Цветное издание

amazon
Bestseller

ДМК
издательство

Себастьян Рашка

Python и машинное обучение

Себастьян Рашка

Python и машинное обучение

Sebastian Raschka

Python Machine Learning

Unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics

[PACKT] PUBLISHING
BIRMINGHAM – MUMBAI

open source
community experience distilled

Себастьян Рашка

Python и машинное обучение

Крайне необходимое издание по новейшей предсказательной аналитике для более глубокого понимания методологии машинного обучения

ДМК Пресс
Издательство
Москва, 2017

УДК 004.438Python:004.6
ББК 32.973.22
Р28

Рашка С.
Р28 Python и машинное обучение / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2017. – 418 с.: ил.

ISBN 978-5-97060-409-0

Книга предоставит вам доступ в мир прогнозной аналитики и продемонстрирует, почему Python является одним из лидирующих языков науки о данных. Охватывая широкий круг мощных библиотек Python, в том числе scikit-learn, Theano и Keras, предлагая руководство и советы по всем вопросам, начиная с анализа мнений и заканчивая нейронными сетями, книга ответит на большинство ваших вопросов по машинному обучению.

Издание предназначено для специалистов по анализу данных, находящихся в поисках более широкого и практического понимания принципов машинного обучения.

УДК 004.438Python:004.6
ББК 32.973.22

ISBN 978-1-78355-513-0 (анг.)
ISBN 978-5-97060-409-0 (рус.)

Содержание

Предисловие

Мы живем в потоке данных. Согласно недавним оценкам, ежедневно генерируется 2,5 квинтилиона (1018) байт данных. Это такой огромный объем данных, что более 90% информации, которую мы храним в наши дни, было сгенерировано за все прошедшее десятилетие. К сожалению, люди не способны воспользоваться подавляющей частью этой информации. Данные либо лежат за пределами возможностей стандартных аналитических методов, либо они просто слишком обширны, чтобы наши ограниченные умы смогли их понять.

Благодаря методам машинного обучения мы наделяем компьютеры способностями обрабатывать большие объемы данных, которые в противном случае стояли бы непроницаемой стеной, даем им возможность обучаться на этих данных и извлекать из них практические выводы. От массивных суперкомпьютеров, которые обеспечивают работу поисковых движков компании Google, до смартфонов, которые мы носим в наших карманах, – везде мы опираемся на машинное обучение, которое приводит в действие значительную часть окружающего нас мира, и нередко мы об этом даже не догадываемся.

Являясь первооткрывателями наших дней дивного нового мира больших данных, нам надлежит узнать еще больше о машинном обучении. Что же такое машинное обучение и как оно работает? Каким образом его применять, чтобы заглянуть в неизведанное, привести в действие свой бизнес либо просто узнать, что Интернет в целом думает о моем любимом фильме? Все это и даже больше будет охвачено в следующих главах, созданных моим добрым другом и коллегой Себастьяном Рашка.

Когда Себастьян не занят одомашниванием моей вспыльчивой собаки, он неустанно посвящает свое свободное время сообществу специалистов, работающих с открытым исходным кодом в области машинного обучения. В течение последних нескольких лет Себастьян разработал десятки популярных учебных руководств, в которых затрагиваются различные темы из области машинного обучения и визуализации данных на Python. Он также является автором и соавтором разработок ряда библиотек Python с открытым исходным кодом, и некоторые из них теперь являются составной частью базового потока операций по машинному обучению на Python.

В силу его обширных экспертных познаний в этой области я уверен, что глубокое понимание Себастьяном мира машинного обучения на Python будет неоценимо для пользователей всех уровней квалификации. Я искренне рекомендую эту книгу любому, кто находится в поисках более широкого и более практического понимания принципов машинного обучения.

Доктор Рандал С. Олсон,
исследователь в области искусственного интеллекта
и машинного обучения, Университет шт. Пенсильвания, США

Об авторе

Себастьян Рашка – аспирант докторантуры в Мичиганском университете, США, занимающийся разработкой новых вычислительных методов в области вычислительной биологии. Веб-сайтом Analytics Vidhya (https://www.analyticsvidhya.com/) сообщества увлеченных профессионалов в области науки о данных отмечен первым местом среди наиболее влиятельных аналитиков данных на GitHub. За его плечами многолетний опыт программирования на Python; он также проводит ряд семинаров по практическому применению науки о данных и машинного обучения. Регулярные выступления и публикации на тему науки о данных, машинного обучения и языка Python на деле мотивировали его написать эту книгу, с тем чтобы помочь людям разрабатывать управляемые данными решения без обязательного наличия предварительной квалификации в области машинного обучения.

Он также является активным соавтором проектов с открытым исходным кодом и автором собственных методов, которые теперь успешно применяются в конкурсах по машинному обучению, таких как Kaggle. В свое свободное время он работает над моделями для спортивного прогнозирования, и если не сидит перед компьютером, то любит проводить время, занимаясь спортом.

Хотел бы поблагодарить своих профессоров, Арун Росса и Панг-Нинг Тана, а также многих других, кто вдохновил меня и сформировал у меня огромный интерес к исследованиям в области классификации образов, машинного обучения и добычи данных.

Хотел бы воспользоваться представившейся возможностью и поблагодарить огромное сообщество пользователей и разработчиков библиотек Python с открытым исходным кодом, которые помогли мне создать совершенную среду для научных исследований и науки о данных.

Особую благодарность передаю базовым разработчикам библиотеки scikit-learn. В качестве одного из соавторов этого проекта было приятно работать вместе с замечательными людьми, которые не только превосходно осведомлены в области машинного обучения, но и являются превосходными программистами.

Наконец, хочу поблагодарить всех за проявление интереса к этой книге и искренне надеюсь, что смогу передать весь свой энтузиазм по поводу моего присоединения к огромным сообществам программистов на Python и в области машинного обучения.

О рецензентах

Ричард Даттон начал заниматься программированием компьютера ZX Spectrum в возрасте 8 лет, и с тех пор это увлечение направляет его по противоречивому массиву технологий и ролей в области промышленности и финансов.

Работал в Microsoft и управляющим в Barclays, его текущим увлечением является гибрид из Python, машинного обучения и цепочки блоков транзакций.

Если он не сидит перед компьютером, то его можно найти в спортзале либо дома с бокалом вина перед смартфоном iPhone. Он называет это равновесием.

Дэйв Джулиан – ИТ-консультант и преподаватель с 15-летним стажем. Работал техником, проектным инженером, программистом и веб-разработчиком. Его текущие проекты состоят из разработки инструмента для анализа урожайности в составе интегрированных стратегий по борьбе с сельскохозяйственными вредителями в теплицах. Его большой интерес пролегает на пересечении биологии и технологии с уверенностью, что умные машины способны помочь решить самые важные глобальные задачи.

Вахид Мирджалили получил звание доктора наук Мичиганского университета по машиностроению, где он разработал новые методы рафинирования белковых структур с использованием молекулярно-динамического имитационного моделирования. Объединив свои знания из областей статистики, добычи данных и физики, разработал мощные управляемые данными подходы, которые помогли ему и его исследовательской группе одержать победу в двух недавних мировых конкурсах по прогнозированию и рафинированию протеиновых структур, CASP, в 2012 и 2014 гг.

Работая над докторской диссертацией, решил присоединиться к факультету информатики и инженерного дела в Мичиганском университете с целью специализации в области машинного обучения. Его текущие исследовательские проекты включают разработку алгоритмов машинного обучения без учителя для добычи массивных наборов данных. Он также является страстным поклонником программирования на Python и делится своими реализациями алгоритмов кластеризации на своем личном веб-сайте http://vahidmirjalili.com.

Хамидреза Саттари – ИТ-профессионал, участвовавший в ряде областей, связанных с разработкой программного обеспечения, от программирования до архитектуры и управления. Владеет степенью магистра в области разработки программного обеспечения Университета Хериота-Уатта, Соединенное Королевство, и степенью бакалавра по электротехнике (электронике) Тегеранского университета Азад, Иран. В последние годы его области интереса составляли большие данные и машинное обучение. Является соавтором книги «*Веб-службы Spring 2. Книга рецептов*» (Spring Web Services 2 Cookbook); ведет свой собственный блог по адресу http://justdeveloped-blog.blogspot.com/.

Дмитрий Тарановский – разработчик программного обеспечения с заинтересованностью и квалификацией в Python, Linux и машинном обучении. Родом из Киева, Украина, он переехал в США в 1996 г. С раннего возраста страстно увлекался наукой и знаниями, побеждая на конкурсах по физике и математике. В 1999 г. был избран членом команды США по физике. В 2005 г. окончил Мичиганский Технологический институт со специализацией по математике. Позже работал программным

инженером над системой трансформации текста для компьютерных медицинских транскрипций (eScription). Изначально работая на Perl, он по достоинству оценил мощь и ясность Python, который позволил ему масштабировать систему до данных больших объемов. Впоследствии работал инженером программного обеспечения и аналитиком на алгоритмическую трейдинговую фирму. Он также внес значительный вклад в математические основы, в том числе создание и совершенствование расширения языка теории множеств и его связи с аксиомами множеств большой мощности, разработав понятийный аппарат конструктивной истины и создав систему порядковой индексации с реализацией на Python. Он также любит читать, быть за городом и старается сделать мир лучше.

Введение

Наверное, не стоит и говорить, что машинное обучение стало одной из самых захватывающих технологий современности. Такие крупные компании, как Google, Facebook, Apple, Amazon, IBM, и еще многие другие небезосновательно вкладывают значительный капитал в разработку методов и программных приложений в области машинного обучения. Хотя может показаться, что термин «машинное обучение» сегодня уже набил оскомину, совершенно очевидно, что весь этот ажиотаж не является результатом рекламной шумихи. Эта захватывающая область исследования открывает путь к новым возможностям и стала неотъемлемой частью нашей повседневной жизни. Разговоры с речевым ассистентом по смартфону, предоставление рекомендаций относительно подходящего продукта для клиентов, предотвращение актов мошенничества с кредитными картами, фильтрация спама из входящих сообщений электронной почты, обнаружение и диагностирование внутренних заболеваний – и этот список можно продолжать.

Если вы хотите стать практиком в области машинного обучения, более основательным решателем задач или, возможно, даже обдумываете карьеру в научно-исследовательской области, связанной с машинным обучением, то эта книга для вас! Однако новичка теоретические идеи, лежащие в основании машинного обучения, нередко могут подавлять своей сложностью. И все же многие из опубликованных в последние годы практических изданий способны помочь вам приступить к работе с машинным обучением на основе реализации мощных алгоритмов обучения. По моему мнению, использование практических примеров программного кода служит важной цели. В них идеи иллюстрируются путем приведения изученного материала непосредственно в действие. Однако помните, что огромная мощь влечет за собой большую ответственность! Идеи, лежащие в основании машинного обучения, слишком красивы и важны, чтобы их прятать в черном ящике. Поэтому моя личная миссия состоит в том, чтобы предоставить вам иную книгу: книгу, в которой обсуждаются важные подробности относительно идей и принципов машинного обучения, предлагаются интуитивные и одновременно информативные объяснения по поводу того, каким образом алгоритмы машинного обучения работают, как их использовать и, самое главное, как избежать наиболее распространенных ловушек.

Если в поисковой системе Академия Google[1] в качестве поискового запроса набрать «машинное обучение», то она вернет обескураживающе большое количество публикаций. В английском сегменте их число составляет 1 800 000 публикаций (для сравнения, в русском – 16 400). Разумеется, мы не сможем обсудить мельчайшие подробности всех основных алгоритмов и приложений, которые появились в течение предыдущих 60 лет. Однако в этой книге мы предпримем увлекательное путешествие, которое затронет все существенные темы и понятия, чтобы дать вам преимущество в этой области. Если вы считаете, что ваша потребность в знаниях из области машинного обучения не удовлетворена, то можно воспользоваться много-

[1] Академия Google (Google Scholar) (https://scholar.google.ru/schhp?hl=ru) – бесплатная поисковая система по полным текстам научных публикаций всех форматов и дисциплин. Проект работает с ноября 2004 г. – *Прим. перев.*

численными полезными ресурсами, чтобы отслеживать существенные прорывы в этой области.

Если вы уже подробно изучили теорию машинного обучения, то эта книга покажет вам, каким образом претворить ваши знания на практике. Если вы использовали методы машинного обучения прежде и хотите получить более глубокое понимание того, каким образом машинное обучение работает в действительности, то эта книга для вас! Не переживайте, если вы абсолютно плохо знакомы с машинным обучением; у вас гораздо больше причин испытывать предвкушение. Я обещаю вам, что машинное обучение изменит ваш способ мышления относительно задач, которые вы хотите решать, и покажу вам, как справляться с ними путем высвобождения мощи данных.

Прежде чем мы погрузимся с головой в область машинного обучения, отвечу на ваш сакраментальный вопрос – «почему, собственно, Python?» Ответ прост: он – мощный и одновременно очень доступный. Python стал самым популярным языком программирования для науки о данных, потому что он позволяет забыть об утомительных сторонах программирования и предлагает нам среду, где мы можем быстро набросать наши мысли и привести идеи непосредственно в действие.

Размышляя над тем путем, который я прошел лично, могу сказать вам совершенно искренне, что изучение методов машинного обучения сделало меня более основательным ученым, мыслителем и решателем задач. В этой книге я хочу поделиться этими знаниями с вами. Знание достигается в процессе исследования, ключом к нему является наш энтузиазм, а истинное освоение навыков может быть достигнуто только практикой. Местами продвижение может быть ухабистым, и некоторые темы могут оказаться более сложными для понимания, чем другие, но я надеюсь, что вы воспользуетесь этой возможностью и сконцентрируетесь на вознаграждении. Помните, что мы отправляемся в это путешествие вместе, и по ходу изложения мы будем пополнять ваш арсенал большим количеством мощных методов, которые помогут нам решать даже самые трудноразрешимые задачи на основе управляемого данными подхода.

О чем эта книга рассказывает

Глава 1 «Наделение компьютеров способностью обучаться на данных» знакомит с основными подобластями машинного обучения, в которых решаются самые разные практические задачи. Кроме того, в ней обсуждаются принципиальные шаги, которые необходимо предпринять, чтобы создать типичную машиннообучаемую модель, и создается конвейер, который будет направлять нас в последующих главах.

Глава 2 «Тренировка алгоритмов машинного обучения для задачи классификации» обращается к истокам области исследований, связанной с машинным обучением, и знакомит с бинарными (двуклассовыми) классификаторами на основе персептрона и адаптивных линейных нейронов. Эта глава является осторожным введением в основополагающие принципы классификации образов, где основное внимание уделено взаимодействию машинного обучения с алгоритмами оптимизации.

Глава 3 «Обзор классификаторов с использованием библиотеки scikit-learn» описывает принципиальные алгоритмы машинного обучения, предназначенные для зада-

чи классификации, и предлагает практические примеры с использованием одной из самых популярных и всеобъемлющих библиотек машинного обучения с открытым исходным кодом scikit-learn.

Глава 4 «Создание хороших тренировочных наборов – предобработка данных» посвящена обсуждению того, как обходиться с наиболее распространенными трудностями, возникающими в работе с исходными наборами данных, такими как пропущенные данные. В ней также обсуждается ряд подходов к идентификации в наборах данных наиболее информативных признаков и будут продемонстрированы способы подготовки переменных различных типов для ввода в алгоритмы машинного обучения.

Глава 5 «Сжатие данных путем снижения размерности» посвящена описанию принципиальных методов, необходимых для сведения числа признаков в наборе данных к наборам меньшего объема при сохранении большей части их полезной и отличительной информации. В ней обсуждается стандартный подход к снижению размерности на основе главных компонент, который сравнивается с методами нелинейного преобразования с учителем.

Глава 6 «Изучение наиболее успешных методов оценки моделей и тонкой настройки гиперпараметров» посвящена обсуждению плюсов и минусов методик оценки качества прогнозных моделей. Кроме того, в ней обсуждаются различные метрики, применяемые для измерения качества моделей, и приемы тонкой настройки алгоритмов машинного обучения.

Глава 7 «Объединение моделей для методов ансамблевого обучения» знакомит с различными принципами эффективного объединения двух и более алгоритмов обучения. В ней будет продемонстрировано создание ансамблей экспертов с целью преодоления недостатков отдельных алгоритмов обучения, которые в результате приводят к более точным и надежным прогнозам.

Глава 8 «Применение алгоритмов машинного обучения в анализе мнений» посвящена обсуждению принципиальных шагов, необходимых для преобразования текстовых данных в содержательные представления для алгоритмов машинного обучения, с целью прогнозирования мнений людей на основе их письменной речи.

Глава 9 «Встраивание алгоритма машинного обучения в веб-приложение» продолжает прогнозную модель из предыдущей главы и проведет вас по основным шагам разработки веб-приложений со встроенными машиннообучаемыми моделями.

Глава 10 «Прогнозирование непрерывных целевых величин на основе регрессионного анализа» посвящена обсуждению принципиальных методов, необходимых для моделирования линейных связей между целевыми переменными и переменной отклика для создания прогнозов на непрерывной шкале. После ознакомления с различными линейными моделями в ней также обсуждаются подходы на основе параболической регрессии и на основе деревьев.

Глава 11 «Работа с немаркированными данными – кластерный анализ» смещает акцент в другую подобласть машинного обучения – обучение без учителя. Мы применим алгоритмы из трех фундаментальных семейств алгоритмов кластеризации с целью нахождения групп объектов, в которых присутствует определенная степень подобия.

Глава 12 «Тренировка искусственных нейронных сетей для распознавания изображений» расширяет принцип градиентной оптимизации, который был впервые представлен в *главе 2 «Тренировка алгоритмов машинного обучения для задачи*

классификации» для создания мощных, многослойных нейронных сетей на основе популярного алгоритма обратного распространения ошибки.

Глава 13 «Распараллеливание тренировки нейронных сетей при помощи Theano» опирается на знания, полученные в предыдущих главах, и предоставляет практическое руководство по более эффективной тренировке нейронных сетей. В центре внимания данной главы находится библиотека Python с открытым исходным кодом Theano, которая предоставит возможность использовать ядра современных многоядерных графических процессоров.

Что требуется для этой книги

Исполнение прилагаемых к данной книге примеров программ требует установки Python версии 3.4.3 или более поздней в Mac OS X, Linux или Microsoft Windows. На протяжении всей книги мы часто будем использовать ключевые библиотеки Python для научных вычислений, в том числе SciPy, NumPy, scikit-learn, matplotlib и pandas.

В первой главе будут предложены инструкции и полезные подсказки по поводу инсталляции среды Python и указанных ключевых библиотек. Мы пополним наш репертуар дополнительными библиотеками, а инструкции по их инсталляции будут предоставлены в соответствующих главах: библиотека NLTK для обработки естественного языка (*глава 8 «Применение алгоритмов машинного обучения в анализе мнений»*), веб-платформа Flask (*глава 9 «Встраивание алгоритма машинного обучения в веб-приложение»*), библиотека seaborn для визуализации статистических данных (*глава 10 «Прогнозирование непрерывных целевых величин на основе регрессионного анализа»*) и Theano для эффективной тренировки нейронных сетей на графических процессорах (*глава 13 «Распараллеливание тренировки нейронной сети при помощи Theano»*).

Для кого эта книга

Если вы хотите узнать, как использовать Python, чтобы начать отвечать на критические вопросы в отношении ваших данных, возьмите книгу «*Python и машинное обучение*» – и не важно, хотите вы приступить к изучению науки о данных с нуля или же намереваетесь расширить свои познания в этой области, – это принципиальный ресурс, которого нельзя упускать.

Условные обозначения

В этой книге вы найдете ряд текстовых стилей, которые выделяют различные виды информации. Вот некоторые примеры этих стилей и объяснение их значения.

Кодовые слова в тексте, имена таблиц баз данных, папок, файлов, расширения файлов, пути, ввод данных пользователем и дескрипторы социальной сети Twitter показаны следующим образом: «И уже установленные пакеты могут быть обновлены при помощи флага `--upgrade`».

Фрагмент исходного кода оформляется следующим образом:

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

При этом если исходный код содержит результат выполнения, то он предваряется значком консоли интерпретатора Python (>>>):

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

Любой ввод или вывод в командной строке записывается следующим образом:

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

Новые термины и **важные слова** показаны полужирным шрифтом. Слова, которые вы видите на экране, например в меню или диалоговых окнах, выглядят в тексте следующим образом: «После щелчка по кнопке **Dashboard** в верхнем правом углу мы получим доступ к панели управления, показанной в верхней части страницы».

Предупреждения или важные примечания появляются в этом поле.

Подсказки и приемы появляются тут.

Дополнения к тексту оригинала книги.

Отзывы читателей

Отзывы наших читателей всегда приветствуются. Сообщите нам, что вы думаете об этой книге – что вам понравилось, а что нет. Обратная связь с читателями для нас очень важна, поскольку она помогает нам формировать названия книг, из которых вы действительно получите максимум полезного.

Отзыв по общим вопросам можно отправить по адресу feedback@packtpub.com, упомянув заголовок книги в теме вашего электронного сообщения.

Если же речь идет о теме, в которой вы профессионально осведомлены, и вы интересуетесь написанием либо содействием в написании книги, то обратитесь к нашему перечню авторов по адресу www.packtpub.com/authors.

Служба поддержки

Теперь, когда вы являетесь довольным владельцем книги издательства «Packt», мы предложим вам ряд возможностей с целью помочь вам получать максимум от своей покупки.

Скачивание исходного кода примеров

Вы можете скачать файлы с исходным кодом примеров из вашего аккаунта по адресу http://www.packtpub.com/ для всех книг издательства «Packt Publishing», которые вы приобрели. Если вы купили эту книгу в другом месте, то можно посетить http://www.packtpub.com/support и зарегистрироваться там, чтобы получить файлы прямо по электронной почте.

Опечатки

Несмотря на то что мы приняли все меры, чтобы гарантировать правильность нашего информационного материала, ошибки действительно случаются. Если вы найдете ошибку в одной из наших книг, возможно, в тексте или исходном коде, то мы будем благодарны, если вы сообщите об этом нам. Поступая так, вы можете уберечь других читателей от разочарования и помочь нам улучшать последующие версии этой книги. Если вы найдете какие-либо опечатки, пожалуйста, сообщите о них, посетив страницу http://www.packtpub.com/submit-errata, выбрав вашу книгу, нажав на ссылку в форме **Errata submission form** для отправки информации об опечатке и введя ваши данные об опечатках. Как только ваши данные будут проверены, сообщение будет принято к рассмотрению, и данные об опечатке будут загружены на наш веб-сайт или добавлены к любому списку из существующих списков опечаток в разделе **Errata** под соответствующим заголовком.

Чтобы просмотреть ранее предоставленные опечатки, перейдите на страницу https://www.packtpub.com/books/content/support и в поле поиска введите название книги. Запрошенная информация появится под разделом **Errata**.

Нарушение авторских прав

Пиратство защищенного авторским правом материала в Интернете является хронической проблемой во всех средствах массовой информации. В издательстве «Packt» мы очень серьезно относимся к защите нашего авторского права и лицензий. Если вы сталкиваетесь с какими-либо недопустимыми копиями наших работ в какой-либо форме в Интернете, пожалуйста, просим вас незамедлительно предоставить нам адрес размещения или название веб-сайта, благодаря чему мы можем добиваться правовой защиты.

Пожалуйста, свяжитесь с нами по электронному адресу copyright@packtpub.com с ссылкой на предполагаемый пиратский материал.

Мы ценим вашу помощь в защите наших авторов и в наших усилиях предоставлять вам ценный информационный материал.

Вопросы

Если у вас есть вопрос по каким-либо аспектам этой книги, то вы можете связаться с нами по электронному адресу questions@packtpub.com, и мы приложим все усилия, чтобы решить ваш вопрос.

Комментарий переводчика

Весь материал книги приведен в соответствие с последними действующими версиями библиотек (время перевода книги – октябрь-ноябрь 2016 г.), дополнен свежей информацией и протестирован в среде Windows 10 и Fedora 24. При тестировании исходного кода за основу взят Python версии 3.5.2.

Большинство содержащихся в книге технических терминов и аббревиатур для удобства кратко определено в сносках, а для некоторых терминов в силу отсутствия единой терминологии приведены соответствующие варианты наименований или пояснения. Книга дополнена «*Глоссарием основных терминов и сокращений*». В приложении к книге особое внимание уделено оценке качества машиннообучаемых моделей.

На веб-сайте Github имеется веб-страница книги, где содержатся обновляемые исходные коды прилагаемых к книге программ, много дополнительных материалов и ссылок, а также обширный раздел часто задаваемых вопросов (https://github.com/rasbt/python-machine-learning-book/tree/master/faq), наиболее интересная, по моему мнению, и лишь незначительная часть ответов на которые сведена в *приложении А* к данной книге.

Прилагаемый к книге адаптированный и скорректированный исходный код примеров должен находиться в подпапке домашней папки пользователя (/home/ваши_проекты_Python или C:\Users\[ИМЯ_ПОЛЬЗОВАТЕЛЯ]\ ваши_проекты_Python). Ниже приведена структура папки с прилагаемыми примерами и дополнительной информацией:

bonus	Дополнительные материалы, документация и другие примеры.
ch01-ch13	Исходный код примеров в виде блокнотов Jupyter и сценарных файлов Python.
data	Наборы данных, используемых в книге и примерах.
faq	Копия раздела часто задаваемых вопросов по книге и машинному обучению репозитория Github на английском языке.
fonts	Шрифты, используемые для оформления блокнотов, книги и графиков.

Для просмотра исходного кода примеров лучше всего пользоваться блокнотами Jupyter. Они более читабельны, содержат графики, цветные рисунки и расширенные пояснения.

Для оформления графиков и диаграмм использовался шрифт Ubuntu Condensed, который прилагается к книге. Его можно найти в папке fonts. В Windows для установки шрифта в операционную сиситему следует правой кнопкой мыши нажать на файле шрифта и выбрать из контекстного меню **Установить**.

Далее приведены особенности установки некоторых используемых программных библиотек Python.

Особенности программных библиотек

Numpy+MKL привязана к библиотеке Intel® Math Kernel Library и включает в свой состав необходимые динамические библиотеки (DLL) в каталоге numpy.core. Для работы SciPy и Scikit-learn в Windows требуется, чтобы в системе была установлена библиотека Numpy+MKL. Ее следует скачать из репозитория whl-файлов (http://www.lfd.uci.edu/~gohlke/pythonlibs/) и установить (например, pip3 install numpy-1.11.2+mkl-cp35-cp35m-win_amd64.whl для 64-разрядного компьютера) как whl.

- **NumPy** – основополагающая библиотека, необходимая для научных вычислений на Python.
- **Matplotlib** – библиотека для работы с двумерными графиками. Требует наличия numpy и некоторых других.
- **Pandas** – инструмент для анализа структурных данных и временных рядов. Требует наличия numpy и некоторых других.
- **Scikit-learn** – интегратор классических алгоритмов машинного обучения. Требует наличия numpy+mkl.
- **SciPy** – библиотека, используемая в математике, естественных науках и инженерном деле. Требует наличия numpy+mkl.
- **Jupyter** – интерактивная вычислительная среда.

Факультативно:

- **Spyder** – инструментальная среда программирования на Python.
- **Mlxtend** – библиотека модулей расширения и вспомогательных инструментов для программных библиотек Python, предназначенных для анализа данных и машинного обучения (автор С. Рашка) для решения каждодневных задач в области науки о данных. Используется в приложении и дополнительных материалах к книге (https://github.com/rasbt/mlxtend). Документацию по библиотеке можно найти в папке bonus.
- **PyQt5**, библиотека программ для программирования визуального интерфейса, требуется для работы инструментальной среды программирования Spyder

Протокол установки программных библиотек

```
python -m pip install --upgrade pip

pip3 install numpy
    либо как whl: pip3 install numpy-1.11.2+mkl-cp35-cp35m-win_amd64.whl
pip3 install matplotlib
pip3 install pandas
pip3 install scikit-learn
    либо как whl: pip3 install scikit_learn-0.18.1-cp35-cp35m-win_amd64.whl
pip3 install scipy
```

```
    либо как whl: pip3 install scipy-0.18.1-cp35-cp35m-win_amd64-any.whl
pip3 install jupyter
pip3 install theano
pip3 install keras
pip3 install nltk
pip3 install seaborn
pip3 install flask
pip3 install pyprind
pip3 install wtforms
    факультативно:
pip3 install mlxtend
pip3 install pyqt5
pip3 install spyder
```

Примечание: в зависимости от базовой ОС, версий языка Python и версий программных библиотек устанавливаемые вами версии whl-файлов могут отличаться от приведенных выше, где показаны последние на декабрь 2016 г. версии для 64-разрядной ОС Windows и Python версии 3.5.2

Установка библиотек Python из whl-файла

Библиотеки для Python можно разрабатывать не только на чистом Python. Довольно часто библиотеки пишутся на C (динамические библиотеки), и для них пишется обертка Python, или же библиотека пишется на Python, но для оптимизации узких мест часть кода пишется на C. Такие библиотеки получаются очень быстрыми, однако библиотеки с вкраплениями кода на C программисту на Python тяжелее установить ввиду банального отсутствия соответствующих знаний либо необходимых компонентов и настроек в рабочей среде (в особенности в Windows). Для решения описанных проблем разработан специальный формат (файлы с расширением .whl) для распространения библиотек, который содержит заранее скомпилированную версию библиотеки со всеми ее зависимостями. Формат whl поддерживается всеми основными платформами (Mac OS X, Linux, Windows).

Установка производится с помощью менеджера пакетов pip. В отличие от обычной установки командой pip3 install <имя_библиотеки>, вместо имени библиотеки указывается путь к whl-файлу pip3 install <путь/к/whl_файлу>. Например:

```
pip3 install C:\temp\networkx-1.11-py2.py3-none-any.whl
```

Откройте окно командой строки и при помощи команды cd перейдите в каталог, где размещен ваш whl-файл. Просто скопируйте туда имя вашего whl-файла. В этом случае полный путь указывать не понадобится. Например:

```
pip3 install networkx-1.11-py2.py3-none-any.whl
```

При выборе библиотеки важно, чтобы разрядность устанавливаемой библиотеки и разрядность интерпретатора совпадали. Пользователи Windows могут брать whl-файлы на веб-странице http://www.lfd.uci.edu/~gohlke/pythonlibs/ Кристофа Голька из Лаборатории динамики флуоресценции Калифорнийского университета в г. Ирвайн. Библиотеки там постоянно обновляются, и в архиве содержатся все, какие только могут понадобиться.

Установка и настройка инструментальной среды Spyder

Spyder – это инструментальная среда для научных вычислений для языка Python (Scientific PYthon Development EnviRonment) для Windows, Mac OS X и Linux. Это простая, легковесная и бесплатная интерактивная среда разработки на Python, которая предлагает функционал, аналогичный среде разработки на MATLAB, включая готовые к использованию виджеты PyQt5 и PySide: редактор исходного кода, редактор массивов данных NumPy, редактор словарей, консоли Python и IPython и многое другое.

Чтобы установить среду Spyder в Ubuntu Linux, используя официальный менеджер пакетов, нужна всего одна команда:

```
sudo apt-get install spyder3
```

Чтобы установить с использованием менеджера пакетов pip:

```
sudo apt-get install python-qt5 python-sphinx
sudo pip3 install spyder
```

И чтобы обновить:

```
sudo pip3 install -U spyder
```

Установка среды Spyder в Fedora 24:

```
dnf install python3-spyder
```

Во всех вышеперечисленных случаях речь идет о версии Spyder для Python 3 (на момент инсталляции это был Python 3.5.2).

Установка среды Spyder в Windows:

```
pip3 install spyder
```

Глава 1

Наделение компьютеров способностью обучаться на данных

По моему убеждению, *машинное обучение* как область практической деятельности и как сфера научных исследований алгоритмов, извлекающих смысл из данных, является самой захватывающей отраслью всей информатики! В наш век переизбытка данных при помощи самообучающихся алгоритмов мы можем превратить эти данные в знание. Благодаря тому что за последние несколько лет были разработаны многочисленные мощные библиотеки с открытым исходным кодом, вероятно, никогда еще не появлялся столь подходящий момент для того, чтобы ворваться в область машинного обучения и научиться использовать мощные алгоритмы, позволяющие обнаруживать в данных повторяющиеся образы и делать прогнозы о будущих событиях.

В этой главе мы узнаем об общих принципах машинного обучения и его типах. Вместе с вводным курсом в соответствующую терминологию мы заложим основу для того, чтобы успешно использовать методы машинного обучения для решения практических задач.

В этой главе мы затронем следующие темы:

- общие принципы машинного обучения;
- три типа обучения и основополагающая терминология;
- базовые элементы успешной разработки систем машинного обучения;
- инсталляция и настройка Python для анализа данных и машинного обучения.

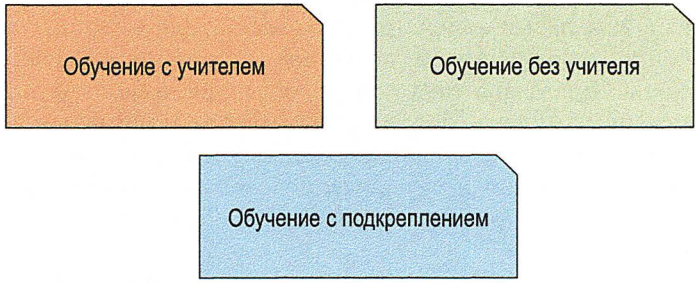
Построение интеллектуальных машин для преобразования данных в знания

В эпоху современных технологий существует один ресурс, которым мы обладаем в изобилии: большой объем структурированных и неструктурированных данных. Во второй половине XX века машинное обучение развилось в подобласть *искусственного интеллекта*, которая охватывала разработку самообучающихся алгоритмов для приобретения из этих данных знаний с целью выполнения прогнозов. Вместо того чтобы в ручном режиме выявлять правила и строить модели на основе анализа больших объемов данных, методология машинного обучения предлагает для вычленения знаний из данных более действенную альтернативу – постепенное улучшение качества прогнозных моделей и принятие решений, управляемых данными. Помимо того что машинное обучение приобретает все большую значимость в научных исследованиях в области информатики, оно начинает играть все более активную роль в нашей повседневной жизни. Благодаря машинному обучению мы наслаждаемся

удобным программным обеспечением распознавания текста и речи, устойчивыми почтовыми спам-фильтрами, надежными поисковыми системами, амбициозными шахматными программами и, надо надеяться, совсем скоро начнем использовать безопасные и эффективные самоходные автомобили.

Три типа машинного обучения

В этом разделе мы рассмотрим три типа машинного обучения: *обучение с учителем* (контролируемое), *обучение без учителя* (неконтролируемое, или спонтанное) и *обучение с подкреплением*:

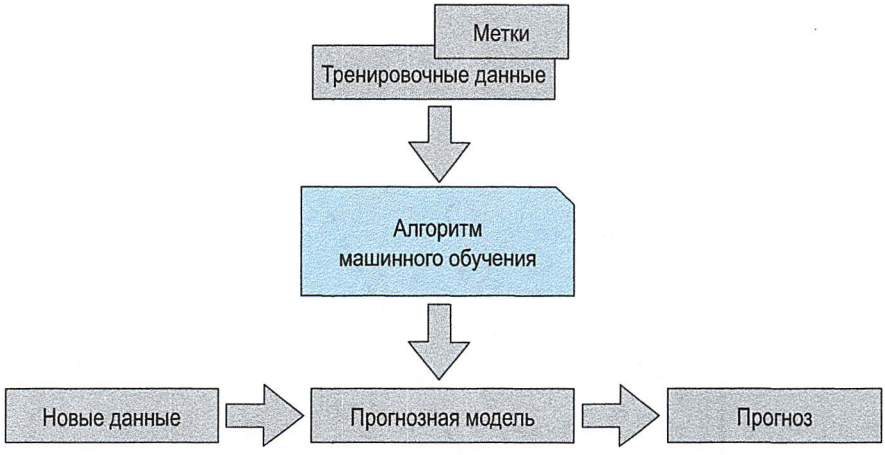


Мы узнаем о принципиальных между ними различиях и, опираясь на концептуальные примеры, разовьем интуитивное понимание того, в каких областях они находят свое практическое применение.

Выполнение прогнозов о будущем на основе обучения с учителем

Основная задача обучения с учителем состоит в том, чтобы на маркированных *тренировочных данных* извлечь модель, которая позволяет делать прогнозы о ранее не встречавшихся или будущих данных. Здесь термин «*с учителем*» относится к подмножеству образцов, в которых нужные выходные сигналы (метки) уже известны.

В качестве примера рассмотрим фильтрацию почтового спама. При помощи алгоритма машинного обучения с учителем мы можем натренировать модель на корпусе маркированных сообщений электронной почты, где почтовые сообщения правильно помечены как спамные либо неспамные, и затем предсказать, к какому из этих двух категорий принадлежит новое сообщение. Методы обучения с учителем, когда имеются дискретные *метки принадлежности к классу*, такие как в приведенном примере с фильтрацией спамных почтовых сообщений, еще называют методами *классификации*. Другую подкатегорию методов обучения с учителем представляет *регрессия*, где результирующий сигнал – непрерывная величина:



Задача классификации – распознавание меток классов

Задача классификации – это подкатегория методов машинного обучения с учителем, суть которой заключается в идентификации категориальных меток классов для новых экземпляров на основе предыдущих наблюдений[1]. Метка класса представляет собой дискретное, неупорядоченное значение, которое может пониматься как *принадлежность группе* экземпляров. Ранее упомянутый пример с обнаружением почтового спама представляет собой типичный пример задачи *бинарной классификации*, где алгоритм машинного обучения вычисляет серию правил для различения двух возможных классов: спамных и неспамных почтовых сообщений.

Однако набор меток классов не обязательно должен иметь двоичную природу. Извлеченная алгоритмом обучения с учителем прогнозная модель может присваивать новому, немаркированному экземпляру любую метку класса, которая была определена в тренировочном наборе данных. Типичным примером задачи *многоклассовой (или мультиномиальной) классификации* является рукописное распознавание символов. Здесь можно было бы собрать тренировочный набор данных, состоящий из большого числа рукописных образцов каждой буквы алфавита. Теперь если пользователь через устройство ввода данных предоставит новый рукописный символ, то наша прогнозная модель с определенной степенью соответствия сможет распознать правильную букву алфавита. Однако наша система машинного обучения была бы не в состоянии правильно распознать любую из цифр от нуля до девяти, в случае если они не входили в состав нашего тренировочного набора данных.

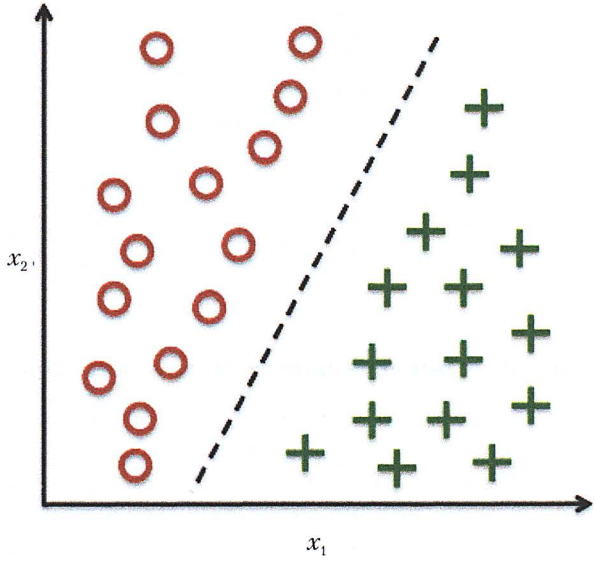
Следующий ниже рисунок иллюстрирует принцип работы задачи бинарной классификации при наличии 30 тренировочных образцов[2]: 15 образцов маркированы как *отрицательный класс* (круги) и другие 15 – как *положительный класс* (знаки «плюс»). В этом сценарии наш набор данных является двумерным, то есть каждый образец имеет два связанных с ним значения: x_1 и x_2. Теперь мы можем применить

---

[1] Класс – это множество всех объектов с данным значением метки. – *Прим. перев.*

[2] Под образцом, или экземпляром (sample), здесь и далее по тексту подразумевается совокупность признаков, которым поставлены в соответствие конкретные значения. – *Прим. перев.*

алгоритм машинного обучения с учителем для извлечения правила – граница решения представлена черной пунктирной линией, – которое может выделить эти два класса и затем распределить новые данные в каждую из этих двух категорий при наличии значений x_1 и x_2:



Задача регрессии – предсказание значений непрерывной целевой переменной

В предыдущем разделе мы узнали, что задача классификации заключается в назначении экземплярам категориальных, неупорядоченных меток. Второй тип обучения с учителем представлен предсказанием непрерывных результатов, который еще именуется регрессионным анализом, или методами восстановления зависимости между переменными. В *регрессионном анализе* нам даны несколько *предикторных* (объясняющих) переменных и непрерывная (результирующая) переменная отклика, и мы пытаемся найти между этими переменными связь, которая позволит нам предсказывать результат.

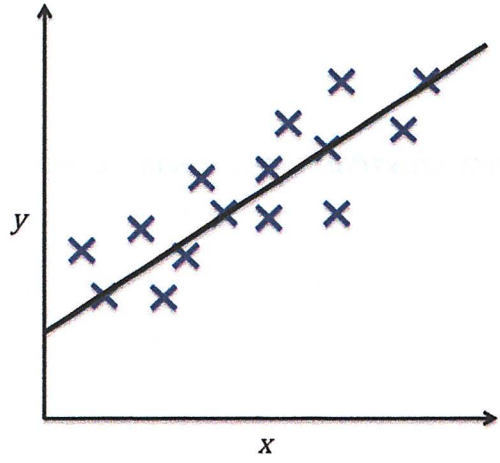
Например, предположим, что нас интересует предсказание оценок студентов за тест по математике SAT Math[1]. Если между затраченным на подготовку к тесту временем и итоговыми оценками существует связь, то мы могли бы воспользоваться ею в качестве тренировочных данных для извлечения модели, в которой время учебы используется для предсказания экзаменационных отметок будущих студентов, планирующих пройти этот тест.

Термин *регрессия* был введен Фрэнсисом Гальтоном в своей статье *«Регрессия к посредственности по наследственному статусу»* (Regression Towards Mediocrity in He-

[1] SAT Math – тест по математике для будущих бакалавров (http://test-sat.ru/sat-math/). – *Прим. перев.*

reditary Stature) в 1886 г. Гальтон описал биологическое явление, заключающееся в том, что дисперсия в росте среди представителей популяции со временем не увеличивается. Он обнаружил, что рост родителей не передается их детям, а рост детей регрессирует к среднеарифметическому росту по популяции.

Следующий ниже рисунок иллюстрирует основную идею *линейной регрессии*. При наличии предикторной переменной x и переменной отклика y мы выполняем под эти данные подгонку прямой, которая минимизирует расстояние – обычно среднеквадратичное – между точками образцоы и подогнанной линией. Далее мы можем воспользоваться полученными из этих данных пересечения оси[1] и наклоном прямой с неким угловым коэффициентом для прогнозирования результирующей переменной в новых данных:



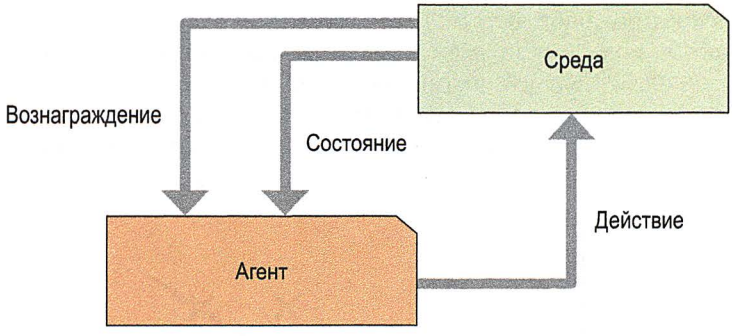
Решение интерактивных задач на основе обучения с подкреплением

Еще одним типом машинного обучения является обучение с подкреплением. Задача обучения с подкреплением состоит в выработке системы (агента), которая улучшает свое качество на основе взаимодействий со *средой*. Поскольку информация о текущем состоянии среды, как правило, содержит и так называемый сигнал *вознаграждения*, обучение с подкреплением можно представить как область, имеющую отношение к обучению с учителем. Однако в обучении с подкреплением эта обратная связь является не меткой или значением, раз и навсегда определенными в результате прямых наблюдений, а мерой того, насколько хорошо действие было оценено функцией *вознаграждения*. В ходе взаимодействия со средой агент на основе разведочного подхода путем проб и ошибок или совещательного планирования может использовать обучение с подкреплением для вычленения серии действий, которые максимизируют это вознаграждение.

---

[1] Геометрически пересечение (intercept) – это точка, в которой линия регрессии пересекает ось *Y*. В формуле линейной регрессии пересечение – это свободный коэффициент, которому равна зависимая переменная, если предиктор, или независимая переменная, равен нулю. – *Прим. перев.*

Популярным примером обучения с подкреплением является шахматный движок. Здесь агент выбирает серию ходов в зависимости от состояния шахматной доски (среды), где вознаграждение можно задать как *выигрыш* либо *проигрыш* в конце игры:



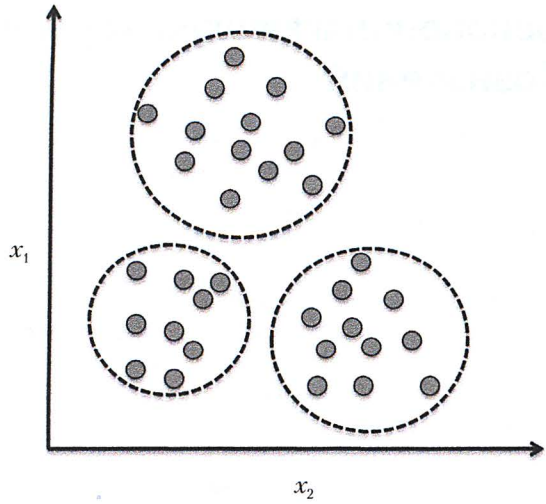
Обнаружение скрытых структур при помощи обучения без учителя

В обучении с учителем, когда мы тренируем нашу модель, мы знаем *правильный ответ* заранее, а в обучении с подкреплением мы определяем меру *вознаграждения* за выполненные агентом отдельно взятые действия. С другой стороны, в обучении без учителя мы имеем дело с немаркированными данными или данными с *неизвестной структурой*. Используя методы обучения без учителя, мы можем разведать структуру данных с целью выделения содержательной информации без контроля со стороны известной результирующей переменной или функции вознаграждения.

Выявление подгрупп при помощи кластеризации

Кластеризация – это метод разведочного анализа данных, который позволяет организовать груду информации в содержательные подгруппы (*кластеры*), не имея никаких предварительных сведений о принадлежности группе. Каждый кластер, который может появиться во время анализа, обозначает группу объектов, которые обладают определенной степенью подобия и одновременно больше отличаются от объектов в других кластерах, поэтому кластеризацию также иногда называют «классификацией без учителя». Кластеризация незаменима для структурирования информации и получения содержательных связей внутри данных. Например, этот метод позволяет специалистам по маркетингу обнаруживать группы потребителей на основе их интересов с целью разработки конкретных маркетинговых программ.

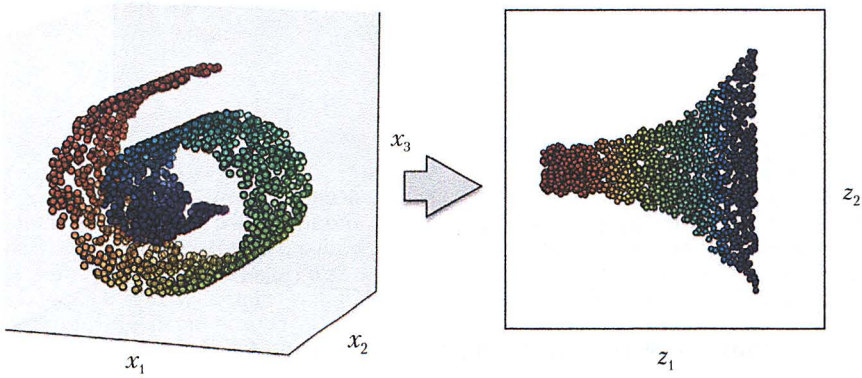
Приведенный ниже рисунок иллюстрирует применение кластеризации для размещения немаркированных данных в три разные группы на основе подобия их признаков x_1 и x_2:



Снижение размерности для сжатия данных

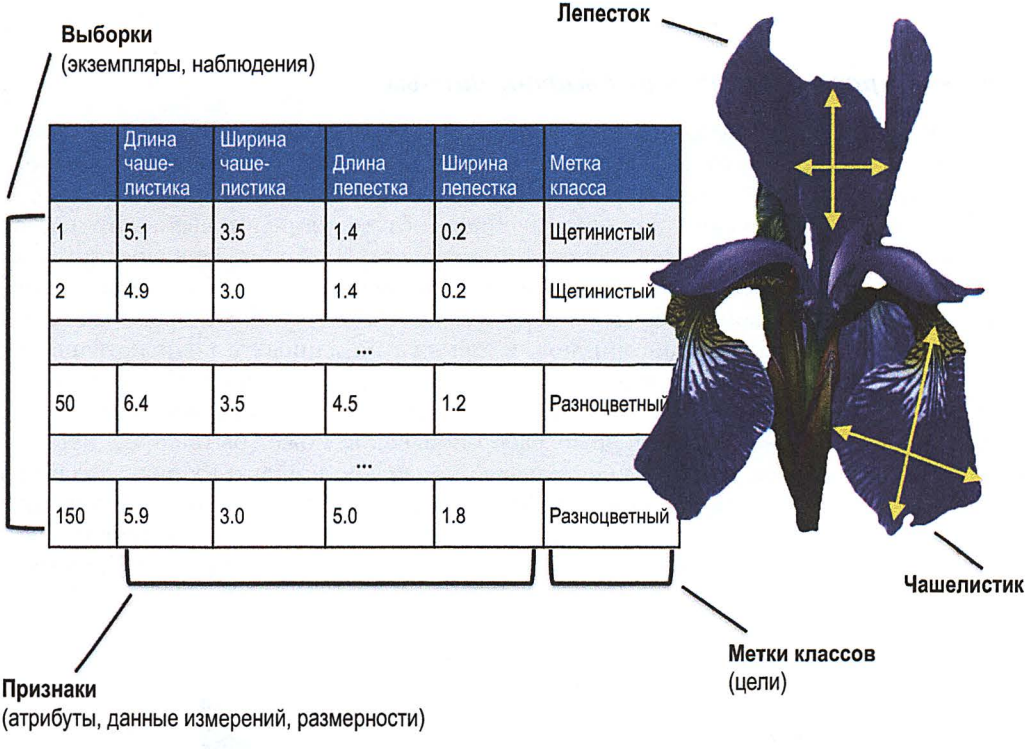
Еще одна подобласть обучения без учителя представлена методом *снижения размерности*. Часто приходится работать с данными высокой размерности – каждое наблюдение состоит из большого числа результатов измерений, которые могут представлять серьезную проблему для ограниченного объема памяти и вычислительной производительности алгоритмов машинного обучения. Подход на основе снижения размерности без учителя широко используется во время предобработки признаков с целью удаления из данных шума, который тоже может ухудшить предсказательную способность некоторых алгоритмов, и для сжатия данных в подпространство меньшей размерности при сохранении большей части релевантной информации.

Кроме того, иногда снижение размерности может быть полезно для визуализации данных – например, высокоразмерный набор признаков может быть спроецирован на одно-, двух- или трехмерные пространства признаков с целью их визуализации при помощи трех- или двухмерных точечных графиков или гистограмм. Рисунок ниже показывает пример использования нелинейного снижения размерности для сжатия трехмерного *швейцарского рулета* (Swiss roll) в новое двумерное подпространство признаков:



Введение в основополагающую терминологию и систему обозначений

Обсудив три обширные категории машинного обучения – с учителем, без учителя и с подкреплением, теперь обратимся к базовой терминологии, которую мы будем использовать в последующих главах. Следующая ниже таблица изображает выдержку из набора данных *Ирисы Фишера*[1], который представляет собой классический пример из области машинного обучения. Этот набор данных содержит данные измерений 150 цветков ириса трех видов: *ирис щетинистый (Iris setosa), ирис виргинский (Iris virginica) и ирис разноцветный (Iris versicolor)* по четырем характеристикам, или признакам. Здесь каждый образец цветков представляет в нашем наборе данных одну строку, а результаты измерений цветков в сантиметрах хранятся в столбцах, называемых также признаками набора данных:



	Длина чаше-листика	Ширина чаше-листика	Длина лепестка	Ширина лепестка	Метка класса
1	5.1	3.5	1.4	0.2	Щетинистый
2	4.9	3.0	1.4	0.2	Щетинистый
...					
50	6.4	3.5	4.5	1.2	Разноцветный
...					
150	5.9	3.0	5.0	1.8	Разноцветный

[1] Ирисы Фишера (Iris) – набор данных для задачи классификации, на примере которого Рональд Фишер в 1936 году продемонстрировал разработанный им метод дискриминантного анализа. Для каждого экземпляра измерялись четыре характеристики/признака (в сантиметрах): длина чашелистика (sepal length), ширина чашелистика (sepal width); длина лепестка (petal length), ширина лепестка (petal width). Этот набор данных уже стал классическим и часто используется в литературе для иллюстрации работы различных статистических алгоритмов (см. https://ru.wikipedia.org/wiki/Ирисы_Фишера). – *Прим. перев.*

Чтобы система обозначений и реализация оставались простыми и одновременно эффективными, мы привлечем несколько базовых элементов *линейной алгебры*. В следующих главах для обозначения наших данных мы воспользуемся *матричной* и *векторной* системами обозначений. Мы будем следовать общепринятому соглашению представлять каждый образец как отдельную строку в матрице признаков $\boldsymbol{X}$, где каждый признак хранится как отдельный столбец.

Тогда набор данных цветков ириса, состоящий из 150 образцов и 4 признаков, можно записать как матрицу размера 150×4, где $X \in \mathbb{R}^{150\times 4}$:

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

В остальной части этой книги мы будем использовать надстрочный индекс (i) для обозначения i-й тренировочного образца и подстрочный индекс j для обозначения j-й размерности тренировочного набора данных.

Мы используем заглавные (прописные) буквы, набранные жирным шрифтом, для обозначения векторов ($x \in \mathbb{R}^{n\times 1}$) и строчные буквы, набранные жирным шрифтом, для обозначения матриц соответственно ($X \in \mathbb{R}^{n\times m}$). Для обозначения в векторе или матрице отдельных элементов мы пишем буквы курсивом, соответственно $x^{(n)}$ или $\boldsymbol{x}_{(m)}^{(n)}$.

Например, x_1^{150} обозначает первое измерение 150-го образца цветков, *длину чашелистика*. Таким образом, каждая строка в этой матрице признаков представляет один экземпляр цветка и может быть записана как четырехмерный вектор-строка $x^{(i)} \in \mathbb{R}^{1\times 4}$, $x^{(i)} = [x_1^{(i)}\ x_2^{(i)}\ x_3^{(i)}\ x_4^{(i)}]$.

Каждое измерение в признаках – это 150-мерный вектор-столбец $x_j \in \mathbb{R}^{150\times 1}$. Например:

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

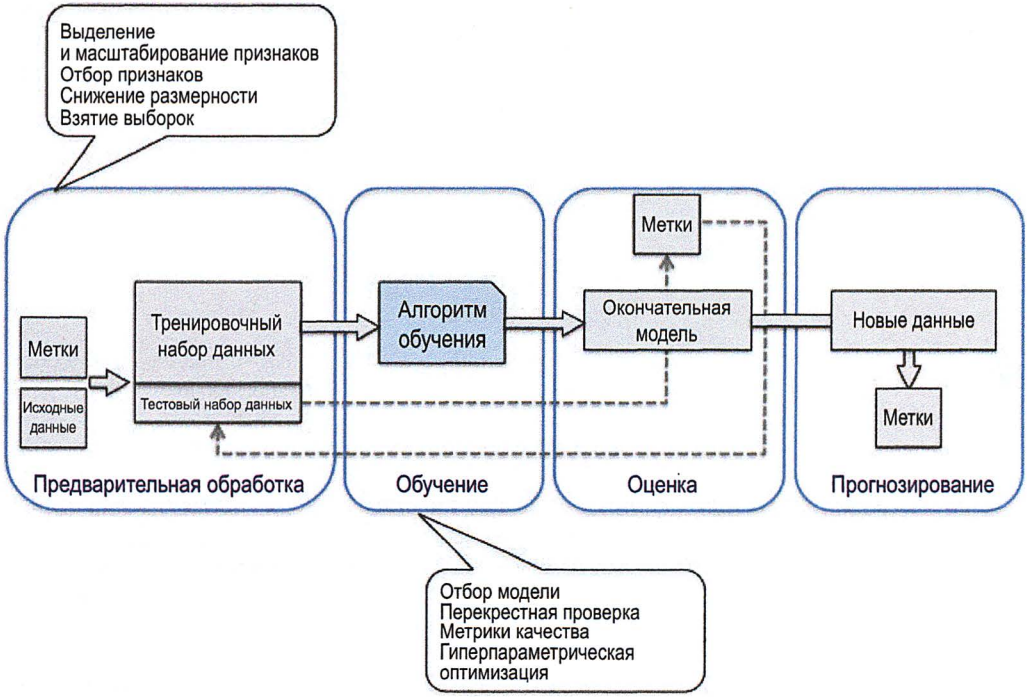
Целевые переменные (в данном случае метки классов) мы храним аналогичным образом, как 150-мерный вектор-столбец:

$$y = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix} (y \in \{\text{Вергинский, Разноцветный, Щетинистый}\}).$$

Дорожная карта для построения систем машинного обучения

В предыдущих разделах мы обсудили основные принципы машинного обучения и три разных типа обучения. В этом разделе мы затронем другие важные составляющие системы машинного обучения, сопровождающие алгоритм обучения. Приве-

денная ниже схема показывает типичную диаграмму потока операций при использовании машинного обучения в *прогнозном моделировании*, которое мы обсудим в последующих подразделах:



Предобработка – приведение данных в приемлемый вид

Исходные (необработанные) данные редко поступают в том виде и в той форме, которые необходимы для оптимальной работы алгоритма обучения. Поэтому *предобработка* данных является одним из самых важных этапов в любом приложении с использованием машинного обучения. Если в качестве примера взять набор данных цветков ириса из предыдущего раздела, то исходные данные можно представить как серию снимков цветков, из которых мы хотим выделить содержательные признаки. Полезными признаками могут быть цвет, оттенок, интенсивность цвета цветков, их высота, длина и ширина. Многие алгоритмы машинного обучения также требуют, чтобы отобранные признаки в целях оптимального качества работы находились в одной и той же шкале, что часто достигается путем приведения признаков к диапазону [0, 1] или стандартному нормальному распределению с нулевым средним значением и единичной дисперсией, как мы убедимся в более поздних главах.

Некоторые отобранные признаки могут высоко коррелироваться и поэтому быть до известной степени избыточными. В этих случаях целесообразно использовать методы снижения размерности в целях сжатия признаков в подпространство более низкой размерности. Преимущество снижения размерности нашего пространства признаков заключается в том, что требуется меньший объем памяти, в результате чего алгоритм обучения может выполняться намного быстрее.

Чтобы подтвердить, что наш алгоритм машинного обучения не только хорошо работает на тренировочном наборе, но и хорошо обобщается на новые данные, нам также понадобится случайным образом подразделить набор данных на два отдельных подмножества: тренировочный и тестовый наборы. Мы используем тренировочный набор, чтобы натренировать и оптимизировать нашу машиннообучаемую модель, в то время как тестовый набор мы храним до самого конца с целью выполнить оценку окончательной модели.

Тренировка и отбор прогнозной модели

Как мы убедимся в последующих главах, для решения различных практических задач было разработано большое количество разных алгоритмов машинного обучения. Из известной статьи Дэвида Вольперта «*Теоремы "Никаких бесплатных обедов"*» (David Wolpert, No Free Lunch Theorems) можно сделать важный вывод[1] – мы не можем осуществлять машинное обучение «бесплатно» («*Отсутствие априорных различий между алгоритмами обучения*», D. H. Wolpert 1996; «*Теоремы No Free Lunch (NFL-теоремы) для оптимизации*», D. H. Wolpert и W. G. Macready, 1997). Интуитивно эту концепцию можно увязать с популярным высказыванием Абрахама Маслоу «*Предполагаю, что когда у вас в руках всего один инструмент— молоток, то заманчиво рассматривать все вокруг как гвозди*» (1966)[2]. Например, каждый алгоритм классификации имеет присущие им смещения, и никакая модель классификации не обладает превосходством перед другими, при условии что мы не делаем допущений в отношении задачи. Поэтому на практике важно сравнить, по крайней мере, несколько различных алгоритмов, чтобы натренировать и отобрать самую качественную модель. Но прежде чем мы сможем сопоставить различные модели, мы сначала должны принять решение относительно метрики для измерения их качества. Одна из самых распространенных метрик – это верность (accuracy), иногда именуемая правильностью, которая определяется как доля правильно классифицированных экземпляров от общего числа предсказаний[3].

Здесь возникает законный вопрос: *а как узнать, какая модель работает хорошо на окончательном тестовом наборе данных и реальных данных, если мы не используем этот тестовый набор для отбора модели и держим его для окончательной оценки модели?* Для решения заложенной в этот вопрос проблемы можно воспользоваться различными методами перекрестной проверки, или кросс-валидации, где тренировочный набор данных далее подразделяется на тренировочное и *провероч-*

---

[1] Д. Вульперт (1996) показал, что попытки найти алгоритм обучения без смещений бесполезны. Кроме того, он продемонстрировал, что в свободном от шума сценарии, где функция потерь эквивалентна коэффициенту ошибочной классификации, если вы интересуетесь ошибкой вне тренировочного набора, то между алгоритмами обучения нет никаких априорных различий. См. http://www.no-free-lunch.org/. – *Прим. перев.*

[2] Иными словами, когда у вас в руках молоток, все задачи кажутся гвоздями. – *Прим. перев.*

[3] В машинном обучении точность (precision) является мерой статистической изменчивости и описывает случайные ошибки. Верность (accuracy) является мерой статистического смещения и описывает систематические ошибки. Резюмируя, верность модели (алгоритма) – это близость результатов измерений к истинному значению и в силу этого часто переводится как «правильность» и «адекватность»; точность результатов измерений (precision) – это их повторяемость, репродуцируемость. – *Прим. перев.*

ное подмножества для оценки *обобщающей способности* модели[1]. Наконец, мы не можем ожидать, что параметры различных алгоритмов обучения, предоставляемых программными библиотеками по умолчанию, оптимальны для нашей конкретной практической задачи. Поэтому в более поздних главах мы часто будем использовать методы *гиперпараметрической оптимизации*, которые помогут выполнить тонкую настройку качества нашей модели. Интуитивно гиперпараметры можно представить как параметры, которые не были извлечены из данных и являются рычагами управления моделью, которые можно изменять с целью улучшения ее качества работы. Их роль станет намного яснее в более поздних главах, когда мы увидим действующие примеры.

