SVM к масштабированным данным, мы должны преобразовать еще тестовый набор. Это снова делается с помощью вызова метода `transform`, на этот раз для `X_test`:

```
In[7]:
# преобразовываем тестовые данные
X_test_scaled = scaler.transform(X_test)
# печатаем значения признаков в тестовом наборе после масштабирования
print("min значение признака после масштабирования:\n{}".format(X_test_scaled.min(axis=0)))
print("max значение признака после масштабирования:\n{}".format(X_test_scaled.max(axis=0)))

Out[7]:
min значение признака после масштабирования:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.  0.  0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.  0. -0.032  0.007
  0.027  0.058  0.02  0.009  0.109  0.026  0.  0. -0.  -0.002]
max значение признака после масштабирования:
[ 0.958  0.815  0.956  0.894  0.811  1.22  0.88  0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07  0.924  1.205  1.631]
```

Возможно, полученные результаты несколько удивят вас: после масштабирования минимальные и максимальные значения признаков в тестовом наборе не равны 0 и 1. Некоторые признаки даже выходят за пределами диапазона 0-1! Объяснить это можно тем, что `MinMaxScaler` (и все остальные типы масштабирования) всегда применяют одинаковое преобразование к обучающему и тестовому наборам. Это означает, что метод `transform` всегда вычитает минимальное значение, вычисленное для обучающего набора, и делит на ширину диапазона, вычисленную также для обучающего набора. Минимальное значение и ширина диапазона для обучающего набора могут отличаться от минимального значения и ширины диапазона для тестового набора.

## Масштабирование обучающего и тестового наборов одинаковым образом

Чтобы модель контролируемого обучения работала на тестовом наборе, важно преобразовать обучающий и тестовый наборы одинаковым

образом. Следующий пример (рис. 3.2) показывает, что произошло бы, если бы мы использовали минимальное значение и ширину диапазона, отдельно вычисленные для тестового набора:

```
In[8]:
from sklearn.datasets import make_blobs
# создаем синтетические данные
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# разбиваем их на обучающий и тестовый наборы
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

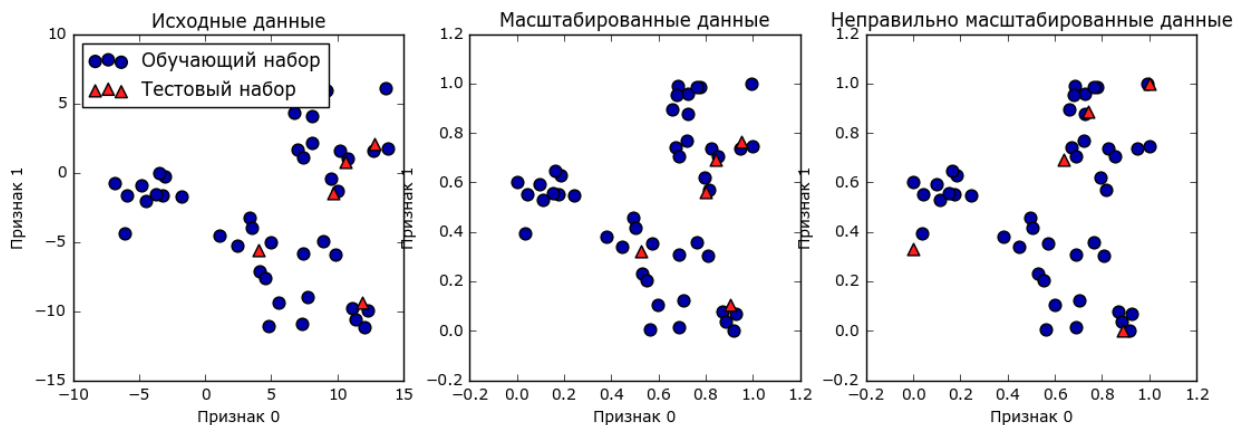
# размещаем на графике обучающий и тестовый наборы
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c=mglearn.cm2(0), label="Обучающий набор", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                c=mglearn.cm2(1), label="Тестовый набор", s=60)
axes[0].legend(loc='upper left')
axes[0].set_title("Исходные данные")

# масштабируем данные с помощью MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# визуализируем правильно масштабированные данные
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="Обучающий набор", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                c=mglearn.cm2(1), label="Тестовый набор", s=60)
axes[1].set_title("Масштабированные данные")

# масштабируем тестовый набор отдельно
# чтобы в тестовом наборе min значение каждого признака было равно 0
# а max значение каждого признака равнялось 1
# НЕ ДЕЛАЙТЕ ТАК! Только в ознакомительных целях.
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# визуализируем неправильно масштабированные данные
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mglearn.cm2(0), label="Обучающий набор", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1],
                marker='^', c=mglearn.cm2(1), label="Тестовый набор", s=60)
axes[2].set_title("Неправильно масштабированные данные")
for ax in axes:
    ax.set_xlabel("Признак 0")
    ax.set_ylabel("Признак 1")
```



**Рис. 3.2** Результаты одинакового масштабирования обучающего и тестового наборов (центр) и отдельного масштабирования обучающего и тестового наборов (справа)

Первый график – это немасштабированный двумерный массив данных, наблюдения обучающего набора показаны кружками, а наблюдения тестового набора показаны треугольниками. Второй график – те же самые данные, но масштабированы с помощью `MinMaxScaler`. Здесь мы вызвали метод `fit` для обучающего набора, а затем вызвали метод `transform` для обучающего и тестового наборов. Как видите, набор данных на втором графике идентичен набору, приведенному на первом графике, изменились лишь метки осей. Теперь все признаки принимают значения в диапазоне от 0 до 1. Кроме того, видно, что минимальные и максимальные значения признаков в тестовом наборе (треугольники) не равны 0 и 1.

Третий график показывает, что произойдет, если отмасштабируем обучающий и тестовый наборы по отдельности. В этом случае минимальные и максимальные значения признаков в обучающем и тестовом наборах равны 0 и 1. Но теперь набор данных выглядит иначе. Тестовые точки причудливым образом сместились, поскольку масштабированы по-другому. Мы изменили расположение данных произвольным образом. Очевидно, это совсем не то, что нам нужно.

Еще один способ задуматься о неправильности этих действий заключается в том, что представить тестовый набор в виде одной точки. Не существует способа правильно масштабировать единственную точку данных, чтобы с помощью `MinMaxScaler` получить значения минимума и максимума. Однако размер тестового набора не должен влиять на обработку данных.

## Быстрые и эффективные альтернативные способы ПОДГОНКИ МОДЕЛЕЙ

Как правило, вам нужно сначала подогнать модель на некотором наборе данных с помощью метода `fit`, а затем выполнить преобразование набора с помощью метода `transform`. Это весьма распространенная задача, которую можно выполнить более эффективно, чем просто вызвать метод `fit`, а затем вызвать метод `transform`. Что касается вышеописанного случая, все модели, которые используют метод `transform`, также позволяют воспользоваться методом `fit_transform`. Ниже дан пример использования `StandardScaler`:

```
In[9]:
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# последовательно вызываем методы fit и transform (используем цепочку методов)
X_scaled = scaler.fit(X).transform(X)
# тот же самый результат, но более эффективный способ вычислений
X_scaled_d = scaler.fit_transform(X)
```

Несмотря на то что применение `fit_transform` вовсе не обязательно будет более эффективным для всех моделей, эффективная практика заключается в использовании этого метода для преобразования обучающего набора.

## Влияние предварительной обработки на машинное обучение с учителем

Теперь давайте вернемся к набору данных `cancer` и посмотрим, как использование `MinMaxScaler` повлияет на обучение `SVC` (мы выполняем то же самое масштабирование, что делали в главе 2, но другим способом). Во-первых, давайте для сравнения снова подгоним `SVC` на исходных данных:

```
In[10]:
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Правильность на тестовом наборе: {:.2f}".format(svm.score(X_test, y_test)))

Out[10]:
Правильность на тестовом наборе: 0.63
```

Теперь давайте отмасштабируем данные с помощью `MinMaxScaler` перед тем, как подгонять `SVC`:

```

In[11]:
# предварительная обработка с помощью шкалирования 0-1
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# построение SVM на масштабированных обучающих данных
svm.fit(X_train_scaled, y_train)
# оценка правильности для масштабированного тестового набора
print("Правильность на масштабированном тестовом наборе: {:.2f}".format(
    svm.score(X_test_scaled, y_test)))

Out[11]:
Правильность на масштабированном тестовом наборе: 0.97

```

Как мы уже видели ранее, эффект масштабирования данных весьма существенен. Хотя масштабирование данных не предполагает каких-либо сложных математических расчетов, эффективная практика заключается в том, чтобы использовать методы масштабирования, предлагаемые **scikit-learn**, а не создавать их заново самостоятельно, поскольку легко ошибиться даже в этих простых вычислениях.

Кроме того, можно легко заменить один алгоритм предварительной обработки на другой, сменив имя используемого класса, поскольку все классы предварительной обработки имеют один и тот же интерфейс, состоящий из методов **fit** и **transform**:

```

In[12]:
# предварительная обработка с помощью масштабирования
# нулевым средним и единичной дисперсией
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# построение SVM на масштабированных обучающих данных
svm.fit(X_train_scaled, y_train)

# оценка правильности для масштабированного тестового набора
print("Правильность SVM на тестовом наборе: {:.2f}".format(svm.score(X_test_scaled, y_test)))

Out[12]:
Правильность SVM на тестовом наборе: 0.96

```

Теперь узнав, как работают простые преобразования, выполняющие предварительную обработку данных, давайте перейдем к более интересным преобразованиям, использующим машинное обучение без учителя.

## Снижение размерности, выделение признаков и множественное обучение

Как мы уже говорили ранее, преобразование данных с помощью неконтролируемого обучения может быть обусловлено многими причинами. Наиболее распространенные причины – визуализация,



сжатие данных, а также поиск такого представления данных, которое даст больше информации в ходе дальнейшей обработки.

Одним из самых простых и наиболее широко используемых алгоритмов контролируемого обучения является анализ главных компонент (principal component analysis, PCA). Кроме того, мы рассмотрим еще два алгоритма: факторизацию неотрицательных матриц (non-negative matrix factorization, NMF), которая обычно используется для выделения признаков, и стохастическое вложение соседей с распределением Стьюдента (t-distributed stochastic neighbor embedding, t-SNE), которое обычно используется для визуализации с использованием двумерных диаграмм рассеяния.

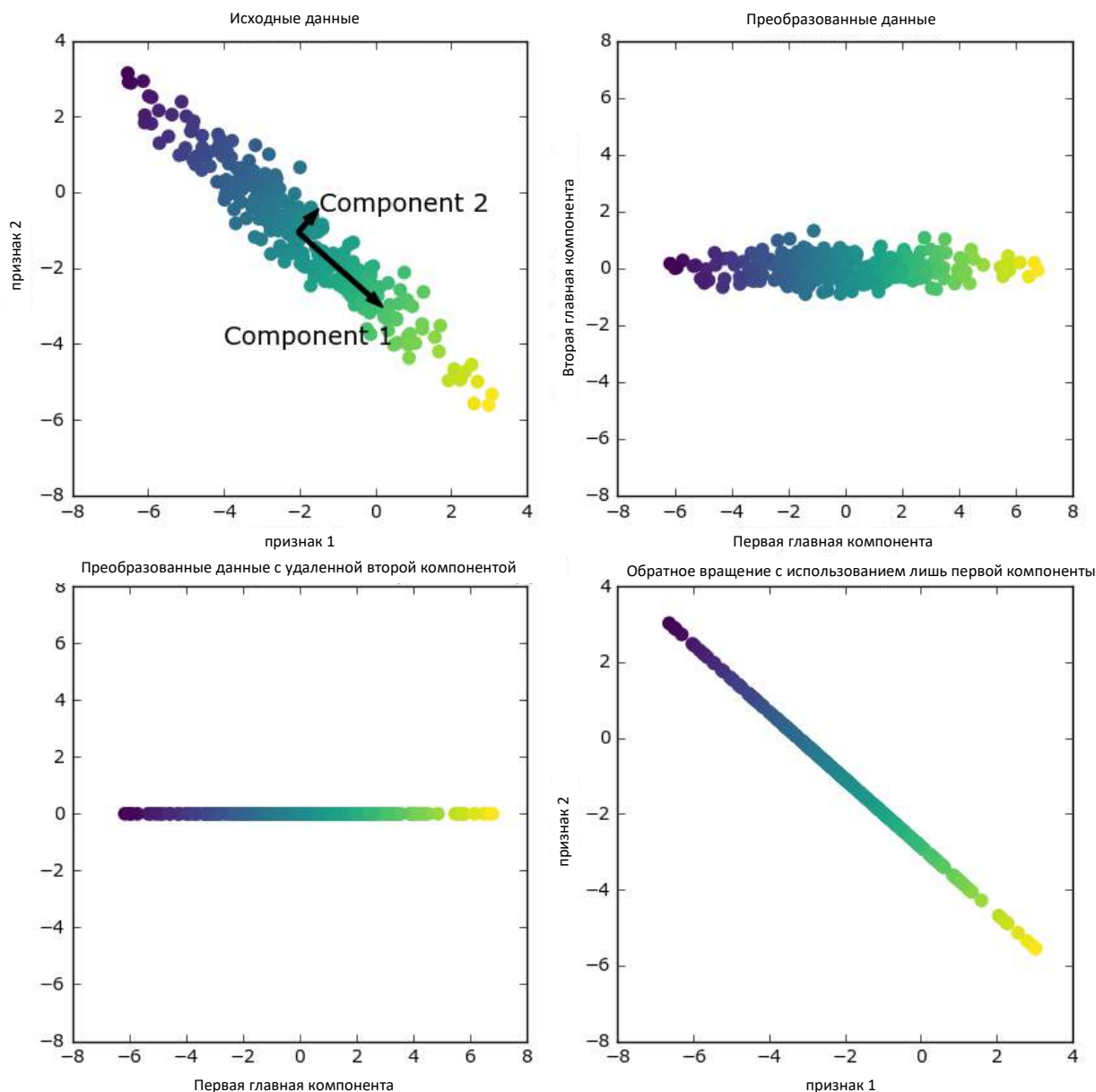
## Анализ главных компонент (PCA)

Анализ главных компонент представляет собой метод, который осуществляет вращение данных с тем, чтобы преобразованные признаки не коррелировали между собой. Часто это вращение сопровождается выбором подмножества новых признаков в зависимости от их важности с точки зрения интерпретации данных. Следующий пример (рис. 3.3) иллюстрирует результат применения PCA к синтетическому двумерному массиву данных:

```
In[13]:  
mglearn.plots.plot_pca_illustration()
```

Первый рис. (вверху слева) показывает исходные точки данных, выделенные цветом для лучшей дискриминации. Алгоритм начинает работу с того, что сначала находит направление максимальной дисперсии, помеченное как «компонента 1». Речь идет о направлении (или векторе) данных, который содержит большую часть информации, или другими словами, направление, вдоль которого признаки коррелируют друг с другом сильнее всего. Затем алгоритм находит направление, которое содержит наибольшее количество информации, и при этом ортогонально (расположено под прямым углом) первому направлению. В двумерном пространстве существует только одна возможная ориентация, расположенная под прямым углом, но в пространствах большей размерности может быть (бесконечно) много ортогональных направлений. Хотя эти две компоненты изображаются в виде стрелок, на самом деле не имеет значения, где начало, а где конец, мы могли бы нарисовать первую компоненту, выходящую из центра в верхний левый угол, а не в нижний правый. Направления, найденные с помощью этого алгоритма, называются *главными компонентами* (*principal components*), поскольку они являются основными

направлениями дисперсии данных. В целом максимально возможное количество главных компонент равно количеству исходных признаков.



**Рис. 3.3** Преобразование данных с помощью PCA

Второй график (вверху справа) показывает те же самые данные, но теперь повернутые таким образом, что первая главная компонента совпадает с осью x, а вторая главная компонента совпадает с осью y. Перед вращением из каждого значения данных вычитается среднее, таким образом, преобразованные данные центрированы около нуля. В новом представлении данных, найденном с помощью PCA, две оси становятся некоррелированными. Это означает, что в новом представлении все элементы корреляционной матрицы данных, кроме диагональных, будут равны нулю.

Мы можем использовать PCA для уменьшения размерности, сохранив лишь несколько главных компонент. В данном примере мы можем оставить лишь первую главную компоненту, как показано на третьем графике рис. 3.3 (внизу слева). Это уменьшит размерность данных: из двумерного массива данных получаем одномерный массив данных. Однако следует отметить, что вместо того, чтобы оставить лишь один из исходных признаков, мы находим наиболее интересное направление (выходящее из верхнего левого угла в нижний правый на первом графике) и оставляем это направление, т.е. первую главную компоненту.

И, наконец, мы можем отменить вращение и добавить обратно среднее значение к значениям данных. В итоге получим данные, показанные на последнем графике рис. 3.3. Эти точки располагаются в пространстве исходных признаков, но мы оставили лишь информацию, содержащуюся в первой главной компоненте. Это преобразование иногда используется, чтобы удалить эффект шума из данных или показать, какая часть информации сохраняется при использовании главных компонент.

### Применение PCA к набору данных cancer для визуализации

Одним из наиболее распространенных применений PCA является визуализация высокоразмерных наборов данных. Как мы видели в главе 1, довольно сложно построить диаграммы рассеяния для данных, которые включают больше двух признаков. Для набора данных Iris мы смогли построить матрицу диаграмм рассеяния (рис. 1.3 в главе 1), которая дала нам частичное представление о данных, показав все возможные комбинации двух признаков. Но если мы захотим взглянуть на набор данных Breast Cancer, использование матрицы диаграмм рассеяния будет затруднительным. Этот набор данных содержит 30 признаков, которые привели бы к  $30 * 14 = 420$  диаграммам рассеяния! Мы никогда не сможем детально просмотреть все эти графики, не говоря уже об их интерпретации.

Впрочем, можно воспользоваться более простой визуализацией, вычислив гистограммы распределения значений признаков для двух классов, доброкачественных и злокачественных опухолей (рис. 3.4):

```
In[14]:
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]

ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color=mplotlib.cm3(0), alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color=mplotlib.cm3(2), alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Значение признака")
ax[0].set_ylabel("Частота")
```

```
ax[0].legend(["доброкачественная", "злокачественная"], loc="best")
fig.tight_layout()
```

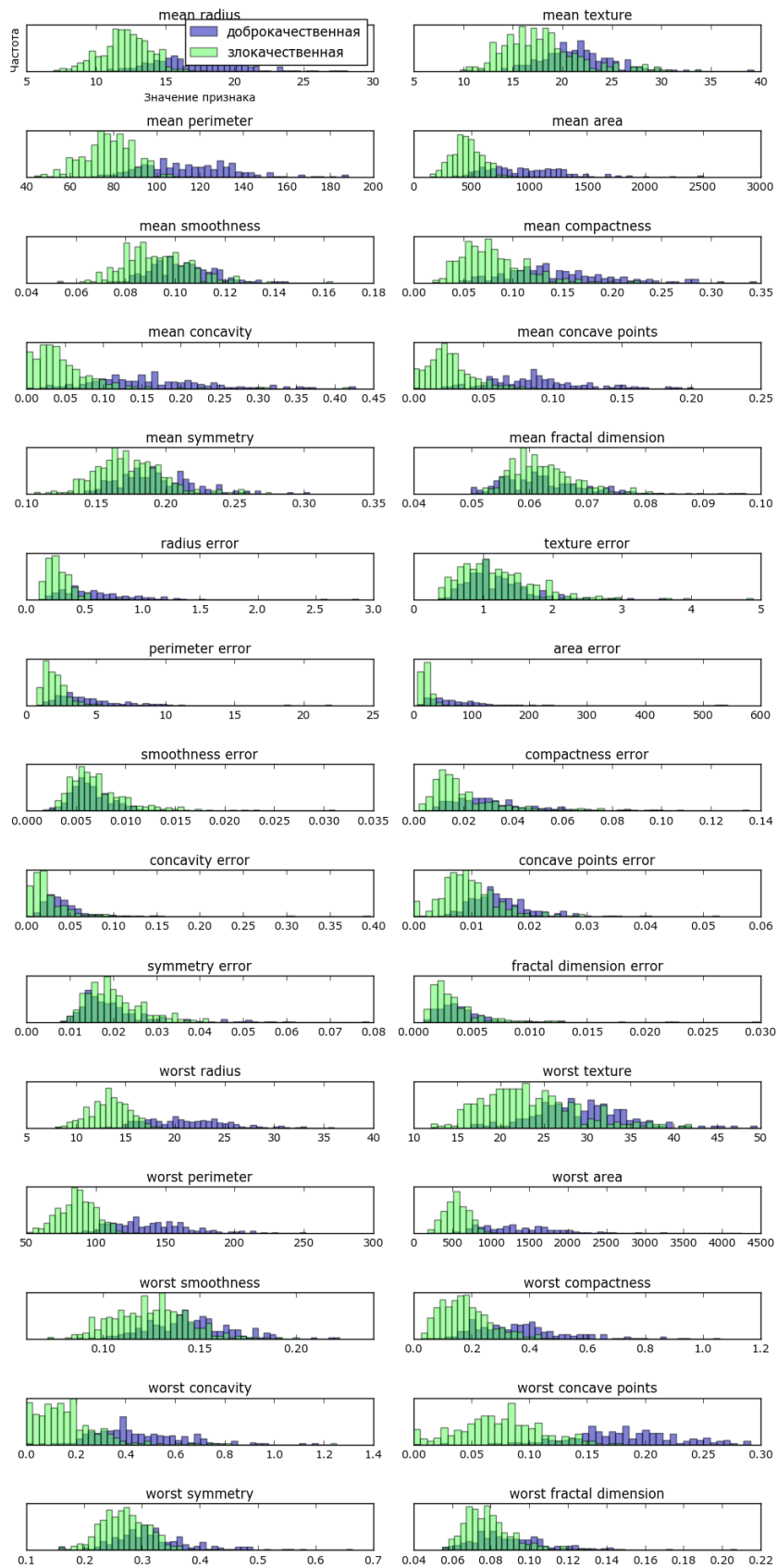


Рис. 3.4 Преобразование данных с помощью PCA

В данном случае мы строим для каждого признака гистограмму, подсчитывая частоту встречаемости точек данных в пределах границ интервалов (этот интервал еще называют бином). Каждый график содержит две наложенные друг на друга гистограммы, первая – для всех точек, относящихся к классу «доброкачественная опухоль» (синий цвет), а вторая – для всех точек, относящихся к классу «злокачественная опухоль» (зеленый цвет). Это дает нам некоторое представление о распределении каждого признака по двум классам и позволяет нам строить предположения о том, какие признаки лучше всего дискриминируют злокачественные и доброкачественные опухоли. Например, признак «smoothness error», похоже, довольно малоинформативен, потому что две гистограммы, построенные для данного признака, большей частью накладываются друг на друга, в то время признак «worst concave points» кажется весьма информативным, поскольку гистограммы, построенные для этого признака, практически не накрывают друг друга.

Однако этот график не дает нам никакой информации о взаимодействии между переменными и взаимосвязях между признаками и классами зависимой переменной. Используя PCA, мы можем учесть главные взаимодействия и получить несколько более полную картину. Мы можем найти первые две главные компоненты и визуализировать данные в этом новом двумерном пространстве с помощью одной диаграммы рассеяния.

Перед тем, как применить PCA, мы отмасштабируем наши данные таким образом, чтобы каждый признак имел единичную дисперсию, воспользовавшись `StandardScaler`:

```
In[15]:
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

Обучение PCA и его применение так же просто, как применение преобразований, выполняющихся в ходе предварительной обработки. Мы создаем экземпляр объекта PCA, находим главные компоненты, вызвав метод `fit`, а затем применяем вращение и снижение размерности, вызвав метод `transform`. По умолчанию PCA лишь поворачивает (и смещает) данные, но сохраняет все главные компоненты. Чтобы уменьшить размерность данных, нам нужно указать, сколько компонент мы хотим сохранить при создании объекта PCA:

```

In[16]:
from sklearn.decomposition import PCA
# оставляем первые две главные компоненты
pca = PCA(n_components=2)
# подгоняем модель PCA на наборе данных breast cancer
pca.fit(X_scaled)

# преобразуем данные к первым двум главным компонентам
X_pca = pca.transform(X_scaled)
print("Форма исходного массива: {}".format(str(X_scaled.shape)))
print("Форма массива после сокращения размерности: {}".format(str(X_pca.shape)))

Out[16]:
Форма исходного массива: (569, 30)
Форма массива после сокращения: (569, 2)

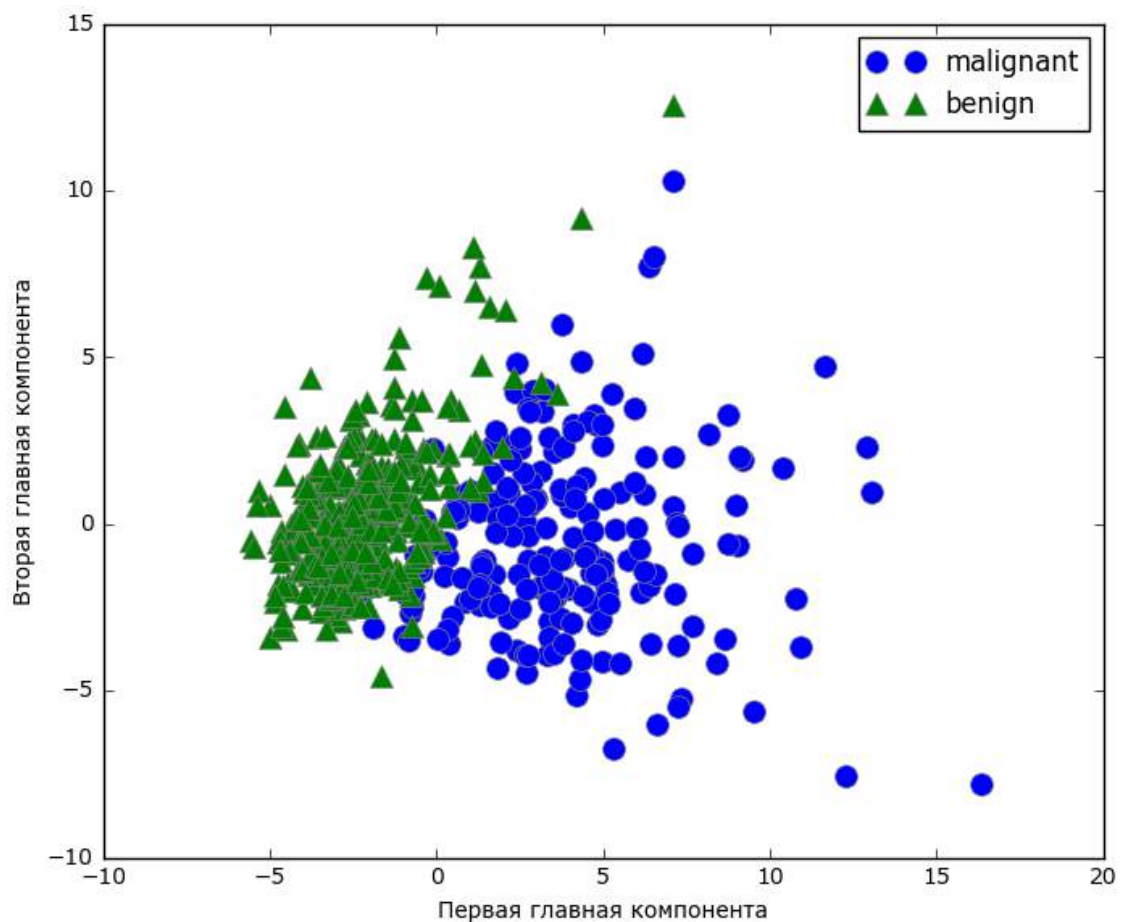
```

Теперь мы можем построить график первых двух главных компонент (рис. 3.5):

```

In[17]:
# строим график первых двух главных компонент, классы выделены цветом
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("Первая главная компонента")
plt.ylabel("Вторая главная компонента")

```



**Рис. 3.5** Двумерная диаграмма рассеяния для набора данных Breast Cancer с использованием первых двух главных компонент



Важно отметить, что РСА является методом машинного обучения без учителя и не использует какой-либо информации о классах при поиске поворота. Он просто анализирует корреляционные связи в данных. Для точечного графика, показанного здесь, мы построили график, где по оси *x* отложена первая главная компонента, по оси *y* – вторая главная компонента, а затем воспользовались информацией о классах, чтобы выделить точки разным цветом. Вы можете увидеть, что в рассматриваемом двумерном пространстве эти два класса разделены достаточно хорошо. Это наводит на мысль, что даже линейный классификатор (который проведет прямую линию в этом пространстве) сможет достаточно хорошо разделить два класса. Кроме того, мы можем увидеть, что случаи злокачественных опухолей (синие точки) более распространены, чем случаи доброкачественных опухолей (зеленые точки) – что отчасти было видно на гистограммах рис. 3.4.

Недостаток РСА заключается в том, что эти две оси графика часто бывает сложно интерпретировать. Главные компоненты соответствуют направлениям данных, поэтому они представляют собой комбинации исходных признаков. Однако, как мы скоро увидим, эти комбинации обычно очень сложны. Сами главные компоненты могут быть сохранены в атрибуте `components_` объекта РСА в ходе подгонки:

```
In[18]:
print("форма главных компонент: {}".format(pca.components_.shape))
```

```
Out[18]:
форма главных компонент: (2, 30)
```

Каждая строка в атрибуте `components_` соответствует одной главной компоненте и они отсортированы по важности (первой приводится первая главная компонента и т.д.). Столбцы соответствуют атрибуту исходных признаков для объекта РСА в этом примере, «mean radius», «mean texture» и т.д. Давайте посмотрим на содержимое атрибута `components_`:

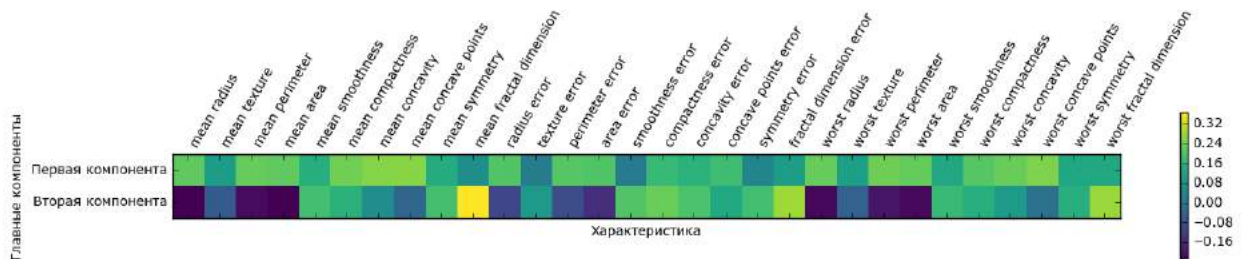
```
components_:
In[19]:
print("компоненты PCA:\n{}".format(pca.components_))

Out[19]:
компоненты PCA:
[[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064
   0.206  0.017  0.211  0.203  0.015  0.17  0.154  0.183  0.042  0.103
   0.228  0.104  0.237  0.225  0.128  0.21  0.229  0.251  0.123  0.132]
 [-0.234 -0.06 -0.215 -0.231  0.186  0.152  0.06 -0.035  0.19  0.367
  -0.106  0.09 -0.089 -0.152  0.204  0.233  0.197  0.13  0.184  0.28
  -0.22 -0.045 -0.2 -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]]
```

Кроме того, с помощью тепловой карты (рис. 3.6) мы можем визуализировать коэффициенты, чтобы упростить их интерпретацию:



```
In[20]:
plt.matshow(pca.components_, cmap='viridis')
plt.yticks([0, 1], ["Первая компонента", "Вторая компонента"])
plt.colorbar()
plt.xticks(range(len(cancer.feature_names)),
            cancer.feature_names, rotation=60, ha='left')
plt.xlabel("Характеристика")
plt.ylabel("Главные компоненты")
```



**Рис. 3.6** Тепловая карта первых двух главных компонент для набора данных рака Breast Cancer

Вы можете увидеть, что в первой компоненте коэффициенты всех признаков имеют одинаковый знак (они положительные, но, как мы уже говорили ранее, не имеет значения, какое направление указывает стрелка). Это означает, что существует общая корреляция между всеми признаками. Высоким значениям одного признака будут соответствовать высокие значения остальных признаков. Во второй компоненте коэффициенты признаков имеют разные знаки. Обе компоненты включают все 30 признаков. Смешивание всех признаков – это как раз то, что усложняет интерпретацию осей на рис. 3.6.

**Метод «собственных лиц» (eigenfaces) для выделения характеристик**  
Еще одно применение PCA, о котором мы уже упоминали ранее, – это выделение признаков. Идея, лежащая в основе выделения признаков, заключается в поиске нового представления данных, которое в отличие от исходного лучше подходит для анализа. Отличный пример, показывающий, что выделение признаков может быть полезно, – это работа с изображениями. Изображения состоят из пикселей, обычно хранящихся в виде интенсивностей красной, зеленой и синей составляющих цвета (RGB). Объекты в изображениях, как правило, состоят из тысяч пикселей и лишь все вместе эти пиксели приобретают смысл.

Мы приведем очень простой пример того, как можно применить выделение признаков к изображениям с помощью PCA. Для этого мы воспользуемся набором данных Labeled Faces in the Wild. Этот набор данных содержит изображения лиц знаменитостей, загруженных из Интернета, и включает в себя лица политиков, певцов, актеров и спортсменов с начала 2000-х годов. Мы преобразуем эти фотографии в

оттенки серого, а также уменьшив их для более быстрой обработки. Вы можете увидеть некоторые изображения на рис. 3.7:

```
In[21]:
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                        subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```



**Рис. 3.7** Некоторые изображения из набора данных Labeled Faces in the Wild

Получаем 3023 изображения размером 87 х 65 пикселей, принадлежащие 62 различным людям:

```
In[22]:
print("форма массива изображений лиц: {}".format(people.images.shape))
print("количество классов: {}".format(len(people.target_names)))
```

```
Out[22]:
форма массива изображений лиц: (3023, 87, 65)
количество классов: 62
```

Однако данные немного асимметричны. Как вы можете здесь увидеть, он содержит большое количество изображений Джорджа Буша и Колина Пауэлла:

```
In[23]:
# вычисляем частоту встречаемости каждого ответа
counts = np.bincount(people.target)
# печатаем частоты рядом с ответами
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{0:25} {1:3}".format(name, count), end=' ')
    if (i + 1) % 3 == 0:
        print()
```

Out[23]:

Alejandro Toledo	39	Alvaro Uribe	35	Amelie Mauresmo	21
Andre Agassi	36	Angelina Jolie	20	Ariel Sharon	77
Arnold Schwarzenegger	42	Atal Bihari Vajpayee	24	Bill Clinton	29
Carlos Menem	21	Colin Powell	236	David Beckham	31
Donald Rumsfeld	121	George Robertson	22	George W Bush	530
Gerhard Schroeder	109	Gloria Macapagal Arroyo	44	Gray Davis	26
Guillermo Coria	30	Hamid Karzai	22	Hans Blix	39
Hugo Chavez	71	Igor Ivanov	20	Jack Straw	28
Jacques Chirac	52	Jean Chretien	55	Jennifer Aniston	21
Jennifer Capriati	42	Jennifer Lopez	21	Jeremy Greenstock	24
Jiang Zemin	20	John Ashcroft	53	John Negroponte	31
Jose Maria Aznar	23	Juan Carlos Ferrero	28	Junichiro Koizumi	60
Kofi Annan	32	Laura Bush	41	Lindsay Davenport	22
Lleyton Hewitt	41	Luiz Inacio Lula da Silva	48	Mahmoud Abbas	29
Megawati Sukarnoputri	33	Michael Bloomberg	20	Naomi Watts	22
Nestor Kirchner	37	Paul Bremer	20	Pete Sampras	22
Recep Tayyip Erdogan	30	Ricardo Lagos	27	Roh Moo-hyun	32
Rudolph Giuliani	26	Saddam Hussein	23	Serena Williams	52
Silvio Berlusconi	33	Tiger Woods	23	Tom Daschle	25
Tom Ridge	33	Tony Blair	144	Vicente Fox	32
Vladimir Putin	49	Winona Ryder	24		

Чтобы данные стали менее асимметричными, мы будем рассматривать не более 50 изображений каждого человека (в противном случае выделение признаков будет перегружено большим количеством изображений Джорджа Буша):

```
In[24]:
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]

# для получения большей стабильности масштабируем шкалу оттенков серого так, чтобы значения
# были в диапазоне от 0 до 1 вместо использования шкалы значений от 0 до 255
X_people = X_people / 255.
```

Общая задача распознавания лиц заключается в том, чтобы спросить, не принадлежит ли незнакомое фото уже известному человеку из базы данных. Она применяется при составлении фотоколлекций, в социальных сетях и программах обеспечения безопасности. Один из способов решения этой задачи заключается в построении классификатора, в котором каждый человек представляет собой отдельный класс. Однако изображения, записанные в базах лиц, обычно принадлежат большому количеству самых различных людей и при этом очень мало фотографий принадлежат одному и тому же человеку (то есть очень мало обучающих примеров, принадлежащих одному классу). Для

большинства классификаторов это представляет проблему. Кроме того, часто необходимо добавить фотографии новых людей, при этом не перестраивая заново огромную модель.

Самое простое решение – использовать классификатор одного ближайшего соседа, который ищет лицо, наиболее схожее с классифицируемым. Этот классификатор в принципе может работать только с одним обучающим примером в классе. Давайте посмотрим, насколько хорошо здесь работает `KNeighborsClassifier`:

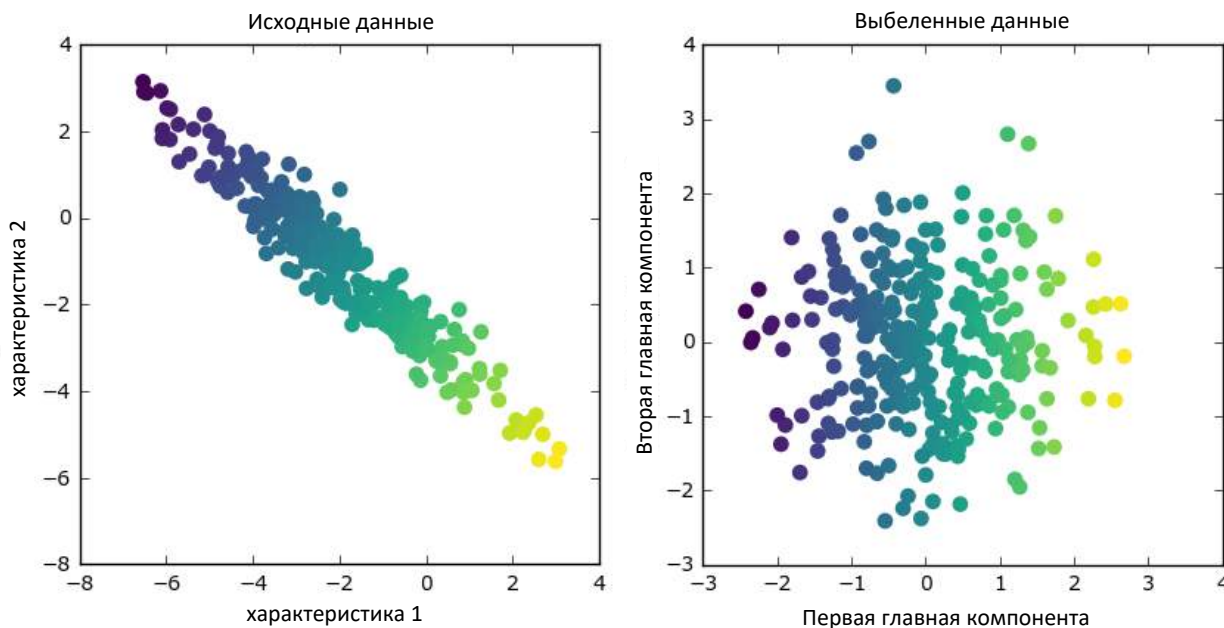
```
In[25]:
from sklearn.neighbors import KNeighborsClassifier
# разбиваем данные на обучающий и тестовый наборы
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# строим KNeighborsClassifier с одним соседом
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Правильность на тестовом наборе для 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

```
Out[25]:
Правильность на тестовом наборе для 1-nn: 0.27
```

Мы получаем правильность 26.6%, на самом деле это неплохо для классификационной задачи с 62 классами (случайное угадывание даст вам правильность около  $1/62 = 1.61\%$ ), но и не так велико. Мы правильно распознаем лишь каждое четвертое изображение человека.

И вот именно здесь применяется PCA. Вычисление расстояний в исходном пиксельном пространстве – довольно неудачный способ измерить сходство между лицами. Используя пиксельное представление для сопоставления двух изображений, мы сравниваем значение каждого отдельного пикселя по шкале градаций серого со значением пикселя в соответствующем положении на другом изображении. Это представление довольно сильно отличается от интерпретации изображений лиц людьми и крайне трудно выделить характеристики лица с использованием этого исходного представления. Например, использование пиксельных расстояний означает, что смещение лица на один пиксель вправо соответствует резкому изменению, дающему совершенно другое представление данных. Мы рассчитываем на то, что использование расстояний вдоль главных компонент может улучшить правильность. Здесь мы воспользуемся опцией PCA *выбеливание* (*whitening*), которая преобразует компоненты к одному и тому же масштабу. Операция выбеливания аналогична применению `StandardScaler` после преобразования. Повторно используя данные, приведенные на рис. 3.3, выбеливание не только поворачивает данные, но и масштабирует их таким образом, чтобы центральный график представлял собой окружность вместо эллипса (см. рис. 3.8):

```
In[26]:
mglearn.plots.plot_pca_whitening()
```



**Рис. 3.8** Преобразование данных с использованием выбеливания

Мы подгоняем объект PCA на обучающих данных и извлекаем первые 100 главных компонент. Затем мы преобразуем обучающие и тестовые данные:

```
In[27]:
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print("обучающие данные после PCA: {}".format(X_train_pca.shape))
```

```
Out[27]:
обучающие данные после PCA: (1537, 100)
```

Новые данные содержат 100 новых признаков, первые 100 главных компонент. Теперь мы можем использовать новое представление, чтобы классифицировать наши изображения, используя классификатор одного ближайшего соседа:

```
In[28]:
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Правильность на тестовом наборе: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

```
Out[28]:
Правильность на тестовом наборе: 0.36
```

Наша правильность улучшилась весьма значительно, с 26.6% до 35.7%, это подтверждает наше предположение о том, что главные компоненты могут дать лучшее представление данных.

Работая с изображениями, мы можем легко визуализировать найденные главные компоненты. Вспомним, что компоненты соответствуют направлениям в пространстве входных данных. Пространство входных данных здесь представляет собой изображения в градациях серого размером 87х65 пикселей, поэтому направления внутри этого пространства также являются изображениями в градациях серого размером 87х65 пикселей.

Давайте посмотрим на первые несколько главных компонент (рис. 3.9):

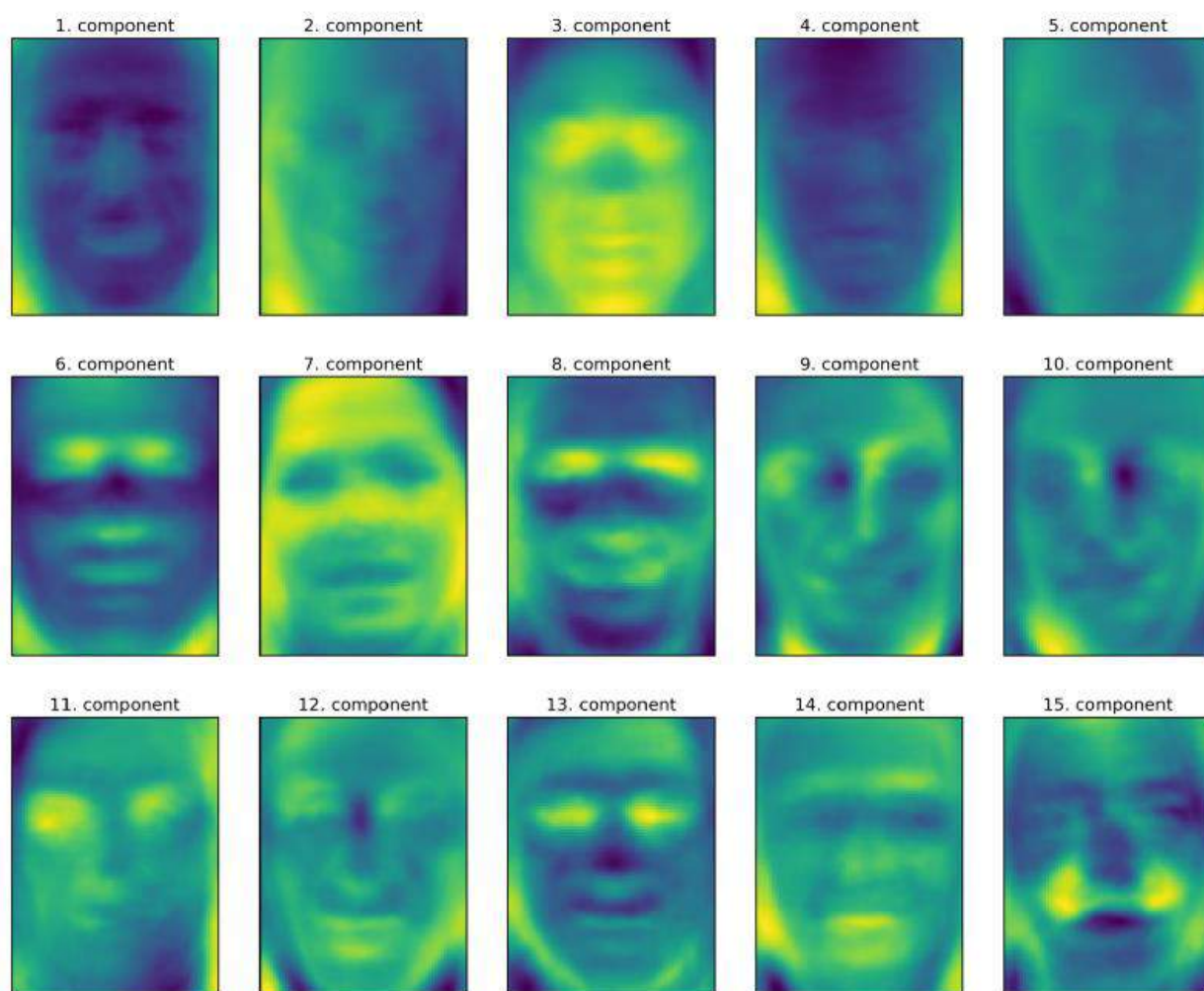
```
In[29]:
print("форма pca.components_: {}".format(pca.components_.shape))

Out[29]:
форма pca.components_: (100, 5655)

In[30]:
fig, axes = plt.subplots(3, 5, figsize=(15, 12),
                          subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
               cmap='viridis')
    ax.set_title("{} component".format(i + 1))
```

Несмотря на то что мы, конечно, не сможем понять весь содержательный смысл этих компонент, мы можем догадаться, какие характеристики изображений лиц были выделены некоторыми компонентами. Похоже, что первая компонента главным образом кодирует контраст между лицом и фоном, а вторая компонента кодирует различия в освещенности между правой и левой половинами лица и т.д. Хотя это представление данных в отличие от исходных значений пикселей немного содержательнее, оно по-прежнему весьма далеко от того, как человек привык воспринимать лицо. Поскольку модель РСА основана на пикселях, выравнивание изображения лица (положения глаз, подбородка и носа) и освещенность оказывают сильное влияние на степень сходства двух пиксельных изображений. Однако выравнивание и освещенность, вероятно, будут совсем не теми характеристиками, которые человек будет воспринимать в первую очередь. Когда людей просят оценить сходство между лицами, они в большей степени руководствуются такими признаками, как возраст, пол, выражение лица и прическа, то есть признаками, которые трудно выделить, исходя из интенсивностей пикселей. Важно помнить, что, как правило, алгоритмы в отличие от человека интерпретируют данные (в частности, визуальные данные, например, изображения популярных людей) совершенно по-другому.





**Рис. 3.9** Собственные векторы первых 15 компонент для набора лиц

Впрочем, давайте вернемся к конкретному случаю использования PCA. Мы кратко рассказали о преобразовании PCA как способе поворота данных с последующим удалением компонент, имеющих низкую дисперсию. Еще одна полезная интерпретация заключается в том, чтобы попытаться вычислить значения новых признаков, полученные после поворота PCA, таким образом, мы можем записать тестовые точки в виде взвешенной суммы главных компонент (см. рис. 3.10).

$$\text{Image} \approx X_0 * \text{Comp}_1 + X_1 * \text{Comp}_2 + X_2 * \text{Comp}_3 + X_3 * \text{Comp}_4 + \dots$$

**Рис. 3.10** Схематическое изображение PCA, осуществляющего разложение изображения на взвешенную сумму компонент



Здесь  $x_1$ ,  $x_2$  и т.д. являются коэффициентами главных компонент для конкретной точки данных, другими словами, они представляют собой изображение в новом пространстве, полученном в результате вращения.

Еще один способ понять, что делает модель PCA – реконструировать исходные данные, используя лишь некоторые компоненты. На третьем графике рис. 3.3 мы удалили вторую компоненту, затем мы отменили вращение и добавили обратно среднее значение, чтобы получить новые точки в исходном пространстве с удаленной второй компонентой, как показано на последнем графике рис. 3.3. Мы можем выполнить аналогичное преобразование для лиц, сократив данные за счет использования лишь некоторых главных компонент и вернувшись затем в исходное пространство. Это возвращение в пространство исходных признаков можно выполнить с помощью метода `inverse_transform`. Здесь мы визуализируем результаты реконструкции некоторых лиц, используя 10, 50, 100, 500 и 2000 компонент (рис. 3.11):

```
In[32]:  
mglearn.plots.plot_pca_faces(X_train, X_test, image_shape)
```

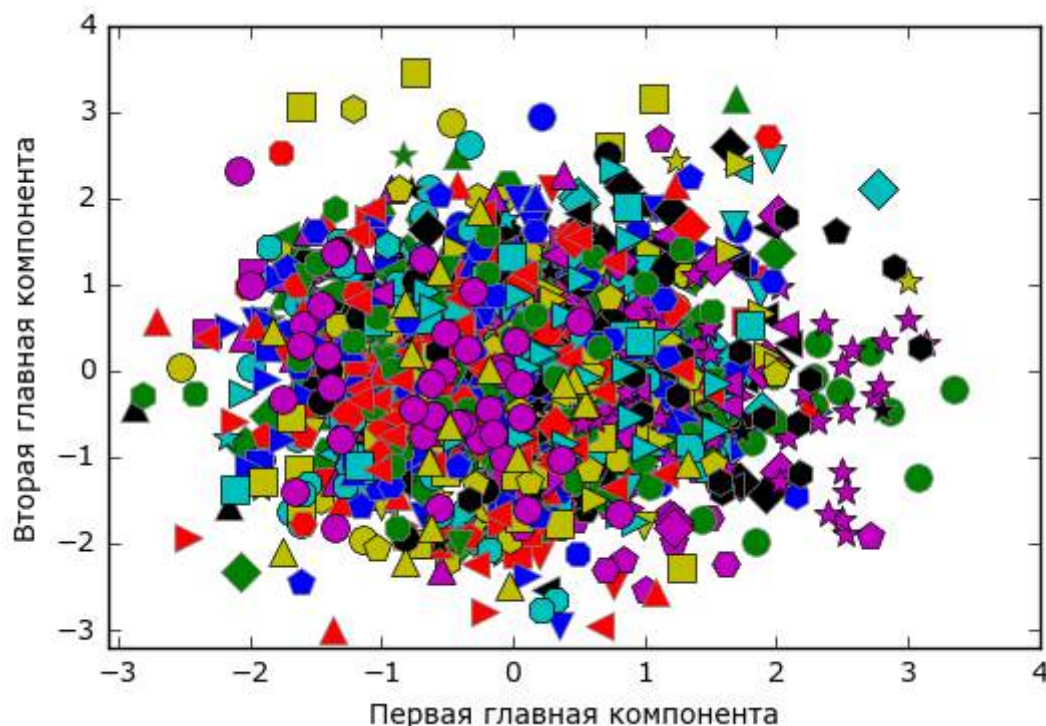


**Рис. 3.11** Реконструкция трех изображений лица с помощью постепенного увеличения числа главных компонент

Вы можете увидеть, что, когда мы используем лишь первые 10 главных компонент, фиксируется лишь общая суть картинки, например, ориентация лица и освещенность. По мере увеличения количества используемых компонент сохраняется все больше деталей изображения. Это соответствует включению большего числа слагаемых в сумму, показанную на рис. 3.10. Использование числа компонент, равного числу имеющихся пикселей, означало бы, что мы, осуществив поворот, сохранили всю информацию и можем идеально реконструировать изображение.

Кроме того, мы можем применить PCA для визуализации всех лиц набора на диаграмме рассеяния, воспользовавшись первыми двумя главными компонентами (рис. 3.12). Для этого мы выделим классы, соответствующие лицам, с помощью определенного цвета и формы (аналогично тому, что делали для набора данных `cancer`):

```
In[33]:
mglearn.discrete_scatter(X_train_pca[:, 0], X_train_pca[:, 1], y_train)
plt.xlabel("Первая главная компонента")
plt.ylabel("Вторая главная компонента")
```



**Рис. 3.12** Диаграмма рассеяния для набора лиц, использующая первые две главные компоненты (см. рис. 3.5 с соответствующим изображением для набора данных `cancer`)

Из рис. видно, когда мы используем лишь первые две главные компоненты, все данные представляют собой просто одно большое скопление данных без видимого разделения классов. Данный факт не удивителен, учитывая, что даже при использовании 10 компонент, как

уже было показано ранее на рис. 3.11, PCA фиксирует самые общие характеристики лиц.

### Факторизация неотрицательных матриц (NMF)

Факторизация неотрицательных матриц – еще один алгоритм машинного обучения без учителя, цель которого – выделить полезные характеристики. Он работает так же, как PCA, а также его можно использовать для уменьшения размерности. Как и в PCA, мы пытаемся записать каждую точку данных в виде взвешенной суммы некоторых компонентов, как показано на рис. 3.10. Однако, если в PCA нам нужно получить ортогональные компоненты, объясняющие максимально возможную долю дисперсии данных, то в NMF нам нужно получить неотрицательные компоненты и коэффициенты, то есть нам нужны компоненты и коэффициенты, которые больше или равны нулю. Поэтому этот метод может быть применен только к тем данным, в которых характеристики имеют неотрицательные значения, поскольку неотрицательная сумма неотрицательных компонентов не может быть отрицательной.

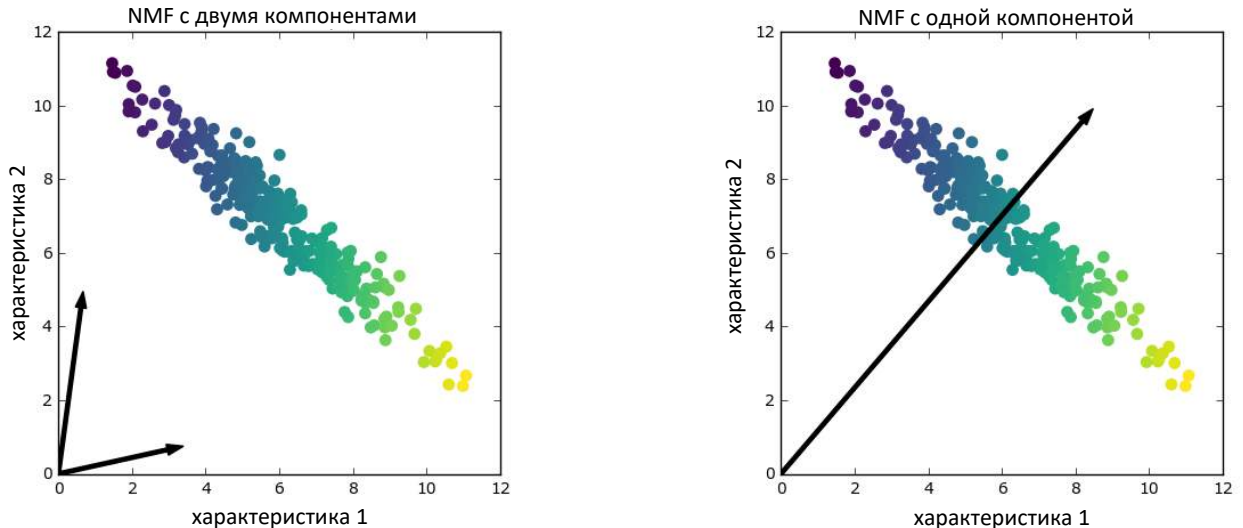
Процесс разложения данных на неотрицательную взвешенную сумму особенно полезен для данных, созданных в результате объединения (или наложения) нескольких независимых источников, например, аудиотреков с голосами нескольких людей, музыки с большим количеством инструментов. В таких ситуациях NMF может найти исходные компоненты, которые лежат в основе объединенных данных. В целом NMF позволяет получить более интерпретабельные компоненты, чем PCA, поскольку отрицательные компоненты и коэффициенты могут привести к получению трудных для интерпретации взаимокомпенсирующих эффектов. Например, собственные лица на рис. 3.9, содержат как положительные, так и отрицательные характеристики, и, как мы уже упоминали в описании PCA, знаки имеют фактически произвольный характер. Перед тем, как применить NMF к набору лиц, давайте заново посмотрим на наши синтетические данные.

#### Применение NMF к синтетическим данным

В отличие от PCA, чтобы применить NMF к данным, мы должны убедиться, что они имеют положительные значения. Это означает, что для NMF расположение данных относительно начала координат  $(0, 0)$  имеет реальное значение. Поэтому извлекаемые неотрицательные компоненты можно представить в виде направлений, выходящих из начала координат  $(0, 0)$  к данным.

Следующий пример (рис. 3-13) показывает результаты применения NMF к двумерным синтетическим данным:

```
In[34]:  
mglearn.plots.plot_nmf_illustration()
```



**Рис. 3.13** Компоненты, найденные в результате факторизации неотрицательных матриц с двумя компонентами (слева) и одной компонентой (справа)

Для NMF с двумя компонентами (график слева) ясно, что все точки данных можно записать в виде комбинации положительных значений этих двух компонент. Если количества компонент достаточно для того, чтобы полностью реконструировать данные (количество компонент совпадает с количеством характеристик), алгоритм будет выбирать направления, указывающие на экстремальные значения данных.

При использовании лишь одной компоненты NMF выделяет компоненту, которая указывает на среднее значение как значение, лучше всего объясняющее данные. Видно, что в отличие от PCA уменьшение числа компонент удаляет не только некоторые направления, но и создает совершенно другой набор компонент! Кроме того, компоненты NMF не упорядочены каким-либо определенным образом, поэтому здесь нет такого понятия, как «первая неотрицательная компонента»: все компоненты играют одинаковую роль.

NMF использует случайную инициализацию, поэтому разные стартовые значения дают различные результаты. В относительно простых случаях (например, синтетические данные с двумя компонентами), где все данные можно прекрасно объяснить, случайность мало влияет на результат (хотя она может изменить порядок или масштаб компонент). В более сложных ситуациях использование

различных случайных значений может привести к радикальным изменениям.

### Применение NMF к изображениям лиц

Теперь давайте применим NMF к набору данных Labeled Faces in the Wild, который мы использовали ранее. Основной параметр NMF – количество извлекаемых компонент. Как правило, количество извлекаемых компонент меньше количества входных характеристик (в противном случае, данные можно объяснить, представив каждый пиксель отдельной компонентой).

Во-первых, давайте выясним, как количество компонент влияет на качество восстановления данных с помощью NMF (рис. 3.14):

In[35]:

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```



**Рис. 3.14** Реконструкция трех изображений лица с помощью постепенного увеличения числа компонент



Качество обратно преобразованных данных аналогично качеству, полученному с помощью PCA, но немного хуже. Это вполне ожидаемо, поскольку PCA находит оптимальные направления с точки зрения реконструкции данных. NMF же, как правило, используется не из-за своей способности реконструировать или представлять данные, а скорее из-за того, что позволяет находить интересные закономерности в данных.

Для начала давайте попробуем извлечь лишь несколько компонент (скажем, 15). Рис. 3.15 показывает результат:

```
In[36]:
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fig, axes = plt.subplots(3, 5, figsize=(15, 12),
                        subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{} component".format(i))
```



**Рис. 3.15** Компоненты, найденные NMF для набора лиц (использовалось 15 компонент)

Все эти компоненты являются положительными и поэтому похожи на прототипы лиц гораздо больше, чем компоненты РСА, показанные на рис. 3.9. Например, четко видно, что компонента 3 показывает лицо, немного повернутое вправо, тогда как компонента 7 показывает лицо, немного повернутое влево. Давайте посмотрим на изображения, для которых эти компоненты имеют наибольшие значения (показаны на рис. 3.16 и 3.17):

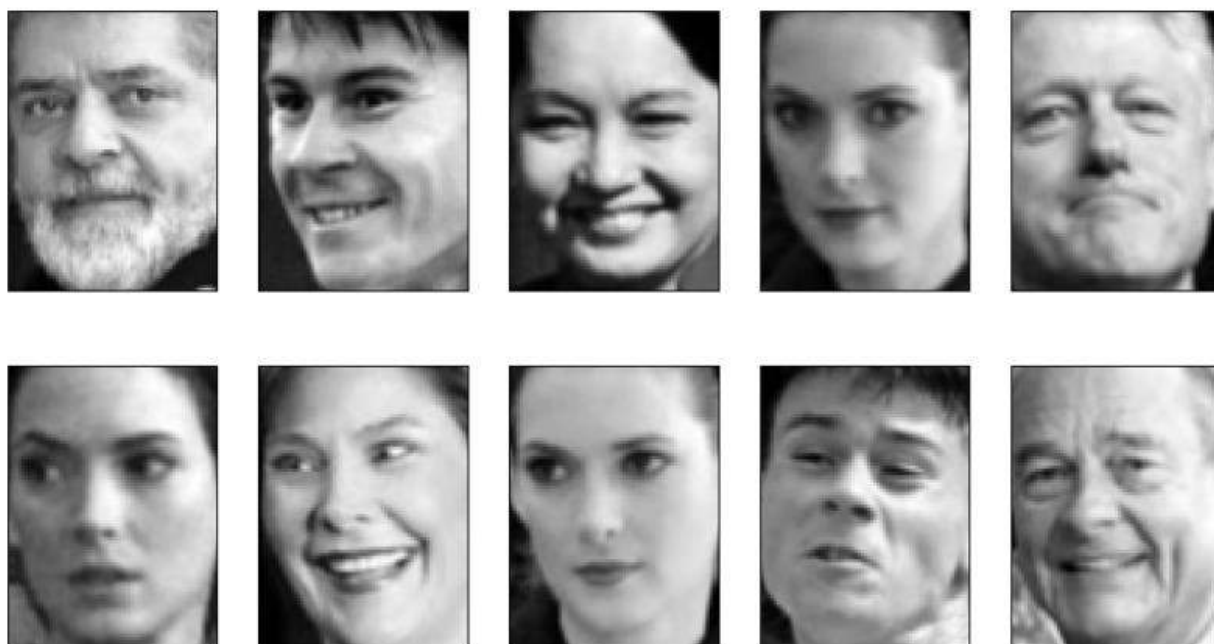
```
In[37]:
compn = 3
# сортируем по 3-й компоненте, выводим первые 10 изображений
inds = np.argsort(X_train_nmf[:, compn])[:, :-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                        subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
# сортируем по 7-й компоненте, выводим первые 10 изображений
inds = np.argsort(X_train_nmf[:, compn])[:, :-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                        subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```



**Рис. 3.16** Лица с большим коэффициентом компоненты 3



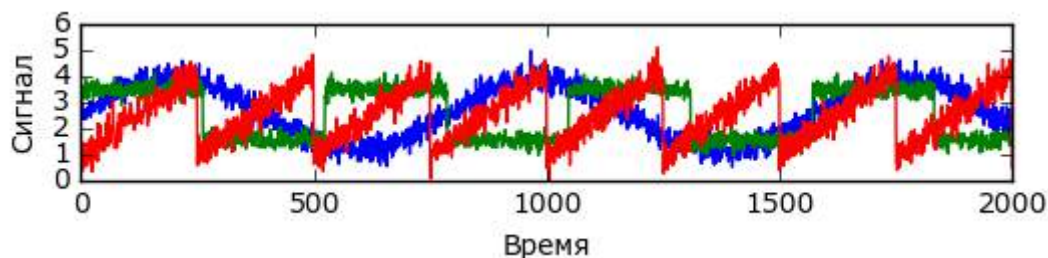


**Рис. 3.17** Лица с большим коэффициентом компоненты 7

Как и следовало ожидать, лица с высоким коэффициентом компоненты 3 – это лица, смотрящие вправо (рис. 3.16), тогда как лица с высоким коэффициентом компоненты 7 смотрят влево (рис. 3.17). Как уже упоминалось ранее, выделение паттернов, аналогичных рассматриваемым изображениям, лучше всего работает в отношении данных с аддитивной структурой, включая аудиоданные, данные экспрессии генов и текстовые данные. Давайте рассмотрим еще один пример на основе синтетических данных, чтобы увидеть, как это будет выглядеть.

Допустим, нас интересует сигнал, который представляет собой комбинацию трех различных источников (рис. 3.18):

```
In[38]:
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Время")
plt.ylabel("Сигнал")
```



**Рис. 3.18** Исходные источники сигнала

К сожалению, мы не можем наблюдать исходные сигналы, лишь аддитивную смесь (сумму) всех трех сигналов. Необходимо восстановить

исходные компоненты из этой смеси. Предположим, у нас есть различные способы фиксировать характеристики этого смешанного сигнала (скажем, у нас есть 100 измерительных приборов), каждый из которых дает нам серию измерений:

```
In[39]:
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Форма измерений: {}".format(X.shape))

Out[39]:
Форма измерений: (2000, 100)
```

Мы можем использовать NMF, чтобы восстановить три сигнала:

```
In[40]:
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Форма восстановленного сигнала: {}".format(S_.shape))

Out[40]:
Форма восстановленного сигнала: (2000, 3)
```

Для сравнения мы еще применим PCA:

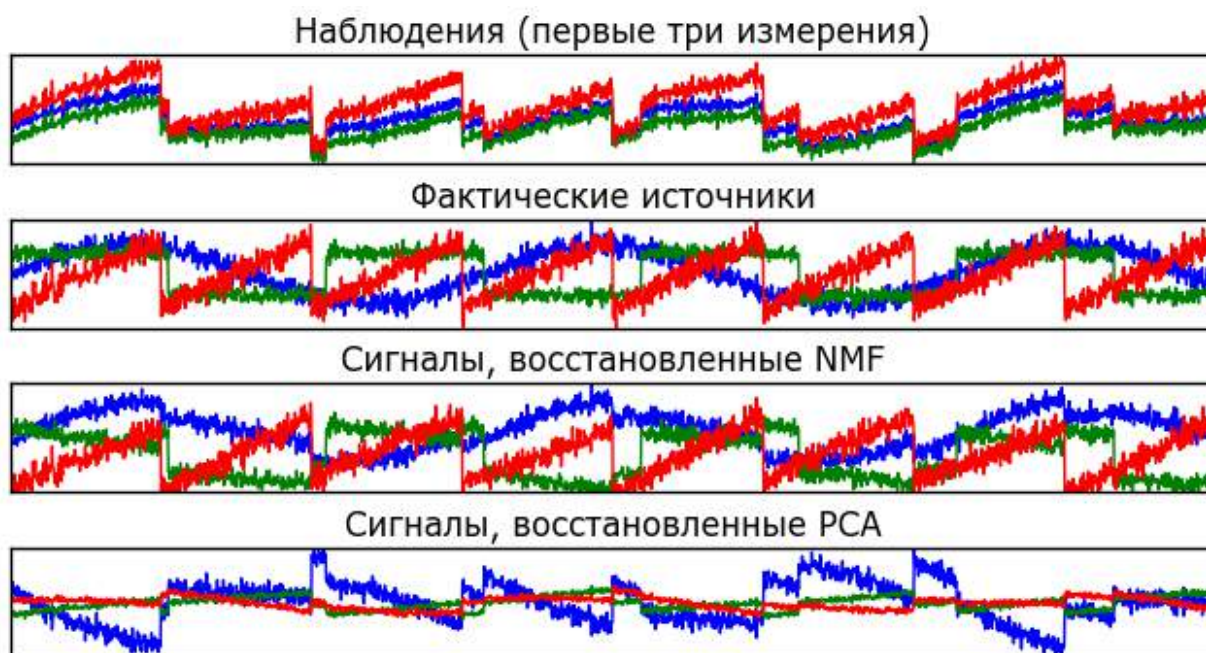
```
In[41]:
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

Рис. 3.19 показывает активность сигнала, обнаруженную с помощью NMF и PCA:

```
In[42]:
models = [X, S, S_, H]
names = ['Наблюдения (первые три измерения)',
         'Фактические источники',
         'Сигналы, восстановленные NMF',
         'Сигналы, восстановленные PCA']

fig, axes = plt.subplots(4, figsize=(8, 4), gridspec_kw={'hspace': .5},
                        subplot_kw={'xticks': (), 'yticks': ()})

for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```



**Рис. 3.19** Восстановление первоначальных источников с помощью NMF и PCA

Этот график включает в себя наблюдения по первым 3 измерениям  $X$ . Как вы можете увидеть, NMF довольно хорошо выделил первоначальные источники, тогда как PCA потерпел неудачу и использовал первую компоненту, чтобы объяснить большую часть дисперсии данных. Помните о том, что компоненты, полученные с помощью NMF, не упорядочены. В этом примере порядок компонент NMF точно такой же, как в исходном сигнале (см. цвет трех кривых), но это носит чисто случайный характер.

Существует множество других алгоритмов, которые можно использовать для разложения каждой точки данных на взвешенную сумму компонент, как это делают PCA и NMF. Обсуждение всех этих алгоритмов выходит за рамки этой книги, а описание ограничений, накладываемых на компоненты и коэффициенты, часто предполагает знание теории вероятностей. Если вас заинтересовал тот или иной алгоритм выделения паттернов, мы рекомендуем вам изучить разделы руководства `scikit-learn`, посвященные *анализу независимых компонент* (*independent component analysis, ICA*), *факторному анализу* (*factor analysis, FA*) и *разреженному кодированию* (*sparse coding*) с *обучением словаря* (*dictionary learning*). Информацию обо всех этих методах можно найти на странице, посвященной [ДЕКОМПОЗИЦИОННЫМ МЕТОДАМ](#).

## Множественное обучение с помощью алгоритма t-SNE

Хотя PCA часто выступает в качестве приоритетного метода, преобразующего данные таким образом, что можно визуализировать их с помощью диаграммы рассеяния, сам характер метода (вращение данных, а затем удаление направлений, объясняющих незначительную дисперсию данных) ограничивает его полезность, как мы уже убедились на примере диаграммы рассеяния для набора данных *Labeled Faces in the Wild*. Существует класс алгоритмов визуализации, называемых *алгоритмами множественного обучения* (*manifold learning algorithms*), которые используют гораздо более сложные графические представления данных и позволяют получить визуализации лучшего качества. Особенно полезным является алгоритм t-SNE.

Алгоритмы множественного обучения в основном направлены на визуализацию и поэтому редко используются для получения более двух новых характеристик. Некоторые из них, в том числе t-SNE, создают новое представление обучающих данных, но при этом не осуществляют преобразования новых данных. Это означает, что данные алгоритмы нельзя применить к тестовому набору, они могут преобразовать лишь те данные, на которых они были обучены. Множественное обучение может использоваться для разведочного анализа данных, но редко используется в тех случаях, когда конечной целью является применение модели машинного обучения с учителем. Идея, лежащая в основе алгоритма t-SNE, заключается в том, чтобы найти двумерное представление данных, сохраняющее расстояния между точками наилучшим образом. t-SNE начинает свою работу со случайного двумерного представления каждой точки данных, а затем пытается сблизить точки, которые в пространстве исходных признаков находятся близко друг к другу, и отдаляет друг от друга точки, которые находятся далеко друг от друга. При этом t-SNE уделяет большее внимание сохранению расстояний между точками, близко расположенными друг к другу. Иными словами, он пытается сохранить информацию, указывающую на то, какие точки являются соседями друг другу.

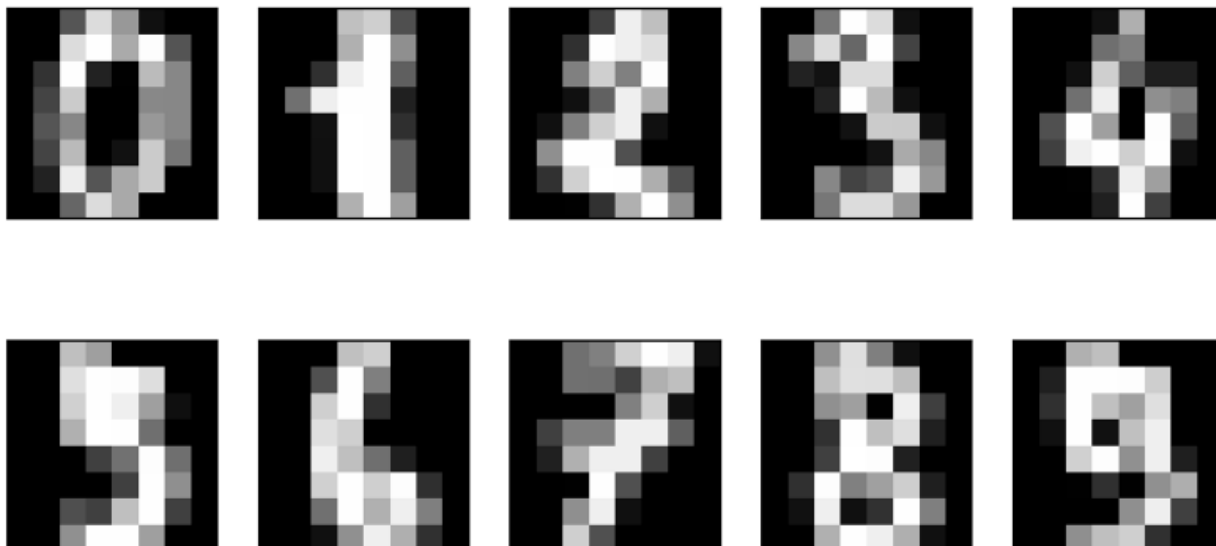
Мы применим алгоритм множественного обучения t-SNE к набору данных рукописных цифр, который включен в `scikit-learn`.<sup>24</sup> Каждая точка данных в этом наборе является изображением цифры в градациях серого. Рис. 3.20 показывает примеры изображений для каждого класса:

---

<sup>24</sup> Не следует путать с гораздо большим набором данных MNIST.