Оценка моделей и прогнозирование на ранее не встречавшихся экземплярах данных

После того как мы отобрали модель, подогнанную под тренировочный набор данных, мы можем воспользоваться тестовым набором данных и оценить, насколько хорошо она работает на этих ранее не встречавшихся ей данных, чтобы вычислить ошибку обобщения. Если мы удовлетворены качеством модели, то теперь мы можем использовать ее для прогнозирования новых будущих данных. Важно отметить, что параметры для таких ранее упомянутых процедур, как масштабирование признаков и снижение размерности, получают исключительно из тренировочного набора данных, и те же самые параметры позже применяют повторно для трансформирования тестового набора данных, а также любых новых образцов данных – в противном случае измеренное на тестовых данных качество модели может оказаться сверхоптимистичной.

Использование Python для машинного обучения

Python – один из самых популярных языков программирования для науки о данных и потому обладает огромным количеством полезных дополнительных библиотек, разработанных его колоссальным сообществом программистов.

Учитывая, что производительность таких интерпретируемых языков, как Python, для вычислительно-емких задач хуже, чем у языков программирования более низкого уровня, были разработаны дополнительные библиотеки, такие как *NumPy* и *SciPy*, которые опираются на низкоуровневые реализации на Fortran и C для быстродействующих и векторизованных операций на многомерных массивах.

---

[1] Перекрестная проверка, валидация (cross validation, CV) – это метод подтверждения работоспособности моделей, который используется для оценки того, насколько точно прогнозная модель будет работать на практике. Задача перекрестной проверки – выделить из тренировочного набора данных подмножество данных, которое будет использоваться с целью «протестировать» модель на этапе ее тренировки (так называемый перекрестно-проверочный набор), с тем чтобы ограничить такие проблемы, как переобучение, и дать представление о том, как модель будет обобщаться на независимый набор данных (т. е. ранее не встречавшихся данных, например из реальной задачи). – *Прим. перев.*

Для выполнения задач программирования машинного обучения мы главным образом будем обращаться к библиотеке *scikit-learn*, которая на сегодня является одной из самых популярных и доступных библиотек машинного обучения с открытым исходным кодом.

Установка библиотек Python

Python доступен для всех трех главных операционных систем – Microsoft Windows, Mac OS X и Linux, – и его установщик и документацию можно скачать с официального веб-сайта Python: https://www.python.org.

Эта книга написана для версии Python >= 3.4.3, при этом рекомендуем использовать самую последнюю версию Python 3 (на момент перевода последней была версия 3.5.2), несмотря на то что большинство примеров программ может также быть совместимо с Python >= 2.7.10 (на момент перевода – 2.7.12). Если для выполнения примеров программ вы решили использовать Python 2.7, то удостоверьтесь, что вы знакомы с главными различиями между этими двумя версиями Python. Хорошее краткое описание различий между Python 3.5 и 2.7 приведено на веб-странице https://wiki.python.org/moin/Python2orPython3.

Дополнительные библиотеки, которые мы будем использовать на протяжении всей книги, можно установить при помощи менеджера пакетов *pip*, который входит в состав стандартной библиотеки Python начиная с версии 3.3. Дополнительную информацию о менеджере пакетов pip можно найти на веб-странице https://docs.python.org/3/installing/index.html.

После успешной инсталляции среды Python мы можем запускать менеджер пакетов pip из командной строки в окне терминала для установки дополнительных библиотек Python:

```
pip install НекаяБиблиотека
```

Уже установленные библиотеки можно обновить при помощи опции --upgrade:

```
pip install НекаяБиблиотека --upgrade
```

Настоятельно рекомендуем к использованию альтернативный дистрибутив Python для научных вычислений Anaconda от компании Continuum Analytics. Это бесплатный, включая коммерческое использование, и готовый к использованию в среде предприятия дистрибутив Python, который объединяет все ключевые библиотеки Python, необходимые для работы в области науки о данных, математики и разработки, в одном удобном для пользователя кросс-платформенном дистрибутиве. Установщик Anaconda можно скачать с http://continuum.io/downloads, а краткое вводное руководство Anaconda доступно по прямой ссылке https://store.continuum.io/static/img/Anaconda-Quickstart.PDF.

После успешной установки дистрибутива Anaconda можно установить новые пакеты Python, используя для этого следующую ниже команду:

```
conda install НекаяБиблиотека
```

Существующие пакеты можно обновить при помощи команды:

```
conda upgrade НекаяБиблиотека
```

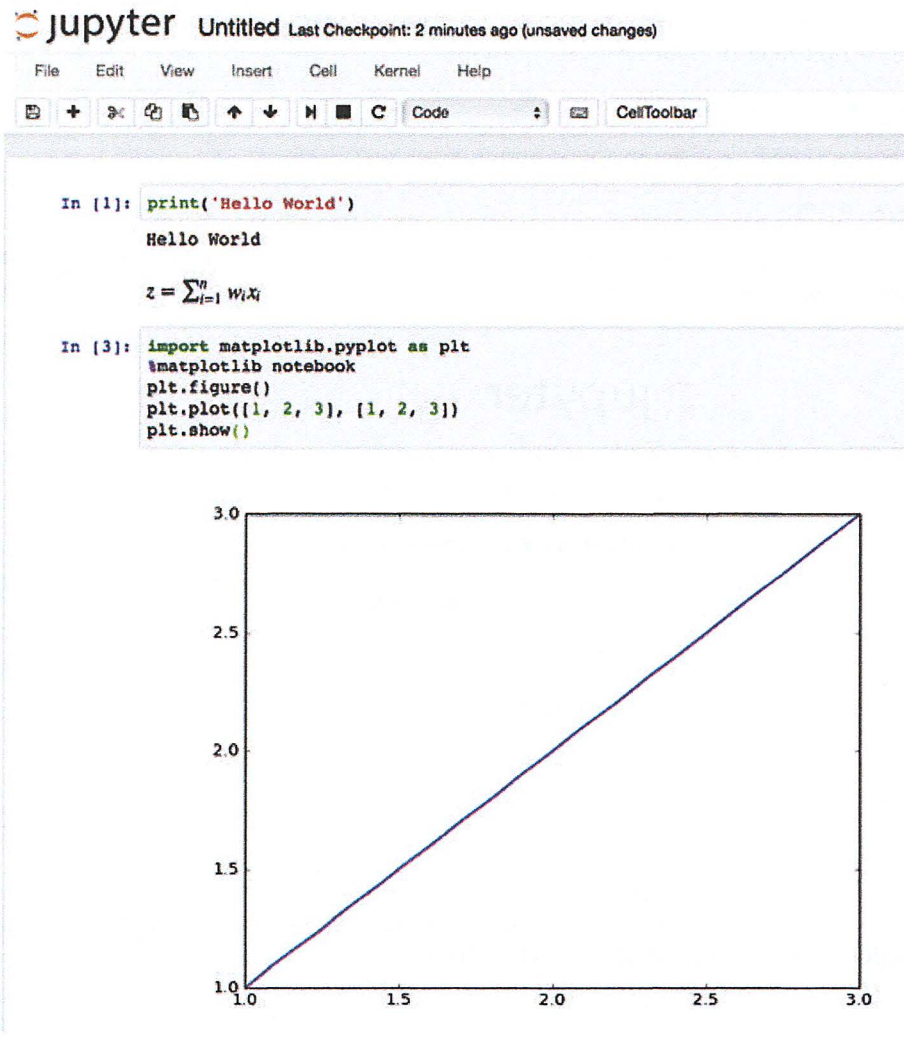
На протяжении всей книги для хранения и управления данными мы будем в основном использовать многомерные массивы *NumPy*. От случая к случаю мы будем пользоваться надстройкой над библиотекой NumPy – библиотекой *pandas*, которая предлагает дополнительные высокоуровневые инструменты управления данными, делая работу с табличными данными еще более удобной. В помощь нашей познавательной деятельности, а также в целях визуализации количественных данных, что порой бывает чрезвычайно полезно делать для интуитивного понимания материала, мы будем пользоваться полностью настраиваемой библиотекой *matplotlib*.

Номера версий основных пакетов Python, которые использовались при написании этой книги, упомянуты ниже. Удостоверьтесь, что номера версий ваших установленных пакетов идентичны приведенным в списке или выше, чтобы примеры программ гарантированно выполнялись корректно:

- NumPy 1.9.1;
- SciPy 0.14.0;
- scikit-learn 0.15.2;
- matplotlib 1.4.0;
- панды 0.15.2.

Блокноты (записные книжки) Jupyter/IPython

Некоторым читателям, возможно, интересно, что это за файлы с расширением «*.ipynb*» среди примеров программ. Это файлы блокнотов Jupyter (ранее называвшихся записными книжками IPython). Блокнотам Jupyter было отдано предпочтение над простыми сценарными файлами Python «*.py*», потому что, по нашему мнению, они наилучшим образом подходят для проектов в области анализа данных! Блокноты Jupyter позволяют иметь все в одном месте: наш исходный код, результаты выполнения исходного кода, графики данных и документацию, которая поддерживает синтаксис упрощенной разметки Markdown и мощный синтаксис LaTeX!



Записные книжки IPython были переименованы в блокноты Jupyter (http://jupyter.org) в начале 2015 г. Jupyter – это зонтичный проект, который наряду с Python предназначен для поддержки других языков программирования, в том числе Julia, R и многих других (уже более 40 языков). Впрочем, переживать не следует, т. к. для пользователя Python имеется всего лишь различие в терминологии.

Блокнот Jupyter можно установить, как обычно, при помощи pip.

```
$ pip install jupyter notebook
```

Как вариант можно воспользоваться установщиком Conda, в случае если мы инсталлировали Anaconda либо Miniconda:

```
$ conda install jupyter notebook
```

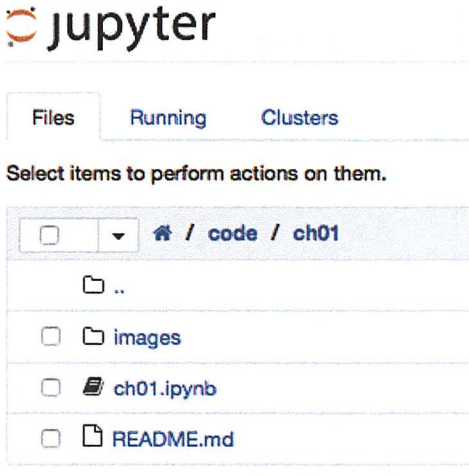
Чтобы открыть блокнот Jupyter, мы при помощи команды cd переходим в каталог с нашими примерами программ, например

```
$ cd ~/code/python-machine-learning-book
```

и запускаем блокнот Jupyter при помощи команды

```
$ jupyter notebook
```

Jupyter запустится в вашем браузере по умолчанию (как правило, по адресу http://localhost:8888/). Теперь мы можем просто выбрать блокнот, который вам нужна, из меню Jupyter.



За дополнительной информацией о проекте Jupyter рекомендуем обращаться к «*Руководству для начинающих по Jupyter*» (http://jupyter-notebook-beginner-guide.readthedocs.org/en/latest/what_is_jupyter.html).

Резюме

В этой главе мы провели широкомасштабную разведку обширной области информатики – машинного обучения и ознакомились с общей картиной и ее главными концепциями, которые мы собираемся разобрать в следующих главах более подробно.

Мы узнали, что обучение с учителем состоит из двух важных подобластей: задач классификации и регрессии. В отличие от модели классификации, которая позволяет распределять (категоризировать) объекты по известным классам, мы можем применять регрессию с целью предсказания непрерывных результатов для целевых переменных. Обучение без учителя не только предлагает удобные методы обнаружения структур в немаркированных данных, но и может использоваться для сжатия данных на этапе предобработки признаков.

Мы кратко прошлись по типичной дорожной карте применения машинного обучения для решения практических задач, которую мы возьмем за основу при более глубоком обсуждении и рассмотрении живых примеров в последующих главах.

В заключение мы инсталлировали среду Python, установили и обновили требуемые библиотеки, чтобы подготовиться к тому, чтобы увидеть машинное обучение в действии.

В следующей главе мы реализуем несколько самых ранних алгоритмов машинного обучения, используемых для решения задачи классификации, которые подготовят нас к *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, где при помощи библиотеки машинного обучения с открытым исходным кодом scikit-learn мы охватим более продвинутые алгоритмы машинного обучения. Поскольку алгоритмы машинного обучения учатся на данных, критически важно подавать им нужную информацию, и в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»* мы рассмотрим важные методы предварительной обработки данных. В *главе 5 «Сжатие данных путем снижения размерности»* мы узнаем о методах снижения размерности, которые позволят сжимать набор данных в подпространство признаков более низкой размерности, что может позитивно сказаться на вычислительной эффективности. Важным аспектом создания машинно- обучаемых моделей является выполнение оценки их качества и оценки того, насколько хорошо они могут делать прогнозы на новых, ранее не встречавшихся им данных. В *главе 6 «Изучение наиболее успешных методов оценки моделей и тонкой настройки гиперпараметров»* мы узнаем все о наиболее успешных практических методах тонкой настройки моделей и их оценки. В определенных сценариях мы все еще можем быть не удовлетворены качеством нашей прогнозной модели, несмотря на то что мы, возможно, провели часы или даже дни, тщательно ее настраивая и проверяя. В *главе 7 «Объединение моделей для методов ансамблевого обучения»* мы узнаем, как объединять различные машиннообучаемые модели для создания еще более мощных систем прогнозирования.

После того как мы охватим все важные концепции типичного конвейера машинного обучения[1], в *главе 8 «Применение алгоритмов машинного обучения в анализе мнений»* займемся реализацией модели, предназначенной для прогнозирования эмоций в тексте, в *главе 9 «Встраивание алгоритма машинного обучения в веб-приложение»* мы встроим ее в веб-приложение, чтобы поделиться им с миром. Затем в *главе 10 «Прогнозирование непрерывных целевых величин на основе регрессионного анализа»* мы воспользуемся алгоритмами машинного обучения для регрессионного анализа, которые позволят прогнозировать непрерывные выходные переменные, и в *главе 11 «Работа с немаркированными данными – кластерный анализ»* мы применим алгоритмы кластеризации, которые позволят находить в данных скрытые структуры. Последние две главы в этой книге будут касаться искусственных нейронных сетей, позволяющих решать такие сложные задачи, как распознавание изображений и речи. Эти темы в настоящее время являются одними из самых горячих среди научных исследований в области машинного обучения.

---

[1] В информатике конвейер представляет собой набор элементов обработки данных, соединенных последовательно, где выход одного элемента является входом следующего. Элементы конвейера часто выполняются параллельно по принципу квантования по времени. – *Прим. перев.*

Глава 2

Тренировка алгоритмов машинного обучения для задачи классификации

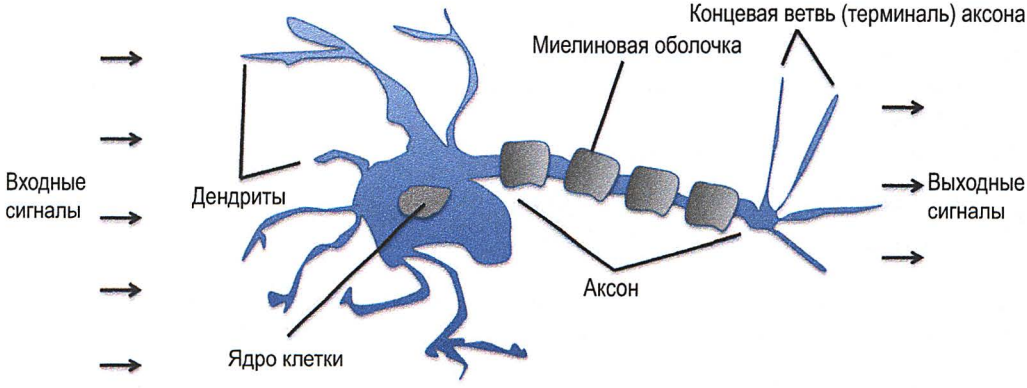
В этой главе мы воспользуемся двумя первыми алгоритмически описанными методами машинного обучения, применяемыми для классификации данных, *персептроном* и *адаптивными линейными нейронами*. Мы начнем с пошаговой реализации персептрона на языке Python и его тренировки для решения задачи классификации различных видов цветков из набора данных Ирисов Фишера (Iris). Это позволит нам понять принцип работы алгоритмов машинного обучения при решении задачи классификации и каким образом их можно эффективно реализовывать на Python. Затем мы перейдем к обсуждению азов оптимизации с использованием адаптивных линейных нейронов, которые заложат основу для использования более мощных классификаторов на основе библиотеки машинного обучения scikit-learn в следующей *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*.

В этой главе мы затронем следующие темы:

- формирование интуитивного понимания алгоритмов машинного обучения;
- использование библиотек pandas, NumPy и matplotlib для чтения, обработки и визуализации данных;
- реализация алгоритмов линейной классификации на Python.

Искусственные нейроны – краткий обзор ранней истории машинного обучения

Прежде чем перейти к более подробному обсуждению персептрона и связанных с ним алгоритмов, выполним небольшое познавательное путешествие по истокам машинного обучения. Стараясь понять, каким образом работает биологический мозг, с целью разработки искусственного интеллекта Уоррен Маккалок и Уолтер Питтс в 1943 г. впервые опубликовали концепцию упрощенной клетки головного мозга, так называемого *нейрона Маккалока–Питтса* (нейрон MCP) (У. Маккалок и У. Питтс, «*Логическое исчисление идей, относящихся к нервной активности*», Bulletin of Mathematical Biophysics (Бюллетень математической биофизики), 5 (4):115-133, 1943). Нейроны – это взаимосвязанные нервные клетки головного мозга, которые участвуют в обработке и передаче химических и электрических сигналов, как проиллюстрировано на нижеследующем рисунке:



Маккалок и Питтс описали такую нервную клетку в виде простого логического элемента с бинарными выходами; множественные входные сигналы поступают в дендриты, затем интегрируются в клеточное тело, и если накопленный сигнал превышает определенный порог, то генерируется выходной сигнал, который аксоном передается дальше.

Всего несколько лет спустя Фрэнк Розенблатт представил научному сообществу концепцию правила обучения персептрона[1], опираясь на модель нейрона MCP (Ф. Розенблатт, «*Персептрон, воспринимающий и распознающий автомат*», Cornell Aeronautical Laboratory (Лаборатория аэронавтики Корнелльского университета), 1957). Вместе с правилом персептрона Розенблатт предложил алгоритм, который автоматически обучался оптимальным весовым коэффициентам, которые затем перемножались с входными признаками для принятия решения о том, активировать нейрон или нет. В контексте обучения с учителем и задачи классификации такой алгоритм можно использовать для распознавания принадлежности образца к тому или иному классу[2].

Более формально эту проблему можно изложить как задачу бинарной классификации, где для простоты мы обозначим наши два класса как 1 (положительный класс) и –1 (отрицательный класс). Затем можно определить передаточную функцию, или *функцию активации* $\phi(z)$, которая принимает линейную комбинацию определенных входных значений, или входной вектор x, и соответствующий весовой вектор $\boldsymbol{w}$, где z – это так называемый чистый вход (от англ. net input) ($z = w_1x_1 + ... + w_mx_m$):

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

---

[1] Термин составлен из двух английских слов: perception (восприятие) и automaton (автомат). Персептрон еще часто называют пороговым сумматором. Правило, или процесс, обучения – это применяемый многократно метод (математическая формула) обновления весов и узла смещения модели, который улучшает ее эффективность. – *Прим. перев.*

[2] После обучения персептрон готов работать в одном из двух режимов: распознавания либо обобщения. – *Прим. перев.*

Теперь если активация отдельно взятого образца $x^{(i)}$, т. е. выход из $\phi(z)$, превышает заданный порог θ, то мы распознаем класс 1, в противном случае – класс –1. В алгоритме персептрона функция активации $\phi(\cdot)$ – это простая *единичная ступенчатая функция*, которую иногда также называют *ступенчатой функцией Хевисайда* (или функцией единичного скачка):

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

Для простоты мы можем перенести порог θ в левую часть равенства и определить вес с нулевым индексом как $w_0 = -\theta$ и $x_0 = -1$ и в результате записать чистый вход z в более компактной форме: $z = w_0x_0 + w_1x_1 + \ldots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x}$ и $\phi(z) = \begin{cases} 1, & \text{если } z \geq 0 \\ -1, & \text{иначе} \end{cases}$.

В следующих разделах мы часто будем пользоваться основами системы обозначения из линейной алгебры. Например, мы сократим сумму произведений значений в x и w, используя скалярное (внутреннее) произведение векторов, при этом T в надстрочном индексе обозначает транспонирование, т. е. операцию, которая трансформирует вектор-столбец в вектор-строку и наоборот:

$$z = w_1x_1 + \ldots + w_mx_m = \sum_{j=0}^{m} \boldsymbol{x}_j\boldsymbol{w}_j = \boldsymbol{w}^T\boldsymbol{x}.$$

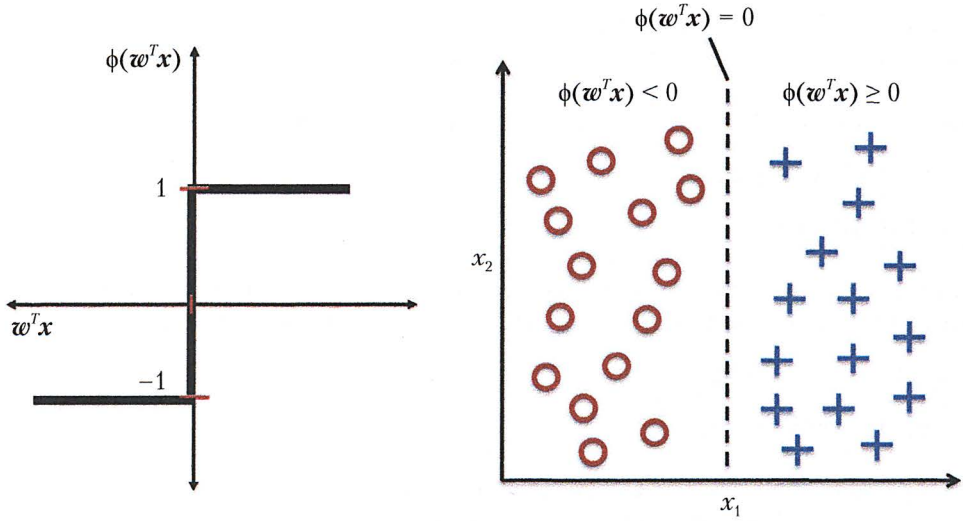
Например: $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32.$

Кроме того, операция транспонирования может применяться к матрице для ее отражения по диагонали. Например:

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

В этой книге мы будем пользоваться только самыми элементарными понятиями из линейной алгебры. Однако если вам требуется быстро освежить свои знания в этой области, то, пожалуйста, обратитесь к превосходному ресурсу «*Обзор линейной алгебры со справочником*» (Linear Algebra Review and Reference) от Зико Кольтера, который находится в свободном доступе по прямой ссылке http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf. В русскоязычном сегменте можно обратиться к лекциям Панова Т. Е. «*Линейная алгебра и геометрия*» (http://higeom.math.msu.su/people/taras/teaching/2012/panov-linalg2012.pdf) и Тыртышникова Е. Е. «*Матричный анализ и линейная алгебра*» (http://www.inm.ras.ru/vtm/lection/all.pdf).

Следующий ниже рисунок иллюстрирует, как чистый вход $z = \boldsymbol{w}^T\boldsymbol{x}$ втискивается функцией активации персептрона в бинарный выход (–1 либо 1) (левая часть рисунка) и как это может использоваться для различения двух линейно разделимых классов (правая часть рисунка):



Весь смысл идеи, лежащей в основе нейрона MCP и персептронной модели Розенблатта *с порогом*, состоит в том, чтобы использовать редукционистский подход для имитации работы отдельного нейрона головного мозга: он либо *активируется*, либо нет. Таким образом, первоначальное правило обучения персептрона Розенблатта, т. е. правило обновления весов в персептроне, было довольно простым и может быть резюмировано следующими шагами:

1. Инициализировать веса нулями либо малыми случайными числами.
2. Для каждого тренировочного образца $\boldsymbol{x}^{(i)}$ выполнить следующие шаги:
   1) вычислить выходное значение $\hat{y}$;
   2) обновить веса.

Здесь выходное значение – это метка класса, идентифицированная единичной ступенчатой функцией, которую мы определили ранее, при этом одновременное обновление каждого веса w_j в весовом векторе $\boldsymbol{w}$ можно более формально записать как

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

Значение Δw_j, которое используется для обновления веса w_j, вычисляется правилом обучения персептрона:

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

где η – это темп обучения (константа между 0.0 и 1.0), $y^{(i)}$– истинная метка класса i-го тренировочного образца и $\hat{y}^{(i)}$ – идентифицированная метка класса. Важно отметить, что все веса в весовом векторе обновляются одновременно, т. е. мы не вычисляем $\hat{y}^{(i)}$ повторно до тех пор, пока все веса Δw_j не будут обновлены. Конкретно для двумерного набора данных мы запишем обновление следующим образом:

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

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

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

Прежде чем реализовать правило персептрона на Python, выполним простой мысленный эксперимент, чтобы проиллюстрировать, каким до прекрасного простым это обучающее правило на самом деле является. В двух сценариях, где персептрон правильно распознает метку класса, веса остаются неизменными:

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

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

Однако в случае неправильного распознавания веса продвигаются в направлении соответственно положительного или отрицательного целевого класса:

$$\Delta w_j = \eta(1 - -1)x_j^{(i)} = \eta(2)x_j^{(i)};$$

$$\Delta w_j = \eta(-1 - 1)x_j^{(i)} = \eta(-2)x_j^{(i)}.$$

Чтобы получить более глубокое интуитивное понимание мультипликативного коэффициента $x_j^{(i)}$, обратимся к еще одному простому примеру, где:

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

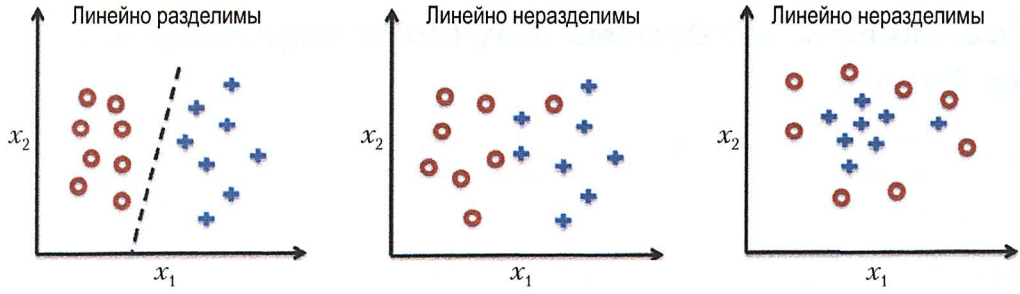
Примем, что $x_j^{(i)} = 0.5$, и мы ошибочно классифицируем этот образец как -1. В этом случае мы увеличиваем соответствующий вес на 1, благодаря чему чистый вход $x_j^{(i)} \times w_j^{(i)}$ будет более положительным при следующей встрече с этим образцом и, таким образом, с большей вероятностью будет выше порога единичной ступенчатой функции для классификации образца как $+1$:

$$\Delta w_j^{(i)} = (1 - -1)0.5 = (2)0.5 = 1.$$

Обновление веса пропорционально значению $x_j^{(i)}$. Например, если имеется еще один образец $x_j^{(i)} = 2$, котоый правильно классифицируется как -1, то мы отодвинем границу решения на еще большее расстояние, чтобы в следующий раз классифицировать этот образец правильно:

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

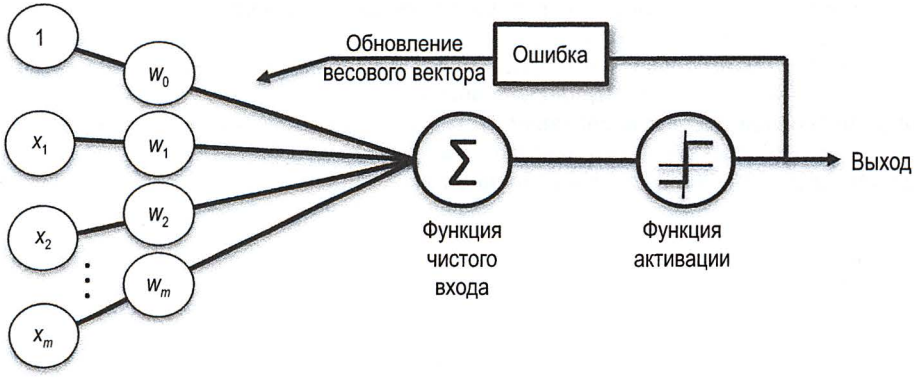
Важно отметить, что сходимость персептрона, т. е. достижение устойчивого состояния, гарантируется, только если эти два класса линейно разделимы и темп обучения достаточно небольшой. Если эти два класса не могут быть разделены линейной границей решения, то мы можем установить максимальное число проходов по тренировочному набору данных (*эпох*) и/или порог на допустимое число случаев ошибочной классификации (число мисклассификаций) – иначе персептрон никогда не прекратит обновлять веса:



Скачивание примеров программ

Вы можете скачать файлы с примерами из вашего аккаунта по адресу http://www.packtpub.com/ для всех книг издательства «Packt Publishing», которые вы приобрели. Если вы купили эту книгу в другом месте, то можно посетить http://www.packtpub.com/support и зарегистрироваться там, чтобы получить файлы прямо по электронной почте.

Теперь, прежде чем мы перейдем к следующему разделу и займемся реализацией, подытожим то, чему мы научились, и на простом рисунке проиллюстрируем общий принцип работы персептрона:



Приведенный выше рисунок иллюстрирует то, каким образом персептрон получает входы из образца x и смешивает их с весами w для вычисления чистого входа. Чистый вход затем передается функции активации (в данном случае единичная ступенчатая функция), которая генерирует бинарный выход –1 или +1 – распознанную метку класса образца. Во время фазы обучения этот выход используется для вычисления ошибки распознавания и для обновления весов.

Реализация алгоритма обучения персептрона на Python

В предыдущем разделе мы познакомились с принципом работы персептрона Розенблатта; теперь пойдем дальше и реализуем его на Python, после чего применим его к набору данных цветков ириса, который мы представили в *главе 1 «Наделение компьютеров способностью обучаться на данных»*. Мы примем объектно-ориентированный подход, определив интерфейс персептрона как класс языка Python. Это позволит нам инициализировать новые объекты персептрона, которые смогут обучаться на данных при помощи метода fit (выполнить подгонку модели) и делать прогнозы при помощи метода predict (распознать). В качестве условного обозначения мы добавим символ подчеркивания в обозначение атрибутов, создаваемых не во время инициализации объекта, а в результате вызова других методов объекта, например self.w_.

Если вы еще незнакомы с научными библиотеками Python или же вам требуется освежить свои знания, пожалуйста, обратитесь к следующим ресурсам:
Библиотека NumPy: http://wiki.scipy.org/Tentative_NumPy_Tutorial
Библиотека Pandas: http://pandas.pydata.org/pandas-docs/stable/ tutorials.html
Библиотека Matplotlib: http://matplotlib.org/users/beginner.html
Кроме того, для того чтобы лучше отслеживать приводимые примеры, рекомендуется скачать блокноты Jupyter с веб-сайта Packt. По поводу общих сведений о записных книжках Jupyter посетите http://jupyter.org/ или обратитесь к документации по IPython http://ipython.readthedocs.io/en/stable/. Исходные коды примеров, блокноты Jupyter и дополнительную информацию можно также получить на странице книги на Github https://github.com/rasbt/python-machine-learning-book.

```
import numpy as np
class Perceptron(object):
    """Классификатор на основе персептрона.

    Параметры
    ---------
    eta : float
        Темп обучения (между 0.0 и 1.0)
    n_iter : int
        Проходы по тренировочному набору данных.

    Атрибуты
    --------
    w_ : 1-мерный массив
        Весовые коэффициенты после подгонки.
    errors_ : список
        Число случаев ошибочной классификации в каждой эпохе.

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Выполнить подгонку модели под тренировочные данные.

        Параметры
```

```
        ---------
        X : {массивоподобный}, форма = [n_samples, n_features]
            тренировочные векторы, где
            n_samples  - число образцов и
            n_features - число признаков.
        y : массивоподобный, форма = [n_samples]
            Целевые значения.

        Возвращает
        ----------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0]  += update
                errors      += int(update != 0.0)
            self.errors_.append(errors)
        return self

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

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

Используя эту реализацию персептрона, мы теперь можем инициализировать новые объекты Perceptron, задавая темп обучения eta и число эпох n_iter, т. е. проходов, по тренировочному набору. При помощи метода для выполнения подгонки fit мы инициализируем веса в атрибуте self.w_ нулевым вектором $\mathbb{R}^{m+1}$, где m обозначает число размерностей (признаков) в наборе данных, причем мы добавляем еще одну, чтобы в нулевой позиции учесть веса (т. е. порог).

В библиотеке NumPy индексация одномерных массивов работает так же, как в списках Python, – в записи при помощи квадратных скобок ([]). Для двумерных массивов первый индексатор обозначает номер строки, второй – номер столбца. Например, чтобы выбрать третью строку и четвертый столбец двумерного массива X, мы используем X[2, 3].

После инициализации весов метод fit в цикле просматривает отдельные образцы тренировочного набора и обновляет веса согласно правилу обучения персептрона, которое мы обсудили в предыдущем разделе. Метки классов распознаются методом predict, который также вызывается в методе fit, для того чтобы идентифицировать метку класса для обновления веса. Однако метод fit может применяться и для распознавания меток классов новых данных уже после подгонки модели. Кроме того, в списке self.errors_ мы также собираем число ошибочных распознаваний классов в каждой эпохе, в результате чего позже мы можем проанализировать, насколько хорошо наш

персептрон работал во время тренировки. Используемая в методе net_input функция np.dot просто вычисляет скалярное (внутреннее) произведение $\boldsymbol{w}^T\boldsymbol{x}$ векторов[1].

Вместо того чтобы для вычисления скалярного произведения двух массивов a и b использовать библиотеку NumPy, применяя для этого a.dot(b) или np.dot(a, b), это вычисление можно выполнить на чистом Python в виде sum([i*j, for i, j in zip(a, b)]. Однако преимущество использования библиотеки NumPy над классическими структурами цикла for на Python заключается в том, что ее арифметические операции векторизованы. **Векторизация** означает, что элементарная арифметическая операция автоматически применяется ко всем элементам в массиве. Формулируя наши арифметические операции в виде последовательности инструкций на массиве вместо выполнения серии операций для каждого элемента по одному, мы можем лучше использовать наши современные процессорные архитектуры с поддержкой **ОКМД** (SIMD), т. е. **одиночного потока команд, множественного потока данных** (single instruction, multiple data). Кроме того, в библиотеке NumPy используются высокооптимизированные динамические библиотеки линейной алгебры, такие как *библиотека основных подпрограмм линейной алгебры* **BLAS** (Basic Linear Algebra Subprograms) и *пакет линейной алгебры* **LAPACK** (Linear Algebra Package), написанные на C или Fortran. Наконец, библиотека NumPy также позволяет писать исходный код в более компактном и интуитивном виде, используя основы линейной алгебры, такие как скалярные произведения векторов и матриц.

Тренировка персептронной модели на наборе данных цветков ириса

Для тестирования нашей реализации персептрона загрузим из набора данных ирисов два класса цветков: *ирис щетинистый (Iris setosa)* и *ирис разноцветный (Iris versicolor)*. Несмотря на то что правило персептрона не ограничивается двумя размерностями, в целях визуализации мы рассмотрим только два признака – *длина чашелистика (sepal length)* и *длина лепестка (petal length)*. Кроме того, по практическим соображениям мы выбрали всего два класса цветков – *ирис щетинистый* и *ирис разноцветный*. Тем не менее алгоритм персептрона можно распространить на многоклассовую классификацию, например методом *один против всех*.

Стратегия «**один против всех**», или «**один против остальных**» (one-vs.-rest, one-vs.-all, one-against-all, OvA, OvR, OAA), – метод, используемый для распространения бинарного классификатора на задачи многоклассовой классификации. При помощи OvA можно тренировать один классификатор в расчете на класс, где отдельно взятый класс трактуется как положительный, а образцы из всех остальных классов – как отрицательные. Если мы будем классифицировать новый образец данных, то будем использовать наши n классификаторов, где n – это число меток классов, и назначим отдельно взятому образцу метку класса с самым высоким коэффициентом доверия. В случае с персептроном мы бы использовали OvA для выбора метки класса, которая связана с наибольшим абсолютным чистым входным значением[2].

---

[1] См. документацию по функции numpy.dot на https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html. – *Прим. перев.*

[2] Эта стратегия требует от базовых классификаторов генерировать не просто метку класса, а вещественные коэффициенты доверия к их решению; одни лишь дискретные метки классов могут привести к двусмысленности, когда для одного образца делается прогноз, состоящий из нескольких классов. Принятие решений означает применение всех классификаторов к ранее не встречавшемуся образцу X и идентификацию метки K, для которой соответствующий классификатор выдает наивысший коэффициент доверия (https://en.wikipedia.org/wiki/Multiclass_classification). – *Прим. перев.*

Сначала мы воспользуемся библиотекой *pandas*, чтобы загрузить набор данных с цветками ириса непосредственно из *Репозитория машинного обучения Калифорнийского университета в Ирвайне (UCI Machine Learning Repository)* в объект DataFrame и распечатать последние пять строк, воспользовавшись для этого методом tail, дабы удостовериться, что данные были загружены правильно:

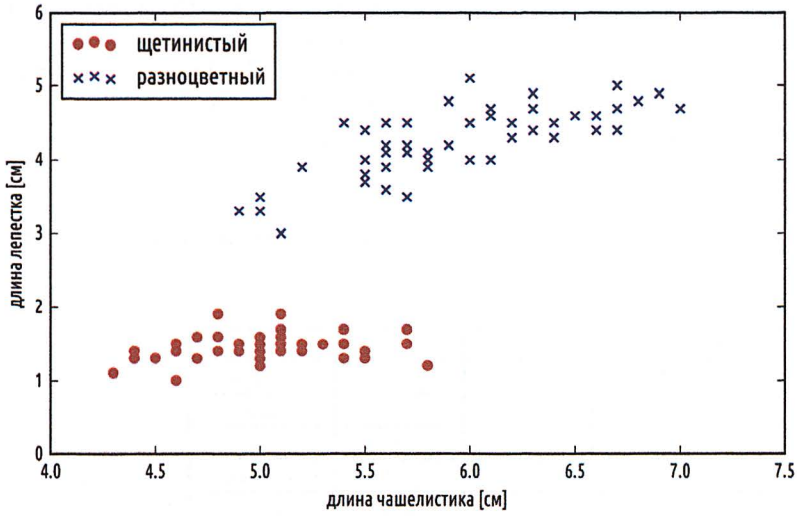
```
>>>
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
df = pd.read_csv(url, header=None)
df.tail()
```

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

Затем выделим первые 100 меток классов, которые соответствуют 50 цветкам *ириса щетинистого* и 50 цветкам *ириса разноцветного*, и преобразуем метки классов в две целочисленные метки классов: 1 (разноцветный Versicolor) и -1 (щетинистый Setosa), – которые мы присвоим вектору y, где метод values объекта DataFrame библиотеки pandas генерирует представление, соответствующее библиотеке NumPy. Аналогичным образом выделим из этих 100 тренировочных образцов столбец первого признака (*длина чашелистика*) и столбец третьего признака (*длина лепестка*) и присвоим их матрице признаков X, которую можно визуализировать при помощи двумерного точечного графика (именуемого также диаграммой рассеяния):

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

y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='щетинистый')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='разноцветный')
plt.xlabel('длина чашелистика')
plt.ylabel('длина лепестка')
plt.legend(loc='upper left')
plt.show()
```

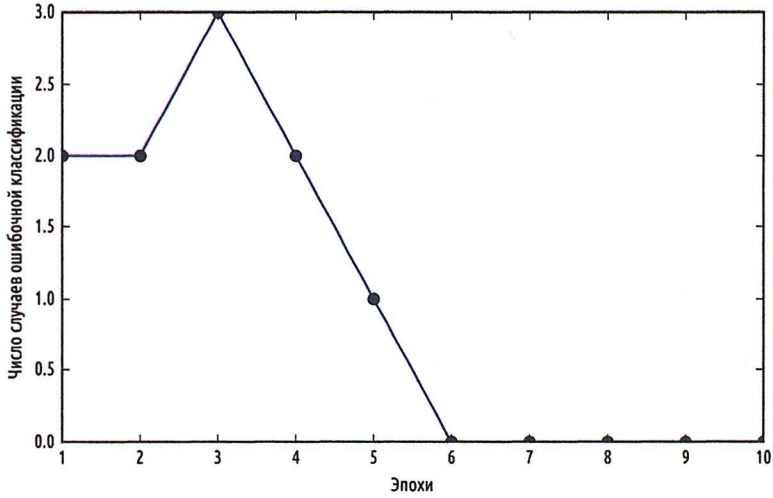
После выполнения приведенного выше примера должен получиться следующий ниже точечный график:



Теперь пора натренировать наш алгоритм персептрона на подмножестве данных цветков ириса, которые мы только что выделили. Помимо этого, мы также построим график *ошибок классификации* для каждой эпохи, чтобы удостовериться, что алгоритм сходился и нашел границу решения, которая разделяет два класса цветков ириса:

```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Эпохи')
# число ошибочно классифицированных случаев во время обновлений
plt.ylabel('Число случаев ошибочной классификации')
plt.show()
```

После выполнения приведенного выше примера мы должны увидеть график ошибок классификации в сопоставлении с числом эпох, как показано ниже:



Как видно из предыдущего графика, наш персептрон стабилизировался уже после шестой эпохи и теперь должен быть в состоянии идеально классифицировать тренировочные образцы. Реализуем небольшую вспомогательную функцию, которая визуально показывает границы решений для двумерных наборов данных:

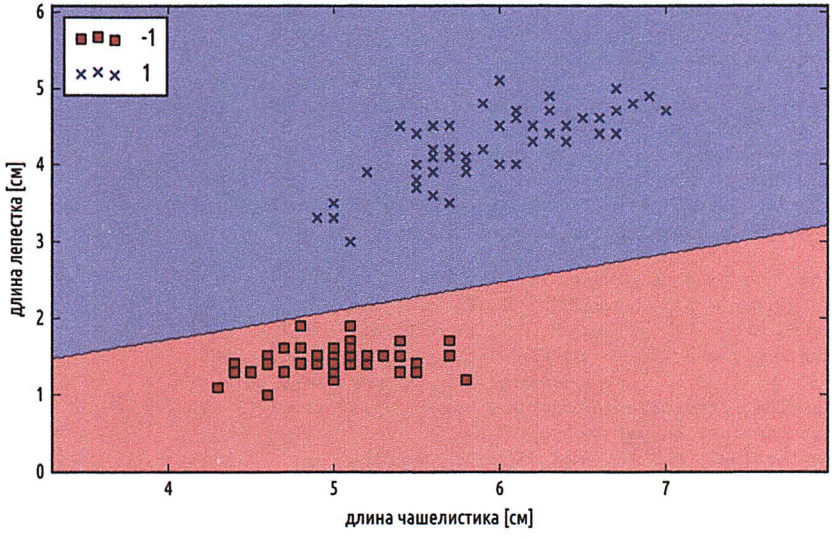
```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

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

    # вывести поверхность решения
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # показать образцы классов
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
```

Сначала мы определяем цвета и маркеры и затем при помощи ListedColormap создаем из списка цветов палитру (карту цветов). Затем мы определяем минимальные и максимальные значения для двух признаков и на основе векторов этих признаков создаем пару матричных массивов xx1 и xx2, используя для этого функцию meshgrid библиотеки NumPy[1]. Учитывая, что мы натренировали наш персептронный классификатор на двупризнаковых размерностях, мы должны сгладить матричные массивы и создать матрицу с тем же самым числом столбцов, что и у тренировочного подмножества цветков ириса, в результате чего можно применить метод predict и идентифицировать метки классов Z для соответствующих точек матрицы. После преобразования формы идентифицированных меток классов Z в матрицу с такими же размерностями, что и у xx1 и xx2, можно начертить контурный график, используя для этого функцию contourf библиотеки matplotlib, которая ставит в соответствие разным областям решения разный цвет по каждому идентифицированному классу в матричном массиве:

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

[1] Матричный массив (grid array), или матрица, – двумерный массив, состоящий из строк и столбцов; термин пришел из Pascal-подобных языков. – *Прим. перев.*

После выполнения предыдущего примера мы должны увидеть график областей решения, как показано на нижеследующем рисунке:



Как видно из предыдущего графика, персептрон обучился границе решения, которая может идеально расклассифицировать все образцы тренировочного подмножества цветков ириса.

Несмотря на то что персептрон полностью распределил цветки ириса на два класса, сходимость представляет для персептрона одну из самых больших проблем. Фрэнк Розенблатт математически доказал, что правило обучения персептрона сходится, только если два класса могут быть разделены линейной гиперплоскостью. Однако если полностью разделить классы такой линейной границей решения невозможно, то без установления максимального числа эпох веса никогда не прекратят обновляться.

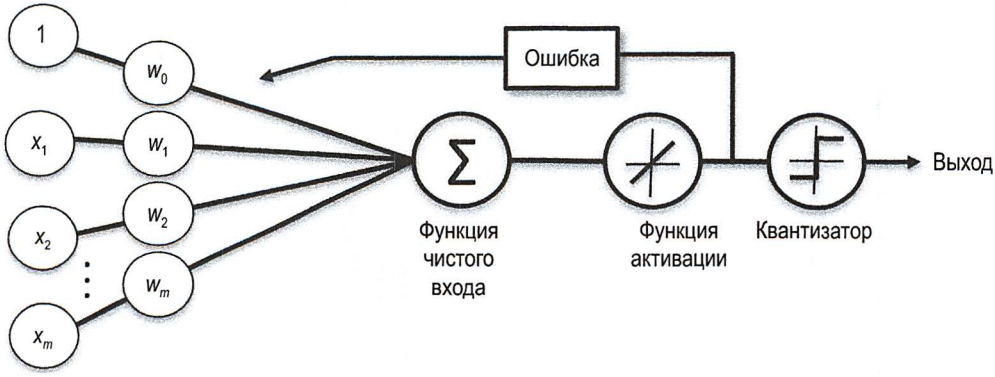
Адаптивные линейные нейроны и сходимость обучения

В этом разделе мы рассмотрим другой тип однослойной нейронной сети: **ADALINE (ADAptive LInear NEuron, адаптивный линейный нейрон)**. Концепция ADALINE была опубликована Берндардом Видроу и его докторантом Тэддом Хоффом спустя всего несколько лет после алгоритма персептрона Фрэнка Розенблатта, который можно рассматривать развитием последнего (Б. Видроу и др., Adaptive «Adaline» neuron using chemical «memistors» (*«Адаптивный нейрон "Adaline" с использованием химических "мемисторов"»*). Number Technical Report Stanford Electron. Labs, Стэнфорд, Калифорния, октябрь 1960). Алгоритм обучения особенно интересен тем, что он иллюстрирует ключевой принцип определения и минимизации функций стои-

мости, который закладывает основу для понимания более продвинутых алгоритмов машинного обучения для задач классификации, таких как логистическая регрессия и метод опорных векторов, а также регрессионных моделей, которые мы обсудим в будущих главах.

Основное отличие правила обучения ADALINE (также известного как *правило Уидроу-Хопфа*, или дельта-правило) от правила обучения персептрона Розенблатта в том, что в нем для обновления весов используется линейная функция активации, а не единичная ступенчатая, как в персептроне. В ADALINE эта функция активации $\phi(z)$ представляет собой просто тождественное отображение чистого входа, в результате чего $\phi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$.

Помимо линейной функции активации, которая используется для извлечения весов, далее с целью распознавания меток классов используется *квантизатор*, аналогичный встречавшейся ранее единичной ступенчатой функции, как проиллюстрировано на нижеследующем рисунке:



Если сравнить приведенный выше рисунок с иллюстрацией алгоритма обучения персептрона, который мы видели ранее, то ADALINE отличается тем, что вместо бинарных меток классов теперь для вычисления ошибки модели и для обновления весов используется непрерывнозначный выход из линейной функции активации.

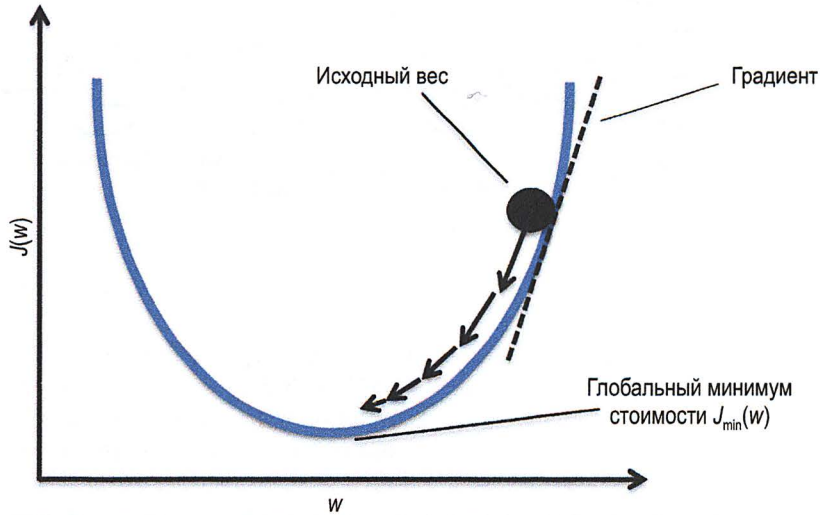
Минимизация функций стоимости методом градиентного спуска

Одна из ключевых составляющих алгоритмов машинного обучения с учителем состоит в задании *целевой функции*, которая подлежит оптимизации во время процесса обучения. Эта целевая функция часто является *функцией стоимости*, или функцией потерь, которую мы хотим минимизировать. В случае с ADALINE можно задать функцию стоимости J, которая извлекает веса в виде **суммы квадратичных ошибок** (sum of squared errors, SSE), или суммы отклонений расчетных результатов от истинных меток классов:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \phi(z^{(i)})\right)^2.$$

Член полусуммы $\frac{1}{2}$ добавлен просто для удобства; как мы убедимся в последующих абзацах, он упростит получение градиента. В отличие от единичной ступенчатой функции, основное преимущество этой непрерывной линейной функции активации состоит в том, что функция стоимости становится дифференцируемой. Еще одно хорошее свойство этой функции стоимости – в том, что она выпуклая; следовательно, мы можем использовать простой и одновременно мощный алгоритм оптимизации – алгоритм *градиентного спуска* для нахождения весов, которые минимизируют нашу функцию стоимости для решения задачи классификации образцов в наборе данных цветков ириса[1].

Как проиллюстрировано на нижеследующем рисунке, мы можем описать лежащую в основе градиентного спуска идею как *спуск вниз по склону холма*, пока не будет достигнут локальный или глобальный минимум стоимости. На каждой итерации мы делаем шаг в противоположную от градиента сторону, причем размер шага определяется значением темпа обучения и наклоном градиента (его угловым коэффициентом):



Теперь можно обновлять веса на основе градиентного спуска путем выполнения шага в противоположную сторону от градиента $\nabla J(\boldsymbol{w})$ нашей функции стоимости $J(\boldsymbol{w})$:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}.$$

Здесь изменение веса $\Delta\boldsymbol{w}$ определяется как отрицательный градиент, помноженный на темп обучения η:

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

---

[1] Градиентный (наискорейший) спуск (gradient descent) – алгоритм градиентного спуска, инкрементный алгоритм оптимизации, или поиска оптимального решения, где приближение к локальному минимуму функции идет шагами, пропорциональными *обратной* величине градиента этой функции в текущей точке. – *Прим. перев.*

Для расчета градиента функции стоимости нам нужно вычислить частную производную функции стоимости относительно каждого веса w_j, $\frac{\delta J}{\delta w_j} = -\sum_i \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)}$, в результате чего можно записать обновление веса w_j как $\Delta w_j = -\eta \frac{\delta J}{\delta w_j} = \eta \sum_i \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)}$.

Учитывая, что мы обновляем все веса одновременно, наше правило обучения адалина принимает вид $\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}$.

Для тех, кто знаком с дифференциальным исчислением, частную производную функции стоимости с извлечением весов как суммы квадратичных ошибок (SSE) относительно j-го веса можно получить следующим образом:

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

Несмотря на то что правило обучения ADALINE выглядит идентичным правилу персептрона, значение $\phi(z^{(i)})$, где $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)}$ – это вещественное число, а не целочисленная метка класса. Кроме того, обновление веса вычисляется на основе всех образцов в тренировочном наборе (вместо инкрементного обновления веса после каждого образца), и поэтому такой подход также называется «пакетным» (batch) градиентным спуском.

Реализация адаптивного линейного нейрона на Python

Учитывая, что правило персептрона и правило ADALINE очень похожи, мы возьмем определенную ранее реализацию персептрона и изменим метод fit таким образом, чтобы веса обновлялись путем минимизации функции стоимости методом градиентного спуска:

```
class AdalineGD(object):
    """Классификатор на основе ADALINE (ADAptive LInear NEuron).

    Параметры
    ---------
    eta : float
        Темп обучения (между 0.0 и 1.0)
    n_iter : int
```

```
        Проходы по тренировочному набору данных.

    Атрибуты
    --------
    w_ : 1-мерный массив
        Веса после подгонки.
    errors_ : список
        Число случаев ошибочной классификации в каждой эпохе.

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

    def fit(self, X, y):
        """ Выполнить подгонку под тренировочные данные.

        Параметры
        ---------
        X : {массивоподобный}, форма = [n_samples, n_features]
            Тренировочные векторы, где
            n_samples - число образцов и
            n_features - число признаков.
        y : массивоподобный, форма = [n_samples]
            Целевые значения.

        Возвращает
        ----------
        self : объект

        """
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

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

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

    def activation(self, X):
        """Рассчитать линейную активацию"""
        return self.net_input(X)

    def predict(self, X):
        """Вернуть метку класса после единичного скачка"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

Вместо того чтобы обновлять веса после выполнения оценки каждого отдельного тренировочного образца, как в персептроне, мы вычисляем градиент на основе всего тренировочного набора данных, используя self.eta * errors.sum() для веса в нулевой позиции и self.eta * X.T.dot(errors) для весов от 1 по m, где X.T.dot(errors) – это *мат-*

рично-векторное умножение[1] нашей матрицы признаков на вектор ошибок. Аналогично предыдущей реализации персептрона мы аккумулируем значения стоимости в списке self.cost_, чтобы после тренировки удостовериться, что алгоритм сходился.

Выполнение матрично-векторного умножения подобно вычислению скалярного произведения векторов, где каждая строка в матрице рассматривается как отдельный вектор-строка. Такой векторизованный подход предлагает более компактную форму записи и приводит к более эффективным вычислениям при помощи библиотеки NumPy. Например:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1\times7+2\times8+3\times9 \\ 4\times7+5\times8+6\times9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}.$$

На практике часто требуется сначала поэкспериментировать, чтобы найти хороший темп обучения η для оптимальной сходимости. Поэтому для начала мы выберем два темпа обучения $\eta = 0.1$ и $\eta = 0.0001$ и построим график функций стоимости в сопоставлении с числом эпох, чтобы увидеть, насколько хорошо реализация алгоритма обучения ADALINE обучается на тренировочных данных.

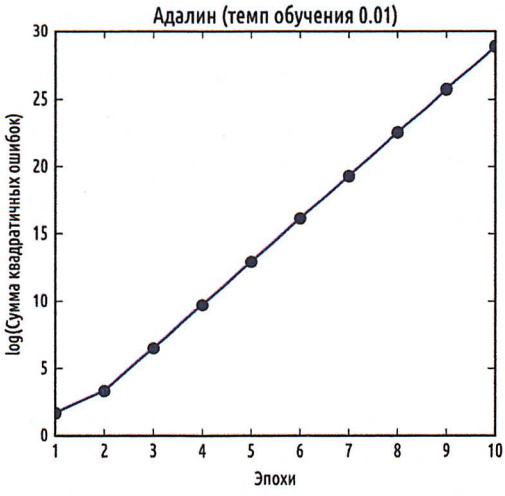
Темп обучения η и число эпох n_iter – это так называемые *гиперпараметры* алгоритмов обучения ADALINE и персептрона. В *главе 4 «Создание хороших тренировочных наборов – предобработка данных»* мы рассмотрим различные методы автоматического нахождения значений различных гиперпараметров, которые приводят к оптимальному качеству модели классификации данных.

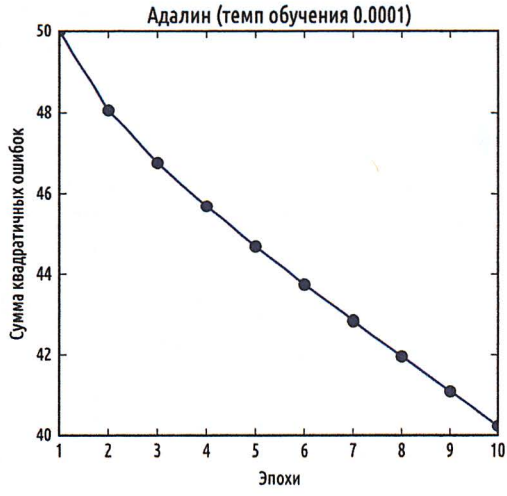
Теперь построим график стоимости в сопоставлении с числом эпох для двух разных темпов обучения:

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))
ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Эпохи')
ax[0].set_ylabel('log(Сумма квадратичных ошибок)')
ax[0].set_title('ADALINE (темп обучения 0.01)')
ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Эпохи')
ax[1].set_ylabel('Сумма квадратичных ошибок')
ax[1].set_title('ADALINE - (темп обучения 0.0001)')
plt.show()
```

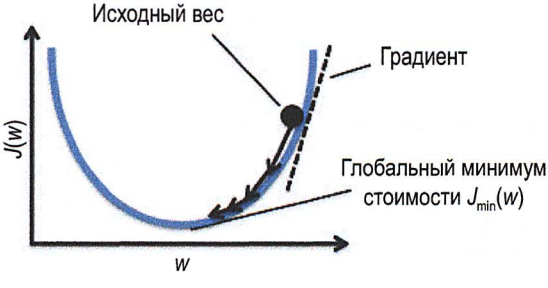
Как видно на получившихся ниже графиках функций стоимостей, мы столкнулись с двумя разными типами проблем. Левая диаграмма показывает, что произойдет, если выбрать слишком высокий темп обучения, – вместо того чтобы минимизировать функцию стоимости, в каждой эпохе ошибка увеличивается, потому что мы *промахиваемся* по глобальному минимуму:

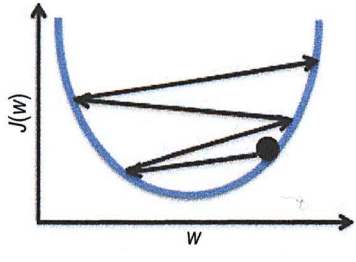
[1] Матрично-векторное умножение – это последовательность вычисления скалярных произведений. – *Прим. перев.*





Когда мы смотрим на правый график, мы видим, что стоимость уменьшается. Вместе с тем выбранный темп обучения $\eta = 0.0001$ настолько низок, что для сходимости алгоритм потребовал бы очень большое количество эпох. Следующий ниже рисунок иллюстрирует, каким образом мы изменяем значение отдельно взятого весового параметра для минимизации функции стоимости J (левый рисунок). На рисунке справа показано, что происходит, если мы выбираем слишком большой темп обучения, – мы промахиваемся по глобальному минимуму:





Многие алгоритмы машинного обучения, с которыми мы будем встречаться на протяжении всей этой книги, для получения оптимального качества так или иначе требуют выполнения масштабирования признаков; мы более подробно обсудим эту методику в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*. Алгоритм градиентного спуска является одним из многих, которые извлекают пользу из масштабирования признаков. Здесь мы будем использовать метод масштабирования признаков под названием *стандартизация*, который придает нашим данным свойство стандартного нормального распределения. Среднее значение каждого признака центрируется в значении 0, а столбец признака имеет единичное стандартное отклонение, т. е. равное 1. Например, для выполнения стандартизации j-го признака нам нужно просто вычесть эмпирическое среднее μ_j из каждого тренировочного образца и разделить результат на его стандартное отклонение σ_j:

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

Здесь x_j – это вектор, состоящий из значения j-го признака из всех тренировочных образцов n.

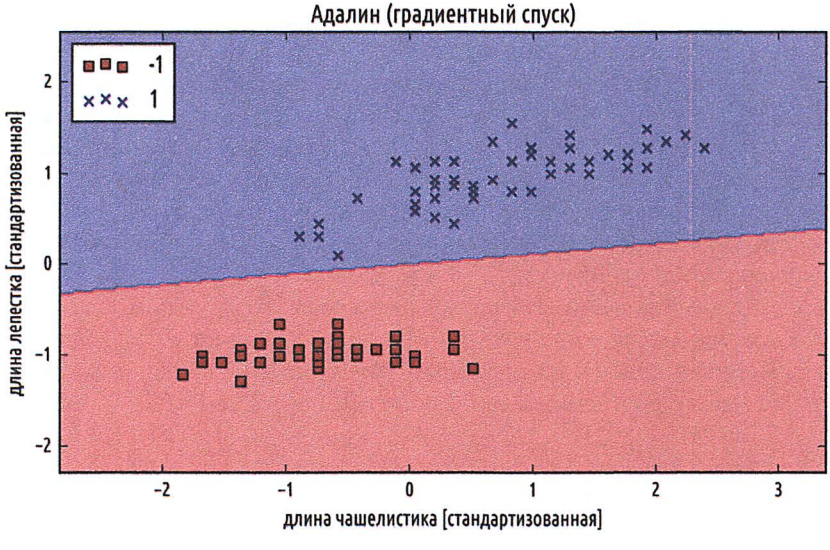
Стандартизацию можно легко получить при помощи методов mean и std библиотеки NumPy:

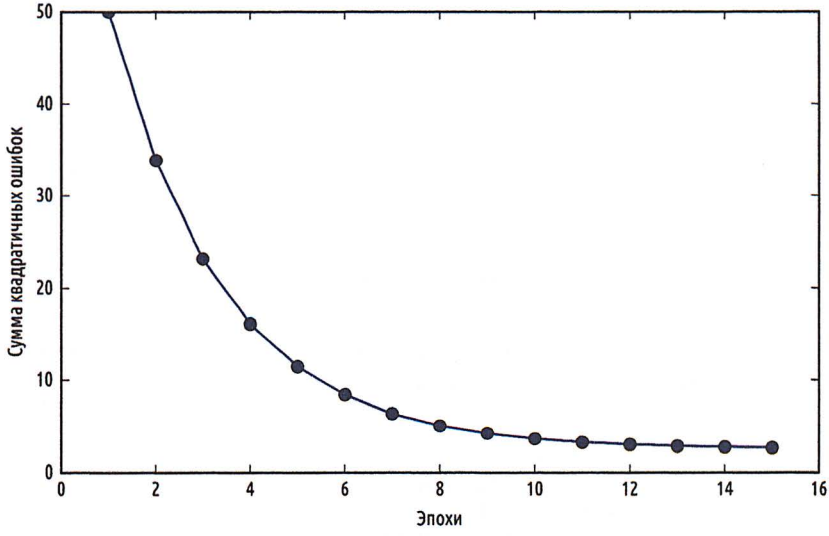
```
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

После выполнения стандартизации снова натренируем ADALINE и увидим, что теперь модель сходится с темпом обучения $\eta = 0.01$:

```
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)
plot_decision_regions(X_std, y, classifier=ada)
plt.title('ADALINE (градиентный спуск)')
plt.xlabel('длина чашелистика [стандартизованная]')
plt.ylabel('длина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Эпохи')
plt.ylabel('Сумма квадратичных ошибок')
plt.show()
```

После выполнения предыдущего примера мы должны увидеть рисунок областей решения, а также график снижающейся стоимости, как показано ниже:





Как видно на приведенных графиках, теперь ADALINE сходится после тренировки на стандартизированных признаках с темпом обучения $\eta = 0.01$. Однако отметим, что сумма квадратичных ошибок (SSE) остается ненулевой даже при том, что все образцы были классифицированы правильно.

Крупномасштабное машинное обучение и стохастический градиентный спуск

В предыдущем разделе мы узнали, как минимизировать функцию стоимости путем выполнения шага в противоположном от градиента направлении (в направлении антиградиента), который рассчитывается из всего тренировочного набора; именно поэтому такой подход иногда также называют *пакетным* градиентным спуском. Теперь представим, что у нас есть очень большой набор данных с миллионами точек данных, что весьма распространено во многих приложениях машинного обучения. В подобных сценариях выполнение пакетного градиентного спуска может быть в вычислительном плане довольно дорогостоящим, поскольку мы должны выполнять переоценку всего тренировочного набора данных каждый раз, когда мы делаем один шаг к глобальному минимуму.

Популярной альтернативой алгоритму пакетного градиентного спуска является *стохастический градиентный спуск*, иногда также именуемый *итеративным*, или *динамическим* (онлайновым), градиентным спуском. Вместо обновления весов, основываясь на сумме накопленных ошибок по всем образцам $\boldsymbol{x}^{(i)}$:

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

мы обновляем веса инкрементно по каждому тренировочному образцу:

$$\eta(y^{(i)} - \phi(z^{(i)})\boldsymbol{x}^{(i)}.$$

Хотя стохастический градиентный спуск можно рассматривать как аппроксимацию градиентного спуска, он обычно достигает сходимости намного быстрее из-за более частых обновлений весов. Учитывая, что каждый градиент рассчитывается на одном-единственном тренировочном примере, поверхность ошибок более зашумлена, чем в градиентном спуске, что также может иметь свое преимущество в том, что стохастический градиентный спуск с большей готовностью может выходить из мелких локальных минимумов. Для получения точных результатов методом стохастического градиентного спуска важно предоставить ему данные в произвольном порядке, и поэтому нам нужно перемешивать тренировочный набор по каждой эпохе для предотвращения зацикливания.

В реализациях стохастического градиентного спуска постоянный темп обучения η часто заменяется на адаптивный темп обучения, который с течением времени уменьшается. Например:

$$\frac{c_1}{[\text{число итераций}] + c_2},$$

где c_1 и c_2 – это константы. Отметим, что стохастический градиентный спуск не достигает глобального минимума, но находится в области, очень близкой к нему. При помощи адаптивного темпа обучения мы можем достигнуть дальнейшего отжига[1] в сторону лучшего глобального минимума.

Еще одно преимущество стохастического градиентного спуска состоит в том, что мы можем использовать его для *динамического (онлайнового, инкрементного) обучения*. В динамическом обучении наша модель тренируется на лету по мере поступления новых тренировочных данных. Это особенно полезно, если мы накапливаем большие объемы данных – например, данные о клиентах в типичных веб-приложениях. Используя динамическое обучение, система может сразу адаптироваться к изменениям, и тренировочные данные могут быть отброшены после обновления модели, в случае если объем памяти ограничен.

Компромиссом между пакетным градиентным спуском и стохастическим градиентным спуском является так называемое мини-пакетное обучение. Мини-пакетное обучение может пониматься как применение пакетного градиентного спуска к подмножествам тренировочных данных меньшего объема – например, по 50 образцов за раз. Его преимущество перед пакетным градиентным спуском состоит в том, что благодаря мини-макетам сходимость достигается быстрее из-за более частых обновлений весов. Кроме того, мини-пакетное обучение позволяет произвести замену циклов `for` по тренировочным образцам в **стохастическом градиентном спуске** (СГС, SGD) на векторизованные операции, которые могут еще более повысить вычислительную эффективность нашего алгоритма обучения.

С учетом того, что мы уже реализовали правило обучения ADALINE на основе градиентного спуска, нам нужно будет внести всего несколько корректировок для модификации алгоритма обучения, связанных с обновлением весов методом стохас-

---

[1] Имеется в виду имитация отжига (simulated annealing) – общий алгоритмический метод аппроксимации глобального оптимума, похожий на градиентный спуск, но который, в отличие от него, за счет случайности выбора промежуточной точки реже попадает в локальные минимумы. – *Прим. перев.*

тического градиентного спуска. В методе fit мы теперь будем обновлять веса после каждого тренировочного образца. Кроме того, мы реализуем дополнительный метод частичной подгонки partial_fit, который повторно не инициализирует веса, чтобы учесть динамическое обучение. Для того чтобы удостовериться, что наш алгоритм сходился после тренировки, мы вычислим стоимость как усредненную стоимость тренировочных образцов в каждой эпохе. Кроме того, мы добавим опцию перемешивания shuffle тренировочных данных перед каждой эпохой для предотвращения зацикливания во время оптимизации функции стоимости; в параметре random_state мы в целях единообразия даем возможность задать случайное начальное число (random seed) генератора случайных чисел:

```
from numpy.random import seed

class AdalineSGD(object):
    """ Классификатор на основе ADALINE (ADAptive LInear Neuron).

    Параметры
    ---------
    eta : float
        Темп обучения (между 0.0 и 1.0)
    n_iter : int
        Проходы по тренировочному набору данных.

    Атрибуты
    --------
    w_ : 1-мерный массив
        Веса после подгонки.
    errors_ : list/список
        Число случаев ошибочной классификации в каждой эпохе.
    shuffle : bool (по умолчанию: True)
        Перемешивает тренировочные данные в каждой эпохе, если True,
        для предотвращения зацикливания.
    random_state : int (по умолчанию: None)
        Инициализирует генератор случайных чисел
        для перемешивания и инициализации весов.

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

    def fit(self, X, y):
        """ Выполнить подгонку под тренировочные данные.

        Параметры
        ---------
        X : { массивоподобный}, форма = [n_samples, n_features]
            Тренировочные векторы, где
            n_samples - число образцов и
            n_features - число признаков.
        y : массивоподобный, форма = [n_samples]
            Целевые значения.
```

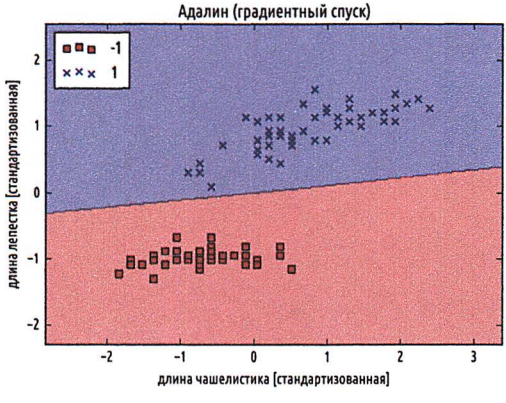
```
        Возвращает
        ----------
        self : объект

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

    def partial_fit(self, X, y):
        """Выполнить подгонку под тренировочные данные
           без повторной инициализации весов"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Перемешать тренировочные данные"""
        r = np.random.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Инициализировать веса нулями"""
        self.w_ = np.zeros(1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Применить обучающее правило ADALINE, чтобы обновить веса"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

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

    def activation(self, X):
        """Рассчитать линейную активацию"""
        return self.net_input(X)

    def predict(self, X):
        """Вернуть метку класса после единичного скачка"""
        return np.where(self.activation(X) >= 0.0, 1, -1)
```

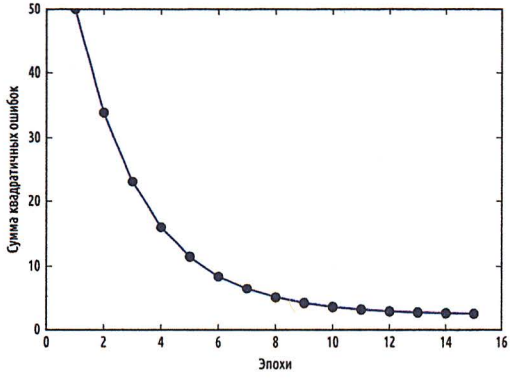
Метод _shuffle, который теперь используется в классификаторе AdalineSGD, работает следующим образом: функцией permutation библиотеки numpy.random мы генерируем случайную последовательность уникальных чисел в диапазоне от 0 до 100. Затем эти числа используются в качестве индексов для перемешивания нашей матрицы признаков и вектора меток классов.

Далее можем применить метод fit, чтобы натренировать классификатор AdalineSGD, и нашу функцию plot_decision_regions, чтобы построить график результатов тренировки:

```
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)
plot_decision_regions(X_std, y, classifier=ada)
plt.title('ADALINE (стохастический градиентный спуск)')
plt.xlabel('длина чашелистика [стандартизованная]')
plt.ylabel('длина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Эпохи')
plt.ylabel('Средняя стоимость')
plt.show()
```

Ниже приведены два графика, которые получены после выполнения предыдущего примера:





Как видно, средняя стоимость довольно быстро уходит вниз, а окончательная граница решения после 15 эпох выглядит аналогично пакетному градиентному спуску. Если нашу модель нужно обновить, например в сценарии динамического обучения с потоковой передачей данных, то мы можем просто вызывать метод partial_fit на индивидуальных образцах, к примеру ada.partial_fit(X_std[0, :], y[0]).

Резюме

В этой главе мы получили хорошее представление об основных принципах работы линейных классификаторов, используемых в обучении с учителем. После реализации персептрона мы познакомились с методами эффективной тренировки адаптивных линейных нейронов (ADALINE) векторизованной версией метода градиентного спуска и с динамическим обучением методом стохастического градиентного спуска. Научившись реализовывать простые классификаторы на Python, теперь мы готовы перейти к следующей главе, где воспользуемся библиотекой машинного обучения scikit-learn языка Python для доступа к более продвинутым и мощным стандартным машиннообучаемым классификаторам, которые обычно используются в научных кругах и в информационной отрасли.

Глава 3

Обзор классификаторов с использованием библиотеки scikit-learn

В этой главе мы пройдемся по массиву популярных и мощных алгоритмов машинного обучения, которые обычно используются в научных кругах и в информационной отрасли. По мере ознакомления с различиями между несколькими алгоритмами обучения с учителем для решения задачи классификации мы также получим интуитивное представление об их индивидуальных достоинствах и недостатках. Кроме того, мы сделаем наши первые шаги в работе с библиотекой scikit-learn, которая предлагает пользователю удобный интерфейс для эффективного и продуктивного использования этих алгоритмов.

В этой главе мы изучим следующие темы:

- введение в принципы работы популярных алгоритмов классификации;
- использование библиотеки машинного обучения scikit-learn;
- вопросы, которые следует задать при выборе алгоритма машинного обучения.

Выбор алгоритма классификации

Выбор надлежащего алгоритма классификации данных, который подходил бы для отдельно взятой задачи, требует практических навыков: каждый алгоритм имеет свои собственные особенности и основывается на определенных допущениях. Перефразируя теорему «Никаких бесплатных обедов»: в любых возможных сценариях ни один классификатор не работает лучше остальных. На практике всегда рекомендуется сравнить качество, по крайней мере, нескольких разных алгоритмов обучения, чтобы выбрать наилучшую модель для отдельно взятой задачи; алгоритмы могут отличаться по числу признаков либо образцов, уровню шума в наборе данных и по тому, являются классы линейно-разделимыми или нет.

В конечном счете качество классификатора, его вычислительная и предсказательная мощность в большой степени зависят от базовых данных, предназначенных для тренировки алгоритма. Во время тренировки алгоритма задействуются пять основных шагов, которые можно подытожить следующим образом:

1) отбор признаков;
2) выбор качественной метрики;
3) выбор классификатора и алгоритма оптимизации;
4) оценка качества модели;
5) тонкая настройка алгоритма.

Поскольку принятый в книге подход состоит в постепенном накоплении знаний в области машинного обучения, в этой главе мы в основном сосредоточимся на основных принципах работы разных алгоритмов и в дальнейшем в целях более детального обсуждения будем пересматривать такие темы, как, например, отбор и предобработка признаков, качественной метрики и тонкая настройка гиперпараметров.

Первые шаги в работе с scikit-learn

В *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* вы узнали о двух связанных между собой алгоритмах обучения для выполнения задачи классификации: правиле **персептрона** и алгоритме адаптивных линейных нейронов (**ADALINE**), – которые мы собственноручно реализовали на Python. Теперь мы рассмотрим программный интерфейс (API) библиотеки scikit-learn, который сочетает удобные для пользователя средства взаимодействия с высоко оптимизированной реализацией нескольких алгоритмов классификации. Вместе с тем библиотека scikit-learn предлагает не только большое разнообразие алгоритмов обучения, но и большое количество удобных функций для предобработки данных в целях тонкой настройки и оценки моделей. Мы обсудим этот вопрос более подробно вместе с базовыми понятиями в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»* и *главе 5 «Сжатие данных путем снижения размерности»*.

Тренировка персептрона в scikit-learn

Чтобы приступить к работе с библиотекой scikit-learn, сначала натренируем персептронную модель, аналогичную той, которую мы реализовали в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*. Для простоты на протяжении всех последующих разделов мы будем использовать уже знакомый нам набор данных **цветков ириса** (Iris). Удобным образом этот набор данных уже имеется в библиотеке scikit-learn, поскольку этот простой и одновременно популярный набор данных часто используется для тестирования алгоритмов и проведения с ним экспериментов. Кроме того, в целях визуализации мы воспользуемся из этого набора данных всего двумя признаками цветков.

Мы присвоим матрице признаков `X` *длину* и *ширину лепестков* из 150 образцов цветков, а вектору `y` – метки классов, которые соответствуют видам цветков:

```
from sklearn import datasets
import numpy as np
iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
```

Если выполнить оператор `np.unique(y)`, который возвращает отдельные метки классов, хранящихся в `iris.target`, то мы увидим, что имена классов цветков *ирис щетинистый* (*Iris setosa*), *ирис виргинский* (*Iris virginica*) и *ирис разноцветный* (*Iris versicolor*) представлены в виде целых чисел (`0`, `1`, `2`), что во многих библиотеках машинного обучения рекомендуется делать в целях оптимальной производительности.

Чтобы оценить, насколько хорошо натренированная модель работает на ранее не встречавшихся ей данных, мы, в свою очередь, разделим набор данных на тренировочный и тестовый наборы. Позже, в *главе 5 «Сжатие данных путем снижения размерности»*, мы более подробно обсудим наиболее успешные методы оценки моделей:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
                                   X, y, test_size=0.3, random_state=0)
```

При помощи функции train_test_split из модуля model_selection библиотеки scikit-learn (в версии библиотеки < 0.18 этот модуль назывался cross_validation) мы произвольным образом разделяем массивы X и y на тестовые данные в размере 30% от общего объема (45 образцов) и тренировочные данные в размере 70% (105 образцов).

Как мы помним из примера с **градиентным спуском** в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, многие алгоритмы машинного обучения и оптимизации в целях улучшения качества также требуют выполнения масштабирования признаков. Здесь мы выполним стандартизацию признаков, воспользовавшись для этого классом для выполнения стандартизации StandardScaler из модуля preprocessing библиотеки scikit-learn:

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

В приведенном выше примера мы загрузили класс StandardScaler из модуля предобработки и инициализировали новый объект StandardScaler, который мы присвоили переменной sc. Затем мы воспользовались методом fit объекта StandardScaler, чтобы вычислить параметры μ (эмпирическое среднее) и σ (стандартное отклонение) для каждой размерности признаков из тренировочных данных. Вызвав метод transform, мы затем стандартизировали тренировочные данные, используя для этого расчетные параметры μ и σ. Отметим, что для стандартизации тестового набора мы использовали те же самые параметры масштабирования, благодаря чему значения в тренировочном и тестовом наборах между собой сопоставимы.

Стандартизировав тренировочные данные, теперь можно натренировать персептронную модель. Большинство алгоритмов в библиотеке scikit-learn поддерживает многоклассовую классификацию уже по умолчанию благодаря методу **один против остальных** (One-vs.-Rest, **OvR**), который позволяет передать в персептрон сразу все три класса цветков. Исходный код выглядит следующим образом:

```
from sklearn.linear_model import Perceptron
ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)
ppn.fit(X_train_std, y_train)
```

Интерфейс библиотеки scikit-learn выглядит аналогично нашей реализации персептрона в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*: после загрузки класса Perceptron из модуля linear_model мы инициализировали новый объект Perceptron и натренировали модель при помощи метода fit. Модельный параметр eta0 здесь эквивалентен темпу обучения eta, который мы

использовали в нашей собственной реализации персептрона, и параметр n_iter задает число эпох (проходов по тренировочному набору). Как мы помним из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, для нахождения надлежащего темпа обучения требуется немного поэкспериментировать. Если темп обучения слишком большой, то алгоритм промахнется по глобальному минимуму стоимости. Если темп обучения слишком малый, то для достижения сходимости алгоритм потребует большего количества эпох, что может замедлить обучение – в особенности для больших наборов данных. Кроме того, мы использовали параметр random_state для воспроизводимости исходного перемешивания тренировочного набора данных после каждой эпохи.

Натренировав модель в scikit-learn, мы можем приступить к выполнению прогнозов, используя для этого метод predict, точно так же, как в нашей собственной реализации персептрона в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*. Соответствующий исходный код выглядит следующим образом:

```
>>>
y_pred = ppn.predict(X_test_std)
print('Число ошибочно классифицированных образцов: %d' % (y_test != y_pred).sum())
Число ошибочно классифицированных образцов: 4
```

После выполнения приведенного выше примера мы увидим, что персептрон ошибочно классифицирует 4 из 45 образцов цветков. Таким образом, ошибка классификации на тестовом наборе данных составляет 0.089, или 8.9% ($4/45 \approx 0.089$).

Вместо ошибки классификации многие практикующие специалисты в области машинного обучения сообщают о метрике **верности** (accuracy) классификации модели, которая легко рассчитывается следующим образом:
1 – *ошибка классификации* = 0.911, или 91.1%.

В библиотеке scikit-learn также реализовано большое разнообразие различных метрик оценки качества работы, которые доступны благодаря модулю metrics. Например, мы можем вычислить оценку верности классификации с использованием персептрона на тестовом наборе следующим образом:

```
>>>
from sklearn.metrics import accuracy_score
print('Верность: %.2f' % accuracy_score(y_test, y_pred))
Верность: 0.91
```

Здесь y_test – это истинные метки классов, y_pred – метки классов, которые мы идентифицировали ранее.

Отметим, что в этой главе мы оцениваем качество наших моделей, основываясь на тестовом наборе. В *главе 5 «Сжатие данных путем снижения размерности»* вы узнаете о полезных методах, включая графический анализ, такой как кривые обучения, чтобы обнаруживать и предотвращать **переобучение**. Термин «переобучение» означает, что модель хорошо захватывает повторяющиеся образы (закономерности, паттерны) в тренировочных данных, но ей не удается хорошо обобщаться на ранее не встречавшиеся ей данные.

В заключение можно применить нашу функцию plot_decision_regions из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, чтобы по-

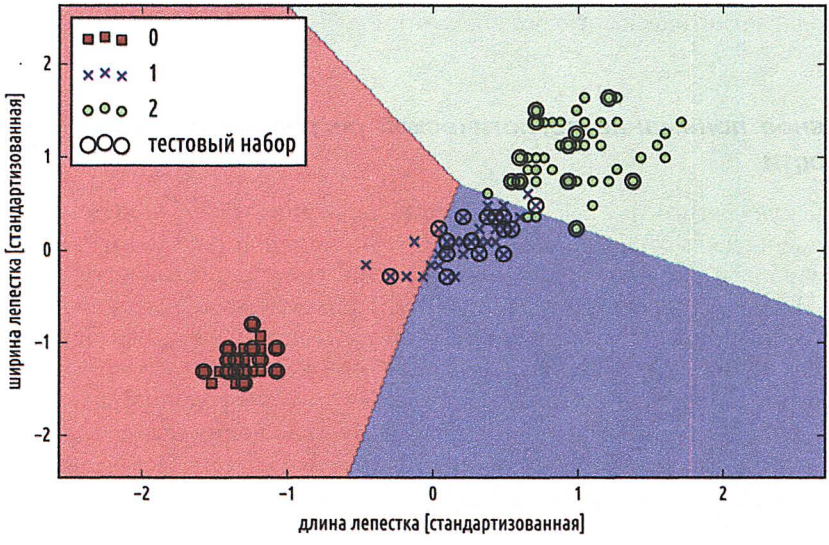
строить график **областей решений** нашей недавно натренированной персептронной модели и в наглядной форме представить, насколько хорошо она разделяет образцы цветков разных видов. Впрочем, немного видоизменим ее, чтобы выделить образцы из тестового набора данных кругами:

```
from matplotlib.colors import ListedColormap

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

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

    # вывести поверхность решения
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # показать все образцы
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)
        # выделить тестовые образцы
        if test_idx:
            X_test, y_test = X[test_idx, :], y[test_idx]
            plt.scatter(X_test[:, 0], X_test[:, 1], c='',
                        alpha=1.0, linewidths=1, marker='o',
                        s=55, label='тестовый набор')
```

Добавив в функцию plot_decision_regions небольшое изменение (в приведенном выше примере выделено жирным шрифтом), теперь можно указывать индексы образцов, которые мы хотим пометить на получающихся графиках. Соответствующий исходный код выглядит следующим образом:

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

Как видно на получившемся графике, полностью разделить три класса цветков линейной границей решения не получается:



Из нашего обсуждения в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* мы помним, что алгоритм обучения персептрона никогда не сходится на наборах данных, которые не полностью линейно разделимы, и поэтому этот алгоритм обычно не рекомендуется применять на практике. В следующих разделах мы рассмотрим более мощные линейные классификаторы, которые сходятся к минимуму стоимости, даже если классы не полностью линейно разделимы.

Класс `Perceptron`, а также другие функции и классы библиотеки scikit-learn имеют дополнительные параметры, которые мы для ясности пропускаем. Дополнительную информацию об этих параметрах можно прочесть, воспользовавшись функцией Python `help` (например, `help(Perceptron)`) или просмотрев превосходную онлайн-документацию по библиотеке scikit-learn на `http://scikit-learn.org/stable/`.

Моделирование вероятностей классов логистической регрессии

Несмотря на то что правило персептрона предлагает простое и удобное введение в алгоритмы машинного обучения для классификации данных, самый большой недостаток правила персептрона заключается в том, что оно не обеспечивает сходимости в случае, если классы не полностью линейно разделимы. Рассмотренная в предыдущем разделе задача классификации являлась примером такого сценария. Интуитивно мы можем аргументировать это тем, что веса постоянно обновляются из-за того, что в каждой эпохе всегда существует, по крайней мере, один ошибочно классифицированный образец. Разумеется, можно изменить темп обучения и увеличить число эпох, но предупреждаем – персептрон на этом наборе данных никогда не сойдется. Ради лучшего применения нашего времени теперь взглянем на другой простой и одновременно более мощный алгоритм для задач линейной и бинарной классификации: **логистическую регрессию**. Отметим, что, несмотря на название

этого метода, логистическая регрессия – это модель для задачи классификации, а не регрессии.

Интуитивное понимание логистической регрессии и условные вероятности

Логистическую регрессию как модель классификации очень просто реализовать, и одновременно она очень хорошо работает на линейно разделимых классах. Это один из наиболее широко используемых алгоритмов для классификации данных в отрасли. Подобно персептрону и ADALINE, логистическая регрессионная модель в этой главе представляет собой линейную модель бинарной классификации, которую методом OvR можно тоже расширить на многоклассовую классификацию.

Для того чтобы объяснить лежащую в основе логистической регрессии идею как вероятностную модель, сначала введем формулу **отношения шансов** (ОШ, odds ratio, OR)[1], т. е. шансов в пользу отдельно взятого события. Отношение шансов можно записать как $\frac{p}{1-p}$, где p обозначает вероятность положительного события. Термин *положительное событие* не обязательно означает *хорошее* событие; оно относится к событиям, которые мы хотим предсказать, например вероятность, что у пациента есть определенное заболевание; положительное событие можно представить как метку класса $y = \mathbf{1}$. Затем, конкретизируя, мы можем определить функцию **логит** (logit), т. е. просто логарифм отношения шансов (или логарифм перевесов log-odds)[2]:

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

Функция логит принимает входные значения в диапазоне от 0 до 1 и трансформирует их в значения по всему диапазону вещественного числа, которые можно использовать для выражения линейной связи между значениями признаков и логарифмами отношения шансов:

$$logit\big(p(y=1\,|\,x)\big) = w_0x_0 + w_1x_1 + \ldots + w_mx_m = \sum_{i=0}^{m} w_ix_i = \boldsymbol{w}^T\boldsymbol{x}.$$

Здесь $p(y = 1|x)$ – это условная вероятность, что отдельно взятый образец принадлежит классу 1 при наличии его признаков x.

Далее нас на самом деле интересует предсказание вероятности, что определенный образец принадлежит отдельно взятому классу, т. е. обратная форма функции логит. Ее также называют *логистической* функцией, иногда просто сокращенно *сигмоидой*, или сигмоидальной функцией, из-за ее характерной формы в виде латинской буквы S.

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

---

[1] Отношение шансов – отношение вероятности наступления интересующего события к вероятности его ненаступления. – *Прим. перев.*

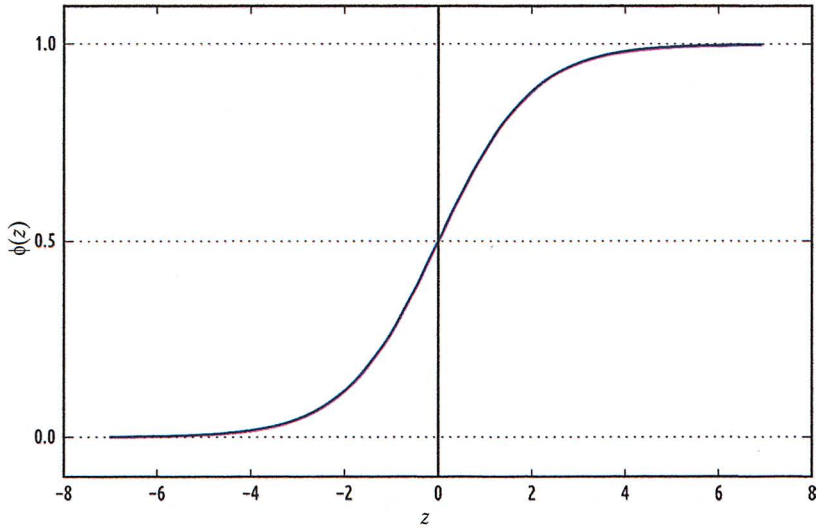
[2] Практически здесь чаще всего используются натуральные логарифмы с основанием *e*. – *Прим. перев.*

Здесь z – это чистый вход, т. е. линейная комбинация весов и признаков в образце, которую можно вычислить как $z = \boldsymbol{w}^T\boldsymbol{x} = w_0x_0 + w_1x_1 + ... + w_mx_m$.

А теперь построим график сигмоидальной функции для нескольких значений в диапазоне от –7 до +7, чтобы посмотреть, как она выглядит:

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

def sigmoid(z):
    return 1.0 / (1.0 + np.exp(-z))

z = np.arange(-7, 7, 0.1)
phi_z = sigmoid(z)
plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.axhspan(0.0, 1.0, facecolor='1.0', alpha=1.0, ls='dotted')
plt.axhline(y=0.5, ls='dotted', color='k')
plt.yticks([0.0, 0.5, 1.0])
plt.ylim(-0.1, 1.1)
plt.xlabel('z')
plt.ylabel('$\phi (z)$')
plt.show()
```

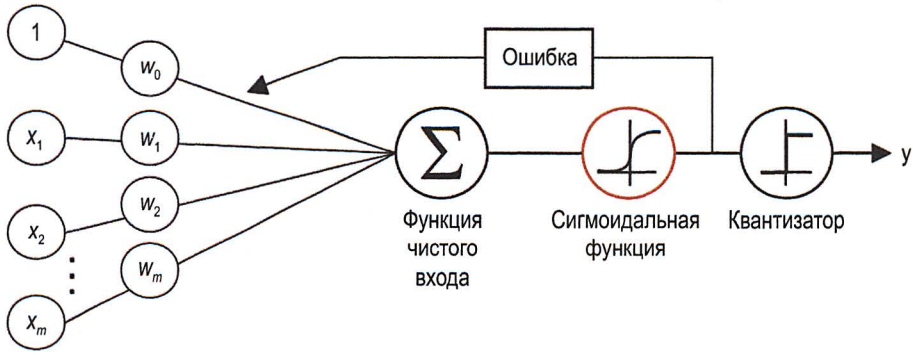
В результате выполнения приведенного выше примера мы должны увидеть **S-образную** (сигмоидальную) кривую:



Мы видим, что $\phi(z)$ приближается к 1, если z стремится к бесконечности ($z \to \infty$), поскольку e^{-z} становится очень малым для больших значений z. Аналогичным образом $\phi(z)$ стремится к 0 для $z \to -\infty$ в результате все более увеличивающегося знаменателя. Таким образом, мы приходим к заключению, что сигмоидальная функция на входе принимает вещественные значения и трансформирует их в значения в диапазоне [0, 1] с точкой пересечения в $\phi(z) = 0.5$.

Для формирования некоторого интуитивного понимания модели логистической регрессии мы можем связать ее с нашей ранней реализацией ADALINE

в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*. В ADALINE мы в качестве функции активации использовали тождественное отображение $\phi(z) = z$. В логистической регрессии эта функция активации просто становится сигмоидальной функцией, которую мы определили ранее, как проиллюстрировано на следующем рисунке:



Выход из сигмоидальной функции затем интерпретируется как вероятность принадлежности отдельно взятого образца классу 1 $\phi(z) = P(y = 1|\boldsymbol{x};\boldsymbol{w})$ при наличии его признаков $\boldsymbol{x}$, параметризованных весами $\boldsymbol{w}$. Например, если для отдельно взятого образца цветков мы вычисляем $\phi(z) = 0.8$, то это означает, что шанс, что этот образец является цветком ирис разноцветный, составляет 80%. Аналогичным образом вероятность, что это цветок ириса щетинистого, можно вычислить как $P(y = 0|\boldsymbol{x};\boldsymbol{w}) = 1 - P(y = 1|\boldsymbol{x};\boldsymbol{w}) = 0.2$, или 20%. Предсказанную вероятность затем можно просто конвертировать квантизатором (единичной ступенчатой функцией) в бинарный результат:

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

Если посмотреть на предыдущий график сигмоиды, то он эквивалентен следующему:

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

На деле во многих приложениях нас интересует не только идентификация меток классов, но и оценка вероятности принадлежности классу, что в некоторых случаях бывает особенно целесообразным. К примеру, логистическая регрессия используется в погодном прогнозировании, чтобы не только предсказывать, будет ли идти дождь в отдельно взятый день, но и сообщать о шансе дождя. Подобным образом логистическая регрессия может использоваться для предсказания шанса, что у пациента есть отдельно взятое заболевание при наличии определенных симптомов; вот почему логистическая регрессия имеет широкую популярность в сфере медицины.

Извлечение весов логистической функции стоимости

Вы узнали, каким образом можно использовать модель логистической регрессии для прогнозирования вероятностей и идентификации меток классов. Теперь вкратце остановимся на параметрах модели, например весах w. В предыдущей главе мы определили функцию стоимости как сумму квадратичных ошибок (SSE):

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

Мы ее минимизировали, чтобы извлечь веса w, используемые в модели классификации на основе ADALINE. Чтобы объяснить, каким образом можно получить функцию стоимости для логистической регрессии, сначала определим правдоподобие L, которое мы хотим максимизировать при построении модели логистической регрессии, принимая, что отдельные образцы в нашем наборе данных независимы друг от друга[1]. Данная формула выглядит следующим образом:

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

На практике проще максимизировать (натуральный) логарифм этого уравнения, т. е. так называемую логарифмическую функцию правдоподобия:

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

Во-первых, применение логарифмической функции снижает потенциальную возможность числовой потери значимости, которая может произойти, если величины правдоподобия принимают слишком малые значения. Во-вторых, произведение факторов можно свести к сумме факторов, что упрощает получение производной этой функции посредством трюка со сложением, как вы, возможно, помните из алгебры[2].

Теперь можно применить алгоритм оптимизации, такой как, например, градиентный подъем[3], для максимизации этой логарифмической функции правдоподобия[4].

---

[1] Правдоподобие (likelihood), или функция правдоподобия, – это совместное распределение образца из параметрического распределения, рассматриваемое как функция параметра. При этом используется совместная функция плотности (в случае образца из непрерывного распределения) либо совместная вероятность (в случае образца из дискретного распределения), вычисленные для значений выборочных данных. – *Прим. перев.*

[2] Во избежание проблемы арифметического переполнения снизу (приводящего к исчезновению разрядов) при работе с числами с плавающей точкой, находящимися слишком близко к нулю, пользуются свойствами логарифмов: $\log ab = \log a + \log b$ и $\exp(\log[(x) = x])$, которые позволяют произведение $p_1 * \ldots * p_n$ представить в виде эквивалентной суммы логарифмов: $\exp(\log(p_1) + \ldots + \log(p_n)$. – *Прим. перев.*

[3] Градиентный (наискорейший) подъем (gradient ascent) – алгоритм градиентного подъема, инкрементный алгоритм оптимизации, или поиска оптимального решения, где приближение к локальному максимуму функции идет шагами, пропорциональными величине градиента этой функции в текущей точке. – *Прим. перев.*

[4] Речь о методе под названием «оценка максимального правдоподобия». Это метод оценивания неизвестного параметра путем максимизации функции вероятности, в результате которой приобретаются значения параметров модели, которые делают данные «ближе» к реальным. – *Прим. перев.*

Как вариант перепишем логарифмическое правдоподобие как функцию стоимости J, которую можно минимизировать при помощи градиентного спуска, как в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*:

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

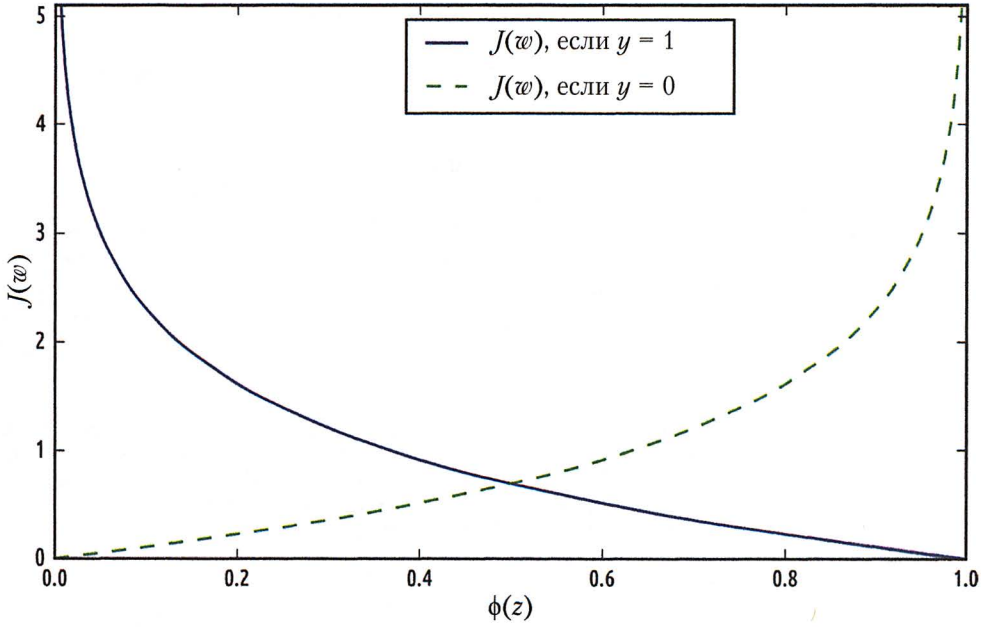
Чтобы глубже вникнуть в эту функцию стоимости, обратимся к стоимости, которая рассчитывается для одного точечного экземпляра:

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

Анализируя приведенное уравнение, видим, что первый член обнуляется при $y = \mathbf{0}$, второй – соответственно при $y = \mathbf{1}$:

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

Следующий ниже график иллюстрирует стоимость во время классификации точечного экземпляра для различных значений $\phi(z)$:



Мы видим, что стоимость приближается к 0 (сплошная линия), в случае если мы правильно предсказываем, что образец принадлежит классу 1. Подобным же образом на оси Y мы видим, что стоимость также приближается к 0, в случае если мы правильно предсказываем $y = 0$ (пунктирная линия). Вместе с тем в случае, если прогноз является ошибочным, стоимость стремится к бесконечности. Мораль заключается в том, что мы штрафуем неправильные прогнозы все большей стоимостью.

Тренировка логистической регрессионной модели в scikit-learn

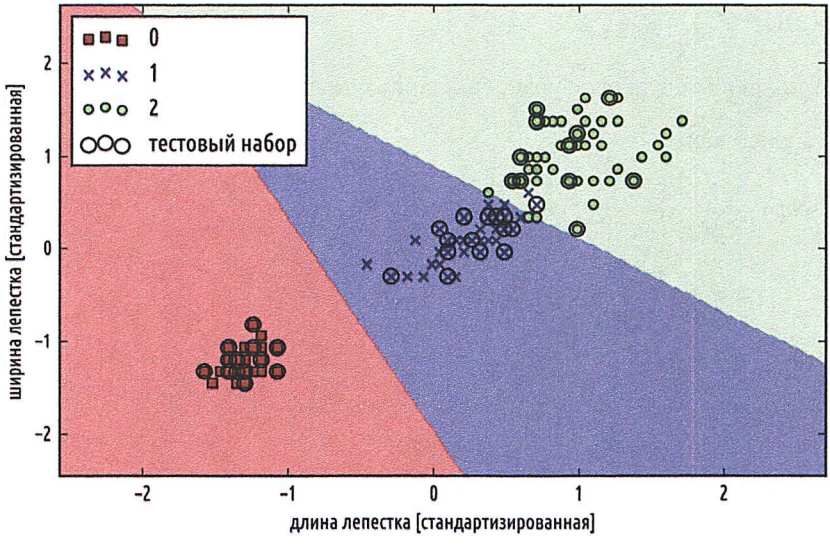
Если нужно реализовать логистическую регрессию самостоятельно, то можно просто заменить функцию стоимости J в нашей реализации ADALINE из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* на новую функцию стоимости:

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

Она вычислит стоимость классифицирования всех тренировочных образцов в расчете на эпоху, и мы в итоге получим рабочую модель логистической регрессии. Однако, учитывая, что в библиотеке scikit-learn реализована высокооптимизированная версия логистической регрессии, которая к тому же поддерживает готовую многоклассовую конфигурацию, мы не станем ее реализовывать и воспользуемся классом sklearn.linear_model.LogisticRegression, а также знакомым методом fit для тренировки модели на стандартизированном тренировочном наборе данных цветков ириса:

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=1000.0, random_state=0)
lr.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=lr,
                      test_idx=range(105,150))
plt.xlabel('длина лепестка [стандартизированная]')
plt.ylabel('ширина лепестка [стандартизированная]')
plt.legend(loc='upper left')
plt.show()
```

После подгонки модели на тренировочных данных мы построили график областей решения и вывели на графике тренировочные и тестовые образцы, как показано ниже:



Глядя на приведенный выше пример исходного кода, который использовался для тренировки модели LogisticRegression, вам, наверное, теперь интересно, что это за загадочный параметр C. Мы разберемся в нем через пару секунд, а пока в следующем подразделе сначала кратко остановимися на понятии переобучения и регуляризации.

В свою очередь, мы можем предсказывать вероятность принадлежности образцов классам при помощи метода predict_proba. Например, можно предсказать вероятности первого образца ириса щетинистого:

```
lr.predict_proba(X_test_std[0,:])
```

Эта команда возвращает следующий массив:

```
array([[ 0.000, 0.063, 0.937]])
```

Приведенный выше массив говорит о том, что модель предсказывает с шансом 93.7% принадлежность образца классу ириса виргинского, и с шансом 6.3%, что образец является цветком ириса разноцветного.

Можно продемонстрировать, что обновление веса в логистической регрессии методом градиентного спуска действительно эквивалентно уравнению, которое мы использовали в ADALINE в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*. Начнем с расчета частной производной логарифмической функции правдоподобия относительно j-го веса:

$$\frac{\delta}{\delta w_j} l(\boldsymbol{w}) = \left(y \frac{1}{\phi(z)} - (1-y) \frac{1}{1-\phi(z)} \right) \frac{\delta}{\delta w_j} \phi(z).$$

Прежде чем продолжить, сначала вычислим частную производную сигмоидальной функции:

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

Теперь можно в наше первое уравнение повторно подставить $\frac{\delta}{\delta z}\phi(z) = \phi(z)(1-\phi(z))$ и получить следующее:

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

Напомним, что задача – найти веса, которые максимизируют логарифмическое правдоподобие, с тем чтобы выполнить обновление по каждому весу, как показано ниже:

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

Учитывая, что мы обновляем все веса одновременно, общее правило обновления можно записать следующим образом:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}.$$

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

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

Поскольку максимизация логарифмического правдоподобия эквивалентна минимизации определенной ранее функции стоимости J, мы можем записать правило обновления на основе градиентного спуска следующим образом:

$$\Delta w_j = -\eta \frac{\delta J}{\delta w_j} = \eta \sum_{i=1}^{n} \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)};$$

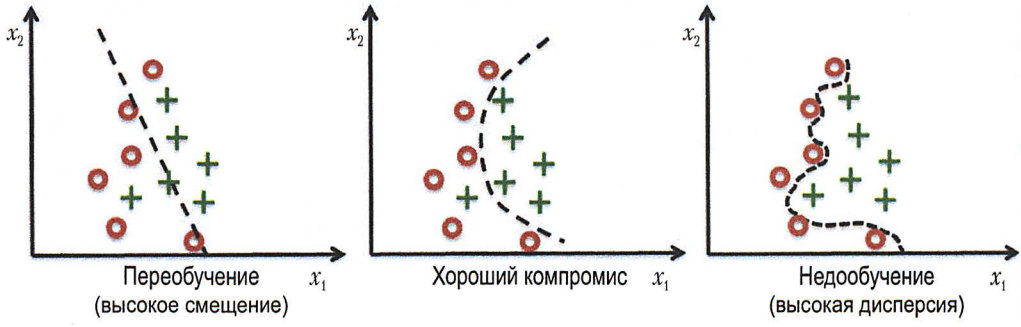
$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}, \quad \Delta \boldsymbol{w} := -\eta \nabla J(\boldsymbol{w}).$$

Таким образом, оно эквивалентно правилу на основе градиентного спуска в адалине из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*.

Решение проблемы переобучения при помощи регуляризации

В машинном обучении **переобучения** представляет собой типичную проблему, проявляющуюся в том, что модель хорошо работает на тренировочных данных, но недостаточно хорошо обобщается на ранее не встречавшихся ей (тестовых) данных. Если модель страдает от переобучения, то мы также говорим, что модель имеет высокую дисперсию, которая может быть вызвана наличием слишком большого числа параметров, приводящих к слишком сложной модели в условиях базовых данных. Аналогичным образом модель может также страдать от **недообучения** (высокого смещения), т. е. модель недостаточно сложна, чтобы хорошо захватывать повторяющиеся в тренировочных данных образы, и, следовательно, также страдает от низкого качества на ранее не встречавшихся ей данных.

Хотя мы до сих пор имели дело только с линейными моделями классификации данных, проблему переобучения и недообучения лучше всего проиллюстрировать при помощи более сложной, нелинейной границы решения, как показано на нижеследующем рисунке:



Дисперсия измеряет постоянство (или изменчивость) модельного прогноза для отдельно взятого экземпляра при многократном переобучении модели, например на разных подмножествах тренировочного набора данных. При этом мы можем говорить, что модель чувствительна к случайности в тренировочных данных. В отличие от дисперсии, смещение измеряет, как далеко прогнозы находятся от правильных значений в целом при многократной перестройке модели на разных тренировочных наборах данных; смещение является мерой систематической ошибки, которая не происходит в силу случайности.

Один из способов найти хороший компромисс между смещением и дисперсией состоит в том, чтобы настроить сложность модели посредством регуляризации. Регуляризация – очень полезный метод для обработки коллинеарности (высокой корреляции среди признаков), фильтрации шума из данных и в конечном счете предотвращения переобучения. В основе регуляризации лежит идея внесения дополнительной информации (смещения) для наложения штрафа на экстремальные веса параметров. Самой стандартной формой регуляризации является так называемая **L2-регуляризация** (иногда также именуемая L2-стягиванием, или сокращением (затуханием), весов, которую можно записать следующим образом:

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

Здесь λ – это так называемый параметр регуляризации лямбда.

Регуляризация является еще одним аргументом в пользу важности масштабирования признаков, такого как, например, стандартизация. Чтобы регуляризация работала должным образом, мы должны обеспечить, чтобы все наши признаки находились в сопоставимых весах.

Для того чтобы применить регуляризацию, нужно в формулу функции стоимости, которую мы определили для логистической регрессии, просто добавить еще одно слагаемое – член регуляризации, который стягивает веса:

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

Благодаря параметру регуляризации λ мы получаем возможность управлять качеством выполняемой подгонки под тренировочные данные, путем удержания весов в малых значениях. Повышая величину λ, мы увеличиваем силу регуляризации.

Параметр C, который в библиотеке scikit-learn реализован для класса LogisticRegression, происходит из формы записи, принятой в методах опорных векторов, которые будут темой следующего раздела. Этот параметр непосредственно связан с параметром регуляризации λ и является его инверсией:

$$C = \frac{1}{\lambda}.$$

Таким образом, мы можем переписать регуляризованную функцию стоимости логистической регрессии следующим образом:

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

Следовательно, снижая величину обратного параметра регуляризации C, мы тем самым увеличиваем силу регуляризации, в чем можно наглядно убедиться, построив график траектории L2-регуляризации для двух весовых коэффициентов:

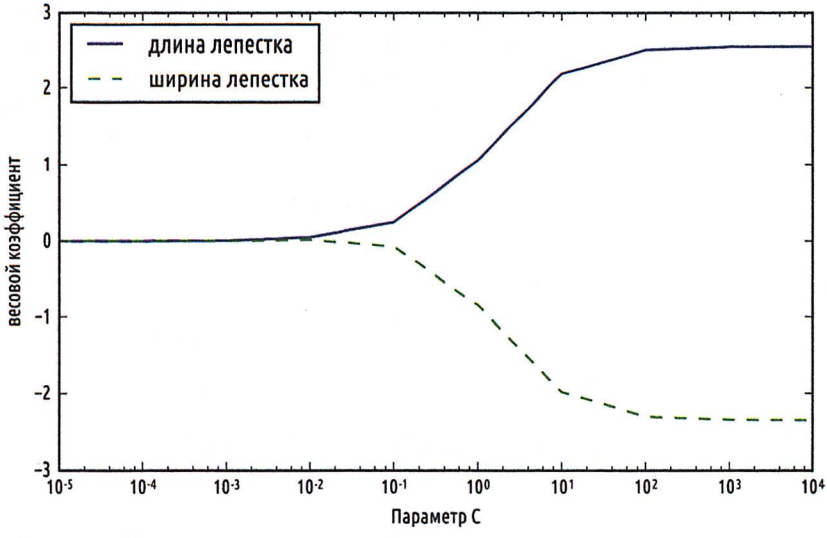
```
weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10**c, random_state=0)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10**c)
weights = np.array(weights)
plt.plot(params, weights[:, 0], label='длина лепестка')
plt.plot(params, weights[:,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

Выполнив приведенный выше исходный код, мы осуществили подгонку десяти моделей логистической регрессии с различными значениями для параметра обратной регуляризации C. В целях иллюстрации мы собрали весовые коэффициенты только класса 2[1], по сравнению со всем классификатором. Напомним, что мы используем метод OvR (один против остальных), используемый при многоклассовой классификации.

Как видно на получившемся графике, весовые коэффициенты сокращаются, если уменьшить параметр **C**, т. е. если увеличить силу регуляризации:

---

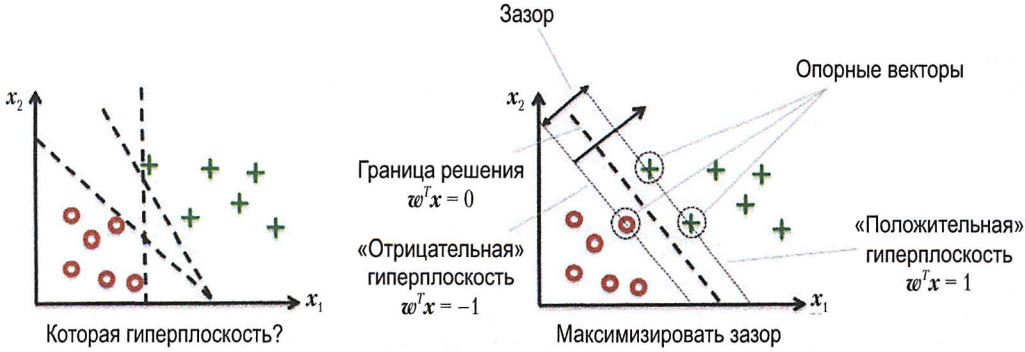
[1] Учитывая, что метки классов в наборе данных iris в библиотеке scikit-learn равны {0, 1, 2}, этот класс должен быть 2-м в наборе отсортированных в восходящем порядке данных, обозначая метку класса 1. – *Прим. перев.*



Поскольку всестороннее освещение отдельных алгоритмов классификации выходит за рамки этой книги, настоятельно рекомендуем читателям, которые хотят узнать о логистической регрессии больше, обратиться к книге доктора Скотта Менарда «*Логистическая регрессия: от вводного курса до продвинутых концепций и приложений*» (Dr. Scott Menard, Logistic Regression: From Introductory to Advanced Concepts and Applications) издательства «Sage».

Классификация с максимальным зазором на основе метода опорных векторов

Еще один мощный и широко используемый алгоритм обучения представлен **метод опорных векторов**, или сетью опорных векторов (support vector machine, SVM), причем этот метод можно рассматривать в качестве расширения персептрона. Используя алгоритм персептрона, мы минимизировали ошибки классификации. С другой стороны, в SVM наша задача оптимизации состоит в том, чтобы максимизировать **зазор**. Зазор определяется как расстояние между разделяющей гиперплоскостью (границей решения) и самыми близкими к этой гиперплоскости тренировочными образцами, так называемыми **опорными векторами**. Это проиллюстрировано на нижеследующем рисунке:



Интуитивное понимание максимального зазора

Обоснованием наличия границ решения с большитит зазорами (полосами), является то, что такие модели, как правило, имеют более низкую ошибку обобщения, тогда как модели с малым зазором более подвержены переобучения. Для получения интуитивного понимания максимизации зазора приглядимся повнимательнее к *положительной* и *отрицательной* гиперплоскостям, которые параллельны границе решения, что можно выразить следующим образом:

$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{\text{pos}} = 1; \quad (1)$$
$$w_0 + \boldsymbol{w}^T \boldsymbol{x}_{\text{neg}} = -1. \quad (2)$$

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

$$\Rightarrow \boldsymbol{w}^T(\boldsymbol{x}_{\text{pos}} - \boldsymbol{x}_{\text{neg}}) = 2.$$

Можно выполнить его нормализацию по длине (величине) вектора $\boldsymbol{w}$, которая определяется следующим образом:

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

Таким образом, мы приходим к следующему уравнению:

$$\frac{\boldsymbol{w}^T(\boldsymbol{x}_{\text{pos}} - \boldsymbol{x}_{\text{neg}})}{\|\boldsymbol{w}\|} = \frac{2}{\|\boldsymbol{w}\|}.$$

Тогда левую часть предыдущего уравнения можно интерпретировать как расстояние между положительной и отрицательной гиперплоскостями, т. е. ту самую маржу, которую мы хотим максимизировать.

Теперь целевая функция SVM сводится к максимизации этого зазаора путем максимизации $\frac{2}{\|\boldsymbol{w}\|}$ с учетом ограничения, что образцы классифицированы правильно. Это можно записать следующим образом:

$$\boldsymbol{w}_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} \geq 1, \text{ если } y^{(i)} = 1;$$
$$\boldsymbol{w}_0 + \boldsymbol{w}^T \boldsymbol{x}^{(i)} < -1, \text{ если } y^{(i)} = -1.$$

Эти два уравнения, в сущности, говорят о том, что все отрицательные образцы должны попасть на одну сторону от отрицательной гиперплоскости, тогда как все положительные образцы должны остаться за положительной гиперплоскостью. Это можно записать более сжато следующим образом:

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

Тем не менее на практике проще минимизировать обратный член $\frac{1}{2}\|\boldsymbol{w}\|^2$, который можно решить квадратичным программированием. Впрочем, подробное обсуждение квадратичного программирования выходит за рамки этой книги, однако если вы интересуетесь, то можете узнать о **методе опорных векторов** (SVM) гораздо больше из книги автора этой методики Владимира Вапника «*Природа теории статистического обучения*» (The Nature of Statistical Learning Theory) издательства «*Springer Science & Business Media*» или из превосходного объяснения Криса Дж. С. Бурга (Chris J. C. Burge) в «*Руководстве по методам опорных векторов для распознавания образов*» (A Tutorial on Support Vector Machines for Pattern Recognition) в сборнике «*Анализ данных и обнаружение знаний*» (Data mining and knowledge discovery), 2 (2):121-167, 1998.

Обработка нелинейно разделимого случая при помощи ослабленных переменных

Хотя мы не собираемся уходить еще глубже в более изощренные математические концепции, которые лежат в основе данной классификации, все же вкратце упомянем ослабленную (slack) переменную ξ. Она была введена Владимиром Вапником в 1995 г. и привела к так называемой классификации с мягким зазором. Мотивация для введения ослабленной переменной ξ состояла в том, что линейные ограничения должны быть ослаблены для нелинейно разделимых данных, с тем чтобы разрешить сходимость оптимизации с участием случаев ошибочной классификации в условиях надлежащего штрафования стоимости.

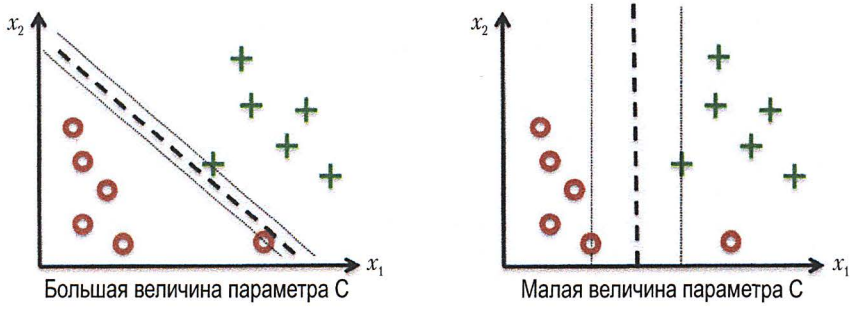
Ослабленная переменная с положительными значениями просто добавляется к линейным ограничениям:

$$\boldsymbol{w}^T\boldsymbol{x}^{(i)} \geq 1 - \xi^{(i)}, \text{ если } y^{(i)} = 1;$$
$$\boldsymbol{w}^T\boldsymbol{x}^{(i)} \leq -1 + \xi^{(i)}, \text{ если } y^{(i)} = -1.$$

Поэтому новой целевой функцией, подлежащей минимизации (и подчиняющейся предыдущим ограничениям), становится:

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

Используя переменную C, затем можно управлять штрафом за ошибочную классификацию. Большие значения переменной C соответствуют штрафам за большие ошибки классификации, тогда как выбор ее менее высоких значений обозначает меньшую строгость в отношении этих ошибок. Затем параметр C можно использовать для управления шириной зазора и, следовательно, настройки компромисса между смещением и дисперсией, как проиллюстрировано на нижеследующем рисунке:

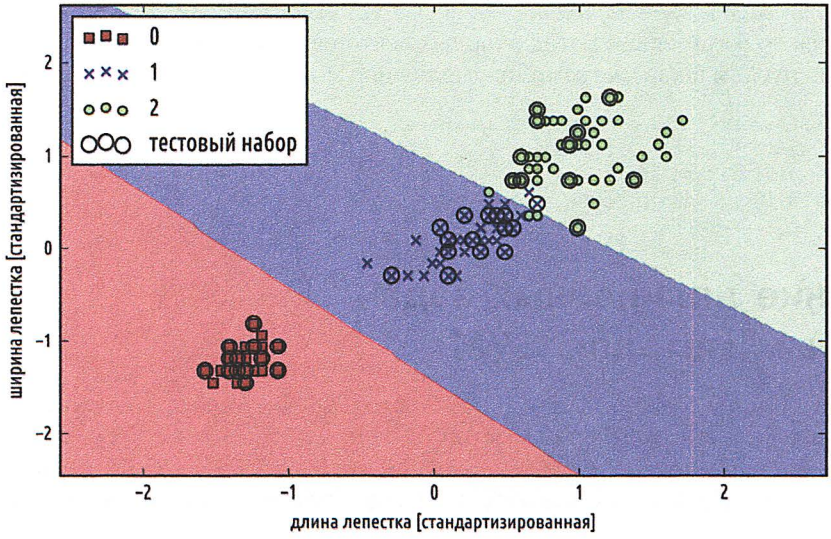


Это идея связана с регуляризацией, которую мы обсуждали ранее в контексте регуляризованной регрессии, где уменьшение величины параметра C увеличивает смещение модели и понижает ее дисперсию.

Изучив ключевые идеи, лежащие в основе линейного метода SVM, теперь натренируем модель SVM для классификации различных цветков в нашем наборе данных цветков ириса:

```
from sklearn.svm import SVC
svm = SVC(kernel='linear', C=1.0, random_state=0)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm,
                      test_idx=range(105,150))
plt.xlabel('длина лепестка [стандартизованая]')
plt.ylabel('ширина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
```

Визуальное представление областей решения SVM после выполнения приведенного выше примера показано на следующем ниже графике:



Логистическая регрессия в сопоставлении с методом опорных векторов
В практических задачах классификации линейная логистическая регрессия и линейные методы опорных векторов часто приводят к очень похожим результатам. Логистическая регрессия пытается максимизировать условные вероятности тренировочных данных, что делает ее более подверженной выбросам, чем методы опорных векторов (SVM). Методы SVM главным образом сосредоточены на точках, ближайших к границе решения (опорных векторах). С другой стороны, преимущество логистической регрессии состоит в том, что это более простая модель, которую проще реализовать. Кроме того, модели логистической регрессии можно легко обновлять, что выглядит привлекательным при работе с потоковой передачей данных.

Альтернативные реализации в scikit-learn

Классы библиотеки scikit-learn Perceptron и LogisticRegression, которые мы использовали в предыдущих разделах, используют высокооптимизированную динамическую библиотеку LIBLINEAR на C/C++, разработанную в Национальном Тайваньском университете (http://www.csie.ntu.edu.tw/~cjlin/liblinear/). Аналогичным образом примененный для тренировки модели SVM класс SVC использует эквивалентную динамическую бибилотеку LIBSVM на C/C++, специализированную под методы опорных векторов (http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

Преимущество использования динамических библиотек LIBLINEAR и LIBSVM над нативными реализациями на Python состоит в том, что они позволяют чрезвычайно быстро тренировать большое число линейных классификаторов. Однако иногда наши наборы данных слишком большие, чтобы уместиться в памяти компьютера. По этой причине библиотека scikit-learn средствами класса SGDClassifier также предлагает альтернативные реализации, причем этот класс, помимо всего прочего, поддерживает и динамическое (онлайновое) обучение, используя для этого метод partial_fit. В основе класса SGDClassifier лежит идея, которая похожа на алгоритм стохастического градиента, реализоованного нами в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* для ADALINE. Инициализировать персептрон, логистическую регрессию и метод опорных векторов в версии со стохастическим градиентным спуском с параметрами по умолчанию можно следующим образом:

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

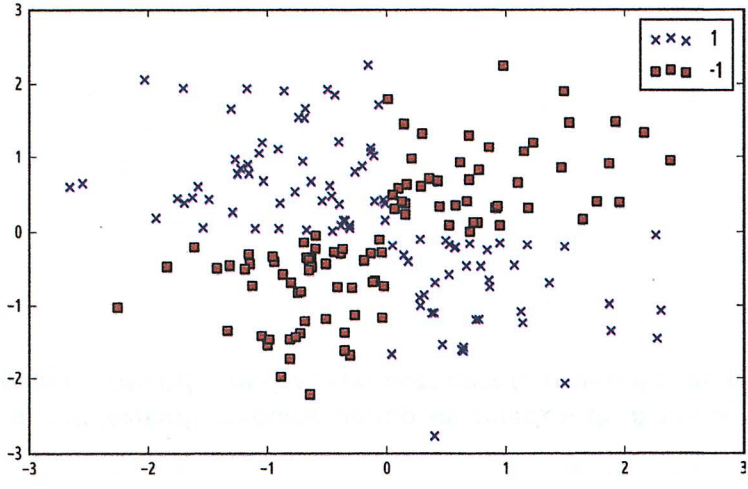
Решение нелинейных задач ядерным методом SVM

Еще одна причина, почему модели SVM обладают высокой популярностью среди практиков машинного обучения, состоит в том, что их можно легко *кернелизировать*, т. е. модифицировать с использованием *ядра*, для решения нелинейных задач классификации. Прежде чем мы обсудим идею, лежащую в основе **ядерного** метода SVM, сначала определим и создадим демонстрационный набор данных, чтобы увидеть, как может выглядеть такая нелинейная задача классификации.

Используя следующий ниже исходный код, мы создадим простой набор данных, который имеет вид логического элемента XOR, используя для этого функцию из библиотеки NumPy logical_xor, где первым 100 образцам назначается метка класса 1, и другим 100 образцам – метка класса –1:

```
np.random.seed(0)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0, X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, -1)

plt.scatter(X_xor[y_xor==1, 0], X_xor[y_xor==1, 1],
            c='b', marker='x', label='1')
plt.scatter(X_xor[y_xor==-1, 0], X_xor[y_xor==-1, 1],
            c='r', marker='s', label='-1')
plt.ylim(-3.0)
plt.legend()
plt.show()
```

После выполнения приведенного выше исходного кода у нас будет набор данных XOR со случайным шумом, как показано на нижеследующем рисунке:

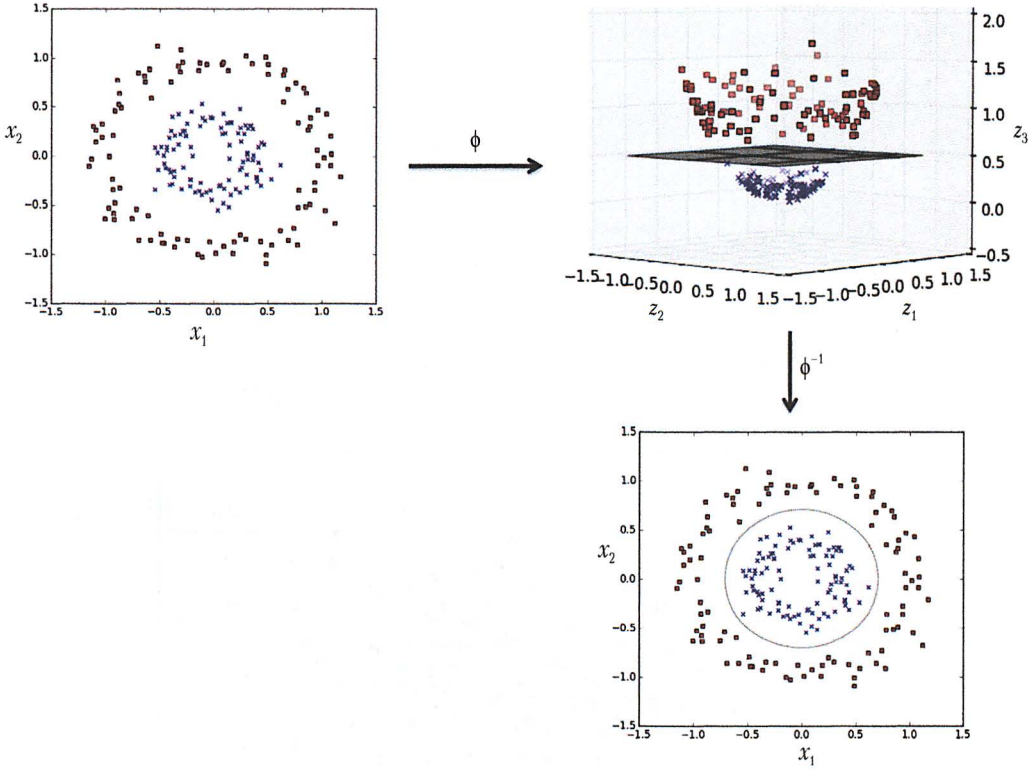


Совершенно очевидно, что нам не удастся очень хорошо разделить образцы из положительного и отрицательного классов, применив в качестве границы решения линейную гиперплоскость, пользуясь методом линейной логистической регрессии либо линейной модели SVM, которые мы обсуждали в более ранних разделах.

Ключевая идея в основе ядерных методов для решения задач с такими линейно неразделимыми данными состоит в том, чтобы создать нелинейные комбинации исходных признаков и функцией отображения $\phi(\cdot)$ спроецировать их на пространство более высокой размерности, где они становятся линейно разделимыми. Как показано на нижеследующем рисунке, мы можем трансформировать двумерный набор данных в новое трехмерное пространство признаков, где классы становятся разделимыми, благодаря следующей проекции:

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

Это дает нам возможность разделить эти два показанных на графике класса линейной гиперплоскостью, которая становится нелинейной границей решения, если ее спроецировать назад на исходное пространство признаков:



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

Чтобы решить нелинейную задачу при помощи SVM, мы переводим тренировочные данные в пространство признаков более высокой размерности, пользуясь функцией отображения $\phi(\cdot)$, и тренируем линейную модель SVM классифицировать данные в этом новом пространстве признаков. Затем ту же самую функцию отображения $\phi(\cdot)$ можно использовать для трансформации новых, ранее ненаблюдавшихся данных, чтобы их классифицировать при помощи линейной модели SVM.

Вместе с тем проблема с таким подходом на основе функции отображения состоит в том, что конструирование новых признаков в вычислительном плане очень затратно, в особенности если имеем дело с высокоразмерными данными. Именно здесь в игру вступает так называемый ядерный трюк (kernel trick), или подмена скалярного произведения функцией ядра. Несмотря на то что мы не вдавались в большое количество подробностей по поводу того, как решать задачу квадратичного программирования для тренировки модели SVM, на практике нам всего лишь нужно подставить $\phi(x^{(i)})^T\phi(x^{(j)})$ вместо скалярного произведения $x^{(i)T}x^j$. Для того чтобы сэкономить на затратном шаге прямого вычисления скалярного произве-

дения между двумя точками, мы определяем так называемую ядерную функцию: $k(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T\phi(x^{(j)})$.

Одна из наиболее широко используемых ядерных функций представлена **ядром из функции радиального базиса (ядром RBF)**, или гауссовым ядром:

$$k(x^{(i)}, x^{(j)}) = \exp\left(-\frac{x^{(i)} - x^{(j)2}}{2\sigma^2}\right).$$

Оно часто сводится к

$$k(x^{(i)}, x^{(j)}) = \exp(-\gamma\|x^{(i)} - x^{(j)}\|^2).$$

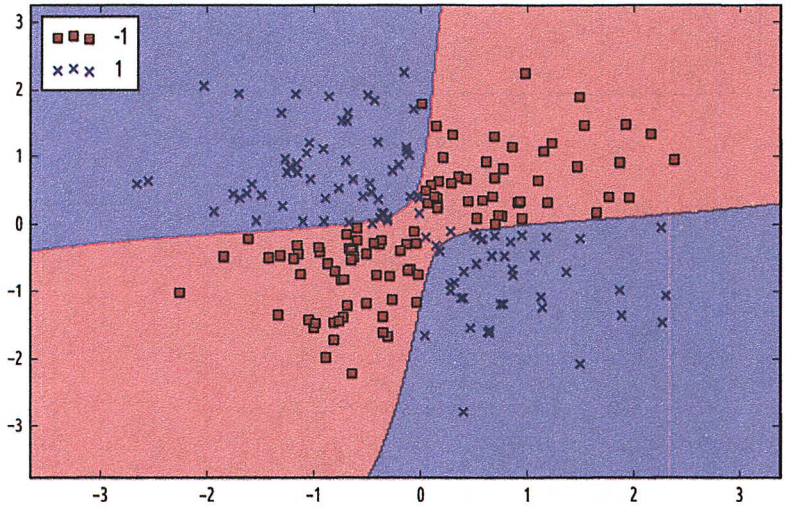
Здесь $\gamma = \frac{1}{2\sigma^2}$ – это свободный параметр, который нужно оптимизировать.

Грубо говоря, термин *ядро* можно интерпретировать как *функцию подобия* между парой образцов. Знак «минус» инвертирует меру расстояния в показатель подобия, и благодаря экспоненциальному члену результирующий показатель подобия попадет в диапазон между 1 (для строго подобных образцов) и 0 (для строго неподобных образцов).

Определив общую картину, лежащую в основе ядерного трюка, теперь посмотрим, сможем ли мы натренировать ядро SVM, чтобы оно могло провести нелинейную границу решения, которая бы хорошо разделила данные XOR. Здесь мы просто воспользуемся ранее импортированным из библиотеки scikit-learn классом SVC и заменим параметр kernel='linear' на kernel='rbf':

```
svm = SVC(kernel='rbf', random_state=0, gamma=0.10, C=10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor, classifier=svm)
plt.legend(loc='upper left')
plt.show()
```

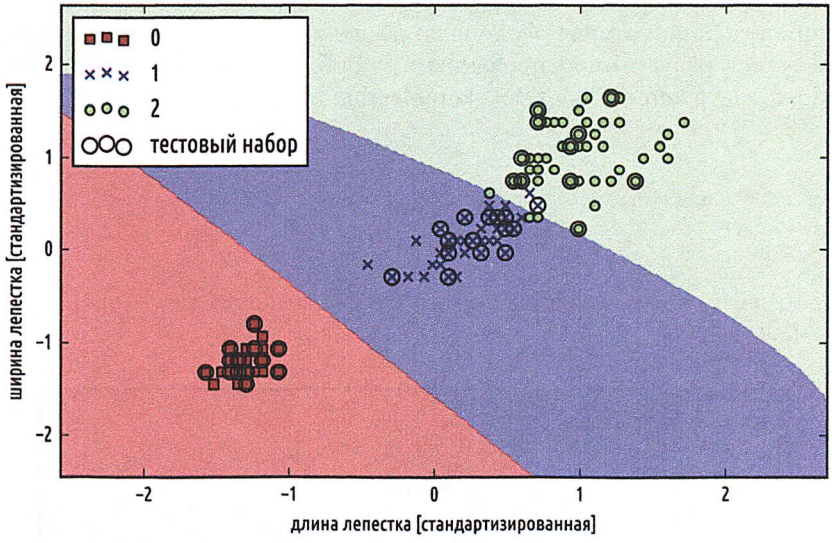
Как видно на итоговом графике, ядерная SVM относительно хорошо разделяет данные XOR:



Параметр γ, который мы устанавливаем в gamma = 0.1, может пониматься как параметр *отсечения* для гауссовой сферы. Если увеличить величину γ, то увеличивается влияние или охват тренировочных образцов, что ведет к более мягкой границе решения. Чтобы получить более полное интуитивное понимание параметра γ, применим SVM с ядром из функции радиального базиса (RBF) к нашему набору данных цветков ириса:

```
svm = SVC(kernel='rbf', random_state=0, gamma=0.2, C=1.0)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm,
                      test_idx=range(105,150))
plt.xlabel('длина лепестка [стандартизованная]')
plt.ylabel('ширина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
```

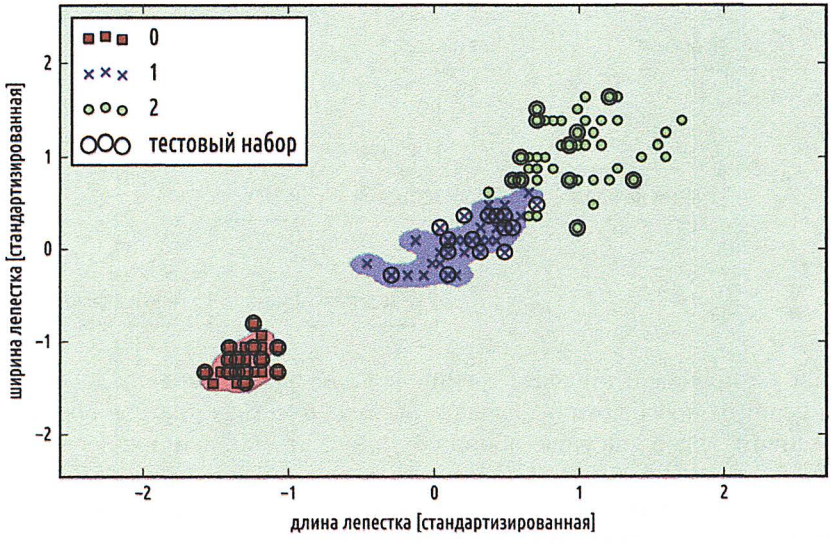
Учитывая, что мы выбрали относительно малую величину для γ, итоговая граница решения модели SVM с ядром из RBF будет относительно мягкой; это показано на нижеследующем рисунке:



Теперь увеличим параметр γ и проследим за влиянием на границу решения:

```
svm = SVC(kernel='rbf', random_state=0, gamma=100.0, C=1.0)
svm.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier=svm,
                      test_idx=range(105,150))
plt.xlabel('длина лепестка [стандартизованная]')
plt.ylabel('ширина лепестка [стандартизованная]')
plt.legend(loc='upper left')
plt.show()
```

Теперь на итоговом графике можно увидеть, что граница решения вокруг классов 0 и 1 намного компактнее относительно большой величины параметра γ:



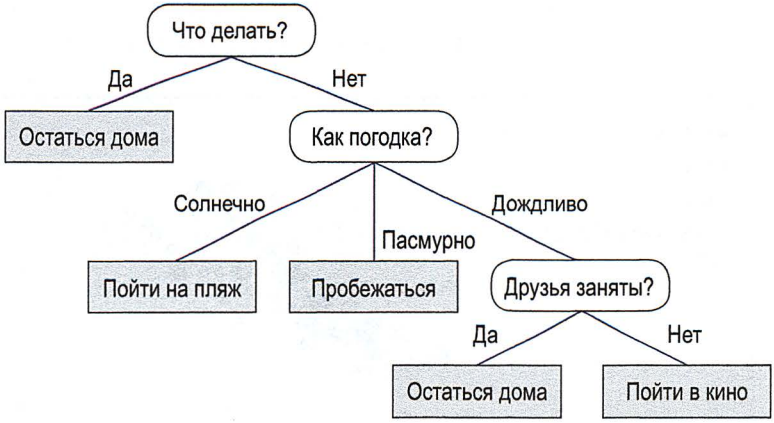
Хотя эта модель очень хорошо подогнана под тренировочный набор данных, т. е. хорошо им соответствует, такой классификатор, скорее всего, будет иметь высокую ошибку обобщения на ранее не наблюдавшихся данных, а это, в свою очередь, свидетельствует о том, что оптимизация параметра γ играет одинаково важную роль в управлении переобучения.

Обучение на основе деревьев решений

Классификаторы на основе **деревьев решений** (decision tree)[1], или решающих деревьев, являются привлекательными моделями, в случае если позаботиться об интерпретируемости. Как предполагает название термина «*дерево решений*», эту модель можно представить как разбиение данных на подмножества путем принятия решений, основываясь на постановке серии вопросов.

Рассмотрим следующий ниже пример, где мы используем дерево решений, чтобы определиться с видом деятельности в тот или иной конкретный день:

[1] В зависимости от поставленной задачи также может называться деревом классификации или регрессионным деревом. Далее будет рассматриваться именно дерево классификации, а в главе 10 – регрессионное дерево. – *Прим. перев.*



Опираясь на признаки в нашем тренировочном наборе, модель дерева решений обучается серии вопросов, чтобы сделать выводы о метках классов образцов. Хотя на приведенном выше рисунке проиллюстрирована концепция дерева решений с опорой на категориальные переменные, то же самое применимо, если наши признаки являются вещественными числами, как в наборе данных цветков ириса. Например, можно просто определить величину отсечения вдоль оси признака **ширина чашелистика** и задать бинарный вопрос «ширина чашелистика ≥ 2.8 см?».

Используя алгоритм выбора решения, мы начинаем в корне дерева и расщепляем данные по признаку, который ведет к самому большому **приросту информации** (information gain, IG); этот показатель получит более подробное объяснение в следующем разделе. Далее мы повторяем процедуру расщепления в итеративном режиме в каждом дочернем узле, пока не получим однородных листов. То есть все образцы в каждом узле принадлежат одному и тому же классу. На практике в результате такой операции может образоваться очень глубокое дерево со многими узлами, что легко может привести к переобучению. В силу этого дерево обычно *подрезается* путем установления предела для его максимальной глубины.

Максимизация прироста информации – получение наибольшей отдачи

Для того чтобы расщепить узлы в самых информативных признаках, нам нужно определить целевую функцию, которую мы хотим оптимизировать алгоритмом обучения на основе дерева. Здесь наша целевая функция состоит в максимизации прироста информации при каждом расщеплении, которую мы определяем следующим образом:

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

Здесь f – это признак, по которому выполняется расщепление, D_p и D_j – набор данных родительского и j-*го* дочернего узла, I – наша мера неоднородности, N_p – общее число образцов в родительском узле и N_j – число образцов в j-ом дочернем узле. Как можно убедиться, прирост информации – это просто разница между неоднородностью родительского узла и суммой неоднородностей дочерних узлов: чем ниже неоднородность дочерних узлов, тем больше прирост информации. Вместе с тем,

для простоты и чтобы уменьшить комбинаторное пространство поиска, в большинстве библиотек (включая scikit-learn) реализованы бинарные деревья решений. То есть каждый родительский узел расщепляется на два дочерних узла, $D_{\text{левый}}$ и $D_{\text{правый}}$:

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

В силу вышесказанного в бинарных деревьях решений обычно используются три меры неоднородности, или критерия расщепления, — **мера неоднородности Джини** (I_G)[1], **энтропия** (I_H) и **ошибка классификации** (I_E). Начнем с определения энтропии для всех **непустых** классов $p(i|t) \neq 0$[2]:

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

Здесь $p(i|t)$ – это доля образцов, которая принадлежит классу i для отдельно взятого узла t. Следовательно, энтропия равна 0, если все образцы в узле принадлежат одному и тому же классу, и энтропия максимальна, если у нас равномерное распределение классов. Например, в конфигурации с бинарными классами энтропия равна 0, если $p(i = 1|t) = 1$ или $p(i = 0|t) = 0$. Если классы распределены равномерно с $p(i = 1|t) = 0.5$ или $p(i = 0|t) = 0.5$, то энтропия равняется 1. Следовательно, можно сказать, что энтропийный критерий пытается максимизировать взаимную информацию в дереве.

Интуитивно мера неоднородности Джини может пониматься как критерий, который минимизирует вероятность ошибочной классификации:

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

Подобно энтропии, мера неоднородности Джини максимальна, в случае если классы полностью перемешаны, например в конфигурации с бинарными классами ($c = 2$):

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

Однако на практике мера неоднородности Джини и энтропия, как правило, дают очень похожие результаты, и не стоит тратить много времени на оценивание деревьев разными критериями неоднородности вместо того, чтобы тратить время на эксперименты с разными методами подрезания дерева путем отсечения ветвей.

---

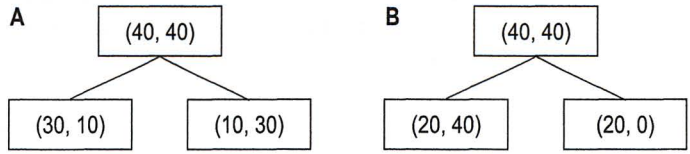
[1] Мера неоднородности, или мера Джини (Gini impurity), – это вероятность, что случайный образец будет классифицирован правильно, если произвольно выбрать метку согласно распределению в ветви дерева. Не следует путать с коэффициентом Джини в статистике, где он представляет собой показатель степени расслоения общества по отношению к какому-либо признаку. – *Прим. перев.*

[2] Речь идет о вероятностной мере неопределенности Шеннона (или информационной энтропии). Данная формула означает, что прирост информации равен утраченной неопределенности. – *Прим. перев.*

Еще одной мерой неоднородности является ошибка классификации:

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

Этот критерий подходит для подрезания дерева, но не рекомендуется для роста дерева, поскольку он менее чувствителен к изменениям в вероятностях классов в узлах. Это можно проиллюстрировать, обратившись к двум возможным сценариям расщепления, показанным на нижеследующем рисунке:



Мы начинаем с набора данных D_p в родительском узле D_p, который состоит из 40 образцов из класса 1 и 40 образцов из класса 2, который мы расщепляем на два набора данных, соответственно на $D_{\text{левый}}$ и $D_{\text{правый}}$. Прирост информации, используя ошибку классификации в качестве критерия расщепления, будет одинаковым ($IG_E = 0.25$) в обоих сценариях A и B:

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

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

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

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

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

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

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

Однако мера неоднородности Джини предпочтет расщепление в сценарии $B\left(IG_E = 0.\overline{16}\right)$ над сценарием $A(IG_E = 0.125)$, который действительно более *однороден*:

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

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

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

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

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

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

$$B: IG_G = 0.5 - \frac{6}{8}0.\overline{4} - 0 = 0.\overline{16}.$$

Аналогичным образом энтропийный критерий будет отдавать предпочтение сценарию $B(IG_H = 0.31)$ перед сценарием $A(IG_H = 0.19)$:

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

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

$$A: I_H(D_{\text{правый}}) = -\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right) = 0.92;$$

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

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

$$B: I_H(D_{\text{правый}}) = 0;$$

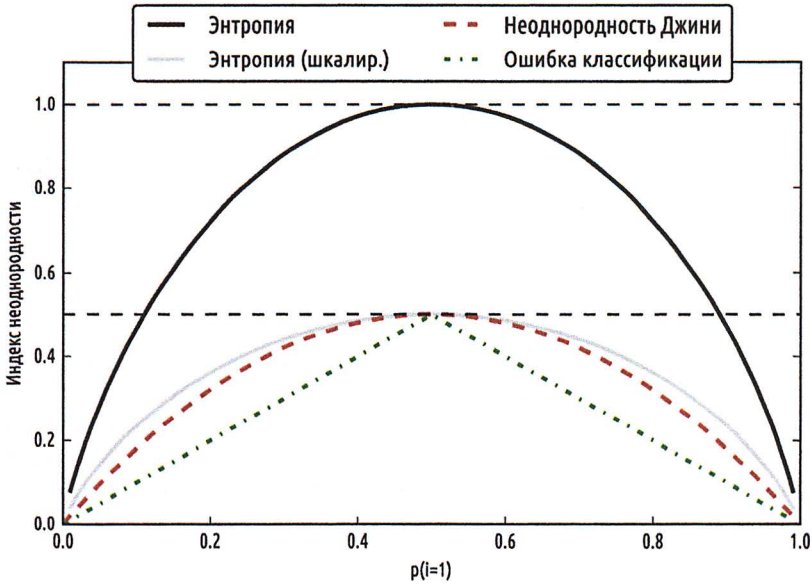
$$B: IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31.$$

Для более наглядного сравнения трех разных критериев неоднородности, которые мы обсудили выше, построим график индексов неоднородности для вероятностного диапазона [0, 1] для класса 1. Отметим, что мы также добавим масштабированную версию энтропии (*энтропия/2*), чтобы учесть, что мера неоднородности Джини является промежуточной между энтропией и ошибкой классификации. Соответствующий исходный код выглядит следующим образом:

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

def gini(p):
    return (p)*(1 - (p)) + (1 - p)*(1 - (1-p))

def entropy(p):
    return - p*np.log2(p) - (1 - p)*np.log2((1 - p))

def error(p):
    return 1 - np.max([p, 1 - p])

x = np.arange(0.0, 1.0, 0.01)
ent = [entropy(p) if p != 0 else None for p in x]
sc_ent = [e*0.5 if e else None for e in ent]
err = [error(i) for i in x]
fig = plt.figure()
ax = plt.subplot(111)
```

```
for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
                  ['Энтропия', 'Энтропия (шкалир.)',
                  ‹Неоднородность Джини›, ‹Ошибка классификации›],
                  ['-', '-', '--', '-.'],
                  ['black', 'lightgray', 'red', 'green', 'cyan']):
    line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
          ncol=3, fancybox=True, shadow=False)
ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
plt.ylim([0, 1.1])
plt.xlabel('p(i=1)')
plt.ylabel('Индекс неоднородности')
plt.show()
```

В результате выполнения приведенного выше примера будет сгенерирован следующий ниже график:

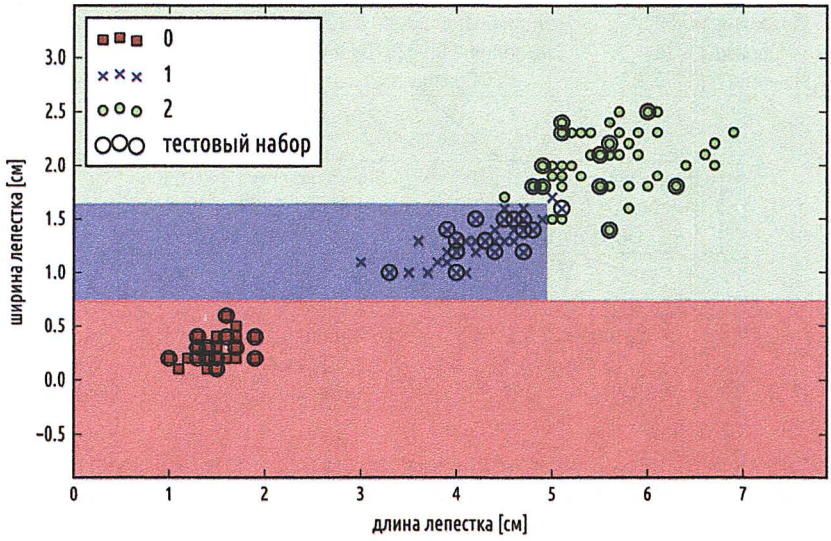


Построение дерева решений

Деревья решений могут создавать сложные границы решения путем деления пространства признаков на прямоугольники. Однако мы должны быть осторожными, поскольку чем глубже дерево решений, тем сложнее становится граница решения, что может легко закончиться переобучением. Теперь, используя scikit-learn, натренируем дерево решений с максимальной глубиной 3, применив в качестве критерия неоднородности энтропию. Хотя для целей визуализации может понадобиться шкалирование признаков, отметим, что масштабирование признаков не является необходимой составной частью алгоритмов деревьев решений. Соответствующий исходный код выглядит следующим образом:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
tree.fit(X_train, y_train)
X_combined = np.vstack((X_train, X_test))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X_combined, y_combined,
                      classifier=tree, test_idx=range(105,150))
plt.xlabel('длина лепестка [см]')
plt.ylabel('ширина лепестка [см]')
plt.legend(loc='upper left')
plt.show()
```

После выполнения приведенного выше примера получаем типичные ось-параллельные границы решения дерева решений.



В библиотеке scikit-learn имеется удобная возможность, позволяющая экспортировать дерево решений после тренировки в формате файла .dot, который затем можно визуализировать при помощи программы GraphViz. Эта программа имеется в свободном доступе на веб-сайте http://www.graphviz.org и поддерживается в Linux, Windows и Mac OS X.

Сначала мы создаем файл.dot, используя для этого функцию export_graphviz из подмодуля tree библиотеки scikit-learn, как показано ниже:

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file='tree.dot',
                feature_names=['petal length', 'petal width'])
```

После установки на компьютере программы GraphViz мы можем преобразовать файл tree.dot в PNG-файл, выполнив в командной строке в месте расположения сохраненного файла tree.dot следующую команду:

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

В Windows, чтобы кириллица в файле .dot отобразилась корректно, следует удостовериться, что файл сохранен в кодировке UTF8. Это можно сделать, например, в редакторе Notepad++ или любом другом, предназначенном для разработчиков. Кроме того, чтобы в Windows утилита запускалась из командной строки в любом расположении, следует в системной переменной **Path** прописать путь к программе (**Пуск** ⇒ **Система** ⇒ **Дополнительные параметры** ⇒ **Переменные среды** ⇒ **Системные переменные** и добавить в переменную **Path** путь к программе: C:\Program Files (x86)\Graphviz2.38\bin).

Анализируя дерево решений, которое мы создали при помощи программы GraphViz, теперь можно удобным образом проследить расщепления, которые дерево решений определило, исходя из нашего тренировочного набора данных. Начав со 105 образцов в корне, мы расщепили его на два дочерних узла с 34 и 71 образцами каждое, используя точку отсечения с **шириной лепестка** $\leq$ 0.75 см. После первого расщепления мы видим, что левый дочерний узел уже однороден и содержит только образцы из класса ириса щетинистого (энтропия = 0). Затем дальнейшие расщепления с правой стороны используются для разделения образцов из классов ириса разноцветного и ириса виргинского.

Объединение слабых учеников для создания сильного при помощи случайных лесов

В течение прошедшего десятилетия огромную популярность в приложениях машинного обучения получили **случайные леса** (random forests), и причина тому – их хорошая классификационная способность, масштабируемость и простота использования. Интуитивно случайный лес можно рассматривать как *ансамбль* деревьев решений. В основе ансамблевого обучения лежит идея объединения **слабых учеников** для создания более устойчивой модели, т. е. **сильного ученика**, с более хорошей ошибкой обобщения и меньшей восприимчивостью к переобучению. Алгоритм случайного леса можно резюмировать в четырех простых шагах:

1. Извлечь случайную **бутстрап-выборку** размера n (случайным образом отобрать из тренировочного набора данных n образцов с возвратом, то есть повторным способом)[1].
2. Вырастить дерево решений из бутстрап-выборки. В каждом узле:
   1) случайным образом отобрать d признаков без возврата, т. е. бесповторным способом;
   2) расщепить узел, используя признак, который обеспечивает наилучшее расщепление согласно целевой функции, например путем максимизации прироста информации;
   3) повторить шаги 1 и 2 k число раз;
   4) для назначения метки класса агрегировать прогноз из каждого дерева на основе **большинства голосов**. Мажоритарное голосование будет обсуждаться более подробно в *главе 7 «Комбинирование моделей для методов ансамблевого обучения»*.

На шаге 2 существует небольшая модификация алгоритма, когда мы тренируем отдельные деревья решений: вместо того чтобы для определения наилучшего расщепления в каждом узле оценивать все признаки, мы рассматриваем только случайное их подмножество.

Несмотря на то что случайные леса не предлагают интерпретируемости того же уровня, что и деревья решений, большое преимущество случайных лесов состоит в том, что не приходится слишком переживать о выборе подходящих значений гиперпараметров. Как правило, нам не нужно подрезать случайный лес, поскольку ансамблевая модель довольно устойчива к шуму из отдельных деревьев решений. Единственный параметр, о котором мы действительно должны позаботиться на практике, – это число деревьев k (шаг 3), который мы выбираем для случайного леса. Как правило, чем больше число деревьев, тем выше качество классификатора на основе случайного леса, достигаемая за счет повышения вычислительной емкости.

Другие, менее распространенные на практике гиперпараметры классификатора на основе случайного леса, которые можно оптимизировать (методы их использования будут обсуждаться в *главе 5 «Сжатие данных путем снижения размерности»*), представлены соответственно размером n бутстрап-выборки (шаг 1) и числом признаков d, которые случайным образом выбираются для каждого расщепления (шаг 2.1). Благодаря размеру n бутстрап-выборки мы управляем компромиссом между смещением и дисперсией случайного леса. Выбирая для n более крупное значение, мы уменьшаем случайность, и, следовательно, лес, скорее всего, будет переобучен. С другой стороны, можно уменьшить степень переобучения путем выбора меньших значений для n, делая это за счет качества модели. В большинстве реализаций, включая реализацию классификатора на основе случайного леса `RandomForestClas-`

---

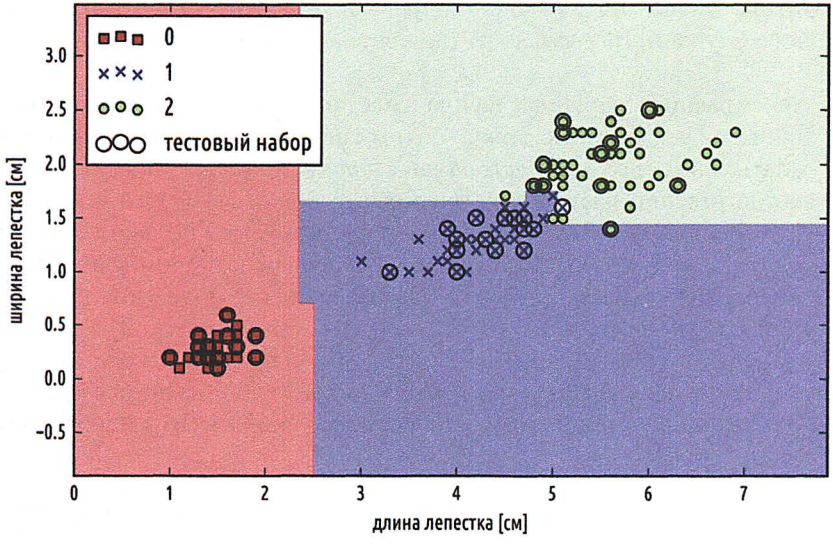
[1] Бустрап-выборка (bootstrap sample) – синтетический набор данных, полученный в результате генерирования повторных выборок (с возвратом) из исследуемой выборки, используемой в качестве «суррогатной генеральной совокупности», в целях аппроксимации выборочного распределения статистики (такой как среднее, медиана и др.). Выборка с возвратом (with replacement) (или отбор, допускающий повторы) аналогична неоднократному вытягиванию случайной карты из колоды игральных карт, когда после каждого вытягивания карта возвращается назад в колоду. В результате время от времени обязательно будет попадаться карта, которая уже выбиралась. – *Прим. перев.*

sifier в библиотеке scikit-learn, размер бутстрап-выборки выбирается так, чтобы он был эквивалентным числу образцов в исходном тренировочном наборе, что обычно обеспечивает хороший компромисс между смещением и дисперсией. Для числа признаков d в каждом расщеплении обычно выбирается значение, которое меньше общего числа признаков в тренировочном наборе. Разумное значение по умолчанию, которое используется в библиотеке scikit-learn и других реализациях, составляет $d = \sqrt{m}$, где m – это число признаков в тренировочном наборе.

Удобно то, что нам не нужно самостоятельно строить классификатор на основе случайного леса из отдельных деревьев решений; в библиотеке scikit-learn уже существует реализация, которую можно использовать:

```
from sklearn.ensemble import RandomForestClassifier
forest = RandomForestClassifier(criterion='entropy',
                                n_estimators=10,
                                random_state=1,
                                n_jobs=2)
forest.fit(X_train, y_train)
plot_decision_regions(X_combined, y_combined,
                      classifier=forest, test_idx=range(105,150))
plt.xlabel('длина лепестка')
plt.ylabel('ширина лепестка')
plt.legend(loc='upper left')
plt.show()
```

После выполнения приведенного выше примера мы должны увидеть области решений, сформированные ансамблем деревьев в случайном лесе, как показано на нижеследующем рисунке:



Используя приведенный выше пример, мы натренировали случайный лес из 10 деревьев решений, исходя из параметра количества оценщиков n_estimators, и использовали в качестве меры неоднородности для расщепления узлов энтропийный критерий. Несмотря на то что мы выращиваем очень небольшой случайный лес из очень небольшого тренировочного набора данных, в демонстрационных целях мы применили параметр n_jobs, который позволяет распараллелить обучение модели с использованием двух или более ядер нашего компьютера (в данном случае двух).

k ближайших соседей – алгоритм ленивого обучения

Последний алгоритм обучения с учителем, который мы хотим обсудить в этой главе, – это **классификатор на основе *k* ближайших соседей** (k-nearest neighbor classifier, KNN), который особенно интересен тем, что он существенно отличается от алгоритмов обучения, которые мы обсуждали до сих пор.

KNN является типичным примером **ленивого ученика**. Его называют *ленивым* не из-за его очевидной простоты, а потому, что он не извлекает различающую (дискриминантную) функцию из тренировочных данных, а вместо этого запоминает тренировочный набор данных.

Параметрические модели в сопоставлении с непараметрическими

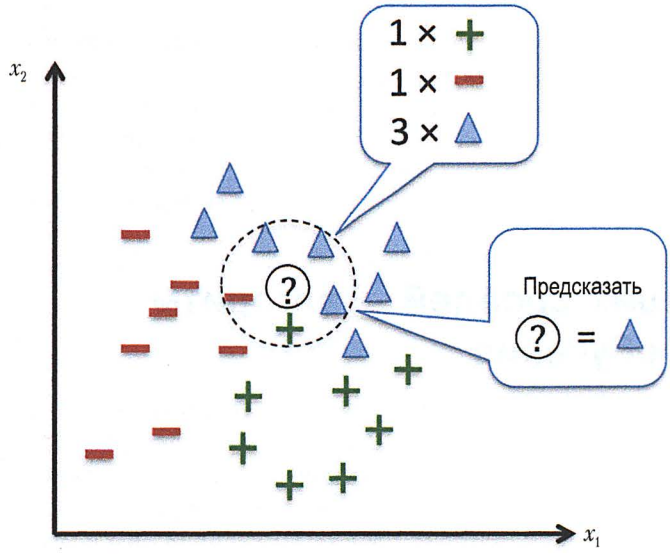
Алгоритмы машинного обучения можно сгруппировать в **параметрические** и **непараметрические** модели. Используя параметрические модели, мы выполняем оценивание параметров из тренировочного набора данных, чтобы извлечь функцию, которая сможет классифицировать новые точки данных, больше не требуя исходного тренировочного набора данных. Типичными примерами параметрических моделей являются персептрон, логистическая регрессия и линейный метод опорных векторов (SVM). Напротив, непараметрические модели не могут быть охарактеризованы фиксированным набором параметров, и число параметров в них растет вместе с тренировочными данными. Двумя примерами непараметрических моделей, которые мы видели до сих пор, являются классификатор на основе дерева решений/случайного леса и ядерный метод SVM.

Модель *k* ближайших соседей (KNN) принадлежит подкатегории непараметрических моделей, которая описывается как **обучение на примерах** (или по прецедентам, instance-based). Модели, в основе которых лежит обучение на примерах, характеризуются запоминанием тренировочного набора данных, и ленивое обучение является особым случаем обучения на примерах, которое связано с отсутствующей (нулевой) стоимостью во время процесса обучения.

Непосредственно сам алгоритм *k* ближайших соседей (KNN) является довольно прямолинейным и может быть резюмирован следующими шагами:

1. Выбрать число *k* и метрику расстояния.
2. Найти *k* ближайших соседей образца, который мы хотим классифицировать.
3. Присвоить метку класса мажоритарным голосованием.

Следующий ниже рисунок иллюстрирует, как новой точке данных (**?**) присваивается треугольная метка класса, основываясь на мажоритарном голосовании среди ее пяти ближайших соседей.



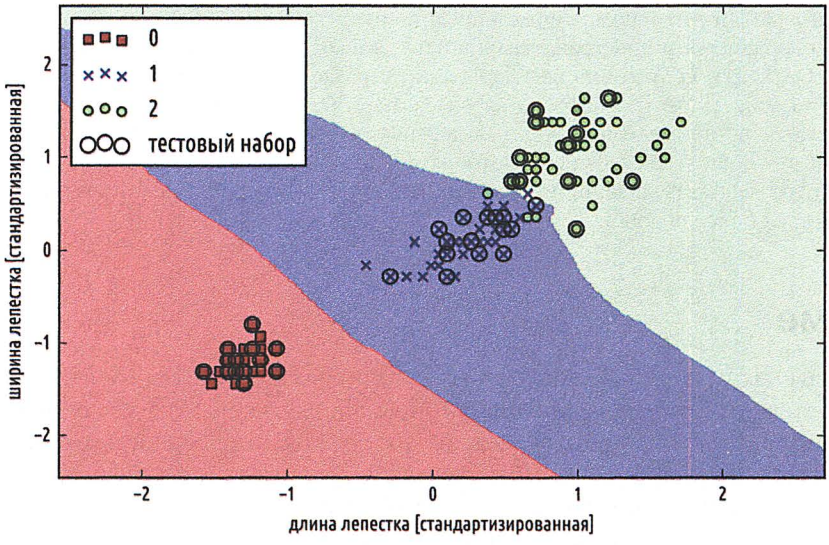
Основываясь на выбранной метрике расстояния, алгоритм KNN находит в тренировочном наборе данных k образцов, которые являются самыми близкими к классифицируемой точке (самыми похожими на нее). Метка класса новой точки данных затем определяется мажоритарным голосованием среди ее k ближайших соседей.

Основное преимущество такого подхода с запоминанием состоит в том, что классификатор немедленно адаптируется по мере сбора новых тренировочных данных. Однако его оборотная сторона – вычислительная сложность классифицирования новых образцов – растет линейно вместе с числом образцов в тренировочном наборе данных в наихудшем случае, если только в наборе данных не очень много размерностей (признаков) и алгоритм не был реализован с использованием эффективных структур данных, таких как KD-деревья (J. H. Friedman, J. L. Bentley and R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, ACM Transactions on Mathematical Software (TOMS) ACM, 3 (3):209-226, 1977 (*«Алгоритм для нахождения наилучших соответствий в логарифмически ожидаемое время»*)). Кроме того, мы не можем отбросить тренировочные образцы, поскольку никакого *тренирующего* шага нет. Вследствие этого, если мы работаем с большими наборами данных, пространство памяти может представлять серьезную проблему.

Выполнив приведенный ниже пример, мы теперь выполним реализацию модели KNN в библиотеке scikit-learn, используя для нее метрику евклидового расстояния:

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

Задав пять соседей в модели KNN для этого набора данных, мы получаем относительно гладкую границу решения, как показано на нижеследующем рисунке:



В случае равного числа голосов реализация алгоритма KNN в библиотеке scikit-learn предпочитает соседей с наименьшим расстоянием до образца. Если у соседей одинаковое расстояние, то алгоритм выбирает метку класса, которая стоит в тренировочном наборе данных на первом месте.

Правильный выбор числа k крайне важен для нахождения хорошего равновесия между переобучением и недообучением. Мы также должны убедиться, что выбираем метрику расстояния, подходящую для признаков в наборе данных. Для образцов с вещественными значениями, как, например, в случае с цветками в наборе данных цветков ириса, чьи признаки измеряются в сантиметрах, нередко используется простая евклидова мера расстояния. При этом если мы используем евклидову меру расстояния, также важно данные стандартизировать, благодаря чему каждый признак вносит в расстояние одинаковый вклад. Расстояние Минковского (`'minkowski'`), которое мы использовали в приведенном выше примере, является простым обобщением евклидова расстояния и расстояния городских кварталов (манхэттенского), которое можно записать следующим образом:

$$d(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \sqrt[p]{\sum_k \left| x_k^{(i)} - x_k^{(j)} \right|^p}.$$

Оно становится евклидовым расстоянием в случае, если установить параметр в `p=2`, или, соответственно, манхэттенским расстоянием при `p=1`. В библиотеке scikit-learn имеется много других метрик расстояния, которые можно передать в параметр `metric`. Они перечислены на веб-странице http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html.

Проклятие размерности

Важно упомянуть, что модель KNN очень восприимчива к переобучению из-за явления **проклятия размерности**, когда пространство признаков становится все более разреженным для растущего числа размерностей тренировочного набора данных фиксированно-

го размера. Интуитивно мы можем представить, что даже ближайшие соседи находятся в высокоразмерном пространстве слишком далеко, чтобы дать хорошую оценку.

Мы обсуждали концепцию регуляризации в разделе, посвященном логистической регрессии, как одном из способов избежать переобучения. Однако в моделях, где регуляризация не применима, таких как деревья решений и k ближайших соседей (KNN), мы можем использовать методы отбора признаков и снижения размерности, которые помогают избежать явления проклятия размерности. В следующей главе они будут обсуждаться более подробно.

Резюме

В этой главе вы узнали о разнообразных алгоритмах машинного обучения, которые используются для решения линейных и нелинейных задач. Мы убедились, что деревья решений особенно привлекательны, в случае если интерпретируемости модели уделено должное внимание. Логистическая регрессия – это не только полезная модель для динамического (онлайнового) обучения методом стохастического градиентного спуска, она позволяет также прогнозировать вероятность отдельно взятого события. Несмотря на то что методы опорных векторов являются мощными линейными моделями, которые могут быть распространены на нелинейные задачи методом ядерного трюка, для создания хороших прогнозов они требуют настройки большого числа параметров. Напротив, ансамблевые методы, такие как случайные леса, не требуют такой настройки и не подвержены переобучению с такой же легкостью, что и деревья решений, что делает эту модель привлекательной для многих областей практического применения. Классификатор на основе k ближайших соседей предлагает альтернативный подход к классификации – использовать ленивое обучение, которое позволяет делать прогнозы без какой-либо тренировки модели, но с более вычислительно затратным шагом прогнозирования.

Однако еще более важное значение, чем выбор надлежащего алгоритма обучения, имеют данные в нашем тренировочном наборе. Никакой алгоритм не сможет делать хороших прогнозов без информативных и отличительных признаков.

В следующей главе мы обсудим важные темы относительно предобработки данных, отбора признаков и снижения размерности, которые нам потребуются для создания мощных машиннообучаемых моделей. Позже, в *главе 6 «Изучение наиболее успешных методов оценки моделей и тонкой настройки гиперпараметров»*, мы увидим, каким образом можно оценивать и сравнивать качество наших моделей, и изучим полезные приемы тонкой настройки различных алгоритмов.

Создание хороших тренировочных наборов – предобработка данных

Качество данных и объем полезной информации, которую они содержат, являются ключевыми факторами, которые определяют, как хорошо алгоритм машинного обучения сможет обучиться. Следовательно, крайне важно сначала набор данных обязательно проверить и подвергнуть предварительной обработке и только потом подавать его на вход обучаемого алгоритма. В этой главе мы обсудим ключевые методы предобработки данных, которые помогут создавать хорошие машиннообучаемые модели.

В этой главе мы охватим следующие темы:

- удаление и импутация[1] пропущенных значений из набора данных;
- приведение категориальных данных в приемлемый для алгоритмов машинного обучения вид;
- отбор соответствующих признаков для конструирования модели.

Решение проблемы пропущенных данных

В реальных приложениях нередки случаи, когда в образцах по разным причинам пропущено одно или несколько значений. Возможно, в процесс сбора данных вкралась ошибка, некоторые данные измерений оказались неприемлемыми, отдельно взятые поля могли просто остаться незаполненными, например при опросе. Мы обычно рассматриваем *пропущенные значения* как пробелы в таблице данных либо как строковые заполнители, такие как `NaN` (т. е. not a number, не число).

К сожалению, большинство вычислительных инструментов неспособно обрабатывать такие пропущенные значения либо генерирует непредсказуемые результаты, если их просто проигнорировать. Поэтому, прежде чем продолжить дальнейший анализ, крайне важно разобраться в этих пропущенных значениях. Но прежде чем мы обсудим несколько методов решения проблемы пропущенных значений, сначала в качестве простого примера создадим простую таблицу, или фрейм, данных из **CSV-файла** (файл значений с разделением полей запятыми), для того чтобы лучше уловить суть проблемы:

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

[1] Импутация (imputation) – процесс замещения пропущенных, некорректных или несостоятельных значений другими значениями. – *Прим. перев.*

```
csv_data = '''A,B,C,D
              1.0,2.0,3.0,4.0
              5.0,6.0,,8.0
              10.0,11.0,12.0,'''
# Если вы используете Python 2.7, то вам нужно
# конвертировать строковое значение в Юникод:
# csv_data = unicode(csv_data)
df = pd.read_csv(StringIO(csv_data))
df
```

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

При помощи приведенного выше исходного кода мы считали данные в формате CSV в таблицу данных DataFrame библиотеки pandas, используя для этого функцию read_csv, при этом отметим, что две пустые ячейки были заменены на NaN. Функция StringIO в приведенном выше примере использовалась просто в целях иллюстрации. Она позволяет считывать строковое значение, присвоенное переменной csv_data, в объект DataFrame библиотеки pandas так, как будто это обычный CSV-файл на нашем жестком диске.

В случае более крупной таблицы данных DataFrame поиск пропущенных значений в ручном режиме может оказаться утомительным; тогдаа можно воспользоваться методом isnull для возврата таблицы данных DataFrame с булевыми значениями, которые покажут, содержит ли ячейка численное значение (False) либо в ней данные пропущены (True). Затем можно вернуть число пропущенных значений по каждому столбцу, воспользовавшись для этого методом sum, как показано ниже:

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

Таким способом мы можем подсчитать число пропущенных значений из расчета на каждый столбец; в следующих подразделах мы рассмотрим различные стратегии в отношении того, как работать с этими пропущенными данными.

Несмотря на то что библиотека scikit-learn предназначена для работы с массивами NumPy, иногда может оказаться более целесообразным предварительно обработать данные при помощи таблицы данных DataFrame библиотеки pandas. Мы всегда можем получить доступ к базовому массиву NumPy таблицы данных DataFrame через атрибут перед тем, как подавать его на вход оценщика библиотеки scikit-learn:

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

Устранение образцов либо признаков с пропущенными значениями

Один из самых простых способов решения проблемы пропущенных данных состоит в том, чтобы просто удалить соответствующие признаки (столбцы) или образцы (строки) из набора данных полностью; строки с пропущенными значениями можно легко отбросить при помощи метода dropna:

```
>>>
df.dropna()
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

Точно так же можно отбросить столбцы, которые в любой строке имеют, по крайней мере, одно значение NaN; это делается путем установки параметра оси axis в 1:

```
>>>
df.dropna (axis=1)
```

	A	B
0	1.0	2.0
1	5.0	6.0
2	10.0	11.0

Метод dropna поддерживается нескольких дополнительных параметров, которые могут пригодиться:

```
# отбросить строки, только если все столбцы содержат NaN
df.dropna(how ='all')
```

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

```
# отбросить строки, если в них менее 4 значений не-NaN
df.dropna(thresh=4)
```

	A	B	C	D
0	1.0	2.0	3.0	4.0

```
# отбросить строки, если в определенных столбцах (здесь 'C') имеется NaN
df.dropna(subset=['C'])
```

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

Несмотря на то что подход на основе удаления пропущенных данных выглядит удобным, он также несет в себе определенные недостатки; например, можно в конечном счете удалить слишком много образцов, которые сделают надежный анализ невозможным. Либо если удалить слишком много признаковых столбцов, то мы

рискуем потерять ценную информацию, в которой наш классификатор нуждается, для того чтобы провести разграничение между классами. Поэтому в следующем разделе мы обратимся к одной из часто используемых альтернатив решения проблемы пропущенных значений – методам интерполяции[1].

Импутация пропущенных значений

Нередко удаление образцов или отбрасывание всех признаковых столбцов просто недопустимо, потому что можно потерять слишком много ценных данных. В этом случае можно воспользоваться различными методами интерполяции для оценки пропущенных значений из других тренировочных образцов в наборе данных. Один из наиболее распространенных методов интерполяции представлен **импутацией простым средним**, где пропущенное значение замещается среднеарифметическим значением всего признакового столбца. Класс Imputer библиотеки scikit-learn позволяет легко это сделать, как показано в следующем ниже примере:

```
>>>
from sklearn.preprocessing import Imputer
imr = Imputer(missing_values='NaN', strategy='mean', axis=0)
imr = imr.fit(df)
imputed_data = imr.transform(df.values)
imputed_data
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  7.5,   8.],
       [ 10.,  11.,  12.,   6.]])
```

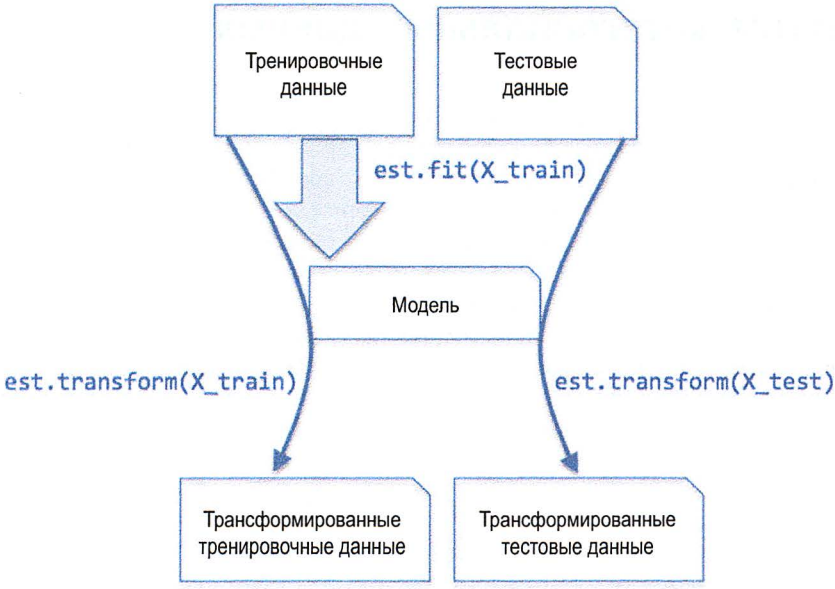
Здесь каждое значение NaN заменено на соответствующее среднее значение, которое вычисляется отдельно для каждого признакового столбца. Если поменять значение оси axis=0 на axis=1, то будет вычислено среднее строки. Другими значениями параметра strategy может быть median либо most_frequent, где в последнем случае пропущенные значения меняются на самые частотные значения. Это удобно при импутации значений категориальных признаков.

Концепция взаимодействия с оценщиками в библиотеке scikit-learn

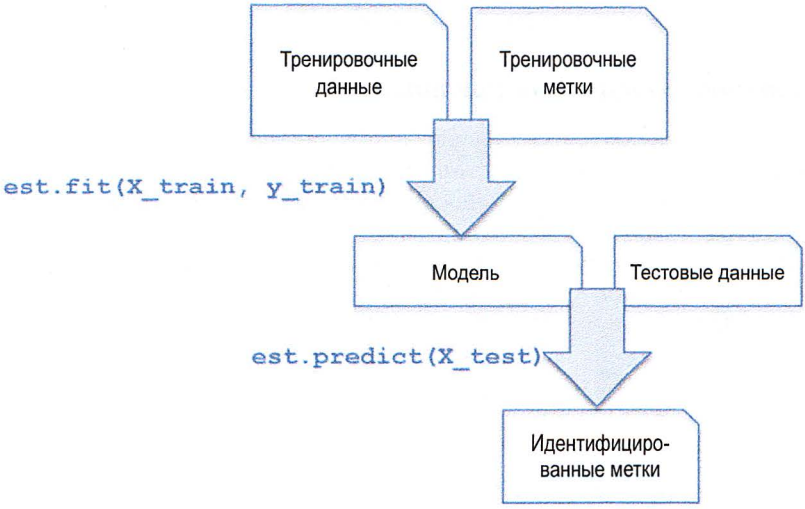
В предыдущем разделе мы использовали класс Imputer из библиотеки scikit-learn, чтобы вставить в набор данных пропущенные значения. В библиотеке scikit-learn класс Imputer принадлежит к так называемым **классам-преобразователям**, которые используются для преобразования данных. Двумя ключевыми методами в таких оценщиках являются метод fit и метод transform. Метод fit используется для извлечения параметров из тренировочных данных, а метод transform применяет эти параметры для преобразования данных. Любой преобразуемый массив данных должен иметь одинаковое число признаков, как и массив данных, который использовался для подгонки модели. На нижеследующем рисунке показано, как преобразователь, подогнанный на тренировочных данных, используется для преобразования тренировочного и нового тестового наборов данных:

---

[1] На веб-странице Криса Элбона (http://chrisalbon.com/python/pandas_missing_data.html) представлен хороший обзор методов работы с пропущенными данными при помощи библиотеки pandas. – *Прим. перев.*



Использованные нами в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* классификаторы библиотеки scikit-learn принадлежат к так называемым оценщикам с интерфейсом, который концептуально очень похож на класс-преобразователь. Оценщики имеют метод predict, но могут также иметь метод transform, как мы позже увидим. Как вы, возможно, помните, мы тоже использовали метод fit для извлечения параметров модели, когда тренировали эти оценщики для задачи классификации. Однако в задачах обучения с учителем мы дополнительно предоставляем метки классов для подгонки модели, которую затем при помощи метода predict можно использовать для выполнения прогнозов о новых образцах данных, как проиллюстрировано на нижеследующем рисунке:



Обработка категориальных данных

До сих пор мы работали только с численными значениями. Однако нередки случаи, когда реальные наборы данных содержат один или несколько столбцов категориальных признаков. Когда мы говорим о категориальных данных, мы должны далее различать **номинальные** и **порядковые** признаки. Порядковые признаки могут пониматься как категориальные значения, которые могут быть отсортированы или упорядочены. Например, *размер футболки* будет порядковым признаком, потому что можно определить их порядок $XL > L > M$. Напротив, номинальные признаки не подразумевают порядка, и, продолжая с предыдущим примером, *цвет футболки* можно представить как номинальный признак, поскольку, как правило, нецелесообразно говорить, что, например, *красный* больше *синего*.

Прежде чем разобрать различные методы обработки таких категориальных данных, создадим новую таблицу данных, чтобы проиллюстрировать задачу:

```
>>>
import pandas as pd
df = pd.DataFrame([
          ['зеленый', 'M', 10.1, 'класс1'],
          [‹красный›, ‹L', 13.5, 'класс2'],
          [‹синий›,  ‹XL', 15.3, 'класс1']])
df.columns = [‹цвет›, ‹размер›, ‹цена›, ‹метка›]  # метка класса
df
```

	Цвет	Размер	Цена	Метка
0	зеленый	M	10.1	класс1
1	красный	L	13.5	класс2
2	синий	XL	15.3	класс1

Как видно по предыдущему результату, вновь созданная таблица данных DataFrame содержит столбец с номинальным признаком (цвет), порядковым признаком (размер) и численным признаком (цена). Метки классов (принимая, что мы создали набор данных для задачи обучения с учителем) хранятся в последнем столбце. В обсуждаемых в этой книге алгоритмах классификации данных порядковая информация в метках классов не используется.

Преобразование порядковых признаков

Чтобы убедиться, что алгоритм обучения интерпретирует порядковые признаки правильно, мы должны перевести категориальные строковые значения в целочисленные. К сожалению, вспомогательная функция, которая может автоматически получать правильный порядок следования меток для нашего признака «размер», отсутствует. Поэтому нам нужно задать соответствия вручную. В следующем простом примере показано решение для случая, когда мы знаем разницу между признаками, например $XL = L + 1 = M + 2$.