```
In[43]:
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                        subplot_kw={'xticks': (), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```



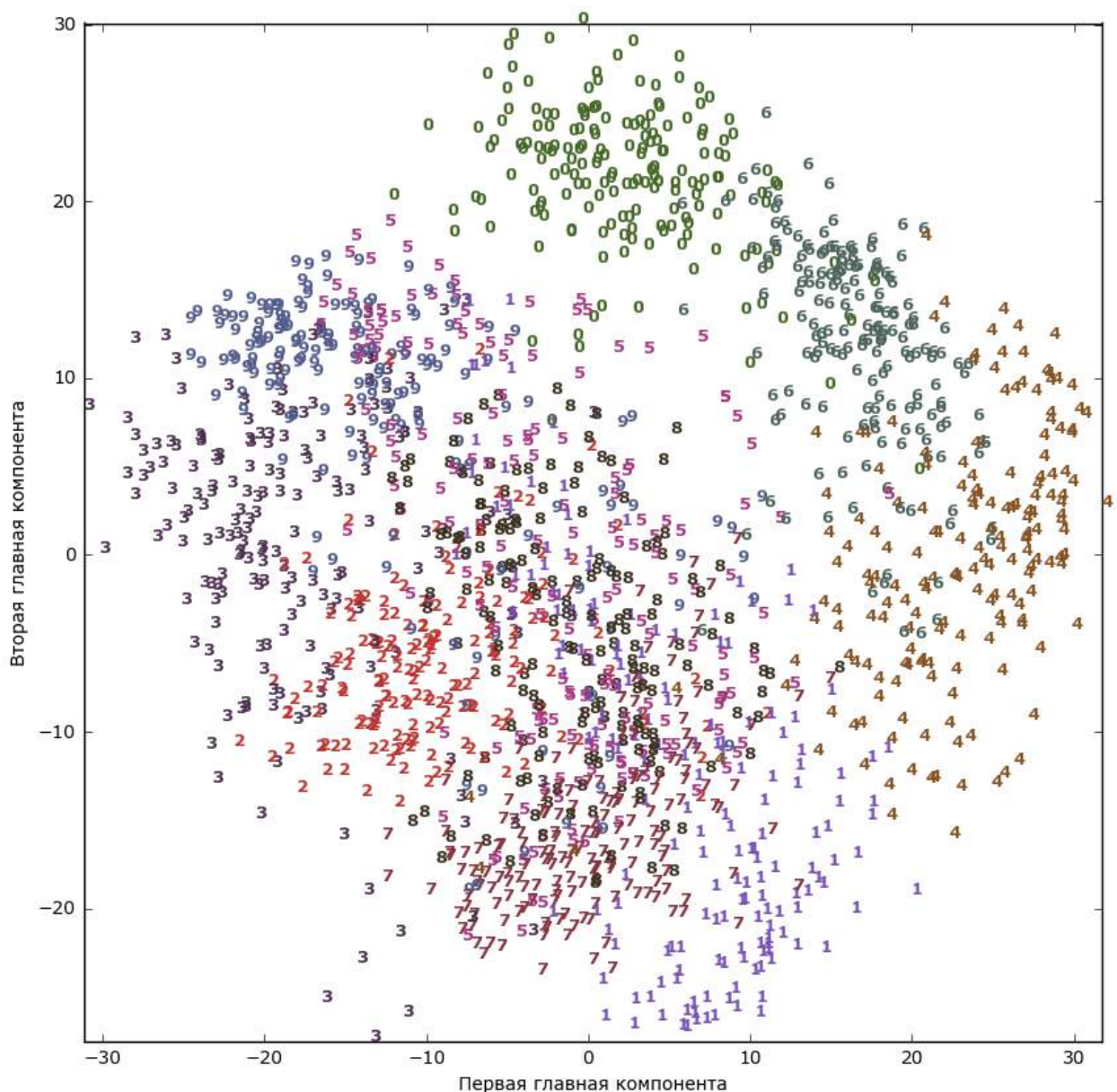
**Рис. 3.20** Примеры изображений из набора данных digits

Давайте используем PCA для визуализации данных, сведя их к двум измерениям. Мы построим график первых двух главных компонент и отметим цветом класс каждой точки (см. рис. 3-21):

```
In[44]:
# строим модель PCA
pca = PCA(n_components=2)
pca.fit(digits.data)
# преобразуем данные рукописных цифр к первым двум компонентам
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # строим график, где цифры представлены символами вместо точек
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
            color = colors[digits.target[i]],
            fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("Первая главная компонента")
plt.ylabel("Вторая главная компонента")
```

Здесь мы вывели фактические классы цифр в виде символов, чтобы визуально показать расположение каждого класса. Цифры 0, 6 и 4 относительно хорошо разделены с помощью первых двух главных компонент, хотя по-прежнему перекрывают друг друга. Большинство остальных цифр значительно перекрывают друг друга.





**Рис. 3.21** Диаграмма рассеяния для набора данных `digits`, использующая первые две главные компоненты

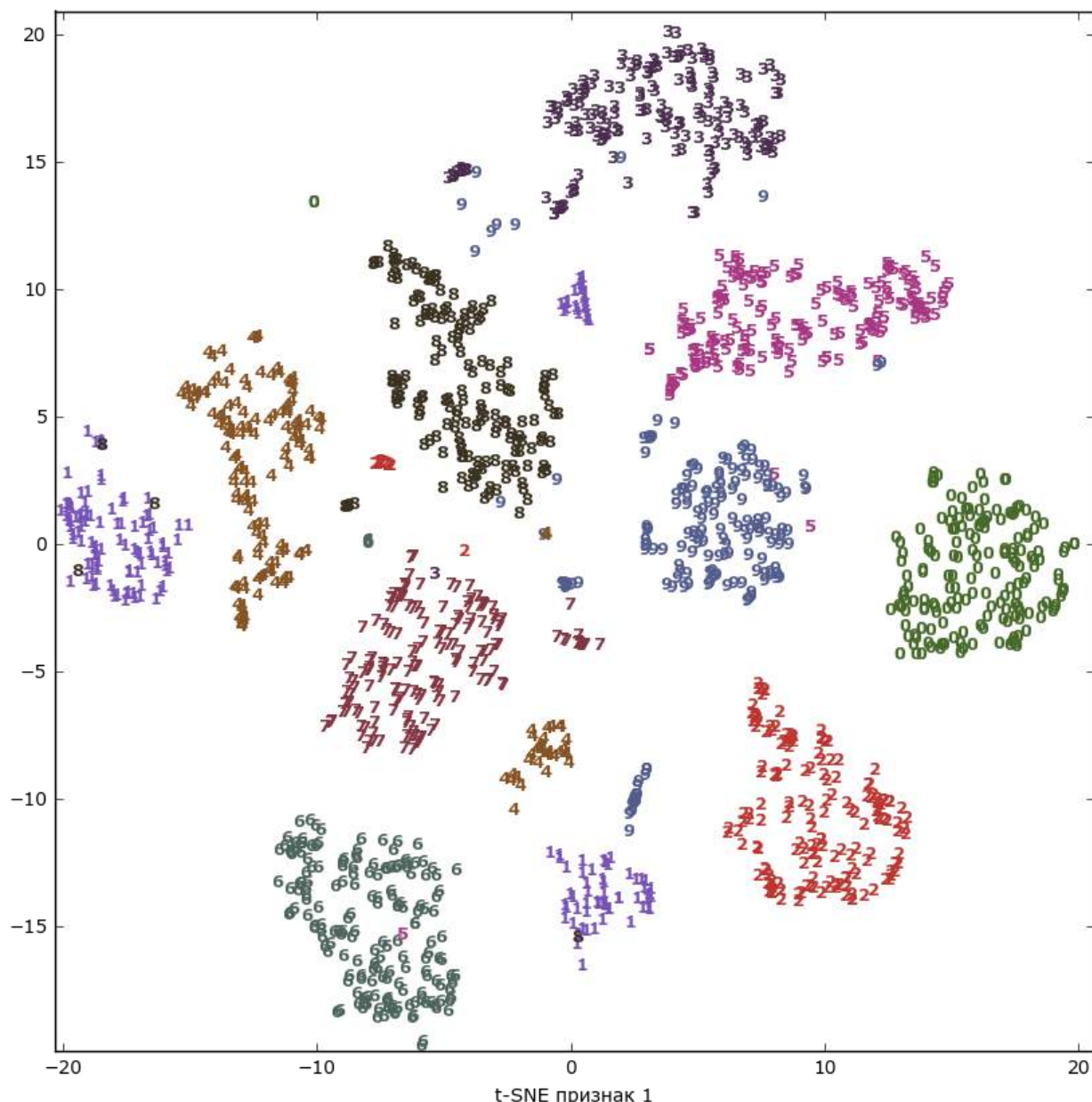
Давайте применим t-SNE к этому же набору данных и сравним результаты. Поскольку t-SNE не поддерживает преобразование новых данных, в классе `TSNE` нет метода `transform`. Вместо этого мы можем вызвать метод `fit_transform`, который построит модель и немедленно вернет преобразованные данные (см. рис. 3.22):

```
In[45]:
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# используем метод fit_transform вместо fit, т.к. класс TSNE не использует метод transform
digits_tsne = tsne.fit_transform(digits.data)
```

```

In[46]:
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # строим график, где цифры представлены символами вместо точек
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
            color = colors[digits.target[i]],
            fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE признак 0")
plt.ylabel("t-SNE признак 1")

```



**Рис. 3.22** Диаграмма рассеяния для набора данных digits, которая использует первые две главные компоненты, найденные с помощью t-SNE

Результат, полученный с помощью t-SNE, весьма примечателен. Все классы довольно четко разделены. Единицы и девятки в некоторой степени распались, однако большинство классов образуют отдельные сплоченные группы. Имейте в виду, что этот метод не использует



информацию о метках классов: он является полностью неконтролируемым. Тем не менее он может найти двумерное представление данных, которое четко разграничивает классы, используя лишь информацию о расстояниях между точками данных в исходном пространстве.

Алгоритм t-SNE имеет некоторые настраиваемые параметры, хотя, как правило, дает хорошее качество, когда используются настройки по умолчанию. Вы можете поэкспериментировать с параметрами `perplexity` и `early_exaggeration`, но эффекты от их применения обычно незначительны.

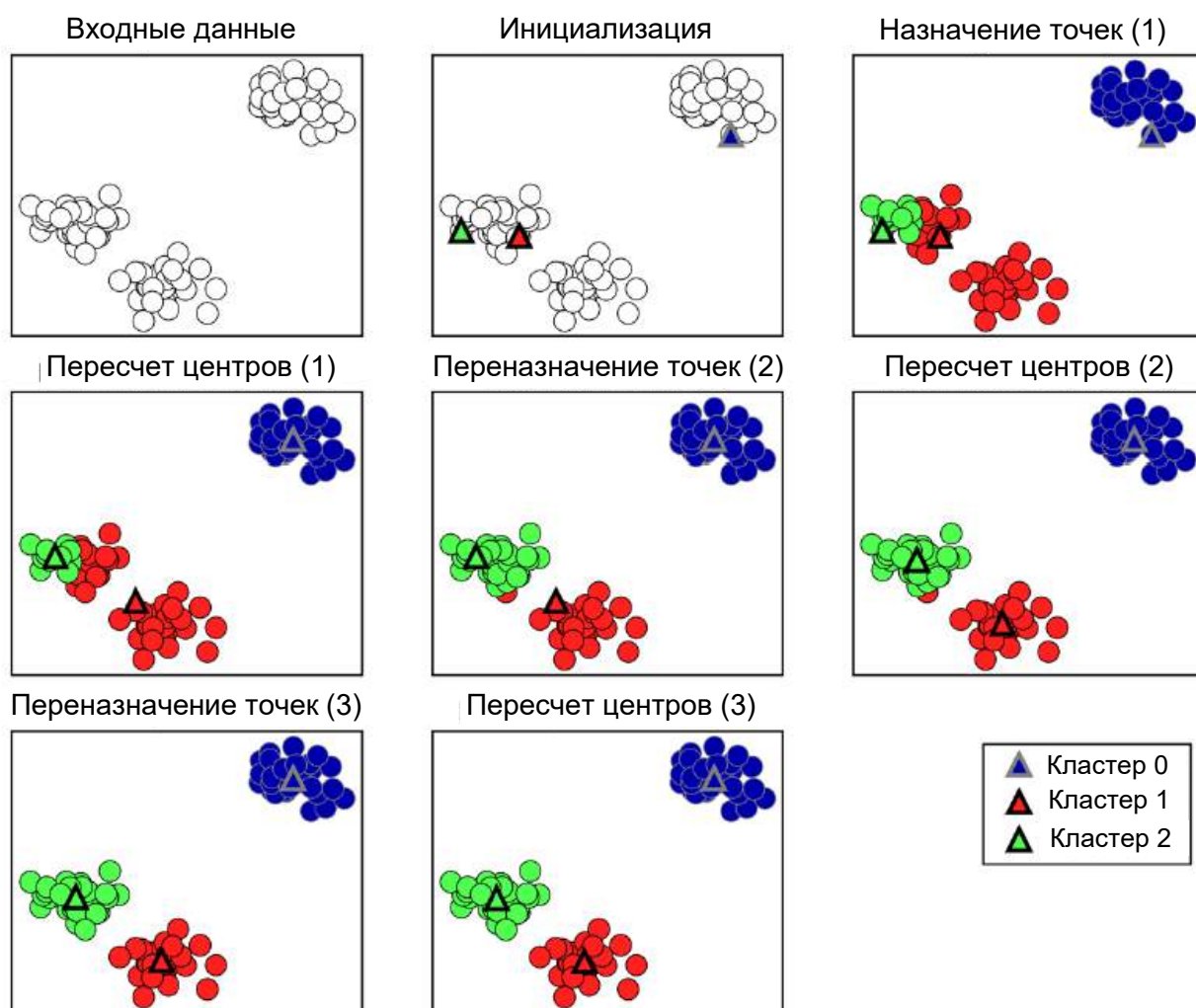
## Кластеризация

Как мы уже говорили выше, *кластеризация* (*clustering*) является задачей разбиения набора данных на группы, называемые кластерами. Цель – разделить данные таким образом, чтобы точки, находящиеся в одном и том же кластере, были очень схожи друг с другом, а точки, находящиеся в разных кластерах, отличались друг от друга. Как и алгоритмы классификации, алгоритмы кластеризации присваивают (или прогнозируют) каждой точке данных номер кластера, которому она принадлежит.

### Кластеризация $k$ -средних

Кластеризация  $k$ -средних – один из самых простых и наиболее часто используемых алгоритмов кластеризации. Сначала выбирается число кластеров  $k$ . После выбора значения  $k$  алгоритм  $k$ -средних отбирает точки, которые будут представлять *центры кластеров* (*cluster centers*). Затем для каждой точки данных вычисляется его евклидово расстояние до каждого центра кластера. Каждая точка назначается ближайшему центру кластера. Алгоритм вычисляет *центроиды* (*centroids*) – центры тяжести кластеров. Каждый центроид – это вектор, элементы которого представляют собой средние значения характеристик, вычисленные по всем точкам кластера. Центр кластера смещается в его центроид. Точки заново назначаются ближайшему центру кластера. Этапы изменения центров кластеров и переназначения точек итеративно повторяются до тех пор, пока границы кластеров и расположение центроидов не перестанут изменяться, т.е. на каждой итерации в каждый кластер будут попадать одни и те же точки данных. Следующий пример (рис. 3.23) иллюстрирует работу алгоритма на синтетическом наборе данных

```
In[47]:  
mglearn.plots.plot_kmeans_algorithm()
```

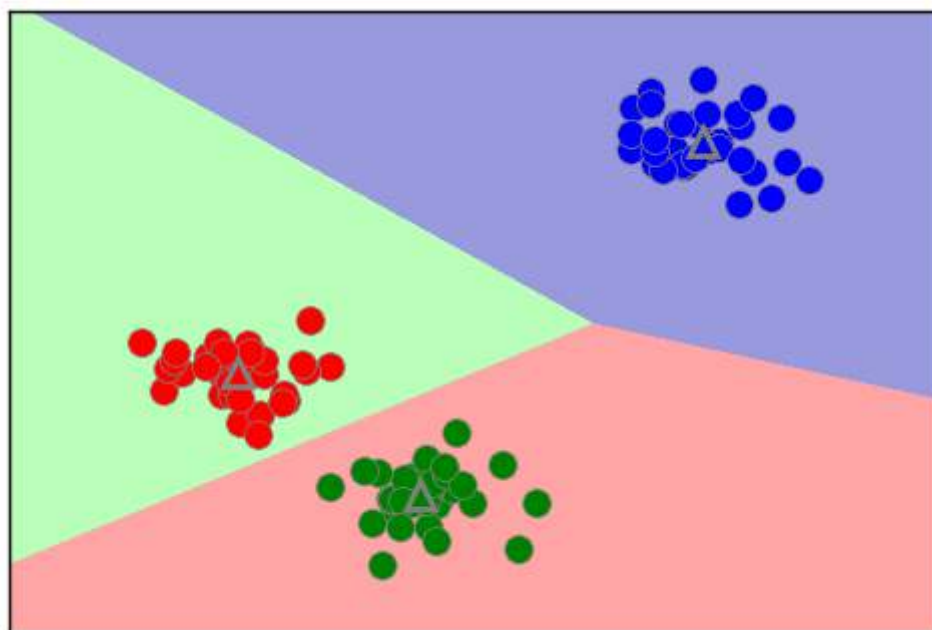


**Рис. 3.23** Исходные данные и этапы алгоритма *k*-средних

Центры кластеров представлены в виде треугольников, в то время как точки данных отображаются в виде окружностей. Цвета указывают принадлежность к кластеру. Мы указали, что ищем три кластера, поэтому алгоритм был инициализирован с помощью случайного выбора трех точек данных в качестве центров кластеров (см. «Инициализация»). Затем запускается итерационный алгоритм. Во-первых, каждая точка данных назначается ближайшему центру кластера (см. «Назначение точек (1)»). Затем центры кластеров переносятся в центры тяжести кластеров (см. «Пересчет центров (1)»). Затем процесс повторяется еще два раза. После третьей итерации принадлежность точек кластерным центрам не изменилась, поэтому алгоритм останавливается.

Получив новые точки данных, алгоритм *k*-средних будет присваивать каждую точку данных ближайшему центру кластера. Следующий пример (рис. 3.24) показывает границы центров кластеров, процесс вычисления которых был приведен на рис. 3.23:

```
In[48]:  
mglearn.plots.plot_kmeans_boundaries()
```



**Рис. 3.24** Центры кластеров и границы кластеров, найденные с помощью алгоритма  $k$ -средних

Применить алгоритм  $k$ -средних, воспользовавшись библиотекой `skikit-learn`, довольно просто. Здесь мы применяем его к синтетическим данным, которые использовали для построения предыдущих графиков. Мы создаем экземпляр класса `KMeans` и задаем количество выделяемых кластеров.<sup>25</sup> Затем мы вызываем метод `fit` и передаем ему в качестве аргумента данные:

```
In[49]:  
from sklearn.datasets import make_blobs  
from sklearn.cluster import KMeans  
  
# генерируем синтетические двумерные данные  
X, y = make_blobs(random_state=1)  
  
# строим модель кластеризации  
kmeans = KMeans(n_clusters=3)  
kmeans.fit(X)
```

Во время работы алгоритма каждой точке обучающих данных `X` присваивается метка кластера. Вы можете найти эти метки в атрибуте `kmeans.labels_`:

---

<sup>25</sup> Если вы не зададите количество выделяемых кластеров, то значение `n_clusters` по умолчанию будет равно 8. При этом нет никаких конкретных причин, в силу которых вы должны использовать именно это значение.

```
In[50]:
print("Принадлежность к кластерам:\n{}".format(kmeans.labels_))
```

```
Out[50]:
Принадлежность к кластерам:
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

Поскольку мы задали три кластера, кластеры пронумерованы от 0 до 2.

Кроме того, вы можете присвоить метки кластеров новым точкам с помощью метода `predict`. В ходе прогнозирования каждая новая точка назначается ближайшему центру кластера, но существующая модель не меняется. Запуск метода `predict` на обучающем наборе возвращает тот же самый результат, что содержится в атрибуте `labels_`:

```
In[51]:
print(kmeans.predict(X))
```

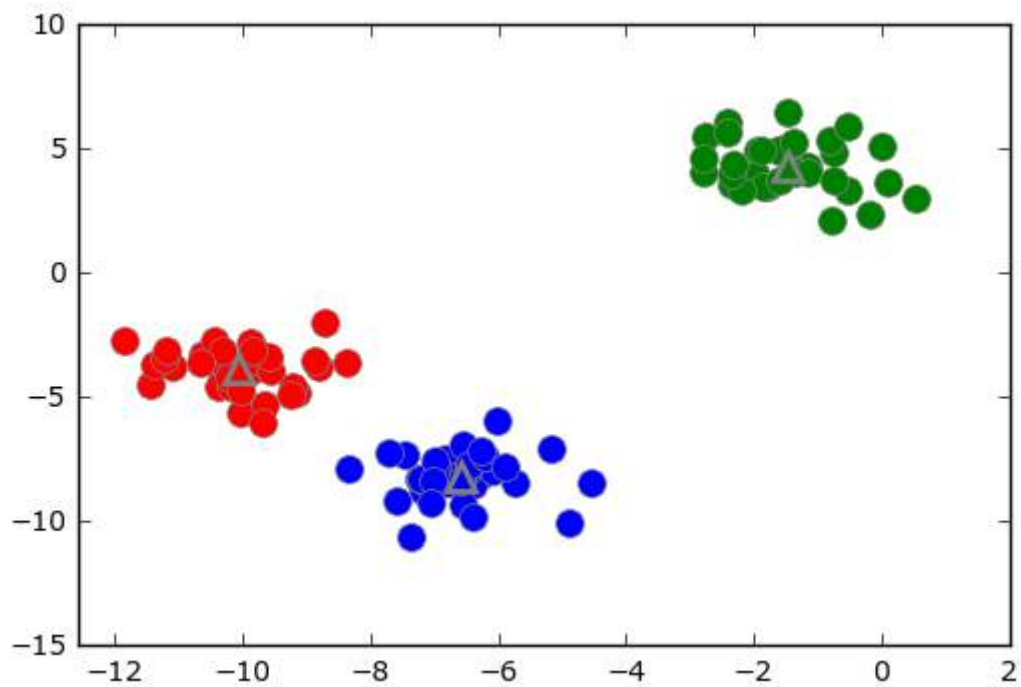
```
Out[51]:
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

Вы можете увидеть, что кластеризация немного похожа на классификацию в том плане, что каждый элемент получает метку. Однако нет никаких оснований утверждать, что данная метка является истинной и поэтому сами по себе метки не несут никакого априорного смысла. Давайте вернемся к примеру с кластеризацией изображений лиц, который мы обсуждали ранее. Возможно, что кластер 3, найденный с помощью алгоритма, содержит лишь лица вашего друга. Впрочем, вы можете узнать это только после того, как взгляните на фотографии, а само число 3 является произвольным. Единственная информация, которую дает вам алгоритм, – это то, что все лица, отнесенные к кластеру 3, схожи между собой.

В случае с кластеризацией, которую мы только что построили для двумерного синтетического набора данных, это означает, что мы не должны придавать значения тому факту, что одной группе был присвоен 0, а другой – 1. Повторный запуск алгоритма может привести к совершенно иной нумерации кластеров в силу случайного характера инициализации.

Ниже приводится новый график для тех же самых данных (рис. 3.25). Центры кластеров записаны в атрибуте `cluster_centers_` и мы наносим их на график в виде треугольников:

```
In[52]:
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(
    kmeans.cluster_centers_[ :, 0], kmeans.cluster_centers_[ :, 1], [0, 1, 2],
    markers='^', markeredgewidth=2)
```



**Рис. 3.25** Принадлежность к кластерам и центры кластеров, найденные с помощью алгоритма  $k$ -средних,  $k=3$

Кроме того, мы можем увеличить или уменьшить количество центров кластеров (рис. 3.26):

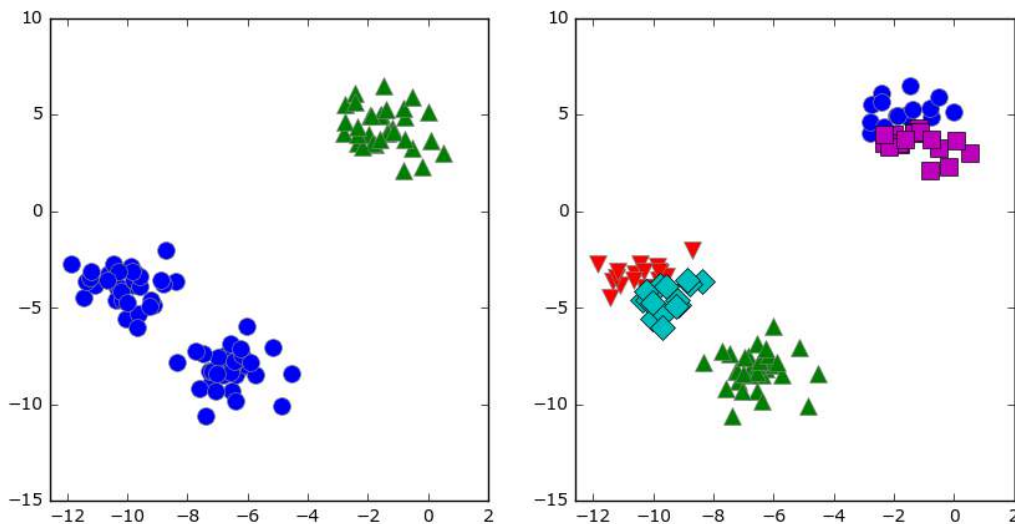
```
In[53]:
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# использование двух центров кластеров:
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

# использование пяти центров кластеров:
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```



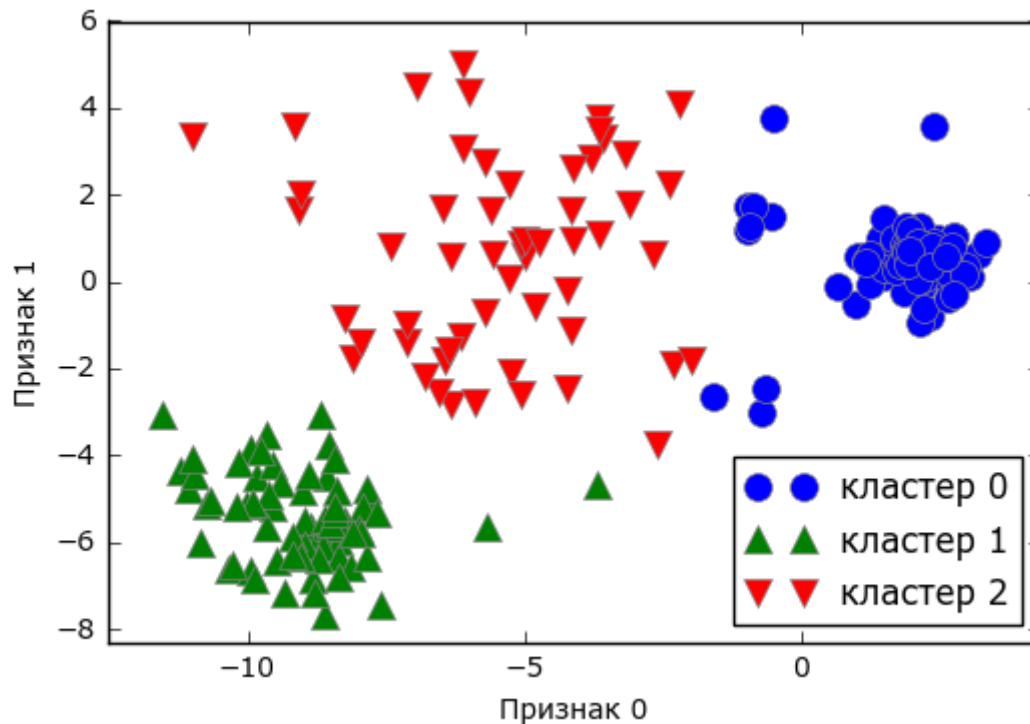
**Рис. 3.26** Принадлежность к кластерам, найденная с помощью алгоритма  $k$ -средних,  $k=3$  (слева) и  $k=5$  (справа)

### Недостатки алгоритма $k$ -средних

Даже если вы знаете «правильное» количество кластеров для конкретного набора данных, алгоритм  $k$ -средних не всегда может выделить их. Каждый кластер определяется исключительно его центром, это означает, что каждый кластер имеет выпуклую форму. В результате этого алгоритм  $k$ -средних может описать относительно простые формы. Кроме того, алгоритм  $k$ -средних предполагает, что все кластеры в определенном смысле имеют одинаковый «диаметр», он всегда проводит границу между кластерами так, чтобы она проходила точно посередине между центрами кластеров. Это иногда может привести к неожиданным результатам, как показано на рис. 3.27:

```
In[54]:
X_varied, y_varied = make_blobs(n_samples=200,
                                cluster_std=[1.0, 2.5, 0.5],
                                random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["кластер 0", "кластер 1", "кластер 2"], loc='best')
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```



**Рис. 3.27** Принадлежность к кластерам, найденная с помощью алгоритма  $k$ -средних, при этом кластеры имеют разные плотности

Можно было бы ожидать плотную область в нижнем левом углу, которая рассматривалась бы в качестве первого кластера, плотную область в верхнем правом углу в качестве второго кластера и менее плотную область в центре в качестве третьего кластера. Вместо этого, у кластера 0 и кластера 1 есть несколько точек, которые сильно удалены от всех остальных точек этих кластеров, «тянущихся» к центру.

Кроме того, алгоритм  $k$ -средних предполагает, что все направления одинаково важны для каждого кластера. Следующий график (рис. 3.28) показывает двумерный набор данных с тремя четко обособленными группами данных. Однако эти группы вытянуты по диагонали. Поскольку алгоритм  $k$ -средних учитывает лишь расстояние до ближайшего центра кластера, он не может обработать данные такого рода:

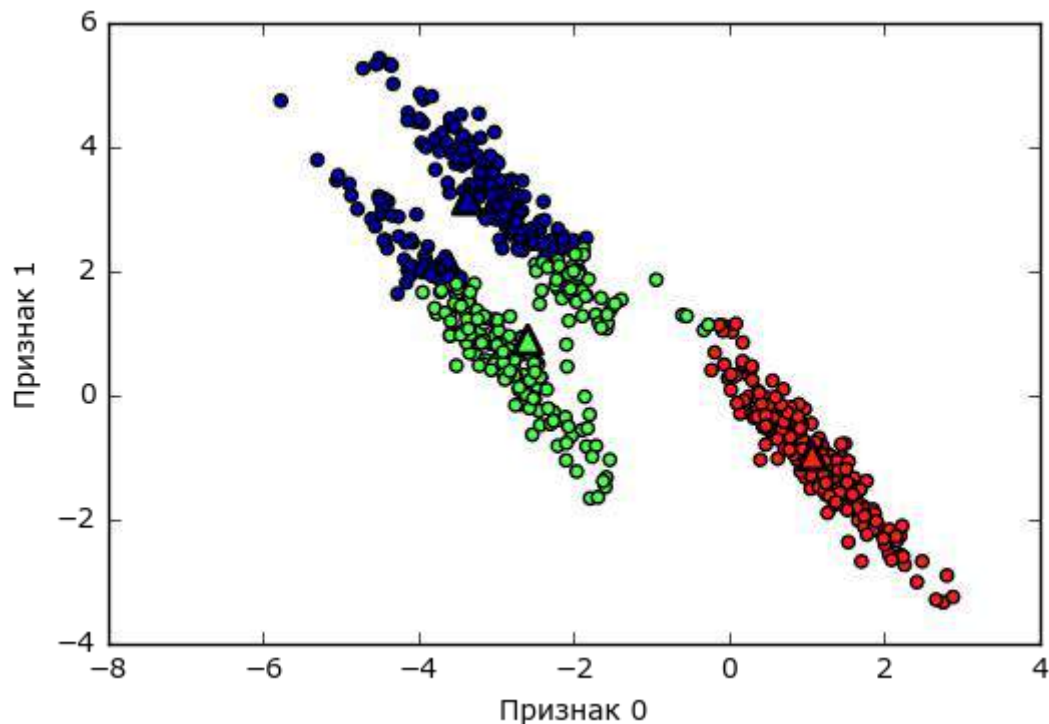
```
In[55]:
# генерируем случайным образом данные для кластеризации
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# преобразуем данные так, чтобы они были вытянуты по диагонали
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)

# группируем данные в три кластера
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)
```



```
# строим график принадлежности к кластерам и центров кластеров
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1],
            marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```



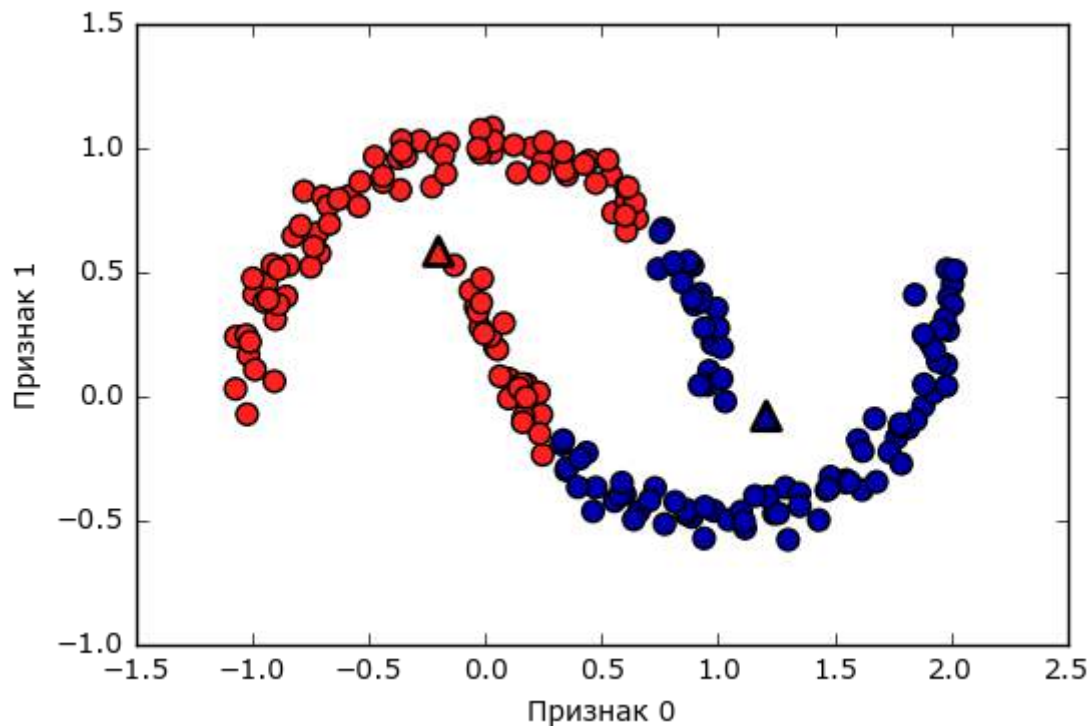
**Рис. 3.28** Алгоритм  $k$ -средних не позволяет выявить несферические кластеры

Кроме того, алгоритм  $k$ -средних плохо работает, когда кластеры имеют более сложную форму, как в случае с данными `two_moons`, с которыми мы столкнулись в главе 2 (см. рис. 3.29):

```
In[56]:
# генерируем синтетические данные two_moons (на этот раз с меньшим количеством шума)
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# группируем данные в два кластера
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# строим график принадлежности к кластерам и центров кластеров
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60)
plt.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1],
            marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```



**Рис. 3.29** Алгоритм  $k$ -средних не позволяет выявить кластеры более сложной формы

В данном случае мы понадеялись на то, что алгоритм кластеризации сможет обнаружить два кластера в форме полумесяцев. Однако определить их с помощью алгоритма  $k$ -средних не представляется возможным.

#### Векторное квантование или рассмотрение кластеризации $k$ -средних как декомпозиционного метода

Несмотря на то что алгоритм  $k$ -средних представляет собой алгоритм кластеризации, можно провести интересные параллели между алгоритмом  $k$ -средних и декомпозиционными методами типа PCA и NMF, которые мы обсуждали ранее. Возможно, вы помните, что PCA пытается найти направления максимальной дисперсии данных, в то время как NMF пытается найти аддитивные компоненты, которые часто соответствуют «экстремумам» или «группам» данных (см. рис. 3.13). Оба метода пытаются представить данные в виде суммы некоторых компонент. Алгоритм  $k$ -средних, напротив, пытается представить каждую точку данных в пространстве, используя центр кластера. Вообразите, что каждая точка представлена с помощью только одной компоненты, которая задается центром кластера. Рассмотрение алгоритма  $k$ -средних как декомпозиционного метода, в котором каждая точка представлена с помощью отдельной компоненты, называется *векторным квантованием* (*vector quantization*).

Давайте сравним PCA, NMF и алгоритм  $k$ -средних, показав выделенные компоненты (рис. 3.30), а также реконструкции изображений лиц из тестового набора с использованием 100 компонент (рис. 3.31). В алгоритме  $k$ -средних реконструкция изображения – это ближайший центр кластера, вычисленный на обучающем наборе:

```
In[57]:
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
nmf = NMF(n_components=100, random_state=0)
nmf.fit(X_train)
pca = PCA(n_components=100, random_state=0)
pca.fit(X_train)
kmeans = KMeans(n_clusters=100, random_state=0)
kmeans.fit(X_train)

X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)

In[58]:
fig, axes = plt.subplots(3, 5, figsize=(8, 8),
                        subplot_kw={'xticks': (), 'yticks': ()})
fig.suptitle("Извлеченные компоненты")
for ax, comp_kmeans, comp_pca, comp_nmf in zip(
    axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):
    ax[0].imshow(comp_kmeans.reshape(image_shape))
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')
    ax[2].imshow(comp_nmf.reshape(image_shape))

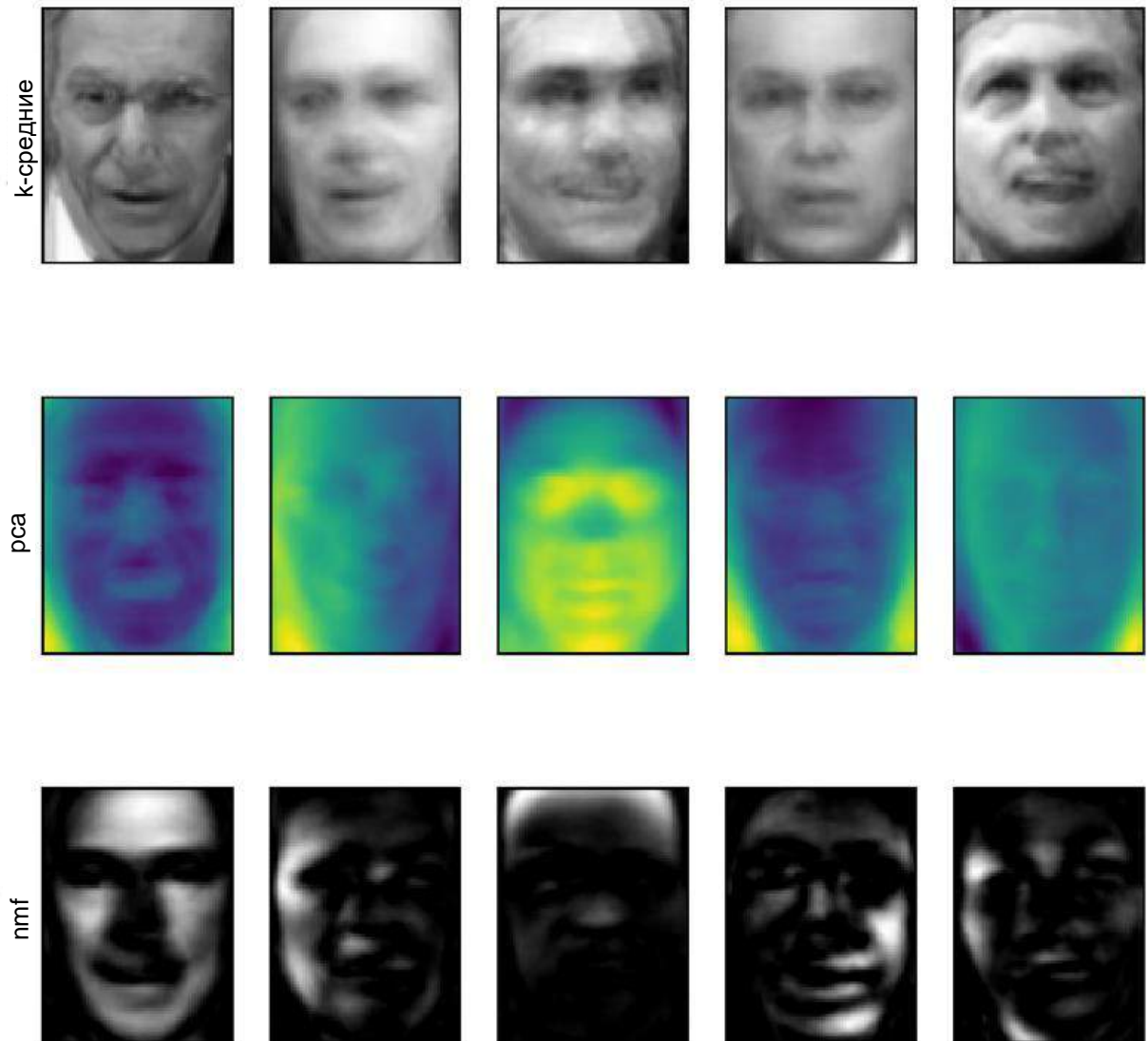
axes[0, 0].set_ylabel("k-средние")
axes[1, 0].set_ylabel("pca")
axes[2, 0].set_ylabel("nmf")

fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()},
                        figsize=(8, 8))
fig.suptitle("Реконструкции")
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(
    axes.T, X_test, X_reconstructed_kmeans, X_reconstructed_pca,
    X_reconstructed_nmf):

    ax[0].imshow(orig.reshape(image_shape))
    ax[1].imshow(rec_kmeans.reshape(image_shape))
    ax[2].imshow(rec_pca.reshape(image_shape))
    ax[3].imshow(rec_nmf.reshape(image_shape))

axes[0, 0].set_ylabel("исходный вид")
axes[1, 0].set_ylabel("k-средние")
axes[2, 0].set_ylabel("pca")
axes[3, 0].set_ylabel("nmf")
```

Извлеченные компоненты



**Рис. 3.30** Сравнение центров кластеров, вычисленных с помощью алгоритма k-средних, и компонент, вычисленных с помощью PCA и NMF

## Реконструкции



**Рис. 3.31** Сравнение реконструкций изображений, полученных с помощью алгоритма  $k$ -средних, PCA и NMF со 100 компонентами (или центрами кластеров), алгоритм  $k$ -средних использует лишь один центр кластера на изображение

Интересная деталь векторного квантования с помощью алгоритма  $k$ -средних заключается в том, что для представления наших данных мы можем использовать число кластеров, намного превышающее число входных измерений. Давайте вернемся к данным `two_moons`. Применив к этим данным PCA или NMF, мы ничего примечательного с ними не сделаем, поскольку данные представлены двумя измерениями. Снижение до одного измерения с помощью PCA или NMF полностью бы разрушило структуру данных. Но мы можем найти более выразительное представление данных с помощью алгоритма  $k$ -средних, используя большее количество центров кластеров (см. рис. 3.32):

```

In[59]:
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)

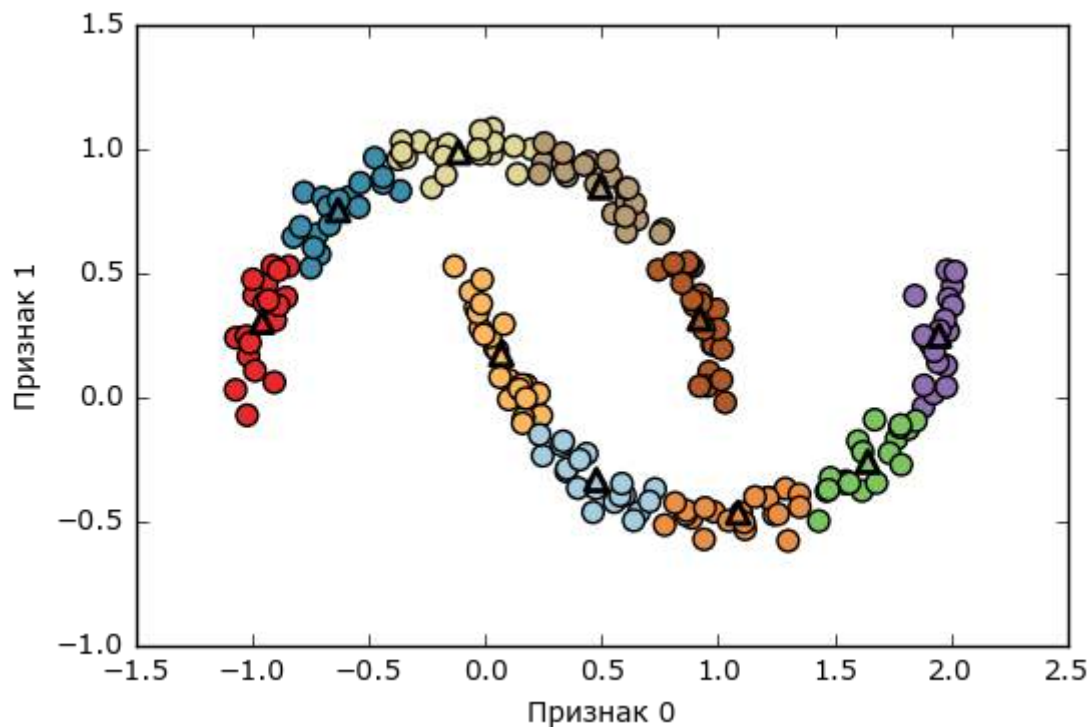
plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[0], kmeans.cluster_centers_[1], s=60,
            marker='^', c=range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
print("Принадлежность к кластерам:\n{}".format(y_pred))

```

```

Out[59]:
Принадлежность к кластерам:
[9 2 5 4 2 7 9 6 9 6 1 0 2 6 1 9 3 0 3 1 7 6 8 6 8 5 2 7 5 8 9 8 6 5 3 7 0
 9 4 5 0 1 3 5 2 8 9 1 5 6 1 0 7 4 6 3 3 6 3 8 0 4 2 9 6 4 8 2 8 4 0 4 0 5
 6 4 5 9 3 0 7 8 0 7 5 8 9 8 0 7 3 9 7 1 7 2 2 0 4 5 6 7 8 9 4 5 4 1 2 3 1
 8 8 4 9 2 3 7 0 9 9 1 5 8 5 1 9 5 6 7 9 1 4 0 6 2 6 4 7 9 5 5 3 8 1 9 5 6
 3 5 0 2 9 3 0 8 6 0 3 3 5 6 3 2 0 2 3 0 2 6 3 4 4 1 5 6 7 1 1 3 2 4 7 2 7
 3 8 6 4 1 4 3 9 9 5 1 7 5 8 2]

```



**Рис. 3.32** Использование большого количества кластеров для выявления дисперсии в сложном наборе данных

Мы использовали 10 центров кластеров, это значит, что каждой точке данных теперь будет присвоен номер от 0 до 9. Мы можем убедиться в этом, поскольку данные теперь представлены с помощью 10 компонент (т.е. у нас теперь появилось 10 новых признаков), при этом все признаки равны 0, за исключением признака, который представляет собой центр кластера, назначенный конкретной точке данных. Применив это 10-мерное представление, теперь мы можем отделить эти два скопления данных в виде полумесяцев с помощью линейной модели, что было бы невозможным, если бы использовали два исходных признака. Кроме того,

можно получить еще более выразительное представление данных, используя расстояния до каждого центра кластера в качестве признаков. Это можно сделать с помощью метода `transform` класса `kmeans`:

```
In[60]:
distance_features = kmeans.transform(X)
print("Форма характеристик-расстояний: {}".format(distance_features.shape))
print("Характеристики-расстояния:\n{}".format(distance_features))

Out[60]:
Форма характеристик-расстояний: (200, 10)
Характеристики-расстояния:
[[ 0.922  1.466  1.14   ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12   ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749   ...,  1.971  0.716  0.944]
 ...,
 [ 0.446  1.106  1.49   ...,  1.791  1.032  0.812]
 [ 1.39   0.798  1.981   ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045   ...,  0.572  2.113  0.882]]
```

Алгоритм  $k$ -средних является очень популярным алгоритмом кластеризации не только потому, что его относительно легко понять и реализовать, но и потому, что он работает сравнительно быстро. Алгоритм  $k$ -средних легко масштабируется на большие наборы данных, а `scikit-learn` еще и включает в себя более масштабируемый вариант, реализованный в классе `MiniBatchKMeans`, который может обрабатывать очень большие наборы данных.

Один из недостатков алгоритма  $k$ -средних заключается в том, что он зависит от случайной инициализации (т.е. результат алгоритма зависит от случайного стартового значения). По умолчанию `scikit-learn` запускает алгоритм 10 раз с 10 различными случайными стартовыми значениями и возвращает лучший результат.<sup>26</sup> Дополнительными недостатками алгоритма  $k$ -средних являются относительно строгие предположения о форме кластеров, а также необходимость задать число выделяемых кластеров (которое в реальной практике может быть неизвестно).

Далее мы рассмотрим еще два алгоритма кластеризации, которые в некотором роде позволяют исправить вышеописанные недостатки.

## Агломеративная кластеризация

*Агломеративная кластеризация (agglomerative clustering)* относится к семейству алгоритмов кластеризации, в основе которых лежат одинаковые принципы: алгоритм начинает свою работу с того, что каждую точку данных заносит в свой собственный кластер и по мере выполнения объединяет два наиболее схожих между собой кластера до

---

<sup>26</sup> В данном случае «лучший результат» означает такое разбиение на кластеры, при котором сумма внутрикластерных (внутригрупповых) дисперсий будет минимальной.



тех пор, пока не будет удовлетворен определенный критерий остановки. Критерий остановки, реализованный в `scikit-learn` — это количество кластеров, поэтому схожие между собой кластеры объединяются до тех пор, пока не останется заданное число кластеров. Есть несколько критериев *связи* (*linkage*), которые задают точный способ измерения «наиболее схожего кластера». В основе этих критериев лежит расстояние между двумя существующими кластерами.

В `scikit-learn` реализованы следующие три критерия:

#### **ward**

метод по умолчанию **ward** (метод Варда) выбирает и объединяет два кластера так, чтобы прирост дисперсии внутри кластеров был минимальным. Часто этот критерий приводит к получению кластеров относительно одинакового размера.

#### **average**

метод **average** (метод средней связи) объединяет два кластера, которые имеют наименьшее среднее значение всех расстояний, измеренных между точками двух кластеров.

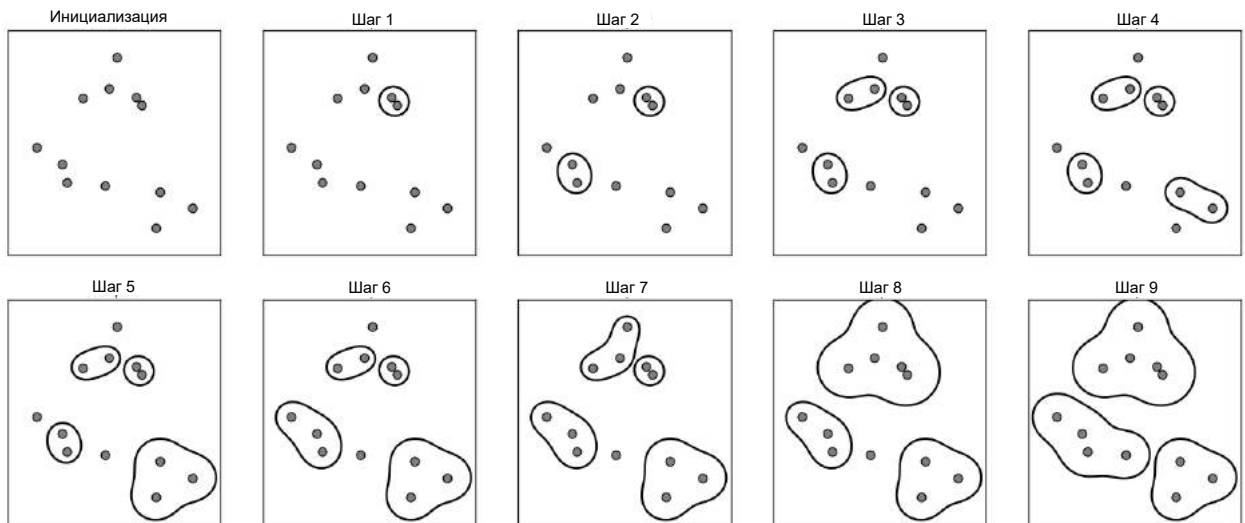
#### **complete**

метод **complete** (метод полной связи или метод максимальной связи) объединяет два кластера, которые имеют наименьшее расстояние между двумя их самыми удаленными точками.

**ward** подходит для большинства наборов данных и мы будем использовать именно его в наших примерах. Если кластеры имеют сильно различающиеся размеры (например, один кластер содержит намного больше точек данных, чем все остальные), использование критериев **average** или **complete** может дать лучший результат.

Следующий график (рис. 3.33) иллюстрирует работу алгоритма агломеративной кластеризации на двумерном массиве данных, который ищет три кластера:

```
In[61]:
mglearn.plots.plot_agglomerative_algorithm()
```



**Рис. 3.33** Алгоритм агломеративной кластеризации итеративно объединяет два ближайших кластера

Изначально количество кластеров равно количеству точек данных. Затем на каждом шаге объединяются два ближайших друг к другу кластера. На первых четырех шагах выбираются кластеры, состоящие из отдельных точек, и объединяются в кластеры, состоящие из двух точек. На шаге 5 один из 2-точечных кластеров вбирает в себя третью точку и т.д. На шаге 9 у нас остается три кластера. Поскольку мы установили количество кластеров равным 3, алгоритм останавливается.

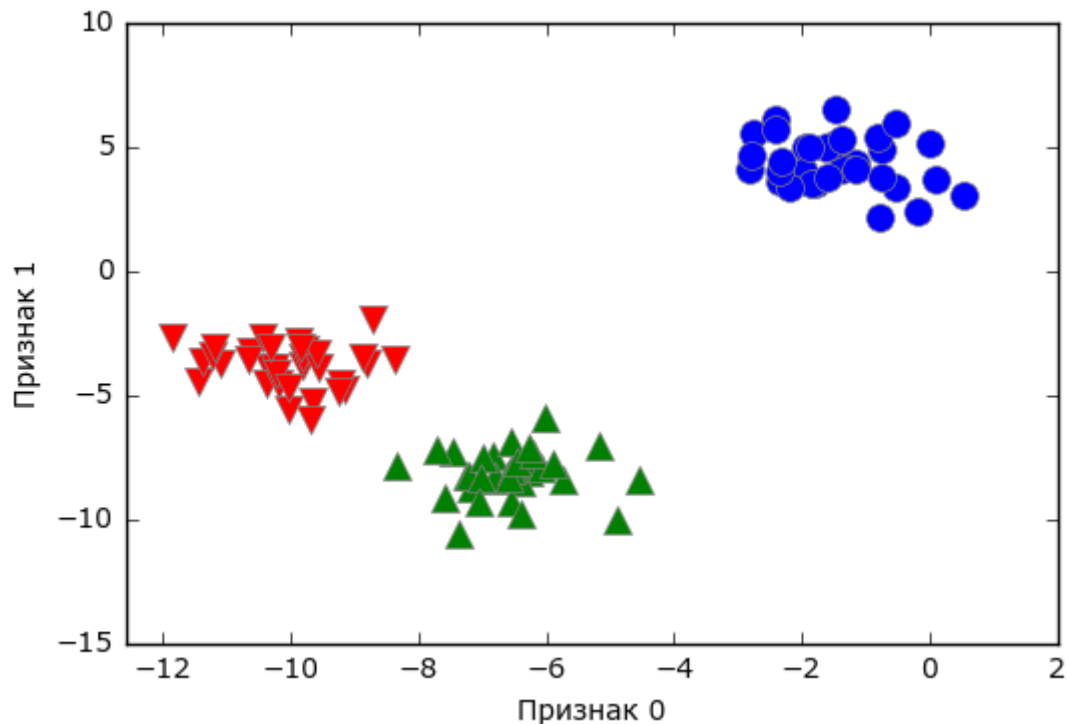
Давайте рассмотрим работу алгоритма агломеративной кластеризации на простых трехкластерных данных, использованных здесь. Из-за своего способа работы алгоритм агломеративной кластеризации не может вычислить прогнозы для новых точек данных. Поэтому алгоритм агломеративной кластеризации не имеет метода `predict`. Для того, чтобы построить модель и вычислить принадлежность к кластерам на обучающем наборе, используйте метод `fit_predict`.<sup>27</sup> Результат показан на рис. 3.34:

```
In[62]:
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

<sup>27</sup> Кроме того, мы могли бы воспользоваться атрибутом `labels_`, как мы это уже делали для алгоритма *k*-средних.



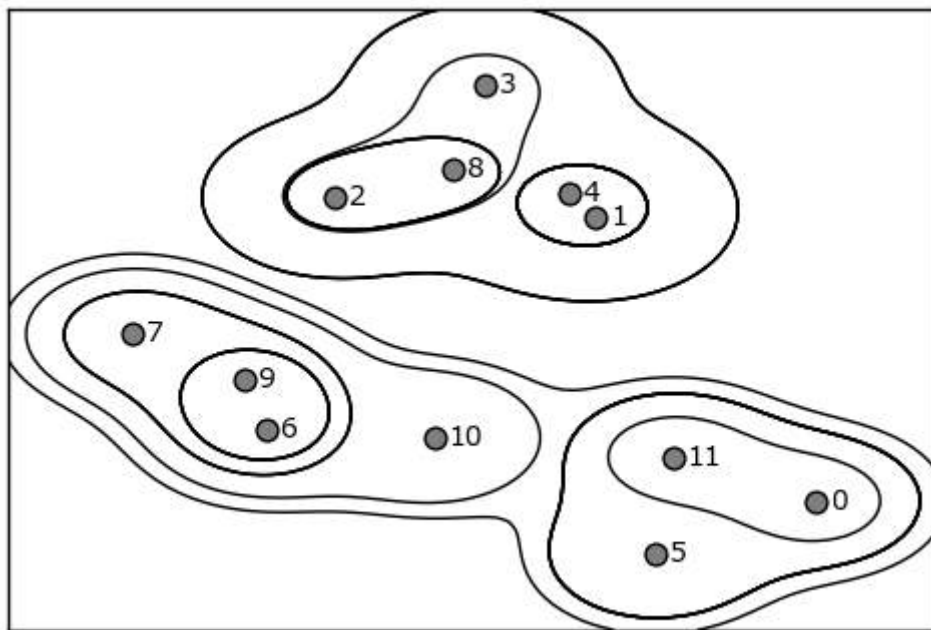
**Рис. 3.34** Принадлежность к кластерам, вычисленная алгоритмом агломеративной кластеризации с тремя кластерами

Как и ожидалось, алгоритм отлично восстанавливает кластеризацию. Хотя алгоритм агломеративной кластеризации, реализованный в `scikit-learn`, требует указать количество выделяемых кластеров, методы агломеративной кластеризации в некоторой степени помогают выбрать правильное количество кластеров, об этом и пойдет речь ниже.

### Иерархическая кластеризация и дендрограммы

Результатом агломеративной кластеризации является *иерархическая кластеризация* (*hierarchical clustering*). Кластеризация выполняется итеративно, и каждая точка совершает путь от отдельной точки-кластера до участника итогового кластера. На каждом промежуточном шаге происходит кластеризация данных (с разным количеством кластеров). Иногда полезно сразу взглянуть на все возможные кластеризации. Следующий пример (рис. 3.35) показывает наложение всех возможных кластеризаций, показанных на рис. 3.33 и дает некоторое представление о том, как каждый кластер распадается на более мелкие кластеры:

```
In[63]:
mglearn.plots.plot_agglomerative()
```



**Рис. 3.35** Иерархическое присвоение кластеров (показаны в виде линий), полученное с помощью алгоритма агломеративной кластеризации, точки данных пронумерованы (см. рис. 3.36)

Хотя эта визуализация дает достаточно детализированное представление о результатах иерархической кластеризации, она опирается на двумерную природу данных и не может быть использована для наборов данных, которые имеют более двух характеристик. Однако есть еще один инструмент для визуализации результатов иерархической кластеризации, называемый *дендрограммой* (*dendrogram*) и позволяющий обрабатывать многомерные массивы данных.

К сожалению, на данный момент в **scikit-learn** нет инструментов, позволяющих рисовать дендрограммы. Однако вы легко можете создать их с помощью SciPy. По сравнению с алгоритмами кластеризации **scikit-learn** алгоритмы кластеризации SciPy имеют немного другой интерфейс. В SciPy используется функция, которая принимает массив данных *X* в качестве аргумента и вычисляет *массив связей* (*linkage array*) с записанными сходствами между кластерами. Затем мы можем скормить этот массив функции SciPy **dendrogram**, чтобы построить дендрограмму (рис. 3.36):

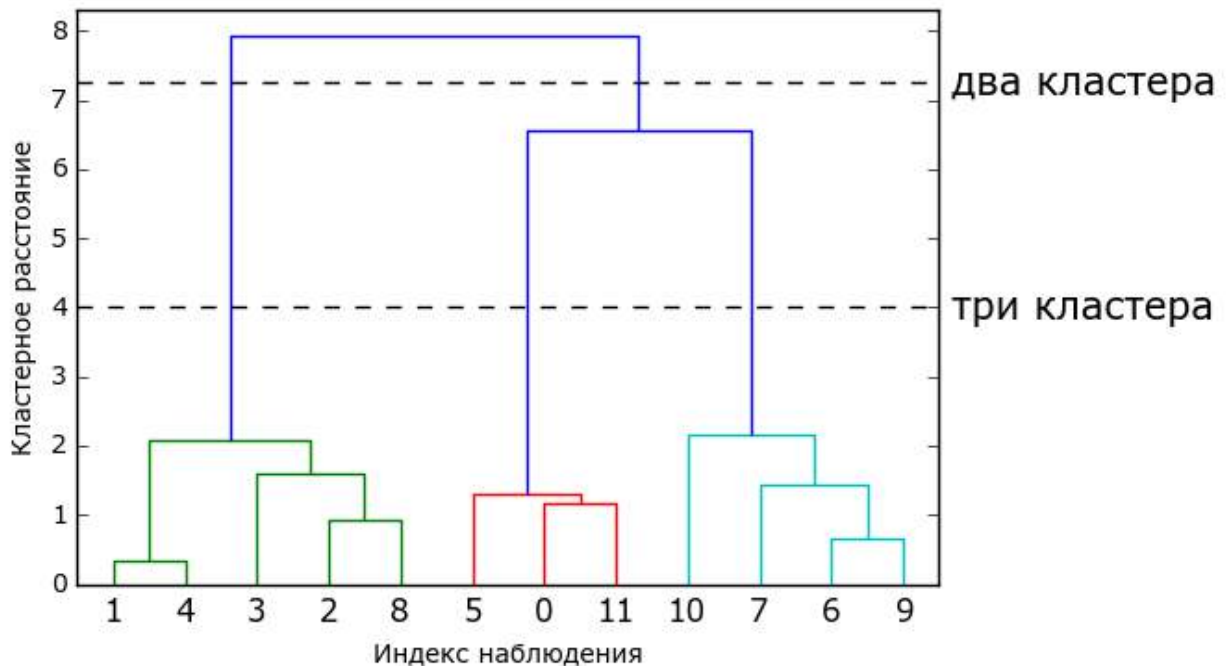
```
In[64]:
# импортируем функцию dendrogram и функцию кластеризации ward из SciPy
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# применяем кластеризацию ward к массиву данных X
# функция SciPy ward возвращает массив с расстояниями
# вычисленными в ходе выполнения агломеративной кластеризации
linkage_array = ward(X)
```

```
# теперь строим дендрограмму для массива связей, содержащего расстояния
# между кластерами
dendrogram(linkage_array)

# делаем отметки на дереве, соответствующие двум или трем кластерам
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, ' два кластера', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' три кластера', va='center', fontdict={'size': 15})
plt.xlabel("Индекс наблюдения")
plt.ylabel("Кластерное расстояние")
```



**Рис. 3.36** Дендрограмма для кластеризации, показанной на рис. 3.35, линии обозначают расщепления на два и три кластера

Точки данных показаны в нижней части дендрограммы (пронумерованы от 0 до 11). Затем строится дерево с этими точками (представляющими собой кластеры-точки) в качестве листьев, и для каждого двух объединенных кластеров добавляется новый узел-родитель.

Чтение дендрограммы происходит снизу вверх. Точки данных 1 и 4 объединяются первыми (как вы уже могли видеть на рис. 3.33). Затем в кластер объединяются точки 6 и 9 и т.д. На самом верхнем уровне остаются две ветви, одна ветвь состоит из точек 11, 0, 5, 10, 7, 6 и 9, а вторая – из точек 1, 4, 3, 2 и 8. Они соответствуют двум крупнейшим кластерам.

Ось y в дендрограмме указывает не только момент объединения двух кластеров в ходе работы алгоритма агломеративной кластеризации. Длина каждой ветви показывает, насколько далеко друг от друга находятся объединенные кластеры. Самыми длинными ветвями в этой

дендрограмме являются три линии, отмеченные пунктирной чертой с надписью «три кластера». Тот факт, что эти линии являются самыми длинными ветвями, указывает на то, что переход от трех кластеров к двум сопровождался объединением некоторых сильно удаленных друг от друга точек. Мы снова видим это в самой верхней части графика, когда объединение двух оставшихся кластеров в единый кластер подразумевает относительно большое расстояние между точками.

К сожалению, алгоритм агломеративной кластеризации по-прежнему не в состоянии обработать сложные данные типа набора `two_moons`. Чего нельзя сказать о DBSCAN, следующем алгоритме, который мы рассмотрим.

## DBSCAN

Еще один очень полезный алгоритм кластеризации – DBSCAN (density-based spatial clustering of applications with noise – плотностный алгоритм кластеризации пространственных данных с присутствием шума). Основные преимущества алгоритма DBSCAN заключаются в том, что пользователю не нужно *заранее* задавать количество кластеров, алгоритм может выделить кластеры сложной формы и способен определить точки, которые не принадлежат какому-либо кластеру. DBSCAN работает немного медленнее, чем алгоритм агломеративной кластеризации и алгоритм  $k$ -средних, но также может масштабироваться на относительно большие наборы данных.

DBSCAN определяет точки, расположенные в «густонаселенных» областях пространства характеристик, когда многие точки данных расположены близко друг к другу. Эти области называются *плотными* (*dense*) областями пространства характеристик. Идея алгоритма DBSCAN заключается в том, что кластеры образуют плотные области данных, которые отделены друг от друга относительно пустыми областями.

Точки, находящиеся в плотной области, называются *ядровыми примерами* (*core samples*) или *ядровыми точками* (*core points*). Алгоритм DBSCAN имеет два параметра: `min_samples` и `eps`. Если по крайней мере `min_samples` точек находятся в радиусе окрестности `eps` рассматриваемой точки, то эта точка классифицируется как ядровая. Ядровые точки, расстояния между которыми не превышают радиус окрестности `eps`, помещаются алгоритмом DBSCAN в один и тот же кластер.

На старте алгоритм выбирает произвольную точку. Затем он находит все точки, удаленные от стартовой точки на расстоянии, не превышающем радиус окрестности `eps`. Если множество точек, находящихся в пределах радиуса окрестности `eps`, меньше значения

`min_samples`, стартовая точка помечается как *шум* (*noise*), это означает, что она не принадлежит какому-либо кластеру. Если это множество точек больше значения `min_samples`, стартовая точка помечается как ядровая и ей назначается метка нового кластера. Затем посещаются все соседи этой точки (находящиеся в пределах `eps`). Если они еще не были присвоены кластеру, им присваивается метка только что созданного кластера. Если они являются ядровыми точками, поочередно посещаются их соседи и т.д. Кластер растет до тех пор, пока не останется ни одной ядерной точки в пределах радиуса окрестности `eps`. Затем выбирается другая точка, которая еще не была посещена, и повторяется та же самая процедура.

В итоге получаем три вида точек: ядровые точки, точки, которые находятся в пределах радиуса окрестности `eps` ядровых точек (так называемые *пограничные точки* или *boundary points*) и шумовые точки. При многократном применении алгоритма DBSCAN к конкретному набору данных результаты кластеризации ядровых точек будут всегда одинаковыми, при этом одни и те же точки всегда будут помечаться как шумовые. Однако пограничная точка может быть соседом для ядровых точек из нескольких кластеров. Поэтому кластерная принадлежность пограничных точек зависит от порядка посещения точек. Как правило, существует лишь несколько пограничных точек, поэтому эта слабая зависимость результатов кластеризации от порядка посещения точек не имеет значения.

Давайте применим алгоритм DBSCAN к синтетическому набору данных, который мы использовали для демонстрации агломеративной кластеризации. Как и алгоритм агломеративной кластеризации, алгоритм DBSCAN не позволяет получать прогнозы для новых тестовых данных, поэтому мы воспользуемся методом `fit_predict`, чтобы сразу выполнить кластеризацию и вернуть метки кластеров:

```
In[65]:
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X)
print("Принадлежность к кластерам:\n{}".format(clusters))

Out[65]:
Принадлежность к кластерам:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

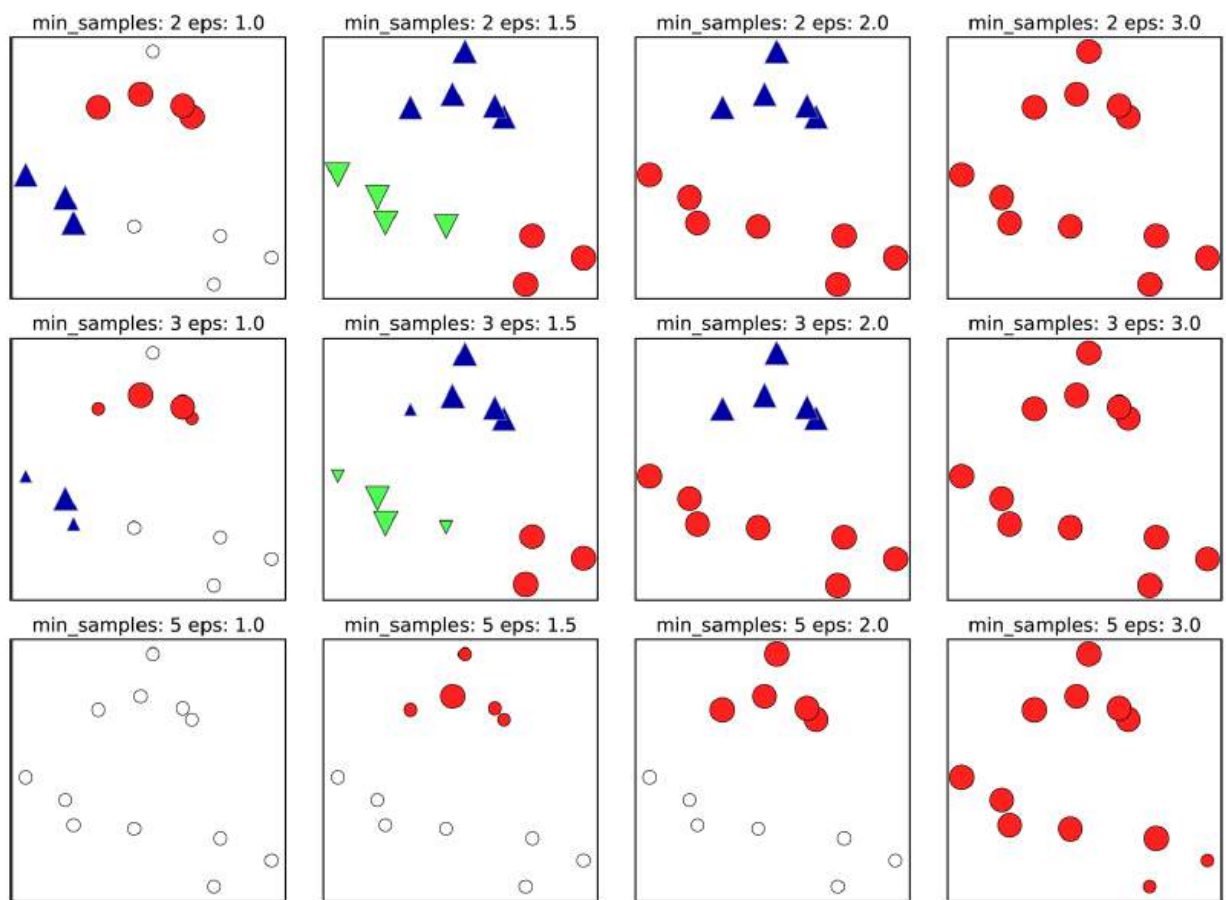
Можно увидеть, что всем точкам данных была присвоена метка -1, которая обозначает шум. Полученная сводка является результатом применения значений `eps` и `min_samples`, установленных по умолчанию и не настроенных для работы с небольшими синтетическими наборами



данных. Принадлежность к кластерам для различных значений `min_samples` и `eps` показана в сводке и визуализирована на рис. 3.37:

```
In[66]:
mglearn.plots.plot_dbscan()

Out[66]:
min_samples: 2 eps: 1.000000 cluster: [-1 0 0 -1 0 -1 1 1 0 1 -1 -1]
min_samples: 2 eps: 1.500000 cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 2 eps: 2.000000 cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 2 eps: 3.000000 cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 3 eps: 1.000000 cluster: [-1 0 0 -1 0 -1 1 1 0 1 -1 -1]
min_samples: 3 eps: 1.500000 cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 3 eps: 2.000000 cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 3 eps: 3.000000 cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 5 eps: 1.000000 cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
min_samples: 5 eps: 1.500000 cluster: [-1 0 0 0 0 -1 -1 -1 0 -1 -1 -1]
min_samples: 5 eps: 2.000000 cluster: [-1 0 0 0 0 -1 -1 -1 0 -1 -1 -1]
min_samples: 5 eps: 3.000000 cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
```



**Рис. 3.37** Принадлежность к кластерам, вычисленная с помощью алгоритма DBSCAN при различных значениях `min_samples` и `eps`

На этом графике точки, которые принадлежат кластерам, окрашены сплошным цветом, а шумовые точки – белым цветом. Ядровые точки показаны в виде больших маркеров, тогда как пограничные точки отображаются в виде небольших маркеров. Увеличение значения `eps` (слева направо на рисунке) означает включение большего количества точек в кластер. Это приводит к росту размеров кластеров, но также

может привести к тому, что несколько кластеров будут объединены в один. Увеличение значения `min_samples` (сверху вниз на рисунке) означает уменьшение количества ядерных точек и увеличение количества шумовых точек.

Параметр `eps` чуть более важен, поскольку он определяет, что подразумевается под «близостью» точек друг к другу. Очень маленькое значение `eps` будет означать отсутствие ядерных точек и может привести к тому, что все точки будут помечены как шумовые. Очень большое значение `eps` приведет к тому, что все точки сформируют один кластер.

Значение `min_samples` главным образом определяет, будут ли точки, расположенные в менее плотных областях, помечены как выбросы или как кластеры. Если увеличить значение `min_samples`, все, что могло бы стать кластером с количеством точек, не превышающим `min_samples`, будет помечено как шум. Поэтому значение `min_samples` задает минимальный размер кластера. Это очень четко можно увидеть на рис. 3.37, когда мы увеличиваем значение `min_samples` с 3 до 5 при `eps=1.5`. При `min_samples = 3` получаем три кластера: первый кластер состоит из четырех точек, второй – из пяти точек и третий – из трех точек. При `min_samples=5` два кластера меньшего размера (с тремя и четырьмя точками) теперь помечены как шум и остается лишь кластер с пятью точками.

Несмотря на то что в алгоритме DBSCAN не нужно явно указывать количество кластеров, значение `eps` неявно задает количество выделяемых кластеров. Иногда подобрать оптимальное значение `eps` становится проще после масштабирования данных с помощью `StandardScaler` или `MinMaxScaler`, так как использование этих методов масштабирования гарантирует, что все характеристики будут иметь одинаковый масштаб.

Рис. 3.38 показывает результат выполнения алгоритма DBSCAN для наборе данных `two_moons`. Алгоритм фактически находит две группы данных в форме полумесяцев и разделяет их, используя настройки по умолчанию:

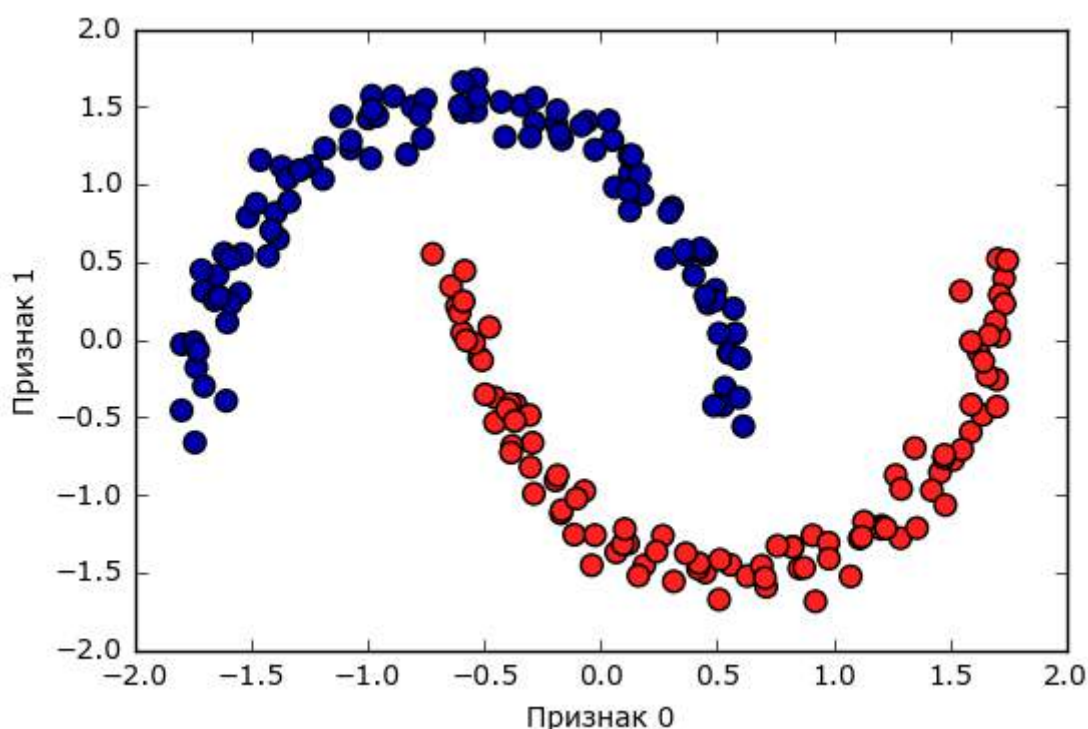
```
In[67]:
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# масштабируем данные так, чтобы получить нулевое среднее и единичную дисперсию
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbscan = DBSCAN()
clusters = dbscan.fit_predict(X_scaled)
# выводим принадлежность к кластерам
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mglearn.cm2, s=60)
plt.xlabel("Признак 0")
plt.ylabel("Признак 1")
```

Поскольку алгоритм выделил нужное количество кластеров (два), настройки параметров, похоже, работают хорошо. Если мы уменьшим значение `eps` до `0.2` (значение по умолчанию `0.5`), мы получим восемь кластеров, что явно слишком много. Увеличение `eps` до `0.7` даст один кластер.

Используя DBSCAN, будьте осторожны при работе с возвращаемыми номерами кластеров. Использование `-1` для обозначения шума может привести к неожиданным эффектам, если метки кластеров будут использоваться для индексирования другого массива.



**Рис. 3.38** Принадлежность к кластерам, вычисленная с помощью алгоритма DBSCAN, использовалось значение по умолчанию `eps=0.5`

### Сравнение и оценка качества алгоритмов кластеризации

Одна из проблем, связанных с применением алгоритмов кластеризации, заключается в том, что очень трудно оценить качество работы алгоритма и сравнить результаты, полученные с помощью различных алгоритмов. Рассказав об алгоритмах *k*-средних, агломеративной кластеризации и DBSCAN, мы теперь сравним их, применив к некоторым реальным наборам данных.

Оценка качества кластеризации с помощью метрик, предполагающих знание истинной кластеризации

Существует показатели, которые можно использовать для оценки результатов с точки зрения истинной кластеризации. Наиболее важными среди них являются *скорректированный коэффициент Рэнда* (*adjusted Rand index*, ARI) и *нормализованная взаимная информация* (*normalized mutual information*, NMI), которые представляют собой количественные показатели. Они принимают значения, близкие к 0, при случайном назначении кластеров, и значение 1, когда полученные результаты кластеризации полностью совпадают с фактическими (обратите внимание, скорректированный коэффициент Рэнда может принимать значения от -1 до 1).

В данном случае мы сравним алгоритмы *k*-средних, агломеративной кластеризации и DBSCAN, используя ARI. Кроме того, для сравнения мы включим результаты кластеризации, полученные при случайном назначении точек двум кластерам (рис. 3.39):

```
In[68]:
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

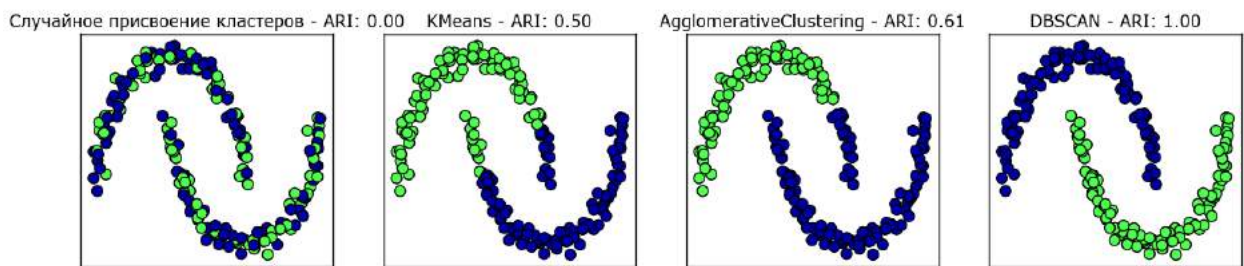
# масштабируем данные так, чтобы получить нулевое среднее и единичную дисперсию
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                        subplot_kw={'xticks': (), 'yticks': ()})

# создаем список используемых алгоритмов
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

# случайно присваиваем точки двум кластерам для сравнения
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# выводим на графике результаты случайного присвоения кластеров
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
               cmap=mpl.cm3, s=60)
axes[0].set_title("Случайное присвоение кластеров - ARI: {:.2f}".format(
    adjusted_rand_score(y, random_clusters)))

for ax, algorithm in zip(axes[1:], algorithms):
    # выводим на графике принадлежность к кластерам и центры кластеров
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
              cmap=mpl.cm3, s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                          adjusted_rand_score(y, clusters)))
```



**Рис. 3.38** Сравнение результатов случайной кластеризации,  $k$ -средних, агломеративной кластеризации и DBSCAN для набора данных `two_moons`, использовался скорректированный коэффициент Рэнда

Скорректированный коэффициент Рэнда показывает интуитивно понятные результаты, случайное присвоение кластеров получает оценку 0, а DBSCAN (который отлично восстанавливает нужные кластеры) – оценку 1.

Наиболее частая ошибка, возникающая при оценке результатов кластеризации, заключается в использовании `accuracy_score` вместо `adjusted_rand_score`, `normalized_mutual_info_score` или какой-либо другого показателя качества кластеризации. Проблема, связанная с использованием правильности, заключается в том, что оценка правильности требует точного соответствия меток кластеров, присвоенных точкам, истинным меткам кластеров (`ground truth`). Однако сами по себе метки кластеров не имеют смысла. Единственное, что имеет значение, это то, какие точки находятся в одном и том же кластере:

```
In[69]:
from sklearn.metrics import accuracy_score

# эти две маркировки точек соответствуют одним и тем же результатам кластеризации
# в clusters1 записаны фактические результаты кластеризации,
# а в clusters2 записаны расчетные результаты кластеризации
clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
# правильность равна нулю, поскольку ни одна из присвоенных меток не отражает
# истинную кластеризацию
print("Правильность: {:.2f}".format(accuracy_score(clusters1, clusters2)))
# значение скорр. коэффициента Рэнда равно 1, поскольку полученные результаты
# точно воспроизводят истинную кластеризацию
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))

Out[69]:
Правильность: 0.00
ARI: 1.00
```

Оценка качества кластеризации без использования метрик, предполагающих знание истинной кластеризации

Несмотря на то, что мы только что показали один из способов оценить работу алгоритмов кластеризации, на практике использование метрик типа ARI сопряжено с большими проблемами. При использовании алгоритмов кластеризации, информация об истинных кластерах, с которой можно было бы сравнить полученные результаты, как правило,

отсутствует. Если бы мы знали, как выглядит правильная кластеризация данных, мы могли бы использовать эту информацию, чтобы построить модель контролируемого обучения типа классификатора. Поэтому, использование таких показателей, как ARI и NMI, как правило, помогает в разработке алгоритмов, но не в оценке их эффективности с точки зрения конкретного применения.

Существуют метрики кластеризации, которые не требуют знания истинных результатов кластеризации, например, *силуэтный коэффициент* (*silhouette coefficient*). Однако на практике они работают плохо. Силуэтная мера вычисляет компактность кластера, более высокое значение соответствует лучшему результату, идеальное значение равно 1. Несмотря на то, что компактные кластеры удобны, компактность не предполагает сложных форм.

Ниже приведен пример сравнения результатов, полученных с помощью алгоритмов *k*-средних, агломеративной кластеризации и DBSCAN для набора `two_moons`, при этом использовалась силуэтная мера (рис. 3.40):

```
In[70]:
from sklearn.metrics.cluster import silhouette_score

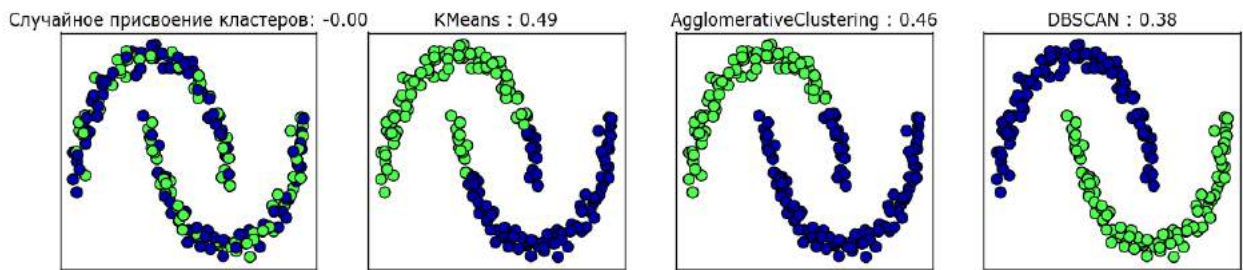
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# масштабируем данные так, чтобы получить нулевое среднее и единичную дисперсию
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                        subplot_kw={'xticks': (), 'yticks': ()})

# случайно присваиваем точки двум кластерам для сравнения
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# выводим на графике результаты случайного присвоения кластеров
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
               cmap=mpl.cm3, s=60)
axes[0].set_title("Случайное присвоение кластеров: {:.2f}".format(
    silhouette_score(X_scaled, random_clusters)))
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    # выводим на графике принадлежность к кластерам и центры кластеров
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mpl.cm3,
              s=60)
    ax.set_title("{} : {:.2f}".format(algorithm.__class__.__name__,
                                     silhouette_score(X_scaled, clusters)))
```





**Рис. 3.38** Сравнение результатов случайной кластеризации,  $k$ -средних, агломеративной кластеризации и DBSCAN для набора данных `two_moons`, использовался силуэтный коэффициент (более интуитивно понятный результат DBSCAN имеет меньшее значение силуэтного коэффициента, чем результат  $k$ -средних)

Можно увидеть, что алгоритм  $k$ -средних получает самое высокое значение силуэтного коэффициента, хотя мы, возможно, предпочтем результаты, полученные с помощью алгоритма DBSCAN. Чуть более лучшая стратегия для оценки результатов кластеризации заключается в использовании метрик кластеризации на основе робастности (*robustness-based clustering metrics*). Эти метрики запускают алгоритм после добавления некоторого шума в данные или применяют различные настройки параметров, а затем сравнивают полученные результаты. Идея заключается в том, что если разные значения параметров и разные возмущения данных возвращают один и тот же результат, ему, вероятно, можно доверять. К сожалению, на момент написания книги эта стратегия не была реализована в `scikit-learn`.

Даже если мы получим очень робастные результаты кластеризации или очень высокое значение силуэтной меры, у нас по-прежнему будет отсутствовать информация о том, несут ли результаты кластеризации какой-то содержательный смысл, отражают ли они те аспекты данных, которые нас интересуют. Давайте вернемся к примеру с изображениями лиц. Мы надеемся выделить группы схожих между собою лиц, скажем, мужчин и женщин, либо пожилых и молодых, или людей с бородой и без бороды. Допустим, мы сгруппировали данные в два кластера, и все алгоритмы единодушны о том, какие точки данных должны быть объединены в кластеры. Мы по-прежнему не знаем, соответствуют ли найденные кластеры интересующим нас гипотезам. Вполне возможно, что они выделили лица в профиль и лица в анфас или снимки, сделанные в ночное время, и снимки, сделанные днем, фото, сделанные с айфонов и фото, сделанные с телефонов с операционной системой Android. Единственный способ узнать, соответствует ли кластеризация интересующей нас информации, проанализировать кластеры вручную.



Сравнение работы алгоритмов на наборе изображений лиц

Давайте применим алгоритмы  $k$ -средних, DBSCAN и агломеративной кластеризации к набору данных Labeled Faces in the Wild и посмотрим, сможет ли какой-либо из этих алгоритмов найти интересную структуру. Мы воспользуемся собственными векторами (собственными лицами), вычисленными для всего набора изображений лиц при помощи PCA(whiten=True), выделялось 100 компонент:

In[71]:

```
# извлекаем собственные лица для набора данных lfw и преобразуем данные
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

Ранее мы видели, что данная операция позволяет получить более содержательную информацию об изображениях лиц в отличие от исходных пикселей. Кроме того, она позволяет увеличить скорость вычислений. Здесь будет полезно запустить алгоритмы на исходных данных, без применения PCA, и выяснить, смогли ли алгоритмы выделить аналогичные кластеры.

Анализ набора изображений лиц с помощью алгоритма DBSCAN

Мы начнем с применения DBSCAN, о котором только что говорили:

In[72]:

```
# применяем алгоритм DBSCAN, используя параметры по умолчанию
dbscan = DBSCAN()
labels = dbscan.fit_predict(X_pca)
print("Уникальные метки: {}".format(np.unique(labels)))
```

Out[72]:

Уникальные метки: [-1]

Мы видим, что все возвращенные метки имеют значение -1, таким образом, согласно алгоритму DBSCAN все данные были помечены как «шум». Здесь у нас есть два инструмента, что исправить эту ситуацию: мы можем увеличить значение `eps`, чтобы расширить окрестность каждой точки и уменьшить значение `min_samples`, чтобы рассматривать в качестве кластеров группы с меньшим количеством точек. Давайте сначала попробуем изменить значение `min_samples`:

In[73]:

```
dbscan = DBSCAN(min_samples=3)
labels = dbscan.fit_predict(X_pca)
print("Уникальные метки: {}".format(np.unique(labels)))
```

Out[73]:

Уникальные метки: [-1]

Даже при количестве точек в группе, равном 3, все точки помечаются как шум. Таким образом, нам нужно увеличить значение `eps`:

```
In[74]:
dbscan = DBSCAN(min_samples=3, eps=15)
labels = dbscan.fit_predict(X_pca)
print("Уникальные метки: {}".format(np.unique(labels)))
```

```
Out[74]:
Уникальные метки: [-1  0]
```

Задав значительно большее значение `eps` (15), мы получаем только один кластер и шумовые точки. Мы можем воспользоваться этим результатом, чтобы выяснить, что представляет из себя «шум» по сравнению с остальными данными. Чтобы лучше понять суть происходящего, давайте выясним, сколько точек являются шумовыми и сколько точек находятся внутри кластера:

```
In[75]:
# Считаем количество точек в кластерах и шум.
# bincount не допускает отрицательных цифр, поэтому нам нужно добавить 1.
# Первая цифра в выводе соответствует количеству шумовых точек.
print("Количество точек на кластер: {}".format(np.bincount(labels + 1)))
```

```
Out[75]:
Количество точек на кластер: [ 27 2036]
```

Шумовых точек оказалось очень мало, около 27, поэтому мы можем все эти точки посмотреть (см. рис. 3.41).

```
In[76]:
noise = X_people[labels==-1]
fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},
                        figsize=(12, 4))
for image, ax in zip(noise, axes.ravel()):
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```



**Рис. 3.41** Примеры из набора изображений лиц, помеченные алгоритмом DBSCAN как шум

Сравнивая эти изображения со случайной выборкой изображений лиц на рис. 3.7, мы можем догадаться, почему они были помечены как шум: на пятом фото в первом ряду изображен человек, пьющий из стакана,

также попадают изображения людей в головных уборах, а на последнем фото изображена рука перед лицом человека. Другие фото сняты с необычного ракурса или имеют необычный план, который может быть крупным или общим.

Подобного рода анализ, который пытается найти «лишнее», называется *детекцией выбросов (outlier detection)*. Если бы это был реальный пример, мы могли бы попытаться более аккуратно кадрировать изображения, чтобы получить более однородные данные. Мы мало что можем сделать с людьми, которые носят головные уборы, пьют или что-то держат перед своими лицами, но имейте в виду, что они представляют собой трудности, с которыми необходимо справиться.

Если мы хотим найти более интересные кластеры, а не просто один большой кластер, нам нужно уменьшить значение **eps**, задать его в интервале между 15 и 0.5 (значение по умолчанию). Давайте посмотрим, к каким результатам приведут различные значения **eps**:

```
In[77]:
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Полученные кластеры: {}".format(np.unique(labels)))
    print("Размеры кластеров: {}".format(np.bincount(labels + 1)))
```

```
Out[78]:
eps=1
Полученные кластеры: [-1]
Размеры кластеров: [2063]

eps=3
Полученные кластеры: [-1]
Размеры кластеров: [2063]

eps=5
Полученные кластеры: [-1]
Размеры кластеров: [2063]

eps=7
Полученные кластеры: [-1 0 1 2 3 4 5 6 7 8 9 10 11 12]
Размеры кластеров: [2006 4 6 6 6 9 3 3 4 3 3 3 3 4]

eps=9
Полученные кластеры: [-1 0 1 2]
Размеры кластеров: [1269 788 3 3]

eps=11
Полученные кластеры: [-1 0]
Размеры кластеров: [ 430 1633]

eps=13
Полученные кластеры: [-1 0]
Размеры кластеров: [ 112 1951]
```

При низких значениях **eps** все точки помечаются как шум. Для **eps=7** мы получаем большое количество шумовых точек и множество мелких кластеров. Для **eps=9** мы все еще получаем большое количество шумовых точек, но при этом у нас появляется большой кластер и несколько

кластеров меньшего размера. Начиная с **eps=11**, мы получаем лишь один большой кластер и шум.

Что интересно отметить, во всех случаях мы не смогли получить больше одного крупного кластера. В лучшем случае мы получаем один крупный кластер, который содержит большую часть точек, а также несколько более мелких кластеров. Это указывает не на присутствие в данных двух или трех различных типов изображений лиц, которые очень легко распознать, а скорее на примерно одинаковую степень сходства всех изображений, попавших в крупный кластер, между собой (или на примерно одинаковую степень отличия изображений, попавших в крупный кластер, от остальных).

Результаты для **eps=7** выглядят наиболее интересно, здесь мы получаем большое количество маленьких кластеров. Мы можем исследовать результаты этой кластеризации более подробно, визуализировав все точки данных для всех 13 кластеров меньшего размера (рис 3-42):

```
In[78]:
dbscan = DBSCAN(min_samples=3, eps=7)
labels = dbscan.fit_predict(X_pca)

for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images = np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4),
                             subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):

        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])
```



**Рис. 3.42** Кластеры, найденные с помощью алгоритма DBSCAN при  $\epsilon=7$

Некоторые кластеры соответствуют людям с очень отчетливыми изображениями лиц (в рамках этого набора данных), например, Ариэлю Шарону или Дзюньитиро Коидзуми. В пределах каждого кластера также фиксируется поворот и выражение лица. Некоторые кластеры содержат лица разных людей, но при этом все лица имеют схожий поворот и выражение лица.

На этом наш анализ набора изображений лиц с помощью алгоритма DBSCAN завершается. Как вы можете видеть, в данном случае мы осуществили ручной анализ данных, который сильно отличается от

метода автоматического поиска, примененного нами для машинного обучения с учителем на основе  $R^2$  или правильности.

Теперь давайте перейдем к применению алгоритмов  $k$ -средних и агломеративной кластеризации.

Анализ набора изображений лиц с помощью алгоритма  $k$ -средних

Мы увидели, что с помощью алгоритма DBSCAN невозможно получить больше одного большого кластера. Алгоритмы агломеративной кластеризации и  $k$ -средних имеют гораздо больше шансов сформировать кластеры одинакового размера, но нам задать нужное количество кластеров. Мы могли бы задать количество кластеров равным известному количеству людей в наборе данных, хотя очень маловероятно, что алгоритм неконтролируемой кластеризации сможет восстановить их. Вместо этого мы можем начать с небольшого количества кластеров (например, с 10), которое, возможно, позволит нам проанализировать каждый кластер:

In[79]:

```
# извлекаем кластеры с помощью k-средних
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Размеры кластеров k-средние: {}".format(np.bincount(labels_km)))
```

Out[79]:

```
Cluster sizes k-средние: [269 128 170 186 386 222 237 64 253 148]
```

Видно, что алгоритм кластеризации  $k$ -средних распределил данные по кластерам, размер которых варьирует от 64 до 386 изображений. Это сильно отличается от результата алгоритма DBSCAN.

Далее мы можем проанализировать результаты алгоритма  $k$ -средних, визуализировав центры кластеров (рис. 3.43). Поскольку мы кластеризовали данные, полученные с помощью PCA, нам нужно повернуть центры кластеров обратно в исходное пространство, чтобы визуализировать их, используя `pca.inverse_transform`:

In[80]:

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()},
                        figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape),
              vmin=0, vmax=1)
```



**Рис. 3.43** Центры кластеров, найденные с помощью  $k$ -средних, количество кластеров равно 10

Центры кластеров, найденные с помощью алгоритма  $k$ -средних, представляют собой сильно сглаженные лица. Это неудивительно, учитывая, что каждый центр – это усредненное изображение лиц, попавших в кластер. Использование уменьшенного с помощью PCA представления данных усиливает сглаженность изображений (сравните с реконструкциями лиц на рис. 3.11, когда использовалось 100 компонент). Похоже, что кластеризация выделила разные повороты лиц, разные выражения лиц (кажется, третий центр кластера показывает улыбающееся лицо), а также наличие воротника у рубашки (смотрите предпоследний центр кластера).

Для более детального просмотра на рис. 3.44 мы выведем для каждого центра кластера пять наиболее типичных изображений в кластере (изображения, присвоенные кластеру и находящиеся ближе всего к центру кластера) и пять самых нетипичных изображений в кластере (изображения, присвоенные кластеру и находящиеся дальше всего от центра кластера):

```
In[81]:
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people,
                                y_people, people.target_names)
```





**Рис. 3.44** Примеры изображений для каждого кластера, найденного с помощью алгоритма  $k$ -средних – центры кластеров находятся слева, затем следуют пять точек, максимально близко расположенных к центру кластера, и пять точек, максимально удаленных от центра кластера

Рис. 3-44 подтверждает наш вывод об улыбающихся лицах в третьем кластере и важности поворота лица для других кластеров. Однако «нетипичные» точки сильно отличаются от центров кластеров и их назначение кластеру кажется несколько произвольным. Это может быть обусловлено тем фактом, что алгоритм  $k$ -средних разбивает все точки данных на группы и в отличие от алгоритма DBSCAN в нем отсутствует понятие «шумовая точка». При использовании большего количества кластеров алгоритм сможет найти более тонкие различия. Однако увеличения числа кластеров сделает ручной анализ еще более трудоемким.

## Анализ набора изображений лиц с помощью алгоритма агломеративной кластеризации

Теперь давайте посмотрим на результаты агломеративной кластеризации:

In[82]:

```
# извлекаем кластеры с помощью агломеративной кластеризации по методу Варда
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("Размеры кластеров для агломеративной кластеризации: {}".format(
    np.bincount(labels_agg)))
```

Out[82]:

Размеры кластеров для агломеративной кластеризации: [255 623 86 102 122 199 265 26 230 155]

Видно, что алгоритм агломеративной кластеризации распределил данные по кластерам, размер которых варьирует от 26 до 623 изображений. В отличие от алгоритма  $k$ -средних размеры кластеров варьируют сильнее, но при этом значительно меньше, если сравнивать их с размерами кластеров, полученными с помощью алгоритма DBSCAN.

Мы можем вычислить ARI, чтобы оценить сходство результатов, полученных с помощью агломеративной кластеризации и кластеризации  $k$ -средних:

In[83]:

```
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

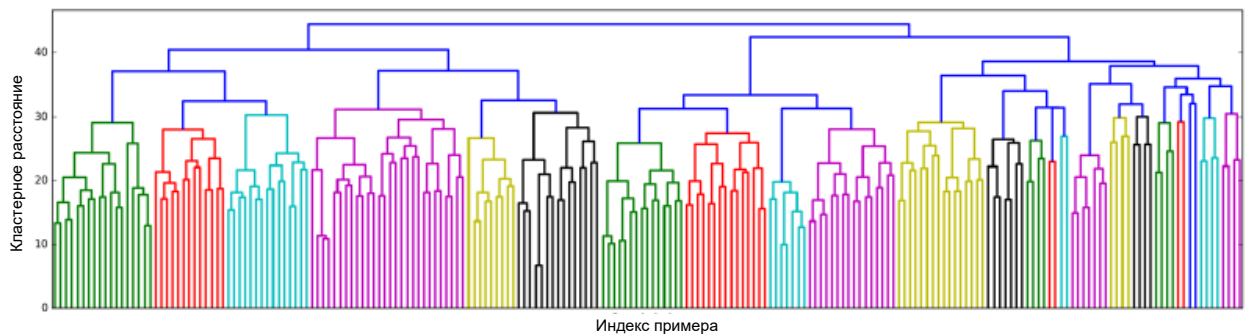
Out[83]:

ARI: 0.13

Значение ARI, равное всего лишь 0.13, означает, что кластеризации `labels_agg` и `labels_km` имеют мало общего между собой. Это неудивительно, учитывая тот факт, что в алгоритме  $k$ -средних точки, удаленные от центров кластеров, по-видимому, имеют мало общего между собой.

Далее мы можем построить дендрограммы (рис. 3.45). Мы ограничим глубину дерева, поскольку ветвление по 2063 отдельным точкам данных приведет к построению нечитаемого очень плотного графика:

```
In[84]:
linkage_array = ward(X_pca)
# строим дендрограмму для linkage_array
# содержащего расстояния между кластерами
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True)
plt.xlabel("Индекс примера")
plt.ylabel("Кластерное расстояние")
```



**Рис. 3.45** Дендрограмма агломеративной кластеризации для набора изображений лиц

Построив 10 кластеров, мы срезаем дерево в самой верхней части, в которой расположены 10 вертикальных линий. На дендрограмме, построенной для синтетических данных (рис. 3.36), вы, проанализировав длину ветвей, могли прийти к выводу, что два или три кластера могут описать данные надлежащим образом. Что касается набора изображений лиц, здесь, по-видимому, не будет какого-то очевидного числа. Некоторые ветви представляют собой более четко обособленные группы, но, по-видимому, это никак не связано с оптимальным количеством кластеров. Это неудивительно, учитывая результаты алгоритма DBSCAN, который попытался сгруппировать все кластеры вместе.

Давайте визуализируем эти 10 кластеров, как мы это делали ранее для алгоритма  $k$ -средних (рис. 3.46). Обратите внимание, что в агломеративной кластеризации не существует такого понятия, как центр кластера (хотя мы могли бы вычислить среднее значение) и мы просто показываем первые несколько точек в каждом кластере. Кроме того, мы покажем количество точек в каждом кластере, выведя его слева от первого изображения каждого ряда:

```
In[85]:
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
```



**Рис. 3.46** Изображения, случайно выбранные из кластеров с помощью вышеприведенного программного кода (каждый ряд соответствует одному кластеру, число слева указывает количество изображений в каждом кластере)

Хотя некоторые кластеры, похоже, имеют содержательную интерпретацию, многие из них слишком велики, чтобы быть на самом

деле однородными. Чтобы получить более однородные кластеры, мы можем запустить алгоритм снова, на этот раз с 40 кластерами, и выбрать некоторые особо интересные кластеры (рис. 3.47):

```
In[86]:
# извлекаем кластеры с помощью агломеративной кластеризации по методу Варда
aggglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = aggglomerative.fit_predict(X_pca)
print("размеры кластеров для аглом. кластеризации: {}".format(np.bincount(labels_agg)))

n_clusters = 40
for cluster in [10, 13, 19, 22, 36]: # вручную выбранные "интересные" кластеры
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()},
                             figsize=(15, 8))
    cluster_size = np.sum(mask)
    axes[0].set_ylabel("#{:}: {}".format(cluster, cluster_size))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                     labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
    for i in range(cluster_size, 15):
        axes[i].set_visible(False)

Out[86]:
размеры кластеров для аглом. кластеризации:
[ 58 80 79 40 222 50 55 78 172 28 26 34 14 11 60 66 152 27
 47 31 54 5 8 56 3 5 8 18 22 82 37 89 28 24 41 40
 21 10 113 69]
```



**Рис. 3.47** Изображения из отобранных кластеров, которые были найдены алгоритмом агломеративной кластеризации (количество кластеров равно 40), текста слева показывает индекс кластера и общее количество точек в кластере



В данном случае кластеризация, похоже, выделила «улыбчивых чернокожих», «любителей рубашек с воротником», «улыбчивых женщин», «Хусейнов» и «высоколобых». Кроме того, мы могли бы найти аналогичные кластеры с помощью дендрограммы, если бы проводили более детальный анализ.

## Выводы по методам кластеризации

Этот раздел показал, что применение алгоритмов кластеризации с последующей оценкой их результатов является сложной и, как правило, очень полезной процедурой на исследовательском этапе анализа данных. Мы рассмотрели три алгоритма кластеризации:  $k$ -средние, DBSCAN и агломеративную кластеризацию. Все три алгоритма имеют возможность настраивать гранулярность кластеризации. Алгоритмы  $k$ -средних и агломеративной кластеризации позволяют задать нужное количество кластеров, в то время как DBSCAN позволяет задать близость между точками с помощью параметра `eps`, который косвенно влияет на размер кластера. Все три метода могут быть использованы на больших реальных наборах данных, имеют относительно простую интерпретацию и допускают разбиение на большое количество кластеров.

Каждый из алгоритмов имеет свои преимущества. Алгоритм  $k$ -средних позволяет описывать кластеры с помощью их средних значений. Кроме того, его можно рассматривать как декомпозиционный метод, в котором каждая точка данных представлена центром кластера. DBSCAN позволяет определить «шумовые точки», которые не присвоены ни одному кластеру, и он может помочь автоматически определить количество кластеров. В отличие от двух остальных методов он допускает наличие кластеров сложной формы, как мы уже видели на примере набора данных `two_moons`. Иногда DBSCAN выделяет кластеры, сильно отличающиеся по своим размерам, что может быть как недостатком, так и преимуществом этого алгоритма. Агломеративная кластеризация позволяет построить исчерпывающую иерархию возможных разбиений данных, которую можно легко исследовать с помощью дендрограмм.

## Выводы и перспективы

Эта глава познакомила вас с целым рядом алгоритмов неконтролируемого обучения, которые можно применить для разведочного анализа данных и предварительной обработки. Наличие адекватных данных часто имеет решающее значение для успешного применения алгоритмов контролируемого или неконтролируемого

обучения, поэтому методы предварительной обработки и декомпозиционные методы играют важную роль в подготовке данных.

Декомпозиционные методы, множественное обучение и кластеризация являются необходимыми инструментами для дальнейшего понимания ваших данных, и могут быть теми единственными средствами, которые придадут смысл вашим данным при отсутствии контрольной информации. Даже при наличии контрольной информации инструменты разведочного анализа имеют важное значение с точки зрения лучшего понимания свойств данных. Зачастую полезность алгоритмов неконтролируемого обучения трудно оценить, однако это не должно удерживать вас от использования этих алгоритмов с целью получения более глубокого представления о данных. Включив в свой арсенал эти методы, вы теперь экипированы всеми необходимыми алгоритмами, которыми специалисты по машинному обучению пользуются ежедневно.

Мы рекомендуем вам применять кластеризацию и декомпозиционные методы как к двумерным синтетическим данным, так и к реальным наборам данных, включенным в `scikit-learn`, типа наборов `digits`, `iris` и `cancer`.



## Выводы по интерфейсу моделей

Давайте дадим краткий обзор API, который мы рассматривали в главах 2 и 3. Все алгоритмы в **scikit-learn**, будь то предварительная обработка, алгоритмы машинного обучения с учителем или машинное обучение без учителя, реализованы в виде классов. Эти классы в **scikit-learn** называются *estimators* (моделями). Чтобы применить алгоритм, вы сперва должны создать экземпляр конкретного класса:

```
In[87]:
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

Этот класс-модель содержит алгоритм, а также записывает модель, обученную на данных с помощью этого алгоритма.

При построении модели вы должны задать те ли иные ее параметры. Эти параметры включают в себя регуляризацию, настройку сложности, количество выделяемых кластеров и т.д. Все модели имеют метод **fit**, который используется для построения модели. Метод **fit** всегда требует в качестве первого аргумента данные **X**, представленных в виде массива NumPy или разреженной матрицы SciPy, в которой каждая строка представляет собой отдельную точку данных. Предполагается, что данные **X** всегда записаны в виде массива NumPy или разреженной матрицы SciPy, которая содержит непрерывные элементы (числа с плавающей точкой). Алгоритмы машинного обучения с учителем дополнительно требуют аргумент **y**, который является одномерным массивом NumPy, содержащим целевые значения для регрессии или классификации (т.е. уже известные метки или ответы).

Есть два основных способа применить обученную модель в **scikit-learn**. Чтобы создать прогноз в виде нового ответа типа **y**, вы должны использовать метод **predict**. Для создания нового представления входных данных **X** используется метод **transform**. Таблица 3.1 обобщает случаи использования методов **predict** и **transform**.

Таблица 3.1 Сводка по API в scikit-learn

<b>estimator.fit(x_train, [y_train])</b>	
<b>estimator.predict(X_text)</b>	<b>estimator.transform(X_test)</b>
Классификация	Предварительная обработка
Регрессия	Сокращение размерности
Кластеризация	Выделение характеристик
	Отбор характеристик

## ГЛАВА 4. ТИПЫ ДАННЫХ И КОНСТРУИРОВАНИЕ ПРИЗНАКОВ

До сих пор мы считали, что наши данные представлены в виде двумерного массива чисел с плавающей точкой, в котором каждый столбец является *непрерывным признаком* (*continuous feature*), описывающим точки данных. Однако во многих случаях это не так. Наиболее распространенным типом признаков являются *категориальные признаки* (*categorical features*). Они еще известны как *дискретные признаки* (*discrete features*), поскольку обычно не имеют числовых значений. Различие между категориальными и непрерывными признаками аналогично различию между классификацией и регрессией, но только с точки зрения входных данных, а не ответов. Примерами непрерывных признаков, которые мы уже рассматривали, являются яркость пикселей и измерения характеристик ирисов. Примерами категориальных признаков являются бренд продукта, цвет продукта или отдел, в котором он продается (книги, одежда, оборудование). Все они являются свойствами, которые могут описать продукт, но при этом не измеряются в непрерывной шкале. Продукт продается либо в отделе одежды, либо в отделе книг. Не существует золотой середины между книгами и одеждой и нет естественного способа упорядочить различные категории (книги не могут быть больше или меньше одежды, оборудование необязательно должно располагаться между книгами и одеждой и т.д.).

Независимо от типов признаков, которыми будут представлены ваши данные, способ их подготовки имеет огромное влияние на качество работы моделей машинного обучения. Мы уже убедились в главах 2 и 3, что масштабирование данных имеет важное значение. Другими словами, если вы не отмасштабируете данные (скажем, к единичной дисперсии), результаты моделирования будут зависеть от единиц измерения признаков. Кроме того, в главе 2 мы уже видели, что улучшить результаты может обогащение данных дополнительными признаками, например, можно добавить взаимодействия (произведения) признаков или полиномы.

Вопрос оптимальной подготовки данных для конкретного прикладного применения известен под названием *feature engineering* (*конструирование признаков*) и является одной из главных задач для специалистов по машинному обучению, пытающихся решить реальные проблемы.

В этой главе мы сначала рассмотрим важные и наиболее распространенные случаи использования категориальных признаков, а

затем приведем некоторые примеры полезных преобразований для конкретных сочетаний признаков и моделей.

## Категориальные переменные

В качестве примера мы будем использовать данные о доходах взрослого населения США, полученные из переписи населения 1994 года. Задача, которую мы будем решать при работе с набором данных `adult`, заключается в том, чтобы спрогнозировать наличие у работника дохода более 50000 \$ и менее 50000 \$. Признаками этого набора данных являются возраст работника, тип занятости (частное предприятие, наемный работник, госслужащий и т.д.), образование, пол, продолжительность рабочей недели, род занятий и многое другое. Таблица 4.1 показывает первые несколько записей в наборе данных.

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K
5	37	Private	Masters	Female	40	Exec-managerial	<=50K
6	49	Private	9th	Female	16	Other-service	<=50K
7	52	Self-emp-not-inc	HS-grad	Male	45	Exec-managerial	>50K
8	31	Private	Masters	Female	50	Prof-specialty	>50K
9	42	Private	Bachelors	Male	40	Exec-managerial	>50K
10	37	Private	Some-college	Male	80	Exec-managerial	>50K

**Таблица 4.1** Первые несколько записей набора данных `adult`

Задача сформулирована в виде классификационной задачи с двумя классами `доход<=50 тыс.` и `доход>50 тыс.`. Можно было бы также спрогнозировать точное значение дохода и это уже была бы регрессионная задача. Однако это было бы гораздо более сложной задачей, а данное разбиение дохода долларов интересно само по себе.

В этом наборе данных `age` и `hours-per-week` являются непрерывными признаками, обработка которых нам уже знакома. Однако признаки `workclass`, `education`, `sex` и `occupation` являются категориальными. Вместо диапазона все они имеют фиксированный список возможных значений и обозначают качественный признак, а не непрерывный.

Для начала предположим, что мы хотим обучить классификатор логистической регрессии на этих данных. Из главы 2 мы знаем, что логистическая регрессия делает прогнозы  $\hat{y}$ , используя следующую формулу:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b > 0$$

где  $w[i]$  и  $b$  – коэффициенты, вычисленные на обучающей выборке, а  $x[i]$  – входные признаки. Эта формула имеет смысл, когда  $x[i]$  являются числовыми значениями, но не тогда, когда  $x[2]$  соответствует "Masters" или "Bachelors". Очевидно, что нам нужно подготовить данные таким способом, чтобы можно было применить логистическую регрессию. В следующем разделе мы расскажем, как можно решить эту проблему.

## Прямое кодирование (дамми-переменные)

На сегодняшний момент наиболее распространенным способом представления категориальных переменных является *прямое кодирование* или, если перевести дословно, *кодирование с одним горячим состоянием* (*one-hot-encoding* или *one-out-of-N encoding*). Идея, лежащая в основе прямого кодирования, заключается в том, чтобы заменить категориальную переменную одной или несколькими новыми признаками, которые могут принимать значения 0 и 1. Значения 0 и 1 придают смысл формуле линейной бинарной классификации (а также всем остальным моделям в `scikit-learn`) и с помощью дамми-переменных мы можем выразить любое количество категорий, вводя по одному новому признаку для каждой категории.

Скажем, признак `workclass` имеет возможные значения "Government Employee", "Private Employee", "Self Employed" и "Self Employed Incorporated". Для того, чтобы закодировать эти четыре возможных значения, мы создаем четыре новых характеристики "Government Employee", "Private Employee", "Self Employed" и "Self Employed Incorporated". Характеристика равна 1, если `workclass` принимает соответствующее значение, или равна 0 в противном случае, поэтому для каждой точки данных только *одна* из четырех новых характеристик будет равна 1. Вот почему данная операция называется *кодированием с одним горячим (активным) состоянием*.

Этот принцип показан в таблице 4.2. Один признак кодируется с помощью четырех новых характеристик. Включив эту информацию в модель машинного обучения, мы не используем исходный признак `workclass`, а работаем только с этими четырьмя характеристиками 0-1.

workclass	Government	Employee	Private Employee	Self Employed	Self Employed Incorporated
Government Employee	1		0	0	0
Private Employee	0		1	0	0
Self Employed	0		0	1	0
Self Employed Incorporated	0		0	0	1

**Таблица 4.2** Прямое кодирование признака workclass



Прямое кодирование, которое мы используем, довольно схоже с дамми-кодированием, применяемым в статистике, но не идентично ему. В целях упрощения мы кодируем категории переменной с помощью бинарных признаков. В статистике категориальную переменную, принимающую  $k$  различных возможных значений (категорий), общепринято кодировать с помощью  $k-1$  признаков, при этом для последней категории все признаки будут иметь нулевые значения. Это делается для упрощения анализа (говоря более техническим языком, это позволяет избежать получения матрицы неполного ранга).

Существует два способа выполнить прямое кодирование категориальных переменных, используя либо **pandas**, либо **scikit-learn**. На момент написания книги использование **pandas** выглядело немного проще, поэтому давайте пойдем по этому пути. Сначала с помощью **pandas** загрузим данные, записанные в CSV-файле:

```
In[2]:
import pandas as pd
# Файл не содержит заголовков столбцов, поэтому мы передаем header=None
# и записываем имена столбцов прямо в "names"
data = pd.read_csv(
    "C:/Data/adult.data", header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education', 'education-num',
           'marital-status', 'occupation', 'relationship', 'race', 'gender',
           'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
           'income'])
# В целях упрощения мы выберем лишь некоторые столбцы
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
# IPython.display позволяет вывести красивый вывод, отформатированный в Jupyter notebook
display(data.head())
```

Таблица 4.3 показывает результат.

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

**Таблица 4.3** Первые пять строк набора данных adult

### Проверка категориальных данных, закодированных в виде строк

Прочитав набор данных, аналогичный приведенному выше, как правило, неплохо было бы сперва проверить, содержит ли столбец на самом деле осмысленные категориальные данные. При работе с данными, которые были введены людьми (например, пользователями на сайте), получить фиксированный набор категорий невозможно, наличие различных вариантов написания слов может потребовать предварительной обработки. Например, некоторые могут определить свой пол как «мужской», а некоторые просто напишут «мужчина» и нам, возможно, потребуется поместить эти два варианта в одну и ту же категорию. Хороший способ проверить содержимое столбца – воспользоваться функцией `value_counts` для пандасовского типа данных `Series` (каждый столбец `DataFrame` является структурой `Series`), чтобы посмотреть, что представляют из себя уникальные значения и как часто они встречаются:

```
In[3]:
print(data.gender.value_counts())
```

```
Out[3]:
Male 21790
Female 10771
Name: gender, dtype: int64
```

Видно, что в этом наборе данных пол имеет строго два значения, `Male` и `Female`, то есть данные уже находятся в подходящем формате, чтобы записать их, используя прямое кодирование. В реальном примере вы должны просмотреть все столбцы и проверить их значения. Мы пропустим этот момент для краткости.

Библиотека `pandas` предлагает очень простой способ кодирования данных с помощью функции `get_dummies`. Функция `get_dummies` автоматически преобразует все столбцы, которые содержат объектные типы (например, строки) или являются категориальными данными (речь идет о специальном понятии `pandas`, о котором мы еще не говорили):

```
In[4]:
print("Исходные признаки:\n", list(data.columns), "\n")
data_dummies = pd.get_dummies(data)
print("Признаки после get_dummies:\n", list(data_dummies.columns))
```

Out[4]:

Исходные признаки:

```
['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation',  
'income']
```

Исходные признаки после get\_dummies:

```
['age', 'hours-per-week', 'workclass_?', 'workclass_Federal-gov',  
'workclass_Local-gov', 'workclass_Never-worked', 'workclass_Private',  
'workclass_Self-emp-inc', 'workclass_Self-emp-not-inc',  
'workclass_State-gov', 'workclass_Without-pay', 'education_10th',  
'education_11th', 'education_12th', 'education_1st-4th',  
...  
'education_Preschool', 'education_Prof-school', 'education_Some-college',  
'gender_Female', 'gender_Male', 'occupation_?',  
'occupation_Adm-clerical', 'occupation_Armed-Forces',  
'occupation_Craft-repair', 'occupation_Exec-managerial',  
'occupation_Farming-fishing', 'occupation_Handlers-cleaners',  
...  
'occupation_Tech-support', 'occupation_Transport-moving',  
'income_<=50K', 'income_>50K']
```

Видно, что непрерывные признаки `age` и `hours-per-week` остались неизменными, тогда как для каждого возможного значения категориального признака были созданы новые характеристики.

In[5]:

```
data_dummies.head()
```

Out[5]:

	age	hours- per- week	workclass_?	workclass_ Federal-gov	workclass_ Local-gov	...	occupation_ Tech- support	occupation_ Transport- moving	income_ <=50K	income_ >50K
0	39	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
1	50	13	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
2	38	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
3	53	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
4	28	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0

5 rows × 46 columns

Теперь мы можем воспользоваться атрибутом `values`, что преобразовать пандасовский дата-фрейм `data_dummies` в массив NumPy, а затем обучить на его основе модель машинного обучения. Перед построением модели убедитесь в том, что зависимая переменная (которая теперь кодируется в двух столбцах `income`) отделена от данных. Включение зависимой переменной или некоторых признаков, являющимися производными от зависимой переменной, в пространство входных признаков является очень распространенной ошибкой при построении моделей машинного обучения с учителем.



Будьте осторожны: индексация столбцов в `pandas` включает конец диапазона, поэтому `'age': 'occupation_Transport-moving'` включает в себя `occupation_Transport-moving`. Данная операция отличается от нарезки массива NumPy, в котором конец диапазона не включается: например, `np.arange(11)[0:10]` не включает в себя элемент с индексом 10.



В данном случае мы извлечем только те столбцы, которые содержат входные признаки, то есть, все столбцы от `age` до `occupation_Transport`. Этот диапазон содержит все входные признаки, при этом зависимая переменная в него не включена:

```
In[6]:
# Берем только те столбцы, которые содержат признаки,
# то есть все столбцы, начиная с 'age' и заканчивая 'occupation_Transport-moving'
# Этот диапазон содержит все признаки, кроме целевой переменной
features = data_dummies.ix[:, 'age':'occupation_Transport-moving']
# Извлекаем массивы NumPy
X = features.values
y = data_dummies['income_>50K'].values
print("форма массива X: {} форма массива y: {}".format(X.shape, y.shape))
```

```
Out[6]:
форма массива X: (32561, 44) форма массива y: (32561,)
```

Теперь данные представлены в том формате, который `scikit-learn` умеет обрабатывать, и мы можем продолжить построение модели в обычном режиме:

```
In[7]:
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Правильность на тестовом наборе: {:.2f}".format(logreg.score(X_test, y_test)))
```

```
Out[7]:
Правильность на тестовом наборе: 0.81
```



В этом примере мы вызвали функцию `get_dummies` и передали ей в качестве аргумента пандасовский `DataFrame`, содержащий как обучающие, так и тестовые данные. Это важно с точки зрения одинакового представления значений категориальных признаков в обучающем и тестовом наборах.

Представьте, что у нас обучающий и тестовые наборы записаны в двух разных пандасовских дата-фреймах. Если в тестовом наборе у признака `workclass` будет отсутствовать значение "Private Employee", `pandas` предположит, что существуют только три возможных значения этого признака и создаст лишь три новых дамми-переменных. Теперь у признака `workclass` разное количество дамми-переменных в обучающем и тестовом наборах и мы уже больше не можем применить к тестовому набору модель, построенную на обучающей выборке. Возьмем ситуацию еще хуже, представьте себе, что признак `workclass` принимает значения "Government Employee" и "Private Employee" в обучающем наборе и "Self Employed" и "Self Employed Incorporated" в тестовом наборе. В обоих случаях `pandas` создаст две новые дамми-переменные, таким образом перекодированные дата-фреймы будут иметь одинаковое количество дамми-переменных. Однако эти две дамми-переменные имеют совершенно различный смысл в обучающем и тестовом наборах. Столбец, соответствующий

значению "Government Employee" в обучающем наборе, будет закодирован как "Self Employed" в тестовом наборе.

Модель машинного обучения, построенная на этих данных, будет работать очень плохо, потому что исходит из того, что столбцы соответствуют одному и тому же возможному значению категориального признака (ведь они имеют одинаковое расположение в массивах), тогда как на самом деле они соответствуют совершенно разным значениям. Чтобы это исправить, вызовите функцию `get_dummies` и передайте ей в качестве аргумента дата-фрейм, содержащий как обучающие, так и тестовые данные, или уже после вызова `get_dummies` убедитесь в том, что имена столбцов одинаковы для обучающего и тестовых наборов и имеют один и тот же смысл.

## Числа можно закодировать в виде категорий

В примере с набором данных `adult` категориальные переменные были закодированы в виде строк. С одной стороны, это чревато орфографическими ошибками, но, с другой стороны, это позволяет четко отнести признак к категориальной переменной. Часто для удобства хранения или из-за способа сбора данных категориальные переменные кодируются в виде целых чисел. Например, представьте, что данные переписи, представленные в наборе `adult`, были собраны с помощью анкеты, и ответы, касающиеся типа занятости (`workclass`), были записаны как 0 (первый пункт отмечен галочкой), 1 (второй пункт отмечен галочкой), 2 (третий пункт отмечен галочкой) и так далее. Теперь столбец будет содержать цифры от 0 до 8, а не строки типа "Private", и если кто-то посмотрит на таблицу, представляющую набор данных, он не сможет с уверенностью отличить непрерывную переменную от категориальной. Хотя мы знаем, что цифры указывают на тип занятости, ясно, что все они соответствуют совершенно различным состояниям и их не следует моделировать с помощью одной непрерывной переменной.



Категориальные признаки часто кодируются с помощью целых чисел. Тот факт, что для кодировки используются числа, вовсе не означает, что они должны обрабатываться как непрерывные признаки. Не всегда ясно, следует ли обрабатывать целочисленные значения признаков как непрерывные или дискретные (а также преобразованные с помощью прямого кодирования). Если кодируемые значения не упорядочены (как в примере с `workclass`), признак должен рассматриваться как дискретный. В остальных случаях, например, когда признак представляет собой пятизвездочный рейтинг, выбор оптимальной схемы кодирования зависит от конкретной задачи и данных, а также используемого алгоритма машинного обучения.

Функция `get_dummies` в `pandas` обрабатывает все числа как непрерывные значения и не будет создавать дамми-переменные для них. Чтобы обойти эту проблему, вы можете либо воспользоваться `OneHotEncoder` в `scikit-learn` (указав, какие переменные являются непрерывными, а какие – дискретными), либо преобразовать столбцы с числами, содержащиеся в дата-фрейме, в строки. Чтобы проиллюстрировать это, давайте создадим объект `DataFrame` с двумя столбцами, один из которых содержит строки, а другой – целые числа:

```
In[8]:
# создаем дата-фрейм с признаком, который принимает целочисленные значения,
# и категориальным признаком, у которой значения являются строками
demo_df = pd.DataFrame({'Целочисленный признак': [0, 1, 2, 1],
                        'Категориальный признак': ['socks', 'fox', 'socks', 'box']})
display(demo_df)
```

Таблица 4.4 показывает результат.

	Категориальный признак	Целочисленный признак
0	socks	0
1	fox	1
2	socks	2
3	box	1

**Таблица 4.4** Дата-фрейм, содержащий категориальный строковый признак и целочисленный признак

Функция `get_dummies` закодирует лишь строковый признак, тогда как целочисленный признак оставит без изменений, как это видно в таблице 4.5:

```
In[9]:
pd.get_dummies(demo_df)
```

	Целочисленный признак	Категориальный признак_box	Категориальный признак_fox	Категориальный признак_socks
0	0	0.0	0.0	1.0
1	1	0.0	1.0	0.0
2	2	0.0	0.0	1.0
3	1	1.0	0.0	0.0

**Таблица 4.5** Данные таблицы 4.4, преобразованные с помощью прямого кодирования, целочисленный признак остался без изменений

Если вы хотите создать дамми-переменные для столбца «Целочисленный признак», вы можете явно указать столбцы, которые

нужно закодировать, с помощью параметра `columns`. И тогда оба признака будут обработаны как категориальные переменные (см. табл. 4.6):

```
In[10]:
demo_df['Целочисленный признак'] = demo_df['Целочисленный признак'].astype(str)
pd.get_dummies(demo_df, columns=['Целочисленный признак', 'Категориальный признак'])
```

	Целочисленный признак_0	Целочисленный признак_1	Целочисленный признак_2	Категориальный признак_box	Категориальный признак_fox	Категориальный признак_socks
0	1.0	0.0	0.0	0.0	0.0	1.0
1	0.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	0.0	1.0
3	0.0	1.0	0.0	1.0	0.0	0.0

**Таблица 4.6** Данные таблицы 4.4, преобразованные с помощью прямого кодирования, закодированы целочисленный и строковый признаки

## Биннинг, дискретизация, линейные модели и деревья

Оптимальный способ представления данных зависит не только от содержательного смысла данных, но и от вида используемой модели. Линейные модели и модели на основе дерева (например, деревья решений, градиентный бустинг деревьев решений и случайный лес), представляющие собой две большие и наиболее часто используемые группы методов, сильно отличаются друг от друга с точки зрения обработки признаков различных типов. Давайте вернемся к набору данных `wave`, который мы использовали для регрессионного анализа в главе 2. Он имеет лишь один входной признак. Ниже приводится сравнение результатов модели линейной регрессии и дерева регрессии для этого набора данных (см. рис. 4.1):

```
In[11]:
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

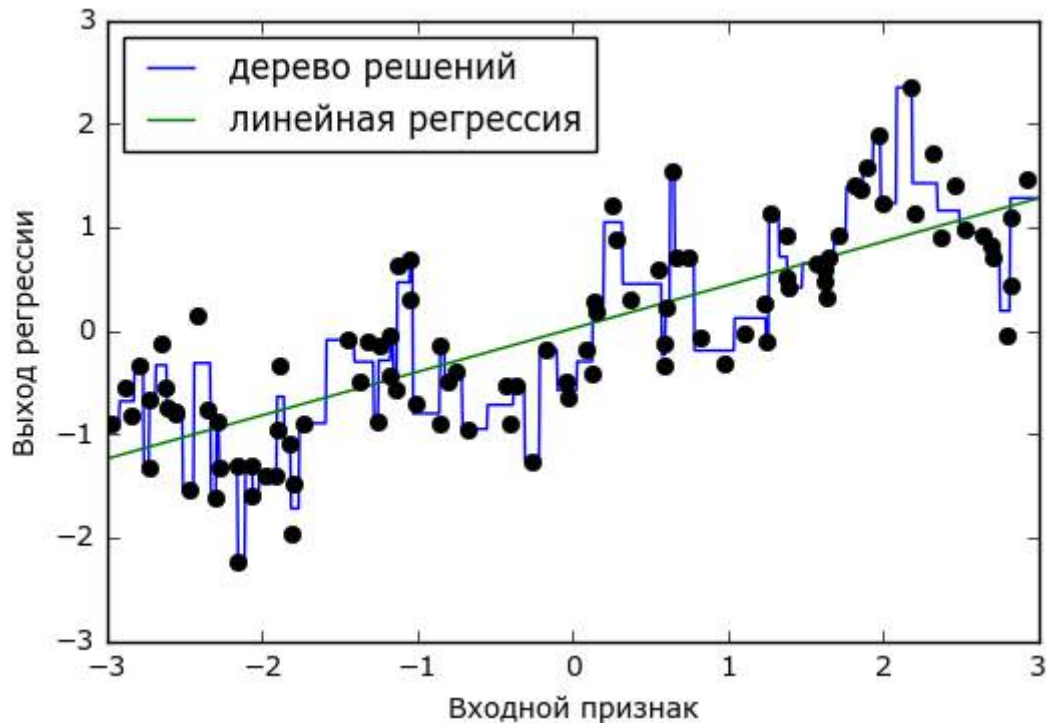
reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="дерево решений")

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="линейная регрессия")

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Выход регрессии")
plt.xlabel("Входной признак")
plt.legend(loc="best")
```

Как вам известно, линейные модели могут моделировать только линейные зависимости, которые представляют собой линии в случае одного признака. Дерево решений может построить гораздо более сложную модель данных. Результаты сильно зависят от представления

данных. Одним из способов повысить прогнозную силу линейных моделей при работе с непрерывными данными является *биннинг* характеристик (*binning*), также известный как *дискретизация* (*discretization*), который разбивает исходный признак на несколько категорий.



**Рис. 4.1** Сравнение результатов модели линейной регрессии и дерева регрессии для набора данных wave

Представим, что диапазон значений входного признака (в данном случае от -3 до 3) разбит на определенное количество *категорий* или *бинов* (*bins*), допустим, на 10 категорий. Точка данных будет представлена категорией, в которую она попадает. Сначала мы должны задать категории. В данном случае мы зададим 10 категорий, равномерно распределенных между -3 и 3. Для этого мы используем функцию `np.linspace`, создаем 11 элементов, которые дадут 10 категорий – интервалов, ограниченных двумя границами:

```
In[12]:
bins = np.linspace(-3, 3, 11)
print("категории: {}".format(bins))

Out[12]:
категории: [-3.  -2.4  -1.8  -1.2  -0.6  0.   0.6  1.2  1.8  2.4  3. ]
```

При этом первая категория содержит все точки данных со значениями признака от -3 до -2.68, вторая категория содержит все точки со значениями признака от -2.68 до -2.37 и так далее.

Далее мы записываем для каждой точки данных категорию, в которую она попадает. Это можно легко вычислить с помощью функции `np.digitize`:

```
In[13]:
which_bin = np.digitize(X, bins=bins)
print("\nТочки данных:\n", X[:5])
print("\nКатегории для точек данных:\n", which_bin[:5])
```

```
Out[13]:
Точки данных:
[[ -0.753]
 [  2.704]
 [  1.392]
 [  0.592]
 [ -2.064]]
```

```
Категории для точек данных:
[[ 4]
 [10]
 [ 8]
 [ 6]
 [ 2]]
```

То, что мы сделали здесь, называется преобразованием непрерывного входного признака набора данных `wave` в категориальный признак. С помощью категориального признака мы задаем категорию для каждой точки данных. Чтобы запустить модель `scikit-learn` на этих данных, мы выполним прямое кодирование этого дискретного признака с помощью функции `OneHotEncoder` из модуля `preprocessing`. Функция `OneHotEncoder` выполняет ту же самую кодировку, что и `pandas.get_dummies`, хотя в настоящее время она работает только с категориальными переменными, которые принимают целочисленные значения:

```
In[14]:
from sklearn.preprocessing import OneHotEncoder
# преобразовываем с помощью OneHotEncoder
encoder = OneHotEncoder(sparse=False)
# encoder.fit находит уникальные значения, имеющиеся в which_bin
encoder.fit(which_bin)
# transform осуществляет прямое кодирование
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

```
Out[14]:
[[ 0.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

Поскольку мы указали 10 категорий, преобразованный набор данных `X_binned` теперь состоит из 10 признаков:

```
In[15]:
print("форма массива X_binned: {}".format(X_binned.shape))
```

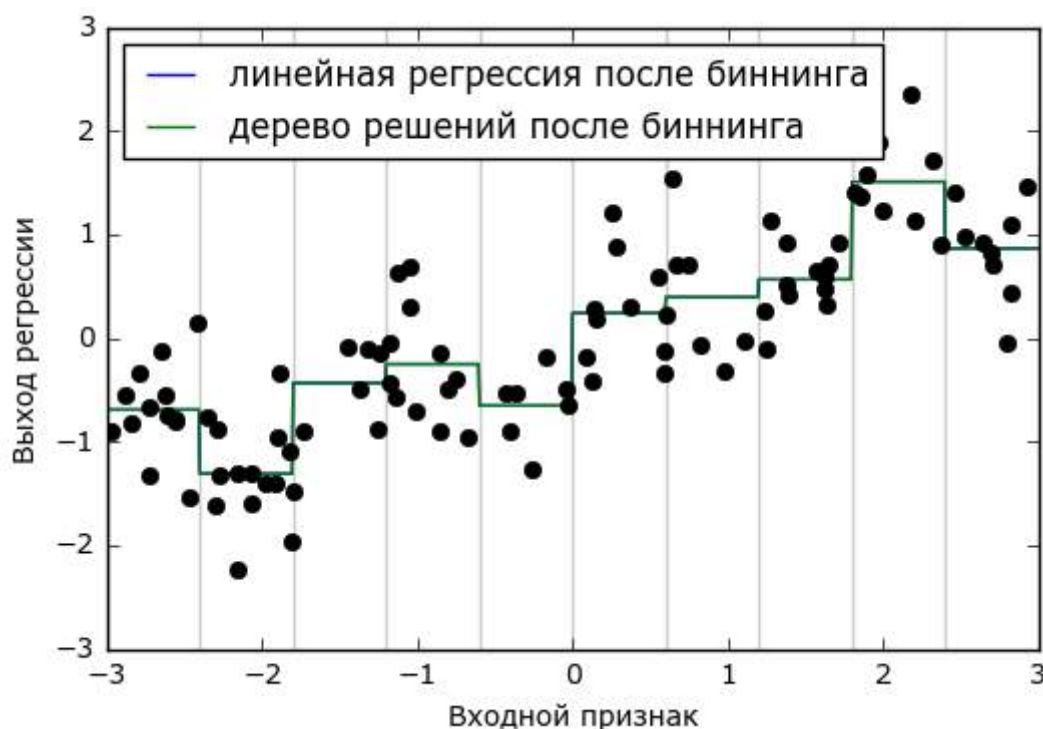
Out[15]:  
форма массива X\_binned: (100, 10)

Сейчас мы строим новую модель линейной регрессии и новую модель дерева решений на основе данных, преобразованных с помощью прямого кодирования. Результат визуализирован на рис. 4.2, также показаны границы категорий, обозначенные вертикальными серыми линиями:

```
In[16]:
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='линейная регрессия после биннинга')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='дерево решений после биннинга')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Выход регрессии")
plt.xlabel("Входной признак")
```



**Рис. 4.2** Сравнение результатов модели линейной регрессии и дерева регрессии после проведения биннинга

Синяя и зеленая линии лежат точно поверх друг друга, это означает, что модель линейной регрессии и дерево решений дают одинаковые прогнозы. Для каждой категории они предсказывают одно и то же значение (константу). Поскольку признаки принимают одно и то же значение в пределах каждой категории, любая модель должна предсказывать одно и то же значение для всех точек, находящихся



внутри категории. Сравнив модели, обученные до и после биннинга переменных, мы видим, что линейная модель стала теперь гораздо более гибкой, потому что теперь она присваивает различные значения категориям, в то время как модель дерева решений стала существенно менее гибкой. В целом биннинг признаков не дает положительного эффекта для моделей на основе дерева, поскольку эти модели сами могут научиться разбивать данные по любому значению. В некотором смысле это означает, что деревья решений могут самостоятельно осуществить биннинг для наилучшего прогнозирования данных. Кроме того, при выполнении разбиений деревья решений рассматривают несколько признаков сразу, в то время как обычный биннинг выполняется на основе анализа одного признака. Однако линейная модель после преобразования данных выиграла с точки зрения эффективности.

Если есть веские причины использовать линейную модель для конкретного набора данных (например, он имеет большой объем и является многомерным), но некоторые признаки имеют нелинейные взаимосвязи с зависимой переменной – биннинг может быть отличным способом увеличить прогнозную силу модели.

## Взаимодействия и полиномы

Еще один способ обогатить пространство признаков, в частности, для линейных моделей, заключается в добавлении *взаимодействий признаков* (*interaction features*) и *полиномиальных признаков* (*polynomial features*). Конструирование признаков подобного рода получило распространение в статистическом моделировании, а также широко используется во многих практических сферах применения машинного обучения.

Для начала снова посмотрите на рис. 4.2. Для каждой категории признака линейная модель предсказывает одно и то же значение. Однако мы знаем, что линейные модели могут вычислить не только значения сдвига, но и значения наклона. Один из способов добавить наклон в линейную модель, построенную на основе категоризированных данных, заключается в том, чтобы добавить обратно исходный признак (ось x на графике). Это приведет к получению 11-мерного массива данных, как показано на рис. 4.3:

```
In[17]:
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

```
Out[17]:
(100, 11)
```

```

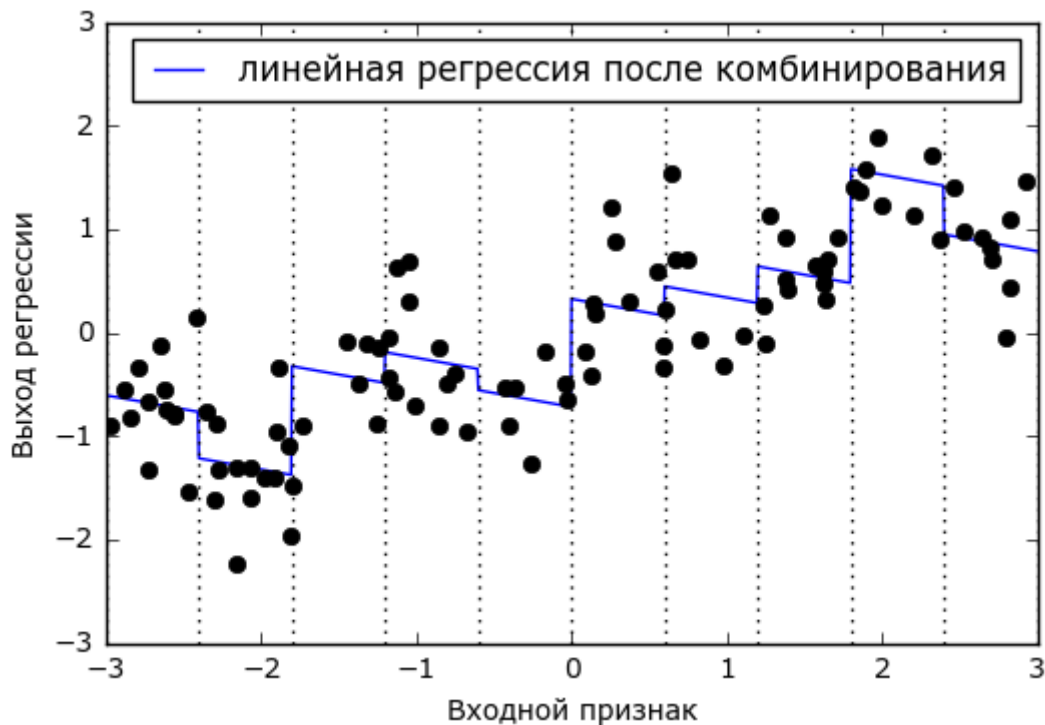
In[18]:
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='линейная регрессия после комбинирования')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')

plt.legend(loc="best")
plt.ylabel("Выход регрессии")
plt.xlabel("Входной признак")
plt.plot(X[:, 0], y, 'o', c='k')

```



**Рис. 4.3** Линейная регрессия с использованием категоризированных признаков и одним глобальным наклоном

В этом примере модель вычислила сдвиг для каждой категории, а также наклон. Вычисленный наклон направлен вниз и он является общим для всех категорий, так как у нас имеется только один признак по оси  $x$  с одним коэффициентом. Поскольку наличие одного наклона для всех категорий не очень сильно поможет с точки зрения моделирования, мы бы хотели вычислить для каждой категории свой собственный наклон! Мы можем добиться этого, добавив взаимодействие или произведение признаков, указывающее категорию точки данных и ее расположение на оси  $x$ . Данный признак является произведением индикатора категории и исходной переменной. Давайте создадим этот набор данных:

```

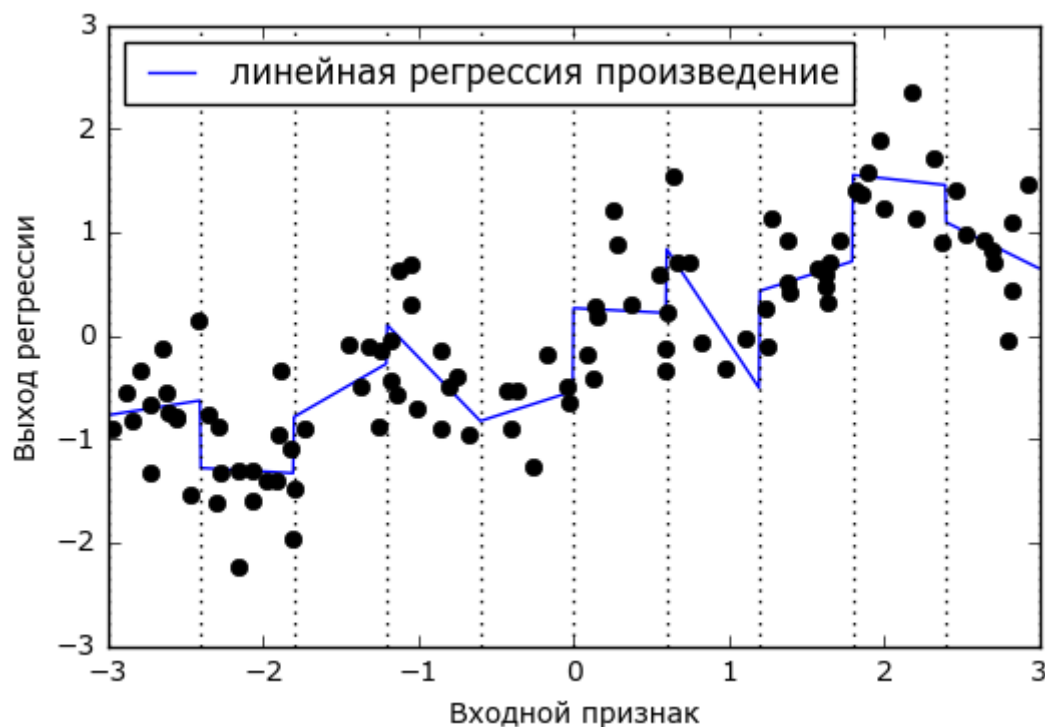
In[19]:
X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)

```

```
Out[19]:  
(100, 20)
```

Теперь набор данных содержит 20 признаков: в него записывается индикатор категории, в которой находится точка данных, а также произведение исходного признака и индикатора категории. Произведение признаков можно представить как отдельную копию признака, отложенного по оси  $x$ , для каждой категории. Оно соответствует исходному признаку, попадающему в данную категорию, или нулю во всех остальных случаях. На рис. 4.4 показан результат линейной модели для этого нового пространства признаков:

```
In[20]:  
reg = LinearRegression().fit(X_product, y)  
  
line_product = np.hstack([line_binned, line * line_binned])  
plt.plot(line, reg.predict(line_product), label='линейная регрессия произведение')  
  
for bin in bins:  
    plt.plot([bin, bin], [-3, 3], ':', c='k')  
  
plt.plot(X[:, 0], y, 'o', c='k')  
plt.ylabel("Выход регрессии")  
plt.xlabel("Входной признак")  
plt.legend(loc="best")
```



**Рис. 4.4** Линейная регрессия с отдельным наклоном для каждой категории

Видно, что теперь каждая категория имеет свое собственное значение сдвига и свое собственное значение наклона.

Использование биннинга – это способ увеличения пространства входных признаков. Еще один способ заключается в использовании

полиномов (*polynomials*) исходных признаков. Для признака  $x$  мы рассмотрим  $x ** 2$ ,  $x ** 3$ ,  $x ** 4$  и так далее. Данную операцию можно выполнить с помощью `PolynomialFeatures` модуля `preprocessing`:

```
In[21]:
from sklearn.preprocessing import PolynomialFeatures

# задаем степень полинома 10:
# значение по умолчанию "include_bias=True" добавляет признак-константу 1
poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)
```

Использование 10-й степени дает 10 признаков:

```
In[22]:
print("форма массива X_poly: {}".format(X_poly.shape))
```

```
Out[22]:
форма массива X_poly: (100, 10)
```

Давайте сравним элементы массива `X_poly` с элементами массива `X`:

```
In[23]:
print("Элементы массива X:\n{}".format(X[:5]))
print("Элементы массива X_poly:\n{}".format(X_poly[:5]))
```

```
Out[23]:
```

Элементы массива X:

```
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
```

Элементы массива X\_poly:

```
[[-0.753    0.567   -0.427    0.321   -0.242    0.182
  -0.137    0.103   -0.078    0.058
 [ 2.704    7.313   19.777   53.482  144.632  391.125
 1057.714 2860.360 7735.232 20918.278]
 [ 1.392    1.938    2.697    3.754    5.226    7.274
 10.125   14.094   19.618   27.307]
 [ 0.592    0.350    0.207    0.123    0.073    0.043
 0.025    0.015    0.009    0.005]
 [-2.064    4.260   -8.791   18.144  -37.448   77.289
 -159.516  329.222 -679.478 1402.367]]
```

Вы можете понять содержательный смысл этих признаков, вызвав метод `get_feature_names`, который выведет название каждого признака:

```
In[24]:
print("Имена полиномиальных признаков:\n{}".format(poly.get_feature_names()))
```

```
Out[24]:
```

Имена полиномиальных признаков:

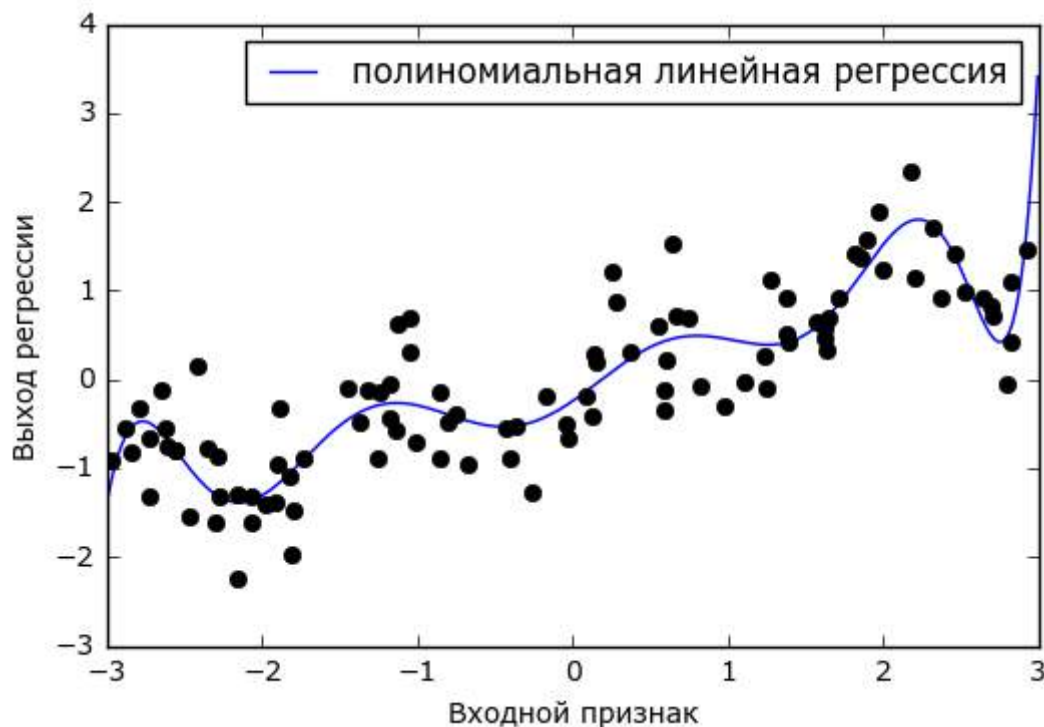
```
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

Видно, что первый столбец `X_poly` точно соответствует `X`, в то время как другие столбцы являются различными степенями первого элемента.

Использование полиномиальных признаков в модели линейной регрессии дает классическую модель *полиномиальной регрессии* (*polynomial regression*), представленную на рис. 4.5:

```
In[26]:
reg = LinearRegression().fit(X_poly, y)

line_poly = poly.transform(line)
plt.plot(line, reg.predict(line_poly), label='полиномиальная линейная регрессия')
plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Выход регрессии")
plt.xlabel("Входной признак")
plt.legend(loc="best")
```



**Рис. 4.5** Полиномиальная линейная регрессия, использовался полином 10-й степени

Видно, что на этом одномерном наборе данных полиномиальные признаки дают очень сглаженную подгонку. Однако полиномы высокой степени, как правило, резко меняют направление на границах области определения или в менее плотных областях данных.

Для сравнения ниже приводится модель ядерного SVM, обученная на исходных данных без каких-либо преобразований (см. рис. 4.6):

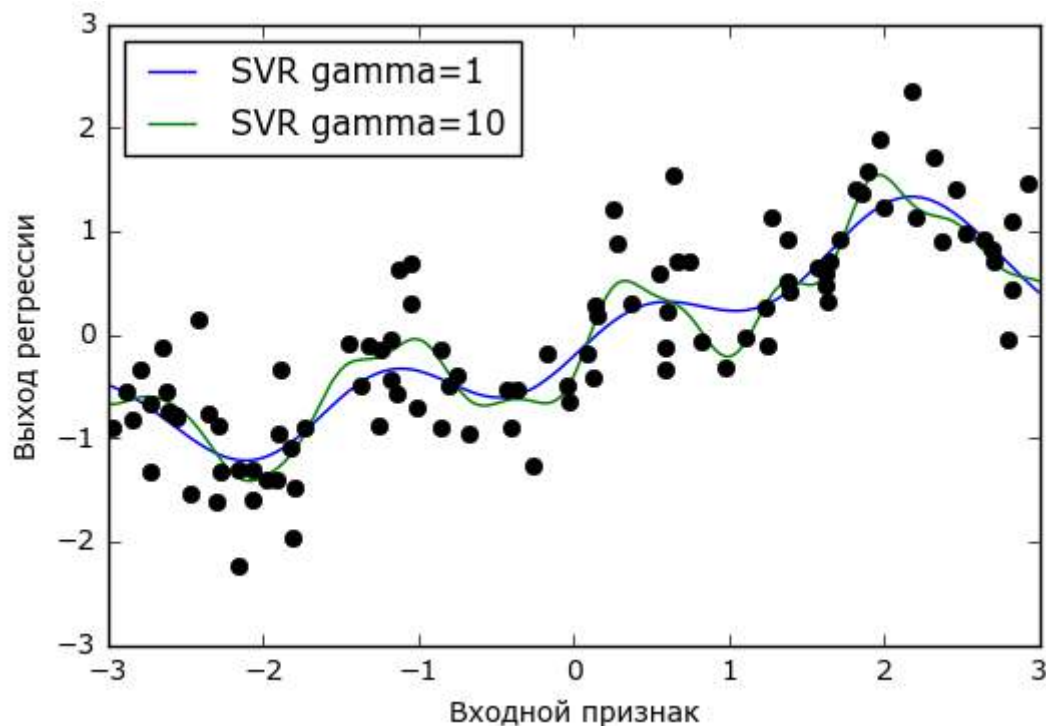
```

In[26]:
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Выход регрессии")
plt.xlabel("Входной признак")
plt.legend(loc="best")

```



**Рис. 4.6** Сравнение различных параметров gamma для SVM с RBF-ядром

Используя более сложную модель, модель ядерного SVM, мы можем получить такой же сложный прогноз, как в случае полиномиальной регрессии, не прибегая к явному преобразованию признаков.

В качестве более реального примера, иллюстрирующего применение взаимодействий и полиномов, давайте еще раз обратимся к набору данных Boston Housing. Мы уже использовали полиномиальные признаки этого набора данных в главе 2. Теперь давайте посмотрим, как были получены эти признаки и выясним, насколько они могут помочь нам улучшить прогноз. Сначала мы загрузим данные и отмасштабируем их с помощью `MinMaxScaler`, чтобы все признаки принимали значения в диапазоне между 0 и 1:

```

In[27]:
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split (boston.data, boston.target,
                                                    random_state=0)

# масштабируем данные
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

```

Теперь, мы выделим полиномиальные признаки и взаимодействия вплоть до 2-й степени:

```

In[28]:
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("форма обучающего массива X: {}".format(X_train.shape))
print("форма обучающего массива X полиномы и взаим: {}".format(X_train_poly.shape))

```

```

Out[28]:
форма обучающего массива X: (379, 13)
форма обучающего массива X полиномы и взаим: (379, 105)

```

Набор данных первоначально содержал 13 признаков, которые в ходе преобразований превратились в 105 новых признаков. Эти новые признаки представляют собой все возможные взаимодействия между двумя различными исходными характеристиками, а также квадраты исходных характеристик. В данном случае `degree=2` означает, что мы рассматриваем признаки, которые являются произведением не более двух исходных характеристик. Точное соответствие между входными и выходными признаками можно найти с помощью метода `get_feature_names`:

```

In[29]:
print("Имена полиномиальных признаков:\n{}".format(poly.get_feature_names()))

```

```

Out[29]:
Имена полиномиальных признаков:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10',
 'x11', 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5', 'x0 x6',
 'x0 x7', 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^2', 'x1 x2',
 'x1 x3', 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1 x9', 'x1 x10',
 'x1 x11', 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5', 'x2 x6', 'x2 x7',
 'x2 x8', 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^2', 'x3 x4', 'x3 x5',
 'x3 x6', 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3 x11', 'x3 x12', 'x4^2',
 'x4 x5', 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x4 x10', 'x4 x11', 'x4 x12',
 'x5^2', 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9', 'x5 x10', 'x5 x11', 'x5 x12',
 'x6^2', 'x6 x7', 'x6 x8', 'x6 x9', 'x6 x10', 'x6 x11', 'x6 x12', 'x7^2',
 'x7 x8', 'x7 x9', 'x7 x10', 'x7 x11', 'x7 x12', 'x8^2', 'x8 x9', 'x8 x10',
 'x8 x11', 'x8 x12', 'x9^2', 'x9 x10', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11',
 'x10 x12', 'x11^2', 'x11 x12', 'x12^2']

```

В данном случае первый новый признак является признаком-константой "1". Следующие 13 признаков являются исходными признаками (от "x0" до "x12"). Затем следует квадрат первого признака



("x0^2"), после него приводятся произведения первого и остальных признаков.

Давайте вычислим правильность прогнозов, применив модель **Ridge** к данным, включающим взаимодействия, и данным без взаимодействий:

```
In[30]:
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Правильность на тестовом наборе без взаимодействий: {:.3f}".format(
    ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Правильность на тестовом наборе с взаимодействиями: {:.3f}".format(
    ridge.score(X_test_poly, y_test)))
```

```
Out[30]:
Правильность на тестовом наборе без взаимодействий: 0.621
Правильность на тестовом наборе с взаимодействиями: 0.753
```

Очевидно, что в случае с гребневой регрессией взаимодействия и полиномиальные признаки позволяют улучшить правильность модели. Впрочем, применение более сложной модели типа случайного леса дает немного другой результат:

```
In[31]:
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("Правильность на тестовом наборе без взаимодействий: {:.3f}".format(
    rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Правильность на тестовом наборе с взаимодействиями: {:.3f}".format(
    rf.score(X_test_poly, y_test)))
```

```
Out[31]:
Правильность на тестовом наборе без взаимодействий: 0.799
Правильность на тестовом наборе с взаимодействиями: 0.763
```

Видно, что даже без дополнительных признаков случайный лес дает более высокую правильность прогнозов, чем гребневая регрессия. Включение взаимодействий и полиномов на самом деле немного уменьшает правильность прогнозов.

## Одномерные нелинейные преобразования

Мы только что увидели, что добавление признаков, возведенных в квадрат или куб, может улучшить линейные модели регрессии. Существуют и другие преобразования, которые часто оказываются полезными в плане трансформации определенных признаков: в частности, применение математических функций типа **log**, **exp** или **sin**. Если модели на основе дерева заботятся лишь о выстраивании признаков в иерархию, то линейные модели и нейронные сети очень привязаны к масштабу и распределению каждого признака, поэтому наличие нелинейной взаимосвязи между признаком и зависимой переменной становится проблемой для модели, особенно для регрессии. Функции **log**

и `exp` позволяют скорректировать относительные шкалы переменных таким образом, чтобы линейная модель или нейронная сеть могли лучше обработать их. Мы видели подобный пример в главе 2, когда работали с набором данных `gam_prices`. Функции `sin` и `cos` могут пригодиться при работе с данными, которые представляют собой периодические структуры.

Большинство моделей работают лучше, когда признаки (а если используется регрессия, то и зависимая переменная) имеют гауссовское распределение, то есть гистограмма каждого признака должна в определенной степени иметь сходство с «колоколообразной кривой». Использование преобразований типа `log` и `exp` является банальным, но в то же время простым и эффективным способом добиться более симметричного распределения. Наиболее характерный случай, когда подобное преобразование может быть полезно, – обработка дискретных данных. Под дискретными данными мы подразумеваем признаки типа «как часто пользователь А входил в систему». Дискретные данные никогда не бывают отрицательными и часто подчиняются конкретным статистическим закономерностям. Здесь мы воспользуемся синтетическим набором дискретных данных с теми же самыми признаками, что можно встретить в реальной практике.<sup>28</sup> Все признаки имеют целочисленные значения, в то время как зависимая переменная является непрерывной:

```
In[32]:
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

Давайте посмотрим на первые 10 элементов первого признака. Все они являются положительными и целочисленными значениями, однако выделить какую-то определенную структуру сложно.

Если посчитать частоту встречаемости каждого значения, распределение значений становится более ясным:

```
In[33]:
print("Частоты значений:\n{}".format(np.bincount(X[:, 0])))
```

```
Out[33]:
Частоты значений:
[28 38 68 48 61 59 45 56 37 40 35 34 36 26 23 26 27 21 23 23 18 21 10 9 17
 9 7 14 12 7 3 8 4 5 5 3 4 2 4 1 1 3 2 5 3 8 2 5 2 1
 2 3 3 2 2 3 3 0 1 2 1 0 0 3 1 0 0 0 1 3 0 1 0 2 0
 1 1 0 0 0 0 1 0 0 2 2 0 1 1 0 0 0 0 1 1 0 0 0 0
 0 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0
 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

<sup>28</sup> Речь идет о распределении Пуассона, которое является основополагающим для дискретных величин.

Значение 2, по-видимому, является наиболее распространенным, оно встречается 68 раз (`bincount` всегда начинает считать с 0), а частоты более высоких значений быстро падают. Однако есть несколько очень высоких значений, например, 84 и 85, которые встречаются два раза. Мы визуализируем частоты на рис. 4.7:

In[34]:

```
bins = np.bincount(X[:, 0])
plt.bar(range(len(bins)), bins, color='w')
plt.ylabel("Частота")
plt.xlabel("Значение")
```

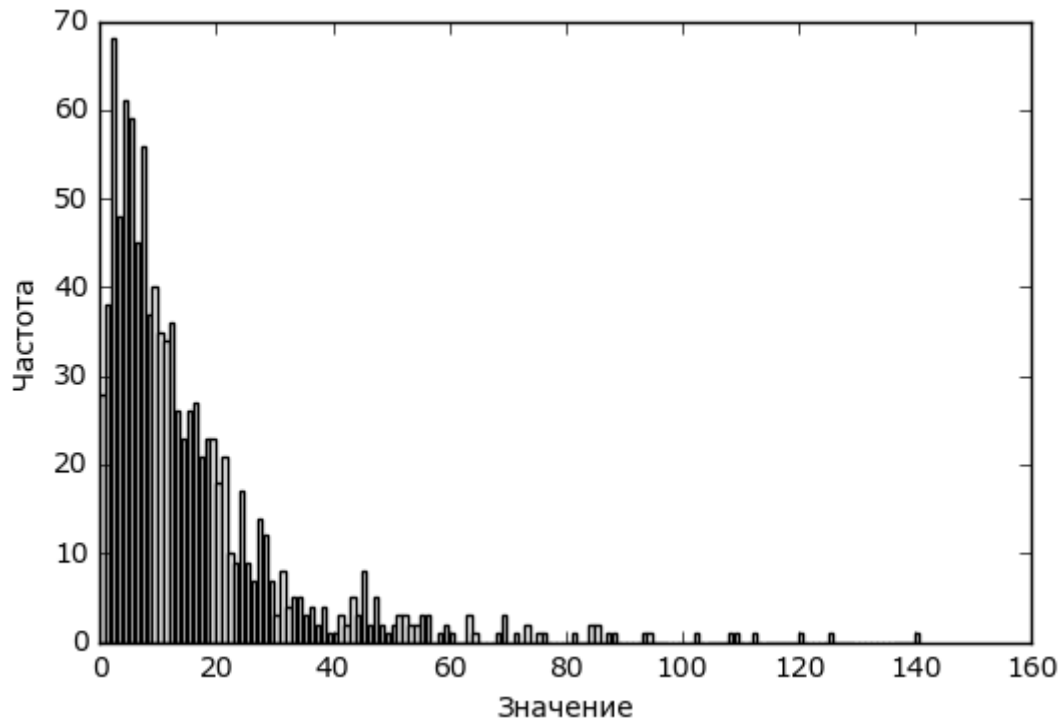


Рис. 4.7 Гистограмма значений  $X[0]$

Признаки  $X[:, 1]$  и  $X[:, 2]$  имеют аналогичные свойства. Полученное распределение значений (высокая частота встречаемости маленьких значений и низкая частота встречаемости больших значений) является очень распространенным явлением в реальной практике. Однако для большинства линейных моделей оно может представлять трудность. Давайте попробуем подогнать гребневую регрессию:

In[35]:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Правильность на тестовом наборе: {:.3f}".format(score))
```

Out[35]:

```
Правильность на тестовом наборе: 0.622
```

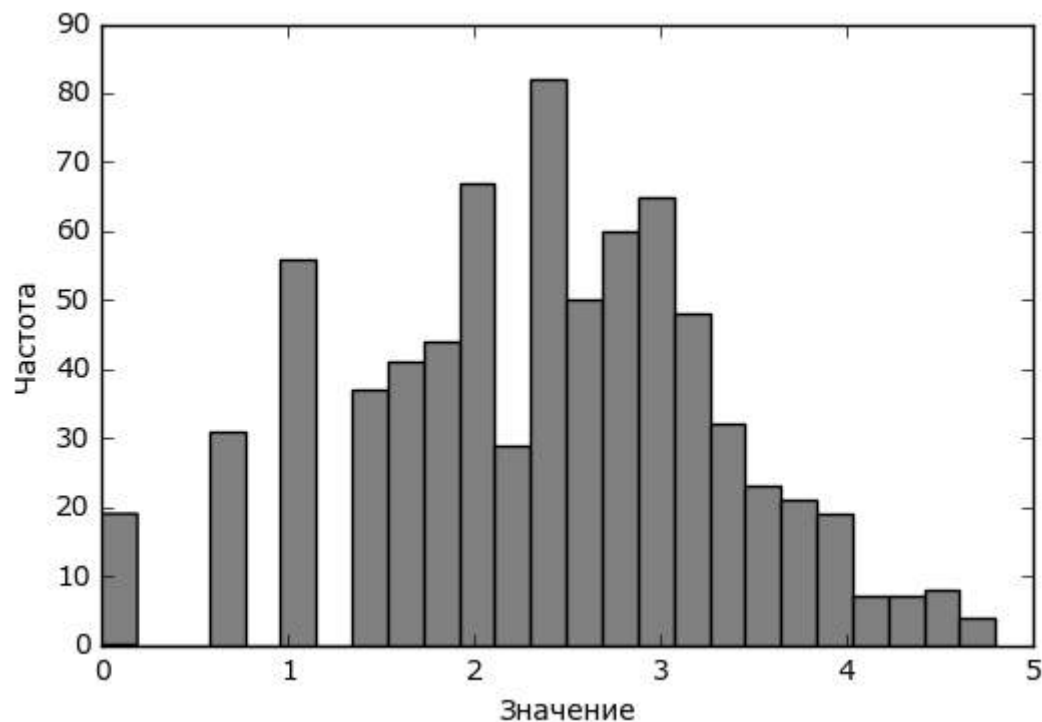
Видно, что из-за относительно низкого значения  $R^2$  гребневая регрессия не может должным образом смоделировать взаимосвязь между

$X$  и  $y$ . Впрочем, применение логарифмического преобразования может помочь. Поскольку в данных появляется значение 0 (а логарифм 0 не определен), мы не можем просто взять и применить  $\log$ , вместо этого мы должны вычислить  $\log(X + 1)$ :

```
In[36]:
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

После преобразования распределение данных стало менее асимметричным и уже не содержит очень больших выбросов (см. рис. 4.8):

```
In[37]:
plt.hist(X_train_log[:, 0], bins=25, color='gray')
plt.ylabel("Частота")
plt.xlabel("Значение")
```



**Рис. 4.8** Гистограмма значений  $X[0]$  после логарифмического преобразования

Построение модели гребневой регрессии на новых данных дает гораздо более лучшее качество подгонки:

```
In[38]:
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Правильность на тестовом наборе: {:.3f}".format(score))
```

```
Out[38]:
Правильность на тестовом наборе: 0.875
```

Поиск преобразования, которое наилучшим образом сработает для конкретного сочетания данных и модели – это в некоторой степени искусство. В этом примере все признаки имели одинаковые свойства. Такое редко бывает на практике, и как правило, лишь некоторые признаки нуждаются в преобразовании, либо в ряде случаев каждый признак необходимо преобразовывать по-разному. Как мы уже упоминали ранее, эти виды преобразований не имеют значения для моделей на основе дерева, но могут иметь важное значение для линейных моделей. Иногда при построении регрессии целесообразно преобразовать зависимую переменную  $y$ . Прогнозирование частот (скажем, количества заказов) является довольно распространенной задачей, и преобразование  $\log(y + 1)$  часто помогает.<sup>29</sup>

Как вы видели в предыдущих примерах, биннинг, полиномы и взаимодействия могут иметь огромное влияние на качество работы модели. Особенно это актуально для менее сложных моделей типа линейных моделей и наивных байесовских моделей. С другой стороны, модели на основе дерева, как правило, могут обнаружить важные взаимодействия самостоятельно и чаще всего не требуют явного преобразования данных. Использование биннинга, взаимодействий и полиномов в ряде случаев может положительно сказаться на работе моделей типа SVM, ближайших соседей и нейронных сетей, однако последствия, возникающие в результате этих преобразований, представляются менее ясными в отличие от преобразований, применяемых в линейных моделях.

## Автоматический отбор признаков

При таком разнообразии способов, позволяющих сгенерировать новые признаки, у вас, возможно, возникнет искушение увеличить размерность данных, превысив количество исходных признаков. Однако добавление новых признаков делает модели более сложными и поэтому увеличивает вероятность переобучения. Добавляя новые признаки или работая с высокоразмерными наборами данных, неплохо бы уменьшить количество признаков и оставить только наиболее полезные из них. Это позволит получить более простые модели с лучшей обобщающей способностью. Однако как узнать, насколько полезен каждый признак? Существуют три основные стратегии: *одномерные статистики* (*univariate statistics*), *отбор на основе модели* (*model-based selection*) и *итеративный отбор* (*iterative selection*). Мы подробно рассмотрим все три стратегии. Все эти методы

---

<sup>29</sup> Это очень грубая аппроксимация с использованием регрессии Пуассона, которое было бы правильным решением с вероятностной точки зрения.

относятся методам машинного обучения с учителем, то есть для подгонки модели им требуется зависимая переменная. Это означает, что нам нужно разбить данные на обучающий и тестовый наборы и осуществить отбор признаков лишь на обучающей выборке.

## Одномерные статистики

С помощью одномерных статистик мы определяем наличие статистически значимой взаимосвязи между каждым признаком и зависимой переменной. Затем отбираем признаки, сильнее всего связанные с зависимой переменной (имеющие уровень значимости, не превышающий заданного порогового значения). В случае классификации эта процедура известна как дисперсионный анализ (ANOVA). Ключевым свойством этих тестов является то, что они являются одномерными, то есть они рассматривают каждую характеристику по отдельности. Следовательно признак будет исключен, если он становится информативным лишь в сочетании с другим признаком. Как правило, одномерные тесты очень быстро вычисляются и не требуют построения модели. С другой стороны, они являются полностью независимыми от модели, которой вы, возможно, захотите применить после отбора признаков.

Чтобы осуществить одномерный отбор признаков в `scikit-learn`, вам нужно выбрать тест, обычно либо `f_classif` (по умолчанию) для классификации или `f_regression` для регрессии, а также метод исключения признаков, основанный на  $p$ -значениях, вычисленных в ходе теста. Все методы исключения параметров используют пороговое значение, чтобы исключить все признаки со слишком высоким  $p$ -значением (высокое  $p$ -значение указывает на то, что признак вряд ли связан с зависимой переменной). Методы отличаются способами вычисления этого порогового значения, самым простым из которых являются `SelectK`, выбирающий фиксированное число  $k$  признаков, и `SelectPercentile`, выбирающий фиксированный процент признаков. Давайте применим отбор признаков для классификационной задачи к набору данных `cancer`. Чтобы немного усложнить задачу, мы добавим к данным некоторые неинформативные шумовые признаки. Мы предполагаем, что отбор признаков сможет определить неинформативные признаки и удалит их:

```

In[39]:
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

# задаем определенное стартовое значение для воспроизводимости результата
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# добавляем к данным шумовые признаки
# первые 30 признаков являются исходными, остальные 50 являются шумовыми
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
# используем f_classif (по умолчанию)
# и SelectPercentile, чтобы выбрать 50% признаков
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# преобразовываем обучающий набор
X_train_selected = select.transform(X_train)

print("форма массива X_train: {}".format(X_train.shape))
print("форма массива X_train_selected: {}".format(X_train_selected.shape))

Out[39]:
форма массива X_train: (284, 80)
форма массива X_train_selected: (284, 40)

```

Как видно, количество признаков уменьшилось с 80 до 40 (на 50% от исходного количества признаков). Мы можем выяснить, какие функции были отобраны, воспользовавшись методом `get_support`, который возвращает булевы значения для каждого признака (визуализированы на рис. 4.9):

```

In[40]:
mask = select.get_support()
print(mask)
# визуализируем булевы значения: черный - True, белый - False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Индекс примера")

Out[40]:
[ True  True  True  True  True  True  True  True  True  True  False  True  False
   True  True  True  True  True  True  True  False  False  True  True  True  True
   True  True  True  True  True  True  True  False  False  False  True  False  True
  False  False  True  False  False  False  False  False  True  False  False  True  False
  False  True  False  True  False  False  False  False  False  False  True  False
   True  False  False  False  False  True  False  True  False  False  False  False
   True  True  False  True  False  False  False  False]

```



**Рис. 4.9** Признаки, отобранные с помощью `SelectPercentile`

Благодаря визуализации видно, что большинство отобранных признаков являются исходными характеристиками, а большинство шумовых признаков были удалены. Тем не менее восстановление исходных признаков далеко от идеала. Давайте сравним правильность логистической регрессии с использованием всех признаков с



правильностью логистической регрессии, использующей лишь отобранные признаки:

```
In[41]:
from sklearn.linear_model import LogisticRegression

# преобразовываем тестовые данные
X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Правильность со всеми признаками: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("Правильность только с отобранными признаками: {:.3f}".format(
    lr.score(X_test_selected, y_test)))

Out[41]:
Правильность со всеми признаками: 0.930
Правильность только с отобранными признаками: 0.940
```

В данном случае удаление шумовых признаков повысило правильность, даже несмотря на то, что некоторые исходные признаки отсутствовали. Это был очень простой синтетический пример, результаты, получающиеся на реальных данных, как правило, получаются смешанными. Однако одномерный отбор признаков может быть очень полезен, если их количество является настолько большим, что невозможно построить модель, используя все эти характеристики, или же вы подозреваете, что многие характеристики совершенно неинформативны.

## Отбор признаков на основе модели

Отбор признаков на основе модели использует модель машинного обучения с учителем, чтобы вычислить важность каждого признака, и оставляет только самые важные из них. Модель машинного обучения с учителем, которая используется для отбора признаков, не должна использоваться для построения итоговой модели. Модель, применяющаяся для отбора признаков, требует вычисления определенного показателя важности для всех признаков, с тем чтобы характеристики можно было ранжировать по этой метрике. В деревьях решений и моделях на основе дерева решений такой показатель реализован с помощью атрибута `feature_importances_`, в котором записывается важность каждого признака. У линейных моделей есть коэффициенты, абсолютные значения которых также можно использовать для оценки важности признаков. Как мы видели в главе 2, линейные модели с L1 штрафом позволяют вычислить разреженные решения <sup>30</sup>, которые используют лишь небольшое подмножество признаков. Поэтому процедуру L1 регуляризации можно рассматривать

---

<sup>30</sup> Решения, при которых большинство коэффициентов тождественно равно 0. – Прим. пер.

как один из способов отбора признаков, выполняемый самой моделью. Кроме того, эту процедуру можно использовать в качестве инструмента предварительной обработки, позволяющего отобрать признаки для другой модели. В отличие от одномерного отбора отбор на основе модели рассматривает все признаки сразу и поэтому может обнаружить взаимодействия (если модель способна выявить их). Чтобы применить отбор на основе модели, мы должны воспользоваться модификатором `SelectFromModel`:

```
In[42]:
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

Класс `SelectFromModel` отбирает все признаки, у которых показатель важности (заданный моделью машинного обучения с учителем) превышает установленное пороговое значение. Чтобы вычислить результат, сопоставимый с тем, который мы получили при однофакторном отборе признаков, мы использовали в качестве порогового значения медиану, поэтому будет отобрана половина признаков. Мы используем случайный лес на основе деревьев классификации (100 деревьев), чтобы вычислить важности признаков. Это довольно сложная модель, обладающая гораздо большей прогнозной силой, нежели одномерные тесты. Теперь давайте подгоним эту модель:

```
In[43]:
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("форма обуч набора X: {}".format(X_train.shape))
print("форма обуч набора X с l1: {}".format(X_train_l1.shape))
```

```
Out[43]:
форма обуч набора X: (284, 80)
форма обуч набора X с l1: (284, 40)
```

И снова мы можем взглянуть на отобранные признаки (рис. 4.10):

```
In[44]:
mask = select.get_support()
# визуализируем булевы значения -- черный - True, белый - False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Индекс примера")
```



**Рис. 4.10** Признаки, отобранные `SelectFromModel` с помощью `RandomForestClassifier`

На этот раз были отобраны все исходные признаки, кроме двух. Поскольку мы задали отбор лишь 40 признаков, некоторые шумовые признаки также будут выбраны. Давайте посмотрим на правильность:

```
In[45]:
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("Правильность на тестовом наборе: {:.3f}".format(score))
```

```
Out[45]:
Правильность на тестовом наборе: 0.951
```

Используя более оптимальный отбор признаков, мы смогли немного улучшить прогноз.

## Итеративный отбор признаков

В одномерном отборе признаков мы не использовали модель, а в отборе признаков на основе модели мы построили одну модель, чтобы выбрать характеристики. В итеративном отборе признаков строится последовательность моделей с различным количеством признаков. Существует два основных метода. Первый метод начинается с шага, когда в модель включена лишь одна константа (входных признаков нет) и затем добавляет признак за признаком до тех пор, пока не будет достигнут критерий остановки. Второй метод начинается с шага, когда все признаки включены в модель, и затем начинает удалять признак за признаком, пока не будет достигнут критерий остановки. Поскольку строится последовательность моделей, эти методы с вычислительной точки зрения являются гораздо более затратными в отличие от ранее обсуждавшихся методов. Одним из таких методов является метод *рекурсивного исключения признаков* (*recursive feature elimination, RFE*), который начинается с включения всех признаков, строит модель и исключает наименее важный признак с точки зрения модели. Затем строится новая модель с использованием всех признаков, кроме исключенного, и так далее, пока не останется лишь заранее определенное количество признаков. Чтобы все получилось, модели, используемой для отбора признаков, необходима определенная метрика, измеряющая важность признаков, как было в случае с модельным отбором. Здесь мы воспользуемся той же самой моделью случайного леса, которую применяли ранее, и получим результаты, показанные на рис. 4.11:

```
In[46]:
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
             n_features_to_select=40)

select.fit(X_train, y_train)
# визуализируем отобранные признаки:
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Индекс примера")
```



**Рис. 4.10** Признаки, отобранные методом рекурсивного исключения признаков с помощью RandomForestClassifier

Отбор признаков стал лучше по сравнению с одномерным отбором и отбором на основе модели, однако одного признака по-прежнему не хватает. Кроме того, выполнение этого программного кода занимает значительно больше времени в отличие от модельного отбора, поскольку модель случайного леса обучается 40 раз, по одной итерации для каждого отбрасываемого признака. Давайте проверим правильность модели логистической регрессии с использованием RFE для отбора признаков:

```
In[47]:
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)
score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Правильность на тестовом наборе: {:.3f}".format(score))
```

```
Out[47]:
Правильность на тестовом наборе: 0.951
```

Кроме того, мы можем применить модель, использованную внутри RFE, чтобы вычислить прогнозы. Она использует лишь набор отобранных признаков:

```
In[48]:
print("Правильность на тестовом наборе: {:.3f}".format(select.score(X_test, y_test)))
```

```
Out[48]:
Правильность на тестовом наборе: 0.951
```

В данном случае правильность случайного леса, используемого внутри RFE, совпадает с правильностью, достигнутой в результате обучения модели логистической регрессии на основе наилучших отобранных признаков. Другими словами, как только мы выбрали правильные признаки, линейная модель показала ту же самую правильность, что и случайный лес.

Если у вас нет уверенности в том, какие признаки использовать в качестве входных данных для вашего алгоритма машинного обучения, автоматический отбор признаков может быть весьма полезен. Кроме того, он отлично подходит для уменьшения количества необходимых признаков, например, чтобы увеличить скорость вычисления прогнозов или получить более интерпретируемые модели. В большинстве реальных примеров применение отбора признаков вряд ли обеспечит большой прирост производительности. Тем не менее, он по-прежнему является ценным инструментом в арсенале специалиста по анализу данных.

## Применение экспертных знаний

Как правило, конструирование признаков является тем важным этапом, на котором применяются *экспертные знания* (*expert knowledge*) в конкретной сфере. Хотя во многих случаях цель машинного обучения заключается в том, чтобы избежать построения набора экспертных правил, это вовсе не означает, что априорные знания в той или иной сфере или области должны быть отброшены. Как правило, эксперты могут помочь выделить полезные признаки, которые являются гораздо более информативным, чем исходные данные. Представьте, что вы работаете в туристическом агентстве и вам нужно спрогнозировать цены на авиарейсы. Допустим, у вас есть цены с датами, названиями авиакомпаний, местами отправления и назначения. Возможно, что модель машинного обучения вполне способна построить достойную модель на основе этих данных. Однако некоторые важные факторы, связанные с ценами на авиарейсы останутся без внимания. Например, стоимость авиарейсов, как правило, становится выше в месяцы, приходящиеся на период отпусков, и в праздничные дни. Хотя даты некоторых праздников (например, Рождество) фиксированы, и поэтому их эффект можно учесть, исходя из даты, другие могут зависеть от фазы луны (например, Ханука и Пасха) или устанавливаться органами власти (например, каникулы). Эти события нельзя извлечь из данных, если каждый рейс записывается только с помощью (григорианской) даты. Однако легко добавить признак, который будет фиксировать день полета как предшествующий дню государственного праздника/дню объявления школьных каникул или следующий после дня государственного праздника/дня объявления школьных каникул. Таким образом, априорное знание можно закодировать в признаки, чтобы помочь алгоритму машинного обучения. Добавление признака не означает его обязательное использование алгоритмом машинного обучения и даже если информация о празднике окажется малоинформативной с точки зрения прогнозирования цен на авиарейсы, обогащение данных этим признаком не принесет вреда.

Теперь мы рассмотрим конкретный случай применения экспертных знаний, хотя в данном случае их с полным правом можно назвать «здравым смыслом». Задача заключается в том, чтобы спрогнозировать количество велосипедов, взятых напрокат перед домом Андреаса.

Система общественнотранспортных велосипедов City Bike в Нью-Йорке представляет собой сеть станций проката велосипедов, воспользоваться которой можно с помощью подписки. Станции расположены по всему городу и обеспечивают удобный способ передвижения. Данные о прокате велосипедов выложены на сайте City Bike в [анонимном виде](#) и были проанализированы различными

способами. Задача, которую мы хотим решить, заключается в том, чтобы предсказать, сколько людей воспользуется прокатом велосипедов перед домом Андреаса, поэтому он знает о количестве оставшихся велосипедов.

Сначала загрузим данные за август 2015 года для этой конкретной станции в виде пандасовского `DataFrame`. Мы разбили данные на 3-часовые интервалы, чтобы выделить основные тренды для каждого дня:

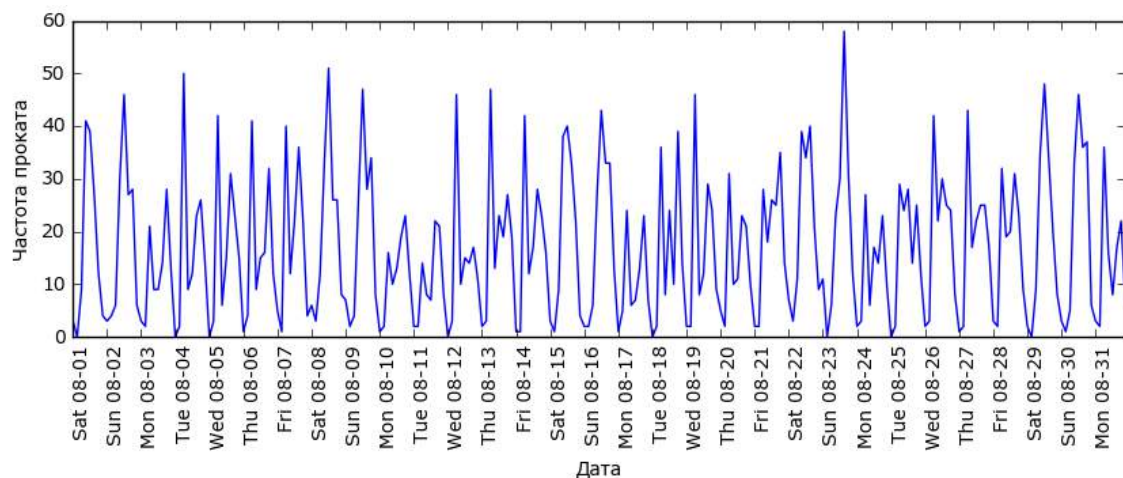
```
In[49]:
def load_citibike():
    data_mine = pd.read_csv("C:/Data/citibike.csv")
    data_mine['one'] = 1
    data_mine['starttime'] = pd.to_datetime(data_mine.starttime)
    data_starttime = data_mine.set_index("starttime")
    data_resampled = data_starttime.resample("3h").sum().fillna(0)
    return data_resampled.one
citibike = load_citibike()
```

```
In[50]:
print("данные Citi Bike:\n{}".format(citibike.head()))
```

```
Out[50]:
данные Citi Bike:
starttime
2015-08-01 00:00:00 3.0
2015-08-01 03:00:00 0.0
2015-08-01 06:00:00 9.0
2015-08-01 09:00:00 41.0
2015-08-01 12:00:00 39.0
Freq: 3H, Name: one, dtype: float64
```

Следующий пример показывает количество велосипедов, взятых в прокат, по дням месяца (рис. 4.12):

```
In[51]:
plt.figure(figsize=(10, 3))
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),
                        freq='D')
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")
plt.plot(citibike, linewidth=1)
plt.xlabel("Дата")
plt.ylabel("Частота проката")
```



**Рис. 4.12** Количество велосипедов, взятых на прокат в течение месяца для определенной станции



Взглянув на данные, мы можем четко выделить день и ночь для каждого 24-часового интервала. Структура данных для будних и выходных дней также выглядит совершенно по-разному. Решая задачу прогнозирования для временных рядов, мы *учимся на прошлом и делаем прогноз на будущее*. Это означает, что при разбиении данных на обучающий и тестовый наборы, нам нужно взять все данные до определенной даты в качестве обучающей выборки и все данные после этой даты в качестве тестовой выборки. Вот как мы обычно используем прогнозирование временных рядов: обладая информацией о прокате велосипедов в прошлом, мы строим предположения о том, что произойдет завтра. Мы используем первые 184 точки данных, соответствующие первым 23 дням, в качестве обучающего набора, а остальные 64 точки данных, соответствующие оставшимся 8 дням, в качестве тестового набора.

Единственный признак, который мы используем в нашей задаче прогнозирования, – дата и время проката. Таким образом, входной признак – это дата и время, например, 2015-08-01 00:00:00, а зависимая переменная – количество велосипедов, взятых на прокат в последующие три часа (в соответствии с нашим DataFrame).

Широко распространенный способ хранения дат на компьютерах – использование POSIX-времени, которое представляет собой количество секунд, прошедших с полуночи (00:00:00) 1 января 1970 года (оно же является точкой отсчета Unix-времени). В качестве первой попытки мы можем воспользоваться им для нашего представления дат:

```
In[52]:
# извлекаем значения зависимой переменной (количество велосипедов, взятых в прокат)
y = citibike.values
# преобразуем время в формат POSIX с помощью "%s"
X = citibike.index.astype("int64").reshape(-1, 1) // 10**9
```

Сначала мы зададим функцию, чтобы разбить данные на обучающий и тестовый наборы, построим модель и визуализируем результат:

```
In[54]:
# используем первые 184 точки данных для обучения, а остальные для тестирования
n_train = 184
# функция, которая строит модель на данном наборе признаков и визуализирует ее
def eval_on_features(features, target, regressor):
    # разбиваем массив признаков на обучающую и тестовую выборки
    X_train, X_test = features[:n_train], features[n_train:]
    # также разбиваем массив с зависимой переменной
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("R^2 для тестового набора: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,
              ha="left")

    plt.plot(range(n_train), y_train, label="обуч")
```



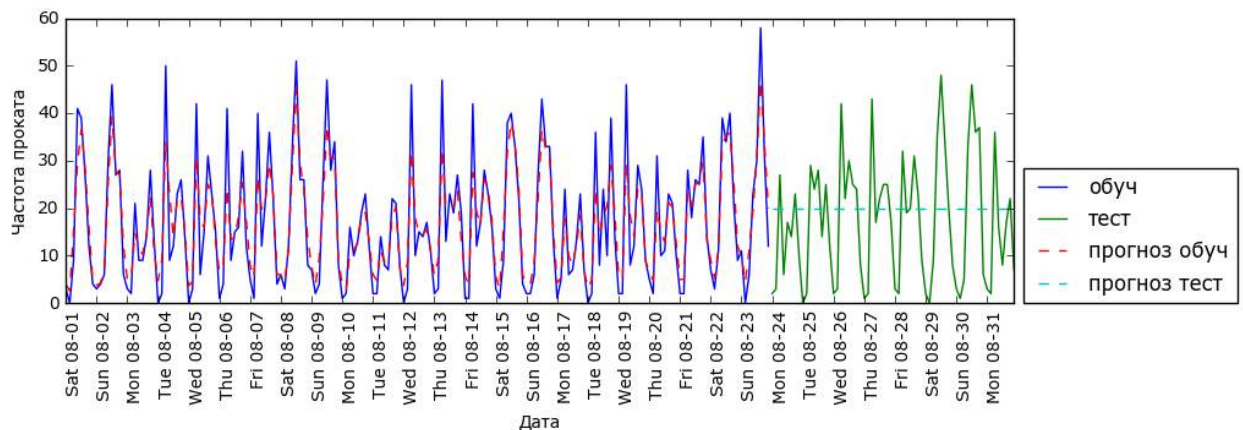
```
plt.plot(range(n_train, len(y_test) + n_train), y_test, '-', label="тест")
plt.plot(range(n_train), y_pred_train, '--', label="прогноз обуч")

plt.plot(range(n_train, len(y_test) + n_train), y_pred, '--',
         label="прогноз тест")
plt.legend(loc=(1.01, 0))
plt.xlabel("Дата")
plt.ylabel("Частота проката")
```

Ранее мы уже видели, что случайный лес требует очень незначительной предварительной обработки данных, что, по-видимому, делает его оптимальной моделью для старта. Мы передаем функции `eval_on_features` массив с признаком  $X$  (даты, преобразованные в POSIX-формат),  $y$  и модель случайного леса. Рис. 4.13 показывает результат:

```
In[55]:
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
plt.figure()
eval_on_features(X, y, regressor)
```

```
Out[55]:
R^2 для тестового набора: -0.04
```



**Рис. 4.13** Прогнозы, вычисленные случайным лесом (использовались даты, преобразованные в формат времени POSIX)

Как это обычно бывает при построении случайного леса, правильность прогнозов на обучающем наборе получилась довольно высокой. Однако для тестового набора прогнозируется ровная линия. Значение  $R^2$  равно  $-0.04$ , это означает, что наша модель ничему не научилась. Что произошло?

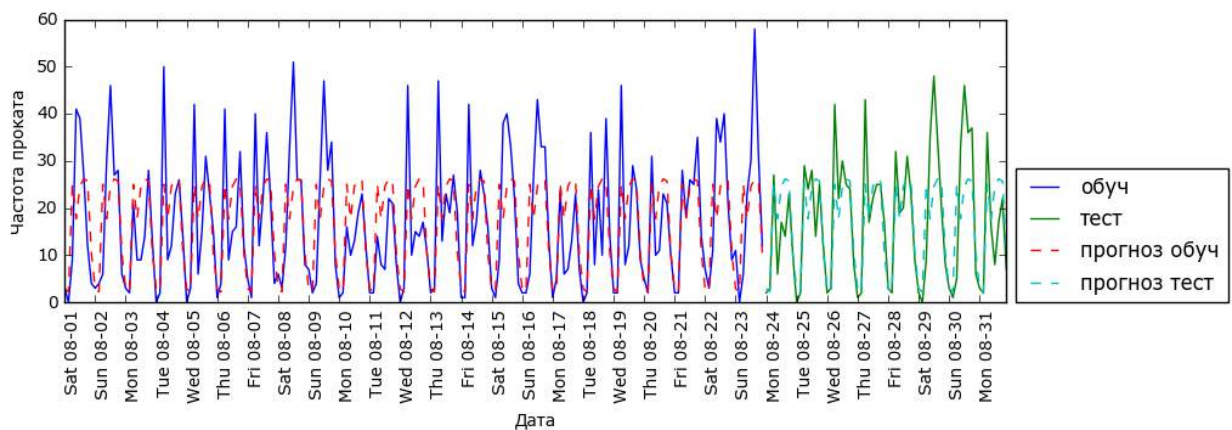
Проблема обусловлена сочетанием типа нашего признака и используемого метода (в данном случае случайного леса). Значения признака на основе POSIX-времени для тестового набора находятся вне диапазона значений этого признака в обучающей выборке: точки тестового набора в отличие от точек обучающего набора имеют более поздние временные метки. Дерево, а следовательно и случайный лес не могут *экстраполировать* (*extrapolate*) значения признаков, лежащие вне диапазона значений обучающих данных. Итог – модель просто

предсказывает значение зависимой переменной для ближайшей точки обучающего набора (для последней временной метки, которую она запомнила).

Ясно, что мы можем улучшить прогноз. Это тот момент, когда на помощь приходят наши «экспертные знания». Взглянув на то, как меняется частота проката в обучающих данных, можно выделить два очень важных фактора: время суток и день недели. Итак, давайте добавим эти два признака. Мы не смогли построить модель, используя время в формате POSIX, поэтому мы отбрасываем этот признак. Для начала давайте попробуем время суток. Как показывает рис. 4.14, теперь прогнозы имеют одинаковую структуру для каждого дня недели:

```
In[56]:
X_hour = citibike.index.hour.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

```
Out[56]:
R^2 для тестового набора: 0.60
```

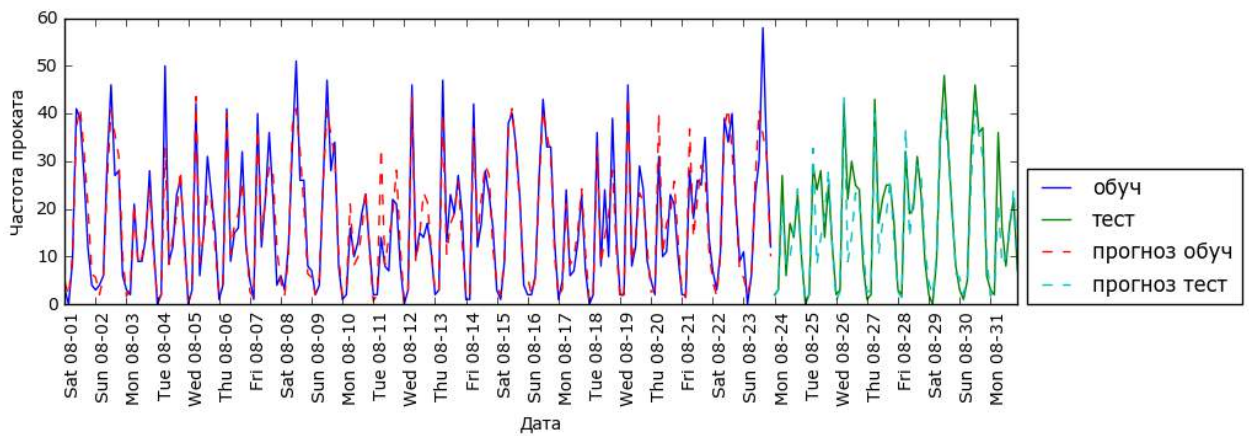


**Рис. 4.14** Прогнозы, вычисленные случайным лесом (использовалось время суток)

Значение  $R^2$  стало уже намного лучше, но прогнозы явно не учитывают эффект, обусловленный днем недели. Теперь давайте еще добавим день недели (см. рис. 4.15):

```
In[57]:
X_hour_week = np.hstack([citibike.index.dayofweek.reshape(-1, 1),
                          citibike.index.hour.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

```
Out[57]:
R^2 для тестового набора: 0.84
```

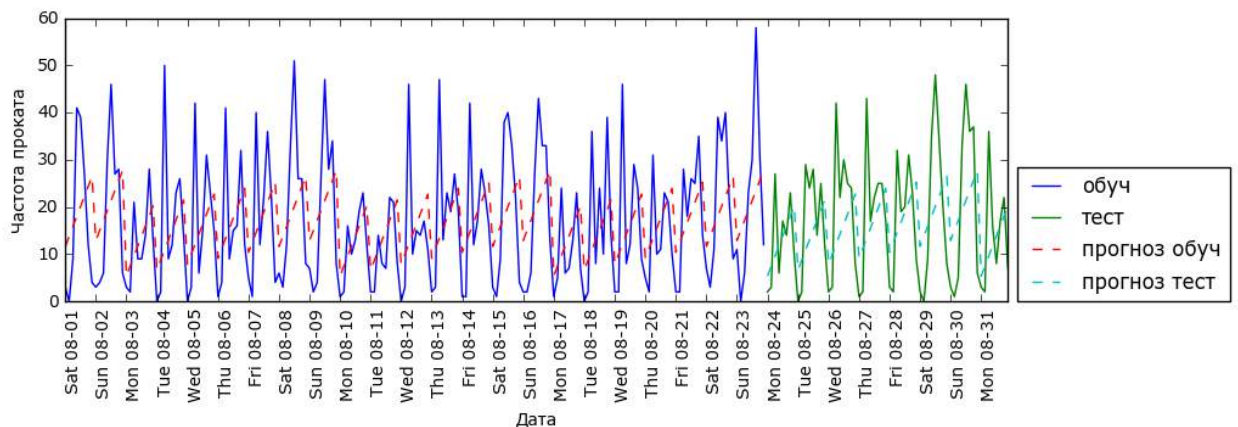


**Рис. 4.15** Прогнозы, вычисленные случайным лесом (использовались день недели и время суток)

Теперь у нас есть модель, которая отражает периодичность поведения, учитывая день недели и время суток. Она имеет значение  $R^2$ , равное 0.84, и демонстрирует довольно хорошую прогностическую способность. Модель научилась прогнозировать среднее количество арендованных велосипедов для каждой комбинации дня недели и времени суток на основе выборки, включающей первые 23 дня августа. На самом деле эта задача не требует такой сложной модели, как случайный лес, поэтому давайте попробуем более простую модель, например, `LinearRegression` (см. рис. 4.16):

```
In[58]:
from sklearn.linear_model import LinearRegression
eval_on_features(X_hour_week, y, LinearRegression())
```

```
Out[58]:
R^2 для тестового набора: 0.13
```



**Рис. 4.16** Прогнозы, вычисленные линейной регрессией (использовались день недели и время суток)

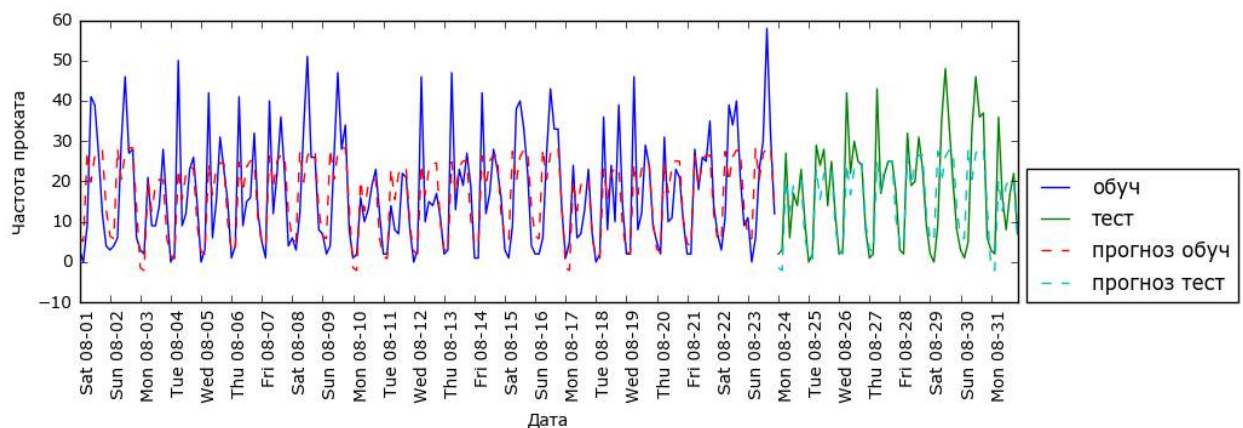
`LinearRegression` работает гораздо хуже, а периодическая структура данных выглядит странно. Причиной этого является тот факт, что мы закодировали день недели и время суток с помощью целочисленных

значений и теперь эти характеристики интерпретируются как непрерывные переменные. В силу этого линейная модель может построить лишь линейную функцию от времени суток – в более позднее время суток наблюдается большее количество арендованных велосипедов. Однако структура данных сложнее, чем предполагает модель. Мы можем учесть это, преобразовав признаки, закодированные целыми числами, в дамми-переменные с помощью `OneHotEncoder` (см. рис. 4.17):

```
In[59]:
enc = OneHotEncoder()
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

```
In[60]:
eval_on_features(X_hour_week_onehot, y, Ridge())
```

```
Out[60]:
R^2 для тестового набора: 0.62
```



**Рис. 4.17** Прогнозы, вычисленные линейной регрессией (использовалось прямое кодирование времени суток и дня недели)

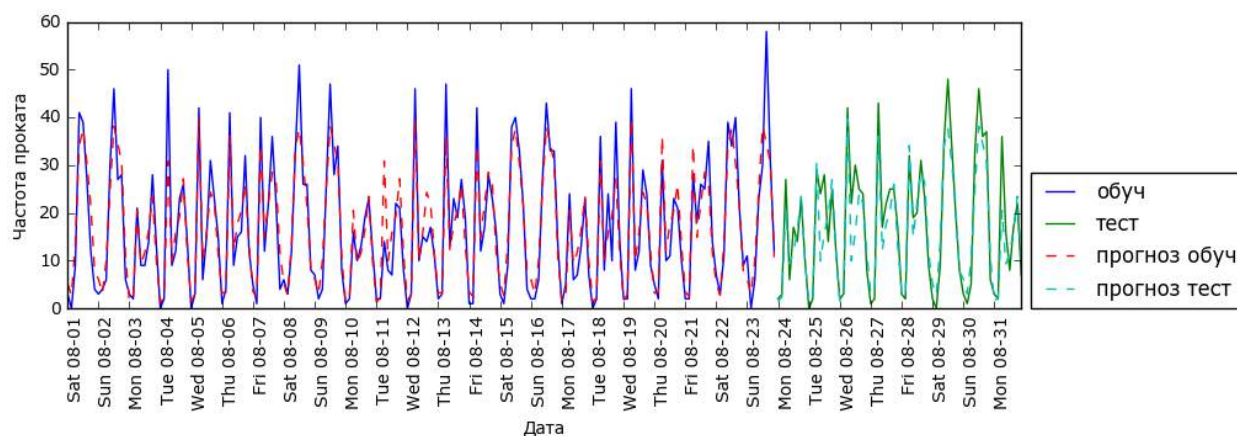
Данная процедура дает гораздо лучший результат в отличие от кодирования наших признаков в виде непрерывных переменных. Теперь линейная модель вычисляет один коэффициент для каждого дня недели и один коэффициент для каждого времени суток. Тем не менее, это означает, что паттерн «времени суток» распределяется по всем дням недели.

Используя взаимодействия, мы можем вычислить коэффициент для каждой комбинации дня недели и времени суток (см. рис. 4.18):

```
In[61]:
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True,
                                     include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

```
Out[61]:
R^2 для тестового набора: 0.85
```





**Рис. 4.18** Прогнозы, вычисленные линейной регрессией (использовались взаимодействия дня недели и времени суток)

Наконец, это преобразование дает модель, которая обладает такой же высокой прогностической способностью, что и случайный лес. Большим преимуществом данной модели является ее понятность: мы вычисляем по одному коэффициенту для каждой комбинации дня недели и времени суток. Мы можем просто построить график коэффициентов, вычисленных с помощью модели, что было бы невозможно для случайного леса.