```
>>>
size_mapping = {   # словарь соответствий
                 'XL': 3,
                 'L' : 2,
                 'M' : 1}
df['размер'] = df['размер'].map(size_mapping)
df
```

	Цвет	Размер	Цена	Метка
0	зеленый	1	10.1	класс1
1	красный	2	13.5	класс2
2	синий	3	15.3	класс1

Если на более позднем шаге нужно перевести целочисленные значения назад в первоначальное строковое представление, можно просто задать словарь соответствий с обратным отображением `inv_size_mapping = {v: k for k, v in size_mapping.items()}`, который затем при помощи метода `map` библиотеки pandas можно применить к преобразованному признаковому столбцу, аналогично словарю соответствий `size_mapping`, который мы использовали ранее.

Кодирование меток классов

Во многих библиотеках машинного обучения требуется, чтобы метки классов кодировались как целочисленные значения. Несмотря на то что большинство эстиматоров для классификации данных в scikit-learn внутри переводит метки классов в целые числа, считается хорошей практикой в целях предотвращения технических сбоев предоставлять метки классов в виде целочисленных массивов. Для кодирования меток классов можно задействовать подход, аналогичный обсуждавшемуся выше преобразованию порядковых признаков. Нужно помнить, что метки классов *не* являются порядковыми, и поэтому не имеет значения, какое целое число мы присваиваем отдельно взятой строковой метке. Поэтому можно просто нумеровать метки классов, начиная с 0:

```
>>>
import numpy as np
class_mapping = {label:idx for idx,label in
                 enumerate(np.unique(df['метка']))}
class_mapping
{'класс1': 0, 'класс2': 1}
```

Затем можно применить словарь соответствий для преобразования меток классов в целые числа:

```
>>>
df['метка'] = df['метка'].map(class_mapping)
df
```

	Цвет	Размер	Цена	Метка
0	зеленый	1	10.1	0
1	красный	2	13.5	1
2	синий	3	15.3	0

Для того чтобы вернуть преобразованные метки классов назад в первоначальное строковое представление, пары ключ-значение в словаре соответствий можно инвертировать следующим образом:

```
>>>
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['метка'] = df['метка'].map(inv_class_mapping)
df
```

	Цвет	Размер	Цена	Метка
0	зеленый	1	10.1	класс1
1	красный	2	13.5	класс2
2	синий	3	15.3	класс1

Как вариант непосредственно в самой библиотеке scikit-learn реализован вспомогательный класс LabelEncoder, который выполняет то же самое:

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

Отметим, что метод fit_transform – это просто укороченная альтернатива раздельному вызову метода fit и метода transform, причем этот метод можно использовать для преобразования целочисленных меток классов назад в их первоначальное строковое представление:

```
>>>
class_le.inverse_transform(y)
array(['класс1', 'класс2', 'класс1'], dtype=object)
```

Прямое кодирование на номинальных признаках

В предыдущем разделе для перевода значений порядкового признака «размер» в целочисленные мы использовали простой подход на основе словаря соответствий. Поскольку оценщики библиотеки scikit-learn рассматривают метки классов без какого-либо порядка, мы использовали вспомогательный класс LabelEncoder для кодирования строковых меток в целые числа. Может показаться, что аналогичный подход можно использовать и для преобразования столбца «цвет» с номинальными значениями, как показано ниже:

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

После выполнения приведенного выше фрагмента первый столбец массива X библиотеки NumPy теперь содержит новые значения цвета, которые закодированы следующим образом:

- синий → 0;
- зеленый → 1;
- красный → 2.

Если остановиться в этой точке и подать массив на вход классификатора, то мы сделаем одну из наиболее распространенных ошибок, которая встречается во время работы с категориальными данными. Можете определить, в чем проблема? Несмотря на то что значения цвета не идут в том или ином конкретном порядке, алгоритм

обучения теперь предположит, что *зеленый* больше *синего*, а *красный* больше *зеленого*. Хотя это допущение является неправильным, алгоритм все равно способен произвести полезные результаты. Однако эти результаты не будут оптимальными.

Общее обходное решение этой проблемы состоит в использовании метода, который называется **прямым кодированием**[1]. В основе этого подхода лежит идея, которая состоит в том, чтобы в столбце номинального признака создавать новый **фиктивный признак** для каждого уникального значения. В данном случае признак «цвет» можно преобразовать в три новых признака: синий, зеленый и красный. Затем можно использовать двоичные значения для указания отдельно взятого цвета образца; например, синий образец может быть закодирована как синий=1, зеленый=0, красный=0. Для выполнения этого преобразования можно воспользоваться классом OneHotEncoder, который реализован в модуле scikit-learn.preprocessing:

```
>>>
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(categorical_features=[0])
ohe.fit_transform(X).toarray()
array([[ 0. , 1. , 0. , 1. , 10.1],
       [ 0. , 0. , 1. , 2. , 13.5],
       [ 1. , 0. , 0. , 3. , 15.3]])
```

При инициализации объекта OneHotEncoder мы при помощи параметра categorical_features задали номер столбца переменной, которую мы хотим преобразовать (отметим, что признак цвет является первым столбцом в матрице признаков X). Когда мы используем метод transform, кодировщик OneHotEncoder по умолчанию возвращает разреженную матрицу, и в целях визуализации мы преобразовали представление в виде разреженной матрицы в регулярный (плотный) массив NumPy, используя для этого метод toarray. Дело в том, что представление в виде разреженных матриц является более эффективным способом хранения больших наборов данных, и он поддерживается многими функциями библиотеки scikit-learn, что в особенности полезно, если в ней содержится много нулей. Чтобы исключить этап с функцией toarray, можно инициализировать генератор прямого кода, указав параметр sparse, как, например, OneHotEncoder(..., sparse=False), в результате возвратив регулярный массив библиотеки NumPy.

Еще более удобный способ создания фиктивных признаков методом прямого кодирования состоит в том, чтобы использовать метод get_dummies, реализованный в библиотеке pandas. Если применить метод get_dummies к объекту DataFrame, то он выполнит преобразование только строковых столбцов и оставит все остальные столбцы без изменений:

```
>>>
pd.get_dummies(df[['цена', 'цвет', 'размер']])
```

	Цена	Размер	Цвет_зеленый	Цвет_красный	Цвет_синий
0	10.1	1	1	0	0
1	13.5	2	0	1	0
2	15.3	3	0	0	1

[1] Прямое кодирование (one-hot encoding) – двоичный код фиксированной длины, содержащий только одну 1. При обратном кодировании имеется только один 0. Длина кода определяется количеством кодируемых признаков, или объектов. – *Прим. перев.*

Разбивка набора данных на тренировочное и тестовое подмножества

В *главе 1 «Наделение компьютеров способностью обучаться на данных»* и в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* мы кратко представили принцип разбивки данных на раздельные наборы данных для тренировки и тестирования. Напомним, что тестовый набор может пониматься как *окончательная проверка* нашей модели, после которой мы выпускаем ее в реальный мир. В этом разделе мы подготовим новый набор данных **Wine**. После того как мы предварительно обработаем этот набор данных, мы разберем различные методы отбора признаков с целью снижения его размерности.

Набор данных сортов вин Wine – это еще один общедоступный набор данных, предлагаемый репозиторием машинного обучения UCI (https://archive.ics.uci.edu/ml/datasets/Wine); он состоит из 178 образцов белых вин с 13 признаками, описывающими их различные химические свойства.

Пользуясь библиотекой pandas, мы прочитаем набор данных вин непосредственно из репозитория машинного обучения UCI:

```
>>>
url = 'https://archive.ics.uci.edu/ml/machinelearning-databases/wine/wine.data'
df_wine = pd.read_csv(url, header=None)
df_wine.columns = ['Метка класса', 'Алкоголь',
                   'Яблочная кислота', 'Зола',
                   ‹Щелочность золы›, ‹Магний›,
                   ‹Всего фенола›, ‹Флаваноиды›,
                   ‹Фенолы нефлаваноидные›, ‹Проантоцианины›,
                   ‹Интенсивность цвета›, ‹Оттенок›,
                   'OD280/OD315 разбавленных вин', 'Пролин']
print('Метки классов: ', np.unique(df_wine['Метка класса']))
Метки классов: [1 2 3]
df_wine.head()
```

В следующей ниже таблице перечислены 13 различных признаков в наборе данных сортов вин **Wine**, описывающих химические свойства 178 образцов вин:

	Метка класса	Алкоголь	Яблочная кислота	Зола	Щелочность золы	Магний	Всего фенолов	Флаваноиды	Фенолы нефлаваноидные	Проантоцианины	Интенсивность цвета	Оттенок	OD280/ OD315 разбавленных вин	Пролин
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Данные образцы принадлежат одному из трех разных классов белых вин – 1, 2 и 3, которые относятся к трем различным типам винограда, выращенного в различных областях Италии.

Функция train_test_split из подмодуля отбора модели model_selection библиотеки scikit-learn (в версии библиотеки < 0.18 этот модуль назывался cross_validation) предоставляет удобный способ случайным образом разделить этот набор данных на отдельные *тестовый* и *тренировочный наборы данных*:

```
from sklearn.model_selection import train_test_split
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = \
         train_test_split(X, y, test_size=0.3, random_state=0)
```

Сначала мы присвоили переменной X представленные в виде массива NumPy признаковые столбцы 1–13, а переменной y присвоили метки классов из первого столбца. Затем мы применили функцию train_test_split, чтобы случайным образом разделить X и y на отдельные тренировочный и тестовый наборы данных. Установив параметр test_size=0.3, мы присвоили 30% образцов вин массивам X_test и y_test и оставшиеся 70% образцов – соответственно массивам X_train и y_train.

Следует учесть, что когда мы делим набор данных на тренировочный и тестовый наборы данных, то мы лишаемся части ценной информации, из которой алгоритм обучения мог бы извлечь выгоду. Поэтому мы не выделяем слишком много информации для тестового набора. Однако чем меньше тестовый набор, тем менее точной является оценка ошибки обобщения. Деление набора данных на тренировочный и тестовый наборы напрямую касается уравновешивания этого компромисса. На практике обычно используются пропорции разделения 60:40, 70:30 или 80:20, в зависимости от размера исходного набора данных. Однако для крупных наборов данных разделение на тренировочный и тестовый наборы в пропорциях 90:10 или 99:1 также распространено и приемлемо. Вместо того чтобы после тренировки и оценки модели отбросить выделенные тестовые данные, неплохо для настройки оптимального качества также переобучить классификатор на всем наборе данных.

Приведение признаков к одинаковой шкале

Масштабирование признаков является ключевым этапом в нашем конвейере **предобработки**, о котором можно легко забыть. Деревья решений и случайные леса – это одни из очень немногих алгоритмов машинного обучения, где нам не нужно беспокоиться о масштабировании признаков. Однако большинство алгоритмов машинного обучения и оптимизации ведет себя гораздо лучше, если признаки находятся в той же самой шкале, как мы убедились в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, где мы реализовали алгоритм оптимизации методом **градиентного спуска**.

Важность масштабирования признаков можно проиллюстрировать на простом примере. Допустим, имеются два признака, где один признак измеряется по шкале от 1 до 10, а второй – по шкале от 1 до 100 000. Если вспомнить функцию квадратичной ошибки в **ADALINE** из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, то первое, что приходит на ум, – это что алгоритм главным образом будет занят оптимизацией весов в соответствии с более крупными ошибками во втором признаке. Еще один пример касается алгоритма ***k* ближайших соседей (KNN)** с евклидовой мерой расстояния; ось второго признака будет занимать доминирующее положение в вычисленных расстояниях между образцами.

В силу вышесказанного существуют два общих подхода к приведению разных признаков к одинаковой шкале: **нормализация** и **стандартизация**. В различных областях эти термины нередко используются довольно нечетко, и их конкретное содержание приходится выводить из контекста. Чаще всего нормализация означает приведение (нормирование) признаков к диапазону [0, 1] и является частным случаем минимаксного масштабирования. Для нормализации наших данных можно к каждому признаковому столбцу просто применить минимаксное масштабирование, где новое значение $x_{\text{norm}}^{(i)}$ из образца $x^{(i)}$ можно вычислить следующим образом:

$$x_{\text{norm}}^{(i)} = \frac{x^{(i)} - x_{\min}}{x_{\max} - x_{\min}}.$$

Здесь $x^{(i)}$ – это отдельно взятый образец, $x_{\min}$ – наименьшее значение в признаковом столбце и $x_{\max}$ – соответственно наибольшее значение.

Процедура минимаксного масштабирования реализована в библиотеке scikit-learn и может быть применена следующим образом:

```
from sklearn.preprocessing import MinMaxScaler
mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

Несмотря на то что широко используемый метод нормализация путем минимаксного масштабирования целесообразно использовать, когда нам нужны значения в ограниченном интервале, для многих алгоритмов машинного обучения может быть более практичной стандартизация. Причина состоит в том, что многие линейные модели, такие как логистическая регрессия и метод опорных векторов (SVM), которые мы помним по *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, инициализируют веса нулями либо малыми случайными величинами, близкими к 0. При помощи стандартизации мы центрируем признаковые столбцы в нулевом среднем значении, т. е. равном 0, с единичным стандартным отклонением, т. е. равным 1, в результате чего признаковые столбцы принимают вид нормального распределения, что упрощает извлечение весов. Кроме того, стандартизация содержит полезную информацию о выбросах и делает алгоритм менее к ним чувствительным, в отличие от минимаксного масштабирования, которое шкалирует данные в ограниченном диапазоне значений.

Процедура стандартизации может быть выражена следующим уравнением:

$$x_{\text{std}}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}.$$

Здесь μ_x – это эмпирическое среднее отдельно взятого признакового столбца и σ_x – соответствующее стандартное отклонение.

Следующая ниже таблица иллюстрирует разницу между двумя обычно используемыми методами масштабирования признаков – стандартизацией и нормализацией, на простом демонстрационном наборе данных, состоящем из чисел от 0 до 5:

Вход	Стандартизировано	Нормализовано
0.0	−1.46385	0.0
1.0	−0.87831	0.2
2.0	−0.29277	0.4
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

Помимо класса минимаксного масштабирования MinMaxScaler, в библиотеке scikit-learn также реализован класс стандартизации StandardScaler:

```
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

Опять же важно подчеркнуть, что мы выполняем подгонку StandardScaler на тренировочных данных всего один раз и используем получившиеся параметры для преобразования тестового набора либо любой новой точки данных.

Отбор содержательных признаков

Если мы замечаем, что модель работает намного лучше на тренировочном наборе данных, чем на тестовом, то это наблюдение является сильным индикатором **переобучения**. Переобучение означает, что модель аппроксимирует параметры слишком близко к отдельно взятым наблюдениям в тренировочном наборе, но не обобщается хорошо на реальные данные – мы говорим, что модель имеет *высокую дисперсию*. Причина переобучения состоит в том, что наша модель слишком сложна для имеющихся тренировочных данных. Общие решения для снижения ошибки обобщения перечислены ниже:

- собрать больше тренировочных данных;
- ввести штраф за сложность на основе регуляризации;
- выбрать более простую модель с меньшим числом параметров;
- снизить размерность данных.

Сбор большего количества тренировочных данных часто выполнить невозможно. В следующей главе мы узнаем о широко применяемом методе проверки целесообразности введения дополнительных тренировочных данных. В следующих разделах и подразделах мы рассмотрим распространенные способы уменьшения переобучения путем регуляризации и снижения размерности, используя для этого отбор признаков.

Разреженные решения при помощи L1-регуляризации

Из *главы 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* мы помним, что **L2-регуляризация** – это один из подходов, применяемый для снижения сложности модели путем наложения штрафа на большие индивидуальные веса, где мы определили норму L2 нашего весового вектора w следующим образом:

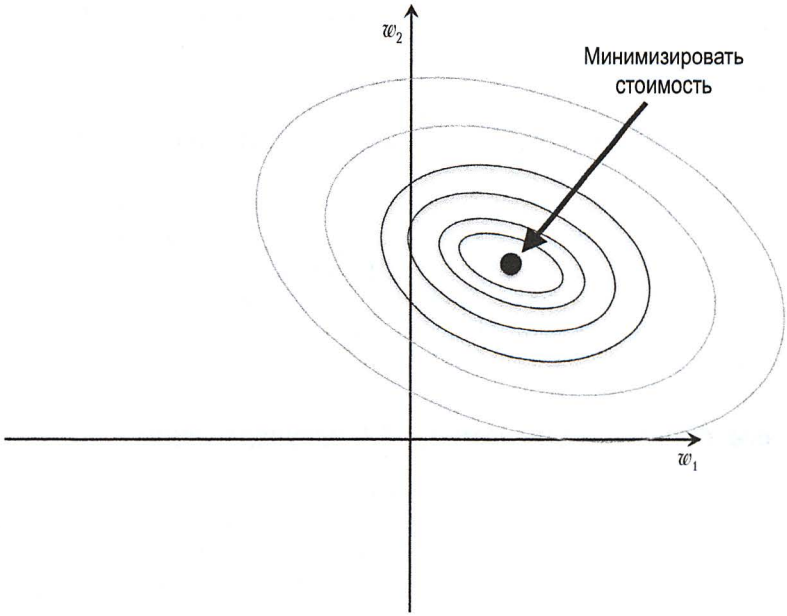
$$L2: \|w\|_2^2 = \sum_{j=1}^{m} w_j^2.$$

Еще один подход для снижения сложности модели связан с **L1-регуляризацией**:

$$L1: \|w\|_1 = \sum_{j=1}^{m} |w_j|.$$

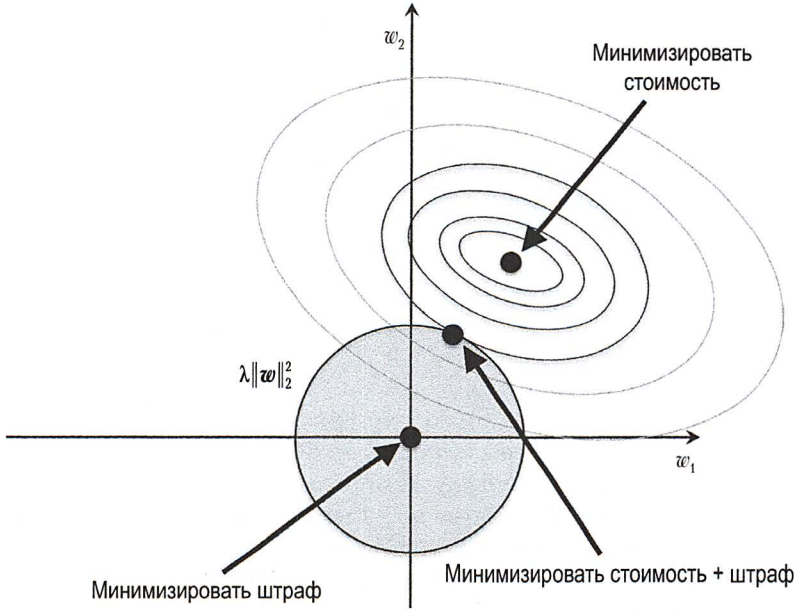
Здесь мы просто заменили квадрат весовых коэффициентов на сумму их абсолютных значений. В отличие от L2-регуляризации, L1-регуляризация производит разреженные векторы признаков; большинство весов признаков будет равно 0. Разреженность может быть полезной на практике при наличии высокоразмерного набора данных с большим числом нерелевантных (лишних) признаков, в особенности в тех случаях, где нерелевантных измерений больше, чем образцов. В этом смысле L1-регуляризация может пониматься как метод отбора признаков.

Чтобы лучше понять, каким образом L1-регуляризация поощряет разреженность, сделаем шаг назад и рассмотрим геометрическую интерпретацию регуляризации. Построим график контуров выпуклой функции стоимости для двух весовых коэффициентов w_1 и w_2. Здесь мы рассмотрим функцию стоимости с извлечением весов в виде **суммы квадратичных ошибок** (SSE), которую мы использовали для ADALINE в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, поскольку она симметрична и проще для изображения на графике, чем функция стоимости логистической регрессии; хотя к последней применимы те же самые принципы. Напомним, что наша задача состоит в том, чтобы найти комбинацию весовых коэффициентов, которые минимизируют функцию стоимости для тренировочных данных, как показано на нижеследующем рисунке (точка посередине эллипсов):



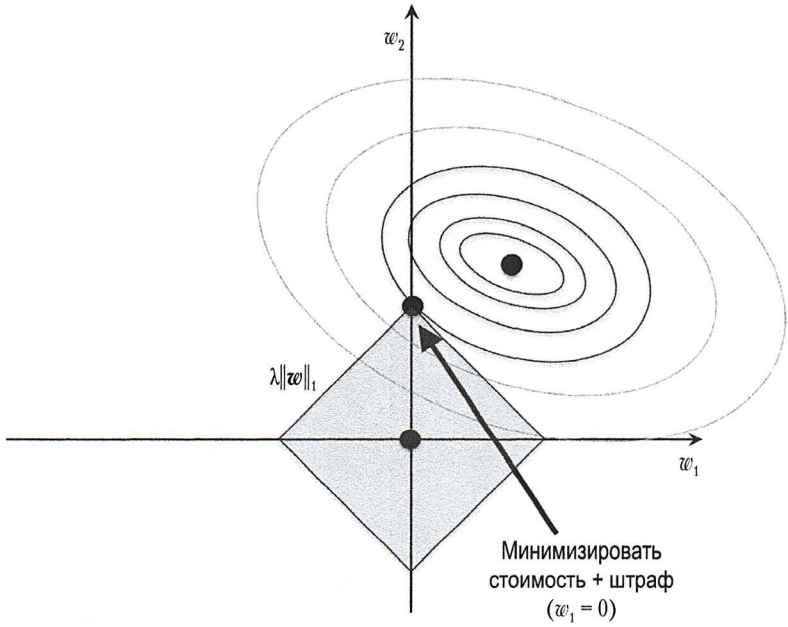
Теперь регуляризация может быть представлена как добавление к функции стоимости штрафного члена, который поощряет веса с меньшими значениями; другими словами, мы штрафуем большие веса.

Таким образом, увеличивая силу регуляризации параметром регуляризации λ, мы стягиваем веса к нулю и уменьшаем зависимость нашей модели от тренировочных данных. Проиллюстрируем эту идею на нижеследующем рисунке для штрафного члена L2.



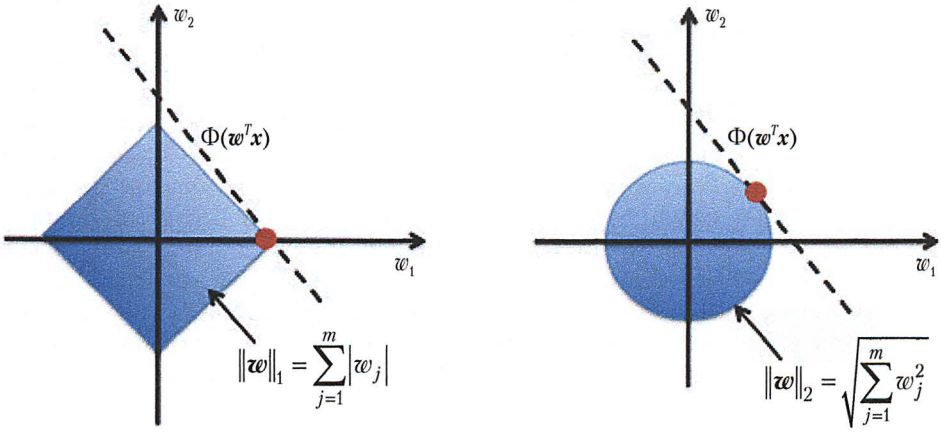
Член квадратичной L2-регуляризации представлен затененным кругом. Здесь наши весовые коэффициенты не могут превысить наш *запас* регуляризации – комбинация весовых коэффициентов не может выйти за пределы затененной области. С другой стороны, нам по-прежнему нужно минимизировать функцию стоимости. При штрафном ограничении самое лучшее – выбрать точки, где круг L2 пересекается с контурами неоштрафованной функции стоимости. Чем больше становится значение параметра регуляризации λ, тем быстрее растет оштрафованная функция стоимости, что в итоге приводит к более узкому кругу L2. Например, если увеличивать параметр регуляризации до бесконечности, то весовые коэффициенты фактически сведутся к нулю, обозначенному центром круга L2. Подытоживая основной смысл примера, наша задача состоит в том, чтобы минимизировать сумму неоштрафованной функции стоимости и штрафного члена, что может пониматься как добавление смещения и предпочтение более простой модели с целью уменьшения дисперсии при отсутствии достаточных тренировочных данных для подгонки модели.

Теперь обсудим L1-регуляризацию и разреженность. Ключевая идея, лежащая в основе L1-регуляризации, аналогична той, что мы здесь уже обсудили. Вместе с тем, поскольку L1-штраф – это сумма абсолютных весовых коэффициентов (напомним, что L2-член квадратичный), мы можем представить его как *запас* ромбовидной формы, как показано на нижеследующем рисунке:



На предыдущем рисунке видно, что контур функции стоимости касается ромба L1 в $w_1 = 0$. Поскольку контуры L1-регуляризованной системы остроконечные, скорее всего, оптимум, т. е. пересечение между эллипсами функции стоимости и границей L1-ромба, расположен на осях, которые поощряют разреженность. Математические подробности того, почему L1-регуляризация может привести к разряженным решениям, выходят за рамки этой книги. Если вам интересно, то превосходный раздел по L2-регуляризации в сопоставлении с регуляризацией L1 можно найти в разделе 3.4 книги «*The Elements of Statistical Learning*», Trevor Hastie, Robert Tibshirani and Jerome Friedman («Элементы статистического обучения», Т. Хасти, Р. Тибширани и Дж. Фридман) издательства «Springer».

Приведенный ниже рисунок резюмирует все вышесказанное о L1- и L2-регуляризации:



Чтобы для упорядоченных моделей, которые поддерживают L1-регуляризацию, получить разряженное решение, в библиотеке scikit-learn можно просто установить штрафной параметр penalty в 'l1':

```
from sklearn.linear_model import LogisticRegression
LogisticRegression(penalty='l1')
```

Если применить L1-регуляризованную логистическую регрессию к стандартизированным данным сортов вин, то она выдаст следующее ниже разряженное решение:

```
>>>
lr = LogisticRegression(penalty='l1', C=0.1)
lr.fit(X_train_std, y_train)
print('Верность на тренировочном наборе:', lr.score(X_train_std, y_train))
Верность на тренировочном наборе: 0.983870967742
print('Верность на тестовом наборе:', lr.score(X_test_std, y_test))
Верность на тестовом наборе: 0.981481481481
```

Оценки верности предсказания на тренировочном и тестовом наборах (обе 98%) не указывают на переобучение нашей модели. Если через атрибут lr.intercept обратиться к пересечениям, то мы увидим, что массив возвращает три значения[1]:

```
>>>
lr.intercept_
array([-0.38379237, -0.1580855 , -0.70047966])
```

Учитывая, что мы выполняем подгонку объекта LogisticRegression на многоклассовом наборе данных, подход **один против остальных** (OvR) используется по умолчанию, при этом первое значение точки пересечения принадлежит модели, которая оптимизирована под класс 1, по сравнению с классами 2 и 3; второе значение – это точка пересечения модели, оптимизированной под класс 2, по сравнению с классами 1 и 3; и третье значение – соответственно, точка пересечения модели, оптимизированной под класс 3, по сравнению с классами 1 и 2:

```
>>>
lr.coef_
array([[ 0.280, 0.000, 0.000, -0.0282, 0.000,
         0.000, 0.710, 0.000, 0.000, 0.000,
         0.000, 0.000, 1.236],
       [-0.644, -0.0688 , -0.0572, 0.000, 0.000,
         0.000, 0.000, 0.000, 0.000, -0.927,
         0.060, 0.000, -0.371],
       [ 0.000, 0.061, 0.000, 0.000, 0.000,
         0.000, -0.637, 0.000, 0.000, 0.499,
        -0.358, -0.570, 0.000
       ]])
```

Массив весов, к которому мы обратились через атрибут lr.coef, содержит три строки весовых коэффициентов, по одному весовому вектору для каждого класса.

---

[1] Функции в виде $y = f(x)$ могут содержать две и более точек пересечения оси X, а некоторые двухмерные математические отношения, такие как круги, эллипсы и гиперболы, могут иметь более одной точки пересечения оси Y. Точки пересечения оси X функций, если они есть, часто труднее локализовать, чем точку пересечения оси Y, т. к. нахождение последней заключается всего лишь в оценке функции при $x = 0$. – *Прим. перев.*

Каждая строка состоит из 13 весов, где каждый вес помножен на соответствующий признак в 13-мерном наборе данных сортов вин для вычисления чистого входа[1]:

$$z = w_1x_1 + \ldots + w_mx_m = \sum_{j=0}^{m} x_jw_j = \boldsymbol{w}^T\boldsymbol{x}.$$

Отмечаем для себя, что весовые векторы разрежены, т. е. имеют всего несколько ненулевых записей. В результате L1-регуляризации, которая служит в качестве метода отбора признаков, мы только что натренировали модель, устойчивую к потенциально нерелевантным (лишним) признакам в этом наборе данных.

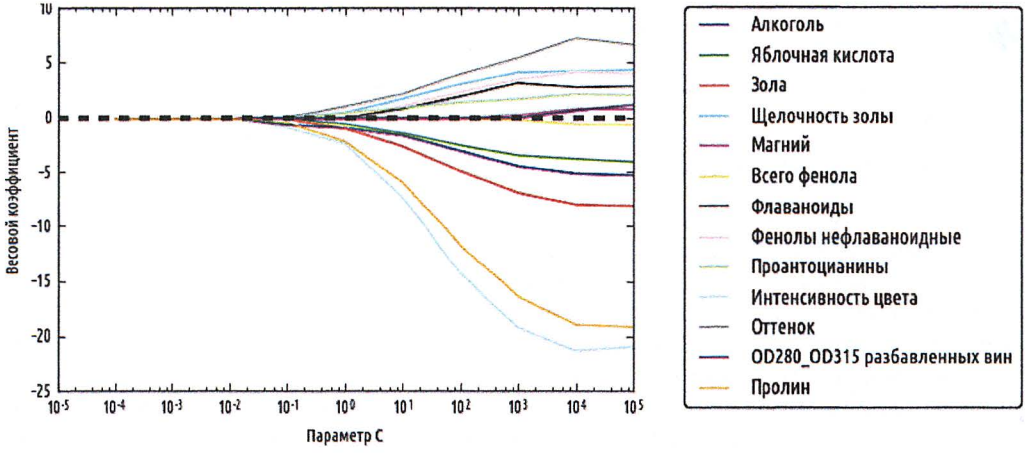
Наконец, построим график траекторий регуляризации, т. е. весовых коэффициентов, разных признаков для разных сил регуляризации:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.subplot(111)
colors = ['blue', 'green', 'red', 'cyan',
          'magenta', 'yellow', 'black',
          'pink', 'lightgreen', 'lightblue',
          'gray', 'indigo', 'orange']
weights, params = [], []
for c in np.arange(-4, 6):
    lr = LogisticRegression(penalty='l1',
                            C=10**c,
                            random_state=0)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10**c)
weights = np.array(weights)
for column, color in zip(range(weights.shape[1]), colors):
    plt.plot(params, weights[:, column],
             label=df_wine.columns[column+1],
             color=color)
plt.axhline(0, color='black', linestyle='--', linewidth=3)
plt.xlim([10**(-5), 10**5])
plt.ylabel('весовой коэффициент')
plt.xlabel('C')
plt.xscale('log')
plt.legend(loc='upper left')
ax.legend(loc='upper center',
          bbox_to_anchor=(1.38, 1.03),
          ncol=1, fancybox=True)
plt.show()
```

Итоговый график дает более глубокое понимание поведения L1-регуляризации. Как видно, все веса признаков будут обнулены, если штрафовать модель сильным параметром регуляризации ($C < 0.1$); C – это инверсия параметра регуляризации λ.

---

[1] Чтобы учесть постоянное смещение (bias unit), единицы должны быть заменены на нули (как в главе 2). При этом отметим, что scikit-learn хранит смещение и веса раздельно, и поэтому лучше записать так: $w_1x_1 + \ldots + w_mx_m + b = \sum_{j=0}^{m} x_jw_j + b = \boldsymbol{w}^T\boldsymbol{x} + b$. – *Прим. перев.*



Алгоритмы последовательного отбора признаков

Альтернативный способ уменьшить сложность модели и избежать переобучения представлен методикой **снижения размерности** посредством отбора признаков, которая особенно подходит для нерегуляризованных моделей. Существуют две основные категории методов снижения размерности: **отбор признаков** и **выделение признаков**. Используя отбор признаков, мы отбираем подмножество из исходных признаков. При выделении признаков мы извлекаем из набора признаков информацию для построения нового подпространства признаков. В этом разделе мы обратимся к классическому семейству алгоритмов отбора признаков. В следующей *главе 5 «Сжатие данных путем снижения размерности»* мы изучим другие методы выделения признаков, применяемые с целью сжатия набора данных в подпространство признаков более низкой размерности.

Последовательные алгоритмы отбора признаков принадлежат к семейству жадных алгоритмов поиска, которые используются для сведения начального *d-мерного* пространства признаков к *k-мерному* подпространству признаков, где $k < d$. В основе алгоритмов отбора признаков лежит побудительный мотив автоматически отобрать подмножество наиболее релевантных задаче признаков с целью повышения вычислительной эффективности либо уменьшить ошибку обобщения модели путем удаления нерелевантных признаков или шума, что может быть целесообразным для алгоритмов, которые не поддерживают регуляризацию. Классическим алгоритмом последовательного отбора признаков является алгоритм **последовательного обратного отбора** (sequential backward selection, SBS)[1], который стремится уменьшить размерность исходного подпространства признаков с минимальным ухудшением работоспособности классификатора при улучшении вычислительной эффективности. В определенных случаях SBS может даже улучшить предсказательную мощность модели, в случае если модель страдает от переобучения.

---

[1] Помимо алгоритма последовательного обратного отбора (SBS), еще используется последовательный прямой отбор (sequential forward selection, SFS), метод отбора «плюс l минус r» (plus-l minus-r selection), двунаправленный (bidirectional search) и плавающий поиск (floating search). См., например: http://research.cs.tamu.edu/prism/lectures/pr/pr_l11.pdf. – *Прим. перев.*

Жадные алгоритмы на каждом шаге комбинаторной задачи поиска делают локально оптимальный выбор и обычно производят субоптимальное решение задачи, в отличие от алгоритмов исчерпывающего поиска, которые оценивают все возможные комбинации и гарантированно находят оптимальное решение. Однако на практике исчерпывающий поиск часто в вычислительном плане не выполним, тогда как жадные алгоритмы допускают менее сложное и в вычислительном плане более эффективное решение.

В основе алгоритма SBS лежит довольно простая идея: SBS последовательно удаляет признаки из полнопризнакового подмножества, пока новое подпространство признаков не будет содержать нужное число признаков. Для определения того, какой признак удалять на каждом шаге, нам нужно определить критериальную функцию J, которую мы хотим минимизировать. Вычисленный данной функцией критерий может быть просто разницей в качестве классификатора после и до удаления отдельно взятого признака. Тогда удаляемый на каждом шаге признак можно просто определить как признак, который максимизирует этот критерий; или в более интуитивных терминах на каждом шаге мы устраняем признак, который вызывает наименьшую потерю качества после удаления. Основываясь на предыдущем определении алгоритма SBS, его можно в общих чертах обрисовать в 4 простых шагах:

1. Инициализировать алгоритм, назначив $k = d$, где d – это размерность полнопризнакового пространства.
2. Определить признак $\bar{x}$, который максимизирует критерий $\bar{x} = \text{argmax} J(X_k - x)$, где $x \in X_k$.
3. Удалить признак $\bar{x}$ из набора признаков: $X_{k-1} := X_k - \bar{x}$; $k := k - 1$.
4. Закончить, если k равняется числу требуемых признаков, в противном случае перейти к шагу 2.

Подробную оценку нескольких алгоритмов последовательного отбора признаков можно найти в работе Comparative Study of Techniques for Large Scale Feature Selection, F. Ferri, P. Pudil, M. Hatef and J. Kittler. «Comparative study of techniques for large-scale feature selection. Pattern Recognition in Practice IV», pages 403–413, 1994 («*Сравнительное исследование методов для крупномасштабного отбора признаков. Распознавание образов на практике IV*»).

К сожалению, алгоритм SBS в библиотеке scikit-learn пока не реализован. Но поскольку он очень прост, мы выполним его реализацию на Python с нуля:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.model_selection import train_test_split  # cross_validation
from sklearn.metrics import accuracy_score

class SBS():
    def __init__(self, estimator, k_features,
        scoring=accuracy_score,
        test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
```

```
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                             X_test, y_test, self.indices_)

        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim-1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train,
                    X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

В приведенной выше реализации мы установили параметр k_features в требуемое число признаков, которые мы хотим получить. По умолчанию мы используем метрику accuracy_score из библиотеки scikit-learn, которая оценивает качество модели, и оценщик для классификации на подмножествах признаков. Внутри цикла while в методе fit созданные функцией itertools.combination подмножества признаков оцениваются и сужаются, пока подмножество признаков не получит требуемую размерность. На каждой итерации оценка верности (accuracy_score) наилучшего подмножества накапливается в списке self.scores_, основываясь на создаваемом внутри тестовом наборе данных X_test. Мы воспользуемся этими показателями позже для оценки результатов. Индексы столбцов окончательного подмножества признаков назначаются списку self.indices_, который используется методом transform для возврата нового массива данных со столбцами отобранных признаков. Отметим, что, вместо того чтобы вычислять критерий явным образом внутри метода fit, мы просто удалили признак, который не содержится в подмножестве наиболее перспективных признаков.

Теперь посмотрим на нашу реализацию SBS в действии, воспользовавшись для этого классификатором KNN из библиотеки scikit-learn:

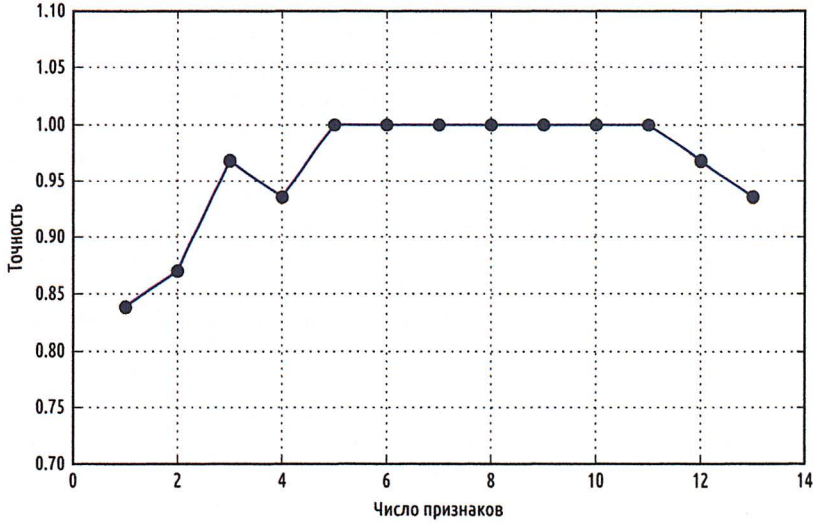
```
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
knn = KNeighborsClassifier(n_neighbors=2)
sbs = SBS(knn, k_features=1)
sbs.fit(X_train_std, y_train)
```

Несмотря на то что наша реализация SBS и так уже разделяет набор данных внутри функции fit на тестовый и тренировочный наборы данных, мы все же подали на вход алгоритма тренировочный набор данных X_train. Метод fit класса SBS затем создаст новые тренировочные подмножества для тестирования (валидации) и тренировки. По этой причине этот тестовый набор также называется **проверочным (валидационным) набором данных**. *Этот подход необходим для того, чтобы не дать нашему исходному тестовому набору стать составной частью тренировочных данных.*

Напомним, что наш алгоритм SBS на каждом шаге накапливает оценки в подмножестве лучших признаков, поэтому перейдем к более захватывающей части нашей реализации и построим график верности классификации классификатора KNN, который был вычислен на перекрестно-проверочном наборе данных. Соответствующий исходный код выглядит следующим образом:

```
k_feat = [len(k) for k in sbs.subsets_]
plt.plot(k_feat, sbs.scores_, marker='o')
plt.ylim([0.7, 1.1])
plt.ylabel('Верность')
plt.xlabel('Число признаков')
plt.grid()
plt.show()
```

Как видно на следующем ниже графике, верность классификатора KNN улучшилась на проверочном наборе данных, когда мы уменьшили число признаков, что, скорее всего, происходит из-за того, что снизилось **проклятие размерности**, явление, которое мы обсудили в контексте алгоритма KNN в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*. Кроме того, на графике видно, что классификатор достиг 100%-ной верности для $k = \{5, 6, 7, 8, 9, 10\}$:



Ради удовлетворения собственного любопытства посмотрим, какие это пять признаков, которые привели к такому хорошему качеству на проверочном наборе данных:

```
>>>
k5 = list(sbs.subsets_[8])
print(df_wine.columns[1:][k5])
Index(['Алкоголь', 'Яблочная кислота', 'Щелочность золы', 'Оттенок', 'Пролин'],
dtype='object')
```

Используя приведенный выше исходный код, мы получили индексы столбцов 5-признакового подмножества из 9-й позиции в атрибуте sbs.subsets_ и вернули соответствующие имена признаков из столбцового индекса таблицы данных о сортах вин DataFrame.

Теперь выполним оценку качества классификатора KNN на исходном тестовом наборе:

```
>>>
knn.fit(X_train_std, y_train)

print('Верность на тренировочном наборе:', knn.score(X_train_std, y_train))
Верность на тренировочном наборе: 0.983870967742
print('Верность на тестовом наборе:', knn.score(X_test_std, y_test))
Верность на тестовом наборе: 0.944444444444
```

В предыдущем примере мы использовали полный набор признаков и получили верность ~98.4% на тренировочном наборе данных. Однако верность на тестовом наборе данных была немного ниже (~94.4%), что указывает на легкую степень переподгонки. Теперь воспользуемся отобранным 5-признаковым подмножеством и посмотрим на качество KNN:

```
>>>
knn.fit(X_train_std[:, k5], y_train)
print('Верность на тренировочном наборе:', knn.score(X_train_std[:, k5], y_train))
Верность на тренировочном наборе: 0.959677419355

print('Верность на тестовом наборе:', knn.score(X_test_std[:, k5], y_test))
Верность на тестовом наборе: 0.962962962963
```

При меньшем, чем наполовину, числе исходных признаков из набора данных сортов вин верность предсказания на тестовом наборе улучшилась почти на 2%. Кроме того, мы уменьшили переобучение, которую можно определить по малому разрыву между верностью на тестовом наборе (~96.3%) и верностью на тренировочном наборе (~96.0%).

Алгоритмы отбора признаков в библиотеке scikit-learn

Библиотека scikit-learn располагает многими другими алгоритмами отбора признаков. Они включают в себя рекурсивный алгоритм обратного устранения признаков на основе весовых коэффициентов признаков, методы отбора признаков по важности на основе деревьев и одномерные статистические методы. Всестороннее обсуждение различных методов отбора признаков выходит за рамки этой книги. Впрочем, хороший сводный обзор с показательными примерами можно найти на http://scikit-learn.org/stable/modules/feature_selection.html.

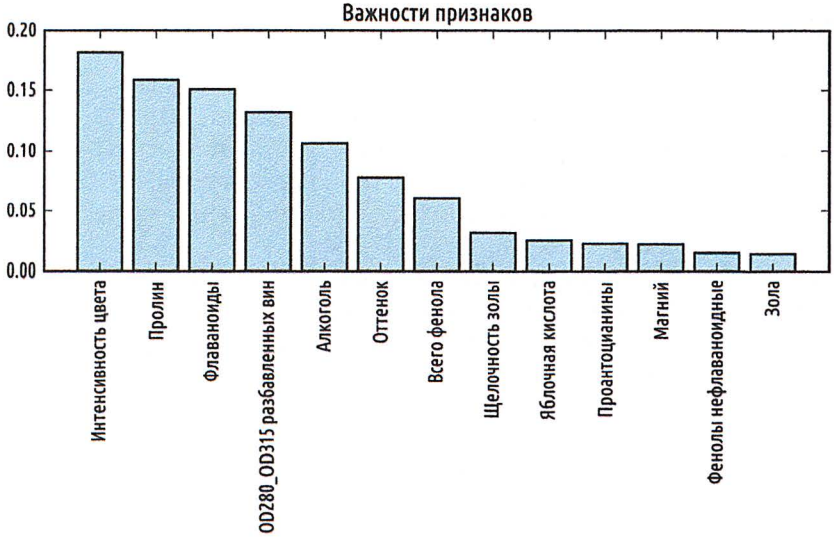
Определение важности признаков при помощи случайных лесов

В предыдущих разделах вы научились использовать L1-регуляризацию для обнуления нерелевантных признаков посредством логистической регрессии и использовать алгоритм SBS для отбора признаков. Еще один подход, который используется для отбора релевантных признаков из набора данных, состоит в применении ансамблевого метода на основе случайного леса, который мы представили в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*. Используя случайный лес, можно измерить важность признака как усредненное уменьшение неоднородности, вычисленное из всех деревьев решений в лесе, не делая никаких допущений по поводу наличия либо отсутствия линейной разделимости данных. Удобно то, что в реализации случайных лесов в библиотеке scikit-learn уже предусмотрено аккумулирование важностей признаков, и к ним можно обращаться через атрибут feature_importances_ после подгонки классификатора RandomForestClassifier. Выполнив приведенный ниже исходный код, натренируем лес из 10 000 деревьев на наборе данных сортов вин и упорядочим 13 признаков по их соответствующим мерам важности. Напомним (из нашего обсуждения в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*), что модели на основе деревьев в стандартизации или нормализации не нуждаются. Соответствующий исходный код выглядит следующим образом:

```
>>>
from sklearn.ensemble import RandomForestClassifier
feat_labels = df_wine.columns[1:]
forest = RandomForestClassifier(n_estimators=10000,
                                random_state=0,
                                n_jobs=-1)
forest.fit(X_train, y_train)
importances = forest.feature_importances_
indices = np.argsort(importances)[::-1]
for f in range(X_train.shape[1]):
    print("%2d) %-*s %f" % (f + 1, 30,
                            feat_labels[indices[f]],
                            importances[indices[f]]))
 1) Интенсивность цвета            0.182483
 2) Пролин                         0.158610
 3) Флаваноиды                     0.150948
 4) OD280_OD315 разбавленных вин   0.131987
 5) Алкоголь                       0.106589
 6) Оттенок                        0.078243
 7) Всего фенола                   0.060718
 8) Щелочность золы                0.032033
 9) Яблочная кислота               0.025400
10) Проантоцианины                 0.022351
11) Магний                         0.022078
12) Фенолы нефлаваноидные          0.014645
13) Зола                           0.013916

plt.title('Важности признаков')
plt.bar(range(X_train.shape[1]), importances[indices],
              color='lightblue', align='center')
plt.xticks(range(X_train.shape[1]),
```

```
            feat_labels[indices], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
plt.show()
```

После выполнения приведенного выше примера будет создан график, в котором разные признаки из набора данных сортов вин упорядочены по их относительной важности; отметим, что важности признаков нормализованы, в результате чего они в сумме составляют 1.0.



Можно заключить, что интенсивность цвета вина является самым отличительным признаком в наборе данных, основываясь на усредненном уменьшении неоднородности в 10 000 деревьях решений. Интересно, что три находящихся на вершине рейтинга признака в приведенном выше графике также находятся среди пяти лучших признаков, полученных в результате отбора, выполненного алгоритмом SBS, который мы реализовали в предыдущем разделе. Однако что касается интерпретируемости, то метод на основе случайных лесов несет в себе один важный *изъян*, о котором стоит упомянуть. Например, если два или больше признаков высоко коррелируются, то один из признаков может получить очень высокую степень важности, тогда как информация о другом признаке (или признаках) может оказаться не захваченной полностью. С другой стороны, нам не нужно беспокоиться по этому поводу, если нас интересует только предсказательная способность модели, а не интерпретация важностей признаков. В заключение этого раздела о важностях признаков и случайных лесах стоит упомянуть, что в библиотеке scikit-learn также реализован метод `transform`, который отбирает признаки, основываясь на определенном пользователями пороге после подгонки модели, который очень часто применяется в случае, если мы хотим использовать `RandomForestClassifier` в качестве селектора признаков и промежуточного звена в конвейере библиотеки scikit-learn; это позволяет соединять разные шаги предобработки с оценщиком, как мы убедимся в *главе 6 «Изучение наиболее успешных методов оценки модели и тонкой настройки гиперпа-*

раметров». К примеру, можно установить порог в 0.15 для сведения набора данных к 3 наиболее важным признакам – **интенсивность цвета**, **пролин** и **флавоноиды**, используя для этого нижеследующий исходный код:

```
>>>
X_selected = forest.transform(X_train, threshold=0.15)
X_selected.shape
(124, 3)
```

Резюме

Мы начали эту главу с того, что рассмотрели часто используемые методы, которые обеспечивают правильную обработку пропущенных данных. Прежде чем передать данные на вход алгоритма машинного обучения, мы также должны удостовериться, что категориальные переменные имеют правильную кодировку, и мы увидели, каким образом можно переводить порядковые и номинальные значения признаков в целочисленные представления.

Кроме того, мы кратко обсудили L1-регуляризацию, которая помогает избежать переобучение путем уменьшения сложности модели. В качестве альтернативного подхода к удалению нерелевантных признаков мы применили алгоритм последовательного отбора признаков для отбора содержательных признаков из набора данных.

В следующей главе вы узнаете о еще одном подходе, который часто используется для снижения размерности: выделении признаков. Он позволяет сжимать признаки в подпространство более низкой размерности без удаления признаков полностью, как при отборе признаков.

Сжатие данных путем снижения размерности

В *главе 4 «Создание хороших тренировочных наборов – предобработка данных»* вы узнали о разных подходах к решению задачи снижения размерности набора данных на основе различных методов отбора признаков. Помимо методики отбора признаков, альтернативным подходом к решению этой задачи является методика *выделения признаков*. В этой главе вы узнаете о трех основополагающих методах, которые помогут резюмировать информационное содержание набора данных путем преобразования его в новое подпространство признаков более низкой размерности, чем исходное. В машинном обучении методы сжатия данных представляют собой важную тему; они помогают накапливать и анализировать увеличивающиеся объемы данных, производимые и собираемые в нашу современную технологическую эпоху. В этой главе мы охватим следующие темы:

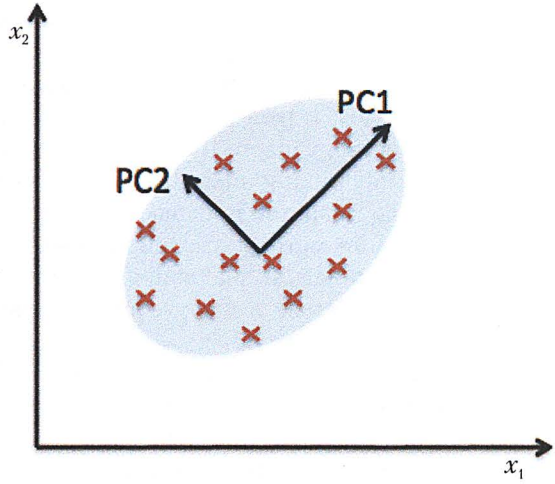
- **анализ главных компонент** для сжатия данных без учителя;
- **линейный дискриминантный анализ** как метод снижения размерности с учителем для максимизации разделимости классов;
- нелинейное снижение размерности **ядерным методом** анализа **главных компонент** (ядерного PCA, kernel PCA).

Снижение размерности без учителя на основе анализа главных компонент

Аналогично отбору признаков выделение признаков можно использовать для снижения числа признаков в наборе данных. Однако, в отличие от использования алгоритмов отбора признаков, таких как *последовательный обратный отбор*, где мы оставляем исходные признаки нетронутыми, выделение признаков используется, чтобы преобразовать или спроецировать данные на новое пространство признаков. В контексте снижения размерности выделение признаков может пониматься как подход к сжатию данных с сохранением большинства релевантной информации. Выделение признаков обычно используется для повышения вычислительной эффективности, но может также помочь уменьшить проблему *проклятия размерности* – в особенности если мы работаем с нерегуляризованными моделями.

Анализ главных компонент (principal component analysis, PCA) – это метод линейного преобразования, относящийся к типу обучения без учителя, который широко используется в самых разных областях, чаще всего для снижения размерности. Другие распространенные применения PCA лежат в области методов разведочного анализа данных и шумоподавления сигналов в биржевой торговле и анализа геном-

ных данных и уровней экспрессии генов в биоинформатике. PCA помогает идентифицировать повторяющиеся образы в данных, основываясь на корреляции между признаками. Если кратко, то PCA стремится находить направления с максимальной дисперсией в высокоразмерных данных и проецирует их на новое подпространство с равным или меньшим числом размерностей, чем исходное. Ортогональные оси (основные компоненты) нового подпространства можно интерпретировать как направления максимальной дисперсии в условиях ограничения, что оси новых признаков ортогональны друг другу, как проиллюстрировано на нижеследующем рисунке. Здесь x_1 и x_2 – это исходные оси признаков, а **PC1** и **PC2** – основные компоненты:



Если PCA используется для снижения размерности, то мы создаем матрицу преобразования $\boldsymbol{W}$ размера $d \times k$, которая позволяет отображать вектор образца $\boldsymbol{x}$ на новое k-мерное подпространство признаков, у которого меньше размерностей, чем у исходного d-мерного пространства признаков:

$\boldsymbol{x} = [x_1, x_2, ..., x_d]$, $\boldsymbol{x} \in \mathbb{R}^d$;
$\downarrow \boldsymbol{x}\boldsymbol{W}$, $\boldsymbol{W} \in \mathbb{R}^{d \times k}$;
$\boldsymbol{z} = [z_1, z_2, ..., z_k]$, $\boldsymbol{z} \in \mathbb{R}^k$.

В результате преобразования исходных d-мерных данных в новое k-мерное подпространство (обычно $k << d$) первая главная компонента будет иметь самую большую дисперсию, а все последующие – самую большую дисперсию при условии, что они некоррелированы (ортогональны) с другими главными компонентами. Отметим, что направления PCA очень чувствительны к шкале данных, и в случае если признаки измеряются в разных шкалах и мы хотим назначить всем признакам равную важность, то *перед* применением PCA нам нужно выполнить стандартизацию признаков.

Прежде чем рассмотреть алгоритм PCA для снижения размерности более подробно, сведем этот подход к нескольким простым шагам:

1) стандартизировать d-мерный набор данных;
2) построить ковариационную матрицу;

3) разложить ковариационную матрицу на ее собственные векторы и собственные значения (числа);
4) выбрать k собственных векторов, которые соответствуют k самым большим собственным значениям, где k – размерность нового подпространства признаков ($k \leq d$);
5) создать проекционную матрицу $\boldsymbol{W}$ из «верхних» k собственных векторов;
6) преобразовать d-мерный входной набор данных $\boldsymbol{X}$, используя проекционную матрицу $\boldsymbol{W}$ для получения нового k-мерного подпространства признаков.

Общая и объясненная дисперсия

В этом подразделе мы займемся первыми четырьмя шагами анализа главных компонент: стандартизацией данных, построением ковариационной матрицы, получением собственных значений и собственных векторов ковариационной матрицы и сортировкой собственных значений в порядке их убывания с целью ранжирования собственных векторов.

Сначала начнем с загрузки набора данных сортов вин, с которым мы работали в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»*:

```
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machinelearning-databases/wine/wine.data'
df_wine = pd.read_csv(url, header=None)
```

Затем обработаем данные сортов вин, разбив их на раздельные тренировочный и тестовый наборы соответственно в пропорции 70% на 30% данных, и стандартизируем их, приведя к единичной дисперсии.

```
from sklearn.model_selection import train_test_split  # cross_validation
from sklearn.preprocessing import StandardScaler
X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
X_train, X_test, y_train, y_test = \
            train_test_split(X, y,
            test_size=0.3, random_state=0)
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)
```

После завершения обязательных шагов предобработки, содержащихся в приведенном выше фрагменте исходного кода, перейдем ко второму шагу: построению ковариационной матрицы. В симметричной ковариационной $d \times d$-матрице, где d – число размерностей в наборе данных, хранятся попарные ковариации (согласованные отклонения) между разными признаками. Например, ковариацию между двумя признаками x_j и x_k на уровне генеральной совокупности можно вычислить при помощи следующего равенства:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}(x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k).$$

Здесь μ_j и μ_k – это эмпирические средние соответственно признаков j и k. Отметим, что эмпирические средние равны 0, если набор данных стандартизирован. Положительная ковариация между двумя признаками указывает на то, что призна-

ки увеличиваются либо уменьшаются вместе, тогда как отрицательная ковариация говорит о том, что признаки варьируются в противоположных направлениях. Тогда, к примеру, ковариационную матрицу из трех признаков можно записать (отметим, что знак Σ обозначает греческую букву *сигма*, которую не следует путать с символом *суммы*) как:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}.$$

Собственные векторы ковариационной матрицы представляют главные компоненты (направления максимальной дисперсии), тогда как соответствующие собственные значения определяют их величину (длину)[1]. В случае набора данных сортов вин мы получим 13 собственных векторов и собственных значений из ковариационной 13×13-матрицы.

Теперь получим собственные пары[2] ковариационной матрицы. Как мы, конечно, помним из вводного курса по линейной алгебре и вычислительным методам, собственный вектор $\boldsymbol{v}$ удовлетворяет следующему условию:

$$\Sigma \boldsymbol{v} = \lambda \boldsymbol{v}.$$

Здесь λ – это скаляр, т. е. собственное значение. Поскольку вычисление собственных векторов и собственных значений ручным способом является несколько утомительной и довольно трудоемкой задачей, для получения собственных пар ковариационной матрицы сортов вин мы воспользуемся функцией linalg.eig из библиотеки NumPy:

```
>>>
import numpy as np
cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
print('\nСобственные значения \n%s' % eigen_vals)
Собственные значения
[ 4.8923083 2.46635032 1.42809973 1.01233462 0.84906459
0.60181514
0.52251546 0.08414846 0.33051429 0.29595018 0.16831254 0.21432212
0.2399553 ]
```

При помощи функции numpy.cov мы вычислили ковариационную матрицу стандартизированного тренировочного набора данных. А при помощи функции linalg.eig, которая вернула вектор eigen_vals, состоящий из 13 собственных значений и соответствующих собственных векторов, хранящихся как столбцы в 13×13-матрице eigen_vecs, выполнили разложение по собственным значениям.

---

[1] В евклидовом векторном пространстве величина вектора (magnitude), или длина вектора, определяется как его евклидова норма (или евклидова длина), т. е. как квадратный корень из суммы квадратов элементов вектора: $\|x\| = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$. – *Прим. перев.*

[2] Собственная пара матрицы – это вещественное число λ и вектор z матрицы A, если они удовлетворяют условию, что $Av = \lambda v$.– *Прим. перев.*

Хотя функция numpy.linalg.eig предназначена для разложения несимметричных квадратных матриц, вы можете обнаружить, что в определенных случаях она возвращает комплексные собственные значения.
Родственная функция numpy.linalg.eigh реализована для разложения эрмитовых (самосопряженных) матриц, которые представляют собой численно более стабильный подход для работы с симметричными матрицами, такими как ковариационная матрица; функция numpy.linalg.eigh всегда возвращает вещественные собственные значения.

Учитывая, что мы хотим снизить размерность нашего набора данных путем его сжатия в новое подпространство признаков, мы отберем лишь подмножество собственных векторов (главных компонент), которые содержат бóльшую часть информации (дисперсии). Поскольку собственные значения определяют величину собственных векторов, нам нужно отсортировать собственные значения в порядке убывания величины векторов; нас интересуют верхние k собственных векторов, основываясь на значениях соответствующих им собственных значений. Но прежде чем собрать эти k наиболее информативных собственных векторов, построим график *долей объясненной дисперсии* собственных значений.

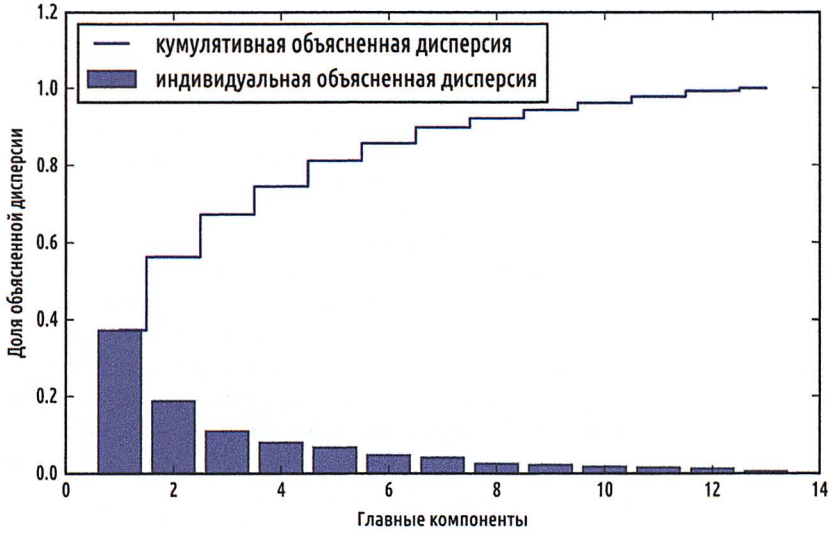
Доля объясненной дисперсии собственного значения λ_j – это просто частное собственного значения λ_j и полной суммы собственных значений:

$$\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}.$$

Тогда при помощи функции cumsum библиотеки NumPy можно вычислить кумулятивную сумму объясненных дисперсий, которую затем можно отобразить на графике, воспользовавшись функцией step библиотеки matplotlib:

```
tot = sum(eigen_vals)
var_exp = [(i / tot) for i in
           sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)
import matplotlib.pyplot as plt
plt.bar(range(1,14), var_exp, alpha=0.5, align='center',
        label='индивидуальная объясненная дисперсия')
plt.step(range(1,14), cum_var_exp, where='mid',
         label='кумулятивная объясненная дисперсия')
plt.ylabel('Доля объясненной дисперсии')
plt.xlabel('Главные компоненты')
plt.legend(loc='best')
plt.show()
```

Итоговый график показывает, что только на одну первую главную компоненту приходятся 40% дисперсии. Кроме того, видно, что первые две объединенные главные компоненты объясняют почти 60% дисперсии в данных:



Несмотря на то что график коэффициентов объясненной дисперсии напоминает нам о важностях признаков, которые мы вычислили с использованием случайных лесов в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»*, следует напомнить, что PCA – это метод машинного обучения без учителя, т. е. информация о метках классов игнорируется. В то время как случайный лес для вычисления неоднородностей в узлах использует информацию о принадлежности классу, дисперсия измеряет разброс значений вдоль оси признака.

Преобразование признаков

После того как мы успешно разложили ковариационную матрицу на собственные пары, теперь перейдем к последними трем шагам, чтобы преобразовать набор данных сортов вин на новые оси главных компонент. В этом разделе мы отсортируем собственные пары в порядке убывания собственных значений, построим проекционную матрицу из отобранных собственных векторов и воспользуемся проекционной матрицей для преобразования данных в подпространство более низкой размерности.

Начнем с сортировки собственных пар в порядке убывания собственных значений:

```
eigen_pairs =[(np.abs(eigen_vals[i]),eigen_vecs[:,i])
              for i in range(len(eigen_vals))]
eigen_pairs.sort(reverse=True)
```

Затем возьмем два собственных вектора, соответствующих двум самым большим значениям, которые захватывают примерно 60% дисперсии в этом наборе данных. Отметим, что мы отобрали всего два собственных вектора в целях иллюстрации, поскольку позже в этом подразделе собираемся отобразить данные на двумерном точечном графике (диаграмме рассеяния). На практике число главных компонент следует определять, исходя из компромисса между вычислительной эффективностью и качеством классификатора:

```
>>>
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
               eigen_pairs[1][1][:, np.newaxis]))
print('Матрица W:\n',w)
Матрица W:
[[ 0.14669811 0.50417079]
[-0.24224554  0.24216889]
[-0.02993442  0.28698484]
[-0.25519002 -0.06468718]
[ 0.12079772  0.22995385]
[ 0.38934455  0.09363991]
[ 0.42326486  0.01088622]
[-0.30634956  0.01870216]
[ 0.30572219  0.03040352]
[-0.09869191  0.54527081]
[ 0.30032535 -0.27924322]
[ 0.36821154 -0.174365  ]
[ 0.29259713  0.36315461]]
```

Выполнив приведенный выше фрагмент кода, мы создали проекционную 13×2-матрицу $\boldsymbol{W}$ из двух верхних собственных векторов. При помощи проекционной матрицы теперь можно преобразовать образец $\boldsymbol{x}$ (представленную как вектор-строка размера 13×1) на подпространство PCA, получив теперь уже 2-мерный вектор $\boldsymbol{x}'$ образца, состоящий из двух новых признаков:

$$\boldsymbol{x}' = \boldsymbol{x}\boldsymbol{W}.$$

```
>>>
X_train_std[0].dot(w)
array([ 2.59891628, 0.00484089])
```

Точно так же можно преобразовать весь 124×13-мерный тренировочный набор данных на две главные компоненты, вычислив скалярное произведение матриц:

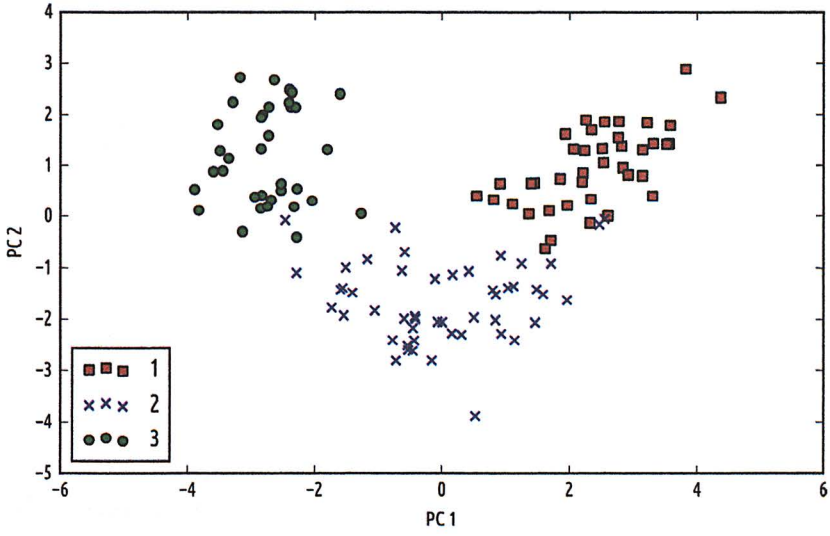
$$\boldsymbol{X}' = \boldsymbol{X}\boldsymbol{W}.$$

```
X_train_pca = X_train_std.dot(w)
```

Наконец, представим преобразованный тренировочный набор сортов вин, теперь уже хранящийся в виде 124×2-матрицы, на двумерном точечном графике:

```
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_pca[y_train==l, 0],
                X_train_pca[y_train==l, 1],
                c=c, label=l, marker=m)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.show()
```

Как видно на итоговом графике, данные более разбросаны вдоль оси X – первая главная компонента, чем вдоль второй главной компоненты (ось Y), что согласуется с графиком долей объясненной дисперсии, который мы построили в предыдущем подразделе. Вместе с тем интуитивно можно отметить, что линейный классификатор, вероятно, сможет хорошо разделить классы:



Хотя в целях иллюстрации в приведенный выше точечный график мы включили информацию о метках классов, следует иметь в виду, что PCA является методом обучения без учителя, в котором информация о метках классов не используется.

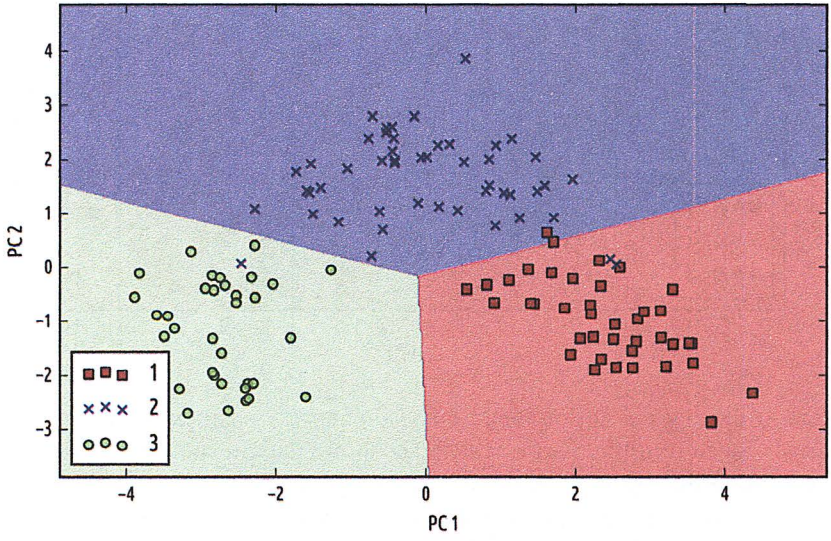
Анализ главных компонент в scikit-learn

Несмотря на то что подробный подход в предыдущем подразделе помог нам взглянуть на работу PCA изнутри, тем не менее сейчас мы обсудим применение класса PCA, который реализован в библиотеке scikit-learn. Класс PCA – это еще один класс-преобразователь, в котором мы сначала выполняем подгонку модели, используя для этого тренировочные данные, прежде чем беремся за преобразование тренировочных и тестовых данных с использованием тех же самых параметров модели. Теперь применим класс PCA библиотеки scikit-learn к тренировочному набору данных сортов вин, выполним классификацию преобразованных образцов посредством логистической регрессии и представим области решений на графике при помощи функции plot_decision_region, которую мы определили в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*:

```
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):

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

    # вывести поверхности решений
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
```

```
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # вывести образцы классов
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=cmap(idx),
                    marker=markers[idx], label=cl)

from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
lr = LogisticRegression()
X_train_pca = pca.fit_transform(X_train_std)
X_test_pca = pca.transform(X_test_std)
lr.fit(X_train_pca, y_train)
plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend(loc='lower left')
plt.show()
```

После выполнения приведенного выше исходного кода мы должны увидеть, что теперь области решений для тренировочной модели снизились до двух главных осей (главных компонент).

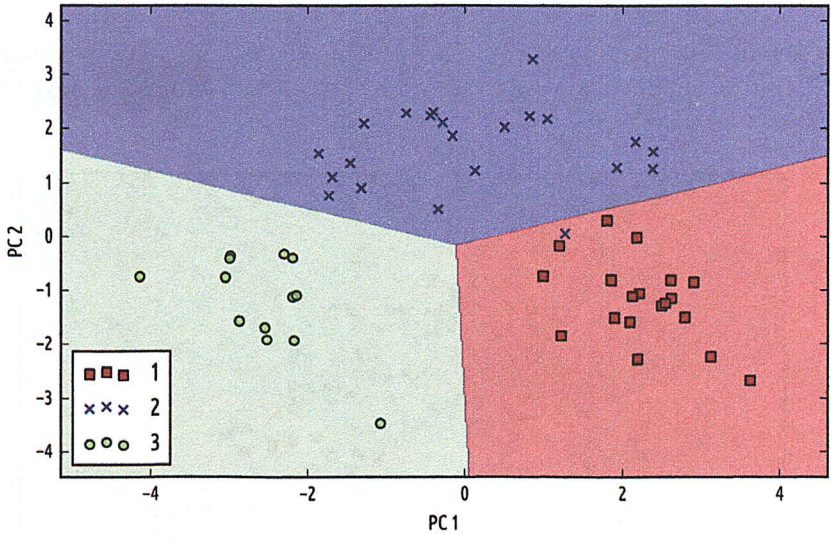


Если сравнить проекцию PCA, выполненную с использованием библиотеки scikit-learn, с нашей собственной реализацией PCA, то можно увидеть, что приведенная выше диаграмма представляет собой зеркальное отражение предыдущей диаграммы PCA, полученной в результате нашего пошагового подхода. Отметим, что это не следствие ошибки в одной из этих двух реализаций; причина такой разницы состоит в том, что в зависимости от алгоритма вычисления собственных векторов и значе-

ний собственные векторы могут иметь отрицательный либо положительный знак. Не то, чтобы это имеет значение, но в случае необходимости мы можем просто обратить зеркальное отражение, умножив данные на -1; отметим также, что собственные векторы, как правило, нормированы, т. е. в результате процедуры масштабирования приведены к единичной длине (равной 1). Ради полноты построим график областей решения логистической регрессии на преобразованном тестовом наборе данных, чтобы увидеть, сможет ли она хорошо разделить классы:

```
plot_decision_regions(X_test_pca, y_test, classifier=lr)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.legend(loc='lower left')
plt.show()
```

Построив график областей решения для тестового набора в результате выполнения предыдущего исходного кода, мы увидим, что логистическая регрессия вполне хорошо работает на этом небольшом двухмерном подпространстве признаков, ошибочно классифицируя в тестовом наборе данных всего один образец.



Если нас интересуют доли объясненных дисперсий разных главных компонент, то мы можем просто инициализировать класс PCA с параметром n_components, установленным в None, чтобы сохранить все главные компоненты, и тогда к доле объясненной дисперсии можно обратиться через атрибут explained_variance_ratio_:

```
>>>
pca = PCA(n_components=None)
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
array([ 0.37329648, 0.18818926, 0.10896791, 0.07724389,
0.06478595,
0.04592014, 0.03986936, 0.02521914, 0.02258181, 0.01830924,
0.01635336, 0.01284271, 0.00642076])
```

Отметим, что при инициализации класса `PCA` мы устанавливаем параметр `n_components=None`, в результате чего вместо выполнения снижения размерности он возвращает все главные компоненты в отсортированном виде.

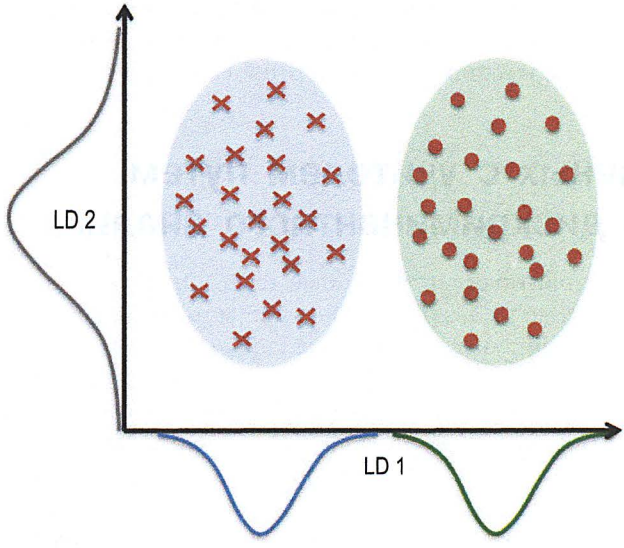
Сжатие данных с учителем путем линейного дискриминантного анализа

Линейный дискриминантный анализ (linear discriminant analysis, LDA), он же канонический, может использоваться в качестве метода для выделения признаков в целях увеличения вычислительной эффективности и уменьшения степени переподгонки из-за проблемы проклятия размерности в нерегуляризованных моделях.

В основе LDA лежит общая идея, которая очень похожа на PCA. Правда, PCA пытается найти ортогональные оси компонент с максимальной дисперсией в наборе данных, тогда как задача LDA – найти подпространство признаков, которое оптимизирует разделимость классов. Оба метода, PCA и LDA, являются методами линейного преобразования, которые могут использоваться для снижения числа размерностей в наборе данных, где первый алгоритм – без учителя, а второй – с учителем. Поэтому интуитивно можно подумать, что метод выделения признаков на основе LDA лучше подходит для задач классификации, чем PCA. Тем не менее, по сведениям А. М. Мартинеса, в определенных случаях предобработка методом PCA, как правило, позволяет добиваться лучших результатов классификации при выполнении задачи распознавания образов, например если каждый класс состоит из небольшого числа образцов (A. M. Martinez, A. C. Kak. *PCA Versus LDA. Pattern Analysis and Machine Intelligence* («PCA против LDA. Анализ образов и искусственный интеллект»). IEEE Transactions on, 23(2):228-233, 2001).

Хотя LDA иногда также называют методом LDA Фишера, Рональд А. Фишер изначально в 1936 г. сформулировал свой линейный дискриминант для задач двуклассовой классификации (R. A. Fisher. *The Use of Multiple Measurements in Taxonomic Problems. Annals of Eugenics* («Использование множественных измерений в таксономических задачах»). 7 (2):179-188, 1936). Позднее в 1948 г. Радхакришна Рао обобщил линейный дискриминант Фишера для многоклассовых задач, приняв допущение о равных ковариациях классов и нормально распределенных классах. Именно с тех пор этот метод получил название LDA (C. R. Rao. *The Utilization of Multiple Measurements in Problems of Biological Classification* («Использование множественных измерений в проблемах биологической классификации»). Journal of the Royal Statistical Society. Series B (Methodological), 10 (2):159-203, 1948).

Следующий ниже рисунок резюмирует идею LDA для двуклассовой задачи. Образцы из класса 1 показаны крестиками, соответственно, образцы из класса 2 – окружностями:



Как показано на оси X (LD 1), линейный дискриминант хорошо разделяет два нормально распределенных класса. Несмотря на то что приведенный в качестве примера линейный дискриминант на оси Y (LD 2) захватывает довольно много дисперсии в наборе данных, он не может быть хорошим линейным дискриминантом, поскольку не захватывает какой-либо информации о различении классов.

Одно из допущений в LDA состоит в том, что данные нормально распределены. Кроме того, мы также допускаем, что классы имеют идентичные ковариационные матрицы и что признаки статистически независимы друг от друга. Однако даже если одно или несколько допущений слегка нарушается, то метод LDA для снижения размерности может все еще работать относительно хорошо (R. O. Duda, P. E. Hart and D. G. Stork. *Pattern Classification* («Классификация образов»). 2nd Edition. New York, 2001).

Прежде чем в следующих подразделах мы рассмотрим внутреннее устройство LDA, сначала резюмируем ключевые шаги алгоритма LDA:

1. Стандартизировать d-мерный набор данных (d – это число признаков).
2. Для каждого класса вычислить d-мерный вектор средних.
3. Создать матрицу разброса между классами $\boldsymbol{S}_B$ и матрицу разброса внутри классов $\boldsymbol{S}_W$.
4. Вычислить собственные векторы и соответствующие собственные значения матрицы $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$.
5. Выбрать k собственных векторов, которые соответствуют k самым большим собственным значениям для построения $d \times k$-матрицы преобразования $\boldsymbol{W}$; собственные векторы являются столбцами этой матрицы.
6. Спроецировать образцы на новое подпространство признаков при помощи матрицы преобразования $\boldsymbol{W}$.

При использовании LDA мы делаем допущение, что признаки нормально распределены и независимы друг от друга. Кроме того, алгоритм LDA делает допущение, что ковариационные матрицы для индивидуальных классов идентичны. Однако даже если

мы до определенной степени эти допущения нарушаем, то LDA сможет относительно хорошо продолжить выполняться в задачах снижения размерности и классификации (R. O. Duda, P. E. Hart and D. G. Stork. *Pattern Classification* («Классификация образов»). 2nd Edition. New York, 2001).

Вычисление матриц разброса

Учитывая, что в разделе PCA в начале этой главы мы уже стандартизировали признаки набора данных сортов вин, мы можем пропустить первый шаг и сразу перейти к вычислению векторов средних значений, которые мы будем использовать для построения матриц разброса соответственно внутри классов и между классами. Каждый вектор средних $\boldsymbol{m}_i$ хранит среднее значение признака μ_m относительно образцов класса i:

$$\boldsymbol{m}_i = \frac{1}{n_i}\sum_{\boldsymbol{x}\in D_i}^{c}\boldsymbol{x}_m.$$

В результате получим три вектора средних (BC):

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,\text{алкоголь}} \\ \mu_{i,\text{яблочная кислота}} \\ \vdots \\ \mu_{i,\text{пролин}} \end{bmatrix}^T \quad i \in \{1, 2, 3\}.$$

```
>>>
np.set_printoptions(precision=4)
mean_vecs = []
for label in range(1,4):
    mean_vecs.append(np.mean(
    X_train_std[y_train==label], axis=0))
    print('BC %s: %s\n' %(label, mean_vecs[label-1]))

BC 1: [ 0.9259 -0.3091 0.2592 -0.7989 0.3039 0.9608 1.0515 -0.6306 0.5354
  0.2209 0.4855 0.798 1.2017]

BC 2: [-0.8727 -0.3854 -0.4437 0.2481 -0.2409 -0.1059 0.0187 -0.0164 0.1095
 -0.8796 0.4392 0.2776 -0.7016]

BC 3: [ 0.1637 0.8929 0.3249 0.5658 -0.01 -0.9499 -1.228 0.7436 -0.7652
  0.979 -1.1698 -1.3007 -0.3912]
```

Используя векторы средних, теперь можно вычислить матрицу разброса точек внутри класса $\boldsymbol{S}_W$:

$$\boldsymbol{S}_W = \sum_{i=1}^{c}\boldsymbol{S}_i.$$

Она вычисляется путем суммирования индивидуальных матриц разброса $\boldsymbol{s}_i$ каждого индивидуального класса i:

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x}\in D_i}^{c}(\boldsymbol{x}-\boldsymbol{m}_i)(\boldsymbol{x}-\boldsymbol{m}_i)^T.$$

```
>>>
d = 13 # число признаков
S_W = np.zeros((d, d))
for label,mv in zip(range(1,4), mean_vecs):
    class_scatter = np.zeros((d, d))
    for row in X_train[y_train == label]:
        row, mv = row.reshape(d, 1), mv.reshape(d, 1)
        class_scatter += (row-mv).dot((row-mv).T)
    S_W += class_scatter
print('Внутриклассовая матрица разброса: %sx%s' % (S_W.shape[0], S_W.shape[1]))
Внутриклассовая матрица разброса: 13x13
```

Во время вычисления матриц разброса мы делаем допущение, что метки классов в тренировочном наборе равномерно распределены. Однако если распечатать число меток классов, то увидим, что это допущение нарушено:

```
>>>
print('Распределение меток классов: %s' % np.bincount(y_train)[1:])
Распределение меток классов: [40 49 35]
```

Вследствие этого мы хотим прошкалировать индивидуальные матрицы разброса $\boldsymbol{S}_i$, прежде чем просуммируем их как матрицу разброса $\boldsymbol{S}_W$. Во время разбивки матрицы разброса на число образцов классов N_i можно увидеть, что вычисление матрицы разброса фактически не отличается от вычисления ковариационной матрицы Σ_i. Ковариационная матрица – это нормализованная версия матрицы разброса:

$$\Sigma_i = \frac{1}{N_i}\boldsymbol{S}_W = \frac{1}{N_i}\sum_{\boldsymbol{x}\in D_i}^{c}(\boldsymbol{x}-\boldsymbol{m}_i)(\boldsymbol{x}-\boldsymbol{m}_i)^T.$$

```
>>>
d = 13 # число признаков
S_W = np.zeros((d, d))
for label,mv in zip(range(1, 4), mean_vecs):
    class_scatter = np.cov(X_train_std[y_train==label].T)
    S_W += class_scatter
print('Масштабированная внутриклассовая матрица разброса: %sx%s'
      % (S_W.shape[0], S_W.shape[1]))
Масштабированная внутриклассовая матрица разброса: 13x13
```

Вычислив масштабированную матрицу разброса внутри классов (или ковариационную матрицу), можно перейти к следующему шагу и вычислить матрицу разброса между классами $\boldsymbol{S}_B$:

$$\boldsymbol{S}_B = \sum_{i=1}^{c} N_i(\boldsymbol{m}_i - \boldsymbol{m})(\boldsymbol{m}_i - \boldsymbol{m})^T.$$

Здесь $\boldsymbol{m}$ – это общее среднее, которое вычисляется, включая образцы из всех классов.

```
>>>
mean_overall = np.mean(X_train_std, axis=0)
d = 13 # число признаков
S_B = np.zeros((d, d))
for i,mean_vec in enumerate(mean_vecs):
```

```
    n = X_train[y_train==i+1, :].shape[0]
    mean_vec = mean_vec.reshape(d, 1)
    mean_overall = mean_overall.reshape(d, 1)
S_B += n * (mean_vec - mean_overall).dot((mean_vec - mean_overall).T)
print('Межклассовая матрица разброса: %sx%s'
       % (S_B.shape[0], S_B.shape[1]))
Межклассовая матрица разброса: 13x13
```

Отбор линейных дискриминантов для нового подпространства признаков

Оставшиеся шаги алгоритма LDA аналогичны шагам алгоритма PCA. Однако вместо выполнения разложения по собственным значениям на ковариационной матрице мы решаем обобщенную задачу на собственные значения матрицы $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$:

```
eigen_vals, eigen_vecs = \
      np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

После того как мы вычислили собственные пары, теперь можно отсортировать собственные значения в убывающем порядке:

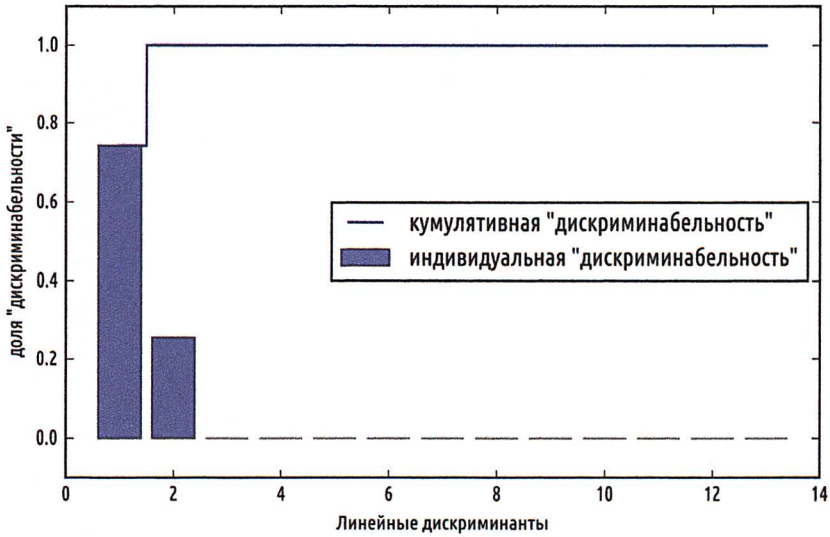
```
>>>
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
               for i in range(len(eigen_vals))]
eigen_pairs = sorted(eigen_pairs,
                     key=lambda k: k[0], reverse=True)
print('Собственные значения в убывающем порядке:\n')
for eigen_val in eigen_pairs:
    print(eigen_val[0])

Собственные значения в убывающем порядке:

452.721581245
156.43636122
8.11327596465e-14
2.78687384543e-14
2.78687384543e-14
2.27622032758e-14
2.27622032758e-14
1.97162599817e-14
1.32484714652e-14
1.32484714652e-14
1.03791501611e-14
5.94140664834e-15
2.12636975748e-16
```

В LDA число линейных дискриминантов не превышает $c - 1$, где c – это число меток классов, поскольку матрица разброса между классами $\boldsymbol{S}_B$ является суммой матриц c с рангом 1 или меньше. Мы действительно видим, что у нас всего два ненулевых собственных значения (собственные значения 3–13 не равны строго нулю, но это происходит в силу арифметики с плавающей точкой в NumPy). Отметим, что в редком случае полной коллинеарности (все точки образцов выровнены вдоль прямой) ковариационная матрица будет с рангом 1, что приведет к образованию всего одного собственного вектора с ненулевым собственным значением.

Чтобы измерить, сколько информации о различении классов захвачено линейными дискриминантами (собственными векторами), построим график линейных дискриминантов по убыванию собственных значений, подобно графику объясненной дисперсии, который мы создали в разделе PCA. Для простоты назовем содержание информации о различении классов *дискриминабельностью*.

```
tot = sum(eigen_vals.real)
discr = [(i / tot) for i in sorted(eigen_vals.real, reverse=True)]
cum_discr = np.cumsum(discr)
plt.bar(range(1, 14), discr, alpha=0.5, align='center',
        label='индивидуальная "дискриминабельность"')
plt.step(range(1, 14), cum_discr, where='mid',
         label='кумулятивная "дискриминабельность"')
plt.ylabel('доля "дискриминабельности"')
plt.xlabel('Линейные дискриминанты')
plt.ylim([-0.1, 1.1])
plt.legend(loc='best')
plt.show()
```

Как видно на итоговом рисунке, первые два линейных дискриминанта захватывают в тренировочном наборе данных сортов вин примерно 100% полезной информации:



Теперь объединим вертикально два наиболее отличительных столбца[1] с собственными векторами, чтобы создать матрицу преобразования $\boldsymbol{W}$:

```
>>>
w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
               eigen_pairs[1][1][:, np.newaxis].real))
print('Матрица W:\n', w)
```

[1] Приведенная в примере функция hstack() библиотеки NumPy объединет массивы вертикально по первым осям. – *Прим. перев.*

```
Матрица W:
[[ 0.0662 -0.3797]
 [-0.0386 -0.2206]
 [ 0.0217 -0.3816]
 [-0.184   0.3018]
 [ 0.0034  0.0141]
 [-0.2326  0.0234]
 [ 0.7747  0.1869]
 [ 0.0811  0.0696]
 [-0.0875  0.1796]
 [-0.185  -0.284 ]
 [ 0.066   0.2349]
 [ 0.3805  0.073 ]
 [ 0.3285 -0.5971]]
```

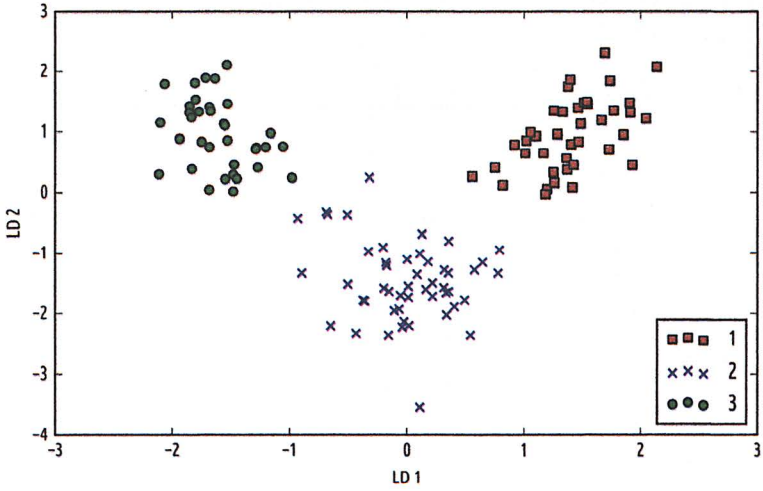
Проецирование образцов на новое пространство признаков

При помощи матрицы преобразования $\boldsymbol{W}$, которую мы создали в предыдущем подразделе, теперь можно преобразовать тренировочный набор данных путем умножения матриц:

$$\boldsymbol{X}' = \boldsymbol{X}\boldsymbol{W}.$$

```
X_train_lda = X_train_std.dot(w)
colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for l, c, m in zip(np.unique(y_train), colors, markers):
    plt.scatter(X_train_lda[y_train==l, 0]*(-1),
                X_train_lda[y_train==l, 1]*(-1),
                c=c, label=l, marker=m)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower right')
plt.show()
```

Как видно на получившемся графике, теперь в новом подпространстве признаков три класса вин линейно разделимы:



Метод LDA в scikit-learn

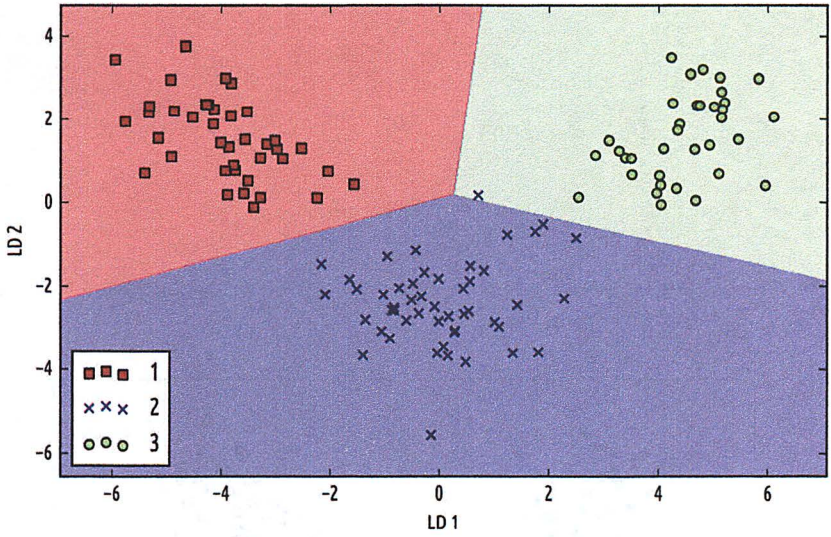
Пошаговая реализация была хорошим упражнением для понимания внутреннего устройства LDA и понимания различий между алгоритмами LDA и PCA. Теперь рассмотрим класс под названием LDA, реализованный в библиотеке scikit-learn:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis(n_components=2)
X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Затем посмотрим, как классификатор логистической регрессии обрабатывает более низкоразмерный тренировочный набор данных после преобразования LDA:

```
lr = LogisticRegression()
lr = lr.fit(X_train_lda, y_train)
plot_decision_regions(X_train_lda, y_train, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.show()
```

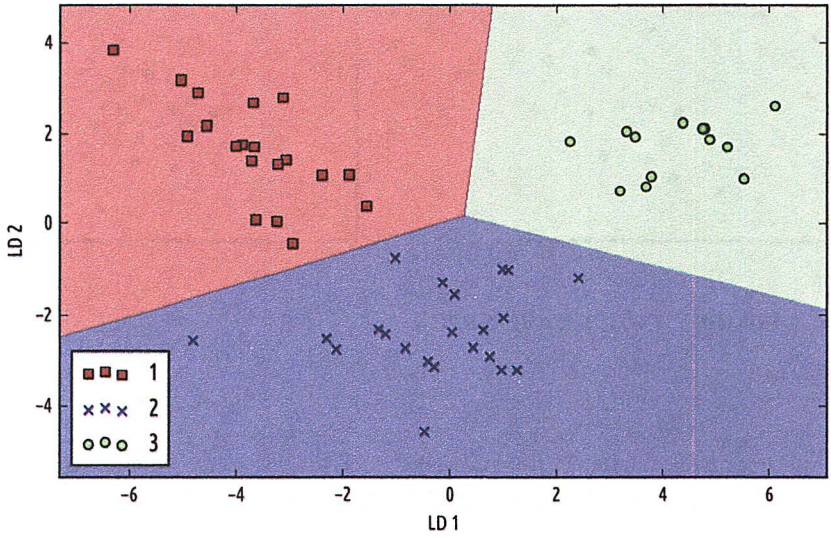
Рассматривая итоговый график, видно, что модель логистической регрессии неправильно классифицирует одну из образцов из класса 2:



Понизив силу регуляризации, по-видимому, можно было бы сместить границы решения, в результате чего модели логистической регрессии будут правильно классифицировать все образцы в тренировочном наборе данных. Впрочем, проанализируем результаты на тестовом наборе:

```
X_test_lda = lda.transform(X_test_std)
plot_decision_regions(X_test_lda, y_test, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.show()
```

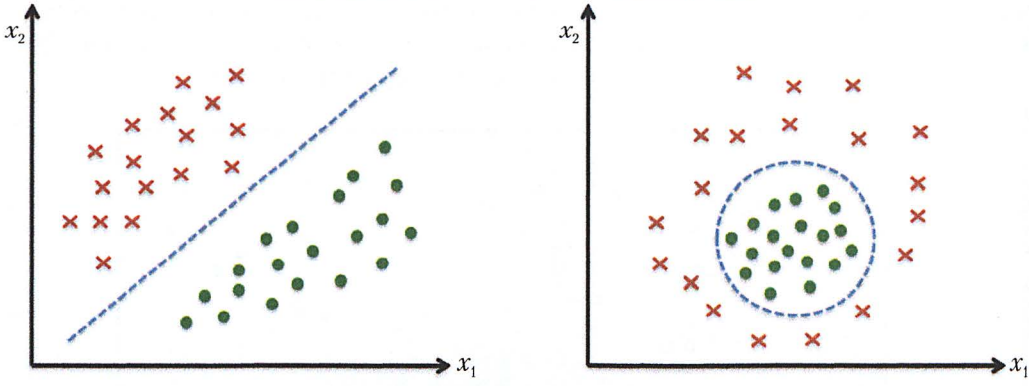
Как видно на получившемся графике, классификатор на основе логистической регрессии в состоянии получить идеальную оценку верности классификации образцов на тестовом наборе данных, используя для этого всего лишь двумерное подпространство признаков вместо исходного набора с 13 признаками вин:



Использование ядерного метода анализа главных компонент для нелинейных отображений

Во многих алгоритмах машинного обучения принимаются допущения относительно линейной разделимости входных данных. Вы узнали, что персептрон для сходимости даже требует полной линейной разделимости тренировочных данных. Другие алгоритмы, которые мы до сих пор рассматривали, допускают, что отсутствие полной линейной разделимости происходит из-за шума: ADALINE, логистическая регрессия и (стандартный) **метод опорных векторов** (SVM), и это не весь список. Однако если мы имеем дело с нелинейными задачами, с которыми можно довольно часто столкнуться в реальных приложениях, то методы линейного преобразования для снижения размерности, такие как PCA и LDA, могут представлять не самый лучший выбор. В этом разделе мы рассмотрим версию PCA с ядром, или *ядерный PCA*, который связан с ядерным методом опорных векторов (SVM), известным нам по *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*. Используя ядерный PCA, мы научимся преобразовывать линейно неразделимые данные на новое подпространство более низкой размерности, которое подходит для линейных классификаторов.

Линейные задачи в сопоставлении с нелинейными



Ядерные функции и ядерный трюк

Как мы помним из нашего обсуждения ядерных SVM в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, мы можем решать нелинейные задачи, проецируя данные на новое пространство признаков более высокой размерности, где классы становятся линейно разделимыми. Чтобы преобразовать образцы $x \in \mathbb{R}^d$ на это более высокое k-мерное подпространство, мы определили нелинейную функцию отображения ϕ:

$$\phi: \mathbb{R}^d \rightarrow \mathbb{R}^k (k >> d).$$

Функцию ϕ можно представить как функцию, которая создает нелинейные комбинации исходных признаков для отображения исходного d-мерного набора данных на более крупное k-мерное пространство признаков. Например, если у нас был вектор признаков $\boldsymbol{x} \in \mathbb{R}^d$ ($\boldsymbol{x}$ – это вектор-столбец, состоящий из d признаков) с двумя размерностями ($d = 2$), то потенциальное отображение на трехмерное пространство может быть следующим:

$$\boldsymbol{x} = [x_1, x_2]^T;$$
$$\downarrow \phi$$
$$\boldsymbol{z} = \left[x_1^2, \sqrt{2x_1x_2}, x_2^2\right]^T.$$

Другими словами, посредством ядерного PCA мы выполняем нелинейное отображение, которое преобразует данные в пространство более высокой размерности, и используем стандартный PCA в этом более высокоразмерном пространстве для проецирования данных назад в пространство более низкой размерности, где образцы могут быть разделены линейным классификатором (при условии, что образцы могут быть разделены во входном пространстве по плотности). Однако оборотной стороной этого подхода является то, что он в вычислительном плане очень затратен, и поэтому мы используем *ядерный трюк*. При помощи ядерного трюка можно вычислить подобие между двумя векторами признаков высокой размерности в исходном пространстве признаков.

Прежде чем мы продолжим более подробный анализ применения ядерного трюка для преодоления проблемы вычислительной затратности, вернемся к подходу на

основе *стандартного* PCA, который мы реализовали в начале этой главы. Мы вычислили ковариацию между двумя признаками k и j следующим образом:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}(x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k).$$

Учитывая, что стандартизация признаков центрирует их в нулевом среднем значении, например $\mu_j = 0$ и $\mu_k = 0$, это равенство можно упростить следующим образом:

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}x_j^{(i)}x_k^{(i)}.$$

Отметим, что предыдущее равенство касается ковариации между двумя признаками; теперь запишем общее равенство для расчета ковариационной *матрицы* Σ:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{x}^{(i)}\boldsymbol{x}^{(i)T}.$$

Этот принцип был обобщен Бернхардом Шолкопфом (B. Scholkopf, A. Smola and K.-R. Muller, «Kernel Principal Component Analysis» («*Ядерный метод главных компонент*»), с. 583–588, 1997), в результате чего посредством ϕ можно подставить нелинейные комбинации признаков вместо скалярных произведений между образцами в исходном пространстве признаков:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^T.$$

Для извлечения собственных векторов – главных компонент – из этой ковариационной матрицы нам нужно решить следующее уравнение:

$$\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$$

$$\Rightarrow \frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^T\boldsymbol{v} = \lambda\boldsymbol{v}$$

$$\Rightarrow \boldsymbol{v} = \frac{1}{n\lambda}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^T\boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{a}^{(i)}\phi(\boldsymbol{x}^{(i)}).$$

Здесь λ и $\boldsymbol{v}$ – это собственные значения и собственные векторы ковариационной матрицы Σ, и $\boldsymbol{a}$ можно получить путем извлечения собственных векторов из матрицы ядра (матрицы подобия) $\boldsymbol{K}$, как мы убедимся в последующих параграфах.

Матрица ядра выводится следующим образом.

Сначала запишем ковариационную матрицу в матричной записи, где $\phi(X)$ – это $n \times k$-матрица:

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n}\phi(\boldsymbol{x}^{(i)})\phi(\boldsymbol{x}^{(i)})^T = \frac{1}{n}\phi(\boldsymbol{X})^T\phi(\boldsymbol{X}).$$

Теперь определение собственного вектора можно записать следующим образом:

$$\boldsymbol{v} = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{a}^{(i)}\phi(\boldsymbol{x}^{(i)}) = \lambda\phi(\boldsymbol{X})^T\boldsymbol{a}.$$

Поскольку $\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$, получим:

$$\frac{1}{n}\phi(\boldsymbol{X})^T\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a} = \lambda\phi(\boldsymbol{X})^T\boldsymbol{a}.$$

Умножив его с обеих сторон на $\phi(X)$, в результате получим следующее:

$$\frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a} = \lambda\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\phi(\boldsymbol{X})\phi(\boldsymbol{X})^T\boldsymbol{a} = \lambda\boldsymbol{a}$$

$$\Rightarrow \frac{1}{n}\boldsymbol{K}\boldsymbol{a} = \lambda\boldsymbol{a}.$$

Здесь $\boldsymbol{K}$ – это матрица подобия (матрица ядра):

$$\boldsymbol{K} = \phi(\boldsymbol{X})\phi(\boldsymbol{X})^T.$$

Как мы помним из раздела, посвященного методам опорных векторов (SVM), *главы 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, ядерный трюк используется для того, чтобы избежать необходимости вычислять в явной форме попарные скалярные произведения образцов $\boldsymbol{x}$ под ϕ, используя для этого ядерную функцию $\boldsymbol{K}$, в результате чего нам не нужно вычислять собственные векторы непосредственно:

$$k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \phi(\boldsymbol{x}^{(i)})^T\phi(\boldsymbol{x}^{(j)}).$$

Другими словами, после ядерного PCA мы получаем образцы, которые уже спроецированы на соответствующие компоненты, вместо того чтобы строить матрицу преобразования, как при подходе на основе стандартного PCA. В сущности, ядерная функция (или просто *ядро*) может пониматься как функция, которая вычисляет скалярное произведение между двумя векторами – меру подобия.

Наиболее распространены следующие ядра:

- полиномиальное ядро:

  $$k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = (\boldsymbol{x}^{(i)^T}\boldsymbol{x}^{(j)} + \theta)^P.$$

  Здесь θ – это порог и P – степень, которая должны быть определена пользователем;
- гиперболическая тангенсная кернфункция (сигмоида):

  $$k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \tanh(\eta\boldsymbol{x}^{(i)^T}\boldsymbol{x}^{(j)} + \theta);$$

- **радиальная базисная функция** (РБФ, radial basis function, RBF), или гауссово ядро, которую мы будем использовать в последующих примерах в следующем подразделе:

$$k(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}) = \exp\left(-\frac{\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2}{2\sigma^2}\right).$$

Она также записывается следующим образом:

$$k(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right).$$

Резюмируя все, что мы обсудили до сих пор, можно определить следующие три шага для реализации ядерного PCA с РБФ в качестве ядра.

1. Вычислить матрицу ядра (матрицу подобия) k, где нам нужно вычислить следующее:

   $k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = \exp(-\gamma\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|^2)$.

   Выполнить это для каждой пары образцов:

   $$\boldsymbol{K} = \begin{bmatrix} k(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(1)}) & k(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(2)}) & \cdots & k(\boldsymbol{x}^{(1)},\boldsymbol{x}^{(n)}) \\ k(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(1)}) & k(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(2)}) & & k(\boldsymbol{x}^{(2)},\boldsymbol{x}^{(n)}) \\ \cdots & \cdots & \ddots & \cdots \\ k(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(1)}) & k(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(2)}) & \cdots & k(\boldsymbol{x}^{(n)},\boldsymbol{x}^{(n)}) \end{bmatrix}.$$

   Например, если набор данных содержит 100 тренировочных образцов, то симметричная матрица ядра попарных подобий была бы размером 100×100.

2. Центрировать матрицу ядра k, используя следующее равенство:

   $\boldsymbol{K}' = \boldsymbol{K} - 1_n\boldsymbol{K} - \boldsymbol{K}1_n + 1_n\boldsymbol{K}1_n$.

   Здесь 1_n – это $n\times n$-матрица (с такой же размерностью, что и у матрицы ядра), где все значения равны $\frac{1}{n}$.

3. Взять верхние k собственных векторов центрированной матрицы ядра, основываясь на соответствующих им собственных значениях, которые ранжированы по убыванию их величины (длины). В отличие от стандартного PCA, собственные векторы являются не осями главных компонент, а спроецированными на эти оси образцами.

В этом месте вам, наверное, интересно, зачем на втором шаге нужно центрировать матрицу ядра. Ранее, когда мы формулировали ковариационную матрицу, мы приняли допущение, что работаем со стандартизированными данными, в которых все признаки имеют нулевое среднее, и посредством ϕ подставили вместо скалярных произведений нелинейные комбинации признаков. Поэтому на втором шаге становится необходимым центрирование матрицы ядра, поскольку мы не вычисляем новое пространство признаков в явной форме и не можем гарантировать, что новое пространство признаков также центрировано в нуле.

В следующем разделе мы приведем все три шага в действие путем реализации ядерного PCA на Python.

Реализация ядерного метода анализа главных компонент на Python

В предыдущем подразделе мы обсудили ключевые идеи, лежащие в основе ядерного PCA. Теперь мы приступим к реализации ядерного PCA с РБФ в качестве ядра на Python в соответствии с тремя шагами, которые резюмируют подход на основе ядерного PCA. Используя вспомогательные функции библиотек SciPy и NumPy, мы увидим, что ядерный PCA на самом деле реализуется очень просто:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np

def rbf_kernel_pca(X, gamma, n_components):
    """
    Реализация ядерного PCA с РБФ в качестве ядра.

    Параметры
    ---------
    X: {NumPy ndarray}, форма = [n_samples, n_features]
    gamma: float
      Настроечный параметр ядра РБФ

    n_components: int
      Число возвращаемых главных компонент

    Возвращает
    ----------
    X_pc: {NumPy ndarray}, форма = [n_samples, k_features]
      Спроецированный набор данных
    """

    # Попарно вычислить квадратичные евклидовы расстояния
    # в наборе данных размера MxN.
    sq_dists = pdist(X, 'sqeuclidean')

    # Попарно конвертировать расстояния в квадратную матрицу.
    mat_sq_dists = squareform(sq_dists)

    # Вычислить симметричную матрицу ядра.
    K = exp(-gamma * mat_sq_dists)

    # Центрировать матрицу ядра.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Извлечь собственные пары из центрированной матрицы ядра
    # функция numpy.eigh возвращает их в отсортированном порядке.
    eigvals, eigvecs = eigh(K)

    # Взять верхних k собственных векторов (спроецированные образцы).

    X_pc = np.column_stack((eigvecs[:, -i]
                            for i in range(1, n_components + 1)))

    return X_pc
```

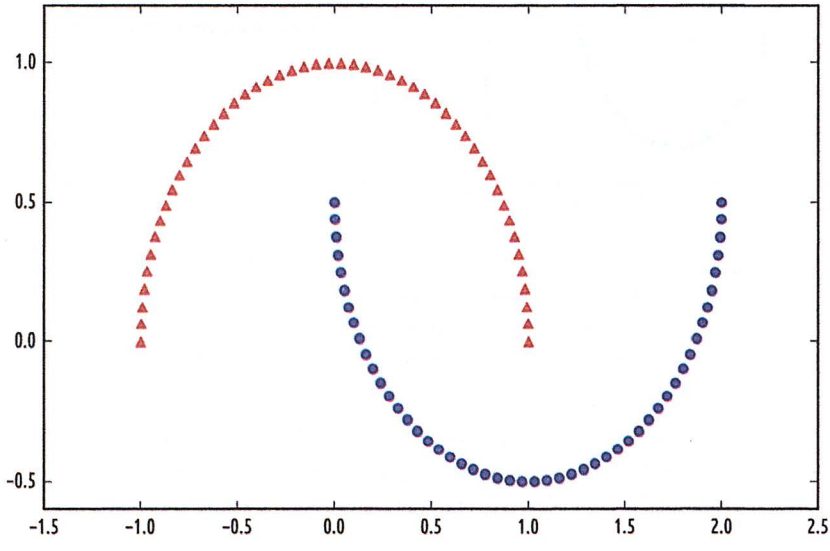
Оборотная сторона использования ядерного PCA с РБФ в качестве ядра для снижения размерности состоит в том, что нам нужно априорно определить параметр γ. Для нахождения надлежащего значения для γ требуется поэкспериментировать, и это лучше всего делать при помощи алгоритмов настройки параметров, например поиска по сетке параметров, который мы более подробно обсудим в *главе 6 «Изучение наиболее успешных методов оценки моделей и тонкой настройки гиперпараметров»*.

Пример 1. Разделение фигур в форме полумесяца

Теперь в качестве примера применим нашу функцию rbf_kernel_pca к неким нелинейным наборам данных. Начнем с создания двумерного набора данных из 100 точек образцов, представляющих собой две фигуры в форме полумесяца:

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, random_state=123)
plt.scatter(X[y==0, 0], X[y==0, 1],
            color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1],
            color='blue', marker='o', alpha=0.5)
plt.show()
```

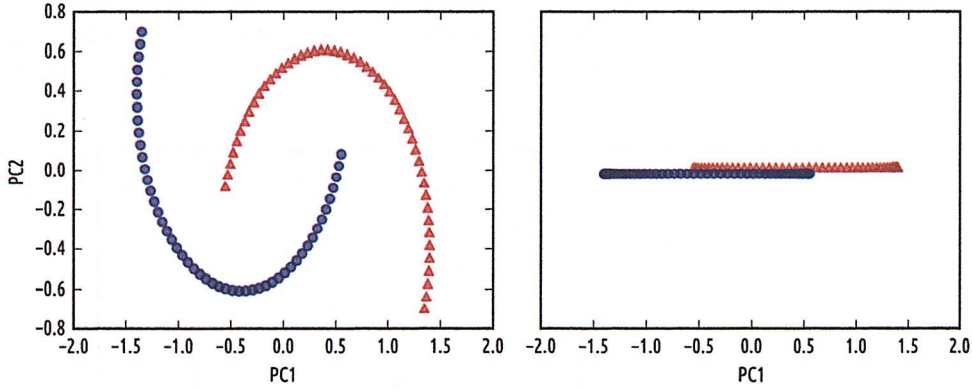
В целях иллюстрации полумесяц, состоящий из треугольных символов, будет представлять один класс, а полумесяц из круглых символов – образцы из другого класса:



Эти две фигуры полумесяца, безусловно, не являются линейно разделимыми, и наша задача состоит в том, чтобы *развернуть* полумесяцы ядерным методом PCA, в результате чего набор данных сможет подойти для передачи на вход линейного классификатора. Но сначала посмотрим, как выглядит набор данных, если его спроецировать на главные компоненты стандартным методом PCA:

```
from sklearn.decomposition import PCA
scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)
fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
              color='red', marker='^', alpha=0.5)
ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
              color='blue', marker='o', alpha=0.5)
ax[1].scatter(X_spca[y==0, 0], np.zeros((50,1))+0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_spca[y==1, 0], np.zeros((50,1))-0.02,
              color='blue', marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
```

На итоговом рисунке четко видно, что линейный классификатор не сможет хорошо сработать на наборе данных, преобразованном стандартным методом PCA:



Отметим, что когда мы построили график только первой главной компоненты (правый подграфик), мы сместили треугольные образцы немного вверх и круглые образцы немного вниз, для того чтобы лучше увидеть, что классы накладываются друг на друга.

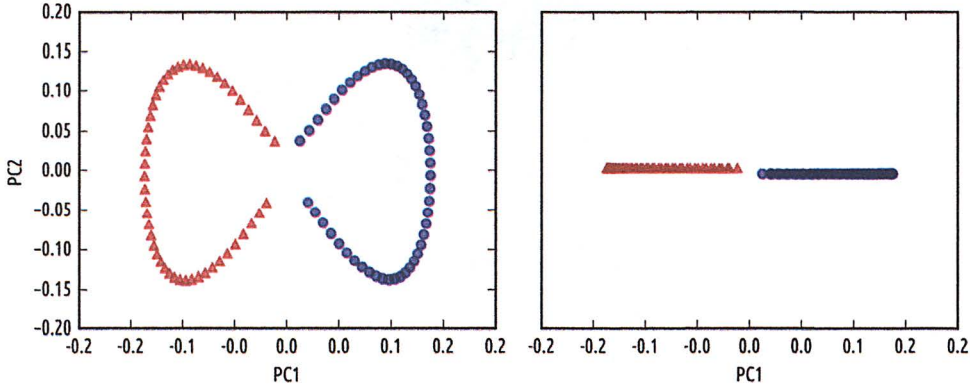
Напомним, что PCA – это метод машинного обучения без учителя, и, в отличие от LDA, в нем информация о метках классов для максимизации дисперсии не используется. В данном случае треугольные и круглые символы были добавлены для целей визуализации, просто чтобы обозначить степень разделения.

Теперь испытаем нашу функцию ядерного PCA rbf_kernel_pca, которую мы реализовали в предыдущем подразделе:

```
from matplotlib.ticker import FormatStrFormatter
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
              color='red', marker='^', alpha=0.5)
```

```
ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
              color='blue', marker='o', alpha=0.5)
ax[1].scatter(X_kpca[y==0, 0], np.zeros((50,1))+0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_kpca[y==1, 0], np.zeros((50,1))-0.02,
              color='blue', marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
ax[0].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
ax[1].xaxis.set_major_formatter(FormatStrFormatter('%0.1f'))
plt.show()
```

Теперь видно, что эти два класса (круги и треугольники) хорошо линейно разделены, и поэтому преобразованные данные стали тренировочным набором, который подходит для передачи на вход линейных классификаторов:



К сожалению, какое-то универсальное значение настроечного параметра γ, которое работает хорошо для разных наборов данных, отсутствует. Чтобы найти значение γ, подходящее для конкретной задачи, необходимо поэкспериментировать. В *главе 6 «Изучение наиболее успешных методов оценки моделей и тонкой настройки гиперпараметров»* мы обсудим методы, которые помогут автоматизировать задачу оптимизации таких настроечных параметров. Здесь я буду использовать те значения для γ, которые в моей практике давали *хорошие* результаты.

Пример 2. Разделение концентрических кругов

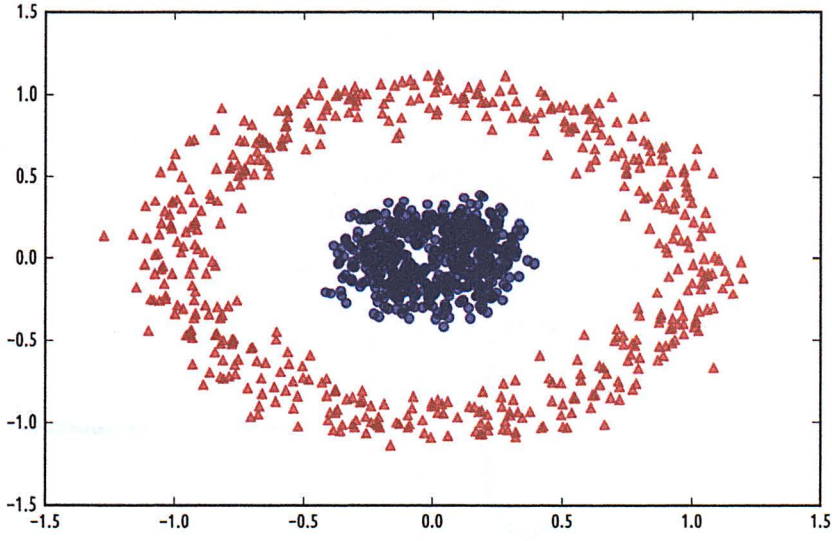
В предыдущем подразделе мы показали, каким образом ядерным методом PCA разделить фигуры полумесяца. Учитывая все те усилия, которые мы приложили, для того чтобы разобраться в понятиях ядерного метода PCA, обратимся к еще одному интересному примеру нелинейной задачи: концентрическим кругам.

Соответствующий исходный код выглядит следующим образом:

```
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=1000,
                    random_state=123, noise=0.1, factor=0.2)
```

```
plt.scatter(X[y==0, 0], X[y==0, 1],
            color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1],
            color='blue', marker='o', alpha=0.5)
plt.show()
```

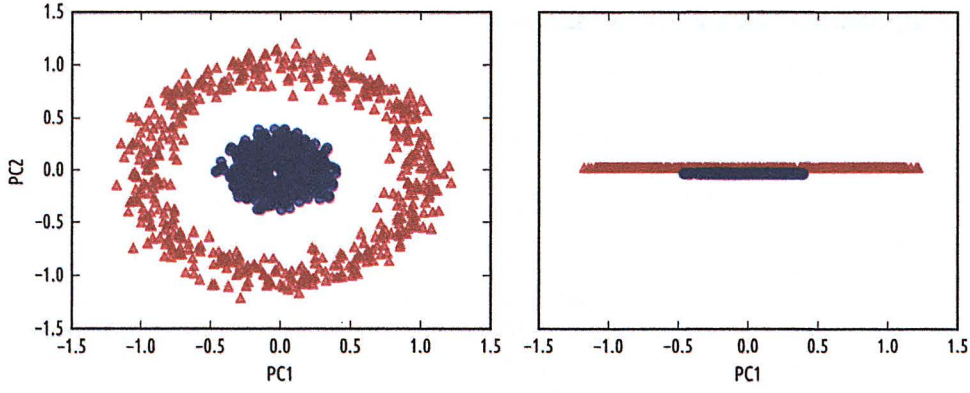
Мы снова берем задачу с двумя классами, где треугольные фигуры представляют один класс, а круглые фигуры – соответственно другой класс:



Начнем с подхода на основе стандартного PCA и сравним его с результатами ядерного PCA с РБФ в качестве ядра:

```
scikit_pca = PCA(n_components=2)
X_spca = scikit_pca.fit_transform(X)
fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_spca[y==0, 0], X_spca[y==0, 1],
              color='red', marker='^', alpha=0.5)
ax[0].scatter(X_spca[y==1, 0], X_spca[y==1, 1],
              color='blue', marker='o', alpha=0.5)
ax[1].scatter(X_spca[y==0, 0], np.zeros((500,1))+0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_spca[y==1, 0], np.zeros((500,1))-0.02,
              color='blue', marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
```

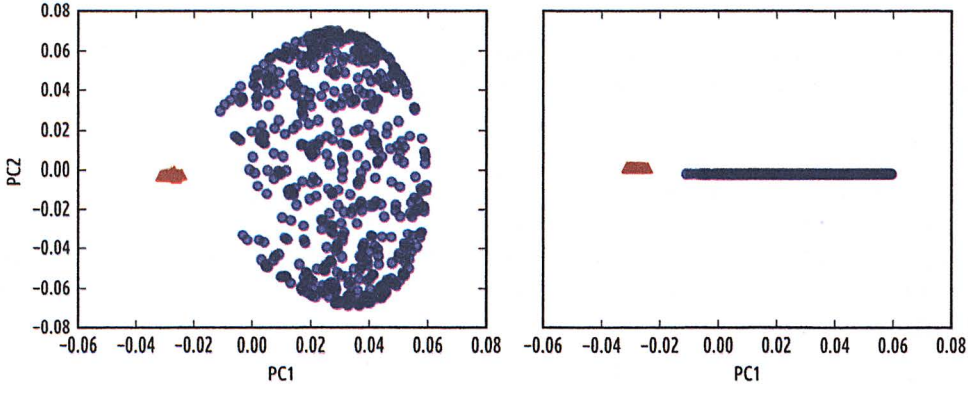
Мы снова видим, что стандартный PCA не в состоянии произвести результаты, подходящие для тренировки линейного классификатора:



При условии, что имеется подходящее значение для γ, посмотрим, будем ли мы удачливее с реализацией ядерного PCA на основе РБФ:

```
X_kpca = rbf_kernel_pca(X, gamma=15, n_components=2)
fig, ax = plt.subplots(nrows=1,ncols=2, figsize=(7,3))
ax[0].scatter(X_kpca[y==0, 0], X_kpca[y==0, 1],
              color='red', marker='^', alpha=0.5)
ax[0].scatter(X_kpca[y==1, 0], X_kpca[y==1, 1],
              color='blue', marker='o', alpha=0.5)
ax[1].scatter(X_kpca[y==0, 0], np.zeros((500,1))+0.02,
              color='red', marker='^', alpha=0.5)
ax[1].scatter(X_kpca[y==1, 0], np.zeros((500,1))-0.02,
              color='blue', marker='o', alpha=0.5)
ax[0].set_xlabel('PC1')
ax[0].set_ylabel('PC2')
ax[1].set_ylim([-1, 1])
ax[1].set_yticks([])
ax[1].set_xlabel('PC1')
plt.show()
```

И снова ядерный PCA на основе РБФ спроецировал данные на новое подпространство, где эти два класса становятся линейно разделимыми:



Проецирование новых точек данных

В двух предыдущих примерах приложений ядерного PCA, с полумесяцами и концентрическими кругами, мы проецировали одиночный набор данных на новый признак. Вместе с тем в реальных приложениях у нас может иметься более одного набора данных, которые мы хотим преобразовать, например тренировочные и тестовые данные, и, как правило, также новые образцы, которые мы собираем после построения и оценки модели. В этом разделе вы научитесь проецировать точки данных, которые не входили в состав тренировочного набора данных.

Как мы помним из варианта реализации на основе стандартного PCA, рассмотренного в начале этой главы, мы проецируем данные путем вычисления скалярного произведения между матрицей преобразования и входными образцами; столбцами проекционной матрицы являются верхние k собственных векторов ($\boldsymbol{v}$), которые мы извлекли из ковариационной матрицы. Теперь вопрос состоит в том, как перенести этот принцип на ядерный PCA. Если мы снова подумаем об идее, лежащей в основе ядерного PCA, то вспомним, что мы извлекли собственный вектор ($\boldsymbol{a}$) центрированной матрицы ядра (нековариационной матрицы), т. е. это образцы, которые уже спроецированы на ось главной компоненты $\boldsymbol{v}$. Вследствие этого, если мы хотим спроецировать новый образец $\boldsymbol{v}'$ на эту ось главной компоненты, нам нужно вычислить следующую проекцию:

$$\phi(\boldsymbol{x}')^T\boldsymbol{v}.$$

К счастью, мы можем применить ядерный трюк, в результате чего нам не придется вычислять проекцию $\phi(\boldsymbol{x}')^T\boldsymbol{v}$ явным образом. Однако стоит отметить, что ядерный PCA, в отличие от стандартного PCA, является методом, основанным на памяти, т. е. для проецирования новых образцов нам приходится применять исходный тренировочный набор каждый раз повторно и попарно вычислять ядро (подобие) РБФ между каждым i-м образцом в тренировочном наборе данных и новым образцом $\boldsymbol{x}'$:

$$\begin{aligned}\phi(\boldsymbol{x}')^T\boldsymbol{v} &= \sum_i a^{(i)}\phi(\boldsymbol{x}')^T\phi(\boldsymbol{x}^{(i)})\\ &= \sum_i a^{(i)}k(\boldsymbol{x}',\boldsymbol{x}^{(i)})^T.\end{aligned}$$

Здесь собственные векторы $\boldsymbol{a}$ и собственные значения λ матрицы ядра $\boldsymbol{K}$ удовлетворяют в уравнении следующему условию:

$$\boldsymbol{K}\boldsymbol{a} = \lambda\boldsymbol{a}.$$

После вычисления подобия между новыми образцами и образцами в тренировочном наборе следует нормализовать собственный вектор $\boldsymbol{a}$ его собственным значением. Вследствие этого внесем изменения в функцию rbf_kernel_pca, которую мы реализовали ранее, в результате чего она также будет возвращать собственные значения матрицы ядра:

```
from scipy.spatial.distance import pdist, squareform
from scipy import exp
from scipy.linalg import eigh
import numpy as np
```

```
def rbf_kernel_pca(X, gamma, n_components):
    """
    Реализация ядерного PCA с РБФ в качестве ядра.

    Параметры
    ---------
    X: {NumPy ndarray}, форма = [n_samples, n_features]

    gamma: float
      Настроечный параметр ядра РБФ

    n_components: int
      Число возвращаемых главных компонент

    Возвращает
    ----------
    X_pc: {NumPy ndarray}, форма = [n_samples, k_features]
      Спроецированный набор данных

    lambdas: список
      Собственные значения
    """

    # Попарно вычислить квадратичные евклидовы расстояния
    # в наборе данных размера MxN.
    sq_dists = pdist(X, 'sqeuclidean')

    # Попарно конвертировать расстояния в квадратную матрицу.
    mat_sq_dists = squareform(sq_dists)

    # Вычислить симметричную матрицу ядра.
    K = exp(-gamma * mat_sq_dists)

    # Центрировать матрицу ядра.
    N = K.shape[0]
    one_n = np.ones((N,N)) / N
    K = K - one_n.dot(K) - K.dot(one_n) + one_n.dot(K).dot(one_n)

    # Извлечь собственные пары из центрированной матрицы ядра
    # функция numpy.eigh возвращает их в отсортированном порядке.
    eigvals, eigvecs = eigh(K)

    # Взять верхних k собственных векторов (спроецированные образцы).
    alphas = np.column_stack((eigvecs[:, -i]
                              for i in range(1, n_components + 1)))

    # Собрать соответствующие собственные значения.
    lambdas = [eigvals[-i] for i in range(1,n_components+1)]

    return alphas, lambdas
```

Теперь создадим набор данных с полумесяцем и спроецируем его на одномерное подпространство, используя для этого обновленную реализацию ядерного PCA с РБФ в качестве ядра:

```
X, y = make_moons(n_samples=100, random_state=123)
alphas, lambdas =rbf_kernel_pca(X, gamma=15, n_components=1)
```

Чтобы удостовериться, что мы реализуем исходный код для проецирования новых образцов, примем, что 26-я точка из набора данных с полумесяцем является

новой точкой данных $\boldsymbol{x}'$, и наша задача состоит в том, чтобы спроецировать ее на это новое подпространство:

```
>>>
x_new = X[25]        # новая точка данных
x_new
array([ 1.8713187 , 0.00928245])

x_proj = alphas[25] # исходная проекция
x_proj
array([ 0.07877284])

def project_x(x_new, X, gamma, alphas, lambdas):
    pair_dist = np.array([np.sum(
                        (x_new-row)**2) for row in X])
    k = np.exp(-gamma * pair_dist)
    return k.dot(alphas / lambdas)
```

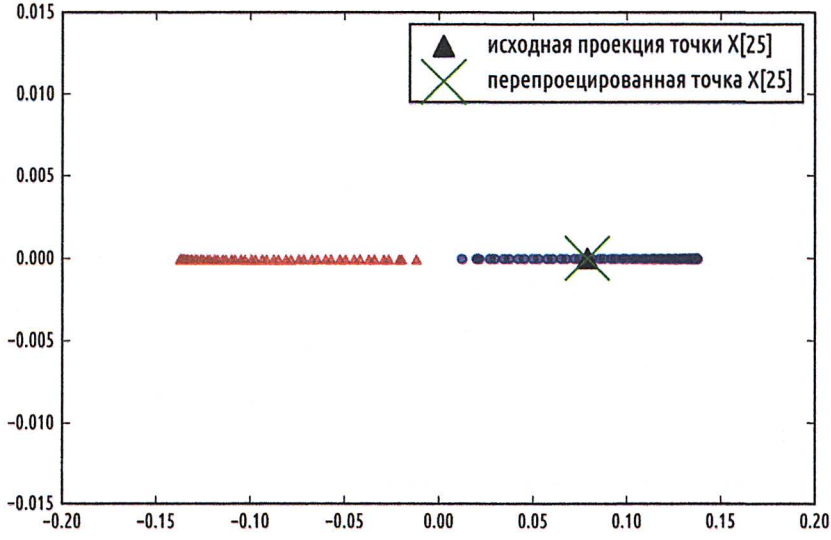
Выполнив следующий ниже фрагмент исходного кода, мы сможем воспроизвести исходную проекцию. При помощи функции project_x мы также сможем проецировать любые новые образцы данных. Соответствующий исходный код выглядит следующим образом:

```
>>>
x_reproj = project_x(x_new, X,   # проецирование "новой" точки данных
                     gamma=15, alphas=alphas, lambdas=lambdas)
x_reproj
array([ 0.07877284])
```

Наконец, выполним визуализацию проекции на первой главной компоненте:

```
plt.scatter(alphas[y==0, 0], np.zeros((50)),
            color='red', marker='^',alpha=0.5)
plt.scatter(alphas[y==1, 0], np.zeros((50)),
            color='blue', marker='o', alpha=0.5)
plt.scatter(x_proj, 0, color='black',
            label='исходная проекция точки X[25]',
            marker='^', s=100)
plt.scatter(x_reproj, 0, color='green',
            label='перепроецированная точка X[25]',
            marker='x', s=500)
plt.legend(scatterpoints=1)
plt.show()
```

Как видно на следующем ниже точечном графике, мы правильно отобразили образец $\boldsymbol{x}'$ на первую главную компоненту:



Ядерный метод анализа главных компонент в scikit-learn

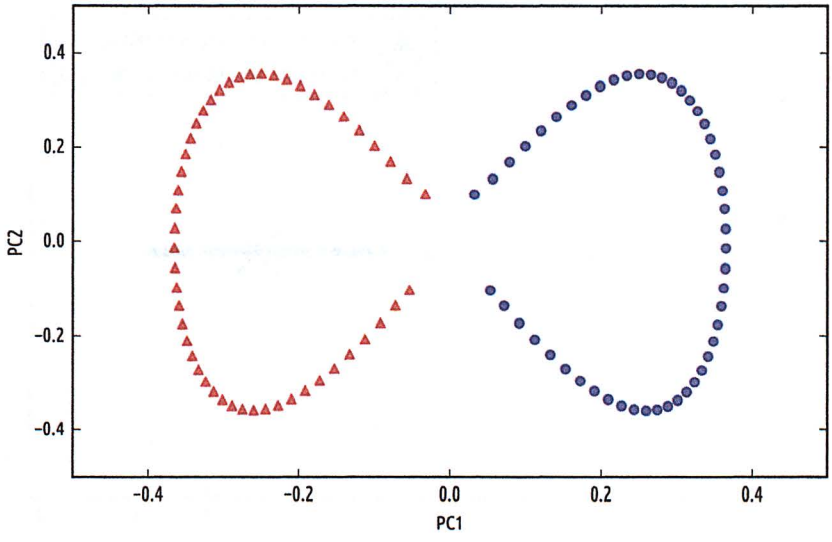
Для нашего удобства в библиотеке scikit-learn в подмодуле sklearn.decomposition реализован класс ядерного PCA. Его применение аналогично классу для стандартного PCA, при этом мы можем определить ядро посредством параметра kernel:

```
from sklearn.decomposition import KernelPCA
X, y = make_moons(n_samples=100, random_state=123)
scikit_kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_skernpca = scikit_kpca.fit_transform(X)
```

Чтобы убедиться, что мы получаем результаты, которые соответствуют нашей собственной реализации ядерного PCA, построим график преобразованных данных в форме полумесяцев на первые две главные компоненты:

```
plt.scatter(X_skernpca[y==0, 0], X_skernpca[y==0, 1],
            color='red', marker='^', alpha=0.5)
plt.scatter(X_skernpca[y==1, 0], X_skernpca[y==1, 1],
            color='blue', marker='o', alpha=0.5)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```

Как видно, результаты KernelPCA библиотеки scikit-learn соответствуют нашей собственной реализации:



В библиотеке scikit-learn также реализованы продвинутые методы нелинейного снижения размерности, рассмотрение которых выходит за рамки этой книги. Хороший обзор их текущих реализаций в библиотеке scikit-learn, дополненный показательными примерами, можно найти на http://scikit-learn.org/stable/modules/manifold.html.