Во-первых, мы создаем имена для наших признаков:

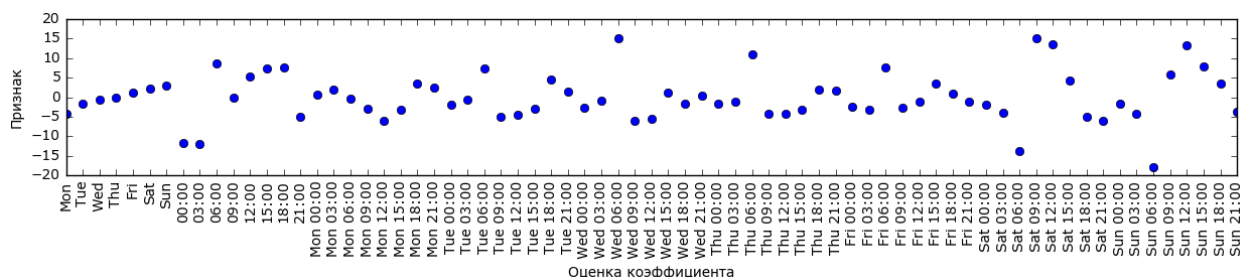
```
In[62]:
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features = day + hour
```

Затем мы присваиваем имена всем взаимодействиям, извлеченным с помощью `PolynomialFeatures`, используя метод `get_feature_names`, и сохраняем лишь те признаки, у которых коэффициенты отличны от нуля:

```
In[63]:
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

Теперь мы можем визуализировать коэффициенты, извлеченные с линейной модели (показаны на рис. 4.19):

```
In[64]:
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Оценка коэффициента")
plt.ylabel("Признак")
```



**Рис. 4.19** Коэффициенты линейной регрессии  
(использовались взаимодействия дня недели и времени суток)

## Выводы и перспективы

В этой главе мы рассмотрели способы обработки различных типов данных (в частности, обработку категориальных переменных). Мы подчеркнули важность представления данных таким способом, который в наибольшей степени подходит для алгоритма машинного обучения, например, рассмотрели прямое кодирование категориальных переменных. Мы также обсудили важность конструирования новых признаков, а также возможность применения экспертных знаний при разработке новых переменных на основе ваших данных.

Создание новых признаков с помощью биннинга, добавления полиномов и взаимодействий может значительно улучшить качество линейных моделей, тогда как более сложные, нелинейные модели типа случайного леса и SVM могут решать более трудные задачи без явного расширения пространства признаков. На практике наличие признаков, подходящих для использования (а также их соответствие применяемой модели), часто является самым важным элементом, обеспечивающим хорошую работу методов машинного обучения.

Теперь у вас есть развернутое представление о том, как закодировать данные надлежащим образом и какой алгоритм использовать для решения определенной задачи. Следующая глава будет посвящена оценке качества моделей машинного обучения и выбору правильных параметров.

## ГЛАВА 5. ОЦЕНКА И УЛУЧШЕНИЕ КАЧЕСТВА МОДЕЛИ

Обсудив основы машинного обучения с учителем и без учителя, теперь мы еще сильнее погрузимся в вопросы, связанные с оценкой моделей и выбором параметров.

Мы сосредоточимся на методах машинного обучения с учителем, регрессии и классификации, поскольку оценка качества и выбор моделей машинного обучения без учителя часто представляют собой очень субъективную процедуру (как мы убедились в главе 3).

Вплоть до настоящего момента для оценки качества модели мы разбивали наши данные на обучающий и тестовый наборы с помощью функции `train_test_split`, строили модель на обучающей выборке, вызвав метод `fit`, и оценивали ее качество на тестовом наборе, используя метод `score`, который для классификации вычисляет долю правильно классифицированных примеров. Вот пример вышеописанной последовательности действий:

```
In[2]:
from sklearn.datasets import make_blobs
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# создаем синтетический набор данных
X, y = make_blobs(random_state=0)
# разобьем данные на обучающий и тестовый наборы
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# создаем экземпляр модели и подгоняем его на обучающем наборе
logreg = LogisticRegression().fit(X_train, y_train)
# оцениваем качество модели на тестовом наборе
print("Правильность на тестовом наборе: {:.2f}".format(logreg.score(X_test, y_test)))

Out[2]:
Правильность на тестовом наборе: 0.88
```

Вспомним, что причина, по которой мы разбиваем наши данные на обучающий и тестовый наборы, заключается в том, что нас интересует, насколько хорошо наша модель *обобщает* результат на новые, ранее неизвестные данные. Нас интересует не качество подгонки модели к обучающим данным, а правильность ее прогнозов для данных, не участвовавших в обучении.

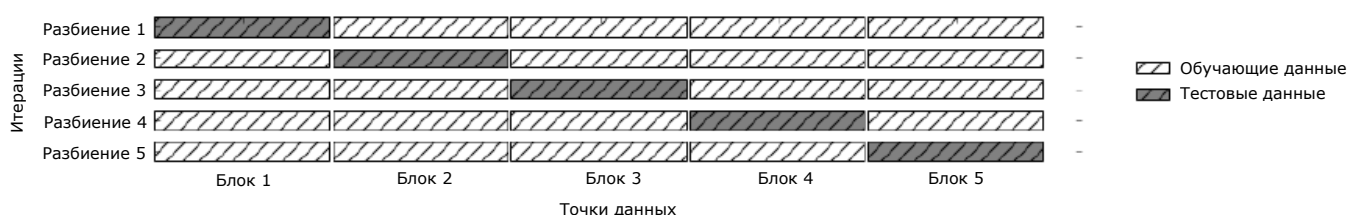
В этой главе мы подробнее остановимся на двух аспектах этой оценки. Сначала мы расскажем о перекрестной проверке, более надежном способе оценки обобщающей способности, а также рассмотрим методы оценки обобщающей способности для классификации и регрессии, которые выходят за рамки традиционных показателей правильности и  $R^2$ , предусмотренных методом `fit`.



Кроме того, мы рассмотрим *решетчатый поиск* (*grid search*), эффективный метод, который предназначен для корректировки параметров в моделях контролируемого машинного обучения с целью получения наилучшей обобщающей способности.

## Перекрестная проверка

Перекрестная проверка представляет собой статистический метод оценки обобщающей способности, который является более устойчивым и основательным, чем разбиение данных на обучающий и тестовый наборы. В перекрестной проверке данные разбиваются несколько раз и строится несколько моделей. Наиболее часто используемый вариант перекрестной проверки – *k-блочная кросс-проверка* (*k-fold cross-validation*), в которой  $k$  – это задаваемое пользователем число, как правило, 5 или 10. При выполнении пятиблочной перекрестной проверки данные сначала разбиваются на пять частей (примерно) одинакового размера, называемых *блоками* (*folds*) складками. Затем строится последовательность моделей. Первая модель обучается, используя блок 1 в качестве тестового набора, а остальные блоки (2-5) выполняют роль обучающего набора. Модель строится на основе данных, расположенных в блоках 2-5, а затем на данных блока 1 оценивается ее правильность. Затем происходит обучение второй модели, на этот раз в качестве тестового набора используется блок 2, а данные в блоках 1, 3, 4, и 5 служат обучающим набором. Этот процесс повторяется для блоков 3, 4 и 5, выполняющих роль тестовых наборов. Для каждого из этих пяти *разбиений* (*splits*) данных на обучающий и тестовый наборы мы вычисляем правильность. В итоге мы зафиксировали пять значений правильности. Процесс показан на рис. 5.1:



**Рис. 5.1** Разбиение данных в пятиблочной перекрестной проверке

Как правило, первая пятая часть данных формирует первый блок, вторая пятая часть данных формирует второй блок и так далее.

## Перекрыстная проверка в scikit-learn

В `scikit-learn` перекрыстная проверка реализована с помощью функции `cross_val_score` модуля `model_selection`. Аргументами функции `cross_val_score` являются оцениваемая модель, обучающие данные и фактические метки. Давайте оценим качество модели `LogisticRegression` на наборе данных `iris`:

```
In[4]:
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Значения правильности перекрыстной проверки: {}".format(scores))

Out[4]:
Значения правильности перекрыстной проверки: [ 0.961 0.922 0.958]
```

По умолчанию `cross_val_score` выполняет трехблочную перекрыстную проверку, возвращая три значения правильности. Мы можем изменить количество блоков, задав другое значение параметра `cv`:

```
In[5]:
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Значения правильности перекрыстной проверки: {}".format(scores))

Out[5]:
Значения правильности перекрыстной проверки: [ 1.  0.967 0.933 0.9  1. ]
```

Наиболее распространенный способ подытожить правильность, вычисленную в ходе перекрыстной проверки, — это вычисление среднего значения:

```
In[6]:
print("Средняя правильность перекрыстной проверки: {:.2f}".format(scores.mean()))

Out[6]:
Средняя правильность перекрыстной проверки: 0.96
```

Используя усредненное значение правильности для перекрыстной проверки, мы можем сделать вывод, что средняя правильность модели составит примерно 96%. Взглянув на все пять значений правильности, полученных в ходе пятиблочной перекрыстной проверки, можно еще сделать вывод о том, что существует относительно высокий разброс значений правильности, вычисленных для блоков, от 100% до 90%. Подобный результат может означать, что модель сильно зависит от конкретных блоков, использованных для обучения, а также это может быть обусловлено небольшим размером набора данных.

## Преимущества перекрестной проверки

По сравнению с однократным разбиением данных на обучающий и тестовый наборы использование перекрестной проверки имеет несколько преимуществ. Во-первых, вспомним что `train_test_split` выполняет случайное разбиение данных. Представьте себе, что при выполнении случайного разбиения данных нам «повезло», и все трудно классифицируемые примеры в конечном итоге попали в обучающий набор. В этом случае в тестовый набор попадут только «легкие» примеры, и правильность на тестовом наборе будет неправдоподобно высокой. И, наоборот, если нам «не повезло», все трудно классифицируемые примеры попадают в тестовый набор и поэтому мы получаем неправдоподобно низкую правильность. Однако при использовании перекрестной проверки на каждой итерации в тестовый набор, использующийся для проверки модели, попадают разные примеры. Таким образом, модель должна хорошо обобщать все примеры в наборе данных, чтобы все значения правильности (или их среднее) были высокими.

Кроме того, наличие нескольких разбиений дает определенную информацию о том, насколько наша модель чувствительна к выбору обучающего набора данных. Для набора данных `iris` мы увидели разброс значений правильности от 90% до 100%. Это довольно широкий диапазон значений и он позволяет нам судить о том, как модель будет работать в худшем и лучшем случае, когда мы применим ее к новым данным.

Еще одно преимущество перекрестной проверки по сравнению с однократным разбиением данных заключается в том, что мы используем наши данные более эффективно. Применяя `train_test_split`, мы обычно используем 75% данных для обучения и 25% данных для оценки качества. Применяя пятиблочную перекрестную проверку, на каждой итерации для подгонки модели мы можем использовать 4/5 данных (80%). При использовании 10-блочной перекрестной проверки мы можем использовать для подгонки модели 9/10 данных (90%). Большой объем данных, как правило, приводит к построению более точных моделей.

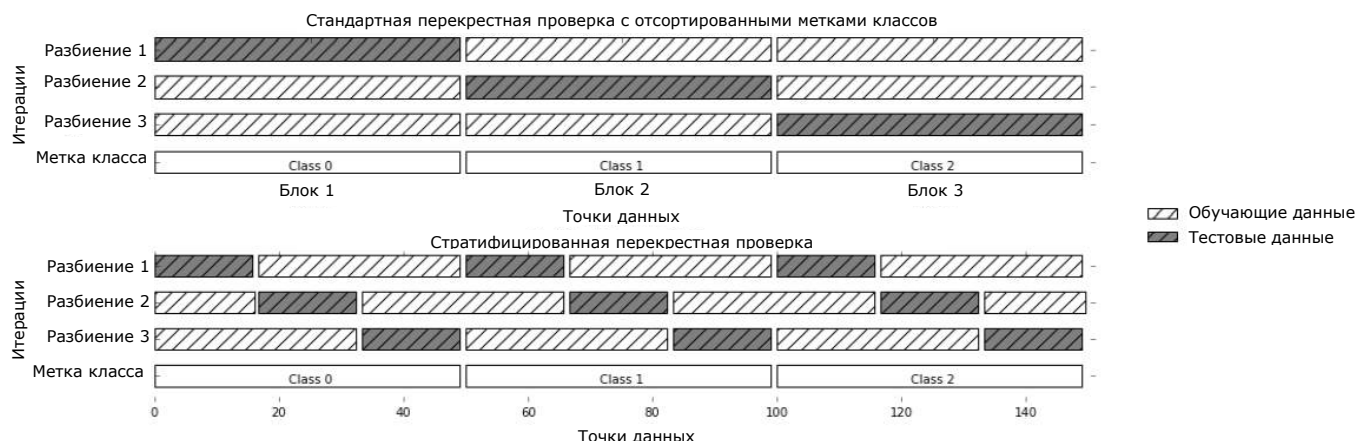
Основной недостаток перекрестной проверки – увеличение стоимости вычислений. Поскольку теперь мы обучаем  $k$  моделей вместо одной модели, перекрестная проверка будет выполняться примерно в  $k$  раз медленнее, чем однократное разбиение данных.



Важно помнить, что кросс-валидация не является способом построения модели, которую можно применить к новым данным. Перекрестная проверка не возвращает модель. При вызове `cross_val_score` строится несколько внутренних моделей, однако цель перекрестной проверки заключается только в том, чтобы оценить обобщающую способность данного алгоритма, обучив на определенном наборе данных.



```
In[8]:
mglearn.plots.plot_stratified_cross_validation()
```



**Рис. 5.2** Сравнение стандартной перекрестной проверки и стратифицированной перекрестной проверки, когда данные упорядочены по меткам классов

Например, если 90% примеров относятся к классу А, а 10% примеров – к классу В, то стратифицированная перекрестная проверка выполняется так, чтобы в каждом блоке 90% примеров принадлежали к классу А, а 10% примеров – к классу В.

Использование для оценки классификатора стратифицированной  $k$ -блочной перекрестной проверки вместо обычной  $k$ -блочной перекрестной является хорошей идеей, поскольку позволяет получить более надежные оценки обобщающей способности. В ситуации, когда лишь 10% примеров принадлежат к классу В, использование стандартной  $k$ -блочной перекрестной проверки может привести к тому, что один из блоков будет полностью состоять из примеров, относящихся к классу А. Использование этого блока в качестве тестового набора не даст особой информации о качестве работы классификатора.

Для регрессии в **scikit-learn** по умолчанию используется стандартная  $k$ -блочная кросс-проверка. Можно было бы еще попытаться создать блоки, представляющие различные значения количественной зависимой переменной, но данный метод не является общераспространенной стратегией и был бы неожиданностью для большинства пользователей.

## Больше контроля над перекрестной проверкой

Ранее мы уже видели, что можно настроить количество блоков, используемое в `cross_val_score`, с помощью параметра `cv`. Однако **scikit-learn** позволяет значительно точнее настроить процесс перекрестной проверки, используя в качестве параметра `cv` *генератор разбиений перекрестной проверки* (*cross-validation splitter*). В большинстве случаев значения параметров, выставленные по умолчанию

для  $k$ -блочной перекрестной проверки в случае регрессии и стратифицированной  $k$ -блочной проверки в случае классификации дают хорошие результаты, однако бывают ситуации, когда вы, возможно, захотите использовать другую стратегию. Допустим, мы хотим применить  $k$ -блочную перекрестную проверку к классификационному набору данных, чтобы воспроизвести чьи-то результаты. Для этого мы должны сначала импортировать класс `KFold` из модуля `model_selection` и создать его экземпляр, задав нужное количество блоков:

```
In[9]:
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
```

Затем мы можем передать генератор разбиений `kfold` в качестве параметра `cv` в функцию `cross_val_score`.

```
In[10]:
print("Значения правильности перекрестной проверки:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

```
Out[10]:
Значения правильности перекрестной проверки:
[ 1.  0.933  0.433  0.967  0.433]
```

Таким образом, мы можем убедиться, что использование трехблочной (нестратифицированной) перекрестной проверки для набора данных `iris` действительно является очень плохой идеей:

```
In[11]:
kfold = KFold(n_splits=3)
print("Значения правильности перекрестной проверки:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

```
Out[11]:
Значения правильности перекрестной проверки:
[ 0.  0.  0.]
```

Вспомним, что в наборе данных `iris` каждый блок соответствует одному классу и поэтому, применив нестратифицированную перекрестную проверку, мы ничего не сможем узнать о правильности модели. Еще один способ решения этой проблемы состоит в том, чтобы вместо стратификации перемешать данные и тем самым нарушить порядок сортировки примеров, определяемый их метками. Мы можем сделать это, передав генератору `KFold` параметр `shuffle=True`. Если мы перемешиваем данные, нам необходимо зафиксировать `random_state`, чтобы воспроизвести результат перемешивания. В противном случае каждый прогон `cross_val_score` будет давать разный результат, поскольку каждый раз используется разное разбиение (это не является проблемой, но может привести к неожиданным результатам). Перемешивание данных перед их разбиением дает гораздо лучший результат:



```
In[12]:
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Значения правильности перекрестной проверки:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

```
Out[12]:
Значения правильности перекрестной проверки:
[ 0.9 0.96 0.96]
```

## Перекрестная проверка с исключением по одному

Еще один часто используемый метод перекрестной проверки – *исключение по одному (leave-one-out)*. Перекрестную проверку с исключением по одному можно представить в виде  $k$ -блочной перекрестной проверки, в которой каждый блок представляет собой отдельный пример. По каждому разбиению вы выбираете одну точку данных в качестве тестового набора. Этот вид проверки может занимать очень много времени, особенно при работе с большими наборами данных, однако иногда позволяет получить более точные оценки на небольших наборах данных:

```
In[13]:
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Количество итераций: ", len(scores))
print("Средняя правильность: {:.2f}".format(scores.mean()))
```

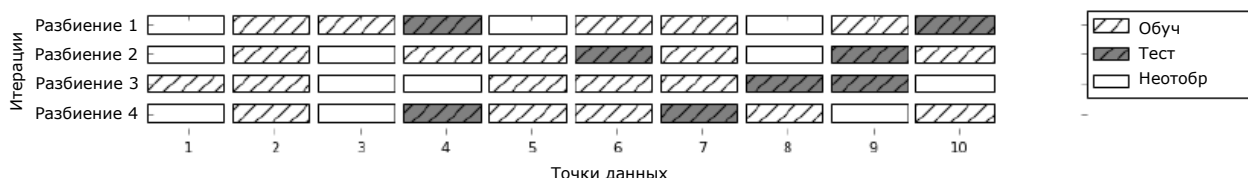
```
Out[13]:
Количество итераций: 150
Средняя правильность: 0.95
```

## Перекрестная проверка со случайными перестановками при разбиении

Еще одной, очень гибкой стратегией перекрестной проверки является *перекрестная проверка со случайными перестановками при разбиении (shuffle-split cross-validation)*. В этом виде проверки каждое разбиение выбирает **train\_size** точек для обучающего набора и **test\_size** точек для тестового набора (при этом обучающее и тестовое подмножества не пересекаются). Точки выбираются с возвращением. Разбиение повторяется **n\_iter** раз. Рис. 5.3 иллюстрирует четырехпроходное разбиение набора данных, состоящего из 10 точек, на обучающий набор из 5 точек и тестовый набор из 2 точек (чтобы задать абсолютные размеры этих подмножеств вы можете использовать для **train\_size** и **test\_size** целочисленные значения, либо числа с плавающей точкой, чтобы задать доли от общей выборки):

```
In[14]:
mglearn.plots.plot_shuffle_split()
```





**Рис. 5.3** Перекрестная проверка со случайными перестановками при разбиении для набора данных из 10 точек, `train_size=5`, `test_size=2` и `n_iter=4`

Программный код, приведенный ниже, 10 раз разбивает данные на 50%-ный обучающий набор и 50%-ный тестовый набор:

```
In[15]:
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Значения правильности перекрестной проверки:\n{}".format(scores))
```

```
Out[15]:
Значения правильности перекрестной проверки:
[ 0.96  0.907  0.947  0.96  0.96  0.907  0.893  0.907  0.92  0.973]
```

Перекрестная проверка со случайными перестановками при разбиении позволяет задавать количество итераций независимо от размеров обучающего и тестового наборов, что иногда может быть полезно. Кроме того, этот метод позволяет использовать на каждой итерации лишь часть данных (значения `train_size` и `test_size` необязательно должны в сумме давать 1). Подобное прореживание данных может быть полезно при работе с большими наборами данных.

Существует также стратифицированный вариант `ShuffleSplit`, названный `StratifiedShuffleSplit`, который позволяет получить более надежные результаты при решении задач классификации.

### Перекрестная проверка с использованием групп

Еще одна весьма распространенная настройка для перекрестной проверки применяется, когда данные содержат сильно взаимосвязанные между собой группы. Допустим, вы хотите построить систему распознавания эмоций по фотографиям лиц и для этого вы собрали набор изображений 100 человек, в котором каждый человек сфотографирован несколько раз, чтобы зафиксировать разные эмоции. Цель заключается в том, чтобы построить классификатор, который сможет правильно определить эмоции людей, не включенных в этот набор изображений. В данном случае для оценки качества работы классификатора вы можете использовать традиционную стратифицированную перекрестную проверку. Однако, вполне вероятно, что фотографии одного и того же человека попадут как в обучающий, так и в тестовый наборы. По сравнению с совершенно новым лицом

классификатору намного проще будет определить эмоции по лицу, которое уже встречалось ему в обучающем наборе. Чтобы точно оценить способность модели обобщать результат на новые лица, необходимо убедиться в том, что обучающий и тестовый наборы содержат изображения разных людей.

Для решения этой задачи мы можем воспользоваться `GroupKFold`, принимающий в качестве аргумента массив `groups`. С помощью него мы указываем, какой человек изображен на снимке. В данном случае массив `groups` указывает группы данных, которые не следует разбивать при создании обучающего и тестового наборов, при этом их не следует путать с метками классов.

Подобные группы данных часто встречаются в медицинской практике, когда у вас, возможно, есть несколько наблюдений по одному и тому же пациенту, но вы заинтересованы в обобщении результатов на новых пациентов. Аналогично в задачах распознавания речи у вас может быть несколько записей одного и того же человека, но вас интересует точность распознавания речи новых людей.

Ниже приведен пример с использованием синтетического набора данных, группировка данных задана массивом `groups`. Набор данных состоит из 12 точек данных, и для каждой точки массив `groups` задает группу (допустим, пациента), к которой относится эта точка. У нас существуют четыре группы, первые три примера принадлежат к первой группе, следующие четыре примера принадлежат ко второй группе и так далее:

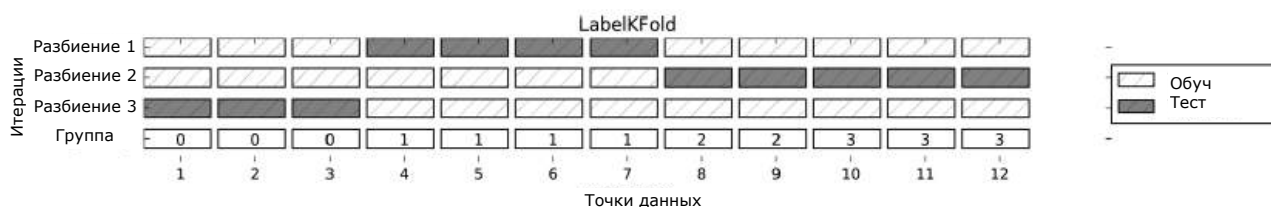
```
In[17]:
from sklearn.model_selection import GroupKFold
# создаем синтетический набор данных
X, y = make_blobs(n_samples=12, random_state=0)
# предположим, что первые три примера относятся к одной и той же группе,
# затем следующие четыре и так далее.
groups = [0, 0, 0, 1, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Значения правильности перекрестной проверки:\n{}".format(scores))
```

```
Out[17]:
Значения правильности перекрестной проверки:
[ 0.75  0.8   0.667]
```

Примеры не нужно сортировать по группам, мы сделали это в иллюстративных целях. Разбиения, вычисляемые на основе этих меток, показаны на рис. 5.4.

Видно, что при выполнении разбиения каждая группа полностью попадает либо в обучающий набор, либо в тестовый набор:

```
In[16]:
mglearn.plots.plot_label_kfold()
```



**Рис. 5.4** Разбиение на основе меток групп с помощью GroupKFold

В библиотеке `scikit-learn` есть и другие стратегии разбиения данных для перекрестной проверки, которые предполагают еще большее разнообразие вариантов использования (вы можете найти их в [руководстве пользователя по scikit-learn](#)). Однако стандартные стратегии `KFold`, `StratifiedKFold` и `GroupKFold` на сегодняшний день используются чаще всего.

## Решетчатый поиск

Теперь, когда мы знаем, как оценивать обобщающую способность, мы можем сделать следующий шаг и улучшить обобщающую способность модели, настроив ее параметры. В главах 2 и 3 мы обсуждали настройки параметров различных алгоритмов в `scikit-learn`, однако важно понять содержательный смысл этих параметров, прежде чем пытаться корректировать их. Поиск оптимальных значений ключевых параметров модели (то есть значений, которые дают наилучшую обобщающую способность) является сложной задачей, но она обязательна почти для всех моделей и наборов данных. Поскольку поиск оптимальных значений параметров является общераспространенной задачей, библиотека `scikit-learn` предлагает стандартные методы, позволяющие решить ее. Наиболее часто используемый метод – это *решетчатый поиск* (*grid search*), который по сути является попыткой перебрать все возможные комбинации интересующих параметров.

Рассмотрим применение ядерного метода SVM с ядром RBF (радиальной базисной функцией), реализованного в классе `SVC`. Как мы уже говорили в главе 2, в ядерном методе опорных векторов есть два важных параметра: ширина ядра `gamma` и параметр регуляризации `C`. Допустим, мы хотим попробовать значения `0.001`, `0.01`, `0.1`, `1`, `10` и `100` для параметра `C` и то же самое для параметра `gamma`. Поскольку нам нужно попробовать шесть различных настроек для `C` и `gamma`, получается 36 комбинаций параметров в целом. Все возможные комбинации формируют таблицу (которую еще называют решеткой или сеткой) настроек параметров для SVM, как показано ниже:

	C = 0.001	C = 0.01	...	C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	...	SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	...	SVC(C=10, gamma=0.01)
...	...	...	...	...
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	...	SVC(C=10, gamma=100)

## Простой решетчатый поиск

Мы можем реализовать простой решетчатый поиск с помощью вложенных циклов `for` по двум параметрам, обучая и оценивая классификатор для каждой комбинации:

```
In[18]:
# реализация наивного решетчатого поиска
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
print("Размер обучающего набора: {} размер тестового набора: {}".format(
    X_train.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # для каждой комбинации параметров обучаем SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # оцениваем качество SVC на тестовом наборе
        score = svm.score(X_test, y_test)
        # если получаем наилучшее значение правильности, сохраняем значение и параметры
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Наилучшее значение правильности: {:.2f}".format(best_score))
print("Наилучшие значения параметров: {}".format(best_parameters))

Out[18]:
Размер обучающего набора: 112 размер тестового набора: 38
Наилучшее значение правильности: 0.97
Наилучшие значения параметров: {'C': 100, 'gamma': 0.001}
```

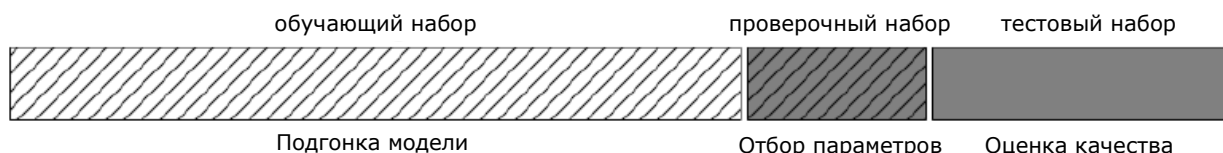
## Опасность переобучения параметров и проверочный набор данных

Получив такой результат, мы могли бы поддаваться искушению и заявить, что нашли модель, которая дает 97%-ную правильность на нашем наборе данных. Однако это заявление может быть чрезмерно оптимистичным (или просто неверным) по следующей причине: мы перебрали множество значений параметров и выбрали ту комбинацию значений, которая дает наилучшую правильность на тестовом наборе, но это вовсе не означает,

что на новых данных мы получим такое же значение правильности. Поскольку мы использовали тестовый набор для настройки параметров, мы больше не можем использовать его для оценки качества модели. Это та же самая причина, по которой нам изначально нужно разбивать данные на обучающий и тестовый наборы. Теперь для оценки качества модели нам необходим независимый набор данных, то есть набор, который не использовался для построения модели и настройки ее параметров.

Один из способов решения этой проблемы заключается в том, чтобы разбить данные еще раз, таким образом, мы получаем три набора: обучающий набор для построения модели, проверочный (валидационный) набор для выбора параметров модели, а также тестовый набор для оценки качества работы выбранных параметров. Рис. 5.5 показывает, как это выглядит:

```
In[19]:
mglearn.plots.plot_threefold_split()
```



**Рис. 5.5** Тройное разбиение данных на обучающий набор, проверочный набор и тестовый набор

После выбора наилучших параметров с помощью проверочного набора проверки, мы можем заново построить модель, используя найденные настройки, но теперь на основе объединенных обучающих и проверочных данных. Таким образом, мы можем использовать для построения модели максимально возможное количество данных. Это приводит к следующему программному коду:

```
In[20]:
from sklearn.svm import SVC
# разбиваем данные на обучающий+проверочный набор и тестовый набор
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# разбиваем обучающий+проверочный набор на обучающий и проверочный наборы
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("Размер обучающего набора: {} размер проверочного набора: {} размер тестового набора:"
      "{}\n".format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # для каждой комбинации параметров обучаем SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # оцениваем качество SVC на тестовом наборе
        score = svm.score(X_valid, y_valid)
        # если получаем наилучшее значение правильности, сохраняем значение и параметры
```

```

    if score > best_score:
        best_score = score
        best_parameters = {'C': C, 'gamma': gamma}
# заново строим модель на наборе, полученном в результате объединения обучающих
# и проверочных данных, оцениваем качество модели на тестовом наборе
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
print("Лучшее значение правильности на проверочном наборе: {:.2f}".format(best_score))
print("Наилучшие значения параметров: ", best_parameters)
print("Правильность на тестовом наборе с наилучшими параметрами: {:.2f}".format(test_score))

Out[20]:
Размер обучающего набора: 84 размер проверочного набора: 28 размер тестового набора: 38
Лучшее значение правильности на проверочном наборе: 0.96
Наилучшие значения параметров: {'C': 10, 'gamma': 0.001}
Правильность на тестовом наборе с наилучшими параметрами: 0.92

```

Лучшее значение правильности на проверочном наборе составляет 96%, что немного ниже значения правильности, полученного для тестового набора ранее, вероятно, из-за того, что мы использовали меньше данных для обучения модели (размер `X_train` теперь стал меньше, поскольку что мы разбили наш набор данных дважды). Однако значение правильности на тестовом наборе, значение, которое показывает реальную обобщающую способность – стало еще ниже, 92%. Таким образом, мы можем утверждать, что правильность классификации новых данных составляет 92%, а не 97%, как мы думали ранее!

Наличие различий между обучающим, проверочным и тестовым наборами имеет принципиально важное значение для применения методов машинного обучения на практике. Любой выбор, сделанный, исходя из правильности на тестовом наборе, «сливает» модели информацию тестового набора. Поэтому важно иметь отдельный тестовый набор, который используется лишь для итоговой оценки. Осуществление всего разведочного анализа и отбора модели с помощью комбинации обучающего и проверочного наборов и резервирование тестового набора для итоговой оценки является хорошей практикой. Данная практика является верной даже при проведении разведочной визуализации. Строго говоря, оценка качества моделей и выбор наилучшей из них с помощью тестового набора, использующегося для отбора параметров, приведет к чрезмерно оптимистичной оценке правильности модели.

## Решетчатый поиск с перекрестной проверкой

Хотя только что рассмотренный нами метод разбиения данных на обучающий, проверочный и тестовый наборы является вполне рабочим и относительно широко используемым, он весьма чувствителен к правильности разбиения данных. Взглянув на вывод, приведенный для предыдущего фрагмента программного кода, мы видим, что `GridSearchCV`



выбрал в качестве лучших параметров 'C': 10, 'gamma': 0.001, тогда как вывод, приведенный для программного кода в предыдущем разделе, сообщает нам, что наилучшими параметрами являются 'C': 100, 'gamma': 0.001. Для лучшей оценки обобщающей способности вместо одного разбиения данных на обучающий и проверочный наборы мы можем воспользоваться перекрестной проверкой. Теперь качество модели оценивается для каждой комбинации параметров по всем разбиениям перекрестной проверки. Этот метод можно реализовать с помощью следующего программного кода:

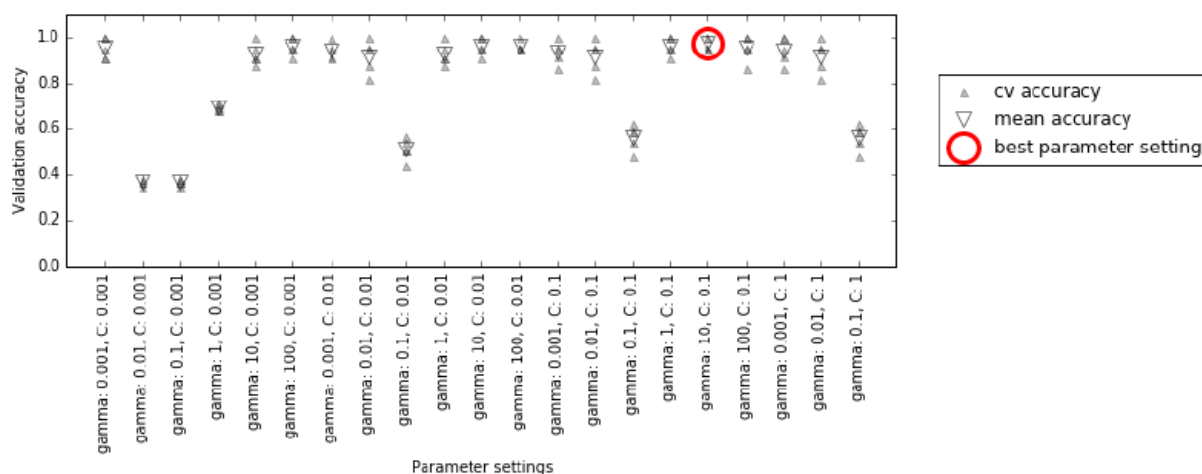
```
In[21]:
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # для каждой комбинации параметров,
        # обучаем SVC
        svm = SVC(gamma=gamma, C=C)
        # выполняем перекрестную проверку
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # вычисляем среднюю правильность перекрестной проверки
        score = np.mean(scores)
        # если получаем лучшее значение правильности, сохраняем значение и параметры
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# заново строим модель на наборе, полученном в результате
# объединения обучающих и проверочных данных
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

Чтобы с помощью пятиблочной перекрестной проверки оценить правильность SVM для конкретной комбинации значений C и gamma, нам необходимо обучить  $36 \times 5 = 180$  моделей. Как вы понимаете, основным недостатком использования перекрестной проверки является время, которое требуется для обучения всех этих моделей.

Следующая визуализация (рис. 5.6) показывает, как в предыдущем программном коде осуществляется выбор оптимальных параметров:

```
In[22]:
mglearn.plots.plot_cross_val_selection()
```





**Рис. 5.6** Результаты решетчатого поиска с перекрестной проверкой

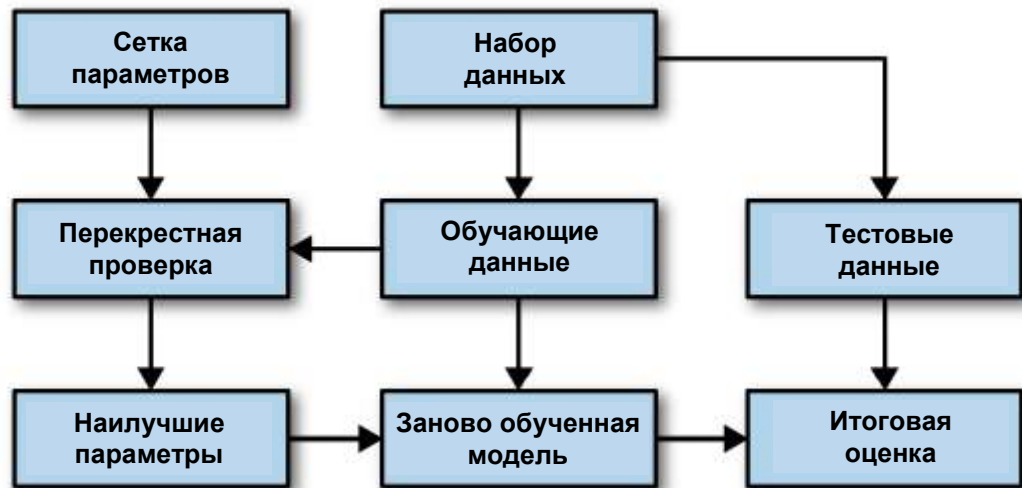
Для каждой комбинации значений  $C$  и  $\gamma$  (здесь показана лишь часть комбинаций) вычисляются пять значений правильности, по одному для каждого разбиения в перекрестной проверке. Затем для каждой комбинации параметров вычисляется среднее значение правильности перекрестной проверки. В итоге выбирается комбинация с наибольшей средней правильностью перекрестной проверки и отмечается кружком.



Как мы уже говорили ранее, перекрестная проверка – это способ оценить качество работы конкретного алгоритма на определенном наборе данных. Однако она часто используется в сочетании с методами поиска параметров типа решетчатого поиска. По этой причине многие люди в разговорной речи под термином *перекрестная проверка* (*cross-validation*) подразумевают решетчатый поиск с перекрестной проверкой.

Общий процесс разбиения данных, запуска решетчатого поиска, а также оценки итоговых параметров показан на рис. 5.7:

In[23]:  
mglearn.plots.plot\_grid\_search\_overview()



**Рис. 5.7** Процесс отбора параметров и оценки модели с помощью GridSearchCV

Поскольку решетчатый поиск с перекрестной проверкой является весьма распространенным методом настройки параметров, библиотека `scikit-learn` предлагает класс `GridSearchCV`, в котором решетчатый поиск реализован в виде модели. Чтобы воспользоваться классом `GridSearchCV`, сначала необходимо указать искомые параметры с помощью словаря. `GridSearchCV` построит все необходимые модели. Ключами словаря являются имена настраиваемых параметров (в данном случае `C` и `gamma`), а значениями – тестируемые настройки параметров. Перебор значений `0.001`, `0.01`, `0.1`, `1`, `10` и `100` для `C` и `gamma` требует словаря следующего вида:

```

In[24]:
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}

print("Сетка параметров:\n{}".format(param_grid))

Out[24]:
Сетка параметров:
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
  
```

Теперь мы можем создать экземпляр класса `GridSearchCV`, передав модель (`SVC`), сетку искомых параметров (`param_grid`), а также стратегию перекрестной проверки, которую мы хотим использовать (допустим, пятиблочную стратифицированную перекрестную проверку):

```

In[25]:
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
  
```

Вместо разбиения на обучающий и проверочный набор, использованного нами ранее, `GridSearchCV` запустит перекрестную проверку. Однако нам по-прежнему нужно разделить данные на обучающий и тестовый наборы, чтобы избежать переобучения параметров:

```
In[26]:
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
```

Созданный нами объект `grid_search` аналогичен классификатору, мы можем вызвать стандартные методы `fit`, `predict` и `score` от его имени.<sup>31</sup> Однако, когда мы вызываем `fit`, он запускает перекрестную проверку для каждой комбинации параметров, указанных в `param_grid`:

```
In[27]:
grid_search.fit(X_train, y_train)
```

Процесс подгонки объекта `GridSearchCV` включает в себя не только поиск лучших параметров, но и автоматическое построение новой модели на всем обучающем наборе данных. Для ее построения используются параметры, которые дают наилучшее значение правильности перекрестной проверки. Поэтому процесс, запускаемый вызовом метода `fit`, эквивалентен программному коду In[21], который мы видели в начале этого раздела. Класс `GridSearchCV` предлагает очень удобный интерфейс для работы с моделью, используя методы `predict` и `score`. Чтобы оценить обобщающую способность найденных наилучших параметров, мы можем вызвать метод `score`:

```
In[28]:
print("Правильность на тестовом наборе: {:.2f}".format(grid_search.score(X_test, y_test)))
```

```
Out[28]:
Правильность на тестовом наборе: 0.97
```

Выбрав параметры с помощью перекрестной проверки, мы фактически нашли модель, которая достигает правильности 97% на тестовом наборе. Главный момент здесь в том, что *мы не использовали тестовый набор* для отбора параметров. Найденная комбинация параметров сохраняется в атрибуте `best_params_`, а наилучшее значение правильности перекрестной проверки (значение правильности, усредненное по всем разбиениям для данной комбинации параметров) – в атрибуте `best_score_`.

---

<sup>31</sup> Модель `scikit-learn`, которая создается с помощью другой модели называется *метамоделью* (*meta-estimator*). `GridSearchCV` является наиболее часто используемой метамоделью, но об этом мы поговорим позже.

```
In[29]:
print("Наилучшие значения параметров: {}".format(grid_search.best_params_))
print("Наилучшее значение кросс-валидац. правильности: {:.2f}".format(grid_search.best_score_))
```

```
Out[29]:
Наилучшие значения параметров: {'C': 100, 'gamma': 0.01}
Наилучшее значение кросс-валидац. правильности: 0.97
```



Опять же, будьте осторожны, чтобы не перепутать `best_score_` со значением обобщающей способности модели, которое вычисляется на тестовом наборе с помощью метода `score`. Метод `score` (оценивающий качество результатов, полученных с помощью метода `predict`) использует модель, *построенную на всем обучающем наборе данных*. В атрибуте `best_score_` записывается средняя правильность перекрестной проверки. Для ее вычисления используется модель, построенная на *обучающем наборе перекрестной проверки*.

В ряде случаев вам необходимо будет ознакомиться с полученной моделью, например, взглянуть на коэффициенты или важности признаков. Посмотреть наилучшую модель, построенную на всем обучающем наборе, вы можете с помощью атрибута `best_estimator_`:

```
In[30]:
print("Наилучшая модель:\n{}".format(grid_search.best_estimator_))
```

```
Out[30]:
Наилучшая модель:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

Поскольку `grid_search` уже сам по себе включает методы `predict` и `score`, использование `best_estimator_` для получения прогнозов и оценки качества модели не требуется.

## Анализ результатов перекрестной проверки

Часто бывает полезно визуализировать результаты перекрестной проверки, чтобы понять, как обобщающая способность зависит от искомых параметров. Поскольку выполнение решетчатого поиска довольно затратно с вычислительной точки зрения, целесообразно начинать поиск с простой и небольшой сетки параметров. Затем мы можем проверить результаты решетчатого поиска, используя перекрестную проверку, и, возможно, расширить наш поиск. Результаты решетчатого поиска можно найти в атрибуте `cv_results`, который является словарем, хранящим все настройки поиска. Как вы можете увидеть в выводе, приведенном ниже, словарь содержит множество

деталей и принимает более привлекательный вид после преобразования в пандасовский `DataFrame`.

```
In[31]:
import pandas as pd
# преобразуем в DataFrame
results = pd.DataFrame(grid_search.cv_results_)
# показываем первые 5 строк
display(results.head())
```

```
Out[31]:
```

	param_C	param_gamma	params	mean_test_score
0	0.001	0.001	{'C': 0.001, 'gamma': 0.001}	0.366
1	0.001	0.01	{'C': 0.001, 'gamma': 0.01}	0.366
2	0.001	0.1	{'C': 0.001, 'gamma': 0.1}	0.366
3	0.001	1	{'C': 0.001, 'gamma': 1}	0.366
4	0.001	10	{'C': 0.001, 'gamma': 10}	0.366

	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	22	0.375	0.347	0.363
1	22	0.375	0.347	0.363
2	22	0.375	0.347	0.363
3	22	0.375	0.347	0.363
4	22	0.375	0.347	0.363

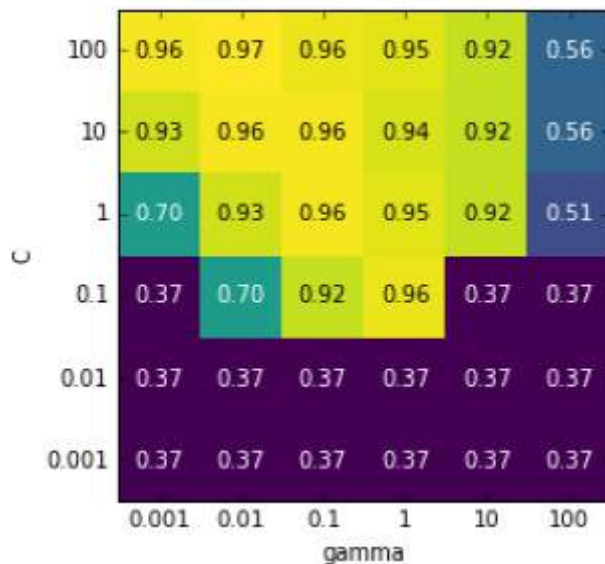
  

	split3_test_score	split4_test_score	std_test_score
0	0.363	0.380	0.011
1	0.363	0.380	0.011
2	0.363	0.380	0.011
3	0.363	0.380	0.011
4	0.363	0.380	0.011

Каждая строка в `results` соответствует одной конкретной комбинации параметров. Для каждой комбинации записываются результаты всех разбиений перекрестной проверки, а также среднее значение и стандартное отклонение по всем разбиениям. Поскольку мы осуществляли поиск на основе двумерной сетки параметров (`C` и `gamma`), наилучший способ визуализировать этот процесс, представить его в виде тепловой карты (рис. 5.8). Сначала мы извлечем средние значения правильности перекрестной проверки, затем изменим форму массива со значениями так, чтобы оси соответствовали `C` и `gamma`:

```
In[32]:
scores = np.array(results.mean_test_score).reshape(6, 6)

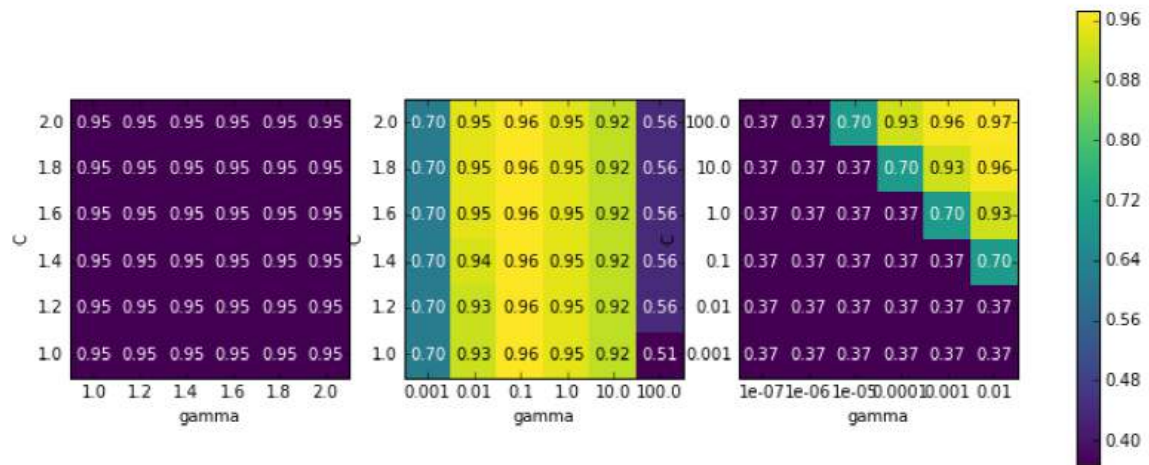
# строим теплокарту средних значений правильности перекрестной проверки
mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param_grid['gamma'],
                      ylabel='C', yticklabels=param_grid['C'], cmap="viridis")
```



**Рис. 5.8** Тепловая карта для усредненной правильности перекрестной проверки, выраженной в виде функции двух параметров  $C$  и  $\gamma$

Каждое значение тепловой карты соответствует средней правильности перекрестной проверки для конкретной комбинации параметров. Цвет передает правильность перекрестной проверки, светлые тона соответствуют высокой правильности, темные тона – низкой правильности. Видно, что SVC очень чувствителен к настройке параметров. Для большинства настроек параметров правильность составляет около 40%, что довольно плохо; для остальных параметров правильность составляет около 96%. Из этого графика мы можем вынести несколько моментов. Во-первых, параметры, которые мы корректировали, *очень важны* для получения хорошей обобщающей способности. Оба параметра ( $C$  и  $\gamma$ ) имеют большое значение, поскольку их корректировка позволяет повысить правильность с 40% до 96%. Кроме того, интервалы значений, которые мы выбрали для параметров, представляют собой диапазоны, в которых мы видим существенные изменения результатов. Кроме того, важно отметить, что диапазоны параметров достаточно велики: оптимальные значения для каждого параметра расположены не по краям, а по центру графика.

Теперь давайте посмотрим еще на несколько графиков (показаны на рис. 5.9), где результат получился менее идеальным, поскольку диапазоны поиска не были подобраны правильно:



**Рис. 5.9** Теплокарты для неправильно подобранных диапазонов поиска

```
In[33]:
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param_grid_linear = {'C': np.linspace(1, 2, 6),
                    'gamma': np.linspace(1, 2, 6)}

param_grid_one_log = {'C': np.linspace(1, 2, 6),
                    'gamma': np.logspace(-3, 2, 6)}

param_grid_range = {'C': np.logspace(-3, 2, 6),
                    'gamma': np.logspace(-7, -2, 6)}

for param_grid, ax in zip([param_grid_linear, param_grid_one_log,
                          param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)

    # строим теплокарту средних значений правильности перекрестной проверки
    scores_image = mglearn.tools.heatmap(
        scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
        yticklabels=param_grid['C'], cmap="viridis", ax=ax)

plt.colorbar(scores_image, ax=axes.tolist())
```

Первый график показывает, что независимо от выбранных параметров никакого изменения правильности не происходит, все значения правильности выделены одним и тем же цветом. В данном случае это вызвано неправильным масштабированием и диапазоном значений параметров **C** и **gamma**. Однако, если различные настройки параметров не приводят к видимому изменению правильности, это еще может указывать на то, что данный параметр просто не важен. Как правило, сначала лучше задать крайние значения, чтобы посмотреть, меняется ли правильность в результате корректировки параметра.

Второй график показывает значения правильности, сгруппированные в виде вертикальных полос. Данный факт указывает на то, что лишь изменение параметра **gamma** влияет на правильность. Это может означать, что для параметра **gamma** заданы более интересные значения, чем для параметра **C**, либо это означает, что параметр **C** не важен.



Третья панель показывает изменения правильности для обеих параметров. Однако мы видим, что в левой нижней части графика ничего интересного не происходит. Вероятно, в будущем мы можем исключить из поиска очень малые значения. Оптимальная комбинация параметров находится в правом верхнем углу. Поскольку оптимальное значение находится на границе графика, можно ожидать, что, вероятно, за пределами этой границы существуют лучшие значения, и мы могли бы изменить наш диапазон поиска, чтобы включить большее количество значений в этой области.

Настройка сетки параметров с помощью перекрестной проверки – это хороший способ исследовать важность различных параметров. Однако, как мы уже обсуждали ранее, значения различных параметров не должны проверяться на итоговом тестовом наборе, качество модели на тестовом наборе должно оцениваться лишь один раз, когда мы точно знаем, какую модель хотим использовать.

### Экономичный решетчатый поиск

В некоторых случаях перебор всех возможных комбинаций по всем параметрам, который обычно выполняет `GridSearchCV`, не является хорошей идеей. Например, `SVC` имеет параметр `kernel`, и в зависимости от того, какое ядро выбрано, все остальные параметры будут иметь соответствующие этому выбору значения. Если `kernel='linear'`, модель является линейной и используется только параметр `C`. Если используется `kernel='rbf'`, используются параметры `C` и `gamma` (однако другие параметры типа `degree` не используются). В этом случае поиск по всем возможным комбинациям `C`, `gamma` и `kernel` не имеет смысла: если `kernel='linear'`, то `gamma` не используется и перебор различных значений `gamma` – это пустая трата времени. Чтобы обработать подобные «условные» параметры, `GridSearchCV` позволяет превратить `param_grid` в *список словарей*. Каждый словарь в списке выделяется в самостоятельную сетку параметров. Возможный решетчатый поиск, включающий настройки ядра и параметров, мог бы выглядеть следующим образом:

```
In[34]:
param_grid = [{'kernel': ['rbf'],
                'C': [0.001, 0.01, 0.1, 1, 10, 100],
                'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
               {'kernel': ['linear'],
                'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
print("List of grids:\n{}".format(param_grid))

Out[34]:
List of grids:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],
  'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
 {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

В первой сетке параметр `kernel` всегда принимает значение `'rbf'` (обратите внимание, элемент параметра `kernel` представляет собой список единичной длины), изменяются значения как параметра `C`, так и параметра `gamma`. Во второй сетке параметр `kernel` всегда принимает значение `'linear'` и поэтому изменяется только параметр `C`. Теперь давайте применим этот более сложный поиск параметров:

```
In[35]:
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Наилучшие значения параметров: {}".format(grid_search.best_params_))
print("Наилучшее значение кросс-валидац. правильности: {:.2f}".format(grid_search.best_score_))
```

```
Out[35]:
Наилучшее значение параметров: {'C': 100, 'kernel': 'rbf', 'gamma': 0.01}
Наилучшее значение кросс-валидац. правильности: 0.97
```

Давайте снова посмотрим на `cv_results_`. Как и следовало ожидать, если `kernel` имеет значение `'linear'`, то меняется только параметр `C`:

```
In[36]:
results = pd.DataFrame(grid_search.cv_results_)
# мы выводим транспонированную таблицу для лучшего отображения на странице:
display(results.T)
```

```
Out[36]:
```

	0	1	2	3	...	38	39	40	41
param_C	0.001	0.001	0.001	0.001	...	0.1	1	10	100
param_gamma	0.001	0.01	0.1	1	...	NaN	NaN	NaN	NaN
param_kernel	rbf	rbf	rbf	rbf	...	linear	linear	linear	linear
params	{C: 0.001, kernel: rbf, gamma: 0.001}	{C: 0.001, kernel: rbf, gamma: 0.01}	{C: 0.001, kernel: rbf, gamma: 0.1}	{C: 0.001, kernel: rbf, gamma: 1}	...	{C: 0.1, kernel: linear}	{C: 1, kernel: linear}	{C: 10, kernel: linear}	{C: 100, kernel: linear}
mean_test_score	0.37	0.37	0.37	0.37	...	0.95	0.97	0.96	0.96
rank_test_score	27	27	27	27	...	11	1	3	3
split0_test_score	0.38	0.38	0.38	0.38	...	0.96	1	0.96	0.96
split1_test_score	0.35	0.35	0.35	0.35	...	0.91	0.96	1	1
split2_test_score	0.36	0.36	0.36	0.36	...	1	1	1	1
split3_test_score	0.36	0.36	0.36	0.36	...	0.91	0.95	0.91	0.91
split4_test_score	0.38	0.38	0.38	0.38	...	0.95	0.95	0.95	0.95
std_test_score	0.011	0.011	0.011	0.011	...	0.033	0.022	0.034	0.034

12 rows x 42 columns

## Применение различных стратегий перекрестной проверки с помощью решетчатого поиска

Как и `cross_val_score`, `GridSearchCV` использует по умолчанию  $k$ -блочную перекрестную проверку для классификации и  $k$ -блочную перекрестную проверку для регрессии. Однако при использовании `GridSearchCV` вы можете дополнительно передать любой генератор разбиения (как было описано в разделе «Больше контроля над перекрестной проверкой») в качестве параметра `cv`. В частности, чтобы получить только одно разбиение на обучающий и проверочный наборы, вы можете воспользоваться `ShuffleSplit` или `StratifiedShuffleSplit` с `n_iter=1`. Данная настройка может оказаться полезной для очень больших наборов данных или очень медленных моделей.

### Вложенная перекрестная проверка

В предыдущих примерах мы прошли путь от использования одного разбиения данных на обучающий, проверочный и тестовый наборы (раздел «Опасность переобучения параметров и проверочный набор данных») до разбиения данных на обучающий и тестовый наборы с проведением перекрестной проверки на обучающем наборе (раздел «Решетчатый поиск с перекрестной проверкой»). Но при использовании `GridSearchCV` ранее описанным способом мы все еще выполняем всего лишь одно разбиение на обучающий и тестовый наборы, что может привести к получению нестабильных результатов и ставит нас в зависимость от этого единственного разбиения данных. Мы можем пойти дальше и вместо однократного разбиения исходных данных на обучающий и тестовый наборы использовать несколько разбиений перекрестной проверки. В результате мы получим *вложенную перекрестную проверку* (*nested cross-validation*). Во вложенной перекрестной проверке используется внешний цикл по разбиениям данных на обучающий и тестовый наборы. Для каждого из них выполняется решетчатый поиск (в результате чего для каждого разбиения внешнего цикла можно получить разные наилучшие параметры). Затем для каждого внешнего разбиения выводится правильность на тестовом наборе с использованием наилучших параметров.

Результатом этой процедуры является не модель и не настройки параметров, а список значений правильности. Значения правильности указывают нам на обобщающую способность модели с использованием лучших параметров, найденных в ходе решетчатого поиска. Поскольку вложенная перекрестная проверка не дает модель, которую можно использовать на новых данных, ее редко используют при поиске прогнозной модели для применения к новым данным. Тем не менее, она

может быть полезна для оценки работы модели на конкретном наборе данных.

Реализовать вложенную перекрестную проверку в `scikit-learn` довольно просто. Мы вызываем `cross_val_score` и передаем ей экземпляр `GridSearchCV` в качестве модели.

```
In[34]:
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                          iris.data, iris.target, cv=5)
print("Значения правильности перекрестной проверки: ", scores)
print("Среднее значение правильности перекрестной проверки: ", scores.mean())

Out[34]:
Значения правильности перекрестной проверки: [ 0.967 1.  0.967 0.967 1. ]
Среднее значение правильности перекрестной проверки: 0.98
```

Результат нашей вложенной перекрестной проверки можно резюмировать так: «на наборе данных `iris` модель `SVC` может достигнуть средней правильности перекрестной проверки 98%» – ни больше, ни меньше.

В данном случае мы использовали стратифицированную пятиблочную перекрестную проверку как во внутреннем, так и во внешнем циклах. Поскольку наша сетка `param_grid` содержит 36 комбинаций параметров, будет построено целых  $36 * 5 * 5 = 900$  моделей, что делает процедуру вложенной перекрестной проверки очень затратной с вычислительной точки зрения. В данном случае во внутреннем и внешнем циклах мы использовали один и тот же генератор разбиений, однако это не является необходимым условием и поэтому для внутреннего и внешнего циклов вы можете использовать любую комбинацию стратегий перекрестной проверки. Понимание процесса, который происходит внутри одной строки, приведенной выше, может представлять определенную сложность. Данный процесс можно визуализировать с помощью циклов `for`, как это сделано в следующей упрощенной реализации программного кода:

```

In[35]:
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    outer_scores = []
    # для каждого разбиения данных во внешней перекрестной проверке
    # (метод split возвращает индексы)
    for training_samples, test_samples in outer_cv.split(X, y):
        # находим наилучшие параметры с помощью внутренней перекрестной проверки
        best_params = {}
        best_score = -np.inf
        # итерируем по параметрам
        for parameters in parameter_grid:
            # собираем значения правильности по всем внутренним разбиениям
            cv_scores = []
            # итерируем по разбиениям внутренней перекрестной проверки
            for inner_train, inner_test in inner_cv.split(
                X[training_samples], y[training_samples]):
                # строим классификатор с данными параметрами на внутреннем обучающем наборе
                clf = Classifier(**parameters)
                clf.fit(X[inner_train], y[inner_train])
                # оцениваем качество на внутреннем тестовом наборе
                score = clf.score(X[inner_test], y[inner_test])
                cv_scores.append(score)
            # вычисляем среднее значение правильности по внутренним блокам
            mean_score = np.mean(cv_scores)
            if mean_score > best_score:
                # если лучше, чем предыдущие, запоминаем параметры
                best_score = mean_score
                best_params = parameters
            # строим классификатор с лучшими параметрами на внешнем обучающем наборе
            clf = Classifier(**best_params)
            clf.fit(X[training_samples], y[training_samples])
            # оцениваем качество на внешнем тестовом наборе
            outer_scores.append(clf.score(X[test_samples], y[test_samples]))
    return np.array(outer_scores)

```

Теперь давайте применим эту функцию к набору данных `iris`:

```

In[36]:
from sklearn.model_selection import ParameterGrid, StratifiedKFold
scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
                    StratifiedKFold(5), SVC, ParameterGrid(param_grid))
print("Значения правильности перекрестной проверки: {}".format(scores))

```

```

Out[36]:
Значения правильности перекрестной проверки: [ 0.967 1. 0.967 0.967 1. ]

```

Распараллеливание перекрестной проверки и решетчатого поиска

Несмотря на то, что выполнение решетчатого поиска с большим количеством параметров на огромных наборах данных может представлять сложность с вычислительной точки зрения, эта задача является *чрезвычайно параллельной* (*embarrassingly parallel*). Это означает, что построение модели с использованием конкретной настройки параметра для конкретного разбиения перекрестной проверки может осуществляться независимо от других настроек параметров и моделей. Данный факт делает решетчатый поиск и перекрестную проверку идеальными кандидатами для распараллеливания по нескольким процессорным ядрам или распараллеливания на кластере. В `GridSearchCV` и `cross_val_score` вы можете использовать несколько процессорных ядер, задав значение параметра `n_jobs` равным нужному

количеству ядер. Вы можете установить `n_jobs=-1`, чтобы использовать все доступные ядра.

Имейте в виду, что `scikit-learn` *не поддерживает вложенность параллельных операций* (*nesting of parallel operations*). Поэтому, если вы используете опцию `n_jobs` для вашей модели (например, для случайного леса), вы не можете использовать ее в `GridSearchCV` для осуществления поиска по этой модели. При работе с большими наборами данных и сложными моделями использование большого числа ядер, возможно, потребует слишком много памяти и вы должны контролировать ее использование памяти при параллельном построении больших моделей.

Кроме того, можно распараллелить решетчатый поиск и перекрестную проверку по нескольким машинам в кластере, хотя на момент написания книги эта возможность в `scikit-learn` не поддерживалась. Однако можно воспользоваться `IPython parallel` для выполнения параллельного решетчатого поиска, если вы согласны писать циклы `for` для параметров, как мы это делали в разделе «Простой решетчатый поиск».

Для пользователей Spark существует недавно разработанный пакет [spark-sklearn](#), который позволяет запускать решетчатый поиск на уже готовом кластере Spark.

## Метрики качества модели и их вычисление

До сих пор мы оценивали качество классификации, используя правильность (долю правильно классифицированных примеров), и качество регрессии, используя  $R^2$ . Однако это лишь два показателя из большого количества возможных метрик, используемых для оценки качества контролируемой модели на данном наборе данных. На практике эти метрики качества могут не соответствовать вашим задачам и поэтому очень важно при отборе моделей и корректировке параметров подобрать правильную метрику.

## Помните о конечной цели

Выбирая метрику, вы всегда должны помнить о конечной цели проекта машинного обучения. На практике мы, как правило, заинтересованы не только в создании точных прогнозов, но и в том, чтобы использовать их в рамках более масштабного процесса принятия решений. Прежде чем выбрать показатель качества машинного обучения, вам стоит подумать о высокоуровневой цели вашего проекта, которую часто называют *бизнес-метрикой* (*business metric*). Последствия, обусловленные выбором конкретного алгоритма для того или иного проекта, называются

влиянием на бизнес (*business impact*).<sup>32</sup> Возможно, высокоуровневой целью является предотвращение дорожно-транспортных происшествий или уменьшение числа случаев госпитализации. Такой целью также может увеличение посещаемости вашего сайта или суммы покупок в вашем магазине. Вы должны выбрать такую модель или такие значения параметров, которые оказывают наибольшее положительное влияние на бизнес-метрику. Часто эта задача является трудной, поскольку оценка влияния конкретной модели на бизнес может потребовать ее внедрения в реальное производство.

Как правило, на ранних этапах разработки, а также при настройке параметров внедрить модель в производство только для тестирования не представляется возможным по причине возникновения высоких коммерческих и человеческих рисков. Представьте себе, что вы, оценивая систему предотвращения столкновения с пешеходами, которой оборудован самопilotируемый автомобиль, просто позволите автомобилю ехать, не проверив его. Если ваша модель имеет низкое качество, пешеходов ждут неприятности! Поэтому нам обычно нужно найти какую-то замещающую процедуру оценки, которая использует легко вычисляемые метрики качества. Например, мы могли бы попробовать классифицировать изображения пешеходов и не-пешеходов и измерить правильность. Помните о том, что данная процедура является замещающей и она оправдывает себя, позволяя найти метрику, максимально близкую к исходной бизнес-цели и поддающуюся оценке. Данная метрика должна использоваться по возможности для оценки и отбора модели. Возможно, что в результате этой процедуры вы не получите какой-то конкретной цифры, например, вывод, найденный с помощью алгоритма, может звучать так: у вас на 10% больше клиентов, но каждый клиент будет тратить на 15% меньше – однако эта процедура должна оценить влияние на бизнес, зависящее от выбора той или иной модели.

В этом разделе мы сначала рассмотрим метрики для бинарной классификации, затем обратимся к мультиклассовой классификации и в заключение обсудим регрессию.

## Метрики для бинарной классификации

Бинарная классификация является, пожалуй, наиболее распространенным и концептуально простым примером практического применения машинного обучения. Однако даже при решении этой

---

<sup>32</sup> Мы просим извинения у научно ориентированных читателей за коммерческий язык в этом разделе. Концентрация на конечной цели в равной степени важна и для науки, правда авторам не знаком аналог фразы «воздействие на бизнес», который мог бы употребляться в этой области.



простой задачи существует целый ряд нюансов. Прежде чем мы углубимся в альтернативные метрики, давайте рассмотрим ситуации, в которых правильность измерения может ввести в заблуждение. Вспомним, что в случае бинарной классификации мы говорим о *положительном (positive)* классе и *отрицательном (negative)* классе, подразумевая под положительным классом интересующий нас класс.

### Типы ошибок

Как правило, правильность не является адекватным показателем прогностической способности, поскольку количество совершаемых ошибок не содержит весь объем интересующей нас информации. Представьте себе скрининговое обследование для раннего обнаружения рака, построенное на основе автоматизированного теста. Если тест отрицателен, пациент будет считаться здоровым, тогда как если тест положителен, пациент будет подвергнут дополнительному обследованию. Здесь мы называем положительным тестом (наличие рака) положительный класс, а отрицательный тест соответствует отрицательному классу. Мы не можем быть уверены в отличной работе модели, она неизбежно будет совершать ошибки. Выполняя тот или иной проект, мы должны спросить себя, какими могут быть последствия этих ошибок в реальном мире.

Одна из возможных ошибок заключается в том, что здоровый пациент будет классифицирован как больной (положительный класс), что даст повод для дополнительного тестирования. Дополнительное обследование приведет к некоторым затратам и неудобствам для пациента (и, возможно, к определенному психическому дискомфорту). Пример, неправильно спрогнозированный как положительный, называется *ложно положительным (false positive)*. Другая возможная ошибка состоит в том, что больной пациент будет классифицирован как здоровый (отрицательный класс), не пройдет дополнительные тесты и не получит лечения. Недиagnosticированный вовремя рак может привести к серьезным проблемам со здоровьем и может даже закончиться смертельным исходом. Пример, неправильно спрогнозированный как отрицательный, называется *ложно отрицательным (false negative)*. В статистике ложно положительный пример также известен как *ошибка I рода (type I error)*, а ложно отрицательный пример – как *ошибка II рода (type II error)*. Мы будем придерживаться определений «ложно отрицательный пример» и «ложно положительный пример», поскольку они являются более явными и их легче запомнить. В примере с диагностикой рака очевидно, что мы хотим минимизировать долю ложно отрицательных примеров, тогда как ложно положительные примеры можно считать гораздо менее значительной неприятностью.

Хотя вышеприведенный пример является довольно ярким, каждый ложно положительный и ложно отрицательный прогноз редко приводит к одним и тем же последствиям. В коммерческих проектах обоим видам ошибок можно присвоить определенные стоимости, которые позволяют измерить погрешность конкретного прогноза в денежном выражении, а не с точки зрения правильности. Для процесса принятия бизнес-решений, использующего модель, данный шаг имеет гораздо большее значение.

### Несбалансированные наборы данных

Типы ошибок играют важную роль, когда один из двух классов встречается гораздо чаще, чем другой. Это очень распространенная ситуация на практике. Хорошим примером является прогноз рейтинга кликов, где каждая точка данных представляет собой «показ» – элемент, предъявленный пользователю. Этим элементом может быть объявление, рассказ, пользователь социальной сети. Цель состоит в том, чтобы предсказать, будет ли пользователь при показе данного элемента кликать по нему (что указывает на его интерес). Большинство из того, что видит пользователь в Интернете (в частности, рекламные объявления), не вызывает у него особого интереса. Вам потребуется показать пользователю 100 объявлений или статей, прежде чем он найдет что-то достаточно интересное для себя, чтобы кликнуть. Это позволяет получить набор данных, в котором 99 точек данных соответствуют ситуации «не кликнул» и 1 точка данных – «кликнул». Другими словами, 99% примеров относятся к классу «отсутствие клика». Наборы данных, в которых один класс встречается гораздо чаще, чем остальные, часто называют *несбалансированными наборами данных (imbalanced datasets)* или *наборами данных с несбалансированными классами (datasets with imbalanced classes)*. В реальности несбалансированные данные являются нормой и редко бывает, что интересующий класс встречался в данных с одинаковой или почти такой же частотой, что и остальные классы.

Теперь предположим, что вы строите классификатор, который при решении задачи прогнозирования кликов имеет правильность 99%. О чем это говорит? Правильность 99% звучит впечатляюще, но она не принимает во внимание дисбаланс классов. Вы можете достичь 99%-ной правильности и без построения модели машинного обучения, всегда прогнозируя «отсутствие клика». С другой стороны, даже для несбалансированных данных модель с 99%-ной правильностью могла бы быть вполне пригодной. Однако в данном случае правильность не позволяет нам отличить модель «постоянно прогнозируем отсутствие клика» от потенциально хорошей модели.

Чтобы проиллюстрировать это, мы на основе набора данных `digits` создадим несбалансированный набор данных с пропорциями 9:1, создав два класса «не-девятка» и «девятка»:

```
In[37]:
from sklearn.datasets import load_digits

digits = load_digits()
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)
```

Мы можем воспользоваться `DummyClassifier`, который всегда предсказывает мажоритарный класс (в данном случае класс «не-девятка»), чтобы проиллюстрировать, насколько малоинформативной может быть правильность:

```
In[38]:
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Уникальные спрогнозированные метки: {}".format(np.unique(pred_most_frequent)))
print("Правильность на тестовом наборе: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

```
Out[38]:
Уникальные спрогнозированные метки: [False]
Правильность на тестовом наборе: 0.90
```

Мы получили 90%-ную правильностью без какого-либо обучения. Это может показаться поразительным, но задумайтесь об этом на минуту. Представьте себе, кто-то говорит вам, что его модель имеет 90%-ную правильность. Можно сделать вывод, что он проделал очень хорошую работу. Но это вполне возможно, лишь правильно прогнозируя один класс! Давайте сравним этот результат с результатом, полученным с помощью реальной модели:

```
In[39]:
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
print("Правильность на тестовом наборе: {:.2f}".format(tree.score(X_test, y_test)))
```

```
Out[39]:
Правильность на тестовом наборе: 0.92
```

С точки зрения правильности `DecisionTreeClassifier` оказался чуть лучше, чем `DummyClassifier`, постоянно предсказывающего мажоритарный класс. Это может означать, что либо мы неправильно использовали `DecisionTreeClassifier`, либо правильность на самом деле не является в данном случае адекватной метрикой.

Для сравнения давайте оценим качество еще двух классификаторов, `LogisticRegression` и обычный `DummyClassifier`, который выдает случайные прогнозы:

```
In[40]:
from sklearn.linear_model import LogisticRegression
dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("правильность dummy: {:.2f}".format(dummy.score(X_test, y_test)))
logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("правильность logreg: {:.2f}".format(logreg.score(X_test, y_test)))

Out[40]:
правильность dummy: 0.80
правильность logreg: 0.98
```

Дамми-классификатор, который генерирует случайные прогнозы, имеет намного худшее качество (с точки зрения правильности), в то время как логистическая регрессия дает очень хорошие результаты. Однако даже случайный классификатор дает 80%-ную правильность. Поэтому очень трудно судить, какой из этих результатов является действительно полезным. Проблема здесь заключается в том, что для несбалансированных наборов данных правильность не является адекватной метрикой, позволяющей количественно оценить прогностическую способность модели. В оставшейся части этой главы мы рассмотрим альтернативные метрики, которые дают более четкие ориентиры при выборе модели. В частности, нам нужны такие метрики, которые позволяют сравнить правильность модели машинного обучения с правильностью классификатора, всегда предсказывающего «наиболее часто встречающийся класс», или случайного классификатора (в данном случае такие классификаторы были вычислены с помощью `pred_most_frequent` и `pred_dummy`). Если мы используем какую-то метрику для оценки модели, она должна уметь отсекаать эти бессмысленные прогнозы.

## Матрица ошибок

Одним из наиболее развернутых способов, позволяющих оценить качество бинарной классификации, является использование матрицы ошибок. Давайте исследуем прогнозы модели `LogisticRegression`, построенной в предыдущем разделе, с помощью функции `confusion_matrix`. Прогнозы для тестового набора данных мы уже сохранили в `pred_logreg`:

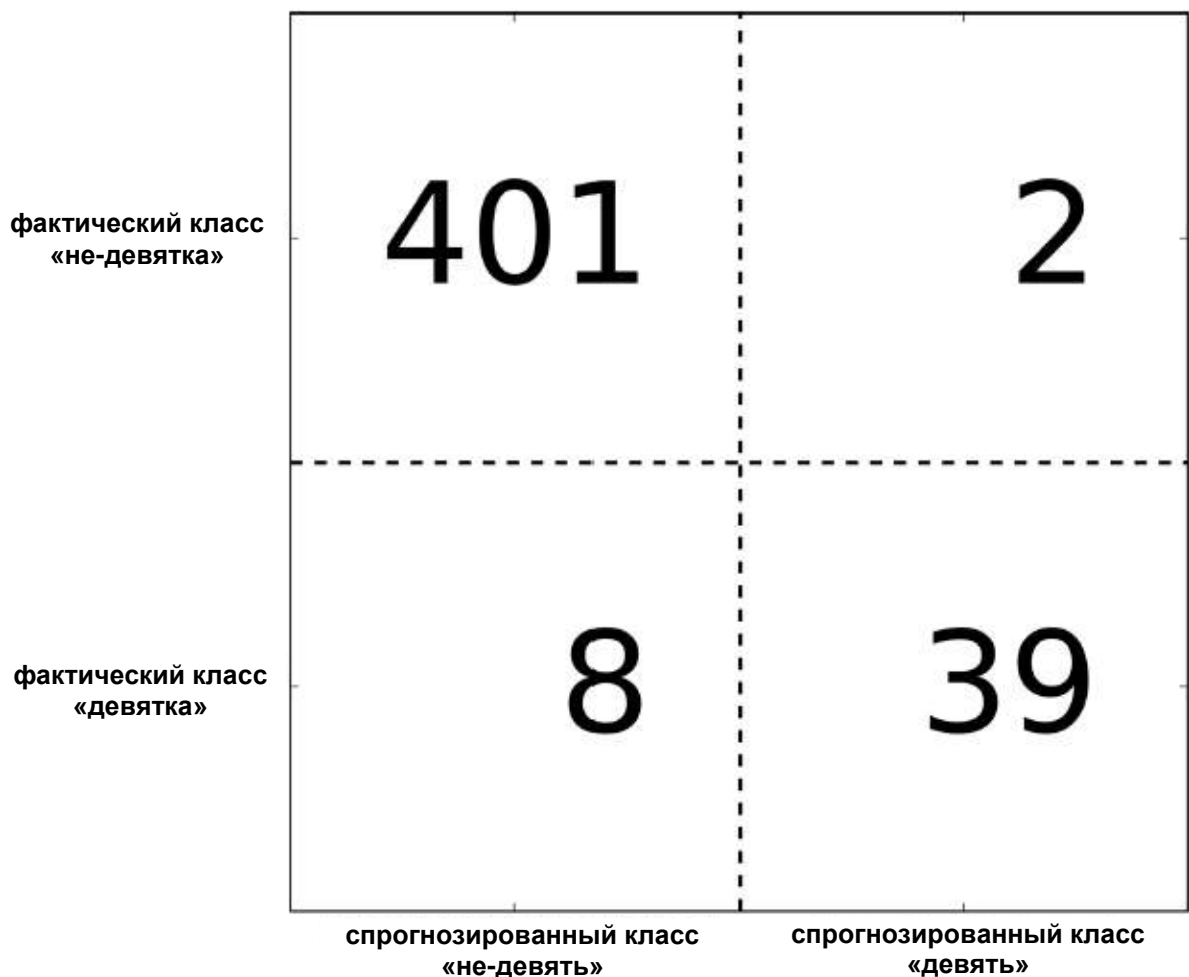
```
In[41]:
from sklearn.metrics import confusion_matrix
confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```

```
Out[41]:
Confusion matrix:
[[401  2]
 [ 8 39]]
```

Вывод `confusion_matrix` представляет собой массив размером 2x2, где строки соответствуют фактическим классам, а столбцы соответствуют спрогнозированным классам. В данном случае речь идет о классах «не-девятка» и «девятка». Число в каждой ячейке показывает количество примеров, когда спрогнозированный класс, представленный столбцом, совпадает или не совпадает с фактическим классом, представленным строкой.

Следующий график (рис. 5.10) иллюстрирует сказанное:

```
In[42]:
mglearn.plots.plot_confusion_matrix_illustration()
```

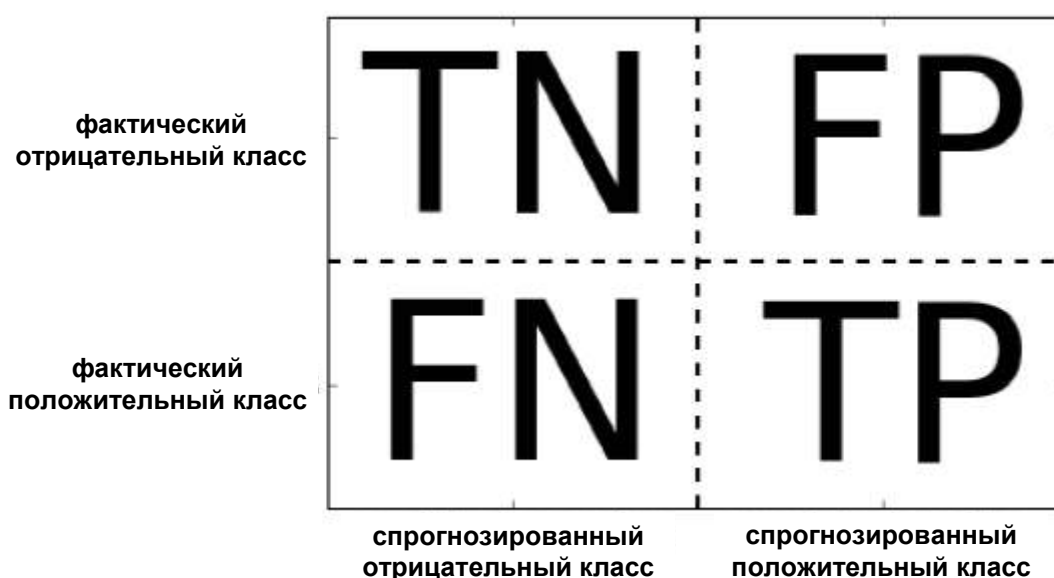


**Рис. 5.10** Матрица ошибок для классификационной задачи «девятка против остальных»

Элементы главной диагонали <sup>33</sup> матрицы ошибок соответствуют правильным прогнозам (результатам классификации), тогда как остальные элементы показывают, сколько примеров, относящихся к одному классу, были ошибочно классифицированы как другой класс.

Объявив «девятку» положительным классом, мы можем рассмотреть элементы матрицы ошибок в терминах *ложно положительных* (*false positive*) и *ложно отрицательных* (*false negative*) примеров, которые мы ввели ранее. Для полноты картины мы назовем правильно классифицированные положительные примеры *истинно положительными* (*true positive*), а правильно классифицированные отрицательные примеры – *истинно отрицательными* (*true negative*). Эти термины, как правило, записывают в сокращенном виде как FP, FN, TP и TN и приводят к следующей интерпретации матрицы ошибок (рис. 5.11):

```
In[43]:
mglearn.plots.plot_binary_confusion_matrix()
```



**Рис. 5.11** Матрица ошибок для бинарной классификации

Теперь давайте воспользуемся матрицей ошибок для сравнения ранее построенных моделей (две дамми-модели, дерево решений, а также логистическая регрессия):

```
In[44]:
print("Наиболее часто встречающийся класс:")
print(confusion_matrix(y_test, pred_most_frequent))
print("\nДамми-модель:")
print(confusion_matrix(y_test, pred_dummy))
print("\nДерево решений:")
print(confusion_matrix(y_test, pred_tree))
```

<sup>33</sup> Главная диагональ двумерного массива или матрицы  $A$  имеет вид  $A[i, i]$ .

```
print("\nЛогистическая регрессия")
print(confusion_matrix(y_test, pred_logreg))
```

Out[44]:

Наиболее часто встречающийся класс:

```
[[403  0]
 [ 47  0]]
```

Дамми-модель:

```
[[361 42]
 [ 43  4]]
```

Дерево решений:

```
[[390 13]
 [ 24 23]]
```

Логистическая регрессия

```
[[401  2]
 [  8 39]]
```

Взглянув на матрицу ошибок, становится совершенно ясно, что с моделью `pred_most_frequent` что-то не так, потому что она всегда предсказывает один и тот же класс. С другой стороны, модель `pred_dummy` характеризуется очень маленьким количеством истинно положительных примеров (4) по сравнению с остальными примерами, при этом количество ложно положительных примеров существенно больше количества истинно положительных примеров! Прогнозы, полученные с помощью дерева решений, несут гораздо больше смысла, чем прогнозы дамми-модели, хотя правильность у этих моделей почти одинаковая. И, наконец, мы видим, что прогнозы логистической регрессии лучше прогнозов `pred_tree` во всех аспектах: она имеет большее количество истинно положительных и истинно отрицательных примеров, в то время количество ложно положительных и ложно отрицательных примеров стало меньше. Из этого сравнения ясно, что лишь дерево решений и логистическая регрессия дают разумные результаты, при этом логистическая регрессия работает лучше дерева во всех отношениях. Однако интерпретация матрицы ошибок немного громоздка и хотя мы получили массу информации, анализируя все аспекты матрицы, процесс работы с матрицей ошибок был трудоемким и сложным. Есть несколько способов обобщить информацию, содержащуюся в матрице ошибок. О них мы поговорим в следующем разделе.

### Связь с правильностью

Мы уже знакомы с одним из способов обобщить результаты матрицы – вычислением правильности, которую можно выразить в виде следующей формулы:

$$\text{Правильность} = \frac{TP + TN}{TP + TN + FP + FN}$$



Другими словами, правильность – это количество верно классифицированных примеров (TP и TN), поделенное на общее количество примеров (суммируем все элементы матрицы ошибок).

### Точность, полнота и F-мера

Есть еще несколько способов подытожить информацию матрицы ошибок, наиболее часто используемыми из них являются точность и полнота. *Точность (precision)* показывает, сколько из предсказанных положительных примеров оказались действительно положительными. Таким образом, точность – это доля истинно положительных примеров от общего количества предсказанных положительных примеров.

$$\text{Точность} = \frac{TP}{TP + FP}$$

Точность используется в качестве показателя качества модели, когда цель состоит в том, чтобы снизить количество ложно положительных примеров. В качестве примера представьте модель, которая должна спрогнозировать, будет ли эффективен новый лекарственный препарат при лечении болезни. Клинические испытания, как известно, дороги, и фармацевтическая компания хочет провести их лишь в том случае, когда полностью убедится, что препарат действительно работает. Поэтому важно минимизировать количество ложно положительных примеров, другими словами, необходимо увеличить точность. Точность также известна как *прогностическая ценность положительного результата (positive predictive value, PPV)*.

С другой стороны, *полнота (recall)* показывает, сколько от общего числа фактических положительных примеров было предсказано как положительный класс. Полнота – это доля истинно положительных примеров от общего количества фактических положительных примеров.

$$\text{Полнота} = \frac{TP}{TP + FN}$$

Полнота используется в качестве показателя качества модели, когда нам необходимо определить все положительные примеры, то есть, когда важно снизить количество ложно отрицательных примеров. Пример диагностики рака, приведенный ранее в этой главе, является хорошей иллюстрацией подобной задачи: важно выявить всех больных пациентов, при этом, возможно, включив в их число здоровых пациентов. Другие названия полноты – *чувствительность (sensitivity)*, *процент результативный ответов* или *хит-рейт (hit rate)* и доля истинно положительных примеров (*true positive rate, TPR*).

Всегда необходимо найти компромисс между оптимизацией полноты и оптимизацией точности. Вы легко можете получить идеальную полноту, спрогнозировав все примеры как положительные – не будет никаких ложно отрицательных и истинно отрицательных примеров. Однако прогнозирование всех примеров как положительных приведет к большому количеству ложно положительных примеров, и, следовательно, точность будет очень низкой. С другой стороны, допустим, у вас есть набор данных из 201 примера и вы строите модель, которая прогнозирует один пример как положительный (и этот пример действительно относится к положительному классу), а все остальные примеры относит к отрицательному классу. Предположим, матрица ошибок выглядит следующим образом.

<b>TN</b> 100 примеров	<b>FP</b> 0 примеров
<b>FN</b> 100 примеров	<b>TP</b> 1 пример

Вычисляем точность и полноту. Точность будет идеальной, а полнота – очень низкой.

$$\text{Прецизионность} = \frac{TP}{TP + FP} = \frac{1}{1 + 0} = 1$$

$$\text{Полнота} = \frac{TP}{TP + FN} = \frac{1}{1 + 100} = 0.0099$$



Точность и полнота – это лишь две метрики из множества показателей классификации, получаемых с помощью TP, FP, TN и FN. Вы можете найти подробное описание метрик в [Википедии](#). Среди специалистов по машинному обучению точность и полнота являются, возможно, наиболее часто используемыми метриками бинарной классификации, однако остальные специалисты могут использовать другие связанные с ними показатели.

Хотя точность и полнота являются очень важными метриками, сами по себе они не дадут вам полной картины. Одним из способов подытожить их является *F-мера* (*F-measure*), которая представляет собой гармоническое среднее точности и полноты:

$$F = 2 \cdot \frac{\text{точность} \cdot \text{полнота}}{\text{точность} + \text{полнота}}$$

Этот вариант вычисления F-меры еще известен как  $f_1$ -мера. Поскольку  $f_1$ -мера учитывает точность и полноту, то для бинарной классификации несбалансированных данных она может быть более лучшей метрикой, чем правильность. Давайте применим ее к прогнозам для нашего набора данных «девятка против остальных», полученным нами ранее. В данном случае мы будем считать класс «девятка» положительным классом (он получает метку `True`, тогда как класс «не-девятка» получает метку `False`), таким образом, положительный класс является миноритарным классом:

```
In[45]:
from sklearn.metrics import f1_score
print("f1-мера наибольшая частота: {:.2f}".format(
    f1_score(y_test, pred_most_frequent)))
print("f1-мера дамми: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("f1-мера дерево: {:.2f}".format(f1_score(y_test, pred_tree)))
print("f1-мера логистическая регрессия: {:.2f}".format(
    f1_score(y_test, pred_logreg)))
```

```
Out[45]:
f1-мера наибольшая частота: 0.00
f1-мера дамми: 0.10
f1-мера дерево: 0.55
f1-мера логистическая регрессия: 0.89
```

Здесь мы можем отметить два момента. Во-первых, мы получаем сообщение об ошибке для прогнозов модели `most_frequent`, поскольку не было получено ни одного прогноза положительного класса (таким образом, знаменатель в формуле расчета  $f$ -меры равен нулю). Кроме того, мы можем увидеть довольно сильное различие между прогнозами дамми-модели и прогнозами дерева, которое не так явно бросается в глаза, когда мы анализируем только правильность. Используя  $f$ -меру для оценки качества, мы снова подытоживаем прогностическую способность с помощью одного числа. Однако, похоже, что  $f$ -мера действительно дает более лучшее представление о качестве модели, чем правильность. Вместе с тем недостаток  $f$ -меры заключается в том, что в отличие от правильности ее труднее интерпретировать и объяснить.

Если мы хотим получить более развернутый отчет о точности, полноте и  $f_1$ -мере, можно воспользоваться удобной функцией `classification_report`, чтобы вычислить все три метрики сразу и распечатать их в привлекательном виде:

```
In[46]:
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent,
                           target_names=["not nine", "nine"]))
```

```
Out[46]:
```

	precision	recall	f1-score	support
not nine	0.90	1.00	0.94	403
nine	0.00	0.00	0.00	47
avg / total	0.80	0.90	0.85	450

Функция `classification_report` печатает отчет, в котором выводятся показатели точности, полноты и  $f$ -меры для отрицательного и положительного классов. Миноритарный класс «девятка» считается положительным классом. Значение  $f$ -меры для него равно 0. Для мажоритарного класса «не-девятка» значение  $f$ -меры равно 0.94. Кроме того, полнота для класса «не-девятка» равна 1, поскольку мы классифицировали все примеры как «не-девятки». Крайний правый столбец – это *поддержка* (*support*), которая равна фактическому количеству примеров данного класса.

В последней строке отчета приводятся средние значения метрик, взвешенные по количеству фактических примеров в каждом классе. Поясним процесс вычисления взвешенного среднего значения для примере  $f$ -метрики. Сначала вычисляем веса отрицательного и положительного классов. Вес отрицательного класса равен  $403/450=0.90$ . Вес положительного класса равен  $47/450=0.10$ . Теперь спрогнозированное значение  $f$ -меры для каждого класса умножаем на вес соответствующего класса, складываем результаты и получаем взвешенное среднее значение  $f$ -меры:  $0.90 \times 0.94 + 0.10 \times 0.00 = 0.85$ . Ниже даны еще два отчета – для дамми-классификатора и логстической регрессии:

```
In[47]:
print(classification_report(y_test, pred_dummy,
                           target_names=["not nine", "nine"]))
```

Out[47]:

	precision	recall	f1-score	support
not nine	0.90	0.92	0.91	403
nine	0.11	0.09	0.10	47
avg / total	0.81	0.83	0.82	450

```
In[48]:
print(classification_report(y_test, pred_logreg,
                           target_names=["not nine", "nine"]))
```

Out[48]:

	precision	recall	f1-score	support
not nine	0.98	1.00	0.99	403
nine	0.95	0.83	0.89	47
avg / total	0.98	0.98	0.98	450

Взглянув на отчеты, можно заметить, что различия между дамми-моделью и моделью логстической регрессии уже не столь очевидны. Решение о том, какой класс объявить положительным, имеет большое влияние на метрики. Несмотря на то, что в дамми-классификаторе  $f$ -мера для класса «девятка» равна 0.13 (в сравнении с 0.89 для логстической регрессии), а для класса «не-девятка» она равна 0.90 (в сравнении с 0.99 для логстической регрессии), похоже, что обе модели дают разумные результаты. Однако проанализировав все показатели вместе, можно

составить довольно точную картину и четко увидеть превосходство модели логистической регрессии.

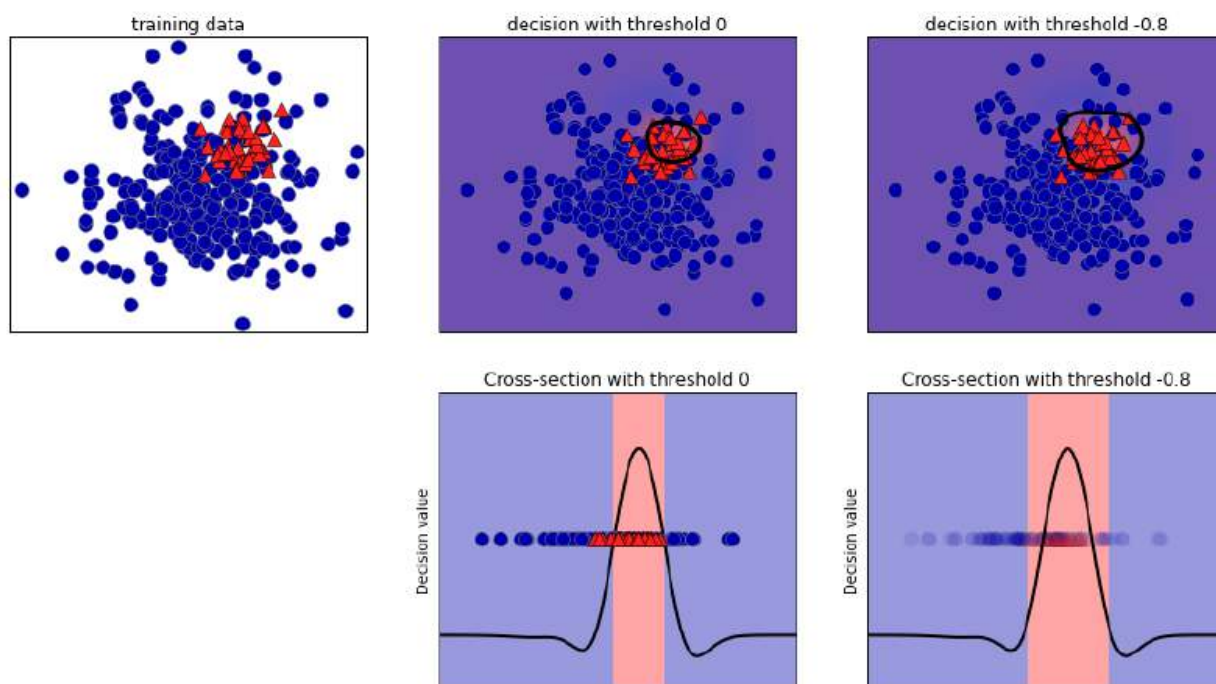
### Принимаем во внимание неопределенность

Матрица ошибок и отчет о результатах классификации позволяют провести очень детальный анализ полученных прогнозов. Однако сами по себе прогнозы лишены большого объема информации, которая собрана моделью. Как мы уже говорили в главе 2, большинство классификаторов для оценки степени достоверности прогнозов позволяют использовать методы `decision_function` или `predict_proba`. Получить прогнозы можно, установив для `decision_function` или `predict_proba` пороговое значение в некоторой фиксированной точке – в случае бинарной классификации мы используем 0 для решающей функции и 0.5 для метода `predict_proba`.

Ниже приведен пример несбалансированной бинарной классификации: 400 точек данных в отрицательном классе и 50 точек данных в положительном классе. Обучающие данные показаны на рис. 5.12 слева. Мы обучаем модель ядерного SVM на этих данных, а также выводим справа графики обучающих данных, показывающие значения решающей функции в виде теплокарты. В самом центре графика можно увидеть черную окружность, который соответствует пороговому значению `decision_function`, равному нулю. Точки внутри этой окружности будут классифицироваться как положительный класс, а точки вне окружности будут отнесены к отрицательному классу:

```
In[49]:
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
```

```
In[50]:
mglearn.plots.plot_decision_threshold()
```



**Рис. 5.12** Тепловая карта решающей функции и влияние изменения порогового значения на результат

Воспользуемся функцией `classification_report`, чтобы оценить точность и полноту для обоих классов:

```
In[51]:
print(classification_report(y_test, svc.predict(X_test)))
```

```
Out[51]:
```

	precision	recall	f1-score	support
0	0.97	0.89	0.93	104
1	0.35	0.67	0.46	9
avg / total	0.92	0.88	0.89	113

Для класса 1 мы получаем довольно небольшое значение полноты и еще более низкое значение точности. Поскольку класс 0 представлен гораздо большим количеством примеров, классификатор точнее прогнозирует класс 0 и гораздо менее точно класс 1.

Давайте предположим, что в нашем примере гораздо важнее получить высокое значение полноты для класса, как в случае со скринингом рака, приведенном ранее. Это означает, что мы готовы допустить большее количество ложных срабатываний (случаев, когда неверно спрогнозирован класс 1), что даст нам большее количество истинно положительных примеров (то есть увеличит значение полноты). Прогнозы, полученные с помощью `svc.predict`, не отвечают этому требованию, но мы можем скорректировать их, чтобы получить более высокое значение полноты для класса 1. Для этого необходимо изменить пороговое значение для принятия решений. По умолчанию точки данных

со значениями решающей функции больше 0 будут классифицироваться как класс 1. Мы хотим увеличить количество точек данных, прогнозируемых как класс 1, поэтому нужно *снизить* пороговое значение:

```
In[52]:
y_pred_lower_threshold = svc.decision_function(X_test) > -.8
```

Давайте взглянем на отчет о результатах классификации, полученный для этого прогноза:

```
In[53]:
print(classification_report(y_test, y_pred_lower_threshold))
```

```
Out[53]:
```

	precision	recall	f1-score	support
0	1.00	0.82	0.90	104
1	0.32	1.00	0.49	9
avg / total	0.95	0.83	0.87	113

Как и следовало ожидать, значение полноты для класса 1 стало высоким, а точность упала. Сейчас для большей области пространства мы прогнозируем класс 1, как это показано в верхней правой части рис. 5.12. Если вам нужно увеличить точность по сравнению с полнотой или наоборот, либо ваши данные в значительной степени не сбалансированы, изменение порогового значения является самым простым способом улучшить результат. Поскольку решающая функция может принимать различные диапазоны значений, трудно сформулировать правило, касающееся выбора порогового значения.



Устанавливая пороговое значение, убедитесь в том, что не используете для этого тестовый набор. Как и в случае с любым другим параметром, пороговое значение, выбранное с помощью тестового набора, вероятно, даст очень оптимистичные результаты. Для выбора порогового значения используйте проверочный набор или перекрестную проверку.

Выбрать пороговое значение для моделей, поддерживающих метод `predict_proba`, проще, поскольку выводом `predict_proba` являются числа, находящиеся в фиксированном диапазоне от 0 до 1 и представляющие собой вероятности. По умолчанию порог 0.5 означает, что если модель более чем на 50% «уверена», что данная точка является положительным классом, точка будет классифицирована как положительный класс. Повышение порогового значения подразумевает, что модели требуется *большая* степень уверенности, чтобы принять решение в пользу положительного класса (или *меньшая* степень уверенности, чтобы принять решение в пользу отрицательного класса). Несмотря на то, что работать с вероятностями проще, чем работать с произвольными пороговыми значениями, не все модели позволяют



получить реалистичные оценки неопределенности (например, дерево решений максимальной глубины всегда на 100% уверено в своих прогнозах, хотя это часто не так). Это связано с понятием *калибровки* (*calibration*): калиброванная модель представляет собой модель, которая позволяет точно измерить неопределенность оценок. Подробное рассмотрение вопросов калибровки выходит за рамки этой книги, но вы можете найти более подробную информацию в [статье Niculescu-Mizil, Caruana «Predicting Good Probabilities with Supervised Learning»](#).

### Кривые точности-полноты и ROC-кривые

Как мы уже сказали, изменение порога, используемого для классификации решений модели – это способ, позволяющий найти компромисс между точностью и полнотой для данного классификатора. Возможно, вы хотите пропустить менее 10% положительных примеров, таким образом, желаемое значение полноты составит 90%. Решение зависит от конкретного примера и оно должно определяться бизнес-целями. Как только поставлена конкретная цель, скажем, задано конкретное значение полноты или точности для класса, можно установить соответствующий порог. Всегда можно задать то или иное пороговое значение для реализации конкретной цели (например, достижения значения полноты 90%). Трудность состоит в разработке такой модели, которая при этом пороге еще и будет иметь приемлемое значение точности, ведь классифицировав все примеры как положительные, вы получите значение полноты, равное 100%, но при этом ваша модель будет бесполезной.

Требование, выдвигаемое к качеству модели (например, значение полноты должно быть 90%), часто называют *рабочей точкой* (*operating point*). Фиксирование рабочей точки часто бывает полезно в контексте бизнеса, чтобы гарантировать определенный уровень качества клиентам или другим группам лиц внутри организации.

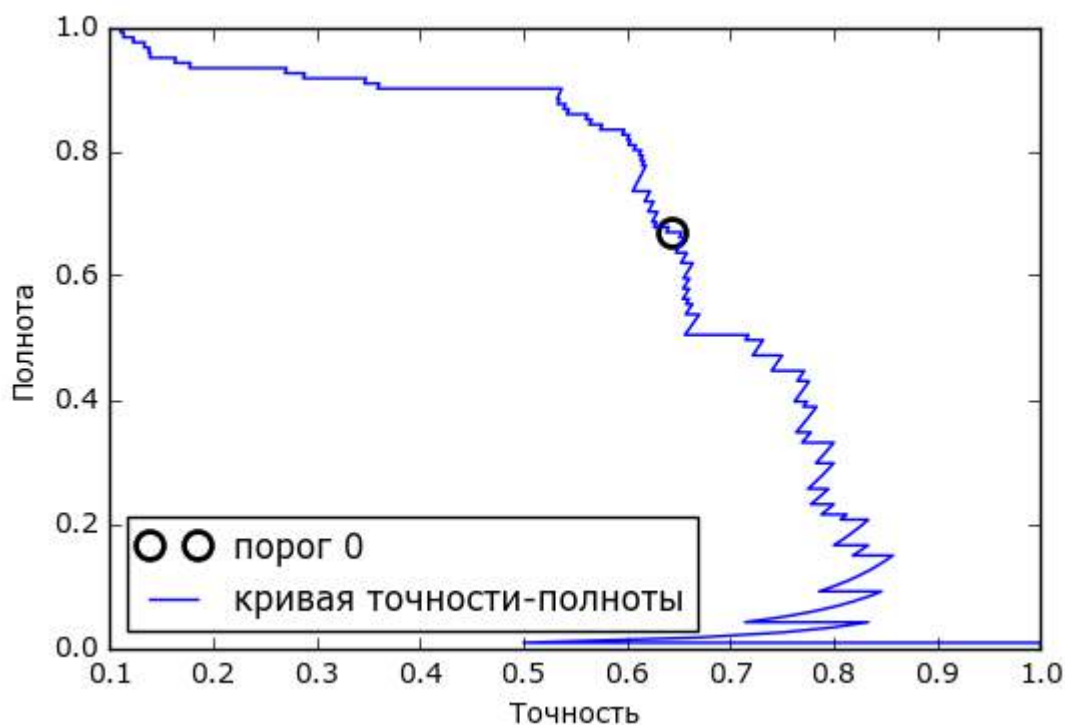
Как правило, при разработке новой модели нет четкого представления о том, что будет рабочей точкой. По этой причине, а также для того, чтобы получить более полное представление о решаемой задаче, полезно сразу взглянуть на все возможные пороговые значения или все возможные соотношения точности и полноты для этих пороговых значений. Данную процедуру можно осуществить с помощью инструмента, называемого *кривой точности-полноты* (*precision-recall curve*). Функцию для вычисления кривой точности-полноты можно найти в модуле `sklearn.metrics`. Ей необходимо передать фактические метки классов и спрогнозированные вероятности, вычисленные с помощью `decision_function` или `predict_proba`:

```
In[54]:
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
```

Функция `precision_recall_curve` возвращает список значений точности и полноты для всех возможных пороговых значений (всех значений решающей функции) в отсортированном виде, поэтому мы можем построить кривую, как показано на рис. 5.13:

```
In[55]:
# используем большой объем данных, чтобы получить более гладкую кривую
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
# находим ближайший к нулю порог
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="порог 0", fillstyle="none", c='k', mew=2)

plt.plot(precision, recall, label="кривая точности-полноты")
plt.xlabel("Точность")
plt.ylabel("Полнота")
plt.legend(loc="best")
```



**Рис. 5.13** Кривая точности-полноты для SVC (gamma=0.05)

Каждая точка на кривой (рис. 5.13) соответствует возможному пороговому значению решающей функции. Например, видно, что мы можем достичь полноты 0.4 при точности около 0.75. Черный кружок отмечает точку, соответствующую порогу 0, пороговому значению по

умолчанию для решающей функции. Данная точка является компромиссом, который выбирается при вызове метода `predict`.

Чем ближе кривая подходит к верхнему правому углу, тем лучше классификатор. Точка в верхнем правом углу означает высокое значение точности *и* высокое значение полноты для соответствующего порога. Кривая начинается в верхнем левом углу, что соответствует очень низкому порогу, все примеры классифицируются как положительный класс. Повышение порога перемещает кривую в сторону более высоких значений точности и в то же время более низких значений полноты. При дальнейшем повышении порога мы получаем ситуацию, в которой большинство точек, классифицированных как положительные, являются истинно положительными, что приводит к очень высокой точности, но более низкому значению полноты. Чем больше модель сохраняет высокое значение полноты при одновременном увеличении точности, тем лучше.

Взглянув на эту кривую чуть более пристально, можно увидеть, что с помощью построенной модели можно добиться точности в районе 0.5 при очень высоком значении полноты. Если мы хотим получить гораздо более высокое значение точности, мы должны в значительной степени пожертвовать полнотой. Другими словами, слева наша кривая выглядит относительно плоской, это означает, что при увеличении точности полнота падает незначительно. Однако, чтобы получить значение точности более 0.5, нам придется пожертвовать значительным снижением полноты.

Различные классификаторы могут давать хорошее качество на различных участках кривой, то есть в разных рабочих точках. Давайте сравним модель SVM с моделью случайного леса, построенной на том же наборе данных. `RandomForestClassifier` вместо `decision_function` использует метод `predict_proba`. Функция `precision_recall_curve` ожидает, что в качестве второго аргумента ей будет передана вероятность положительного класса (класса 1), то есть `rf.predict_proba(X_test)[: , 1]`. В бинарной классификации пороговое значение по умолчанию для `predict_proba` равно 0.5, поэтому мы отметили эту точку на кривой (см. рис. 5.14):

```
In[56]:
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)

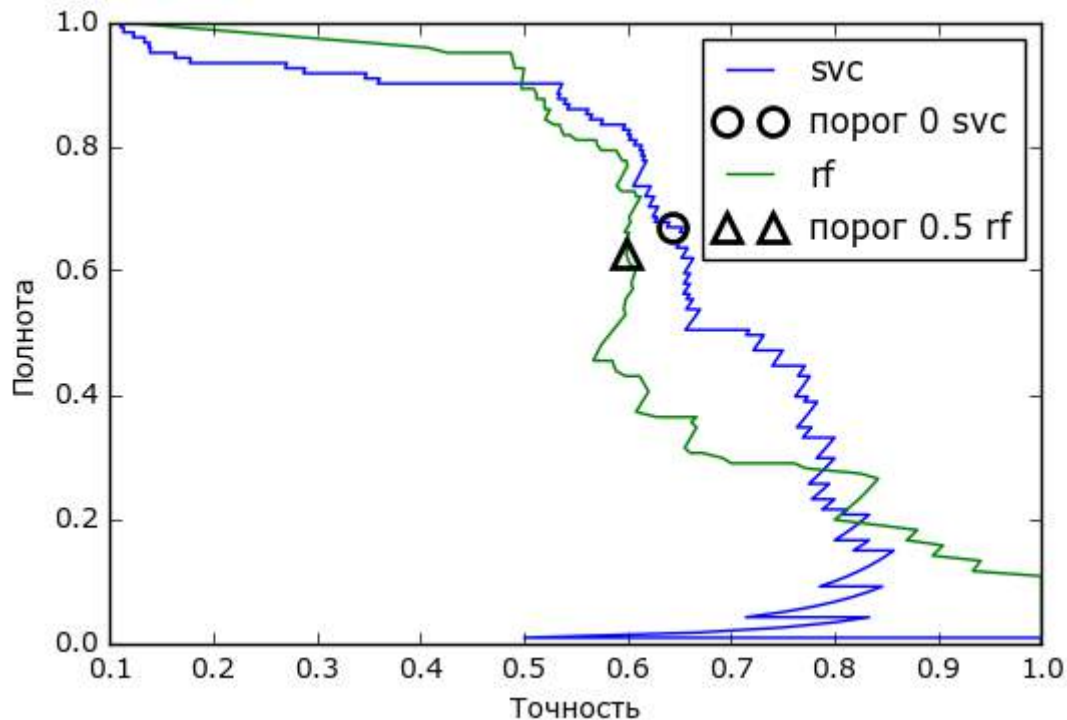
# в RandomForestClassifier есть predict_proba, но нет decision_function
precision_rf, recall_rf, thresholds_rf = precision_recall_curve(
    y_test, rf.predict_proba(X_test)[: , 1])

plt.plot(precision, recall, label="svc")

plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
        label="порог 0 svc", fillstyle="none", c='k', mew=2)
```

```
plt.plot(precision_rf, recall_rf, label="rf")

close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^', c='k',
         markersize=10, label="порог 0.5 rf", fillstyle="none", mew=2)
plt.xlabel("Точность")
plt.ylabel("Полнота")
plt.legend(loc="best")
```



**Рис. 5.14** Сравнение кривых точности-полноты для SVM и случайного леса

Из сравнительного графика видно, что случайный лес дает лучшее качество, чем в SVM, для крайних пороговых значений, позволяя получить очень высокое значение точности или очень высокое значение полноты. Что касается центральной части кривой (соответствует примерной точности=0.7), то SVM работает лучше. Если бы мы для сравнения обобщающей способности в целом анализировали лишь  $f_1$ -меру, мы упустили бы из виду эти тонкости.  $f_1$ -мера учитывает только одну точку на кривой точности-полноты, точку, определяемую порогом по умолчанию.

```
In[57]:
print("f1-мера random forest: {:.3f}".format(
f1_score(y_test, rf.predict(X_test))))
print("f1-мера svc: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))
```

```
Out[57]:
f1-мера random forest: 0.610
f1-мера svc: 0.656
```

Сравнение двух кривых точности-полноты дает много детальной информации, но представляет собой довольно трудоемкий процесс. Чтобы выполнить автоматическое сравнение моделей мы могли бы обобщить информацию, содержащуюся в кривой, не ограничиваясь конкретным пороговым значением или рабочей точкой. Один из способов подытожить информацию кривой заключается в вычислении интеграла или площади под кривой точности-полноты, он также известен как метод *средней точности* (*average precision*).<sup>34</sup> Для вычисления средней точности вы можете воспользоваться функцией `average_precision_score`. Поскольку нам нужно вычислить ROC-кривую и рассмотреть несколько пороговых значений, функции `average_precision_score` вместо результата `predict` нужно передать результат `decision_function` или `predict_proba`:

```
In[58]:
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[: , 1])
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))
print("Средняя точность random forest: {:.3f}".format(ap_rf))
print("Средняя точность svc: {:.3f}".format(ap_svc))

Out[58]:
Средняя точность random forest: 0.666
Средняя точность of svc: 0.663
```

При усреднении по всем возможным пороговым значениям мы видим, что случайный лес и SVC дают примерно одинаковое качество модели, при этом случайный даже чуть-чуть вырывается вперед. Это в значительном мере отличается от результата, полученного нами ранее с помощью `f1_score`. Поскольку средняя точность равна площади под кривой, которая принимает значения от 0 до 1, средняя точность всегда возвращает значение от 0 (худшее значение) до 1 (лучшее значение). Средняя точность случайного классификатора равна доле положительных примеров в наборе данных.

## Рабочая характеристика приемника (ROC) и AUC

Еще один инструмент, который обычно используется для анализа поведения классификаторов при различных пороговых значениях – это *кривая рабочей характеристики приемника* (*receiver operating characteristics curve*) или кратко *ROC-кривая* (*ROC curve*). Как и кривая точности-полноты, ROC-кривая позволяет рассмотреть все пороговые значения для данного классификатора, но вместо точности и полноты она показывает *долю ложно положительных примеров* (*false positive rate, FPR*) в сравнении с *долей истинно положительных примеров* (*true*

---

<sup>34</sup> С технической точки зрения существуют некоторые незначительные различия между площадью под кривой точности-полноты и средней точностью. Однако приведенное объяснение передает общую идею.

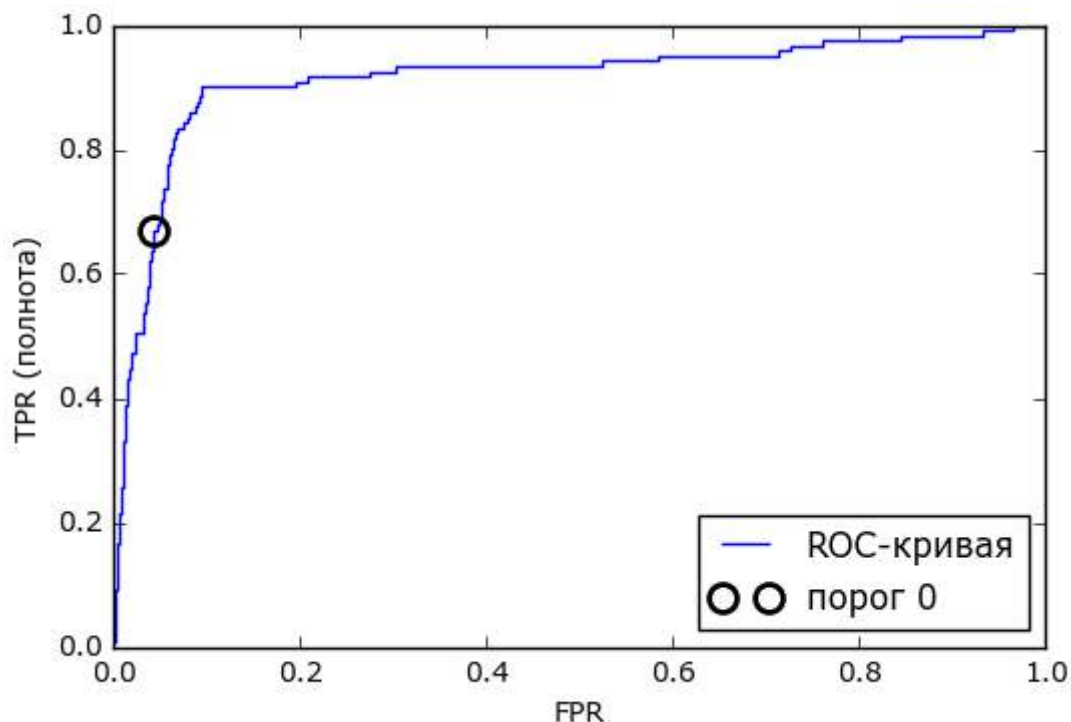
*positive rate*). Вспомним, что доля истинно положительных примеров – это просто еще одно название полноты, тогда как доля ложно положительных примеров – это доля ложно положительных примеров от общего количества отрицательных примеров:

$$FPR = \frac{FP}{FP + TN}$$

ROC-кривую можно вычислить с помощью функции `roc_curve` (см. рис. 5.15):

```
In[59]:
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))

plt.plot(fpr, tpr, label="ROC-кривая")
plt.xlabel("FPR")
plt.ylabel("TPR (полнота)")
# находим пороговое значение, ближайшее к нулю
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
        label="порог 0", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```



**Рис. 5.15** ROC-кривая для SVM

Идеальная ROC-кривая проходит через левый верхний угол, соответствуя классификатору, который дает *высокое значение полноты* при *низкой доле ложно положительных примеров*. Проанализировав значения полноты и FPR для порога по умолчанию 0, мы видим, что можем достичь гораздо более высокого значения полноты (около 0.9) лишь при незначительном увеличении FPR. Точка, ближе всего



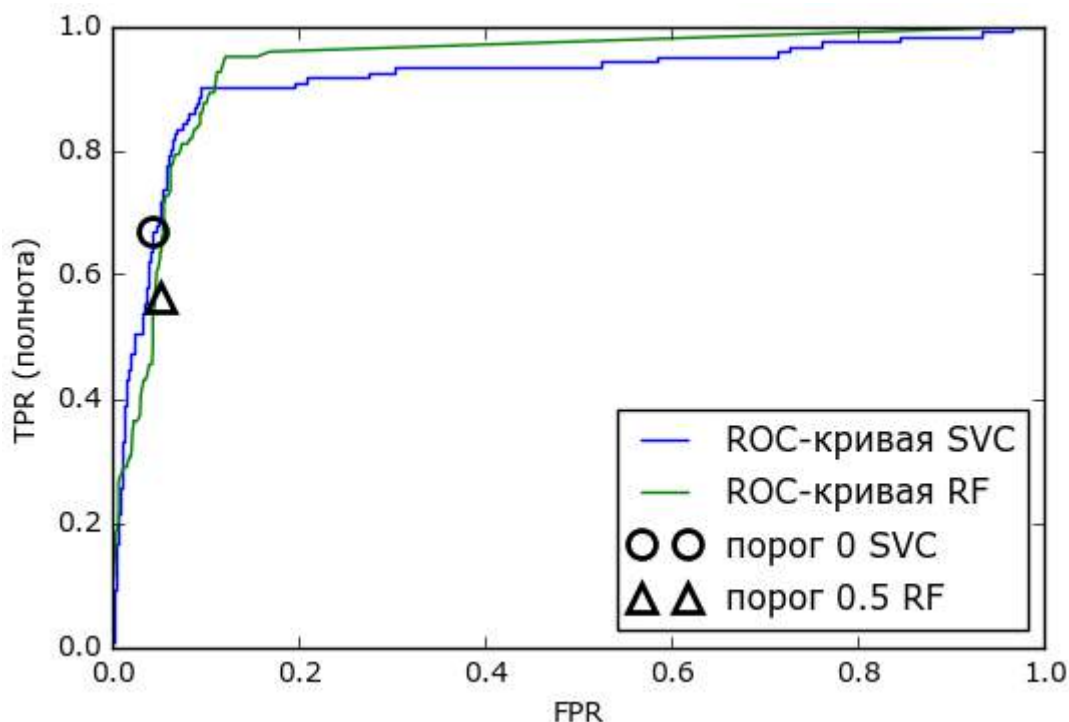
расположенная к верхнему левому углу, возможно, будет лучшей рабочей точкой, чем та, что выбрана по умолчанию. Опять же, имейте в виду, что для выбора порогового значения следует использовать отдельный проверочный набор, а не тестовые данные.

На рис. 5.16 вы можете сравнить случайный лес и SVM с помощью ROC-кривых:

```
In[60]:
from sklearn.metrics import roc_curve
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[: , 1])

plt.plot(fpr, tpr, label="ROC-кривая SVC")
plt.plot(fpr_rf, tpr_rf, label="ROC-кривая RF")

plt.xlabel("FPR")
plt.ylabel("TPR (полнота)")
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
        label="порог 0 SVC", fillstyle="none", c='k', mew=2)
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(fpr_rf[close_default_rf], tpr_rf[close_default_rf], '^', markersize=10,
        label="порог 0.5 RF", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```



**Рис. 5.16** Сравнение ROC-кривых для SVM и случайного леса

Как и в случае с кривой точности-полноты, мы хотим подытожить информацию ROC-кривой с помощью одного числа, площади под кривой (обычно ее просто называют AUC, при этом имейте в виду, что речь идет о ROC-кривой). Мы можем вычислить площадь под ROC-кривой с помощью функции `roc_auc_score`:



```

In[61]:
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[: , 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC для случайного леса: {:.3f}".format(rf_auc))
print("AUC для SVC: {:.3f}".format(svc_auc))

Out[61]:
AUC для случайного леса: 0.937
AUC для SVC: 0.916

```

Сравнив случайный лес и SVM с помощью AUC, мы можем сделать вывод, что случайный лес дает чуть более лучшее качество модели, чем SVM. Напомним, поскольку средняя точность – это площадь под кривой, которая принимает значения от 0 до 1, средняя точность всегда возвращает значение от 0 (худшее значение) до 1 (лучшее значение). Случайный классификатор соответствует значению AUC 0.5, независимо от того, как сбалансированы классы в наборе данных. Поэтому метрика AUC является более оптимальной, чем правильность при решении задач несбалансированной классификации. AUC можно интерпретировать как меру качества *ранжирования* положительных примеров. Значение площади под кривой эквивалентно вероятности того, что согласно построенной модели случайно выбранный пример положительного класса будет иметь более высокий балл, чем случайно выбранный пример отрицательного класса. Таким образом, идеальное значение AUC, равное 1, означает, что все положительные примеры в отличие от отрицательных имеют более высокий балл. В задачах несбалансированной классификации применение AUC для отбора модели зачастую является более целесообразным, чем использование правильности.

Давайте вернемся к задаче, которую мы решали ранее, классифицируя в наборе **digits** девятки и остальные цифры. Мы классифицируем наблюдения, используя SVM с тремя различными настройками ширины ядра и **gamma** (см. рис. 5.17):

```

In[62]:
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)

plt.figure()

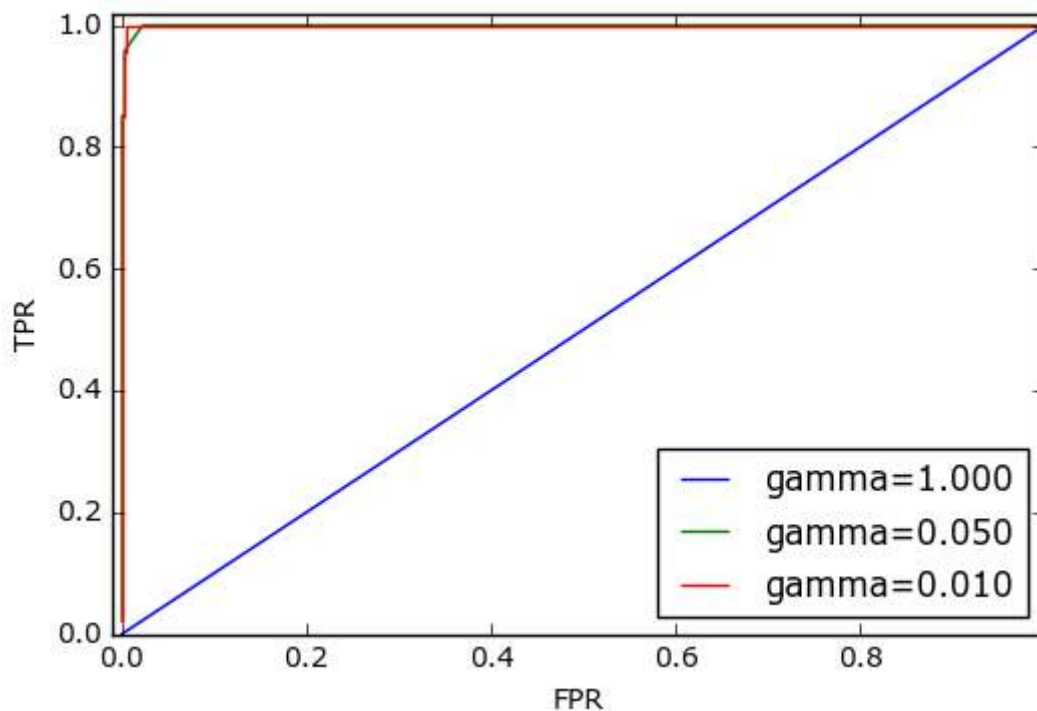
for gamma in [1, 0.05, 0.01]:
    svc = SVC(gamma=gamma).fit(X_train, y_train)
    accuracy = svc.score(X_test, y_test)
    auc = roc_auc_score(y_test, svc.decision_function(X_test))
    fpr, tpr, _ = roc_curve(y_test, svc.decision_function(X_test))
    print("gamma = {:.2f} правильность = {:.2f} AUC = {:.2f}".format(
        gamma, accuracy, auc))
    plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.xlim(-0.01, 1)

```

```
plt.ylim(0, 1.02)
plt.legend(loc="best")
```

Out[62]:

```
gamma = 1.00  правильность = 0.90 AUC = 0.50
gamma = 0.05  правильность = 0.90 AUC = 0.90
gamma = 0.01  правильность = 0.90 AUC = 1.00
```



**Рис. 5.17** Сравнение ROC-кривых для SVM с различными настройками `gamma`

Правильность при использовании различных значений `gamma` остается одинаковой и составляет 90%. Одинаковое значение правильности может быть случайностью, а может быть нет. Однако взглянув на AUC и соответствующую кривую, мы видим четкое различие между этими тремя моделями. При `gamma=1.0` значение AUC фактически соответствует случайному классификатору (случайному результату `decision_function`). При `gamma=0.05` качество модели резко повышается. И, наконец, при `gamma=0.01`, мы получим идеальное значение AUC, равное 1.0. Это означает, что в соответствии с решающей функцией все положительные примеры получают более высокий балл, чем все отрицательные примеры. Другими словами, с помощью правильного порогового значения эта модель может идеально классифицировать данные!<sup>35</sup> Зная это, мы можем скорректировать пороговое значение для

<sup>35</sup> Взглянув на кривую с `gamma=0.01` более внимательно, вы можете увидеть небольшой излом ближе к верхнему левому углу. Это означает, что по крайней мере одна точка данных была ранжирована

этой модели и получить правильные прогнозы. Если бы мы использовали только одну точность, у нас не было бы этой информации.

По этой причине мы настоятельно рекомендуем использовать AUC для оценки качества моделей на несбалансированных данных. Однако имейте в виду, что в AUC не используется порог по умолчанию, таким образом, чтобы на основе модели с высоким значением AUC получить полезный классификатор, возможно, потребуется корректировка порогового значения.

## Метрики для мультиклассовой классификации

Теперь, когда мы подробно рассмотрели вопросы, связанные с оценкой качества бинарной классификации, давайте перейдем к метрикам для оценки качества мультиклассовой классификации. В основном, все метрики для мультиклассовой классификации являются производными от метрик классификации, но при этом усредняются по всем классам. В мультиклассовой классификации правильность вновь определяется как доля правильно классифицированных примеров. И опять же, когда классы не сбалансированы, правильность перестает быть адекватной метрикой оценки качества. Представьте себе задачу трехклассовой классификации, когда 85% точек данных принадлежат к классу А, 10% – к классу В и 5% – к классу С. Что означает среднее значение правильности 85% применительно к этому набору данных? В целом результаты мультиклассовой классификации труднее интерпретировать, чем результаты бинарной классификации. Помимо правильности часто используемыми инструментами являются матрица ошибок и отчет о результатах классификации, которые мы рассматривали, разбирая случай бинарной классификации в предыдущем разделе. Давайте применим эти два метода оценки для классификации 10 различных рукописных цифр в наборе данных **digits**:

```
In[63]:
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("Accuracy: {:.3f}".format(accuracy_score(y_test, pred)))
print("Confusion matrix:\n{}".format(confusion_matrix(y_test, pred)))
```

---

неправильно. Значение AUC, равное 1.0, является результатом округления до второго знака после десятичной точки.

```

Out[63]:
Accuracy: 0.953
Confusion matrix:
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]

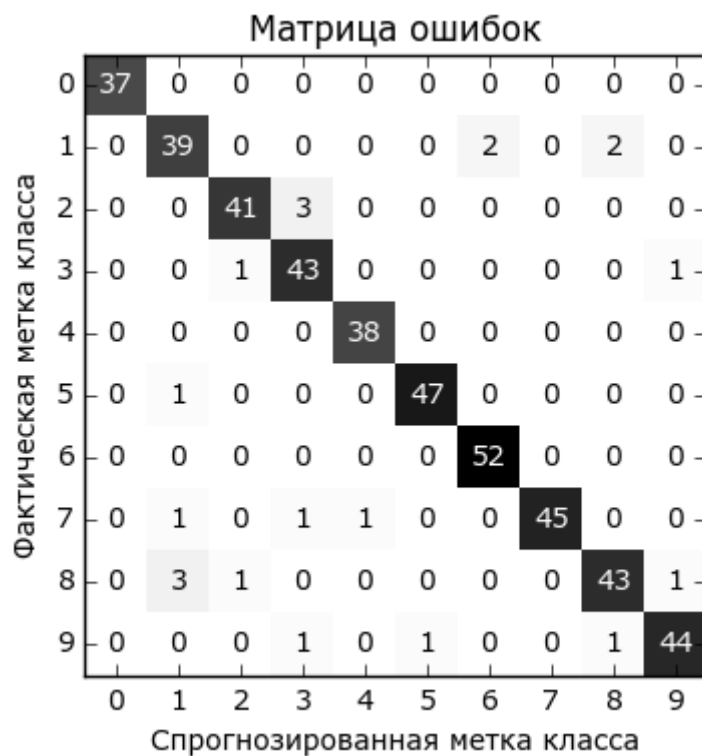
```

Модель имеет точность 95.3%, что уже говорит нам об очень хорошем качестве модели. Матрица ошибок дает нам несколько более подробную информацию. Как и в случае бинарной классификации, каждая строка соответствует фактической метке класса, а каждый столбец соответствует спрогнозированной метке класса. Вы можете построить более наглядный график, приведенный на рис. 5.18:

```

In[64]:
scores_image = mglearn.tools.heatmap(
    confusion_matrix(y_test, pred), xlabel='Спрогнозированная метка класса',
    ylabel='Фактическая метка класса', xticklabels=digits.target_names,
    yticklabels=digits.target_names, cmap=plt.cm.gray_r, fmt="%d")
plt.title("Матрица ошибок")
plt.gca().invert_yaxis()

```



**Рис. 5.18** Матрица ошибок для десятиклассовой задачи распознавания рукописных цифр

Фактическое количество примеров, относящихся к первому классу (цифре 0), равно 37 и все эти примеры были классифицированы как класс 0 (то есть ложно отрицательные примеры для класса 0 отсутствуют). Об этом говорит тот факт, что все остальные элементы первой строки матрицы ошибок имеют нулевые значения. Кроме того, ни одна из остальных цифр не была ошибочно классифицирована как 0, поскольку все остальные элементы первого столбца имеют нулевые значения (то есть ложно положительные примеры для класса 0 отсутствуют). Однако некоторые цифры были спутаны с другими, например, цифра 2 (третья строка), три примера, являющиеся цифрой 2, были классифицированы как цифра 3 (четвертый столбец). Кроме того, у нас есть одна цифра 3, классифицированная как 2 (третий столбец, четвертая строка), и одна цифра 8, классифицированная как 2 (третий столбец, девятая строка).

С помощью функции `classification_report` мы можем вычислить точность, полноту и  $F$ -меру для каждого класса:

```
In[65]:
print(classification_report(y_test, pred))
```

```
Out[65]:
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	37
1	0.89	0.91	0.90	43
2	0.95	0.93	0.94	44
3	0.90	0.96	0.92	45
4	0.97	1.00	0.99	38
5	0.98	0.98	0.98	48
6	0.96	1.00	0.98	52
7	1.00	0.94	0.97	48
8	0.93	0.90	0.91	48
9	0.96	0.94	0.95	47
avg / total	0.95	0.95	0.95	450

Неудивительно, что для класса 0 получены идеальные значения точности и полноты, равные 1, поскольку все примеры классифицированы правильно. Для класса 7 получена идеальная точность, поскольку отсутствуют ложно положительные примеры (ни один из остальных классов не был ошибочно классифицирован как класс 7), тогда как для класса 6 получена идеальная полнота, поскольку отсутствуют ложно отрицательные примеры. Кроме того, видно, что модель испытывает ряд трудностей при классификации цифр 8 и 3.

Наиболее часто используемой метрикой для оценки качества мультиклассовой классификации для несбалансированных наборов данных является мультиклассовый вариант  $F$ -меры. Идея, лежащая в основе мультиклассовой  $F$ -меры, заключается в вычислении одной бинарной  $F$ -меры для каждого класса, интересующий класс становится положительным, а все остальные – отрицательными классами. Затем эти

$f$ -меры для каждого класса усредняются с использованием одной из следующих стратегий:

- "macro" усреднение вычисляет  $f$ -меры для каждого класса и находит их невзвешенное среднее. Всем классам, независимо от их размера, присваивается одинаковый вес.
- "weighted" усреднение вычисляет  $f$ -меры для каждого класса и находит их среднее, взвешенное по поддержке (количеству фактических примеров для каждого класса). Эта стратегия используется в классификационном отчете по умолчанию.
- "micro" усреднение вычисляет общее количество ложно положительных примеров, ложно отрицательных примеров и истинно положительных примеров по всем классам, а затем вычисляет точность, полноту и  $f$ -меру с помощью этих показателей.

Если вам необходимо присвоить одинаковый вес каждому *примеру*, рекомендуется использовать микро-усреднение  $f_1$ -меры, если вам необходимо присвоить одинаковый вес каждому *классу*, рекомендуется использовать макро-усреднение  $f_1$ -меры:

```
In[66]:
print("Микро-усредненная f1-мера: {:.3f}".format
      (f1_score(y_test, pred, average="micro")))
print("Макро-усредненная f1-мера: {:.3f}".format
      (f1_score(y_test, pred, average="macro")))
```

```
Out[66]:
Микро-усредненная f1-мера: 0.953
Макро-усредненная f1-мера: 0.954
```

## Метрики регрессии

Оценить качество регрессии можно таким же способом, который мы использовали для классификации, например, сравнив количество завышенных и заниженных расчетных значений зависимой переменной. Однако в большинстве рассмотренных примеров будет достаточно применения  $R^2$ , который в методе `score` используется по умолчанию для всех моделей регрессии. Иногда бизнес-решения принимаются на основе среднеквадратической ошибки или средней абсолютной ошибки, что является стимулом для использования этих метрик при настройке моделей. Однако в целом мы пришли к выводу, что с точки зрения оценки качества регрессионных моделей  $R^2$  является более понятной метрикой.

## Использование метрик оценки для отбора модели

Мы подробно рассмотрели множество методов оценки и обсудили их применение с учетом фактических и спрогнозированных результатов.

Однако часто нам нужно воспользоваться метриками типа AUC для отбора модели, выполняемого на основе `GridSearchCV` или `cross_val_score`. К счастью, `scikit-learn` предлагает очень простой способ решения этой задачи с помощью аргумента `scoring`, который можно использовать как в `GridSearchCV`, так и в `cross_val_score`. Вы можете просто задать строку с описанием необходимой метрики оценки. Допустим, мы хотим оценить качество классификатора SVM при решении задачи «девять против остальных» для набора данных `digits`, используя значение AUC. Чтобы поменять метрику оценки с правильности, установленной по умолчанию, на AUC, достаточно указать `"roc_auc"` в качестве параметра `scoring`:

```
In[67]:
# метрика качества классификационной модели по умолчанию – правильность
print("Метрика качества по умолчанию: {}".format(
    cross_val_score(SVC(), digits.data, digits.target == 9)))
# значение параметра scoring="accuracy" не меняет результатов
explicit_accuracy = cross_val_score(SVC(), digits.data, digits.target == 9,
                                    scoring="accuracy")
print("Метрика качества явно заданная правильность: {}".format(explicit_accuracy))
roc_auc = cross_val_score(SVC(), digits.data, digits.target == 9,
                          scoring="roc_auc")
print("Метрика качества AUC: {}".format(roc_auc))
```

```
Out[67]:
Метрика качества по умолчанию: [ 0.9 0.9 0.9]
Метрика качества явно заданная правильность: [ 0.9 0.9 0.9]
Метрика качества AUC: [ 0.994 0.99 0.996]
```

Точно так же мы можем изменить метрику, используемую для отбора наилучших параметров в `Grid-SearchCV`:

```
In[68]:
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target == 9, random_state=0)

# задаем не самую удачную сетку параметров для иллюстрации:
param_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}
# используем метрику по умолчанию, то есть правильность:
grid = GridSearchCV(SVC(), param_grid=param_grid)
grid.fit(X_train, y_train)
print("Решетчатый поиск с использованием правильности")
print("Наилучшие параметры:", grid.best_params_)
print("Наилучшее значение перекр проверки (правильность): {:.3f}".format(grid.best_score_))
print("AUC на тестовом наборе: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Правильность на тестовом наборе: {:.3f}".format(grid.score(X_test, y_test)))
```

```
Out[68]:
Решетчатый поиск с использованием правильности
Наилучшие параметры: {'gamma': 0.0001}
Наилучшее значение перекр проверки (правильность): 0.970
AUC на тестовом наборе: 0.992
Правильность на тестовом наборе: 0.973
```

```
In[69]:
# используем метрику качества AUC:
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc")
grid.fit(X_train, y_train)
print("\nРешетчатый поиск с использованием AUC")
print("Наилучшие параметры:", grid.best_params_)
```



```
print("Наилучшее значение перекр проверки (AUC): {:.3f}".format(grid.best_score_))
print("AUC на тестовом наборе: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Правильность на тестовом наборе: {:.3f}".format(grid.score(X_test, y_test)))
```

Out[69]:

```
Решетчатый поиск с использованием AUC
Наилучшие параметры: {'gamma': 0.01}
Наилучшее значение перекр проверки (AUC): 0.997
AUC на тестовом наборе: 1.000
Правильность на тестовом наборе: 1.000
```

Когда использовалась правильность, был выбран параметр `gamma=0.0001`, тогда как при использовании AUC был выбран `gamma=0.01`. В обоих случаях правильность перекрестной проверки соответствует правильности на тестовом наборе. Однако использование AUC позволила найти настройку параметра, оптимальную с точки зрения AUC и даже с точки зрения правильности.<sup>36</sup>

Наиболее важными значениями параметра `scoring` для классификации являются `accuracy` (по умолчанию), `roc_auc` для площади под ROC-кривой, `average_precision` (площадь под кривой точности-полноты), `f1`, `f1_macro`, `f1_micro` и `f1_weighted` для бинарной  $f_1$ -меры и различных стратегий усреднения. Для регрессии, наиболее часто используемыми значениями являются `r2` для  $R^2$ , `mean_squared_error` для среднеквадратической ошибки и `mean_absolute_error` для средней абсолютной ошибки. Полный список поддерживаемых аргументов вы можете найти, ознакомившись с [документацией](#) или взглянув на словарь `SCORERS` в модуле `metrics.scorer`:

In[70]:

```
from sklearn.metrics.scorer import SCORERS
print("Доступные объекты scorer:\n{}".format(sorted(SCORERS.keys())))
```

Out[70]:

```
Доступные объекты scorer:
['accuracy', 'adjusted_rand_score', 'average_precision', 'f1', 'f1_macro',
'f1_micro', 'f1_samples', 'f1_weighted', 'log_loss', 'mean_absolute_error',
'mean_squared_error', 'median_absolute_error', 'precision', 'precision_macro',
'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'recall',
'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc']
```

## Выводы и перспективы

В этой главе мы обсудили перекрестную проверку, решетчатый поиск, а также метрики, играющие ключевую роль в оценке и улучшении алгоритмов машинного обучения. Метрики, описанные в этой главе, вместе с алгоритмами, рассмотренными в главах 2 и 3, являются

---

<sup>36</sup> Вероятно, решение с более высоким значением правильности, полученное с помощью AUC – это следствие того, что правильность не является адекватной метрикой качества модели при несбалансированных данных.

основными инструментами для каждого специалиста по машинному обучению.

В этой главе есть два довольно важных момента, которые нужно повторить, потому что начинающие специалисты, как правило, игнорируют их. Первый момент связан с перекрестной проверки. Перекрестная проверка или использование тестового набора позволяют оценить модель машинного обучения с точки зрения того, как она будет работать в будущем. Однако, если мы с помощью тестового набора или перекрестной проверки осуществляем отбор модели или отбор параметров модели, мы «растрачиваем» тестовые данные, а использование тех же самых данных для оценки работы модели в будущем приведет к чрезмерно оптимистичным прогнозам. Поэтому нам необходимо разбить данные на обучающий набор для построения модели, проверочный набор для отбора модели параметров и тестовый набор для оценки качества моделей. Вместо одного разбиения мы можем использовать разбиения перекрестной проверки. Наиболее часто используемым вариантом (как описывалось ранее) является разбиение обучение/тест для оценки, а также использование перекрестной проверки на обучающем наборе для отбора модели и параметров.

Второй момент связан с важностью метрики качества или функции оценки, которые используются для отбора модели и оценки модели. Теории, связанные с принятием бизнес-решений на основе прогнозов моделей машинного обучения, в некоторой степени выходят за рамки данной книги.<sup>37</sup> Однако в проектах машинного обучения построение модели с высоким значением правильности редко бывает конечной целью. Убедитесь в том, что метрика, используемая для оценки и отбора модели, является точным приближением решаемой задачи. В реальности классификационные задачи редко характеризуются сбалансированностью классов и зачастую ложно положительные и ложно отрицательные примеры ведут к совершенно различным последствиям. Убедитесь в том, что вы правильно интерпретируете эти последствия и выберите соответствующую метрику.

Методы оценки и отбора модели, которые мы описывали до сих пор, являются важнейшими инструментами в арсенале специалиста по анализу данных. Решетчатый поиск и перекрестную проверку, описанные нами в этой главе, можно применить только к одной модели машинного обучения. Однако ранее мы уже видели, что многие модели требуют предварительной обработки данных и в некоторых ситуациях, например, при распознавании лиц, описанном в главе 3, получение нового представления данных может быть полезным. В следующей главе мы

---

<sup>37</sup> Мы настоятельно рекомендуем вам книгу Provost, Fawcett «[Data Science for Business](#)» для получения дополнительной информации по этой теме.

познакомимся с классом `Pipeline`, который позволяет использовать решетчатый поиск и перекрестную проверку для сложных цепочек алгоритмов.

## ГЛАВА 6. ОБЪЕДИНЕНИЕ АЛГОРИТМОВ В ЦЕПОЧКИ И КОНВЕЙЕРЫ

Как мы уже говорили в главе 4, для многих алгоритмов машинного обучения очень важное значение имеет определенное преобразование данных. Оно начинается с масштабирования данных и объединения признаков вручную, а также включает в себя процесс преобразования признаков с помощью неконтролируемого обучения, как мы видели в главе 3. Поэтому большая часть проектов машинного обучения требует не разового использования какого-то одного алгоритма, а применения различных операций предварительной обработки и моделей машинного обучения, объединенных в цепочку. В этой главе мы расскажем, как использовать класс `Pipeline`, чтобы упростить процесс построения цепочек преобразований и моделей. В частности, мы увидим, как можно объединить `Pipeline` и `GridSearchCV` для поиска параметров по всем операциям предварительной обработки сразу.