Резюме

В этой главе вы узнали о трех разных основополагающих методах снижения размерности, применяемых для выделения признаков: стандартном методе PCA, LDA и ядерном методе PCA. Используя PCA, мы спроецировали данные на подпространство более низкой размерности для максимизации дисперсии вдоль ортогональных осей признаков, при этом игнорируя метки классов. Метод LDA, в отличие от PCA, применяется для снижения размерности с учителем, т. е. в нем при попытке максимизировать разделимость классов в линейном пространстве признаков учитывается информация о классах в тренировочном наборе данных. Наконец, вы узнали о нелинейном методе выделения признаков – ядерной версии PCA. На основе ядерного трюка и вре́менной проекции в пространство признаков более высокой размерности вы в итоге получаете возможность сжимать наборы данных, состоящие из нелинейных признаков, в подпространство меньшей размерности, где классы становятся линейно разделимыми.

Имея в распоряжении эти ключевые методы предобработки данных, вы теперь достаточно подготовлены, для того чтобы в следующей главе узнать о наиболее успешных практических приемах эффективного объединения различных методов предобработки данных и методах оценки качества различных моделей.

Изучение наиболее успешных методов оценки моделей и тонкой настройки гиперпараметров

В предыдущих главах вы узнали о ключевых алгоритмах машинного обучения для выполнения классификации и о методах приведения данных в приемлемый вид перед подачей их на вход этих алгоритмов. Теперь пора узнать об успешно зарекомендовавших себя на практике методах создания хороших машиннообучаемых моделей посредством тонкой настройки алгоритмов и о методах оценки качества моделей! В этой главе мы узнаем, как:

- извлекать объективные оценки качества моделей;
- диагностировать типичные проблемы алгоритмов машинного обучения;
- выполнять тонкую настройку машиннообучаемых моделей;
- оценивать прогнозные модели при помощи различных метрик оценки качества.

Оптимизация потоков операций при помощи конвейеров

Когда в предыдущих главах мы применяли различные методы предварительной обработки, такие как **стандартизация** для масштабирования признаков в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»* или **анализ главных компонент** для сжатия данных в *главе 5 «Сжатие данных путем снижения размерности»*, вы узнали, что мы должны повторно использовать параметры, полученные во время подгонки тренировочных данных, чтобы в дальнейшем шкалировать и сжимать любые новые данные, к примеру образцы в отдельном тестовом наборе данных. В этом разделе вы узнаете о чрезвычайно удобном инструменте, классе-конвейере Pipeline библиотеки scikit-learn. Он позволяет выполнять подгонку модели, в том числе с произвольным числом шагов преобразований, и применять его для выполнения прогнозов в отношении новых данных.

Загрузка набора данных Breast Cancer Wisconsin

В этой главе мы будем работать с набором данных о **раке молочной железы по шт. Висконсин, США** (Breast Cancer Wisconsin, BCW), который содержит 569образцов клеток **злокачественных** и **доброкачественных опухолей**. Первые два столбца в наборе данных содержат уникальные идентификационные номера об-

разцов и соответствующих диагнозов (*M = злокачественная, от англ. malignant, B = доброкачественная, от англ. benign*). Столбцы 3–32 содержат 30 вещественных признаков, вычисленных из оцифрованных изображений клеточных ядер, которые можно использовать для построения модели для идентификации, является ли опухоль доброкачественной или злокачественной. Набор данных BCW размещен в *репозитории машинного обучения UCI*, и более подробную информацию о нем можно найти на веб-странице https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

В этом разделе мы за три шага прочитаем набор данных и разделим его на тренировочный и тестовый наборы данных.

1. Начнем с того, что прочитаем набор данных непосредственно с веб-сайта UCI при помощи библиотеки pandas:

```
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machinelearning-databases/breast-cancer-
wisconsin/wdbc.data'
df = pd.read_csv(url, header=None)
```

2. Затем присвоим 30 признаков массиву X библиотеки NumPy. При помощи LabelEncoder преобразуем метки классов из их исходного строкового представления (M и B) в целочисленное:

```
from sklearn.preprocessing import LabelEncoder
X = df.loc[:, 2:].values
y = df.loc[:, 1].values
le = LabelEncoder()
y = le.fit_transform(y)
```

После кодирования меток классов (диагноза) в массиве y злокачественные опухоли теперь представлены как класс 1, доброкачественные опухоли – соответственно как класс 0, что можно проиллюстрировать, вызвав метод transform объекта LabelEncoder на двух фиктивных метках классов:

```
>>>
le.transform(['M', 'B'])
array([1, 0])
```

3. Прежде чем в следующем подразделе мы построим наш первый модельный конвейер, сначала разделим набор данных на отдельный тренировочный набор (80% данных) и отдельный тестовый набор (20% данных):

```
from sklearn.model_selection import train_test_split   # cross_validation
X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=0.20, random_state=1)
```

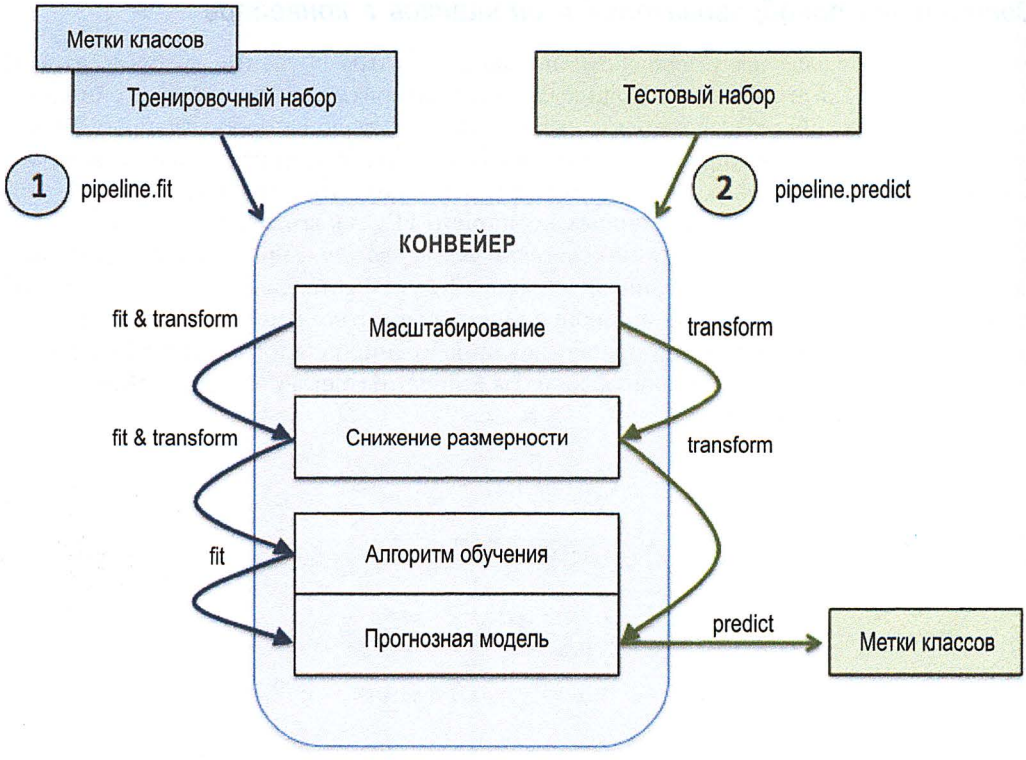
Совмещение преобразователей и оценщиков в конвейере

В предыдущей главе вы узнали, что многие алгоритмы обучения требуют, чтобы входные признаки в целях улучшения качества алгоритмов находились в одинаковой шкале. Поэтому прежде чем мы сможем подать столбцы набора данных BCW на вход линейного классификатора, такого как классификатор на основе логистической регрессии, нам нужно их стандартизировать. Кроме того, мы предположим, что нам нужно, используя **анализ главных компонент** (PCA), который мы представили в *главе 5 «Сжатие данных путем снижения размерности»*, применяемый для выделения признаков с целью снижения размерности, сжать данные с исходных 30 размерностей в более низкое двухмерное подпространство. Вместо раздельного выполнения шагов по подгонке и преобразованию тренировочного и тестового наборов данных мы можем расположить объекты StandardScaler, PCA и LogisticRegression друг за другом в конвейере:

```
>>>
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
pipe_lr = Pipeline([('scl', StandardScaler()),
                    ('pca', PCA(n_components=2)),
                    ('clf', LogisticRegression(random_state=1))])
pipe_lr.fit(X_train, y_train)
print('Верность на тестовом наборе: %.3f' % pipe_lr.score(X_test, y_test))
Верность на тестовом наборе: 0.947
```

Объект-конвейер Pipeline в качестве входа принимает список кортежей, где первое значение в каждом кортеже, как мы увидим далее в этой главе, – это произвольный строковый идентификатор, который можно использовать для доступа к индивидуальным элементам в конвейере, а второй элемент – это преобразователь либо оценщик библиотеки scikit-learn.

Промежуточные шаги в конвейере состоят из используемых в scikit-learn преобразователей, последний шаг – это оценщик. В приведенном выше примере мы собрали конвейер, который состоял из двух промежуточных шагов, преобразователей StandardScaler и PCA и классификатора на основе логистической регрессии в качестве заключительного оценщика. Когда в конвейере pipe_lr мы выполнили метод fit, объект StandardScaler применил к тренировочным данным методы fit и transform, и затем преобразованные тренировочные данные были переданы в следующий объект конвейера, PCA. Аналогично предыдущему шагу объект PCA также применил к масштабированным входным данным методы fit и transform и передал их заключительному элементу конвейера, оценщику. Следует отметить, что в конвейере число промежуточных шагов не ограничено. Общий принцип работы конвейера резюмирован в приведенном ниже рисунке:



Использование *k*-блочной перекрестной проверки для оценки качества модели

Один из ключевых шагов в создании машиннообучаемой модели состоит в оценке ее качества на данных, которые модель не видела прежде. Допустим, мы выполнили подгонку нашей модели на тренировочном наборе данных и используем те же самые данные для оценки ее качества на практике. Из раздела «*Решение проблемы переобучения посредством регуляризации*» в *главе 3* «*Обзор классификаторов с использованием библиотеки scikit-learn*» мы помним, что модель может страдать от недообучения (высокого смещения), в случае если она слишком простая, либо она может быть чересчур подогнанной под тренировочные данные (высокая дисперсия), в случае если она слишком сложная для базовых тренировочных данных. Для нахождения приемлемого компромисса между смещением и дисперсией нам нужно выполнить тщательную оценку нашей модели. В этом разделе вы узнаете о широко применяемых методах **перекрестной проверки с откладыванием данных** и ***k*-блочной перекрестной проверки**, которые способны помочь получить надежные оценки ошибки обобщения модели, т. е. того, насколько хорошо модели работали на ранее не встречавшихся данных.

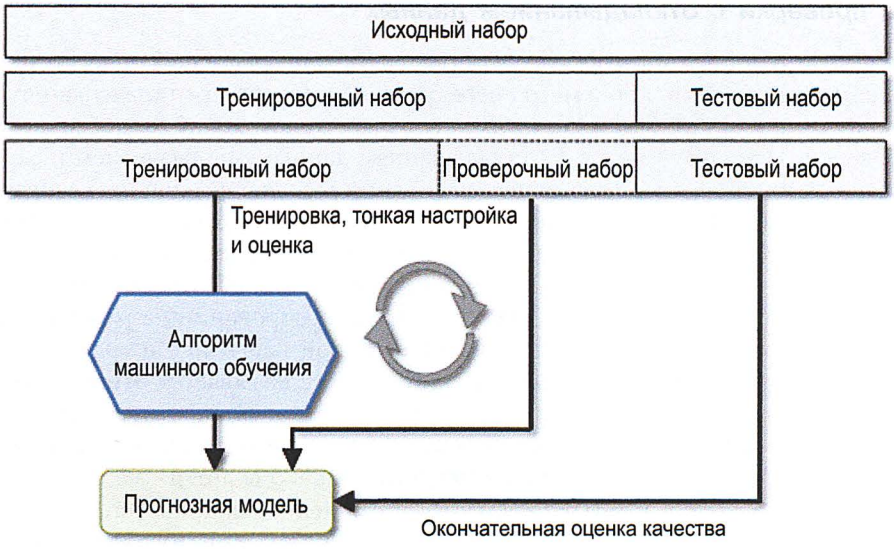
Метод проверки с откладыванием данных

Классическим и популярным подходом к оценке обобщающей способности машинообучаемых моделей является метод перекрестной проверки с откладыванием данных (holdout cross-validation), также именуемый методом двухблочной перекрестной проверки. Используя метод с откладыванием данных, мы разделяем исходный набор данных на отдельные тренировочный и тестовый наборы данных: первый используется для тренировки модели, второй – для оценки ее качества, а затем наборы меняются местами и второй используется для тренировки, а первый – для тестирования[1]. Однако в типичных приложениях машинного обучения, помимо этого, в целях дальнейшего улучшения качества при прогнозировании на ранее не встречавшихся данных мы также заинтересованы в тонкой настройке и сравнении различных параметрических конфигураций. Этот процесс называется **отбором модели** (model selection), где термин «отбор модели» обозначает конкретную задачу классификации, для которой мы хотим отобрать *оптимальные* значения настроечных параметров (так называемых **гиперпараметров**). Однако если тот же самый тестовый набор данных использовать во время отбора модели неоднократно, то он станет частью тренировочных данных, и поэтому модель, скорее всего, будет переобучена. Несмотря на эту проблему, многие практикующие специалисты для отбора модели до сих пор пользуются тестовым набором, что не является удачным приемом в практике машинного обучения.

Более эффективный способ применения метода с откладыванием данных для отбора модели состоит в том, чтобы разделить данные на три части: тренировочный, проверочный (валидационный, контрольный) и тестовый наборы. Тренировочный набор используется для выполнения подгонки разных моделей, а полученная на проверочном наборе оценка их качества затем используется для отбора модели. Преимущество тестового набора, который модель не видела ранее на этапах тренировки и отбора модели, состоит в том, что мы можем получить менее смещенную оценку ее способности обобщаться на новые данные. Приведенный ниже рисунок иллюстрирует принцип работы перекрестной проверки с откладыванием данных, где мы используем проверочный набор данных для неоднократной оценки качества модели после ее тренировки при помощи разных значений параметров. Как только мы удовлетворены тонкой настройкой значений параметров, мы оцениваем ошибку обобщения моделей на тестовом наборе данных:

---

[1] Иными словами, для каждого блока двум наборам одинакового размера, d0 и d1, случайно назначаются точки данных (обычно это делается путем перемешивания массива данных и его разбивки на две части). Затем мы тренируем на d0 и тестируем на d1, после чего тренируем на d1 и тестируем на d0. – *Прим. перев.*



Недостаток метода проверки с откладыванием данных заключается в том, что оценка качества модели чувствительна к тому, как мы делим тренировочный набор на тренировочное и проверочное подмножества; результат оценивания будет варьироваться для разных образцов данных. В следующем подразделе мы рассмотрим более устойчивый метод оценки качества моделей, метод k-блочной перекрестной проверки, где мы повторяем метод с откладыванием данных k раз на k подмножествах тренировочных данных.

k-блочная перекрестная проверка

При k-блочной перекрестной проверке мы случайным образом подразделяем тренировочный набор данных на k блоков без возврата (т. е. бесповторным способом), где $k - 1$ блоков используются для тренировки модели и один блок – для тестирования. Эта процедура повторяется k раз, в результате чего мы получаем k моделей и оценок их качества.

В случае если вы незнакомы с терминами «отбор с возвратом» и «отбор без возврата», проведем простой мысленный эксперимент. Допустим, мы играем в лотерею, где случайным образом вытягиваем числа из урны. Вначале урна содержит пять уникальных чисел 0, 1, 2, 3 и 4, и мы вытягиваем ровно одно число во время каждого розыгрыша. В первом раунде шанс вытянуть определенное число из урны был бы 1/5. Теперь при отборе без возврата мы не возвращаем число в урну после каждого розыгрыша. Следовательно, вероятность вытянуть определенное число из оставшегося набора чисел в следующем раунде зависит от предыдущего раунда. Например, если бы у нас в оставшемся наборе были числа 0, 1, 2 и 4, то шанс вытянуть число 0 стал бы 1/4 во время следующего розыгрыша.

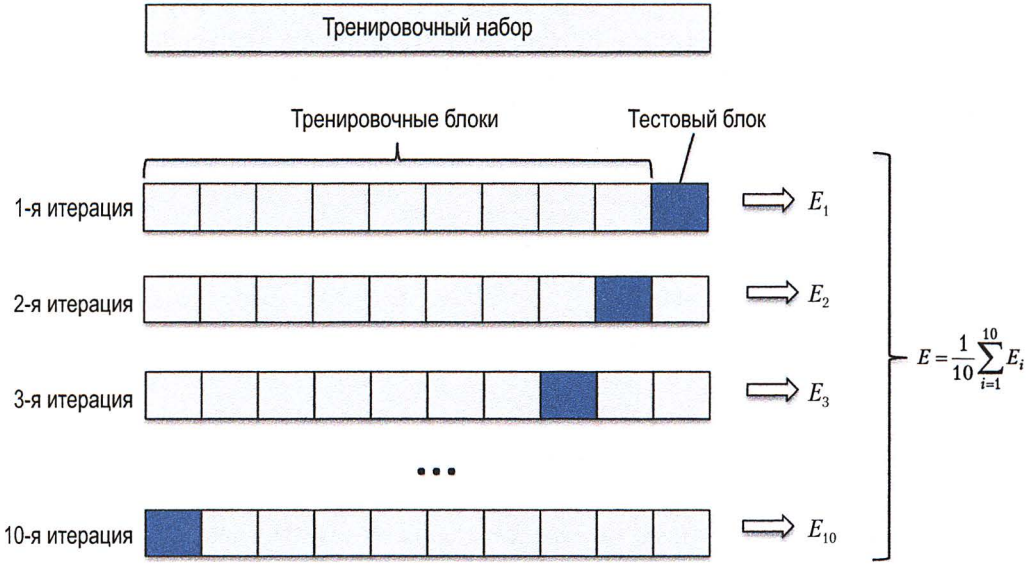
С другой стороны, при случайном отборе с возвратом мы всегда возвращаем взятое число в урну, в результате чего вероятности вытянуть определенное число при каждом розыгрыше не меняются; мы можем вытянуть то же самое число несколько раз. Другими словами, при отборе с возвратом образца (числа) независимы и имеют нулевую

ковариацию. Например, результаты пяти раундов вытягивания случайных чисел могли быть выглядеть следующим образом:

- случайный отбор без возврата: 2, 1, 3, 4, 0;
- случайный отбор с возвратом: 1, 3, 3, 4, 1.

Затем мы вычисляем среднее качество работы моделей, основываясь на разных, независимых блоках с целью получения оценки, которая менее чувствительна к разделению тренировочных данных на блоки, по сравнению с методом с откладыванием данных. Как правило, k-блочная перекрестная проверка используется для тонкой настройки модели, т. е. для нахождения оптимальных значений гиперпараметров, которые дают удовлетворительную обобщающую способность. Как только мы нашли удовлетворительные значения гиперпараметров, мы можем повторно натренировать модель на полном тренировочном наборе и получить окончательную оценку ее качества, используя для этого независимый тестовый набор.

Учитывая, что k-блочная перекрестная проверка – это метод генерирования повторных образцов без возврата (непересекающихся образцов), преимущество этого подхода заключается в том, что каждая точка образца будет частью тренировочного и тестового наборов данных ровно один раз, что в итоге дает более низкодисперсную оценку качества модели, чем метод с откладыванием данных. Следующий ниже рисунок резюмирует идею, лежащую в основе k-блочной перекрестной проверки при $k = 10$. Тренировочный набор данных разделен на 10 блоков, и во время этих 10 итераций 9 блоков используются для тренировки, и 1 блок будет использован в качестве тестового набора для оценки модели. Кроме того, оценочные характеристики E_1 (например, верность или ошибка классификации) для каждого блока используются для вычисления предполагаемой средней E модели:



Стандартное значение для k в k-блочной перекрестной проверке равно 10 блокам, что, как правило, является разумным вариантом для большинства приложений. Однако если мы работаем с относительно небольшими тренировочными наборами, то может оказаться целесообразным их число увеличить. Если мы увеличим значение k, то на каждой итерации будет использоваться больше тренировочных данных, в результате чего будет получено более низкое смещение в сторону оценки качества обобщения путем усреднения индивидуальных модельных оценок. Однако крупные значения k одновременно увеличивают время выполнения алгоритма перекрестной проверки, давая при этом оценки с более высокой дисперсией, поскольку тренировочные блоки будут более похожи друг на друга. С другой стороны, если мы работаем с большими наборами данных, то мы можем выбрать меньшее значение для k, например $k = 5$, и при этом продолжать получать точную оценку среднего качества модели, одновременно сокращая вычислительные затраты на повторную подгонку и оценивание модели на разных блоках.

Особым случаем k-блочной перекрестной проверки является метод перекрестной проверки **с исключением по одному** (leave-one-out, LOO). В этом методе мы устанавливаем кратность равной числу тренировочных образцов ($k = n$), так чтобы только один тренировочный образец использовался для тестирования во время каждой итерации. Этот подход рекомендуется к применению для работы с очень малыми наборами данных[1].

Стратифицированная k-блочная перекрестная проверка[2] дает легкое улучшение, по сравнению со стандартным подходом на основе k-блочной перекрестной проверки, которая, в отличие от последнего, может давать оценки с более хорошими смещением и дисперсией, в особенности в случаях неравных пропорций классов, как было показано в исследовании Р. Кохави и др. (R. Kohavi et al. *A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection* («Исследование методов перекрестной проверки и бутстрапирования для оценки верности и отбора моделей»). In Ijcai, том 14, с. 1137–1145, 1995). При стратифицированной перекрестной проверке пропорции классов сохраняются в каждом блоке, гарантируя, что каждый блок отражает пропорции классов в тренировочном наборе данных, что мы проиллюстрируем при помощи итератора StratifiedKFold библиотеки scikit-learn:

```
>>>
import numpy as np
from sklearn.model_selection import StratifiedKFold
kfold = StratifiedKFold(y=y_train,
                        n_folds=10,
                        random_state=1)
scores = []
```

[1] Перекрестная проверка с исключением по одному – это простая перекрестная проверка, где каждый обучающий набор создается с использованием всех образцов, за исключением одной, и этот отложенный образец становится тестовым набором. Так, для n образцов будет иметься n разных тренировочных и n разных тестовых наборов. Эта процедура перекрестной проверки не тратит слишком много данных, т. к. из тренировочного набора удаляется всего один образец. – *Прим. перев.*

[2] Стратифицированная k-блочная перекрестная проверка (т. е. с разбивкой по классам) – это вариант k-блочной перекрестной проверки, которая возвращает стратифицированные блоки: каждый набор содержит приблизительно тот же процент образцов каждого целевого класса, что и полный набор. – *Прим. перев.*

```
for k, (train, test) in enumerate(kfold):
    pipe_lr.fit(X_train[train], y_train[train])
    score = pipe_lr.score(X_train[test], y_train[test])
    scores.append(score)
    print('Блок: %s, распр. классов: %s, верность: %.3f' % (k+1,
                np.bincount(y_train[train]), score))
Блок: 1, распр. классов: [256 153], верность: 0.891
Блок: 2, распр. классов: [256 153], верность: 0.978
Блок: 3, распр. классов: [256 153], верность: 0.978
Блок: 4, распр. классов: [256 153], верность: 0.913
Блок: 5, распр. классов: [256 153], верность: 0.935
Блок: 6, распр. классов: [257 153], верность: 0.978
Блок: 7, распр. классов: [257 153], верность: 0.933
Блок: 8, распр. классов: [257 153], верность: 0.956
Блок: 9, распр. классов: [257 153], верность: 0.978
Блок: 10, распр. классов: [257 153], верность: 0.956
print('Перекрестно-проверочная верность: %.3f +/- %.3f' % (  # CV accuracy
                                np.mean(scores), np.std(scores)))
Перекрестно-проверочная верность: 0.950 +/- 0.029
```

Сначала мы инициализировали итератор StratifiedKfold из модуля sklearn.model_selection (в версии библиотеки < 0.18 этот модуль назывался cross_validation) метками классов y_train в тренировочном наборе и задали число блоков при помощи параметра n_folds. Когда мы использовали итератор kfold для перебора всех k блоков в цикле, мы использовали возвращаемые в train индексы для подгонки заданного нами в начале этой главы конвейера с логистической регрессией. Использование конвейера pile_lr обеспечивает соответствующее масштабирование образцов в каждой итерации (например, их стандартизацию). Затем мы использовали индексы test для вычисления оценки верности модели, которые мы аккумулировали в списке оценок scores с целью вычисления средней верности и стандартного отклонения оценки.

Предыдущий пример был полезен в качестве иллюстрации того, как работает *k*-блочная перекрестная проверка. Вместе с тем в библиотеке scikit-learn также реализован регистратор оценок (scorer) *k*-блочной перекрестной проверки, который позволяет эффективнее оценивать модель при помощи стратифицированной *k*-блочной перекрестной проверки:

```
>>>
from sklearn.model_selection import cross_val_score
scores = cross_val_score(estimator=pipe_lr,
                         X=X_train,
                         y=y_train,
                         cv=10,
                         n_jobs=1)
print('Оценки перекрестно-проверочной верности: %s' % scores)
Оценки перекрестно-проверочной верности: [ 0.89130435  0.97826087  0.97826087
                                            0.91304348  0.93478261  0.97777778
                                            0.93333333  0.95555556  0.97777778
                                            0.95555556]
print('Перекрестно-проверочная верность: %.3f +/- %.3f' %
                                            (np.mean(scores), np.std(scores)))
Перекрестно-проверочная верность: 0.950 +/- 0.029
```

Чрезвычайно полезное свойство подхода на основе функции cross_val_score состоит в том, что мы можем распределять оценивание различных блоков по двум

и более центральным процессорам (ЦП) компьютера. Если установить параметр `n_jobs` в `1`, то для оценки качества будет использоваться всего один ЦП, так же как в предыдущем примере `StratifiedKFold`. Однако, задав `n_jobs=2`, мы могли бы распределить 10 раундов перекрестной проверки на два ЦП (при их наличии на компьютере), и, задав `n_jobs =-1`, можно использовать все имеющиеся на нашей машине ЦП, чтобы выполнять вычисление параллельно.

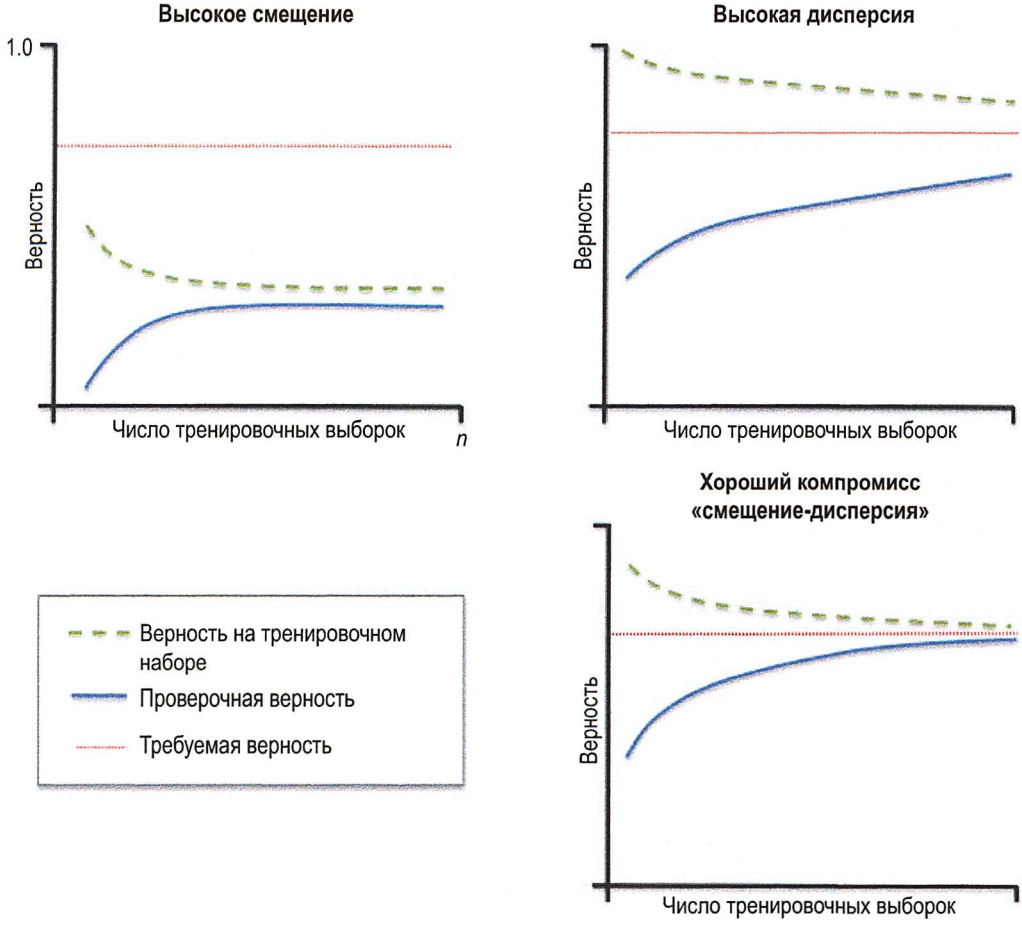
Отметим, что детальное обсуждение того, как в перекрестной проверке оценивается дисперсия обобщающей способности модели, выходит за рамки этой книги, однако детальное обсуждение этой темы можно найти в превосходной статье М. Маркату и др. (M. Markatou, H. Tian, S. Biswas, G. M. Hripcsak. *Analysis of Variance of Cross-validation Estimators of the Generalization Error* («Анализ дисперсии в кросс-валидационных эстиматорах ошибки обобщения»). Journal of Machine Learning Research, 6:1127-1168, 2005). Можно также почитать материалы по альтернативным методам перекрестной проверки, таким как метод кросс-валидации с 632+-бутстрапированием (B. Efron, R. Tibshirani. *Improvements on Cross-validation: The 632+ Bootstrap Method* («Совершенствование перекрестной проверки: Метод 632+-бутстрапирования»). Journal of the American Statistical Association, 92(438):548-560, 1997).

Отладка алгоритмов при помощи кривой обучения и проверочной кривой

В этом разделе мы рассмотрим два очень простых и одновременно мощных инструмента диагностики, которые способны помочь улучшить качество работы алгоритма обучения: **кривые обучения** (learning curves) и **проверочные кривые** (validation curves). В следующих подразделах мы обсудим способы применения кривых обучения для диагностирования у алгоритма обучения проблемы с переобучением (высокой дисперсией) или недообучением (высоким смещением). Кроме того, мы коснемся проверочных кривых, которые способны помочь решать общие проблемы, касающиеся алгоритмов обучения.

Диагностирование проблем со смещением и дисперсией при помощи кривых обучения

Если модель для данного тренировочного набора данных слишком сложна, например в этой модели имеется слишком много степеней свободы либо параметров, то она демонстрирует тенденцию к переобучению под тренировочные данные и недостаточно хорошо обобщается на ранее не встречавшиеся данные. Нередко сократить степень переобучения помогает аккумулирование большего количества тренировочных образцов. Однако на практике эта работа часто может оказаться очень затратной или просто невыполнимой. Построив график модельных верностей на тренировочном и проверочном (валидационном) наборах как функций размера тренировочного набора, можно легко установить, страдает ли модель от высокой дисперсии или высокого смещения и поможет ли аккумулирование большего количества данных решить эту проблему. Но прежде чем мы приступим к обсуждению того, как построить график кривых обучения в библиотеке sckit-learn, обсудим эти две распространенные модельные проблемы, вкратце рассмотрев следующую ниже иллюстрацию:



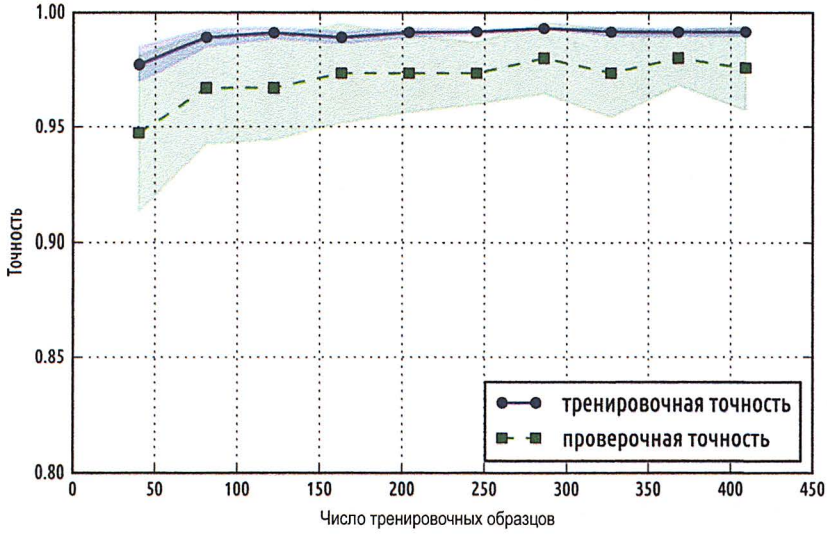
Верхний левый график показывает модель с высоким смещением. Эта модель имеет как низкую верность на тренировочном наборе, так и низкую верность на перекрестно-проверочных данных, что указывает на то, что она недообучена под тренировочные данные. Чаще всего эта проблема решается за счет увеличения числа параметров модели, например путем аккумулирования или создания дополнительных признаков либо уменьшения степени регуляризации, например в классификаторах с логистической регрессией или методами опорных векторов (SVM). Верхний правый график показывает модель, которая страдает от высокой дисперсии, что обозначено большим разрывом между верностью на тренировочном наборе и точностью на перекрестно-проверочных данных. Для решения этой проблемы переподгонки можно аккумулировать больше тренировочных данных либо уменьшить сложность модели, например путем увеличения параметра регуляризации; в случае нерегуляризованных моделей также помогает сокращение числа признаков методами отбора признаков (*глава 4 «Создание хороших тренировочных наборов – предобработка данных»*) либо выделения признаков (*глава 5 «Сжатие данных путем снижения размерности»*). Отметим, что аккумулирование большего количества тренировочных данных уменьшает шанс переобучения, однако оно помогает не всегда,

например когда тренировочные данные чрезвычайно зашумлены либо модель уже очень близка к оптимальной.

В следующем подразделе мы увидим, как решать эти модельные проблемы при помощи проверочных кривых, однако сначала посмотрим, как для оценивания модели можно использовать функцию кривой обучения в библиотеке scikit-learn:

```
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve
pipe_lr = Pipeline([
           ('scl', StandardScaler()),
           ('clf', LogisticRegression(
                        penalty='l2', random_state=0))])
train_sizes, train_scores, test_scores = \
       learning_curve(estimator=pipe_lr,
       X=X_train,
       y=y_train,
       train_sizes=np.linspace(0.1, 1.0, 10),
       cv=10,
       n_jobs=1)
train_mean = np.mean(train_scores, axis=1)
train_std  = np.std(train_scores, axis=1)
test_mean  = np.mean(test_scores, axis=1)
test_std   = np.std(test_scores, axis=1)
plt.plot(train_sizes, train_mean,
         color='blue', marker='o',
         markersize=5,
         label='тренировочная верность')
plt.fill_between(train_sizes,
                 train_mean + train_std,
                 train_mean - train_std,
                 alpha=0.15, color='blue')
plt.plot(train_sizes, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='проверочная верность')
plt.fill_between(train_sizes,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')
plt.grid()
plt.xlabel('Число тренировочных образцов')
plt.ylabel('Верность')
plt.legend(loc='lower right')
plt.ylim([0.8, 1.0])
plt.show()
```

После того как мы успешно выполнили приведенный выше исходный код, мы получим следующий ниже график кривой обучения:



Благодаря параметру train_sizes в функции кривой обучения learning_curve можно управлять абсолютным либо относительным числом тренировочных образцов, которые используются для генерирования кривых обучения. Здесь мы задаем train_sizes=np.linspace (0.1, 1.0, 10), чтобы использовать 10 равноотстоящих относительных интервалов для размеров тренировочных наборов. Для вычисления верности на перекрестно-проверочных данных функция learning_curve по умолчанию использует стратифицированную k-блочную перекрестную проверку, и в параметре cv мы устанавливаем $k = 10$. Затем мы просто вычисляем средние верности из полученных оценок на перекрестно-проверенных тренировочных данных и тестовых данных для разных размеров тренировочного набора, чей график мы построили при помощи функции plot библиотеки matplotlib. Кроме того, чтобы показать дисперсию оценки, мы добавили в график стандартное отклонение средних верностей, воспользовавшись для этого функцией fill_between.

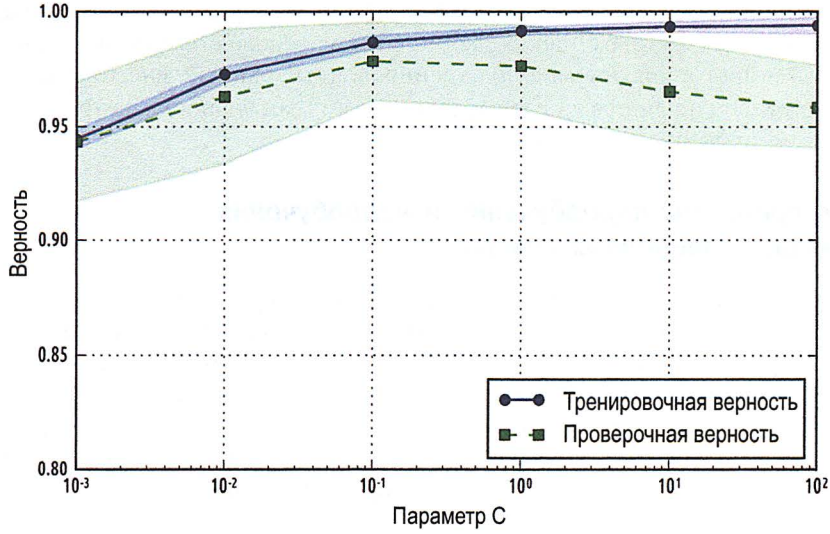
Как видно на приведенном выше графике функции кривой обучения, наша модель показывает на тестовом наборе данных хорошее качество работы. Однако она может быть немного переобучена под тренировочные данные, на что указывает относительно малый, но видимый разрыв между кривыми верности на тренировочных и перекрестно-проверочных данных.

Решение проблемы переобучения и недообучения при помощи проверочных кривых

Проверочные кривые – это полезный инструмент для улучшения качества модели благодаря решению таких проблем, как переобучение или недообучение. Проверочные кривые связаны с кривыми обучения, но, вместо того чтобы строить график верностей на тренировочном и тестовом наборах как функции от объема образца, мы варьируем значения модельных параметров, например параметр обратной регуляризации C в логистической регрессии. Пойдем дальше и посмотрим, как при помощи библиотеки sckit-learn создаются проверочные кривые:

```
from sklearn.model_selection import validation_curve   # cross_validation
param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
train_scores, test_scores = validation_curve(
         estimator=pipe_lr,
         X=X_train,
         y=y_train,
         param_name='clf__C',
         param_range=param_range,
         cv=10)
train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)
plt.plot(param_range, train_mean,
         color='blue', marker='o',
         markersize=5,
         label='тренировочная верность')
plt.fill_between(param_range, train_mean + train_std,
                 train_mean - train_std, alpha=0.15,
                 color='blue')
plt.plot(param_range, test_mean,
         color='green', linestyle='--',
         marker='s', markersize=5,
         label='проверочная верность')
plt.fill_between(param_range,
                 test_mean + test_std,
                 test_mean - test_std,
                 alpha=0.15, color='green')
plt.grid()
plt.xscale('log')
plt.legend(loc='lower right')
plt.xlabel('Параметр C')
plt.ylabel('Верность')
plt.ylim([0.8, 1.0])
plt.show()
```

При помощи приведенного выше исходного кода мы получили график проверочной кривой для параметра C:



Аналогично функции learning_curve, функция validation_curve для оценки работоспособности модели по умолчанию использует стратифицированную k-блочную перекрестную проверку, в случае если мы используем алгоритмы классификации данных. Внутри функции validation_curve мы указали параметр, который мы хотели оценить. В данном случае это параметр обратной регуляризации C классификатора LogisticRegression. Мы его обозначили как 'clf__C', чтобы иметь доступ к объекту LogisticRegression внутри конвейера библиотеки scikit-learn для заданного диапазона значений, который мы задали посредством параметра param_range. Аналогично примеру кривой обучения в предыдущем разделе мы построили график средних верностей на тренировочных данных и перекрестно-проверочных данных и соответствующих им стандартных отклонений.

Несмотря на то что различия в верности для варьирующихся значений параметра C незначительны, мы видим, что при увеличении силы регуляризации (малые значения C) модель слегка недоадаптируется под данные. Однако крупные значения C приводят к снижению силы регуляризации, и поэтому модель начинает демонстрировать тенденцию к легкому переобучению под данные. В этом случае наиболее перспективная зона, похоже, находится вокруг C=0.1.

Тонкая настройка машиннообучаемых моделей методом сеточного поиска

В машинном обучении имеются два типа параметров: параметры, извлекаемые из тренировочных данных, например весовые коэффициенты в логистической регрессии, и параметры алгоритма обучения, которые оптимизируются отдельно. Вторые из перечисленных – это настроечные параметры, или так называемые гиперпараметры модели, например параметр **регуляризации** в логистической регрессии или параметр **глубины** дерева решений.

В предыдущем разделе мы использовали проверочные кривые для улучшения качества модели путем тонкой настройки одного из ее гиперпараметров. В этом разделе мы рассмотрим мощную методику гиперпараметрической оптимизации под названием «метод **поиска по сетке параметров**»[1], который поможет еще более улучшить качество модели путем нахождения *оптимальной* комбинации значений гиперпараметров.

Настройка гиперпараметров методом поиска по сетке параметров

Принцип работы метода поиска по сетке параметров довольно прост, это парадигма исчерпывающего поиска «путем грубой силы», где мы задаем список значений для различных гиперпараметров, и компьютер оценивает качество модели для их каждой комбинации с целью получения оптимального набора:

---

[1] Оптимизационный поиск по сетке параметров, или сеточный поиск (grid search) — это попросту исчерпывающий поиск в заданном вручную подмножестве гиперпараметрического пространства с целью нахождения оптимальной комбинации гиперпараметров. – *Прим. перев.*

```
>>>
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
pipe_svc = Pipeline([('scl', StandardScaler()),
                     ('clf', SVC(random_state=1))])
param_range = [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0]
param_grid = [{'clf__C': param_range,
               'clf__kernel': ['linear']},
              {'clf__C': param_range,
               'clf__gamma': param_range,
               'clf__kernel': ['rbf']}]
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=10,
                  n_jobs=-1)
gs = gs.fit(X_train, y_train)
print(gs.best_score_)  # перекрестно-проверочная верность
0.978021978022
print(gs.best_params_) # наилучшие параметры
{'clf__C': 0.1, 'clf__kernel': 'linear'}
```

Используя приведенный выше исходный код, мы инициализировали объект GridSearchCV из модуля sklearn.model_selection, чтобы натренировать и настроить конвейер **метода опорных векторов** (SVM). Мы назначаем параметру param_grid объекта GridSearchCV список словарей с определением параметров, которые мы хотели бы настроить. Для линейного SVM мы оценили всего один параметр обратной регуляризации C; для ядерного SVM с РБФ в качестве ядра мы выполнили настройку двух параметров: C и gamma. Отметим, что параметр gamma имеет непосредственное отношение к ядерным SVM. По результатам применения тренировочных данных с выполнением оптимизационного поиска по сетке параметров в атрибуте best_score мы получили оценку наиболее качественной модели и обратились к его параметрам через атрибут best_params_. В данном отдельно взятом случае линейная модель SVM с 'clf__C'=0.1 выдала наилучшую верность на данных k-блочной перекрестной проверки: 97.8%.

Наконец, мы воспользуемся независимым тестовым набором данных для оценки качества наилучшей отобранной модели, которая доступна через атрибут best_estimator_ объекта GridSearchCV:

```
>>>
clf = gs.best_estimator_
clf.fit(X_train, y_train)
print('Верность на тестовом наборе: %.3f' % clf.score(X_test, y_test))
Верность на тестовом наборе: 0.965
```

Поиск по сетке параметров представляет собой мощный метод нахождения оптимального набора параметров. Между тем в вычислительном плане оценка всех возможных комбинаций параметров является одновременно очень затратной. Альтернативный подход к отбору разных комбинаций параметров в scikit-learn представлен методом рандомизированного поиска. Используя класс RandomizedSearchCV библиотеки scikit-learn, из выборочных распределений[1] с заданным запасом можно отбирать случайные комби-

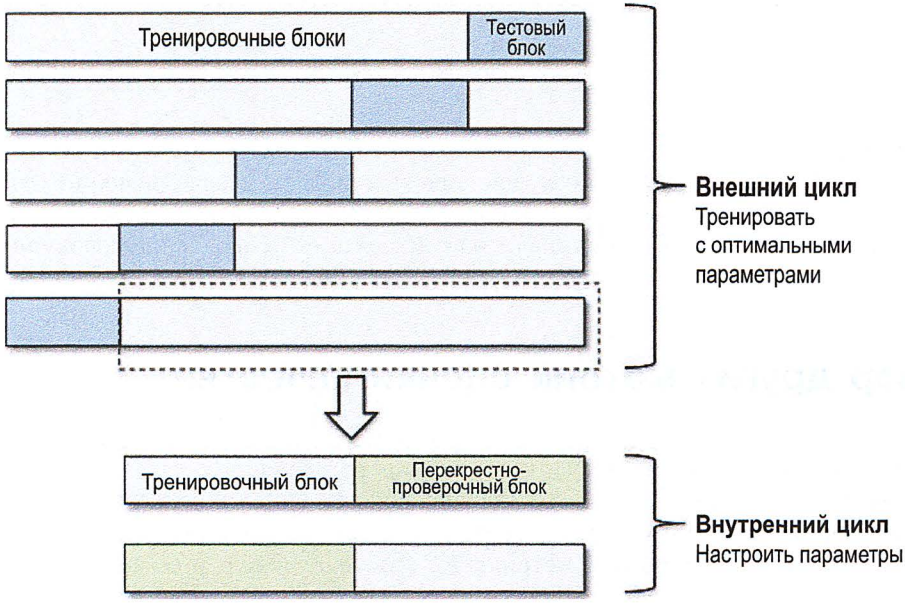
[1] Выборочное распределение (sampling distribution) – распределение вероятностей заданной статистики (например, среднего или медианы), полученное с помощью большого числа образцов, отобранных из конкретной генеральной совокупности. – *Прим. перев.*

нации параметров. Более подробно и с большим количеством примеров эта тема рассматривается на http://scikit-learn.org/stable/modules/grid_search.html#randomized-parameter-optimization.

Отбор алгоритмов методом вложенной перекрестной проверки

Как мы убедились в предыдущем подразделе, использование k-блочной перекрестной проверки в сочетании с оптимизационным поиском по сетке параметров является полезным подходом для тонкой настройки качества машиннообучаемой модели, который выполняется путем варьирования значений ее гиперпараметров. Если же мы хотим выполнить отбор среди разных алгоритмов машинного обучения, то рекомендуется использовать еще один подход – вложенную перекрестную проверку. В своем замечательном исследовании смещения в оценке ошибок Варма и Саймон пришли к заключению, что действительная ошибка оценки при использовании вложенной перекрестной проверки почти не смещается относительно тестового набора (S. Varma and R. Simon, «Bias in Error Estimation When Using Cross-validation for Model Selection», BMC bioinformatics, 7(1):91, 2006 (*«Смещение в оценке погрешности при использовании кросс-валидации для отбора моделей»*)).

Во вложенной перекрестной проверке имеется внешний цикл k-блочой перекрестной проверки для разделения данных на тренировочные и тестовые блоки и внутренний цикл, который используется для отбора модели при помощи k-блочной перекрестной проверки на тренировочном блоке. После отбора модели затем тестовый блок используется для оценки качества модели. Следующий ниже рисунок объясняет идею вложенной перекрестной проверки с пятью внешними и двумя внутренними блоками, которая может применяться для больших наборов данных, где важна вычислительная производительность; этот конкретный тип вложенной перекрестной проверки также называется **перекрестной проверкой 5×2**:



В библиотеке scikit-learn выполнить вложенную перекрестную проверку можно следующим образом:

```
>>>
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring='accuracy',
                  cv=2,
                  n_jobs=-1)
scores = cross_val_score(gs, X_train, y_train, scoring='accuracy', cv=5)
print('Перекрестно-проверочная верность: %.3f +/- %.3f' % (
                    np.mean(scores), np.std(scores)))
Перекрестно-проверочная верность: 0.965 +/- 0.025
```

Результирующая средняя верность на перекрестно-проверочных данных дает хорошую оценку того, чего ожидать, если настроить гиперпараметры модели и затем использовать ее на ранее не встречавшихся данных. Например, мы можем применить подход с вложенной перекрестной проверкой для сравнения модели на основе SVM с классификатором на основе простого дерева решений; для простоты мы настроим лишь ее параметр глубины:

```
>>>
from sklearn.tree import DecisionTreeClassifier
gs = GridSearchCV(
         estimator=DecisionTreeClassifier(random_state=0),
         param_grid=[
         {'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
         scoring='accuracy',
         cv=5)
scores = cross_val_score(gs,
                         X_train,
                         y_train,
                         scoring='accuracy',
                         cv=2)
print('Перекрестно-проверочная верность: %.3f +/- %.3f' % (
                        np.mean(scores), np.std(scores)))
Перекрестно-проверочная верность: 0.921 +/- 0.029
```

Как здесь видно, качество работы на данных вложенной перекрестной проверки модели SVM (97.8%) заметно лучше качества модели дерева решений (90.8%). И поэтому она ожидаемо может быть более подходящим выбором для классификации новых данных, которые поступают из той же самой генеральной совокупности, что и этот отдельно взятый набор данных.

Обзор других метрик оценки качества

В предыдущих разделах и главах мы оценивали наши модели при помощи метрики верности модели (model accuracy), которая широко применяется для количественной оценки качества модели в целом. Однако существует и ряд других метрик оценки качества, которые могут использоваться для измерения релевантности модели, такие как **точность**, **полнота** и **метрика F1** (соответственно precision, recall и F1 score).

Прочтение матрицы несоответствий

Прежде чем мы подробно остановимся на различных оценочных метриках, сначала распечатаем так называемую **матрицу несоответствий** (матрицу ошибок, confusion matrix), в которой качество алгоритма обучения разложено по ячейкам. Матрица несоответствий – это просто квадратная таблица, которая сообщает о числе **истинно положительных**, **истинно отрицательных**, **ложноположительных** и **ложноотрицательных** предсказаний классификатора, как показано на нижеследующем рисунке[1]:

		Спрогнозированный класс	
		P	*N*
Фактический класс	*P*	Истинно положительные (TP)	Ложно-отрицательные (FN)
	N	Ложно-положительные (FP)	Истинно отрицательные (TN)

Хотя эти метрики можно легко вычислить вручную путем сравнения истинных и предсказанных меток классов, библиотека scikit-learn предлагает вспомогательную функцию confusion_matrix, которую можно использовать следующим образом:

```
>>>
from sklearn.metrics import confusion_matrix
pipe_svc.fit(X_train, y_train)
y_pred = pipe_svc.predict(X_test)
confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
print(confmat)
[[71  1]
 [ 2 40]]
```

После выполнения приведенного выше исходного кода был получен массив, который предоставляет информацию о различных типах ошибок, сделанных классификатором на тестовом наборе данных, которые можно проиллюстрировать, изобразив матрицу несоответствий из предыдущего рисунка при помощи функции matshow библиотеки matplotlib:

```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
    for j in range(confmat.shape[1]):
        ax.text(x=j, y=i,
                s=confmat[i, j],
                va='center', ha='center')
plt.xlabel('распознанная метка')
plt.ylabel('истинная метка')
plt.show()
```

[1] Для справки: true positives (TP) – истинно положительные исходы, true negatives (TN) – истинно отрицательные исходы, false positives (FP) – ложноположительные исходы и false negatives (FN) – ложноотрицательные исходы. – *Прим. перев.*

Теперь график матрицы несоответствий, как показано здесь, должен немного упростить интерпретацию исходов:

Приняв, что в этом примере класс 1 (злокачественный) – это положительный класс, наша модель правильно классифицировала 71 образец, которые принадлежат классу 0 (истинно отрицательные исходы), и соответственно 40 образцов, которые принадлежат классу 1 (истинно положительные исходы). Однако наша модель также ошибочно классифицировала 1 образец из класса 0 как класс 1 (ложноположительные исходы) и предсказала, что 2 образца представляют собой доброкачественную опухоль, несмотря на то что это злокачественная опухоль (ложноотрицательные исходы). В следующем разделе мы узнаем, каким образом можно использовать эту информацию для вычисления разнообразных метрик ошибок (погрешностей).

Оптимизация точности и полноты классификационной модели

Ошибка (ERR) и **верность** (ACC) прогноза предоставляют общую информацию о том, сколько образцов неправильно/правильно классифицировано. Под ошибкой может пониматься сумма всех ложных прогнозов, деленная на общее число прогнозов, а верность, соответственно, вычисляется как сумма правильных прогнозов, деленная на это же число прогнозов:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}.$$

В этом случае верность прогноза можно вычислить непосредственно из ошибки:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR.$$

Метрики оценки качества «**доля истинно положительных исходов**» (true positive rate, TPR) и «**доля ложноположительных исходов**» (false positive rate, FPR) особенно полезны для задач с несбалансированными классами[1]:

[1] Доля истинно положительных исходов, или *чувствительность* (также полнота), измеряет долю положительных исходов, которые правильно определены как таковые (например, доля больных людей, которые правильно определены как имеющие заболевание). Доля истинно отрицательных исходов, или *специфичность*, измеряет долю отрицательных исходов, которые правильно определены как таковые (например, доля здоровых людей, которые правильно определены как не имеющие заболевания). – *Прим. перев.*

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN};$$
$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}.$$

В диагностике опухоли, например, мы более заинтересованы в обнаружении злокачественных опухолей, для того чтобы помочь пациенту с соответствующим лечением. Однако также важно уменьшить число неправильно классифицированных как злокачественные (ложноположительные исходы) доброкачественных опухолей, с тем чтобы не беспокоить пациента без необходимости. В отличие от FPR, доля истинно положительных исходов предоставляет полезную информацию о доле положительных (или релевантных) образцов, которые были правильно идентифицированы из общего пула положительных исходов (P).

Точность (PRE) и **полнота** (REC) — это метрики оценки качества, которые связаны с долями истинно положительных и истинно отрицательных исходов, при этом фактически полнота является синонимом доли истинно положительных исходов:

$$PRE = \frac{TP}{TP + FP};$$
$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}.$$

На практике часто комбинация точности и полноты используется в так называемой метрике **F1**:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}.$$

Все эти оценочные метрики реализованы в scikit-learn и могут быть импортированы из модуля sklearn.metrics, как показано в следующем ниже фрагменте:

```
>>>
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score
print('Точность: %.3f' % precision_score(
                        y_true=y_test, y_pred=y_pred))
Точность: 0.976
print('Полнота: %.3f' % recall_score(
                        y_true=y_test, y_pred=y_pred))
Полнота: 0.952
print('Метрика F1: %.3f' % f1_score(
                        y_true=y_test, y_pred=y_pred))
Метрика F1: 0.964
```

Кроме того, в классе оптимизационного поиска по сетке параметров GridSearch мы можем использовать другую оценочную метрику, помимо метрики верности, используя для этого параметр scoring. Полный список разных значений, которые принимаются этим параметром, можно найти на http://scikit-learn.org/stable/modules/model_evaluation.html.

Напомним, что положительный класс в библиотеке scikit-learn – это класс, который маркирован как класс 1. Если мы хотим указать другую *положительную метку*, то мы можем создать наш собственный регистратор оценок (scorer), воспользо-

вавшись для этого функцией make_scorer, которую можно затем непосредственно предоставить в качестве значения параметра scoring в GridSearchCV:

```
from sklearn.metrics import make_scorer, f1_score
scorer = make_scorer(f1_score, pos_label=0)
gs = GridSearchCV(estimator=pipe_svc,
                  param_grid=param_grid,
                  scoring=scorer,
                  cv=10)
```

Построение графика характеристической кривой

Графики характеристической кривой (рабочей характеристики получателя, receiver operator characteristic, ROC), или графики ROC-кривой, или кривой ошибок, – это широко используемый инструмент для отбора моделей для классификации данных, основываясь на их качестве относительно долей ложноположительных и истинно-положительных исходов, которые рассчитываются путем смещения порога решения классификатора. Диагональ ROC-графика можно интерпретировать как случайное гадание, и модели классификации, которые попадают ниже диагонали, считаются хуже случайного гадания. Идеальный классификатор попал бы в верхний левый угол графика с долей истинно положительных исходов, равной 1, и долей ложноположительных исходов, равной 0. Основываясь на ROC-кривой, затем можно вычислить показатель так называемой **площади под кривой** (area under the curve, AUC)[1], чтобы охарактеризовать качество классификационной модели.

Аналогично ROC-кривым можно вычислить кривые точности-полноты для различных вероятностных порогов классификатора. Функция для построения графика кривых точности-полноты также реализована в scikit-learn и задокументирована на веб-странице http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html.