В качестве примера, который подчеркивает важность построения цепочек моделей, можно привести случай применения ядерного SVM к набору данных `cancer`. Значительного улучшения работы модели можно добиться, используя `MinMaxScaler` для предварительной обработки. Ниже приводится программный код для разбиения данных, вычисления минимума и максимума, масштабирования данных и построения SVM:

```
In[1]:
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# загружаем и разбиваем данные
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

# вычисляем минимум и максимум по обучающим данным
scaler = MinMaxScaler().fit(X_train)

In[2]:
# масштабируем обучающие данные
X_train_scaled = scaler.transform(X_train)

svm = SVC()
# строим SVM на масштабированных обучающих данных
svm.fit(X_train_scaled, y_train)
# масштабируем тестовые данные и оцениваем качество на масштабированных данных
X_test_scaled = scaler.transform(X_test)
print("Правильность на тестовом наборе: {:.2f}".format(svm.score(X_test_scaled, y_test)))

Out[2]:
Правильность на тестовом наборе: 0.95
```

## Отбор параметров с использованием предварительной обработки

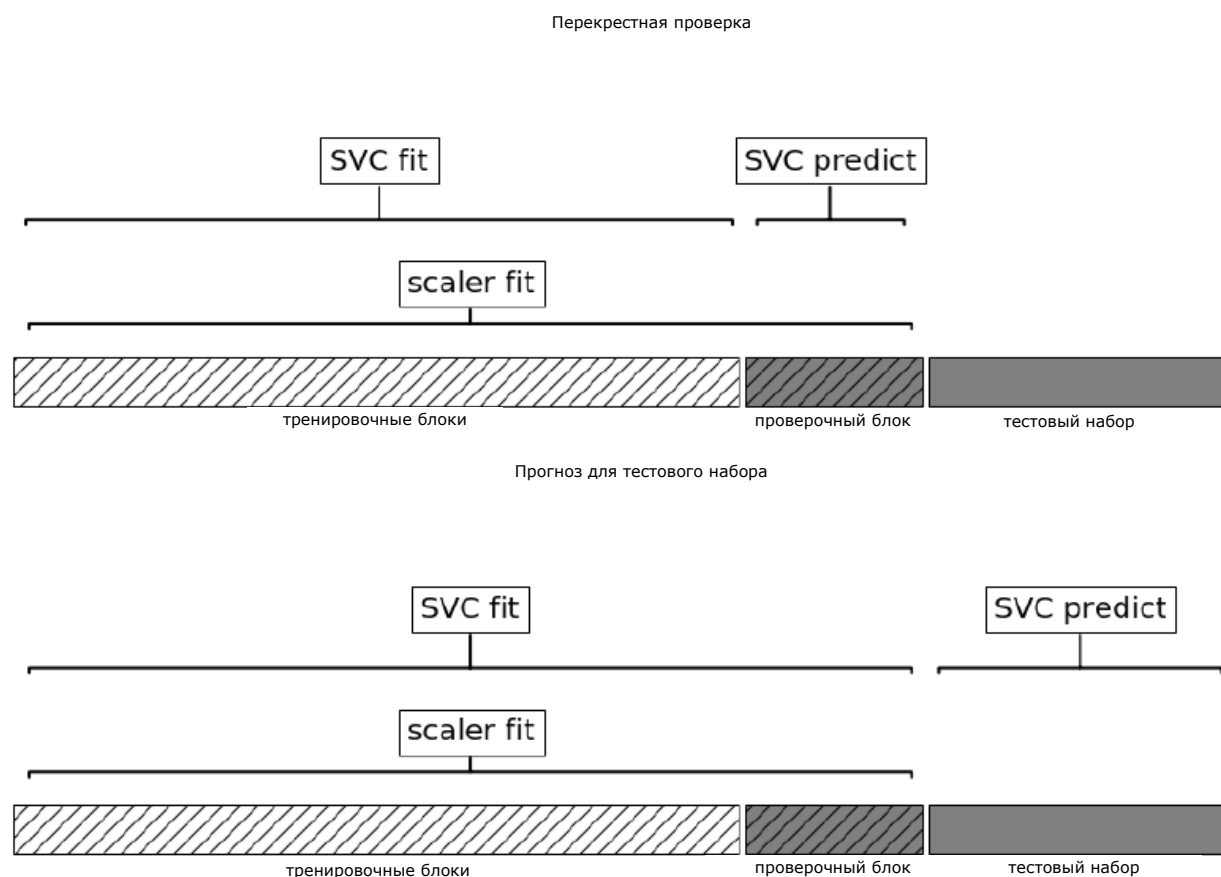
Теперь предположим, мы хотим найти более оптимальные параметры для SVC с помощью GridSearchCV, рассмотренного в главе 5. Как нам выполнить это? Наивный подход может выглядеть следующим образом:

```
In[3]:
from sklearn.model_selection import GridSearchCV
# только в иллюстративных целях, не используйте этот код!
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
grid.fit(X_train_scaled, y_train)
print("Наил знач правильности перекр проверки: {:.2f}".format(grid.best_score_))
print("Наил знач правильности на тесте: {:.2f}".format(grid.score(X_test_scaled, y_test)))
print("Наил параметры: ", grid.best_params_)
```

```
Out[3]:
Наил знач правильности перекр проверки: 0.98
Наил знач правильности на тесте: 0.97
Наил параметры: {'gamma': 1, 'C': 1}
```

Здесь мы запустили решетчатый поиск по параметрам SVC, использовав масштабированные данные. Однако нюанс заключается в том, как мы сейчас это сделали. При масштабировании данных мы использовали *все данные обучающего набора*, чтобы вычислить минимальные и максимальные значения признаков. Затем мы используем *масштабированные обучающие данные*, чтобы запустить наш решетчатый поиск с использованием перекрестной проверки. При каждом разбиении перекрестной проверки определенная часть исходного обучающего набора становится тренировочными блоками, а другая часть – проверочным блоком. Проверочный блок используется для оценки работы обученной модели на новых данных. Однако мы уже использовали информацию, содержащуюся в проверочном блоке, когда масштабировали данные. Вспомним, что проверочный блок в каждом разбиении перекрестной проверки является частью обучающего набора, а мы использовали информацию всего обучающего набора для поиска правильного масштаба данных. *Мы получим совершенно другое представление новых данных в модели.* Новые данные (скажем, представленные в виде тестового набора) не будут использованы при масштабировании обучающих данных и могут иметь значения минимума и максимума, отличающиеся от значений минимума и максимума для обучающих данных. Следующий пример (рис. 6.1) показывает различие между обработкой данных в ходе перекрестной проверки и итоговой оценкой:

```
In[4]:
mglearn.plots.plot_improper_processing()
```



**Рис. 6.1** Использование данных: предварительная обработка вынесена за пределы цикла перекрестной проверки

Таким образом, разбиения перекрестной проверки не позволяют больше адекватно моделировать новые данные. Мы уже «поделились» информацией, содержащейся в этих блоках, с моделью. Это приведет к чрезмерно оптимистичным результатам перекрестной проверки, и, возможно, к выбору субоптимальных параметров.

Чтобы обойти эту проблему, разбиения набора данных во время перекрестной проверки должны быть выполнены *перед предварительной обработкой данных*. Любой процесс, извлекающий знания из данных, должен осуществляться на обучающей части набора данных, и поэтому его следует разместить внутри цикла перекрестной проверки.

Для решения этой задачи в библиотеке `scikit-learn` наряду с функцией `cross_val_score` и функцией `GridSearchCV` мы можем воспользоваться классом `Pipeline`. Класс `Pipeline` позволяет «склеивать» вместе несколько операций обработки данных в единую модель `scikit-learn`. Класс `Pipeline` предусматривает методы `fit`, `predict` и `score` и имеет все те же свойства, что и любая модель `scikit-learn`. Чаще всего класс `Pipeline` используется для объединения операций предварительной обработки (например, масштабирования

данных) с моделью контролируемого машинного обучения типа классификатора.

## Построение конвейеров

Давайте посмотрим, как мы можем использовать класс `Pipeline`, чтобы осуществить обучение SVM после масштабирования данных с помощью `MinMaxScaler` (на этот раз не будем использовать решетчатый поиск). Во-первых, мы создаем объект-конвейер, передав ему список необходимых этапов. Каждый этап представляет собой кортеж, содержащий имя (любая строка на ваш выбор<sup>38</sup>) и экземпляр модели:

```
In[5]:
from sklearn.pipeline import Pipeline
pipe = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC())])
```

Здесь мы создали два этапа: первый этап, названный "scaler", является экземпляром `MinMaxScaler`, а второй, названный "svm", является экземпляром `SVC`. Теперь мы можем построить конвейер точно так же, как и любую другую модель `scikit-learn`:

```
In[6]:
pipe.fit(X_train, y_train)
```

В данном случае `pipe.fit` сначала вызывает метод `fit` объекта `scaler`, преобразует обучающие данные, используя `MinMaxScaler`, и, наконец, строит модель SVM на основе масштабированных данных. Чтобы оценить правильность модели на тестовых данных, мы просто вызываем `pipe.score`:

```
In[7]:
print("Правильность на тестовом наборе: {:.2f}".format(pipe.score(X_test, y_test)))
```

```
Out[7]:
Правильность на тестовом наборе: 0.95
```

Когда мы вызываем `pipe.score`, сначала тестовые данные масштабируются с помощью `MinMaxScaler`, затем к масштабированным тестовым данным применяется построенная модель SVM (происходит вызов метода `score` объекта `svm`). Видно, что приведенный вывод идентичен результату, который мы получили, используя программный код в начале главы, когда выполняли преобразования вручную. С помощью конвейера мы сократили программный код, необходимый для нашего процесса «предварительная обработка + классификация». Однако главное преимущество конвейера заключается в том, что сейчас мы

---

<sup>38</sup> За одним исключением: имя не должно содержать символ двойного подчеркивания `__`.



можем использовать эту отдельную модель в качестве аргумента функции `cross_val_score` или `GridSearchCV`.

## Использование конвейера, помещенного в объект `GridSearchCV`

Использование конвейера в объекте `GridSearchCV` аналогично использованию любой другой модели. Мы задаем сетку параметров для поиска и строим `GridSearchCV` на основе конвейера и сетки параметров. Однако теперь определение сетки параметров выглядит несколько иначе. Для каждого параметра нам нужно указать этап конвейера, к которому он относится. Оба параметра, которые мы хотим скорректировать, `C` и `gamma` являются параметрами `SVC`, то есть относятся ко второму этапу. Мы назвали этот этап "`svm`". Синтаксис, позволяющий настроить сетку параметров для конвейера, заключается в том, чтобы для каждого параметра указать имя этапа, затем символ двойного подчеркивания `__`, а потом имя параметра. Чтобы выполнить поиск по параметру `C` для `SVC`, мы в качестве ключа (сетка параметров представляет собой словарь) должны задать "`svm__C`", затем ту же самую процедуру нужно выполнить для `gamma`:

```
In[8]:
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],
              'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

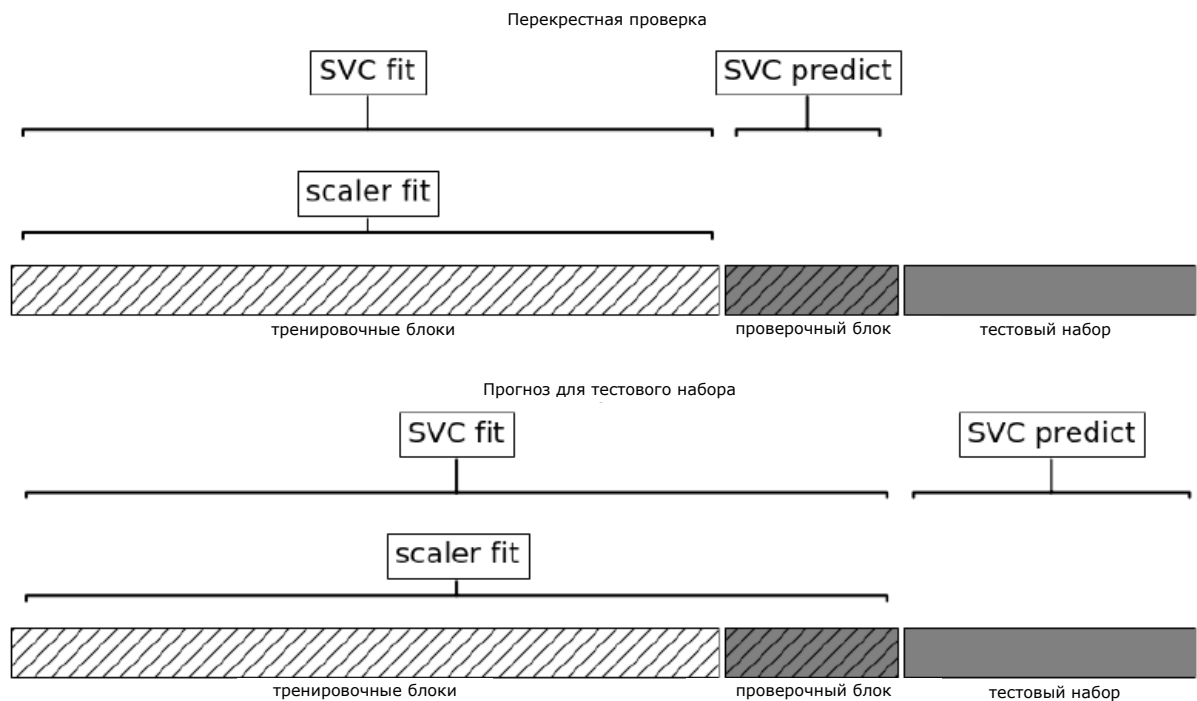
Задав сетку параметров, мы можем использовать `GridSearchCV` обычным образом:

```
In[9]:
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наил значение правильности перекр проверки: {:.2f}".format(grid.best_score_))
print("Правильность на тестовом наборе: {:.2f}".format(grid.score(X_test, y_test)))
print("Наилучшие параметры: {}".format(grid.best_params_))
```

```
Out[9]:
Наил значение правильности перекр проверки: 0.98
Правильность на тестовом наборе: 0.97
Наилучшие параметры: {'svm__C': 1, 'svm__gamma': 1}
```

В отличие от решетчатого поиска, выполненного ранее, теперь для каждого разбиения перекрестной проверки `MinMaxScaler` выполняет масштабирование данных, используя лишь обучающие блоки разбиений, и теперь информация тестового блока не передается модели при поиске параметров. Сравните выполнение перекрестной проверки и итоговой оценки теперь (рис. 6.2) и ранее (рис. 6.1):

```
In[10]:
mglearn.plots.plot_proper_processing()
```



**Рис. 6.2** Использование данных: предварительная обработка внутри цикла перекрестной проверки, используется конвейер

Последствия утечки информации, возникающей в ходе перекрестной проверки, обусловлены характером предварительной обработки. Масштабирование данных с использованием проверочного блока, как правило, не имеет серьезных последствий, в то время как использование проверочного блока для выделения и отбора признаков может привести к существенно различающимся результатам.

### Иллюстрация утечки информации

Замечательный пример утечки информации при проведении перекрестной проверки дан в книге Hastie, Tibshirani, Friedman *The Elements of Statistical Learning*, а мы приведем здесь адаптированный вариант. Рассмотрим синтетическую задачу регрессии со 100 наблюдениями и 10000 признаками, которые извлекаем независимо друг от друга из гауссовского распределения. Мы также сгенерируем зависимую переменную из гауссовского распределения.

```
In[11]:
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

При таком способе создания набора данных взаимосвязь между данными  $X$  и зависимой переменной  $y$  отсутствует (они независимы), поэтому невозможно построить модель на этих данных (модели нечему научиться). Теперь мы сделаем следующее. Во-первых, выберем самые информативные признаки с помощью `SelectPercentile`, а затем оценим качество регрессионной модели `Ridge` с помощью перекрестной проверки:

```
In[12]:
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print("форма массива X_selected: {}".format(X_selected.shape))
```

```
Out[12]:
форма массива X_selected: (100, 500)
```

```
In[13]:
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
print("Правильность перекр проверки (cv только для ridge): {:.2f}".format(
    np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))))
```

```
Out[13]:
Правильность перекр проверки (cv только для ridge): 0.91
```

Среднее значение  $R^2$ , вычисленное в результате перекрестной проверки, равно 0.91, что указывает на очень хорошее качество модели. Ясно, что данный результат не может быть правильным, поскольку наши данные получены совершенно случайным образом. То, что произошло здесь, обусловлено тем, что из 10000 случайных признаков были выбраны некоторые характеристики, которые (по чистой случайности) имеют сильную корреляцию с зависимой переменной. Поскольку мы осуществляли отбор признаков *вне* перекрестной проверки, это позволило нам найти признаки, которые коррелировали с зависимой переменной как в обучающем, так и в тестовом блоках. Информация, которая «просочилась» из тестовых наборов, была очень информативной и привела к весьма нереалистичным результатам. Давайте сравним этот результат с результатом правильной перекрестной проверки, использующей конвейер:

```
In[14]:
pipe = Pipeline([("select", SelectPercentile(score_func=f_regression,
                                             percentile=5)),
                 ("ridge", Ridge())])
print("Правильность перекр проверки (конвейер): {:.2f}".format(
    np.mean(cross_val_score(pipe, X, y, cv=5))))
```

```
Out[14]:
Правильность перекр проверки (конвейер): -0.25
```

На этот раз мы получаем *отрицательное* значение  $R^2$ , что указывает на очень плохое качество модели. Когда используется конвейер, отбор признаков осуществляется *внутри* цикла перекрестной проверки. Это означает, что для отбора признаков могут использоваться только обучающие блоки, а не тестовый блок. Процедура отбора признаков находит характеристики, которые коррелируют с зависимой переменной в обучающем наборе, но поскольку данные выбраны случайным образом, то в тестовом наборе корреляции между найденными признаками и зависимой переменной не обнаруживаются. В этом примере устранение утечки информации при выборе признаков привело к получению двух взаимоисключающих выводов о качестве модели: модель работает очень хорошо и модель вообще не работает.

## Общий интерфейс конвейера

Класс `Pipeline` не ограничивается предварительной обработкой и классификацией, с его помощью можно объединить любое количество моделей. Например, можно создать конвейер, включающий в себя выделение признаков, отбор признаков, масштабирование и классификацию, в общей сложности четыре этапа. Кроме того, последним этапом вместо классификации может быть регрессия или кластеризация.

Единственное требование, предъявляемое к моделям в конвейере, заключается в том, что все этапы, кроме последнего, должны использовать метод `transform`, таким образом, они позволяют сгенерировать новое представление данных, которое можно использовать на следующем этапе.

Во время вызова `Pipeline.fit` конвейер поочередно вызывает метод `fit`, а затем метод `transform` каждого этапа, вводная информация представляет собой вывод метода `transform` для предыдущего этапа. Для последнего этапа конвейера просто вызывается метод `fit`.

Опустив некоторые мелкие детали, все вышесказанное можно реализовать с помощью программного кода, приведенного ниже. Следует помнить, что `pipeline.steps` является списком кортежей, поэтому `pipeline.steps[0][1]` является первой моделью, а `line.steps[1][1]` – второй моделью и так далее:

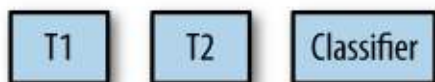
```
In[15]:
def fit(self, X, y):
    X_transformed = X
    for name, estimator in self.steps[:-1]:
        # перебираем все этапы, кроме последнего
        # подгоняем и преобразуем данные
        X_transformed = estimator.fit_transform(X_transformed, y)
    # осуществляем подгонку на последнем этапе
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

При прогнозировании с помощью конвейера мы одинаковым образом преобразуем данные на всех этапах, кроме последнего, а затем вызываем метод `predict` на последнем этапе:

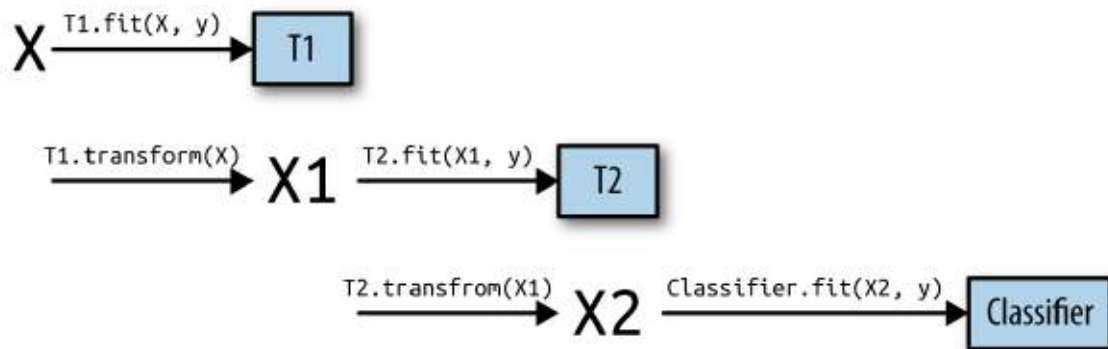
```
In[16]:
def predict(self, X):
    X_transformed = X
    for step in self.steps[:-1]:
        # перебираем все этапы, кроме последнего
        # преобразуем данные
        X_transformed = step[1].transform(X_transformed)
    # получаем прогнозы на последнем этапе
    return self.steps[-1][1].predict(X_transformed)
```

На рис. 6.3 проиллюстрирован конвейер, включающий два модификатора T1 и T2 и классификатор (`Classifier`).

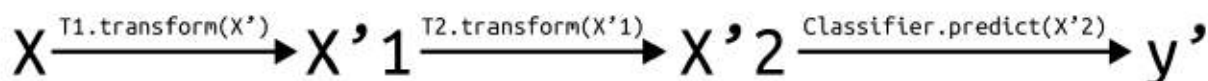
```
pipe = make_pipeline(T1(), T2(), Classifier())
```



```
pipe.fit(X, y)
```



```
pipe.predict(X')
```



**Рис. 6.3** Схема конвейера, предназначенного для обучения и получения прогнозов

На самом деле конвейер может иметь более общий вид. Использование `predict` на последнем этапе конвейера не является обязательным требованием и мы могли бы создать конвейер, который содержит `scaler` и `PCA`. Затем, поскольку последний шаг (`PCA`) использует метод `transform`, мы могли бы вызвать метод `transform` конвейера, чтобы получить вывод `PCA.transform`, примененный к данным, которые были обработаны на предыдущем этапе. Последний этап конвейера требуется только для применения метода `fit`.

## Удобный способ построения конвейеров с помощью функции `make_pipeline`

Построение конвейера с помощью вышеописанного синтаксиса иногда выглядит немного громоздким и, как правило, нам нет необходимости присваивать имя каждому этапу. Существует удобная функция `make_pipeline`, которая позволяет создать конвейер и автоматически присвоить имя каждому этапу, исходя из его класса (напомним, что

каждый этап представляет собой кортеж, содержащий имя и экземпляр модели). Синтаксис `make_pipeline` выглядит следующим образом:

```
In[17]:
from sklearn.pipeline import make_pipeline
# стандартный синтаксис
pipe_long = Pipeline([("scaler", MinMaxScaler()), ("svm", SVC(C=100))])
# сокращенный синтаксис
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

Объекты-конвейеры `pipe_long` и `pipe_short` выполняют одну и ту же последовательность операций, но в случае с `pipe_short` имена этапов присваиваются автоматически. Мы можем взглянуть на имена этапов с помощью атрибута `steps`:

```
In[18]:
print("Этапы конвейера:\n{}".format(pipe_short.steps))

Out[18]:
Этапы конвейера:
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),
 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
            decision_function_shape=None, degree=3, gamma='auto',
            kernel='rbf', max_iter=-1, probability=False,
            random_state=None, shrinking=True, tol=0.001,
            verbose=False))]
```

Этапам присвоены имена `minmaxscaler` и `SVC`. В общем, имена этапов – это просто названия классов, написанные строчными буквами. Если несколько этапов используют один и тот же класс, добавляется номер:

```
In[19]:
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())
print("Этапы конвейера:\n{}".format(pipe.steps))

Out[19]:
Этапы конвейера:
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_std=True)),
 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,
            svd_solver='auto', tol=0.0, whiten=False)),
 ('standardscaler-2', StandardScaler(copy=True, with_mean=True, with_std=True))]
```

Видно, что первый этап `StandardScaler` был назван `standardscaler-1`, а второй – `standardscaler-2`. Однако в данной ситуации было бы лучше использовать архитектуру конвейера с явными именами, чтобы присвоить этапам более содержательные названия.

## Работа с атрибутами этапов

Часто бывают ситуации, когда вам нужно посмотреть атрибуты одного из этапов конвейера, например, коэффициенты линейной модели или компоненты, извлекаемые с помощью `PCA`. Самый простой способ

получить подробную информацию об этапах конвейера заключается в том, чтобы воспользоваться атрибутом `named_steps`, который является словарем с именами этапов и моделями:

```
In[20]:
# подгоняем заранее заданный конвейер к набору данных cancer
pipe.fit(cancer.data)
# извлекаем первые две главные компоненты на этапе "pca"
components = pipe.named_steps["pca"].components_
print("форма components: {}".format(components.shape))

Out[20]:
форма components: (2, 30)
```

## Работа с атрибутами конвейера, помещенного в объект GridSearchCV

Как мы уже говорили ранее в этой главе, одна из главных причин использования конвейеров – это выполнение решетчатого поиска. Общераспространенная задача – получить доступ к некоторым этапам конвейера внутри объекта `GridSearchCV`. Давайте запустим решетчатый поиск для классификатора `LogisticRegression` на наборе данных `cancer`, используя `Pipeline` и `StandardScaler`, чтобы отмасштабировать данные перед тем, как передать их в классификатор `LogisticRegression`. Сначала мы создаем конвейер с помощью функции `make_pipeline`:

```
In[21]:
from sklearn.linear_model import LogisticRegression

pipe = make_pipeline(StandardScaler(), LogisticRegression())
```

Далее мы создаем сетку параметров. Как объяснялось в главе 2, параметр регуляризации `C` позволяет настроить модель логистической регрессии (класс `LogisticRegression`). Мы используем логарифмическую сетку для этого параметра, поиск осуществляется в диапазоне значений от 0.01 до 100. Поскольку мы использовали функцию `make_pipeline`, имя этапа `LogisticRegression` записывается в нижнем регистре как `logisticregression`. Чтобы настроить параметр `C`, мы должны задать сетку параметров в виде `logisticregression__C`:

```
In[22]:
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

И как обычно мы разбиваем набор данных `cancer` на обучающий и тестовый наборы и запускаем решетчатый поиск:

```
In[23]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
```



```
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

Итак, каким образом мы можем посмотреть коэффициенты наилучшей модели логистической регрессии, которые были найдены с помощью `GridSearchCV`? Из главы 5 мы знаем, что наилучшая модель, найденная с помощью `GridSearchCV` и построенная на всех обучающих данных, хранится в `grid.best_estimator_`:

```
In[24]:
print("Лучшая модель:\n{}".format(grid.best_estimator_))
```

```
Out[24]:
Лучшая модель:
Pipeline(steps=[
  ('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)),
  ('logisticregression', LogisticRegression(C=0.1, class_weight=None,
dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100,
multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,
solver='liblinear', tol=0.0001, verbose=0, warm_start=False))])
```

В нашем случае наилучшей моделью (`best_estimator_`) является конвейер, состоящий из двух этапов `standardscaler` и `logisticregression`. Как уже говорилось ранее, получить информацию об этапе `logisticregression` мы можем с помощью атрибута конвейера `named_steps`:

```
In[25]:
print("Этап логистической регрессии:\n{}".format(
  grid.best_estimator_.named_steps["logisticregression"]))
```

```
Out[25]:
Этап логистической регрессии:
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
verbose=0, warm_start=False)
```

Теперь, когда мы построили логистическую регрессию, можно взглянуть на регрессионные коэффициенты (веса), связанные с входными признаками:

```
In[26]:
print("Коэффициенты логистической регрессии:\n{}".format(
  grid.best_estimator_.named_steps["logisticregression"].coef_))
```

```
Out[26]:
Коэффициенты логистической регрессии:
[[-0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39  -0.058  0.209
 -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21  0.224
 -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

В результате мы получим довольно длинное регрессионное уравнение, но оно полезно для понимания модели.

## Находим оптимальные параметры этапов конвейера с помощью решетчатого поиска

С помощью конвейеров мы можем инкапсулировать все этапы предварительной обработки в одной модели `scikit-learn`. Еще одно преимущество конвейеров заключается в том, что теперь мы можем *настроить параметры предварительной обработки*, используя результат, полученный с помощью модели контролируемого машинного обучения (то есть результат решения регрессионной или классификационной задачи). При работе с набором данных `boston` мы перед применением гребневой регрессии создали полиномиальные признаки. Теперь давайте используем конвейер. Конвейер включает три этапа – масштабирование данных, вычисление полиномиальных признаков и построение гребневой регрессии:

```
In[27]:
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target,
                                                    random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

Как мы узнаем, какие степени полиномов нужно выбрать, выбирать ли полиномы или взаимодействия вообще? В идеале мы хотим выбрать значение параметра `degree`, основываясь на результатах классификации. С помощью нашего конвейера мы можем осуществить поиск значений параметра `degree` для полиномиальных преобразований значениями одновременно с поиском значений параметра `alpha` модели гребневой регрессии. Для этого мы задаем сетку параметров в необходимом формате: после каждого имени этапа следует двойной символ подчеркивания и соответствующий параметр:

```
In[28]:
param_grid = {'polynomialfeatures__degree': [1, 2, 3],
              'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Теперь мы можем запустить наш решетчатый поиск снова:

```
In[29]:
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

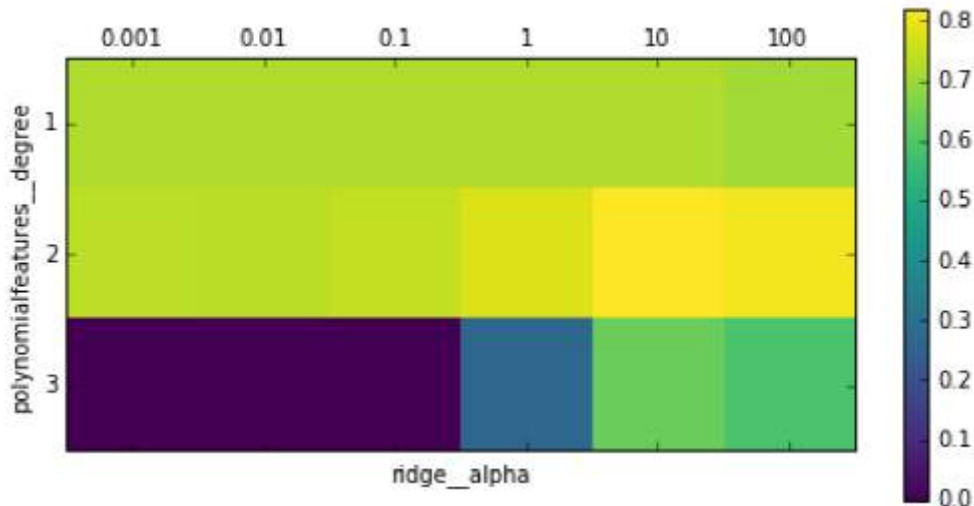
Результат перекрестной проверки можно визуализировать с помощью теплокарты (рис. 6.4), как мы уже делали это в главе 5:

```

In[30]:
plt.matshow(grid.cv_results_['mean_test_score'].reshape(3, -1),
             vmin=0, cmap="viridis")
plt.xlabel("ridge__alpha")
plt.ylabel("polynomialfeatures__degree")
plt.xticks(range(len(param_grid['ridge__alpha'])), param_grid['ridge__alpha'])
plt.yticks(range(len(param_grid['polynomialfeatures__degree'])),
           param_grid['polynomialfeatures__degree'])

plt.colorbar()

```



**Рис. 6.4** Теплокarta для усредненной правильности перекрестной проверки, выраженной в виде функции двух параметров: параметра degree для полиномиального преобразования и параметра alpha для гребневой регрессии

Взглянув на результаты, полученные с помощью перекрестной проверки, мы можем увидеть, что степень полинома 2 помогает, однако степень полинома 3 дает гораздо худший результат, чем степень 1 или степень 2. Данный факт отражается в найденных наилучших параметрах:

```

In[31]:
print("Наилучшие параметры: {}".format(grid.best_params_))

```

```

Out[31]:
Наилучшие параметры: {'polynomialfeatures__degree': 2, 'ridge__alpha': 10}

```

которые дают следующее значение правильности:

```

In[32]:
print("Правильность на тестовом наборе: {:.2f}".format(grid.score(X_test, y_test)))

```

```

Out[32]:
Правильность на тестовом наборе: 0.77

```

Давайте для сравнения запустим решетчатый поиск без полиномиального преобразования:

```

In[33]:
param_grid = {'ridge__alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)

```

```
print("Правильность без полином. преобразования: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[33]:

Правильность без полином. преобразования: 0.63

Как мы и предполагали, анализируя результаты решетчатого поиска, приведенные на рис. 6.4, отказ от использования полиномиальных признаков привел к существенно худшим результатам.

Одновременный поиск параметров предварительной обработки и параметров модели является очень мощной стратегией. Однако имейте в виду, что `GridSearchCV` перебирает *все возможные комбинации* заданных параметров. Поэтому включение в сетку большого количества параметров ведет к экспоненциальному росту моделей.

## Выбор оптимальной модели с помощью решетчатого поиска

Вы можете пойти дальше, объединив `GridSearchCV` и `Pipeline`: можно осуществлять поиск лишь по фактическим этапам, выполняемым в конвейере (например, речь может идти о целесообразности использования `StandardScaler` или `MinMaxScaler`). Подобное действие приведет к еще большему пространству поиска и нужно тщательно взвесить его целесообразность. Как правило, перебор всех возможных моделей не является оптимальной стратегией машинного обучения. Однако ниже приводится пример сравнения результатов работы `RandomForestClassifier` и `SVC` на наборе данных `iris`. Мы знаем, что `SVC`, возможно, потребуются отмасштабированные данные, поэтому необходимо решить, использовать `StandardScaler` или обойтись без предварительной обработки. Что касается `RandomForestClassifier`, мы знаем, что для него предварительная обработка данных не требуется. Мы начинаем с построения конвейера. В данном случае мы задаем имена этапов в явном виде. Наш конвейер будет включать два этапа: один – для предварительной обработки, второй – для классификатора. Создаем экземпляры объектов с помощью `SVC` и `StandardScaler`:

In[34]:

```
pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC())])
```

Теперь мы можем задать сетку параметров для поиска. Нам нужно выбрать либо `RandomForestClassifier`, либо `SVC`. Поскольку они используют разные параметры для настройки, один метод нуждается в предварительной обработке, а другой – нет, мы можем воспользоваться списком словарей, в котором каждый словарь представляет отдельную сетку параметров (см. раздел «Экономичный решетчатый поиск»). Чтобы задать модель для этапа, мы должны указать имя этапа в качестве

названия параметра. Если нам нужно пропустить какой-то этап в конвейере (например, потому что нам не нужна предварительная обработка для `RandomForest`), мы можем задать для этапа значение `None`:

```
In[35]:
from sklearn.ensemble import RandomForestClassifier

param_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
```

Мы можем создать экземпляр класса `GridSearchCV` и запустить решетчатый поиск в обычном режиме на наборе данных `cancer`.

```
In[36]:
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)

print("Наилучшие параметры:\n{}\n".format(grid.best_params_))
print("Наил значение правильности перекр проверки: {:.2f}".format(grid.best_score_))
print("Правильность на тестовом наборе: {:.2f}".format(grid.score(X_test, y_test)))
```

```
Out[36]:
Наилучшие параметры:
{'classifier':
  SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
      decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
      max_iter=-1, probability=False, random_state=None, shrinking=True,
      tol=0.001, verbose=False),
 'preprocessing':
  StandardScaler(copy=True, with_mean=True, with_std=True),
 'classifier__C': 10, 'classifier__gamma': 0.01}
```

```
Наил значение правильности перекр проверки: 0.99
Правильность на тестовом наборе: 0.98
```

По итогам решатчатого поиска становится ясно, что модель `SVC` с предварительной обработкой `StandardScaler`, параметрами `C=10` и `gamma=0.01` дает наилучший результат.

## Выводы и перспективы

В этой главе мы рассказали о классе `Pipeline`, инструменте, позволяющем объединять в одну цепочку несколько этапов предварительной обработки. В реальности проекты машинного обучения редко состоят из одной лишь модели, чаще всего они представляют собой последовательность этапов предварительной обработки. Конвейеры позволяет инкапсулировать несколько этапов в один питоновский объект, который поддерживает уже знакомый интерфейс `scikit-learn`, предлагая воспользоваться методами `fit`, `predict`, `transform`. Если говорить более конкретно, применение класса `Pipeline`, охватывающего

все этапы предварительной обработки, важно для правильной оценки качества модели. Кроме того, класс `Pipeline` позволяет писать более лаконичный код и уменьшает вероятность ошибок, которые могут быть допущены при построении цепочек операций без использования класса `pipeline` (например, мы можем забыть применить все преобразования к тестовому набору или можем применить их в неправильном порядке). Выбор оптимального сочетания извлеченных признаков, стратегии предварительной обработки, а также модели – это в определенной степени искусство, овладеть которым можно методом проб и ошибок. Однако использование конвейеров довольно существенно облегчает «экспериментирование» с различными операциями предварительной обработки данных. При проведении экспериментов постарайтесь не слишком усложнять процессы подготовки данных и убедитесь в том, что каждый оцениваемый компонент, включенный в ваш конвейер, является необходимым этапом.

Этой главой мы завершаем наш обзор инструментов и алгоритмов библиотеки `scikit-learn`. Теперь вы обладаете всеми необходимыми навыками и знакомы с механизмами применения машинного обучения на практике. В следующей главе мы более подробно разберем еще один конкретный тип данных, который часто встречается на практике, и его правильная обработка требует специальных знаний. Речь пойдет о текстовых данных.

## ГЛАВА 7. РАБОТА С ТЕКСТОВЫМИ ДАННЫМИ

В главе 4 мы говорили о двух типах признаков, которые могут представлять свойства данных: непрерывных признаках, описывающих количество, и категориальных признаках, которые являются элементами фиксированного списка. Существует еще и третий тип признаков, который можно встретить в различных областях – текст. Например, при классификации сообщений электронной почты на спам и действительно нужные письма, сам по себе текст письма, безусловно, будет содержать важную информацию для данной классификационной задачи. Или нам нужно узнать мнение какого-то политика об иммиграции. В данном случае тексты его выступлений или твиты могут дать полезную информацию. При обслуживании клиентов нам часто требуется выяснить, является ли сообщение жалобой или запросом. Проанализировав тему и содержание сообщения, мы можем автоматически определить намерения клиента, а это в свою очередь позволит нам направить сообщение в соответствующий отдел или даже отправить клиенту автоматический ответ.

Текстовые данные обычно представлены в виде строк, состоящих из символов. Во всех приведенных примерах длина текстовых данных будет разной. Текстовая информация очень отличается от ранее рассмотренных нами непрерывных признаков, и нам сначала предстоит обработать данные, прежде чем мы сможем применить к ним алгоритмы машинного обучения.

### Строковые типы данных

Прежде чем углубиться в вопросы предварительной обработки текстовых данных, предшествующие применению машинного обучения, мы хотим кратко рассмотреть различные типы текстовых данных, с которыми можно столкнуться на практике. Текст, как правило, представлен в наборе данных в виде обычной строки, однако далеко не все строковые признаки обрабатываются как текст. Строковый признак иногда может представлять собой категориальные переменные, о чем мы уже говорили в главе 5. Обработка строковых признаков невозможна без предварительного анализа данных.

На практике можно встретить четыре типа строковых данных:

- Категориальные данные
- Неструктурированные строки, которые по смыслу можно сгруппировать в категории
- Структурированные строки
- Текстовые данные



*Категориальные данные (categorical data)* представляют собой данные, которые берутся из фиксированного списка. Например, вы собираете данные с помощью онлайн-опроса, в ходе которого просите людей назвать их любимый цвет и для регистрации ответов используете выпадающий список из 8 значений («красный», «зеленый», «синий», «желтый», «черный», «белый», «фиолетовый» и «розовый»), позволяющий респондентам выбрать нужный вариант. В итоге мы получим восемь различных возможных значений, которые явно можно закодировать в виде категориальной переменной. Чтобы удостовериться в том, что строковые данные можно закодировать в виде категориальной переменной, достаточно взглянуть на них (если вы увидите очень много различных строковых значений, то скорее всего эти значения не являются категориальной переменной) и подтвердить свои предположения, вычислив количество уникальных значений по всему набору данных и, возможно, построив гистограмму частот встречаемости каждого значения. Кроме того, вам, возможно, потребуется проверить, соответствуют ли полученные категории интересующим вас категориям. Возможно в ходе опроса кто-то обнаружит, что слово «черный» написано с ошибкой как «ченый» и внесет соответствующую правку. В результате ваш набор данных будет содержать как значения «черный», так и значения «ченый», которые имеют одно и то же смысловое значение и должны быть объединены в одну категорию.

Теперь представьте себе, что вместо выпадающего меню вы позволите пользователям самим заполнять текстовое поле, отвечая на вопрос о любимом цвете. Многие, возможно, напишут ответы типа «черный» или «синий». Другие могут допустить орфографические ошибки, использовать различные варианты написания, например, «черный» или «чёрный», либо использовать более запоминающиеся и специфические названия типа «полуночно-синий». Кроме того, будут и вовсе странные названия. Несколько хороших примеров можно привести из [XKCD Color Survey](#), где люди должны были назвать цвета и придумывали названия типа «VelociRaptor cloaka» и «оранжевый как кабинет моего стоматолога. Я до сих пор вспоминаю его перхоть, медленно залетающую в мой широко раскрытый рот», которые довольно трудно автоматически (или вообще) сопоставить определенным цветам. Ответы, записанные в текстовом поле, относятся ко второй категории списка, *неструктурированным строкам, которые по смыслу можно сгруппировать в категории (free strings that can be semantically mapped to categories)*. Вероятно, лучшее всего закодировать их в виде категориальной переменной. Вы можете выбрать категории, выявив часто встречающиеся записи или задав категории, которые включают в себя содержательные

ответы. Возможно, вам потребуется задать несколько категорий и для вполне стандартных цветов, например, категорию «несколько цветов» для людей, которые дали ответы типа «зеленые и красные полосы», а также категорию «другое» для ответов, которые не могут быть закодированы иным способом. Предварительная обработка строковых данных может быть очень трудоемкой и ее сложно автоматизировать. Если у вас есть возможность повлиять на процесс сбора данных и требуется проанализировать понятия, которые лучше всего описываются с помощью категориальных переменных, настоятельно рекомендуем вам отказаться от ручной фиксации данных.

Как правило, строковые значения, введенные вручную, не соответствуют фиксированным категориям, но при этом все же имеют некоторую базовую *структуру (structure)*, например, адреса, названия мест, имена и фамилии людей, даты, номера телефонов и другие идентификаторы. Этот тип строк очень трудно спарсить и их обработка сильно зависит от контекста и предметной области. Обработка подобных данных выходит за рамки этой книги.

Последняя категория строковых данных – это *текстовые данные (text data)*, которые состоят из фраз или предложений. Примерами таких данных могут быть твиты, логи чата или отзывы о гостинице, а также собрание сочинений Шекспира, содержание Википедии или проекта «Гутенберг», включающего 50000 электронных книг. Все эти коллекции содержат информацию, представленную преимущественно в виде предложений, составленных из слов.<sup>39</sup> Для простоты давайте предположим, что все наши документы написаны на одном, английском языке.<sup>40</sup> С точки зрения анализа текста набор данных часто называют *корпусом (corpus)* и каждая точка данных, представленная в виде отдельного текста, называется *документом (document)*. Эти термины берут свое начало из *информационного поиска (information retrieval, IR)* и *обработки естественного языка (natural language processing, NLP)*, которые главным образом применяются при анализе текстовых данных.

## Пример применения: анализ тональности киноотзывов

В качестве примера мы воспользуемся набором данных, который содержит киноотзывы, оставленные на сайте IMDb (Internet Movie

---

<sup>39</sup> Возможно, что контент веб-сайтов, упомянутых в твитах, содержит гораздо больше информации, чем текст самих твитов.

<sup>40</sup> Большая часть материала, которая будет изложена нами в оставшейся части этой главы, применима и к другим языкам, использующим латиницу, а также частично к языкам, в которых можно определить границы слов. В китайском языке границы слова на письме не обозначаются и поэтому возникают проблемы, которые затрудняют применение методов, изложенных в этой главе.

Database) и собранные исследователем Стэнфордского университета Эндрю Маасом.<sup>41</sup> Этот набор данных содержит тексты отзывов, а также метки, которые указывают тональность отзыва («положительный» или «отрицательный»). Сайт IMDb имеет собственную систему оценки фильмов от 1 до 10. Чтобы упростить процесс моделирования, эта система оценки будет сведена к двум классам, отзывы с оценкой 6 или выше помечаются как положительные, а остальные отзывы помечаются как отрицательные. Вопрос, касающийся качества подготовки исходных данных, мы оставляем открытым и просто используем данные в том виде, в каком их нам предоставил Эндрю Маас.

После распаковки набор данных представляет собой две отдельные папки с текстовыми файлами, одна папка – для обучения, а вторая – для тестирования. Каждая папка в свою очередь содержит две подпапки, одна называется *pos*, а другая – *neg*.

```
In[2]:  
!tree -L 2 C:/Data/aclImdb
```

```
Out[2]:  
C:/Data/aclImdb  
├── test  
│   ├── neg  
│   └── pos  
├── train  
│   ├── neg  
│   └── pos  
6 directories, 0 files
```

Папка *pos* содержит все положительные отзывы, каждый отзыв записан в виде отдельного текстового файла, папка *neg* содержит все отрицательные отзывы и так же каждый отзыв представлен в виде отдельного текстового файла. В библиотеке **scikit-learn** есть вспомогательная функция **load\_files**. Она позволяет загрузить файлы, для хранения которых используется такая структура папок, где каждая вложенная папка соответствует определенной метке. Сначала мы применим функцию **load\_files** к обучающим данным:

```
In[3]:  
from sklearn.datasets import load_files  
reviews_train = load_files("C:/Data/aclImdb/train/")  
# load_files возвращает коллекцию, содержащую обучающие тексты и обучающие метки  
text_train, y_train = reviews_train.data, reviews_train.target  
print("тип text_train: {}".format(type(text_train)))  
print("длина text_train: {}".format(len(text_train)))  
print("text_train[1]:\n{}".format(text_train[1]))
```

```
Out[3]:  
тип text_train: <class 'list'>  
длина text_train: 25000  
text_train[1]:  
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing  
only. You have too see it for yourself to get at grip of how horrible a movie  
really can be. Not that I recommend you to do that. There are so many  
clich\x3\xa9s, mistakes (and all other negative things you can imagine) here
```

<sup>41</sup> Набор данных доступен по ссылке <http://ai.stanford.edu/~amaas/data/sentiment/>.

that will just make you cry. To start with the technical first, there are a LOT of mistakes regarding the airplane. I won't list them here, but just mention the coloring of the plane. They didn't even manage to show an airliner in the colors of a fictional airline, but instead used a 747 painted in the original Boeing livery. Very bad. The plot is stupid and has been done many times before, only much, much better. There are so many ridiculous moments here that i lost count of it really early. Also, I was on the bad guys' side all the time in the movie, because the good guys were so stupid. "Executive Decision" should without a doubt be you're choice over this one, even the "Turbulence"-movies are better. In fact, every other movie in the world is better than this one.'

Видно, что `text_train` представляет собой список длиной 25000, в котором каждый элемент представляет собой строку, содержащую отзыв. Мы напечатали отзыв с индексом 1. Кроме того, можно увидеть, что отзыв содержит разрывы строк HTML (`<br />`). Хотя эти разрывы вряд ли сильно повлияют на модель машинного обучения, лучше выполнить очистку данных и удалить символы форматирования перед тем, как начать работу.

```
In[4]:
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

Тип элементов `text_train` будет зависеть от вашей версии Python. В Python 3 они будут иметь тип `bytes`, который представляет собой двоичное кодирование строковых данных. В Python 2 `text_train` состоит из строк. В данном случае мы не будем вдаваться в подробности различных строковых типов данных, имеющих в Python, но мы рекомендуем вам прочитать разделы документации по [Python 2](#) и/или [Python 3](#), касающиеся строковых данных и Unicode.

Набор данных был собран таким образом, чтобы положительный и отрицательный классы были сбалансированы, поэтому количество строк с положительными отзывами и количество строк с отрицательными отзывами одинаковое:

```
In[5]:
print("Количество примеров на класс (обучение): {}".format(np.bincount(y_train)))
```

```
Out[5]:
Количество примеров на класс (обучение): [12500 12500]
```

Аналогичным образом загружаем тестовые данные:

```
In[6]:
reviews_test = load_files("C:/Data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Количество документов в текстовых данных: {}".format(len(text_test)))
print("Количество примеров на класс (тест): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

```
Out[6]:
Количество документов в текстовых данных: 25000
Количество примеров на класс (тест): [12500 12500]
```

Задача, которую мы хотим решить, можно сформулировать следующим образом: каждому отзыву нам нужно присвоить метку «положительный» или «отрицательный» на основе текста отзыва. Это стандартная задача бинарной классификации. Однако текстовые данные представлены в формате, который модель машинного обучения не умеет обрабатывать. Нам нужно преобразовать строковое представление текста в числовое представление, к которому можно применить алгоритмы машинного обучения.

## Представление текстовых данных в виде «мешка слов»

Один из самых простых, но эффективных и широко используемых способов подготовки текста для машинного обучения – это представление текстовой информации в виде «мешка слов» (*bag-of-words*). Используя это представление, мы удаляем структуру исходного текста, например, главы, параграфы, предложения, форматирование, и лишь подсчитываем частоту встречаемости каждого слова в каждом документе корпуса. Удаление структуры и подсчет частоты каждого слова позволяет получить образное представление текста в виде «мешка слов». Получение представления «мешок слов» включает следующие три этапов:

1. *Токенизация (tokenization)*. Разбиваем каждый документ на слова, которые встречаются в нем (*токены*), например, с помощью пробелов и знаков пунктуации.
2. *Построение словаря (vocabulary building)*. Собираем словарь всех слов, которые появляются в любом из документов, и пронумеровываем их (например, в алфавитном порядке).
3. *Создание разреженной матрицы (sparse matrix encoding)*. Для каждого документа подсчитываем, как часто каждое из слов, занесенное в словарь, встречается в документе.

Этапы 1 и 2 имеют некоторые нюансы, которые мы обсудим подробнее чуть ниже. Сейчас давайте посмотрим, как мы можем применить обработку данных «мешок слов», используя `scikit-learn`. Рис. 7.1 иллюстрирует процесс на примере строки "This is how you get ants". В итоге каждый документ можно представить в виде вектора частот слов. Для каждого слова, записанного в словаре, мы подсчитываем частоту его встречаемости в каждом документе. Это означает, что в нашем числовом представлении каждый признак соответствует каждому уникальному слову набора данных. Обратите внимание, порядок слов в исходной строке абсолютно не имеет никакого значения для представления признаков «мешок слов».



**Рис. 7.1** Обработка «мешок слов»

## Применение модели «мешка слов» к синтетическому набору данных

Модель «мешка слов» реализована в классе `CountVectorizer`, который выполняет соответствующее преобразование. Давайте сначала применим `CountVectorizer` к синтетическому набору данных, состоящему из двух примеров, чтобы проиллюстрировать его работу:

```
In[7]:
bards_words = ["The fool doth think he is wise,",
               "but the wise man knows himself to be a fool"]
```

Мы импортируем класс `CountVectorizer`, создаем экземпляр класса и подгоняем модель к нашим синтетическим данным следующим образом:

```
In[8]:
from sklearn.feature_extraction.text import CountVectorizer
vect = CountVectorizer()
vect.fit(bards_words)
```

Процесс подгонки `CountVectorizer` включает в себя токенизацию обучающих данных и построение словаря, к которому мы можем получить доступ с помощью атрибута `vocabulary_`:



```

In[9]:
print("Размер словаря: {}".format(len(vect.vocabulary_)))
print("Содержимое словаря:\n {}".format(vect.vocabulary_))

Out[9]:
Размер словаря: 13
Содержимое словаря:
{'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,
 'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}

```

Словарь состоит из 13 слов, начинается со слова «be» и заканчивается словом «wise».

Чтобы получить представление «мешок слов» для обучающих данных, мы вызываем метод **transform**:

```

In[10]:
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))

Out[10]:
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
with 16 stored elements in Compressed Sparse Row format>

```

Представление «мешок слов» записывается в разреженной матрице SciPy, которая хранит только ненулевые элементы (см. главу 1). Матрица имеет форму 2 x 13, в ней каждая строка соответствует точке данных, а каждый столбец (признак) соответствуют слову, записанному в словаре. Использование разреженной матрицы обусловлено тем, что документы, как правило, содержат лишь небольшое количество слов, записываемое в словарь, таким образом, большая часть элементов массива будет равна 0. Подумайте о том, сколько различных слов может встретиться в киноотзыве, учитывая словарный запас английского языка. Хранение всех этих нулей – ненужные затраты памяти. Чтобы взглянуть на фактическое содержимое разреженной матрицы, мы можем преобразовать ее в «плотный» массив NumPy (который помимо ненулевых элементов также хранит все нулевые элементы) с помощью метода **toarray**:<sup>42</sup>

```

In[11]:
print("Плотное представление bag_of_words:\n{}".format(
    bag_of_words.toarray()))

Out[11]:
Плотное представление bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]

```

Видно, что частоты слов равны либо 0, либо 1. Ни одна из двух строк массива **bards\_words** не содержит слов, которые встречались бы дважды. Давайте разберемся, как нужно работать с этими векторами признаков.

---

<sup>42</sup> Это возможно благодаря тому, что мы используем небольшой синтетический набор данных, который содержит лишь 13 слов. Если бы мы взяли реальный набор данных, то получили бы исключение **MemoryError**.



Первая строка ("The fool doth think he is wise") соответствует первому ряду элементов, и слово "be", записанное в словаре первым, встречается в ней ноль раз. Второе слово "but" тоже встречается в этой строке ноль раз. Третье слово "doth" встречается один раз. Взглянув на оба ряда, мы можем увидеть, что четвертое слово "fool", десятое слово "the" и тринадцатое слово "wise" встречаются в обеих строках.

## Модель «мешка слов» для киноотзывов

Теперь, когда мы детально разобрали процесс построения модели «мешка слов», давайте применим ее для анализа тональности киноотзывов. Ранее мы загрузили наши обучающие и тестовые данные, сформированные на основе отзывов IMDb, в виде списков строк (`text_train` и `text_test`) и сейчас обрабатываем их:

```
In[12]:
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))

Out[12]:
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
  with 3431196 stored elements in Compressed Sparse Row format>
```

Матрица `X_train` соответствует обучающим данным, представленным в виде «мешка слов». Она имеет форму 25000 x 74849, указывая на то, что словарь включает 74849 элементов. Как видим, данные снова записаны в виде разреженной матрицы SciPy. Давайте более детально исследуем словарь. Еще один способ получить доступ к словарю – это использование метода `get_feature_name`. Он возвращает удобный список, в котором каждый элемент соответствует одному признаку:

```
In[13]:
feature_names = vect.get_feature_names()
print("Количество признаков: {}".format(len(feature_names)))
print("Первые 20 признаков:\n{}".format(feature_names[:20]))
print("Признаки с 20010 по 20030:\n{}".format(feature_names[20010:20030]))
print("Каждый 2000-й признак:\n{}".format(feature_names[::2000]))

Out[13]:
Количество признаков: 74849
Первые 20 признаков:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
 '01', '01pm', '02']
Признаки с 20010 до 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Каждый 2000-й признак:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bete', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
```

```
'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

Возможно, факт того, что первые 10 элементов словаря являются числами, немного удивителен. Все эти числа встречаются в отзывах и поэтому рассматриваются как слова. Большинство из этих чисел не имеют никакого смысла, за исключением числа "007", которое скорее всего связано с фильмами о Джеймсе Бонде.<sup>43</sup> Выделение значимой информации из незначимых «слов» является иногда довольно сложной задачей. Далее мы находим в словаре ряд английских слов, начинающихся с "dra". Можно заметить, что единственное число слов "draught", "drawback" и "drawer" обрабатывается отдельно от их множественного числа. Данные термины очень тесно связаны между собой по смыслу и обработка этих слов как разных, соответствующих различным признакам, не может быть оптимальным решением.

Перед тем как мы пытаемся улучшить выделение признаков, давайте измерим качество модели, построив классификатор. У нас есть обучающие метки, хранящиеся в `y_train` и обучающие данные, представленные в виде «мешка слов» `X_train`, таким образом, мы можем обучить классификатор по этим данным. Как правило, для подобных высокоразмерных разреженных данных лучше всего работают линейные модели типа `LogisticRegression`.

Давайте сначала применим `LogisticRegression` с использованием перекрестной проверки:<sup>44</sup>

```
In[14]:
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Средняя правильность перекр проверки: {:.2f}".format(np.mean(scores)))
```

```
Out[14]:
Средняя правильность перекр проверки: 0.88
```

Мы получаем среднее значение правильности перекрестной проверки, равное 88%, что указывает на приемлемое качество модели для задачи сбалансированной бинарной классификации. Известно, что логистическая регрессия имеет параметр регуляризации `C`, который мы можем настроить с помощью перекрестной проверки:

```
In[15]:
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
```

---

<sup>43</sup> Беглый анализ данных подтверждает, что это действительно так. Попробуйте самостоятельно убедиться в этом!

<sup>44</sup> Внимательный читатель может заметить, что в данном случае мы нарушаем принципы перекрестной проверки, изложенные в главе 6. На самом деле используя настройки, установленные для `CountVectorizer` по умолчанию, мы не собираем какие-либо статистики, поэтому наши результаты достоверны. Использование конвейера с самого начала было бы наилучшим выбором, но мы отложим его ради простоты.

```

grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наилучшее значение перекрестной проверки: {:.2f}".format(grid.best_score_))
print("Наилучшие параметры: ", grid.best_params_)

```

**Out[15]:**

Наилучшее значение перекрестной проверки: 0.89  
Наилучшие параметры: {'C': 0.1}

Используя  $C=0.1$ , мы получаем значение перекрестной проверки 89%. Теперь мы можем оценить на тестовом наборе обобщающую способность при использовании данной настройки параметра:

**In[16]:**

```

X_test = vect.transform(text_test)
print("Правильность на тестовом наборе: {:.2f}".format(grid.score(X_test, y_test)))

```

**Out[16]:**

Правильность на тестовом наборе: 0.88

Теперь, давайте посмотрим, можно ли улучшить процесс извлечения слов. `CountVectorizer` извлекает токены с помощью регулярных выражений. По умолчанию используется регулярное выражение `"\b\w\w+\b"`. Для тех, кто не знаком с регулярными выражениями, поясним: это выражение позволяет найти все последовательности символов, которые состоят как минимум из двух букв или цифр (`\w`) и отделены друг от друга границами слов (`\b`). Его не интересуют слова, состоящие из одного символа, сокращения типа «doesn't» или «bit.ly» оно разбивает на два слова, однако последовательность символов «h8ter» будет обработана как отдельное слово. Затем `CountVectorizer` преобразует все слова в строчные, поэтому слова «soon», «Soon» и «sOon» соответствуют одному и тому же токenu (и, следовательно, одному и тому же признаку). Этот простой принцип достаточно хорошо работает на практике, однако, как мы уже видели ранее, можно получить массу неинформативных признаков (например, числа). Один из способов решить эту проблему – использовать только те токены, которые встречаются по крайней мере в двух документах (или по крайней мере в пяти документах и т.д.). Токен, который встретился только в одном документе, вряд ли встретится в тестовом наборе и поэтому бесполезен. С помощью параметра `min_df` мы можем задать минимальное количество документов, в котором должен появиться токен.

**In[17]:**

```

vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train c min_df: {}".format(repr(X_train)))

```

**Out[17]:**

X\_train c min\_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>' with 3354014 stored elements in Compressed Sparse Row format>

Задав `min_df=5`, мы уменьшаем количество признаков до 27271, и если сравнить этот результат с предыдущим выводом, теперь мы используем лишь треть исходных признаков. Давайте снова взглянем на токены:

```
In[18]:
feature_names = vect.get_feature_names()

print("Первые 50 признаков:\n{}".format(feature_names[:50]))
print("Признаки с 20010 по 20030:\n{}".format(feature_names[20010:20030]))
print("Каждый 700-й признак:\n{}".format(feature_names[::700]))

Out[18]:
Первые 50 признаков:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',
 '160', '1600', '16mm', '16s', '16th']
Признаки с 20010 по 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',
 'replays', 'replete', 'replica']
Каждый 700-й признак:
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',
 'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',
 'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',
 'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional', 'parking',
 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas', 'shea',
 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

Четко видно, что намного реже стали встречаться числа и, похоже, что исчезли некоторые странные или неправильно написанные слова. Давайте оценим качество нашей модели, вновь выполнив решатчатый поиск:

```
In[19]:
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))

Out[19]:
Наилучшее значение перекр проверки: 0.89
```

Наилучшее значение правильности, полученное в ходе перекрестной проверки, по-прежнему равно 89%. Мы не смогли улучшить качество нашей модели, однако сокращение количества признаков ускорит предварительную обработку, а исключение бесполезных признаков, возможно, улучшит интерпретабельность модели.



Если метод `transform` класса `CountVectorizer` применяется для документа, который содержит слова, отсутствующие в обучающем наборе, эти слова будут проигнорированы, поскольку не являются частью словаря. Это не относится к проблеме классификации, так как невозможно узнать какую-либо информацию о словах,

отсутствующих в обучающих данных. Однако в некоторых прикладных задачах, например, для обнаружения спама, возможно, было бы полезным добавить признак, который бы фиксировал, сколько слов, отсутствующих в словаре, встретилось в каждом документе. Для этого вам необходимо задать `min_df`, в противном случае этот признак не будет использоваться в ходе обучения.

## Стоп-слова

Еще один способ, с помощью которого мы можем избавиться от неинформативных слов – исключение слов, которые встречаются слишком часто, чтобы быть информативными. Существуют два основных подхода: использование списка стоп-слов (на основе соответствующего языка), или удаление слов, которые встречаются слишком часто. Библиотека `scikit-learn` предлагает встроенный список английских стоп-слов, реализованный в модуле `feature_extraction.text`:

```
In[20]:
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Количество стоп-слов: {}".format(len(ENGLISH_STOP_WORDS)))
print("Каждое 10-е стоп-слово:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

```
Out[20]:
Количество стоп-слов: 318
Каждое 10-е стоп-слово:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
 'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
 'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
 'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

Очевидно, что удаление стоп-слов с помощью списка может уменьшить количество признаков лишь ровно на то количество, которое есть в списке (318), но, возможно, это позволит улучшить качество модели. Давайте попробуем:

```
In[21]:
# настройка stop_words="english" задает встроенный список стоп-слов.
# мы можем также расширить его и передать свой собственный.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train с использованием стоп-слов:\n{}".format(repr(X_train)))
```

```
Out[21]:
X_train с использованием стоп-слов:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
  with 2149958 stored elements in Compressed Sparse Row format>
```

Теперь у нас стало на 305 признаков меньше (количество признаков уменьшилось с 27271 до 26966). Это означает, что большинство стоп-слов (но не все) встретились в корпусе документов. Давайте снова запустим решетчатый поиск:

```
In[22]:
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))
```

```
Out[22]:
Наилучшее значение перекр проверки: 0.88
```

При использовании стоп-слов качество модели чуть снизилось. Это не повод для беспокойства, однако учитывая тот факт, что исключение 305 признаков из более чем 27000 вряд ли сильно изменит качество или интерпретабельность модели, признаем, использование списка стоп-слов в данном случае бесполезно. Как правило, фиксированные списки могут быть полезны при работе с небольшими наборами данных. Небольшие наборы данных не имеют достаточного объема информации, позволяющего модели самостоятельно определить, какие слова являются стоп-словами. В качестве примера вы можете попробовать другой подход, исключив часто встречающиеся слова, задав опцию `max_df` для `CountVectorizer` и посмотреть, как это повлияет на количество признаков и качество модели.

## Масштабирование данных с помощью tf-idf

Следующий подход вместо исключения несущественных признаков пытается масштабировать признаки в зависимости от степени их информативности. Одним из наиболее распространенных способов такого масштабирования является метод *частота термина-обратная частота документа* (*term frequency-inverse document frequency*, *tf-idf*). Идея этого метода заключается в том, чтобы присвоить большой вес термину, который часто встречается в конкретном документе, но при этом редко встречается в остальных документах корпуса. Если слово часто появляется в конкретном документе, но при этом редко встречается в остальных документах, оно, вероятно, будет описывать содержимое этого документа лучше. В библиотеке `scikit-learn` метод *tf-idf* реализован в двух классах: `TfidfTransformer`, который принимает на вход разреженную матрицу, полученную с помощью `CountVectorizer`, и преобразует ее, и `TfidfVectorizer`, который принимает на вход текстовые данные и выполняет как выделение признаков «мешок слов», так и преобразование *tf-idf*. Для преобразования *tf-idf* существует несколько вариантов взвешивания частот, о которых вы можете прочитать в [Википедии](#). Значение *tf-idf* для слова  $w$  в документе  $d$  вычисляется с помощью классов `TfidfTransformer` и `TfidfVectorizer` по формуле:<sup>45</sup>

---

<sup>45</sup> Мы привели здесь эту формулу для большей ясности. Чтобы выполнить *tf-idf* преобразование, вам не обязательно помнить ее.



$$\text{tfidf}(w, d) = \text{tf} \log \left( \frac{N+1}{N_w+1} \right) + 1$$

где  $N$  – это количество документов в обучающем наборе,  $N_w$  – это количество документов обучающего набора, в которых встретилось слово  $w$ , и  $\text{tf}$  (частота термина) – это частота встречаемости термина в запрашиваемом документе  $d$  (документе, который вы хотите преобразовать). Кроме того, оба класса применяют L2 нормализацию после того, как вычисляют представление  $\text{tf-idf}$ . Другими словами, они масштабируют векторизованное представление каждого документа к единичной евклидовой норме (длине). Подобное масштабирование означает, что длина документа (количество слов) не меняет его векторизованное представление.

Поскольку  $\text{tf-idf}$  фактически использует статистические свойства обучающих данных, мы воспользуемся конвейером (о котором рассказывалось в главе 6), чтобы убедиться в достоверности результатов нашего решетчатого поиска. Для этого пишем следующий программный код:

```
In[23]:
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None),
                     LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))
```

```
Out[23]:
Наилучшее значение перекр проверки: 0.89
```

Видно, что применение преобразования  $\text{tf-idf}$  вместо обычных частот слов дало определенное улучшение. Кроме того, мы можем выяснить, какие слова в результате преобразования  $\text{tf-idf}$  стали наиболее важными. Имейте в виду, что масштабирование  $\text{tf-idf}$  призвано найти слова, которые лучше всего дискриминируют документы, но при этом оно является методом неконтролируемого обучения. Таким образом, «важное» не обязательно должно быть связано с интересующими метками «положительный отзыв» и «отрицательный отзыв». Сначала мы извлекаем из конвейера наилучшую модель `TfidfVectorizer`, найденную с помощью решетчатого поиска:

```
In[24]:
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# преобразуем обучающий набор данных
X_train = vectorizer.transform(text_train)
# находим максимальное значение каждого признака по набору данных
```



```

max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# получаем имена признаков
feature_names = np.array(vectorizer.get_feature_names())

print("Признаки с наименьшими значениями tfidf:\n{}".format(
    feature_names[sorted_by_tfidf[:20]]))

print("Признаки с наибольшими значениями tfidf: \n{}".format(
    feature_names[sorted_by_tfidf[-20:]]))

Out[24]:
Признаки с наименьшими значениями tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']
Признаки с наибольшими значениями tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']

```

Признаки с низкими значениями tf-idf – это признаки, которые либо встречаются во многих документах, либо используются редко и только в очень длинных документах. Интересно отметить, что многие признаки с высокими значениями tf-idf на самом деле соответствуют названиям некоторых шоу или фильмов. Эти термины встречаются лишь в отзывах, посвященных конкретному шоу или франшизе, но при этом они встречаются в данных отзывах очень часто. Это очевидно для таких терминов как "smallville" и "doodlebops", но в нашем случае и вполне нейтральное слово "scanners" тоже относится к названию фильма. Эти слова вряд ли помогут нам классифицировать тональность отзывов (если только некоторые франшизы не оцениваются всеми зрителями положительно или отрицательно), однако они, разумеется, содержат много конкретной информации об отзывах.

Кроме того, мы можем найти слова, которые имеют низкое значение обратной частоты документа, то есть слова, которые встречаются часто и поэтому считаются менее важными. Значения обратной частоты документа, найденные для обучающего набора, хранятся в атрибуте `idf_`:

```

In[25]:
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Признаки с наименьшими значениями idf:\n{}".format(
    feature_names[sorted_by_idf[:100]]))

Out[25]:
Признаки с наименьшими признаками idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
 'him']

```

Как и следовало ожидать, словами с низкими значениями idf стали английские стоп-слова типа "the" и "no". Но некоторые из них характерны для киноотзывов. Это слова типа "movie", "film", "time", "story" и так далее. Интересно, что в соответствии с метрикой tf-idf слова "good", "great" и "bad" также были отнесены к самым часто встречающимся и потому «самым нерелевантным» словам, хотя можно было бы ожидать, что они будут иметь очень важное значение для анализа тональности.

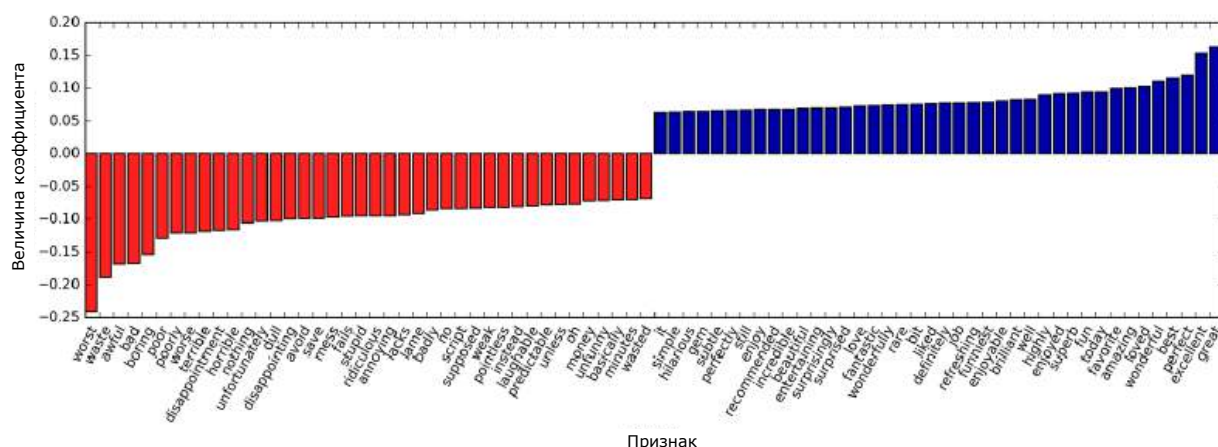
## Исследование коэффициентов модели

И, наконец, давайте посмотрим чуть более детально на информацию, полученную с помощью модели логистической регрессии. Поскольку у нас имеется большое количество признаков (27271 после удаления малоинформативных слов), мы не можем посмотреть все коэффициенты сразу. Однако мы можем посмотреть на коэффициенты, получившие максимальные значения, а также сопоставить их словам. Мы воспользуемся последней построенной моделью на основе признаков tf-idf.

Следующая гистограмма (рис. 7.2) показывает 25 наибольших и 25 наименьших коэффициентов модели логистической регрессии, каждый столбик соответствует величине коэффициента:

In[26]:

```
mglearn.tools.visualize_coefficients(
    grid.best_estimator_.named_steps["logisticregression"].coef_,
    feature_names, n_top_features=40)
```



**Рис. 7.2** Наибольшие и наименьшие значения коэффициентов логистической регрессии, построенной на основе признаков tf-idf

Отрицательные коэффициенты, расположенные в левой части гистограммы, относятся к словам, которые в соответствии с моделью указывают на негативные отзывы, а положительные коэффициенты,

расположенные в правой части гистограммы, принадлежат словам, которые означают положительные отзывы. Большая часть терминов интуитивно понятна, например, слова "worst", "waste", "disappointment" и "laughable" указывают на плохие киноотзывы, в то время как слова "excellent", "wonderful", "enjoyable" и "refreshing" свидетельствуют о положительных киноотзывах. Что касается слов типа "bit", "job" и "today", их связь с тональностью киноотзыва менее ясна, но они могут быть частью фразы, например, «good job» или «best today».

## Модель «мешка слов» для последовательностей из нескольких слов (n-грамм)

Один из главных недостатков представления «мешок слов» заключается в полном игнорировании порядка слов. Таким образом, две строки «it's bad, not good at all» и «it's good, not bad at all» будут иметь одинаковое представление, хотя противоположны по смыслу. Употребление частицы «not» перед словом – это лишь один из примеров того, какое важное значение имеет контекст. К счастью, существует способ, позволяющий учитывать контекст при использовании представления «мешок слов», фиксируя не только частоты одиночных токенов, но и пары, тройки токенов, которые появляются рядом друг с другом. Пары токенов называют *биграммами* (*bigrams*), тройки токенов известны как *триграммы* (*trigrams*), а в более широком смысле последовательности токенов известны как *n-граммы* (*n-grams*). Мы можем изменить диапазон токенов, которые рассматриваются в качестве признаков, изменив параметр `ngram_range` для `CountVectorizer` или `TfidfVectorizer`. Параметр `ngram_range` задает нижнюю и верхнюю границы диапазона *n*-значений для различных извлекаемых *n*-грамм. Таким образом, будут использованы все значения *n*, которые удовлетворяют условию  $\min\_n \leq n \leq \max\_n$ . Ниже приводится пример на основе синтетических данных, использованных нами ранее:

```
In[27]:
print("bards_words:\n{}".format(bards_words))

Out[27]:
bards_words:
['The fool doth think he is wise,',
 'but the wise man knows himself to be a fool']
```

По умолчанию для каждой последовательности токенов с `min_n=1` и `max_n=1` (одиночные токены еще называются *юниграммами* или *unigrams*) `CountVectorizer` или `TfidfVectorizer` создает один признак:

```
In[28]:
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
```

```
print("Размер словаря: {}".format(len(cv.vocabulary_)))
print("Словарь:\n{}".format(cv.get_feature_names()))
```

Out[28]:

Размер словаря: 13

Словарь:

```
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

Чтобы посмотреть только биграммы, то есть последовательности из двух токенов, следующих друг за другом, мы можем задать `ngram_range` равным (2, 2):

In[29]:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Размер словаря: {}".format(len(cv.vocabulary_)))
print("Словарь:\n{}".format(cv.get_feature_names()))
```

Out[29]:

Размер словаря: 14

Словарь:

```
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
 'think he', 'to be', 'wise man']
```

Использование более длинных последовательностей токенов, как правило, приводит к гораздо большему числу признаков и большей детализации признаков. Нет ни одной биграммы, которая встретила бы в обоих строках массива `bard_words`:

In[30]:

```
print("Преобразованные данные (плотн):\n{}".format(cv.transform(bards_words).toarray()))
```

Out[30]:

Преобразованные данные (плотн):

```
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

В большинстве прикладных задач минимальное количество токенов в последовательности должно быть равно единице, поскольку одиночные слова позволяют зафиксировать множество смысловых значений. Добавление биграмм помогает в большинстве случаев. Включение в анализ более длинных последовательностей, вплоть до 5-грамм, тоже, вероятно, поможет, но это вызовет взрывной рост количества признаков и может привести к переобучению, поскольку появится большое количество детализированных признаков. В принципе, количество биграмм может быть равно количеству юниграмм, возведенному в квадрат, а количество триграмм может быть равно количеству юниграмм в кубе, что приведет к очень большому пространству признаков. На практике количество сгенерированных *n*-грамм большей длины хоть и будет внушительным, но получится значительно меньше вышеназванных расчетных значений из-за структуры (английского) языка.

Теперь попробуем использовать для `bards_words` юниграммы, биграммы и триграммы:

```
In[31]:
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Размер словаря: {}".format(len(cv.vocabulary_)))
print("Словарь:\n{}".format(cv.get_feature_names()))

Out[31]:
Размер словаря: 39
Словарь:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
'to be fool', 'wise', 'wise man', 'wise man knows']
```

Давайте применим `TfidfVectorizer` к киноотзывам, собранных на сайте IMDb, и найдем оптимальное значение `ngram_range` с помощью решетчатого поиска:

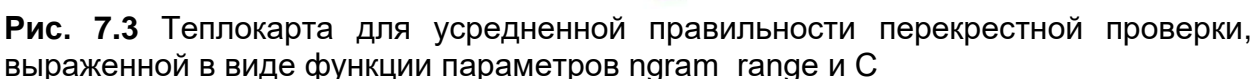
```
In[32]:
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# выполнение решетчатого поиска займет много времени из-за
# относительно большой сетки параметров и включения триграмм
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Наилучшее значение перекр проверки: {:.2f}".format(grid.best_score_))
print("Наилучшие параметры:\n{}".format(grid.best_params_))

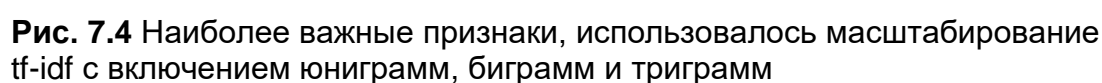
Out[32]:
Наилучшее значение перекр проверки: 0.91
Наилучшие параметры:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

Из результатов видно, что мы улучшили качество чуть более чем на один процент, добавив биграммы и триграммы. Мы можем представить правильность перекрестной проверки в виде функции параметров `ngram_range` и `C`, используя теплокарту, как это уже делали в главе 5 (см. рис. 7.3):

```
In[33]:
# извлекаем значения правильности, найденные в ходе решетчатого поиска
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# визуализируем теплокарту
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt="%.3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```



```
In[34]: # извлекаем названия признаков и коэффициенты
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```



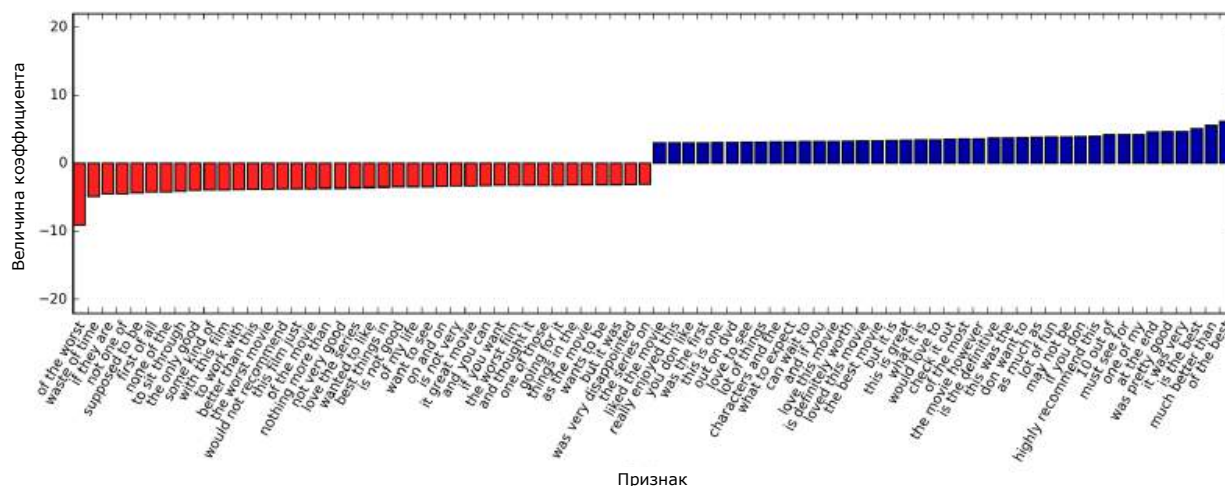
365



указывает на отрицательный отзыв, в то время как "definitely worth" и "worth" свидетельствуют о положительном отзыве. Это яркий пример того, как контекст влияет на смысл слова «worth».

Далее мы визуализируем только триграммы, чтобы лучше понять, какие признаки являются полезными. Многие полезные биграммы и триграммы типа "none of the", "the only good", "on and on", "this is one", "of the most" и другие состоят из общеупотребимых слов, которые не были бы информативными сами по себе. Однако, как можно увидеть на рис. 7.5, влияние триграммных признаков по сравнению с важностью униграммных признаков выражено гораздо слабее:

```
In[35]:
# находим триграммные признаки
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# визуализируем только 3-граммные признаки
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                   feature_names[mask], n_top_features=40)
```



**Рис. 7.5** Визуализация наиболее важных триграммных признаков модели модели

## Продвинутая токенизация, стемминг и лемматизация

Как упоминалось ранее, выделение признаков в `CountVectorizer` и `TfidfVectorizer` является относительно простым процессом, однако вы можете применить гораздо более сложные методы. В более сложных задачах обработки текста часто возникает необходимость улучшить токенизацию, которая является первым этапом создания модели «мешка слов». Этот этап определяет, что представляет собой слово в плане извлечения признаков.

Ранее мы видели, что словарь часто содержит одновременно единственное и множественное число одинаковых по смыслу слов, например, "drawback" и "drawbacks", "drawer" и "drawers", "drawing" и



"drawings". При построении модели «мешка слов» необходимо учитывать близость слов "drawback" и "drawbacks" по смыслу, присутствие этих слов в виде отдельных признаков лишь увеличит переобучение вместо того, чтобы позволить модели в полной мере использовать обучающие данные. Аналогично, мы обнаружили, что словарь включает в себя такие слова, как "replace", "replaced", "replacement", "replaces" и "replacing", которые представляют собой разные глагольные формы и существительное, связанное с глаголом "replace". Как и в случае с единственным и множественным числом, обработка различных глагольных форм и взаимосвязанных слов как отдельных токенов является препятствием, не позволяющим добиться хорошей обобщающей способности модели.

Эту проблему можно решить, найдя для каждого слова его *основу* (*word stem*). Это подразумевает идентификацию или *объединение* (*conflating*) всех слов с одной и той же основой. Если этот процесс выполняется с помощью эвристик на основе правил (например, удаление общих суффиксов), его обычно называют *стеммингом* (*stemming*). Если вместо этого используется словарь с заранее заданными формами слов (явный процесс, контролируемый человеком) и учитывается роль слова в предложении (то есть принимаем во внимание, к какой части речи относится слово), то этот процесс называется *лемматизацией* (*lemmatization*), а стандартизированная форма слова называется *леммой* (*lemma*). Лемматизация и стемминг являются способами *нормализации* (*normalization*), которые пытаются извлечь определенную нормальную (то есть начальную) форму слова. Еще один интересный случай нормализации – это исправление орфографических ошибок, которое может быть полезно на практике, однако выходит за рамки данной книги.

Чтобы получить более полное представление о нормализации, давайте сравним стемминг (мы воспользуемся стеммингом Портера, широко используемым набором эвристик, в данном случае импортируем его из пакета `nltk`) с лемматизацией, реализованной в пакете `spacy`:<sup>46</sup>

```
In[36]:
```

```
import spacy
import nltk
```

```
# загружаем модели пакета spacy для английского языка
```

```
en_nlp = spacy.load('en')
```

```
# создаем экземпляр стеммера Портера из пакета nltk
```

```
stemmer = nltk.stem.PorterStemmer()
```

```
# задаем функцию, сравнивающую лемматизацию в spacy со стеммингом в nltk
```

```
def compare_normalization(doc):
```

---

<sup>46</sup> Для получения дополнительной информации по интерфейсу обратитесь к документации по [nltk](#) и [spacy](#). В данном случае нас интересуют общие принципы работы. Если вы используете Anaconda, установите `spacy` с помощью команды `conda install -c spacy spacy=0.101.0`. Также не забудьте установить языковую модель `spacy` с помощью команды `python -m spacy.en.download`.

```

# токенизируем документ в spacy
doc_spacy = en_nlp(doc)
# печатаем леммы, найденные с помощью spacy
print("Лемматизация:")
print([token.lemma_ for token in doc_spacy])
# печатаем токены, найденные с помощью стеммера Портера
print("Стемминг:")
print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])

```

Мы сравним результаты лемматизации и использования стеммера Портера на простом примере:

```

In[37]:
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")

```

```

Out[37]:
Лемматизация:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']

```

```

Стемминг:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', 'm',
'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']

```

Стемминг всегда выполняет обрезку слова до его основы, поэтому "was" становится "wa", в то время как лемматизация может извлечь правильную базовую форму глагола "be". Аналогично, лемматизация может нормализовать "worse" к форме "bad", тогда как в результате стемминга мы получим "wors". Еще одно важное отличие состоит в том, что стемминг сокращает оба слова "meeting" до "meet". При использовании лемматизации первое вхождение слова "meeting" будет распознано как существительное и оставлено в неизменном виде, тогда как второе вхождение слова будет распознано как глагол и сокращено до "meet".

Несмотря на то что в библиотеке `scikit-learn` не реализован ни один из способов нормализации, `CountVectorizer` позволяет задать собственный токенизатор, который преобразует каждый документ в список токенов с помощью параметра `tokenizer`. Мы можем использовать лемматизацию из пакета `spacy`, чтобы создать функцию, которая примет в качестве аргумента строку и сгенерирует список лемм:

```

In[38]:
# С технической точки зрения мы хотим применить токенизатор на основе
# регулярных выражений (regex), который используется в CountVectorizer, а
# пакет spacy использовать лишь для лемматизации. Для этого мы
# заменим en_nlp.tokenizer (токенизатор пакета spacy)
# токенизатором на основе регулярных выражений.
import re
# regex, используемые в CountVectorizer
regex = re.compile('( ?u)\\b\\w\\w+\\b')

# загружаем языковую модель spacy и сохраняем старый токенизатор
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# заменяем токенизатор старым на основе регулярных выражений
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(

```

```

    regexp.findall(string))

# создаем пользовательский токенизатор с помощью конвейера обработки документов spacy
# (теперь используем наш собственный токенизатор)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# задаем countvectorizer с пользовательским токенизатором
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)

```

Давайте преобразуем данные и проверим размер словаря:

```

In[39]:
# преобразуем text_train, используя CountVectorizer с лемматизацией
X_train_lemma = lemma_vect.fit_transform(text_train)
print("форма X_train_lemma: {}".format(X_train_lemma.shape))

# стандартный CountVectorizer для сравнения
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("форма X_train: {}".format(X_train.shape))

Out[39]:
форма X_train_lemma: (25000, 21596)
форма X_train: (25000, 27271)

```

Как видно из вывода, в результате лемматизации количество признаков с 27271 (когда использовалась стандартная обработка CountVectorizer) снизилось до 21596. Лемматизацию можно рассматривать как своего рода регуляризацию, так как она объединяет некоторые признаки. Таким образом, мы ожидаем, что лемматизация улучшает качество модели, когда набор данных невелик. Чтобы проиллюстрировать пользу от применения лемматизации, мы применим стратегию перекрестной проверки StratifiedShuffleSplit, задав лишь 1% данных для обучения, а остальные данные будем использовать для тестирования:

```

In[40]:
# строим модель решетчатого поиска, используя 1% данных в качестве обучающего набора
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99,
                           train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# выполняем решетчатый поиск, используя данные, к которым был
# применен стандартный CountVectorizer
grid.fit(X_train, y_train)
print("Наилучшее значение перекрестной проверки "
      "(стандартный CountVectorizer): {:.3f}".format(grid.best_score_))
# выполняем решетчатый поиск, используя данные, к которым была
# применена лемматизация
grid.fit(X_train_lemma, y_train)
print("Наилучшее значение перекрестной проверки "
      "(лемматизация): {:.3f}".format(grid.best_score_))

Out[40]:
Наилучшее значение перекрестной проверки (стандартный CountVectorizer): 0.721
Наилучшее значение перекрестной проверки (лемматизация): 0.731

```

В данном случае лемматизация дала незначительное улучшение качества. Как и в случае с другими методами извлечения признаков, результат варьирует в зависимости от набора данных. Лемматизация и стемминг иногда могут помочь построить более качественные (или, по крайней мере, более компактные модели) модели, поэтому, если вы из последнего пытаетесь выжать качество модели, мы предлагаем вам воспользоваться этими методами.

## Моделирование тем и кластеризация документов

Еще один метод, который часто применяется к текстовым данным – *моделирование тем (topic modeling)*. Моделирование тем – это зонтичный термин, описывающий процедуру присвоения каждому документу одной или нескольких тем, которая осуществляется, как правило, без учителя. Хорошим примером моделирования тем являются новостные данные, которые можно сгруппировать по таким темам, как «политика», «спорт», «финансы» и так далее. Если каждый документ может иметь только одну тему, то речь идет о задаче кластеризации документов, которая рассматривалась в главе 3. Если каждый документ может иметь несколько тем, эта задача относится к декомпозиционным методам, освещавшимся в главы 3. Каждая полученная компонента соответствует одной теме, а коэффициенты компонент, которые описывают документ, позволяют нам судить о том, насколько тесно данный документ связан с конкретной темой. Часто, когда люди говорят о моделировании тем, они имеют в виду конкретный декомпозиционный метод под названием *латентное размещение Дирихле (Latent Dirichlet Allocation, LDA)*.<sup>47</sup>

## Латентное размещение Дирихле

Говоря простым языком, модель LDA пытается найти группы слов (темы или топики), которые часто появляются вместе. LDA также подразумевает, что каждый документ можно интерпретировать как «смесь» из нескольких тем. Важно понимать, что для модели машинного обучения «тема» - это далеко не то же самое, что мы подразумеваем под «темой» в повседневной речи. В данном случае «тема» больше напоминает извлекаемые с помощью PCA или NMF компоненты (о котором мы говорили в главе 3), которые могут иметь или не иметь

---

<sup>47</sup> Существует еще одна модель машинного обучения, которую тоже часто сокращенно называют LDA: линейный дискриминантный анализ (Linear Discriminant Analysis), являющийся линейной моделью классификации. Это приводит к некоторой путанице. В этой книге под LDA подразумевается латентное размещение Дирихле (Latent Dirichlet Allocation).

смысловое значение. Даже если «тема», полученная с помощью LDA, и имеет смысловое значение, все равно она не тождественна «теме» в ее традиционном понимании. Вернемся к примеру с новостными статьями. Представьте, у нас есть набор статей о спорте, политике и финансах, написанных двумя конкретными авторами. В политической статье мы могли бы ожидать появление слов типа «губернатор», «голос», «партия» и т.д., тогда как в спортивной статье мы могли бы встретить слова типа «сборная», «очко» и «сезон». Слова в каждой из этих групп с большой вероятностью встречаются вместе, в то время как вероятность совместного появления слов «сборная» и «губернатор» будет существенно меньше. Однако это не единственные группы слов, которые, по нашему мнению, встречаются вместе. Возможно, что два репортера предпочитают употреблять различные фразы или различные варианты слов. Возможно, один любит использовать слово «разграничивать», а другому нравится слово «поляризовать». Тогда «темами» уже будут «слова, часто используемые репортером А» и «слова, часто используемые репортером В», хотя они не являются темами в обычном смысле этого слова.

Давайте применим LDA к нашим киноотзывам, чтобы посмотреть, как этот метод работает на практике. Для моделей неконтролируемого обучения, применяющихся к текстовым документам, часто бывает полезно удалить наиболее часто употребляемые слова, поскольку в противном случае они в ходе анализа будут выбраны в качестве самых важных. Мы удалим слова, которые появляются по крайней мере в 15% документов, и ограничим модель «мешка слов» до 10000 слов, которые представляют собой наиболее часто встречающиеся слова, оставшиеся после удаления:

```
In[41]:  
vect = CountVectorizer(max_features=10000, max_df=.15)  
X = vect.fit_transform(text_train)
```

Мы построим модель, выделив 10 тем, что довольно мало для рассмотрения всех возможных вариантов.<sup>48</sup> Аналогично компонентам в NMF темы не имеют какого-то внутреннего порядка и изменение количества извлекаемых тем изменит содержательную суть всех тем. Мы воспользуемся методом обучения "batch", который работает несколько медленнее по сравнению с методом "online", установленным по умолчанию, но дает, как правило, лучшие результаты. Кроме того, мы увеличим значение параметра `max_iter`, что также позволит построить модель лучшего качества:

---

<sup>48</sup> На самом деле NMF и LDA решают во многом аналогичные задачи и мы могли бы также использовать NMF для извлечения тем.

```

In[42]:
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                               max_iter=25, random_state=0)
# Мы строим модель и преобразуем данные в один этап
# Преобразование займет некоторое время,
# и мы можем сэкономить время, выполнив обе операции сразу
document_topics = lda.fit_transform(X)

```

Как и декомпозиционные методы, рассмотренные нами в главе 3, `LatentDirichletAllocation` имеет атрибут `components_`, который хранит информацию о том, насколько каждое слово важно для каждой выделенной темы. Атрибут `components_` имеет форму `(n_topics, n_words)`:

```

In[43]:
lda.components_.shape

```

```

Out[43]:
(10, 10000)

```

Чтобы лучше понять содержательный смысл каждой темы, мы проанализируем наиболее важные слова для каждой из тем. Функция `print_topics` позволяет представить эти признаки в удобном формате:

```

In[44]:
# Для каждой темы (строки в components_) сортируем признаки (по возрастанию)
# Инвертируем строки с помощью[:, ::-1], чтобы получить сортировку по убыванию
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Получаем имена признаков из векторизатора
feature_names = np.array(vect.get_feature_names())

In[45]:
# Выводим 10 тем:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)

```



Out[45]:

topic 0	topic 1	topic 2	topic 3	topic 4
between	war	funny	show	didn
young	world	worst	series	saw
family	us	comedy	episode	am
real	our	thing	tv	thought
performance	american	guy	episodes	years
beautiful	documentary	re	shows	book
work	history	stupid	season	watched
each	new	actually	new	now
both	own	nothing	television	dvd
director	point	want	years	got

topic 5	topic 6	topic 7	topic 8	topic 9
horror	kids	cast	performance	house
action	action	role	role	woman
effects	animation	john	john	gets
budget	game	version	actor	killer
nothing	fun	novel	oscar	girl
original	disney	both	cast	wife
director	children	director	plays	horror
minutes	10	played	jack	young
pretty	kid	performance	joe	goes
doesn	old	mr	performances	around

Если судить по словам, связанных с той или иной темой, похоже, что тема 1 соответствует историческим и военным фильмам, тема 2, возможно, связана с плохими комедиями, а тема 3, вероятно, соответствует ТВ-сериалам. Похоже, что тема 4 вобрала в себя некоторые очень распространенные слова, тогда как тема 6, по всей видимости, связана с детскими фильмами. Тема 8, по-видимому, содержит отзывы, связанные с кинонаградами. При `n_topics=10` темы должны быть очень широкими, чтобы вообразить в себя все многообразие киноотзывов.

Теперь мы построим еще одну модель, на этот раз выделив 100 тем. Увеличение `n_topics` в значительной мере усложняет анализ, но при этом повышает вероятность найти с помощью полученных тем интересные подмножества данных:

In[46]:

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                   max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

Вывод всех 100 тем было бы немного громоздким, поэтому мы выбрали лишь некоторые интересные и характерные темы:

In[47]:

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])

sorting = np.argsort(lda100.components_, axis=1)[: , ::-1]
feature_names = np.array(vect.get_feature_names())
mglearn.tools.print_topics(topics=topics, feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=7, n_words=20)
```



Out[47]:

topic 7 -----	topic 16 -----	topic 24 -----	topic 25 -----	topic 28 -----
thriller	worst	german	car	beautiful
suspense	awful	hitler	gets	young
horror	boring	nazi	guy	old
atmosphere	horrible	midnight	around	romantic
mystery	stupid	joe	down	between
house	thing	germany	kill	romance
director	terrible	years	goes	wonderful
quite	script	history	killed	heart
bit	nothing	new	going	feel
de	worse	modesty	house	year
performances	waste	cowboy	away	each
dark	pretty	jewish	head	french
twist	minutes	past	take	sweet
hitchcock	didn	kirk	another	boy
tension	actors	young	getting	loved
interesting	actually	spanish	doesn	girl
mysterious	re	enterprise	now	relationship
murder	supposed	von	night	saw
ending	mean	nazis	right	both
creepy	want	spock	woman	simple
topic 36 -----	topic 37 -----	topic 41 -----	topic 45 -----	topic 51 -----
performance	excellent	war	music	earth
role	highly	american	song	space
actor	amazing	world	songs	planet
cast	wonderful	soldiers	rock	superman
play	truly	military	band	alien
actors	superb	army	soundtrack	world
performances	actors	tarzan	singing	evil
played	brilliant	soldier	voice	humans
supporting	recommend	america	singer	aliens
director	quite	country	sing	human
oscar	performance	americans	musical	creatures
roles	performances	during	roll	miike
actress	perfect	men	fan	monsters
excellent	drama	us	metal	apes
screen	without	government	concert	clark
plays	beautiful	jungle	playing	burton
award	human	vietnam	hear	tim
work	moving	ii	fans	outer
playing	world	political	prince	men
gives	recommended	against	especially	moon

Похоже, что темы, извлеченные на этот раз, более конкретны, хотя многие из них трудно интерпретировать. Тема 7, по-видимому, соответствует фильмам ужасов и триллерам, темы 16 и 54 зафиксировали плохие отзывы, тогда как тема 63 преимущественным образом вобрала в себя положительные отзывы о комедиях. Если мы хотим сделать дополнительные выводы о выделенных темах, мы должны подтвердить свои предположения, выдвинутые нами, исходя из анализа наиболее важных слов по каждой теме. Для этого необходимо взглянуть на документы, которые были отнесены к этим темам. Например, тема 45,

похоже, связана с музыкой. Давайте посмотрим, какие отзывы отнесены к этой теме:

In[48]:

```
# сортируем документы по весу темы 45 "музыка"
music = np.argsort(document_topics100[:, 45])[:, :-1]
# печатаем пять документов, в которых данная тема является наиболее важной
for i in music[:10]:
    # выводим первые два предложения
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

Out[48]:

```
b'I love this movie and never get tired of watching. The music in it is great.\n'
b'I enjoyed Still Crazy more than any film I have seen in years. A successful
band from the 70's decide to give it another try.\n'
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
Warner Bros. His directing style had changed or evolved to the point that
this film does not contain his signature overhead shots or huge production
numbers with thousands of extras.\n'
b'What happens to washed up rock-n-roll stars in the late 1990's?
They launch a comeback / reunion tour. At least, that's what the members of
Strange Fruit, a (fictional) 70's stadium rock group do.\n'
b'As a big-time Prince fan of the last three to four years, I really can't
believe I've only just got round to watching "Purple Rain". The brand new
2-disc anniversary Special Edition led me to buy it.\n'
b'This film is worth seeing alone for Jared Harris' outstanding portrayal
of John Lennon. It doesn't matter that Harris doesn't exactly resemble
Lennon; his mannerisms, expressions, posture, accent and attitude are
pure Lennon.\n'
b'The funky, yet strictly second-tier British glam-rock band Strange Fruit
breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
band members go their separate ways and uncomfortably settle into lackluster
middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
winds up penniless and down on his luck, vain, neurotic, pretentious lead
singer Bill Nighy tries (and fails) to pursue a floundering solo career,
paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
installs roofs for a living.\n'
b'I just finished reading a book on Anita Loos' work and the photo in TCM
Magazine of MacDonald in her angel costume looked great (impressive wings),
so I thought I'd watch this movie. I'd never heard of the film before, so I
had no preconceived notions about it whatsoever.\n'
b'I love this movie!!! Purple Rain came out the year I was born and it has had
my heart since I can remember. Prince is so tight in this movie.\n'
b'This movie is sort of a Carrie meets Heavy Metal. It's about a highschool
guy who gets picked on alot and he totally gets revenge with the help of a
Heavy Metal ghost.\n"
```

Как видно из вывода, данная тема охватывает широкий спектр музыкальных отзывов, посвященных мюзиклам, биографическим фильмам и трудно определимым жанрам, как в последнем отзыве. Еще один интересный способ исследовать темы – посмотреть, какой вес получает каждая тема в целом, просуммировав `document_topics` по всем отзывам. Каждой теме мы дадим названия, используя два самых часто встречаемых слова. Рис. 7.6 показывает вычисленные веса тем:

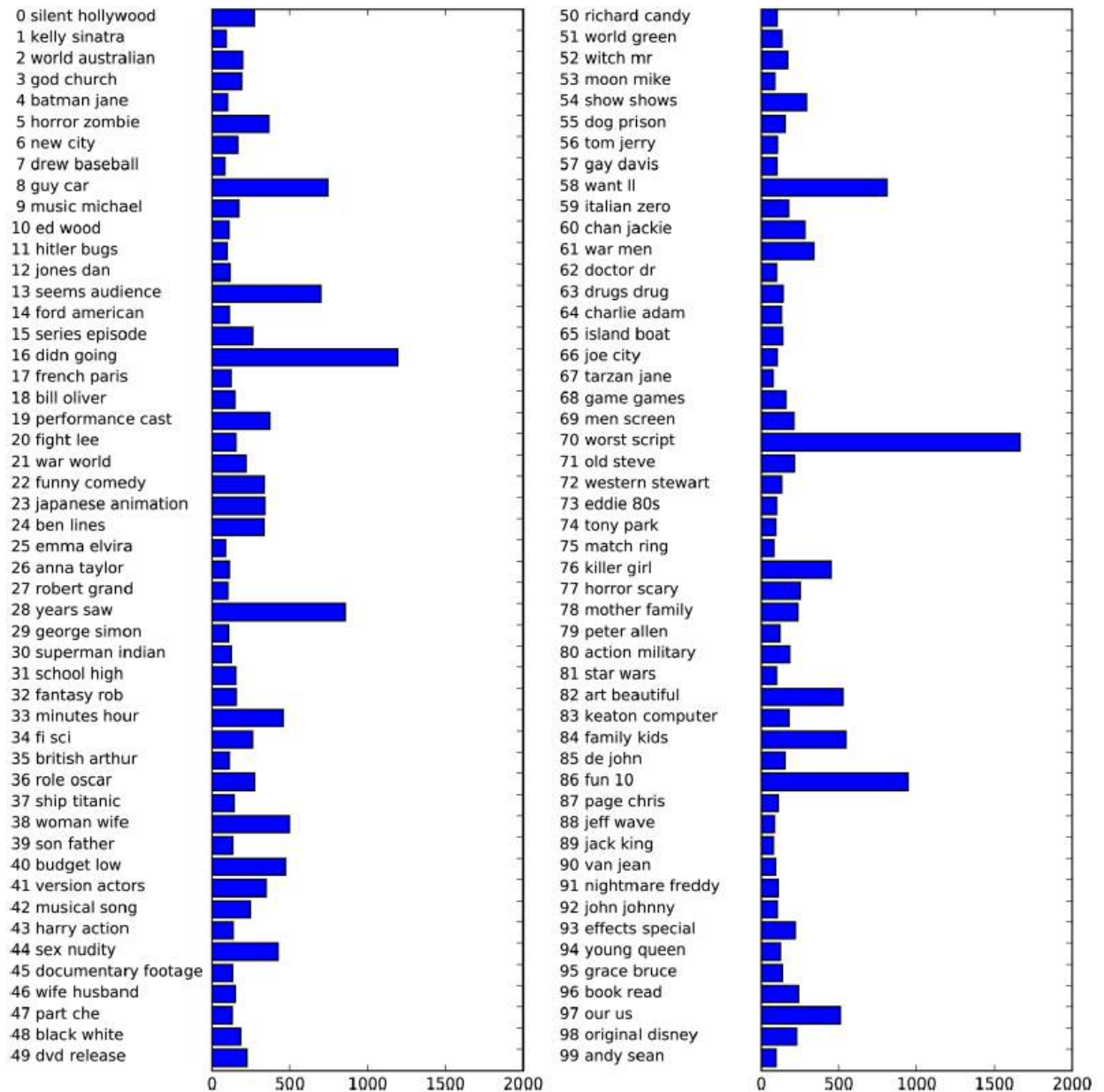
In[49]:

```
fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = [{">2} ".format(i) + " ".join(words)
               for i, words in enumerate(feature_names[sorting[:, :2]])]
# две столбиковые диаграммы:
for col in [0, 1]:
    start = col * 50
    end = (col + 1) * 50
```

```

ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
ax[col].set_yticks(np.arange(50))
ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
ax[col].invert_yaxis()
ax[col].set_xlim(0, 2000)
yax = ax[col].get_yaxis()
yax.set_tick_params(pad=130)
plt.tight_layout()

```



**Рис. 7.6** Веса топигов, полученные с помощью LDA

Наиболее важными темами являются темы 70 и 16, которые, по-видимому, содержат отрицательные отзывы.

Похоже, что LDA в основном выделил два типа тем: темы, описывающие жанровые особенности фильмов и темы, обосновывающие ту или иную рейтинговую оценку. Кроме того, несколько тем не удалось отнести к конкретному типу. Это интересное открытие, так как большинство отзывов составлены из нескольких комментариев,

связанных с жанром описываемого фильма, и нескольких комментариев, в которых автор обосновывает или подчеркивает свою оценку.

Модели, получаемые с помощью LDA, представляют собой интересные методы, позволяющие интерпретировать огромные корпуса текстов, когда метки классов отсутствуют или когда они имеются, как в данном случае. Однако алгоритм LDA является рандомизированным и разные значения параметра `random_state` могут привести к совершенно различным результатам. Несмотря на то что выделение тем может быть полезным, любые выводы, которые можно сделать, исходя из результатов модели неконтролируемого обучения, нужно принимать с определенной долей сомнения и мы рекомендуем проверять выводы, анализируя документы, присвоенные определенной теме. Кроме того, темы, полученные с помощью метода `LDA.transform`, можно иногда использовать в качестве входного признака для машинного обучения с учителем.

## Выводы и перспективы

В этой главе мы рассмотрели основы обработки текста, которая еще известна как *обработка естественного языка* (*natural language processing*, *NLP*), используя в качестве примера классификацию киноотзывов. Рассмотренные здесь инструменты должны стать идеальной отправной точкой для обработки текстовых данных. В частности, при решении задач классификации текстов типа обнаружения спама и мошенничества или анализа настроений представление «мешок слов» является простым и эффективным методом. Как это часто бывает в машинном обучении, представление данных играет ключевую роль в прикладных задачах NLP, а исследование извлекаемых токенов и *n*-грамм позволяет глубже понять суть процесса моделирования. При решении прикладных задач обработки текста (как контролируемых, так и неконтролируемых) часто существует возможность заглянуть внутрь модели, как мы уже видели в этой главе. Вы должны в полной мере воспользоваться этой возможностью при использовании методов NLP на практике.

Естественный язык и обработка текста – это крупная научная область, и обсуждение деталей передовых методов выходит за рамки данной книги. Если вы хотите получить больше информации, мы рекомендуем книгу издательства O'Reilly [Natural Language Processing with Python](#), написанную Стивеном Бердом, Эваном Кляйном и Эдвардом Лопером, в которой дан обзор NLP, а также рассказывается о питоновском пакете `nltk` для NLP. Еще одна интересная и более концептуальная книга – это стандартное справочное издание [Introduction to Information Retrieval](#), написанная Кристофером Меннингом, Прабхакаром Рагхаваном и

Генрихом Шютце и посвященная основным алгоритмам информационного поиска, NLP и машинного обучения. Обе книги имеют онлайн-версии, которые можно получить бесплатно. Как мы уже говорили ранее, классы `CountVectorizer` и `TfidfVectorizer` позволяют реализовать только относительно простые методы обработки текста. Чтобы воспользоваться более продвинутыми методами обработки текста, мы рекомендуем питоновские пакеты `spacy` (относительно новый, но очень эффективный и хорошо разработанный пакет), `nlTK` (очень хорошо отлаженная библиотека, хотя и несколько устаревшая) и `gensim` (пакет, предназначенный для NLP, с упором на моделирование тем).

В последние годы в области обработки текста появилось несколько очень интересных направлений, которые выходят за рамки этой книги и относятся к нейронным сетям. Первое направление – это использование непрерывных векторных представлений (также известных как векторы слов или распределенные представления слов), которые реализованы в библиотеке `word2vec`. Оригинальная статья [\*Distributed Representations of Words and Phrases and Their Compositionality\*](#), написанная Томасом Миколовым с соавторами, является прекрасным введением в тему. Пакеты `spacy` и `gensim` обеспечивают функциональные возможности для реализации методов, рассмотренных в данной статье и ее продолжениях.

Еще одно направление в NLP, которое стало популярным в последние годы – это применение *рекуррентных нейронных сетей* (*recurrent neural networks, RNN*) для обработки текста. Рекуррентные нейронные сети – это особенно мощный вид нейронной сети, который в качестве вывода может снова генерировать текст в отличие от моделей классификации, которые могут лишь назначать метки классов. Способность генерировать текст в качестве вывода позволяет с успехом использовать рекуррентные нейронные сети для автоматического перевода и реферирования. Познакомиться с этой темой можно в исключительно технической статье [\*Sequence to Sequence Learning with Neural Networks\*](#) Ильи Сускевера, Ориоля Виньялса и Куока Ле. Более практичное пособие по использованию фреймворка `tensorflow` можно найти на [веб-сайте TensorFlow](#).



## ГЛАВА 8. ПОДВЕДЕНИЕ ИТОГОВ

Итак, теперь вы умеете применять важнейшие алгоритмы контролируемого и неконтролируемого машинного обучения, которые позволяют решать широкий спектр задач. Прежде чем оставить вас наедине с этой книгой для самостоятельного исследования всех тех возможностей, которые предлагает машинное обучение, мы хотим дать несколько заключительных советов, рассказать о некоторых дополнительных ресурсах и дать рекомендации по поводу того, как можно дополнительно улучшить навыки в области машинного обучения и науки о данных.

### Общий подход к решению задач машинного обучения

Теперь, когда все эти замечательные методы, о которых мы рассказали в этой книге, находятся в вашем распоряжении, возникает соблазн, чтобы сразу, пропустив несколько важных моментов, приступить к решению конкретных задач, просто запустив свой любимый алгоритм. Однако это, как правило, не лучший способ начать анализ. Обычно алгоритм машинного обучения – это лишь небольшая деталь более серьезного процесса анализа данных и принятия решений. Чтобы эффективно использовать машинное обучение, нам нужно сделать шаг назад и рассмотреть задачу в целом. Во-первых, вам стоит подумать о том, на какой вопрос вы хотите ответить. Вы хотите провести разведочный анализ и выяснить, содержат ли данные что-то интересное? Или у вас уже есть конкретная цель? Как правило, вы сначала формулируете цель, например, обнаружение мошеннических транзакций, получение рекомендаций по фильмам или поиск неизвестных планет. Если у вас уже есть такая цель, прежде чем строить систему машинного обучения для ее реализации, вы должны сначала подумать о том, как определить и измерить эффективность достижения цели и какое влияние окажет это эффективное решение на ваши деловые или научные цели. Допустим, ваша цель заключается в том, чтобы обнаружить мошенничество.

Тогда необходимо прояснить следующие вопросы:

- Как определить, что моя система прогнозирования мошенничества на самом деле работает?
- Есть ли у меня подходящие данные для оценки качества алгоритма?
- Если я получил эффективное решение, каким будет его влияние на бизнес?

Как мы уже говорили в главе 5, лучше всего измерять качество алгоритма с использованием бизнес-метрики (увеличение прибыли или снижение убытков). Однако это часто трудно сделать. Вопрос, на который легче будет ответить, звучит так: «Что если я построю

идеальную модель?» Если модель, идеально определяющая любые мошеннические операции, позволит вашей компании экономить 100\$ в месяц, вероятно, этих средств будет недостаточно, чтобы оправдать усилия, направленные на разработку алгоритма. С другой стороны, если модель позволит вашей компании экономить десятки тысяч долларов ежемесячно, задача стоит того, чтобы ее решать.

Допустим, вы определили задачу, которую нужно решить, вы знаете, что решение может иметь значительное влияние для вашего проекта, и вы убедились, что у вас есть необходимая информация, позволяющая измерить эффективность решения. Следующие шаги – это получение данных и построение рабочего прототипа. В этой книге мы рассказывали о различных моделях, которыми вы можете воспользоваться, а также о том, как правильно оценить качество модели и настроить ее параметры. Однако экспериментируя с выбором моделей, помните, что модель – это лишь небольшая деталь большого рабочего потока и построение модели, как правило, является частью обратного цикла, включающего сбор новых данных, очистки данных, построения моделей и анализа моделей. Часто анализ ошибок, допущенных моделью, позволяет получить информацию о том, что пропущено в данных, какие дополнительные данные можно еще собрать или как можно переформулировать задачу, чтобы повысить эффективность машинного обучения. Сбор большего количества данных или данных из других источников, незначительное изменение формулировки задачи, возможно, позволят получить гораздо большую отдачу, чем запуск бесконечных процедур решетчатого поиска для настройки параметров.

## Вмешательство человека в работу модели

Кроме того, вы должны рассмотреть вопрос о том, будут ли люди вмешиваться в работу модели и если да, то как это будет реализовано.<sup>49</sup> Некоторые процессы (например, обнаружение пешеходов беспилотными автомобилями) требуют немедленного принятия решений. Другие процессы, возможно, не требуют немедленной реакции и поэтому человеку предоставляется возможность подтвердить те или иные решения. Например, в медицине, может понадобиться очень высокий

---

<sup>49</sup> Авторы имеют в виду построение модели «человек-в-системе-управления» (human-in-the-loop). Термин пришел из теории искусственного интеллекта и робототехники. Все роботы обладают автономностью, то есть способны осуществлять какие-либо действия без вмешательства человека. Степень автономности роботов определяется заложенной моделью. Условно можно выделить три типа модели: модель «человек-в-системе-управления» (human-in-the-loop), когда окончательное решение принимает только человек, модель «человек-над-системой-управления» (human-on-the-loop), когда решения принимает система, но человек, выполняющий роль наблюдателя, всегда может вмешаться в процесс, и модель «человек-вне-системы-управления» (human-out-of-the-loop), когда система может принимать решения вообще без человеческого вмешательства. – *Прим. пер.*



уровень точности, которого невозможно добиться с помощью лишь одного алгоритма машинного обучения. Но если алгоритм может принять 90%, 50% или, возможно, даже только 10% решений автоматически, то можно увеличить время отклика или снизить издержки. Как правило, большая часть примеров – это «простые случаи», по которым алгоритм может принять решение, а относительно небольшая доля примеров представляет собой «сложные случаи», которые можно перенаправить человеку.

## От прототипа к производству

Инструменты, которые мы обсудили в этой книге? прекрасно подходят для различных проектов машинного обучения и позволяют очень быстро проводить анализ и прототипирование. Кроме того, Python и библиотека **scikit-learn** используется в производственных системах различных организаций, в том числе очень крупных, например, в производственных системах международных банков и глобальных медиакомпаний. Однако многие организации имеют сложную инфраструктуру и интеграция Python в эти системы далеко не всегда является легким процессом. Впрочем, это не является непреодолимой проблемой. Во многих компаниях команды аналитиков используют языки типа Python и R, которые позволяют осуществить быстрое тестирование идей, тогда как команды, занимающиеся внедрением моделей в производственный процесс, работают с такими языками, как Go, Scala, C++, Java, позволяющими строить надежные, масштабируемые системы. Анализ данных и построение надежно работающих сервисов – совершенно разные задачи с разными требованиями, поэтому использование различных языков для решения этих задач имеет смысл. Относительно распространенное решение – реализовать модель, найденную командой аналитиков, в рамках более крупной платформы, используя высокопроизводительный язык. Это проще, чем встраивание целой библиотеки или языка программирования и бесконечные преобразования данных из одного формата в другой.

Вне зависимости от того, будете ли вы использовать **scikit-learn** в производственной системе или нет, важно помнить, что в отличие от скриптов, используемых для выполнения одноразового анализа, к производственным системам предъявляются совершенно другие требования. Если алгоритм встроен в более крупную систему, то актуальными становятся такие аспекты программного обеспечения, как надежность, предсказуемость, время запуска и требования к памяти. Простота – ключевой фактор, позволяющий построить эффективные системы машинного обучения. Внимательно исследуйте каждый этап

вашего конвейера, предназначенного для обработки данных и прогнозирования, и спросите себя, в какой мере каждый этап увеличивает сложность, насколько каждый компонент устойчив к изменениям в данных или вычислительной инфраструктуре, и оправдана ли сложность каждого этапа. Если вы строите сложные системы машинного обучения, мы настоятельно рекомендуем прочитать статью [Machine Learning: The High Interest Credit Card of Technical Debt](#), опубликованную специалистами по машинному обучению компании Google. В статье особое внимание уделено созданию и сопровождению программного обеспечения, которое предназначено для применения машинного обучения в крупномасштабных проектах. Проблема «технического долга»<sup>50</sup> особенно актуальна для крупных и долгосрочных проектов, однако уроки, извлеченные из статьи, помогут вам создать более качественное программное обеспечение даже для краткосрочных и небольших систем.

## Тестирование производственных систем

В этой книге мы рассказали о том, как оценить качество прогнозов, используя заранее подготовленный тестовый набор. Данная процедура известна как *оценка оффлайн (offline evaluation)*. Однако если ваша система машинного обучения взаимодействует с пользователем, это лишь первый этап оценки качества алгоритма. Следующим шагом, как правило, является *онлайн-тестирование (online testing)* или *тестирование в реальных условиях (live testing)*, где оцениваются последствия от применения алгоритма в общей системе. Изменение рекомендаций или результатов поиска, отображаемых веб-сайтом, может коренным образом изменить поведение пользователей и привести к неожиданным последствиям. Для защиты от таких сюрпризов, разработчики интерактивных сервисов используют *A/B тестирование (A/B testing)*, форму слепого тестирования пользователей. В A/B тестировании определенной группе пользователей без их ведома предъявляется веб-сайт или сервис, использующий алгоритм А, а другой группе пользователей предъявляется веб-сайт или сервис, использующий алгоритм В. В обеих группах фиксируются релевантные метрики эффективности за определенный период времени. Затем метрики для алгоритмов А и В сравниваются, выбор оптимального алгоритма

---

<sup>50</sup> Технический долг или долг кодинга (technical debt) — это метафора-неологизм, обозначающая плохую продуманность структуры системы, непродуманную архитектуру программного обеспечения или некачественную разработку ПО. Долг может рассматриваться в виде работы, которую необходимо проделать, пока задача не сможет считаться выполненной. Если долг не погашается, то он будет продолжать увеличиваться, что усложнит дальнейшую разработку. — Прим. пер.

осуществляется согласно этим показателям. Использование А/В тестирования позволяет нам оценить качество алгоритма в реальных условиях и обнаружить различные непредвиденные последствия, когда пользователи взаимодействуют с нашей моделью. Как правило, А – это новая модель, тогда как В – это действующая система. Существуют и более сложные механизмы онлайн-тестирования, выходящие за рамки А/В тестирования, например, *бандитские алгоритмы* (*bandit algorithms*). Замечательным руководством по этой теме является книга издательства О’Reilly [\*Bandit Algorithms for Website Optimization\*](#) за авторством Джона Майлса Уайта.

## Создание своего собственного класса Estimator

Эта книга охватывает множество инструментов и алгоритмов библиотеки **scikit-learn**, которые можно использовать для широкого круга задач. Однако часто вам потребуются какие-то конкретные процедуры предварительной обработки, которые не реализованы в **scikit-learn**. Достаточно лишь предварительно обработать данные перед тем как передать их в модель **scikit-learn** или конвейер. Однако, если ваша предварительная обработка зависит от данных и вы хотите применить решетчатый поиск или перекрестную проверку, все становится сложнее. В главе 6 мы обсуждали важность размещения всех процедур обработки, зависящих от данных, внутри цикла перекрестной проверки. Так как же можно использовать собственные процедуры обработки наряду с инструментами **scikit-learn**? Существует простое решение: построить свою собственную модель! Реализация модели, которая будет совместима с интерфейсом **scikit-learn** (и таким образом ею можно будет воспользоваться с помощью классов **Pipeline**, **GridSearch** и функции **cross\_val\_score**), выглядит довольно просто. Вы можете найти подробные инструкции в [документации по scikit-learn](#), здесь же приводится самая суть. Самый простой способ реализовать класс, выполняющий преобразование – воспользоваться наследованием, то есть на основе базовых классов **BaseEstimator** и **TransformerMixin** создать специализированный класс, а затем создать функции **\_\_init\_\_**, **fit** и **predict**, как показано в нижеприведенном программном коде:

```

In[1]:
from sklearn.base import BaseEstimator, TransformerMixin

class MyTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, first_parameter=1, second_parameter=2):
        # Все параметры должны быть заданы в функции __init__
        self.first_parameter = 1
        self.second_parameter = 2

    def fit(self, X, y=None):
        # fit должна принимать в качестве аргументов только X и y
        # Даже если ваша модель является неконтролируемой, вы должны принять аргумент y!

        # Подгонка модели осуществляется прямо здесь
        print("подгоняем модель прямо здесь")
        # fit возвращает self
        return self

    def transform(self, X):
        # transform принимает в качестве аргумента только X

        # Применяем преобразование к X
        X_transformed = X + 1
        return X_transformed

```

Реализация собственного классификатора или регрессора выглядит аналогично, только вместо `TransformerMixin` вам нужно наследовать от классов `ClassifierMixin` или `RegressorMixin`. Кроме того, вместо `transform` вы можете реализовать `predict`.

Как видно из примера, приведенного здесь, реализация своей собственной модели требует незначительного объема программного кода, и большинство пользователей `scikit-learn` со временем создают целый набор собственных моделей.

## Куда двигаться дальше

Эта книга представляет собой введение в машинное обучение и позволит вам стать эффективным специалистом. Однако если вы хотите совершенствовать навыки машинного обучения, здесь даются некоторые книги и специализированные ресурсы для более глубокого изучения.

## Теория

В этой книге мы попытались дать вам представление о работе наиболее часто используемых алгоритмов машинного обучения, не требуя от вас прочных знаний в области математики или компьютерной науки. Однако многие рассмотренные нами модели используют принципы, взятые из теории вероятностей, линейной алгебры и методов оптимизации. Хотя понимание всех деталей этих алгоритмов не является обязательным, мы считаем, что знание некоторых теорий, лежащих в основе рассмотренных алгоритмов, позволят вам стать хорошим специалистом по анализу данных. По теории машинного обучения написана масса хороших книг и

если бы мы могли заинтересовать вас теми возможностями, которые открывает машинное обучение, мы предложили бы вам выбрать по крайней мере одну из них и уйти в нее с головой. Мы уже упоминали в предисловии книгу Хасты, Тибширани и Фридмана *The Elements of Statistical Learning*, однако стоит повторить эту рекомендацию здесь. Еще одной вполне доступной книгой, к которой прилагаются примеры программного кода Python, является книга Стивена Марсланда *Machine Learning: An Algorithmic Perspective* (издательство Chapman and Hall/CRC). Еще две классические книги, которые настоятельно рекомендуются к прочтению – это книга Кристофера Бишопа *Pattern Recognition and Machine Learning* (издательство Springer), в ней особое внимание уделяется вероятности, и книга Кевина Мерфи *A Probabilistic Perspective* (издательство MIT Press), исчерпывающая научная работа (объемом более 1000 страниц) по методам машинного обучения, которая включает детальное рассмотрение новейших методов и выходит далеко за рамки того, что мы могли охватить в этой книге.

## Другие фреймворки и пакеты машинного обучения

Несмотря на то что `scikit-learn` – это наш любимый пакет для машинного обучения<sup>51</sup>, а Python является нашим любимым языком машинного обучения, существует еще масса пакетов за пределами среды Python. В силу тех или иных задач, стоящих перед вами, вполне возможна ситуация, когда Python и `scikit-learn` не будут являться оптимальным выбором. Как правило, Python отлично подходит для апробации и оценки качества моделей, но крупные веб-сервисы и приложения чаще всего написаны на Java или C++ и для развертывания вашей модели может потребоваться ее интеграция в эти системы. Еще одна причина, по которой вы, возможно, не захотите ограничиваться рамками `scikit-learn` – ситуация, когда вас больше будут интересовать не прогнозы, а возможности статистического моделирования и статистического вывода. В этом случае вы должны обратить внимание на питоновский пакет `statsmodel`, в нем реализовано несколько линейных моделей с интерфейсом, в большей степени ориентированным на статистиков. Если вы не являетесь фанатом Python, вы можете также рассмотреть вопрос об использовании R, еще одном универсальном языке, который используют специалисты по работе с данными. R – язык, разработанный специально для статистического анализа, он славится своими превосходными возможностями визуализации и большим количеством пакетов для статистического моделирования (часто узкоспециализированных).

---

<sup>51</sup> Андреас не может быть полностью объективным в этом вопросе.

Еще популярный пакет для машинного обучения – `vowpal wabbit` (часто называемый `vw`, чтобы не сломать язык), высоко оптимизированный пакет машинного обучения, написанный на C++, с интерфейсом командной строки. `vw` особенно полезен для больших массивов данных и для потоковой передачи данных. Для распределенного запуска алгоритмов машинного обучения на кластере одним из самых популярных решений на момент написания книги была `mllib`, библиотека Scala, реализованная на базе `spark`, среды распределенных вычислений.

## Ранжирование, рекомендательные системы и другие виды обучения

Поскольку данная книга является вводной, мы сосредоточили свое внимание на наиболее распространенных задачах машинного обучения: задачах классификации и регрессии для машинного обучения с учителем, а также задачах кластеризации и декомпозиции сигнала для машинного обучения без учителя. Существует масса других видов машинного обучения с различными задачами, которые не были рассмотрены нами в этой книге. Существует две особенно важные темы, которые мы не охватили в этой книге. Первая тема – это *ранжирование* (*ranking*), когда мы хотим получить ответы на конкретный запрос, упорядоченные по их релевантности. Сегодня вы уже наверняка использовали систему ранжирования, она лежит в основе любой поисковой системы. Вы вводите поисковый запрос и получаете отсортированный список ответов, ранжированных по степени их релевантности. Замечательным вводным пособием, посвященным вопросам ранжирования, является книга Мэннинга, Рагхавана и Шютце *Introduction to Information Retrieval*. Вторая тема – *рекомендательные системы* (*recommender systems*), которые делают предложения пользователям в зависимости от их предпочтений. Вы, наверное, уже сталкивались с рекомендательными системами типа «Люди, которых вы можете знать», «Покупатели, которые купили этот товар, также покупают» или «Лучшие предложения для вас». Существует масса литературы по этой теме, и если вы хотите глубже изучить ее, вас, возможно, заинтересует конкурс [«Netflix Prize challenge»](#). В рамках этого конкурса потоковый видеосервис Netflix выложил на своем сайте большой набор данных, содержащий кинопредпочтения пользователей, и предложил приз 1 млн \$ команде, которая сможет предложить наилучшую рекомендательную систему. Еще одна распространенная задача машинного обучения – это прогнозирование временных рядов (например, цен на акции), по которой также существует целый раздел литературы. Задач машинного обучения



гораздо больше, чем мы можем перечислить здесь, и мы рекомендуем вам обратиться за информацией к книгам, научным статьям и онлайн-сообществам, чтобы найти подходы, которые будут максимально применимы к вашей ситуации.

## Вероятностное моделирование, теория статистического вывода и вероятностное программирование

Большинство пакетов машинного обучения предлагают уже готовые модели машинного обучения на базе какого-то одного конкретного алгоритма. Однако многие реальные задачи имеют определенную структуру и при условии, что эта структура будет правильно представлена в модели, мы можем получить прогнозы гораздо лучшего качества. Часто структуру конкретной задачи можно выразить в терминах теории вероятностей. Получение такой структуры становится возможным благодаря наличию математической модели прогнозируемой ситуации. Для пояснения того, что мы подразумеваем под структурированной задачей, рассмотрим следующий пример.

Допустим, вы хотите создать мобильное приложение, которое позволяет очень точно определить положение в открытом пространстве и тем самым помогает пользователям перемещаться по историческим достопримечательностям. Мобильный телефон снабжен множеством датчиков для точного определения местоположения (GPS, акселерометр, компас и т.д.). У вас также есть точная карта местности. Эта задача является хорошо структурированной. Благодаря карте вы знаете, где проходят интересующие вас маршруты и где находятся интересующие вас объекты. Кроме того, у вас есть грубые оценки местоположения, полученные с помощью GPS, а акселерометр и компас, имеющиеся в телефоне, позволяют вам получить относительно точные измерения. Однако загрузить все эти данные в систему машинного обучения, использующую алгоритм «черного ящика», будет не самой лучшей идеей. Это все равно что выбросить всю фактическую информацию, которую вы уже знали. Если компас и акселерометр говорят вам, что пользователь направляется на север, а GPS сообщает, что пользователь направляется на юг, вам, вероятно, не стоит доверять GPS. Если оценка местоположения говорит вам, что пользователь просто прошел сквозь стену, вы также должны быть настроены весьма скептически. Можно представить эту ситуацию с помощью вероятностной модели, а затем использовать машинное обучение или теорию статистического вывода, чтобы выяснить, в какой степени вам следует доверять каждому измерению и затем строить предположения по поводу того, что представляет собой наиболее точная оценка местоположения.



Как только вы дали правильное описание ситуации и создали корректную вероятностную модель взаимодействия различных факторов, к вашим услугам – специальные методы, которые вычисляют прогнозы, непосредственно используя эти модели. Большинство этих методов реализованы с помощью языков вероятностного программирования, которые позволяют очень элегантно и компактно описать изучаемую задачу. Примерами популярных языков вероятностного программирования являются PyMC (который можно использовать в Python) и Stan (фреймворк, в котором можно использовать несколько языков, в том числе Python). Хотя эти пакеты и требуют определенного понимания теории вероятностей, они значительно упрощают создание новых моделей значительно.

## Нейронные сети

Несмотря на то что мы затронули тему нейронных сетей лишь кратко в главах 2 и 7, данное направление является быстро развивающейся областью машинного обучения, в рамках которой мы каждую неделю узнаем о новых достижениях и новых сферах применения нейронных сетей. Последние успехи в области машинного обучения и искусственного интеллекта, например победа программы Alpha Go над чемпионом по игре Go, постоянное улучшение качества распознавания речи, возможность почти мгновенного перевода речи, полностью обусловлены этими достижениями. Хотя прогресс в этой области движется столь быстрыми темпами, что любая ссылка на передовой метод вскоре устареет, недавно вышедшая книга *Deep Learning* за авторством Яна Гудфеллоу, Йошуа Бенгио и Аарона Курвилля (издательство MIT Press) представляет собой комплексное введение в тему нейронных сетей.<sup>52</sup>

## Масштабирование на больших наборах данных

В этой книге принято, что обрабатываемые данные можно хранить в оперативной памяти в виде массива NumPy или разреженной матрицы SciPy. Даже несмотря на то что современные серверы часто оснащены сотнями гигабайт (ГБ) оперативной памяти, память является фундаментальным ограничением, накладываемым на размер обрабатываемых данных. Не каждый может позволить себе купить такой большой сервер, или даже арендовать у провайдера облачных услуг. Однако в большинстве проектов данные, которые используются для построения системы машинного обучения, имеют относительно небольшой объем. Наборов, состоящих из сотен гигабайтов данных и

---

<sup>52</sup> Препринт книги *Deep Learning* можно посмотреть по адресу <http://www.deeplearningbook.org/>.

более, довольно мало. Данный факт позволяет считать, что во многих случаях увеличение объема памяти или аренда сервера у провайдера облачных услуг является жизнеспособным решением. Однако если вам необходимо обработать терабайт данных или вам нужно обработать большие объемы данных при ограниченном бюджете, существует две базовые стратегии: *out-of-core learning* (обучение во внешней памяти) и *parallelization over a cluster* (распараллеливание на кластере).

Обучение во внешней памяти – это обучение на основе данных, которые не могут быть сохранены в основной памяти, но при этом процесс осуществляется на одном компьютере (при этом может использоваться даже одноядерный процессор компьютера). Данные считываются из источника типа жесткого диска или сети либо по одному примеру за проход, либо в виде блоков, состоящих из нескольких примеров, с тем чтобы поместить каждый блок в оперативную память. Затем этот пример или блок примеров обрабатывается и модель обновляется с учетом информации, вычисленной по этим данным.<sup>53</sup> Затем этот блок данных удаляется и считывается следующий поднабор данных. В библиотеке **scikit-learn** обучение во внешней памяти реализовано для некоторых моделей, и вы можете найти подробную информацию о нем в руководстве пользователя. Поскольку обучение во внешней памяти подразумевает обработку всех данных на одном компьютере, то в случае больших наборов данных выполнение подобных вычислений займет много времени. Кроме того, не для всех алгоритмов можно осуществить обучение во внешней памяти.

Еще одна стратегия масштабирования – это распределение данных по нескольким машинам вычислительного кластера, когда каждая машина обрабатывает свою часть данных. Для некоторых моделей эта стратегия может дать гораздо более существенное ускорение и размер обрабатываемых данных ограничен лишь размером кластера. Однако подобные вычисления часто требуют относительно сложной инфраструктуры. На данный момент одной из наиболее популярных платформ распределенных вычислений является платформа **spark**, входящая в экосистему **Hadoop**. В рамках пакета **MLlib** для платформы **spark** реализованы некоторые алгоритмы машинного обучения. Если ваши данные уже записаны в файловой системе **HDFS** или вы уже используете **spark** для предварительной обработки данных, описываемый способ вычислений может стать наиболее простым вариантом. Однако если у вас еще нет такой инфраструктуры, установка и интеграция

---

<sup>53</sup> Например, при оценивании параметров конкретного линейного метода (весов в логистической регрессии) будет инициализировано некоторое начальное значение этих параметров, после чего получая на вход очередной пример или блок примеров из обучающей выборки, веса будут обновляться. – *Прим. пер.*

кластера `spark` могут потребовать очень больших усилий. Ранее упомянутый пакет `vm` предлагает ряд возможностей для осуществления распределенных вычислений и, возможно, будет лучшим решением в данном случае.

## Оттачивание навыков

Как и во многих аспектах жизни, только практика позволит вам стать экспертом в вопросах, рассмотренных нами в этой книге. Процедуры выделения признаков, предварительной обработки, визуализации и построения модели могут сильно варьировать в зависимости от различных задач и разных наборов данных. Возможно, в вашем распоряжении уже имеются разные наборы данных с различными задачами. Если вы еще не решаете конкретную задачу, хорошей стартовой площадкой станут конкурсы по машинному обучению, в рамках которых публикуются данные с конкретной задачей и команды соревнуются в получении наилучших прогнозов. Многие компании, некоммерческие организации и университеты проводят такие соревнования. Одна из самых популярных площадок, на которой можно найти эти конкурсы - это [Kaggle](#), веб-сайт, который регулярно проводит соревнования по анализу данных, некоторые из них имеют солидный призовой фонд.

Кроме того, форум сайта Kaggle – это хороший источник информации о новейших инструментах и техниках машинного обучения, на сайте также можно найти различные наборы данных. Еще больше наборов данных с соответствующими задачами можно найти на [платформе OpenML](#), на которой размещено более 20000 наборов данных с более чем 50000 задачами машинного обучения. Работа с этими наборами данных открывает замечательные возможности для оттачивания навыков в области машинного обучения. Недостаток соревнований заключается в том, что решение должно удовлетворять конкретной и заранее заданной метрике оптимизации, и, как правило, данные представляют собой фиксированный предварительно обработанный набор данных. Имейте в виду, постановка задачи и сбор данных также являются важными аспектами, и правильное понимание задачи, возможно, гораздо важнее, чем выжимание максимального процента правильности из классификатора.

## Заключение

Мы надеемся, что убедили вас в полезности и легкости применения машинного обучения для решения самых разнообразных задач.

Продолжайте исследовать данные, но не упускайте из виду картину в целом.

## Об авторах

**Андреас Мюллер** получил ученую степень PhD по машинному обучению в Боннском университете. Занимал в течение года должность специалиста по машинному обучению в Amazon, где решал прикладные задачи в области компьютерного зрения. В настоящий момент Андреас работает в Центре изучения данных Нью-Йоркского университета. В течение последних четырех лет Андреас стал куратором и одним из ключевых разработчиков библиотеки `scikit-learn`, инструмента машинного обучения, широко используемого в промышленности и науке. Кроме того, Андреас является автором и разработчиком еще нескольких популярных пакетов машинного обучения. Свою миссию он видит в том, чтобы создавать инструменты с открытым программным кодом, которые убирают препятствия, мешающие более активному использованию машинного обучения в прикладных задачах, содействуют продвижению воспроизводимой науки<sup>54</sup> и упрощают применение высокоточных алгоритмов машинного обучения.

**Сара Гвидо** – специалист по анализу данных, имеет большой опыт работы в стартапах. Ее сфера интересов – язык Python, машинное обучение, большие объемы данных и мир технологий. Совсем недавно Сара стала ведущим специалистом по анализу данных в компании Bitly, является постоянным спикером конференций по машинному обучению. Кроме того, Сара имеет степень магистра по информатике Мичиганского университета и в настоящее время проживает в Нью-Йорке.

## Колофон

Животное, изображенное на обложке книги *Введение в машинное обучение с помощью Python* – аллеганский скрытожаберник (лат. *Cryptobranchus alleganiensis*), амфибия, обитающая в восточной части США (ареал обитания простирается от Нью-Йорка до Джорджии). Это земноводное имеет множество колоритных прозвищ, в их числе «аллеганский аллигатор», «сопли выдры» и «грязь дьявола». Происхождение официального англоязычного названия «hellbender salamander» неясно: согласно одной из теорий, первые поселенцы посчитали внешний вид саламандры отталкивающим и предположили, что это существо из ада, в который оно стремится вернуться.

---

<sup>54</sup> Воспроизводимая наука (reproducible science) – научная деятельность, в процессе которой открытия осуществляются по заранее известному и отработанному «маршруту», как это, например, происходит в структурной геномике при открытии новых молекул. – *Прим. пер.*

Аллеганский скрытожаберник является представителем семейства гигантских саламандр и может достигать 73 сантиметров в длину. Это третий по величине вид водных саламандр в мире. Саламандра имеет уплощенное тело с толстыми складками кожи по бокам. Хотя саламандры и имеют по одной жаберной щели с каждой стороны, большую часть кислорода они поглощают через складки кожи: газ проникает и выделяется через капилляры, расположенные на поверхности кожи.

В силу этого их идеальная среда обитания – это чистые, быстрые и неглубокие ручьи, где вода хорошо насыщена кислородом. Скрытожаберник прячется под камнями и охотится главным образом с помощью обоняния, хотя способен ощущать малейшие колебания в воде. Его рацион состоит из речных раков, мелких рыб. Иногда скрытожаберник поедает яйца саламандр своего вида. Скрытожаберник является ключевым участником своей экосистемы, будучи добычей для крупной рыбы, черепах и змей.

В течение последних нескольких десятилетий популяция аллеганского скрытожаберника значительно снизилась. Качество воды стало самой большой проблемой, поскольку дыхательная система скрытожаберников делает их очень чувствительными к загрязненной или мутной воде. Активизация человеческой деятельности вблизи мест обитания скрытожаберников означает увеличение количества химических осадков в воде. Пытаясь сохранить этот исчезающий вид, биологи стали разводить амфибий в неволе и выпускать их в естественную среду обитания по достижении ими менее уязвимого возраста.

Многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, все они имеют важное значение для окружающего мира. Чтобы подробнее узнать о том, как вы можете помочь, зайдите на сайт [animals.oreilly.com](http://animals.oreilly.com).

Изображение для обложки взято из книги *Wood's Animate Creation*. Шрифты обложки – URW Typewriter and Guardian Sans. Шрифт текста – Adobe Minion Pro, шрифт заголовков – Adobe Myriad Condensed, шрифт программного кода – Dalton Maag's Ubuntu Mono.

# Обратная сторона обложки

## Введение в машинное обучение с помощью Python

Машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, однако эта область не является прерогативой больших компаний с мощными аналитическими командами. Даже если вы еще новичок в использовании Python, эта книга познакомит вас с практическими способами построения систем машинного обучения. При всем многообразии данных, доступных на сегодняшний день, применение машинного обучения ограничивается лишь вашим воображением.

Вы изучите этапы, необходимые для создания успешного проекта машинного обучения, используя Python и библиотеку `scikit-learn`. Авторы Андреас Мюллер и Сара Гвидо сосредоточили свое внимание на практических аспектах применения алгоритмов машинного обучения. Знание библиотек `NumPy` и `matplotlib` позволит вам извлечь из этой книги еще больше полезной информации.

С помощью этой книги вы изучите:

- Фундаментальные понятия и сферы применения машинного обучения
- Преимущества и недостатки широко используемых алгоритмов машинного обучения
- Способы загрузки данных, обрабатываемых в ходе машинного обучения, включая различные аспекты работы с данными
- Продвинутое методы оценивания модели и тонкая настройка параметров
- Принципы построения конвейеров для объединения моделей в цепочки и инкапсуляции рабочего потока
- Методы работы с текстовыми данными
- Рекомендации по улучшению навыков, связанных с машинным обучением и наукой о данных

*Эта книга – фантастический, суперпрактический ресурс для каждого, кто хочет начать использовать машинное обучение в Python – как жаль, что когда я начинала использовать `scikit-learn`, этой книги не было.*

Ханна Уоллок,  
старший научный сотрудник Microsoft Research

**Андреас Мюллер** получил ученую степень PhD по машинному обучению в Боннском университете. Занимал должность специалиста по машинному обучению в Amazon, где занимался разработкой проектов компьютерного зрения. В настоящий момент Андреас работает в Центре изучения данных Нью-Йоркского университета. Кроме того, Андреас – куратор и один из ключевых разработчиков библиотеки `scikit-learn`.

**Сара Гвидо** – специалист по анализу данных, имеет большой опыт работы в стартапах, совсем недавно стала ведущим специалистом по анализу данных в компании Bitly, постоянный спикер конференций по машинному обучению. Кроме того, Сара имеет степень магистра по информатике Мичиганского университета.