Выполнив следующий ниже пример исходного кода, мы построим график ROC-кривой классификатора, в котором используются всего два признака из набора данных о раке молочной железы, чтобы идентифицировать, является опухоль доброкачественной или злокачественной. Несмотря на то что мы собираемся использовать тот же самый конвейер с логистической регрессией, который мы определили ранее, мы усложняем задачу классификации для классификатора так, чтобы итоговая ROC-кривая стала визуально интереснее. По аналогичным соображениям мы также снижаем количество блоков до 3 в валидаторе стратифицированной k-блочной перекрестной проверки StratifiedKFold. Соответствующий исходный код выглядит следующим образом:

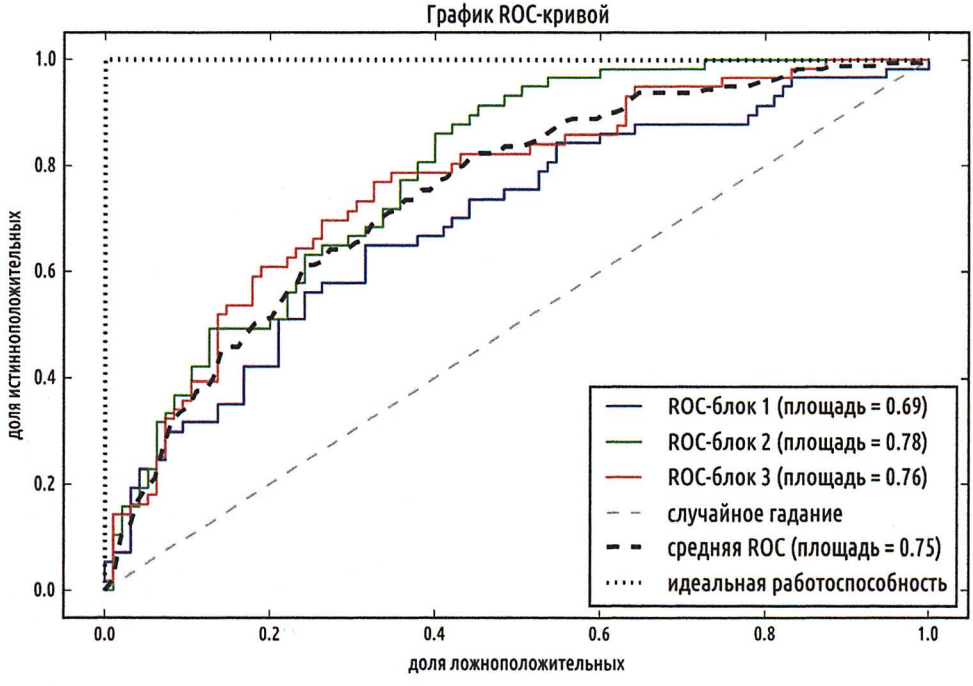
```
from sklearn.metrics import roc_curve, auc
from scipy import interp
pipe_lr = Pipeline([('scl', StandardScaler()),
                    ('pca', PCA(n_components=2)),
                    ('clf', LogisticRegression(penalty='l2',
```

[1] Площадь под ROC-кривой – площадь, ограниченная ROC-кривой и осью доли ложноположительных классификаций. Чем выше оценка AUC, тем качественнее классификатор, при этом значение 0.5 демонстрирует непригодность выбранного метода классификации (соответствует случайному гаданию). – *Прим. перев.*

```
                                                random_state=0,
                                                C=100.0))])
X_train2 = X_train[:, [4, 14]]
cv = StratifiedKFold(y_train,
                     n_folds=3,
                     random_state=1)
fig = plt.figure(figsize=(7, 5))
mean_tpr = 0.0
mean_fpr = np.linspace(0, 1, 100)
all_tpr = []

for i, (train, test) in enumerate(cv):
    probas = pipe_lr.fit(X_train2[train],
                         y_train[train]).predict_proba(X_train2[test])
    fpr, tpr, thresholds = roc_curve(y_train[test],
                                     probas[:, 1],
                                     pos_label=1)
    mean_tpr += interp(mean_fpr, fpr, tpr)
    mean_tpr[0] = 0.0
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr,
             tpr,
             lw=1,
             label='ROC-блок %d (площадь = %0.2f)'
                   % (i+1, roc_auc))
plt.plot([0, 1],
         [0, 1],
         linestyle='--',
         color=(0.6, 0.6, 0.6),
         label='случайное гадание')

mean_tpr /= len(cv)
mean_tpr[-1] = 1.0
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, 'k--',
         label='средняя ROC (площадь = %0.2f)' % mean_auc, lw=2)
plt.plot([0, 0, 1],
         [0, 1, 1],
         lw=2,
         linestyle=':',
         color='black',
         label='идеальное качество')
plt.xlim([-0.05, 1.05])
plt.ylim([-0.05, 1.05])
plt.xlabel('доля ложноположительных')
plt.ylabel('доля истинноположительных')
plt.title('График ROC-кривой')
plt.legend(loc="lower right")
plt.show()
```

В приведенном выше примере мы использовали уже знакомый нам класс StratifiedKFold библиотеки scikit-learn и при помощи функции ROC-кривой roc_curve модуля sklearn.metrics отдельно для каждой итерации вычислили ROC-качество логистического регрессионного классификатора LogisticRegression из конвейера pipe_lr. Кроме того, при помощи функции interp, которую мы импортировали из библиотеки SciPy, мы интерполировали среднюю ROC-кривую, исходя из трех блоков, и вычислили функцией auc площадь под кривой. Получившаяся ROC-кривая показывает,

что между разными блоками имеется определенная степень дисперсии, и средняя площаль под ROC-кривой (ROC AUC) (0.75) попадает между идеальной оценкой (1.0) и случайным гаданием (0.5):



Если мы интересуемся просто оценкой площади под ROC-кривой (ROC AUC), то из подмодуля sklearn.metrics можно также непосредственно импортировать функцию roc_auc_score. В следующем ниже фрагменте исходного кода вычисляется оценка площади под ROC-кривой классификатора на независимом тестовом наборе данных после его подгонки под тренировочный набор из двух признаков:

```
>>>
pipe_lr = pipe_lr.fit(X_train2, y_train)
y_pred2 = pipe_lr.predict(X_test[:, [4, 14]])
from sklearn.metrics import roc_auc_score
from sklearn.metrics import accuracy_score
print('ROC AUC: %.3f' % roc_auc_score(
      y_true=y_test, y_score=y_pred2))
ROC AUC: 0.662
print('Верность: %.3f' % accuracy_score(
      y_true=y_test, y_pred=y_pred2))
Верность: 0.711
```

Сведения о качестве классификатора как площади под ROC-кривой (ROC AUC) помогают глубже оценить качество работы классификатора относительно несбалансированных образцов. Однако хотя оценку верности можно интерпретировать как

единичную точку отсечения на кривой ROC, А. П. Брэдли показал, что площадь под ROC-кривой и метрики верности в основном друг с другом согласуются (A. P. Bradley, «The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms», Pattern recognition, 30 (7):1145-1159, 1997 (*«Использование площади под кривой ROC при оценке алгоритмов машинного обучения»*)).

Оценочные метрики для многоклассовой классификации

Оценочные метрики, которые мы обсудили в этом разделе, характерны для систем бинарной классификации. Однако в библиотеке scikit-learn также реализованы методы **макро-** и **микроусреднения**, которые расширяют эти оценочные метрики, применяемые в задачах многоклассовой классификации, за счет классификации данных методом **один против всех** (OvA). Микросреднее рассчитывается из индивидуальных истинно положительных, истинно отрицательных, ложноположительных и ложноотрицательных исходов системы. Например, микросреднее оценки точности в k-классовой системе может быть вычислено следующим образом:

$$PRE_{\text{micro}} = \frac{TP_1 + \ldots + TP_k}{TP_1 + \ldots + TP_k + FP_1 + \ldots + FP_k}.$$

Макросреднее просто рассчитывается как средние оценки разных систем:

$$PRE_{\text{macro}} = \frac{PRE_1 + \ldots + PRE_k}{k}.$$

Микроусреднение полезно применять, если мы хотим одинаково взвесить каждый экземпляр либо прогноз, тогда как макроусреднение взвешивает все классы одинаково с целью получения оценки общего качества работы классификатора относительно самых частотных меток классов.

Если мы используем бинарные метрики качества для оценки моделей многоклассовой классификации, то в этом случае в библиотеке scikit-learn по умолчанию используется нормализованный либо взвешенный вариант макросреднего. Взвешенное макросреднее рассчитывается путем взвешивания оценки каждой метки класса на число истинных экземпляров при вычислении среднего. Взвешенное макросреднее полезно, если мы имеем дело с неустойчивостями классов, т. е. разными количествами экземпляров для каждой метки.

Несмотря на то что в библиотеке scikit-learn взвешенное макросреднее по умолчанию используется в многоклассовых задачах, тем не менее в параметре `average` мы можем определять метод усреднения внутри разных оценочных функций, которые мы импортируем из модуля `sklearn.metrics`, например внутри функций `precision_score` или `make_scorer`:

```
pre_scorer = make_scorer(score_func=precision_score,
                         pos_label=1,
                         greater_is_better=True,
                         average='micro')
```

Резюме

В начале этой главы мы обсудили, как соединять в цепочку различные методы преобразования и классификаторы в виде удобных модельных конвейеров, которые помогли нам эффективнее тренировать и тестировать машиннообучаемые модели. Затем мы использовали эти конвейеры для выполнения одного из ключевых методов отбора и оценивания модели – k-блочную перекрестную проверку. Используя k-блочную перекрестную проверку, мы построили график с кривой обучения и проверочной (валидационной) кривой для диагностирования типичных проблем алгоритмов обучения, таких как переобучение и недообучение. Затем, используя оптимизационный поиск по сетке параметров, мы выполнили тонкую настройку модели. Мы завершили эту главу рассмотрением матрицы несоответствий и всевозможных метрик оценки качества, которые могут быть полезными для дальнейшей оптимизации модели для конкретной практической задачи. Теперь мы достаточно хорошо оснащены ключевыми методами, чтобы успешно строить модели машинного обучения с учителем для классификации данных.

В следующей главе мы рассмотрим ансамблевые методы, т. е. методы, которые позволяют комбинировать две и более моделей и алгоритмов классификации данных для дальнейшего повышения предсказательной способности системы машинного обучения.

Глава 7

Объединение моделей для методов ансамблевого обучения

В предыдущей главе мы сосредоточились на наиболее успешных практических методах тонкой настройки и оценки разных моделей классификации данных. В этой главе, опираясь на эти методы, мы разберем методы построения наборов классификаторов, которые нередко могут давать более хорошую предсказательную способность, чем любой из ее индивидуальных членов. Вы узнаете о том, как:

- выполнять прогнозы на основе мажоритарного голосования;
- сократить переобучение путем извлечения случайных комбинаций из тренировочного набора с повторами;
- строить мощные модели из *слабых учеников*, которые обучаются на своих ошибках.

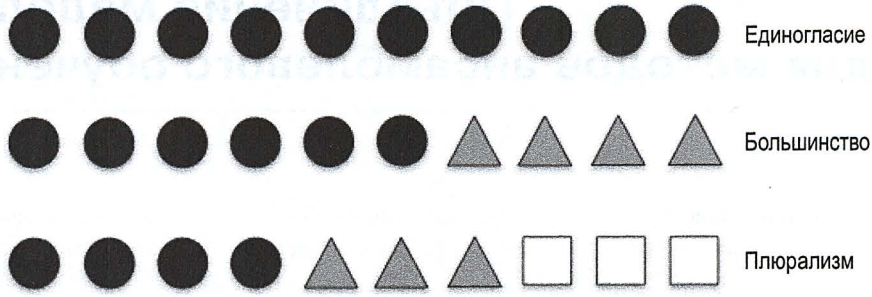
Обучение при помощи ансамблей

Задача **ансамблевых методов** состоит в том, чтобы объединять разные классификаторы в метаклассификатор, чья обобщающая способность лучше, чем у каждого индивидуального классификатора в отдельности. Например, допустим, мы собрали прогнозы от 10 экспертов. В этой ситуации ансамблевые методы позволяют стратегически объединить эти 10 экспертных прогнозов, чтобы выйти с прогнозом, который будет точнее и устойчивее, чем прогноз любого отдельно взятого эксперта. Как мы увидим далее в этой главе, для создания ансамбля классификаторов существует несколько разных подходов. В этом разделе мы представим основной принцип работы ансамблей и объясним, почему они получили признательность за свою, как правило, хорошую обобщающую способность.

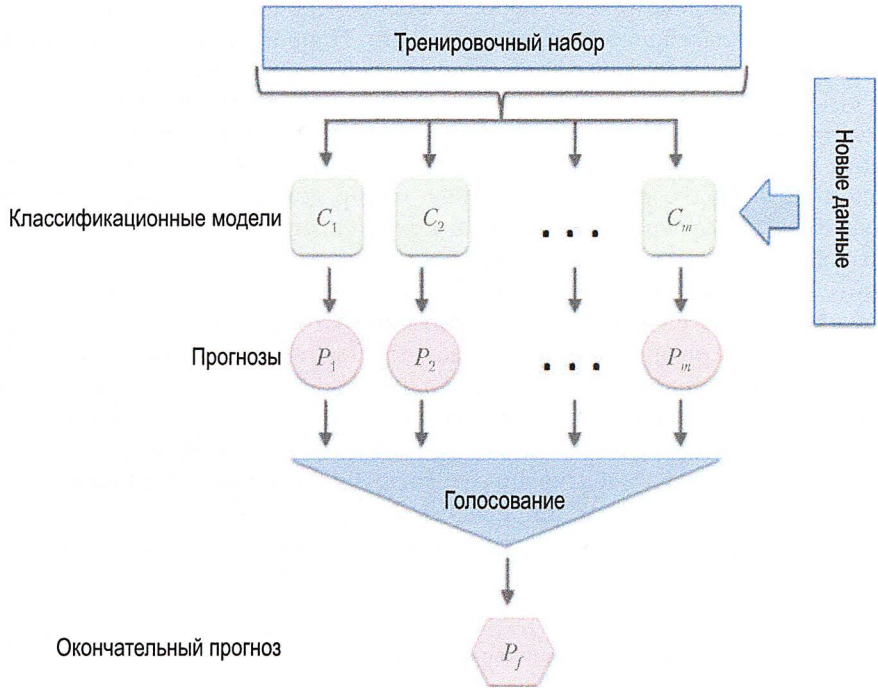
В этой главе мы сосредоточимся на самых популярных ансамблевых методах, в которых используется принцип **мажоритарного голосования** (majority voting). Мажоритарное голосование просто означает, что мы отбираем метку класса, идентифицированную большинством классификаторов, т. е. получившую более 50% голосов. Строго говоря, термин «мажоритарное голосование» относится исключительно к двухклассовой конфигурации. Однако этот принцип голосования легко обобщается на многоклассовую конфигурацию, и тогда он называется **плюралистическим голосованием** (plurality voting). В последнем случае мы отбираем метку класса, получившую максимум голосов (взяв моду[1]). Следующая ниже схема иллюстрирует

[1] Мода (mode) – значение во множестве наблюдений, которое встречается наиболее часто, т. е. значение признака, имеющее наибольшую частоту в статистическом ряду распределения. – *Прим. перев.*

идею мажоритарного и плюралистического голосования для ансамбля из 10 классификаторов, где каждый уникальный символ (треугольник, квадрат и круг) представляет уникальную метку класса:



Мы начинаем с того, что при помощи тренировочного набора данных тренируем m разных классификаторов (C_1, ..., C_m). В зависимости от методики ансамбль может быть построен из разных алгоритмов классификации, например деревьев решений, метода опорных векторов, логистических регрессионных классификаторов и т. д. Как вариант можно также использовать одинаковый базовый алгоритм классификации для подгонки разных подмножеств тренировочного набора данных. Видным примером этого подхода может быть алгоритм случайного леса, объединяющий разные классификаторы на основе деревьев решений. Следующая ниже схема иллюстрирует общий принцип работы ансамблевого подхода с использованием мажоритарного голосования:



Для идентификации метки класса простым мажоритарным либо плюралистическим голосованием мы объединяем идентифицированные метки классов от каждого отдельного классификатора C_j и отбираем метку $\hat{y}$ класса, которая получила максимум голосов:

$$\hat{y} = \text{mode}\{C_1(x), C_2(x), ..., C_m(x)\}.$$

Например, в задаче двухклассовой классификации, где class1 = −1 и class2 = +1, мы можем записать прогноз на основе мажоритарного голосования следующим образом:

$$C(\boldsymbol{x}) = \text{sign}\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1, & \text{если} \sum_i C_j(\boldsymbol{x}) \geq 0 \\ -1, & \text{иначе} \end{cases}.$$

Для иллюстрации того, почему ансамблевые методы могут работать лучше индивидуальных классификаторов, взятых в отдельности, применим простые принципы из комбинаторики. Пусть в следующем ниже примере все n базовых классификаторов для задачи двухклассовой классификации имеют одинаковую частоту появления ошибок ε. Кроме того, примем, что классификаторы независимы и частоты ошибок не коррелированы. При таких допущениях можно выразить вероятность появления ошибки ансамбля из базовых классификаторов просто как функцию массы вероятности биномиального распределения[1]:

$$P(y \geq k) = \sum_k^n \left\langle {n \atop k} \right\rangle \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{\text{ансамбль}}.$$

Здесь $\left\langle {n \atop k} \right\rangle$ – это биномиальный коэффициент *из n по k*. Другими словами, мы вычисляем вероятность, что прогноз ансамбля ошибочный. Теперь обратимся к более конкретному примеру из 11 базовых классификаторов (n = 11) с частотой ошибок 0.25 (ε = 0.25):

$$P(y \geq k) = \sum_{k=6}^{11} \left\langle {11 \atop k} \right\rangle 0.25^k (1-0.25)^{11-k} = 0.034.$$

Как видно, частота ошибок ансамбля (0.034) намного ниже частоты ошибок каждого индивидуального классификатора (0.25), в случае если все допущения соблюдены. Отметим, что в этом упрощенном примере разделение 50 на 50 на равное число классификаторов n трактуется как ошибка, поскольку верно лишь в половине случаев. Чтобы сравнить такой идеализированный ансамблевый классификатор с базовым классификатором на диапазоне разных базовых частот ошибок, реализуем функцию массы вероятности на Python:

---

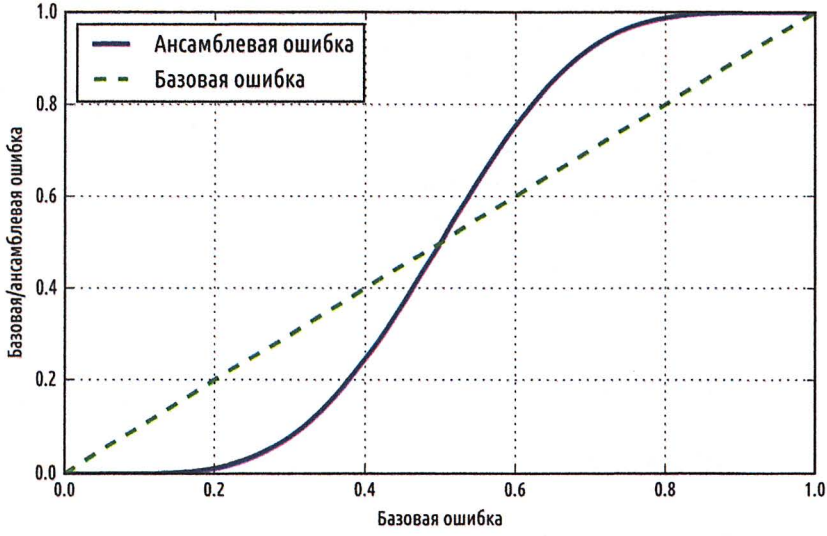
[1] Функция массы вероятности (probability mass function, PMF) дает относительную вероятность того, что случайная дискретная величина в точности равна некоторому значению. Биноминальное распределение – распределение количества «успехов» в последовательности из n независимых случайных экспериментов, таких что вероятность «успеха» в каждом из них равна p. – *Прим. перев.*

```
>>>
from scipy.misc import comb
import math
def ensemble_error(n_classifier, error):
    k_start = int(math.ceil(n_classifier / 2.0)) #int для совместим. с Python 2.7
    probs = [comb(n_classifier, k) *
             error**k *
             (1-error)**(n_classifier - k)
             for k in range(k_start, n_classifier + 1)]
    return sum(probs)
ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

Реализовав функцию ensemble_error, можно вычислить частоты ансамблевых ошибок применительно к диапазону разных базовых ошибок от 0.0 до 1.0, чтобы представить связь между ансамблевыми и базовыми ошибками в наглядном виде на линейном графике:

```
import numpy as np
error_range = np.arange(0.0, 1.01, 0.01)
ens_errors = [ensemble_error(n_classifier=11, error=error)
              for error in error_range]
import matplotlib.pyplot as plt
plt.plot(error_range, ens_errors,
         label='Ансамблевая ошибка',
         linewidth=2)
plt.plot(error_range, error_range,
         linestyle='--', label='Базовая ошибка',
         linewidth=2)
plt.xlabel('Базовая ошибка')
plt.ylabel('Базовая/ансамблевая ошибка')
plt.legend(loc='upper left')
plt.grid()
plt.show()
```

Как видно на итоговом графике, вероятность ансамблевой ошибки всегда лучше того же показателя индивидуального базового классификатора до тех пор, пока базовые классификаторы работают лучше случайного гадания ($\varepsilon < 0.5$). Отметим, что ось Y изображает базовую ошибку, т. е. ошибку базового классификатора (пунктирная линия) и ансамблевую ошибку (сплошная линия):



Реализация простого классификатора с мажоритарным голосованием

После краткого введения в ансамблевое обучение в предыдущем разделе начнем с разминочного упражнения и реализуем на Python простой ансамблевый классификатор с мажоритарным голосованием. Несмотря на то что следующий ниже алгоритм на основе плюралистического голосования обобщается и на многоклассовые конфигурации, мы, как это нередко делается в литературе, используем термин *мажоритарное голосование* для простоты.

Мы собираемся реализовать алгоритм, который позволит объединить разные алгоритмы классификации, которым в качестве коэффициента доверия поставлены в соответствие индивидуальные веса. Наша задача – построить более мощный метаклассификатор, который уравновешивает слабые стороны индивидуальных классификаторов на отдельно взятом наборе данных. В более четкой математической формулировке мы можем записать взвешенное мажоритарное голосование следующим образом:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A(C_j(\boldsymbol{x}) = i).$$

Здесь w_j – это вес, поставленный в соответствие базовому классификатору C_j, $\hat{y}$ – распознанная метка класса ансамбля, χ_A (греческая буква *хи*) – характеристическая функция $[C_j(\boldsymbol{x}) = i \in A]$ и A – набор уникальных меток классов. Для равных весов мы можем упростить это уравнение и записать его следующим образом:

$$\hat{y} = \text{mode}\{C_1(x), C_2(x), ..., C_m(x)\}.$$

Чтобы получше разобраться в принципе работы *взвешивания*, обратимся теперь к более конкретному примеру. Пусть имеется ансамбль из трех базовых классификаторов C_j ($j \in \{0,1\}$) и нужно идентифицировать метку класса конкретного экземпляра x. Два из трех базовых классификаторов идентифицируют метку класса 0, и один C_3 прогнозирует, что образец принадлежит классу 1. Если взвесить прогнозы каждого базового классификатора одинаково, то мажоритарное голосование спрогнозирует, что образец принадлежит классу 0:

$$C_1(\boldsymbol{x}) \rightarrow 0,\ C_2(\boldsymbol{x}) \rightarrow 0,\ C_3(\boldsymbol{x}) \rightarrow 1;$$
$$\hat{y} = \text{mode}\{0, 0, 1\} = 0.$$

Теперь назначим вес 0.6 классификатору C_3 и соответственно взвесим классификаторы C_1 и C_2 на коэффициент 0.2.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A(C_j(\boldsymbol{x}) = i)$$
$$= \arg\max_i \left[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1\right] = 1.$$

В более интуитивном представлении, поскольку $3 \times 0.2 = 0.6$, можно сказать, что прогноз, сделанный классификатором C_3, имеет вес в три раза больше, чем прогнозы классификаторов, соответственно, C_1 или C_2. Можно записать это следующим образом:

$$\hat{y} = \text{mode}\{0, 0, 1, 1, 1\} = 1.$$

Для перевода принципа взвешенного мажоритарного голосования в исходный код на Python можно воспользоваться вспомогательными функциями библиотеки NumPy argmax и bincount:

```
>>>
import numpy as np
np.argmax(np.bincount([0, 0, 1], weights=[0.2, 0.2, 0.6]))
1
```

Как обсуждалось в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, в библиотеке scikit-learn определенные классификаторы благодаря методу predict_proba также могут возвращать вероятность спрогнозированной метки класса. Использование для мажоритарного голосования спрогнозированных вероятностей классов вместо меток классов может оказаться полезным, если классификаторы в ансамбле хорошо откалиброваны. Модифицированную версию мажоритарного голосования для прогнозирования меток классов из вероятностей можно записать следующим образом:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}.$$

Здесь p_{ij} – это спрогнозированная вероятность j-го классификатора для метки класса i.

Продолжая с нашим предыдущим примером, пусть имеется задача двухклассовой классификации с метками классов $i \in \{0,1\}$ и ансамбль из трех классификаторов C_j ($j \in \{1,2,3\}$). И пусть классификатор C_j возвращает следующие вероятности принадлежности классу для отдельно взятого образца $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1],\ C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2],\ C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6].$$

Тогда отдельные вероятности класса можно вычислить следующим образом:

$$p(i_0|\boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58;$$
$$p(i_1|\boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42;$$
$$\hat{y} = \arg\max_i \left[p(i_0 \mid \boldsymbol{x}), p(i_1 \mid \boldsymbol{x}) \right] = 0.$$

Чтобы реализовать взвешенное мажоритарное голосование, основываясь на вероятностях классов, можно снова воспользоваться библиотекой NumPy, а именно ее функциями numpy.average и np.argmax:

```
>>>
ex = np.array([[0.9, 0.1],
               [0.8, 0.2],
               [0.4, 0.6]])
p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
p
array([ 0.58, 0.42])
np.argmax(p)
0
```

Соединив все вместе, теперь реализуем классификатор на основе мажоритарного голосования MajorityVoteClassifier на Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator

class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    """ Ансамблевый классификатор на основе мажоритарного голосования

    Параметры
    ---------
    classifiers : массивоподобный, форма = [n_classifiers]
      Разные классификаторы для ансамбля

    vote : str, {'classlabel', 'probability'}
      По умолчанию: ‹classlabel'
      Если метка класса ‹classlabel', то прогноз
      основывается на argmax меток классов.
      В противном случае если ‹probability', то для
      прогноза метки класса используется
      argmax суммы вероятностей
      (рекомендуется для откалиброванных классификаторов).
```

```
    weights : массивоподобный, форма = [n_classifiers]
      Факультативно, по умолчанию: None
      Если предоставлен список из значений `int` либо `float`, то
      Классификаторы взвешиваются по важности;
      Если `weights=None`, то используются равномерные веса
    """

    def __init__(self, classifiers,
                 vote='classlabel', weights=None):
        self.classifiers = classifiers
        self.named_classifiers = {key: value for
                                  key, value in
                                  _name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        """ Выполнить подгонку классификаторов.

        Параметры
        ---------
        X : {массивоподобный, разреженная матрица},
            форма = [n_samples, n_features]
            Матрица с тренировочными образцами.

        y : массивоподобный, форма = [n_samples]
            Вектор целевых меток классов.

        Возвращает
        ----------
        self : объект
        """

        # Использовать LabelEncoder, чтобы гарантировать, что
        # метки классов начинаются с 0, что важно для
        # вызова np.argmax в self.predict

        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X,
                                        self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self
```

Для лучшего понимания отдельных частей в исходный код было добавлено много комментариев. Но прежде чем мы реализуем оставшиеся методы, сделаем небольшой перерыв и обсудим фрагмент исходного кода, который на первый взгляд может показаться озадачивающим. Мы использовали родительские классы BaseEstimator и ClassifierMixin, чтобы *бесплатно* получить немного базовой функциональности, включая методы get_params и set_params для установки и возврата значений параметров классификатора, и соответственно метод score для вычисления оценки точности прогнозирования. Кроме того, отметим, что мы импортировали модуль six, чтобы сделать ансамблевый мажоритарный классификатор MajorityVoteClassifier совместимым с Python 2.7.

Далее мы добавим метод predict для идентификации метки класса мажоритарным голосованием, основываясь на метках классов, в случае если мы инициализируем новый объект ансамблевого мажоритарного классификатора MajorityVoteClassifier параметром vote='classlabel'. Как вариант можно инициализировать этот классификатор, указав параметр vote='probability', для прогнозирования метки класса, основываясь на вероятностях принадлежности классу. Кроме того, мы также добавим метод predict_proba для возврата средних вероятностей, что пригодится для вычисления оценки **площади под ROC-кривой** (ROC AUC).

```
def predict(self, X):
    """ Спрогнозировать метки классов для X.
    Параметры
    ---------
    X : {массивоподобный, разреженная матрица},
        форма = [n_samples, n_features]
        Матрица тренировочных образцов.

    Возвращает
    ----------
    maj_vote : массивоподобный, форма = [n_samples]
      Спрогнозированные метки классов.
    """

    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X), axis=1)
    else: # голосование 'classlabel'

        # Аккумулировать результаты из вызовов clf.predict
        predictions = np.asarray([clf.predict(X)
                                  for clf in
                                  self.classifiers_]).T

        maj_vote = np.apply_along_axis(
                       lambda x:
                       np.argmax(np.bincount(x,
                                             weights=self.weights)),
                       axis=1,
                       arr=predictions)
    maj_vote = self.lablenc_.inverse_transform(maj_vote)
    return maj_vote

def predict_proba(self, X):
    """ Спрогнозировать вероятности классов для X.

    Параметры
    ---------
    X : { массивоподобный, разреженная матрица},
        форма = [n_samples, n_features]
        Тренировочные векторы,
        где n_samples - число образцов и
        n_features - число признаков.

    Возвращает
    ----------
    avg_proba : массивоподобный,
```

```
        форма = [n_samples, n_classes]
        Взвешенная средняя вероятность для
        каждого класса в расчете на образец.
    """

    probas = np.asarray([clf.predict_proba(X)
                         for clf in self.classifiers_])
    avg_proba = np.average(probas,
                           axis=0, weights=self.weights)
    return avg_proba

def get_params(self, deep=True):
    """Получить имена параметров классификатора для GridSearch"""
    if not deep:
        return super(MajorityVoteClassifier,
                     self).get_params(deep=False)
    else:
        out = self.named_classifiers.copy()
        for name, step in six.iteritems(self.named_classifiers):
            for key, value in six.iteritems(
                    step.get_params(deep=True)):
                out['%s__%s' % (name, key)] = value
        return out
```

Кроме того, отметим, что мы определили нашу собственную модифицированную версию методов get_params, чтобы использовать функцию _name_estimators для доступа в ансамбле к параметрам индивидуальных классификаторов. На первый взгляд она может показаться немного сложной, но позднее в этом разделе, когда мы займемся поиском по сетке параметров для выполнения тонкой настройки гиперпараметров, она приобретет понятный смысл.

Наша реализация ансамблевого мажоритарного классификатора MajorityVoteClassifier очень подходит для демонстрационных целей. Вместе с тем я также реализовал более сложную версию классификатора на основе мажоритарного голосования с использованием библиотеки scikit-learn. Она станет доступной как sklearn.ensemble.VotingClassifier в следующей версии (v0.17).

Объединение разных алгоритмов классификации методом мажоритарного голосования

Теперь настало время привести в действие реализованный нами в предыдущем разделе ансамблевый мажоритарный классификатор MajorityVoteClassifier. Но сначала подготовим набор данных, на котором мы сможем его проверить. Учитывая, что мы уже знакомы с методами загрузки наборов данных из CSV-файлов, мы сэкономим время, загрузив набор данных цветков **ириса** из модуля библиотеки scikit-learn. Кроме того, мы выберем всего два признака, **ширину чашелистика** и **длину лепестка**, чтобы еще больше усложнить задачу классификации. Несмотря на то что наш мажоритарный классификатор MajorityVoteClassifier обобщается на многоклассовые задачи, мы будем классифицировать образцы цветков только из двух классов, **ирис разноцветный** (Iris versicolor) и **ирис виргинский** (Iris virginica), чтобы вычислить оценку **площади под ROC-кривой** (ROC AUC). Соответствующий исходный код выглядит следующим образом:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
iris = datasets.load_iris()
X, y = iris.data[50:, [1, 2]], iris.target[50:]
le = LabelEncoder()
y = le.fit_transform(y)
```

Отметим, что для вычислений оценки площади под ROC-кривой (ROC AUC) библиотека scikit-learn использует метод predict_proba (если он применим). В *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* мы увидели, как вычисляются вероятности классов в логистических регрессионных моделях. В деревьях решений вероятности вычисляются из частотного вектора, создаваемого для каждого узла во время тренировки. Вектор аккумулирует значения частоты каждой метки класса, вычисляемой из распределения меток классов в этом узле. Затем частоты нормализуются (нормируются) так, чтобы в сумме составлять 1. В алгоритме *k* ближайших соседей метки классов *k* ближайших соседей агрегируются аналогичным образом, в результате чего возвращаются нормированные частоты меток классов. Несмотря на то что возвращаемые классификаторами на основе дерева решений и *k* ближайших соседей нормированные вероятности могут выглядеть похожими на вероятности, полученные из логистической регрессионной модели, мы должны осознавать, что они все-таки выводятся не из функций масс вероятностей.

Далее мы разделим образцы цветков ириса на тренировочные (50%) и тестовые (50%) данные:

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.5,
                                                    random_state=1)
```

Теперь при помощи тренировочного набора данных натренируем три разных классификатора – классификатор на основе логистический регрессии, дерева решений и *k* ближайших соседей – и посмотрим на их индивидуальные качественные характеристики путем 10-блочной перекрестной проверки на тренировочном наборе данных, перед тем как объединить их в ансамблевый классификатор:

```
>>>
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
import numpy as np
clf1 = LogisticRegression(penalty='l2',
                          C=0.001,
                          random_state=0)
clf2 = DecisionTreeClassifier(max_depth=1,
                              criterion='entropy',
                              random_state=0)
clf3 = KNeighborsClassifier(n_neighbors=1,
                            p=2,
                            metric='minkowski')
pipe1 = Pipeline([['sc', StandardScaler()],
                  ['clf', clf1]])
```

```
pipe3 = Pipeline([['sc', StandardScaler()],
                  ['clf', clf3]])
clf_labels = [‹Логистическая регрессия›, ‹Дерево решений›,
              'K ближайших соседей›]
print('10-блочная перекрестная проверка:\n')
for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train,
                             y=y_train,
                             cv=10,
                             scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
          % (scores.mean(), scores.std(), label))
print("* ROC AUC = площадь под ROC-кривой")
```

Полученные выходные данные, приведенные в следующем ниже фрагменте, демонстрируют, что предсказательная способность индивидуальных классификаторов почти одинаковая:

```
10-блочная перекрестная проверка:

ROC AUC: 0.92 (+/- 0.20) [Логистическая регрессия]
ROC AUC: 0.92 (+/- 0.15) [Дерево решений]
ROC AUC: 0.93 (+/- 0.10) [K ближайших соседей]
* ROC AUC = площадь под ROC-кривой
```

Вам, наверное, интересно, почему мы натренировали классификатор на основе логистической регрессии и *k* ближайших соседей в качестве составной части **конвейера**. Причина заключается в том, что, как уже обсуждалось в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, оба алгоритма: алгоритм логистической регрессии и алгоритм *k* ближайших соседей (использующий евклидову метрику расстояния) – не инвариантны к шкале данных, в отличие от алгоритма деревьев решений. Несмотря на то что все признаки цветков ириса измеряются в одинаковой шкале данных (см), работа со стандартизированными признаками должна войти в привычку.

Теперь перейдем к более захватывающей части и объединим индивидуальные классификаторы для голосования по принципу большинства голосов в нашем ансамблевом мажоритарном классификаторе MajorityVoteClassifier:

```
>>>
mv_clf = MajorityVoteClassifier(classifiers=[pipe1, clf2, pipe3])
clf_labels += ['Мажоритарное голосование']
all_clf = [pipe1, clf2, pipe3, mv_clf]
for clf, label in zip(all_clf, clf_labels):
    scores = cross_val_score(estimator=clf,
                             X=X_train,
                             y=y_train,
                             cv=10,
                             scoring='roc_auc')
    print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
                     % (scores.mean(), scores.std(), label))
ROC AUC: 0.92 (+/- 0.20) [Логистическая регрессия]
ROC AUC: 0.92 (+/- 0.15) [Дерево решений]
ROC AUC: 0.93 (+/- 0.10) [K ближайших соседей]
ROC AUC: 0.97 (+/- 0.10) [Мажоритарное голосование]
```

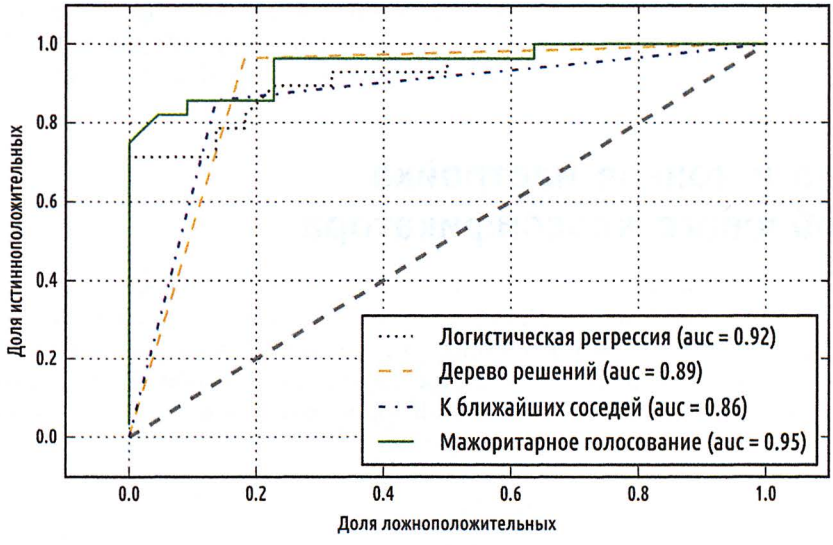
Как видно, в результате оценивания на основе 10-блочной перекрестной проверки качество ансамблевого мажоритарного классификатора MajorityVotingClassifier существенно улучшилась, по сравнению с индивидуальными классификаторами.

Оценка и тонкая настройка ансамблевого классификатора

В этом разделе мы вычислим ROC-кривые из тестового набора, чтобы убедиться, что классификатор MajorityVoteClassifier хорошо обобщается на ранее не встречавшиеся данные. Здесь следует напомнить, что тестовый набор не должен использоваться для отбора модели; его единственная задача – сообщать о несмещенной оценке обобщающей способности классификационной системы. Соответствующий исходный код выглядит следующим образом:

```
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
colors = ['black', 'orange', 'blue', 'green']
linestyles = [':', '--', '-.', '-']
for clf, label, clr, ls in zip(all_clf, clf_labels, colors, linestyles):
    # принимая, что метка положительного класса равна 1
    y_pred = clf.fit(X_train,
                     y_train).predict_proba(X_test)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_true=y_test,
                                     y_score=y_pred)
    roc_auc = auc(x=fpr, y=tpr)
    plt.plot(fpr, tpr,
             color=clr,
             linestyle=ls,
             label='%s (auc = %0.2f)' % (label, roc_auc))
plt.legend(loc='lower right')
plt.plot([0, 1], [0, 1],
         linestyle='--',
         color='gray',
         linewidth=2)
plt.xlim([-0.1, 1.1])
plt.ylim([-0.1, 1.1])
plt.grid()
plt.xlabel('Доля ложноположительных')
plt.ylabel('Доля истинно положительных')
plt.show()
```

Как видно из получившегося графика ROC-кривой, ансамблевый классификатор работает на тестовом наборе тоже хорошо (*ROC AUC = 0.95*), в то время как классификатор на основе *k* ближайших соседей, похоже, переобучен под тренировочные данные (тренировочный *ROC AUC = 0.93*, тестовый *ROC AUC = 0.86*):

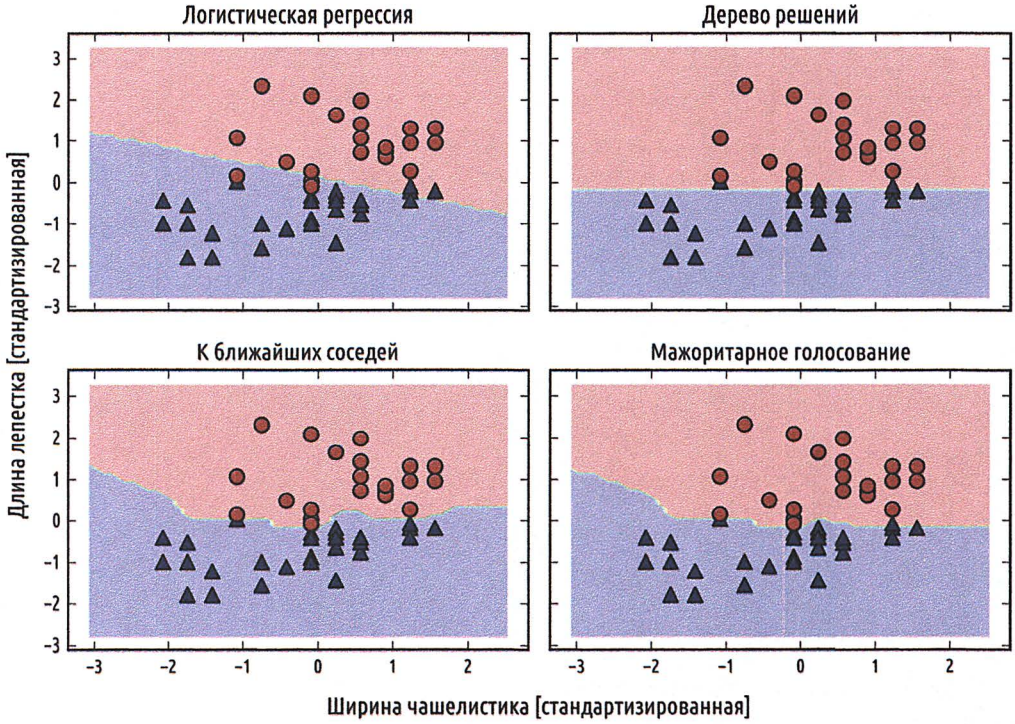


С учетом того, что для примеров классификации мы отобрали всего два признака, будет интересно увидеть, на что фактически похожа область решения ансамблевого классификатора. Несмотря на то что перед подгонкой модели нет необходимости стандартизировать тренировочные признаки, потому что наши конвейеры логистической регрессии и k ближайших соседей автоматически об этом позаботятся, в целях визуализации мы выполним стандартизацию тренировочного набора, в результате чего области решений дерева решений будут в той же самой шкале. Соответствующий исходный код выглядит следующим образом:

```
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
from itertools import product
x_min = X_train_std[:, 0].min() - 1
x_max = X_train_std[:, 0].max() + 1
y_min = X_train_std[:, 1].min() - 1
y_max = X_train_std[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
f, axarr = plt.subplots(nrows=2, ncols=2,
                        sharex='col',
                        sharey='row',
                        figsize=(7, 5))
for idx, clf, tt in zip(product([0, 1], [0, 1]),
                        all_clf, clf_labels):
    clf.fit(X_train_std, y_train)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
                                  X_train_std[y_train==0, 1],
                                  c='blue',
                                  marker='^',
                                  s=50)
```

```
    axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
                                  X_train_std[y_train==1, 1],
                                  c='red',
                                  marker='o',
                                  s=50)
    axarr[idx[0], idx[1]].set_title(tt)
plt.text(-3.5, -4.5,
         s='Ширина чашелистика [стандартизованная]',
         ha='center', va='center', fontsize=12)
plt.text(-10.5, 4.5,
         s='Длина лепестка [стандартизованная]',
         ha='center', va='center',
         fontsize=12, rotation=90)
plt.show()
```

Интересно и одновременно ожидаемо, что области решений ансамблевого классификатора, похоже, представляют собой гибрид областей решения индивидуальных классификаторов. На первый взгляд мажоритарная граница решения во многом выглядит как граница решения классификатора на основе *k* ближайших соседей. Однако мы видим, что она ортогональна к оси *Y* для *ширины чашелистика* ≥ 1, точно так же, как в одноуровневом дереве решений[1]:



[1] Одноуровневое дерево решений, также именуемое *пеньком решения* (decision stump), представляет собой дерево с одним внутренним узлом (корнем), который непосредственно подключен к терминальным узлам (своим листьям). Пенек решения выполняет прогноз, исходя из значения всего одного входного объекта. – *Прим. перев.*

Перед тем как вы узнаете, как выполнять тонкую настройку отдельных параметров классификатора для ансамблевой классификации, вызовем метод get_params, чтобы получить общее представление о том, каким образом обращаться к отдельным параметрам внутри объекта для поиска по сетке параметров GridSearch:

```
>>>
mv_clf.get_params()

{'decisiontreeclassifier': DecisionTreeClassifier(class_weight=None,
criterion='entropy', max_depth=1,

             max_features=None, max_leaf_nodes=None, min_samples_
leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...]
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
 'pipeline-1': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', LogisticRegression(C=0.001, class_
weight=None, dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear', tol=0.0001,
           verbose=0))]),
 'pipeline-1__clf': LogisticRegression(C=0.001, class_weight=None,
dual=False, fit_intercept=True,
           intercept_scaling=1, max_iter=100, multi_class='ovr',
           penalty='l2', random_state=0, solver='liblinear', tol=0.0001,
           verbose=0),
 'pipeline-1__clf__C': 0.001,
 'pipeline-1__clf__class_weight': None,
 'pipeline-1__clf__dual': False,
 [...]
 'pipeline-1__sc__with_std': True,
 'pipeline-2': Pipeline(steps=[('sc', StandardScaler(copy=True, with_
mean=True, with_std=True)), ('clf', KNeighborsClassifier(algorithm='au
to', leaf_size=30, metric='minkowski',
            metric_params=None, n_neighbors=1, p=2,
weights='uniform'))]),
 'pipeline-2__clf': KNeighborsClassifier(algorithm='auto', leaf_
size=30, metric='minkowski',
            metric_params=None, n_neighbors=1, p=2,
weights='uniform'),
 'pipeline-2__clf__algorithm': 'auto',
 [...]
 'pipeline-2__sc__with_std': True}
```

Основываясь на значениях, возвращенных методом get_params, мы теперь знаем, как получать доступ к индивидуальным атрибутам классификатора. Теперь в демонстрационных целях методом поиска по сетке параметров выполним тонкую настройку параметра обратной регуляризации C логистического регрессионного классификатора и глубины дерева решений. Соответствующий исходный код выглядит следующим образом:

```
from sklearn.model_selection import GridSearchCV
params = {'decisiontreeclassifier__max_depth': [1, 2],
          'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
grid = GridSearchCV(estimator=mv_clf,
                    param_grid=params,
                    cv=10,
                    scoring='roc_auc')
grid.fit(X_train, y_train)
```

После завершения поиска по сетке параметров можно распечатать различные комбинации значений гиперпараметров и средние оценки ROC AUC, вычисленные в результате 10-блочной перекрестной проверки. Соответствующий исходный код выглядит следующим образом:

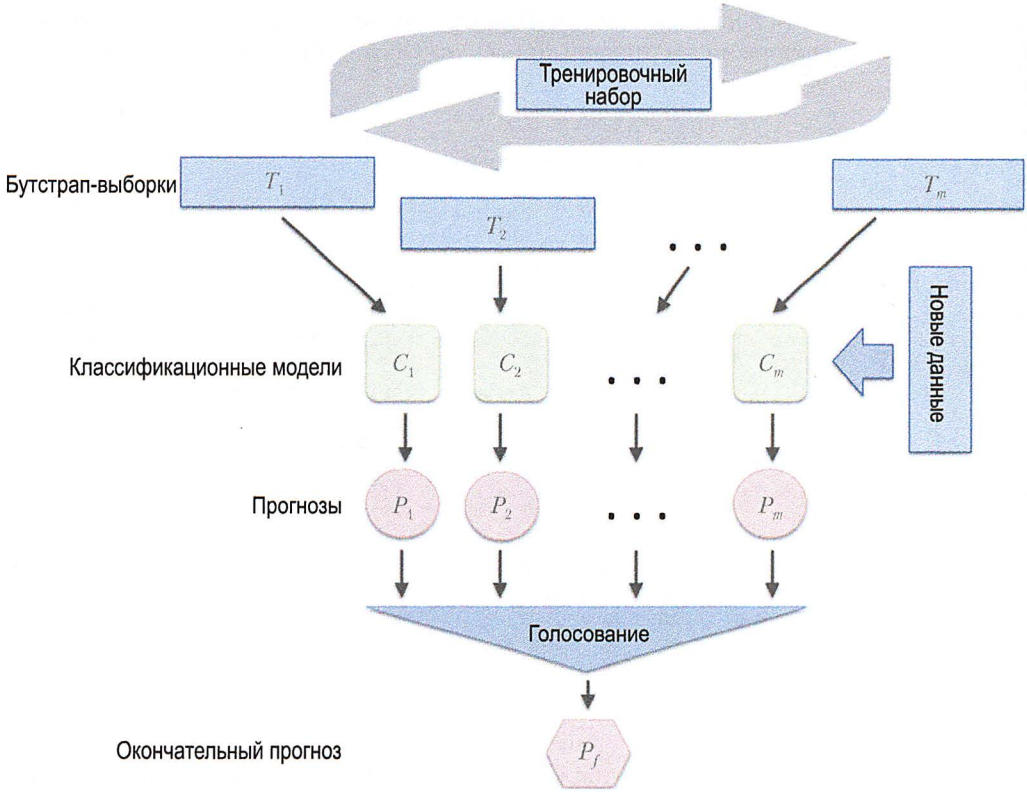
```
>>>
for params, mean_score, scores in grid.grid_scores_:
    print("%0.3f+/-%0.2f %r"
          % (mean_score, scores.std() / 2, params))
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 1}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 1}
0.967+/-0.05 {'pipeline-1__clf__C': 0.001, 'decisiontreeclassifier__
max_depth': 2}
0.967+/-0.05 {'pipeline-1__clf__C': 0.1, 'decisiontreeclassifier__max_
depth': 2}
1.000+/-0.00 {'pipeline-1__clf__C': 100.0, 'decisiontreeclassifier__
max_depth': 2}

print('Лучшие параметры: %s' % grid.best_params_)
Лучшие параметры: {‹pipeline-1__clf__C': 100.0,
'decisiontreeclassifier__max_depth': 1}

print('Верность: %.2f' % grid.best_score_)
Верность: 1.00
```

Как видно, мы получаем наилучшие результаты перекрестной проверки, когда выбираем более низкую силу регуляризации (C = 100.0), тогда как глубина дерева, похоже, вообще не влияет на качество, предполагая, что одноуровневого дерева (пенька решения) для разделения данных достаточно. Учитывая, что использование тестового набора данных более одного раза для оценки модели является порочной практикой, в этом разделе мы не будем оценивать обобщающую способность настроенных гиперпараметров. И быстро перейдем к рассмотрению альтернативного подхода к ансамблевому обучению: **бэггингу**.

Подход на основе мажоритарного голосования, который мы реализовали в этом разделе, иногда также упоминается как **многоярусное обобщение** (stacking). Однако алгоритм многоярусного обобщения чаще используется в сочетании с логистической регрессионной моделью, которая прогнозирует окончательную метку класса, используя в качестве входа прогнозы индивидуальных классификаторов ансамбля, как более подробно описано Дэвидом Х. Уолпертом в: D. H. Wolpert, «Stacked generalization», Neural networks, 5(2):241-259, 1992 (*«Многоярусное обобщение. Нейронные сети»*).

Бэггинг – сборка ансамбля классификаторов из бутстрап-выборок

Бэггинг – это метод ансамблевого обучения, тесно связанный с мажоритарным классификатором MajorityVoteClassifier, который мы реализовали в предыдущем разделе, как проиллюстрировано на следующей ниже схеме:



Однако, вместо того чтобы для подгонки индивидуальных классификаторов в ансамбле использовать один и тот же тренировочный набор, мы извлекаем из исходного тренировочного набора бутстрап-выборки (случайные образцы с возвратом), отсюда и название термина «бэггинг» (bagging), т. е. bootstrap aggregating, **агрегирование бутстрап-выборок**. В качестве более конкретного примера того, как работает бутстрапирование[1], рассмотрим пример, приведенный на нижеследующем рисунке. Здесь имеются семь разных тренировочных экземпляров (обозначены как индексы 1–7), которые в каждом раунде бэггинга, т. е. аггрегирования бутстрап-выборок,

[1] Бутстрапирование (bootstrapping) – это, по существу, метод размножения выборок, когда из выборки из изучаемой генеральной совокупности генерируют новые случайные выборки размером n точек с возвратом, а затем, исходя из этих синтетических наборов данных, вычисляют статистики (среднее, медиану и др.), позволяющие сделать вывод обо всей генеральной совокупности в целом. – *Прим. перев.*

случайным образом отбираются с возвратом. Каждая бутстрап-выборка затем используется для подгонки классификатора C_j, которым, как правило, является неподрезанное дерево решений:

Индексы выборок	Раунд 1 бэггинга	Раунд 2 бэггинга	...
1	2	7	...
2	2	3	...
3	1	2	...
4	3	1	...
5	7	1	...
6	2	7	...
7	4	7	...
	C_1	C_2	C_m

Бэггинг также связан с классификатором на основе случайного леса, который мы представили в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*. Фактически случайные леса являются частным случаем бэггинга, где мы также используем случайные подмножества признаков для подгонки индивидуальных деревьев решений. Бэггинг был впервые предложен Лео Брейманом в техническом отчете за 1994 г.; он также показал, что бэггинг может улучшить оценку верности нестабильных моделей и уменьшить степень переобучения. Настоятельно рекомендуем прочитать его исследование в: L. Breiman, «Bagging Predictors», Machine Learning, 24(2):123-140, 1996 (*«Бэггинг предикторов»*), – которое находится в свободном доступе онлайн, чтобы поближе познакомиться с бэггингом.

Чтобы увидеть бэггинг в действии, создадим более сложную задачу классификации, воспользовавшись набором данных сортов вин, который мы представили в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»*. Здесь мы рассмотрим только классы вин 2 и 3 и отберем два признака: **алкоголь** и **оттенок**.

```
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machinelearning-databases/wine/wine.data'
df_wine = pd.read_csv(url, header=None)
df_wine.columns = ['Метка класса', 'Алкоголь',
                   'Яблочная кислота', 'Зола',
                   ‹Щелочность золы›, ‹Магний›,
                   ‹Всего фенола›, ‹Флаваноиды›,
                   ‹Фенолы нефлаваноидные›, ‹Проантоцианины›,
                   ‹Интенсивность цвета›, ‹Оттенок›,
                   'OD280_OD315 разбавленных вин', 'Пролин']

df_wine = df_wine[df_wine['Метка класса'] != 1]
y = df_wine['Метка класса'].values
X = df_wine[['Алкоголь', 'Оттенок']].values
```

Затем переведем метки классов в двоичный формат и разделим набор данных соответственно на тренировочный (60%) и тестовый (40%) наборы:

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = \
          train_test_split(X, y,
                           test_size=0.40,
                           random_state=1)
```

В библиотеке scikit-learn алгоритм бэггинг-классификатора BaggingClassifier уже реализован, и мы его можем импортировать из подмодуля ensemble. Здесь в качестве базового классификатора мы будем использовать неподрезанное дерево решений и создадим ансамбль из 500 деревьев решений, подогнанных на разных бутстрап-выборках из тренировочного набора данных:

```
from sklearn.ensemble import BaggingClassifier
tree = DecisionTreeClassifier(criterion='entropy',
                              max_depth=None,
                              random_state=1)
bag = BaggingClassifier(base_estimator=tree,
                        n_estimators=500,
                        max_samples=1.0,
                        max_features=1.0,
                        bootstrap=True,
                        bootstrap_features=False,
                        n_jobs=1,
                        random_state=1)
```

Затем вычислим оценку верности прогноза на тренировочном и тестовом наборах данных, чтобы сравнить качество бэггинг-классификатора с качеством одиночного неподрезанного дерева решений:

```
>>>
from sklearn.metrics import accuracy_score
tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)
tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print('Верность дерева решений на тренировочном/тестовом наборах %.3f/%.3f'
       % (tree_train, tree_test))
Верность дерева решений на тренировочном/тестовом наборах 1.000/0.833
```

Исходя из значений верности, которые мы распечатали, выполнив приведенный выше фрагмент исходного кода, неподрезанное дерево решений правильно идентифицирует все метки классов тренировочных образцов; однако существенно более низкая оценка верности на тестовом наборе указывает на высокую дисперсию (переобучение) модели:

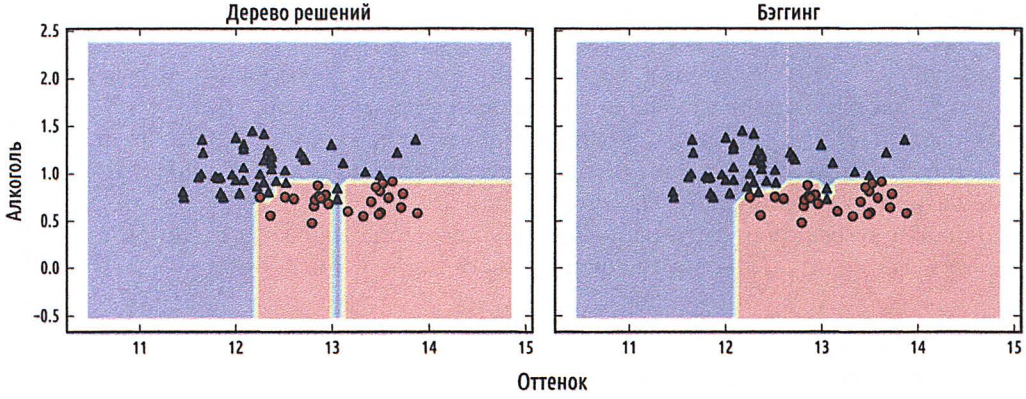
```
bag = bag.fit(X_train, y_train)
y_train_pred = bag.predict(X_train)
y_test_pred = bag.predict(X_test)
bag_train = accuracy_score(y_train, y_train_pred)
bag_test = accuracy_score(y_test, y_test_pred)
print('Верность бэггинга на тренировочном/тестовом наборах %.3f/%.3f'
```

```
        % (bag_train, bag_test))
Верность бэггинга на тренировочном/тестовом наборах 1.000/0.896
```

Несмотря на то что оценки верности классификатора на основе дерева решений и беггинга аналогичны на тренировочном наборе (обе 1.0), мы видим, что согласно оценке на тестовом наборе у бэггинг-классификатора обобщающая способность слегка получше. Далее сравним области решений дерева решений и бэггинг-классификатора:

```
x_min = X_train[:, 0].min() - 1
x_max = X_train[:, 0].max() + 1
y_min = X_train[:, 1].min() - 1
y_max = X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
f, axarr = plt.subplots(nrows=1, ncols=2,
                        sharex='col',
                        sharey='row',
                        figsize=(8, 3))
for idx, clf, tt in zip([0, 1],
                        [tree, bag],
                        [‹Дерево решений›, ‹Бэггинг›]):
    clf.fit(X_train, y_train)

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx].scatter(X_train[y_train==0, 0],
                       X_train[y_train==0, 1],
                       c='blue', marker='^')
    axarr[idx].scatter(X_train[y_train==1, 0],
                       X_train[y_train==1, 1],
                       c='red', marker='o')
    axarr[idx].set_title(tt)
axarr[0].set_ylabel('Алкоголь', fontsize=12)
plt.text(10.2, -1.2,
         s='Оттенок',
         ha='center', va='center', fontsize=12)
plt.show()
```

Как видно на итоговом графике, кусочно-линейная граница решения дерева решений глубиной в три узла выглядит более гладко в ансамбле на основе бэггинга:



В этом разделе мы рассмотрели лишь очень простой пример бэггинга. На практике более сложные задачи классификации и высокая размерность наборов данных могут легко привести к переобучению в одиночных деревьях решений, и именно здесь алгоритм бэггинга может по-настоящему проявить свою силу. В завершение отметим, что алгоритм бэггинга может оказаться эффективным подходом для снижения дисперсии модели. Однако бэггинг неэффективен для снижения смещения модели, и, следовательно, необходимо выбирать ансамбль классификаторов с низким смещением, например неподрезанные деревья решений.

Усиление слабых учеников методом адаптивного бустинга

В этом разделе об ансамблевых методах мы обсудим **бустинг** (boosting), или усиление, форсирование, с особым акцентом на его наиболее распространенной реализации, **AdaBoost** (сокращение для adaptive boosting, т. е. **адаптивное усиление**).

Лежащая в основе AdaBoost оригинальная идея была сформулирована Робертом Шапире в 1990 г. (R. E. Schapire, «The Strength of Weak Learnability», Machine learning, 5(2):197-227, 1990 («*Сильные стороны слабой обучаемости*»)). После того как Р. Шапире и Й. Фройнд представили алгоритм AdaBoost в Сборнике Тринадцатой международной конференции (ICML 1996), в последующие годы AdaBoost стал одним из наиболее широко используемых ансамблевых методов (Y. Freund, R. E. Schapire et al., «Experiments with a New Boosting Algorithm». In ICML, vol. 96, p. 148–156, 1996 («*Эксперименты с новым алгоритмом усиления*»)). В 2003 г. Фреунд и Шапайр за свою инновационную работу получили премию Геделя, престижную премию за самые выдающиеся публикации в области информатики.

В бустинге ансамбль состоит из очень простых базовых классификаторов, нередко также именуемых **слабыми учениками**, имеющих лишь небольшое преимущество в качестве над случайным гаданием. Типичным примером слабого ученика является одноуровневое дерево решений (пенек решения). Лежащая в основе бустинга ключевая идея состоит в том, что он сосредоточен на тренировочных образцах, которые трудно классифицировать[1], т. е. с целью улучшения качества ансамбля дает слабым ученикам впоследствии обучиться на ошибочно классифицированных тренировочных образцах. В отличие от бэггинга, алгоритм бустинга в своей исходной формулировке использует случайные подмножества тренировочных образцов, извлеченные из тренировочного набора данных без возврата. Исходная процедура бустинга резюмирована в четырех ключевых шагах следующим образом:

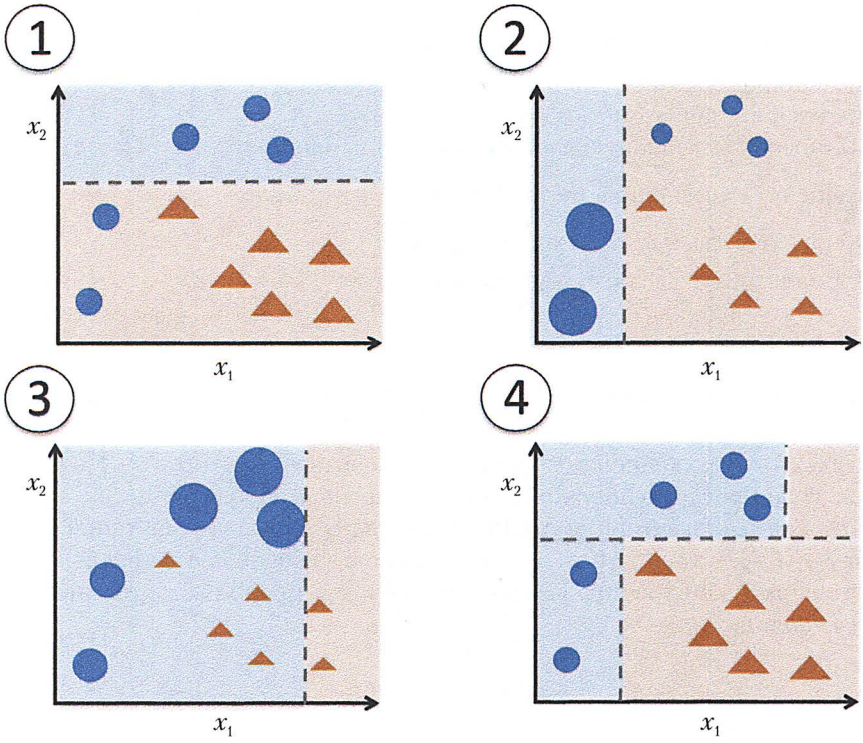
1) извлечь случайное подмножество тренировочных образцов d_1 без возврата из тренировочного набора D для тренировки слабого ученика C_1;
2) извлечь второе случайное тренировочное подмножество d_2 без возврата из тренировочного набора и добавить 50% ранее ошибочно классифицированных образцов для тренировки слабого ученика C_2;
3) найти в тренировочном наборе D тренировочные образцы d_3, по которым C_1 и C_2 расходятся, для тренировки третьего слабого ученика C_3;

[1] В этом и заключается адаптивность алгоритма – в процессе обучения он подстраивается под наиболее «сложные» случаи. – *Прим. перев.*

4) объединить слабых учеников C_1, C_2 и C_3 посредством мажоритарного голосования.

Как указывалось Лео Брейманом (L. Breiman, «Bias, Variance and Arcing Classifiers», 1996 (*«Смещение, дисперсия и ARC-классификаторы с адаптивным ресэмплингом и комбинированием»*)), бустинг может привести к уменьшению смещения и дисперсии, по сравнению с моделями на основе бэггинга. На практике, однако, алгоритмы бустинга, такие как AdaBoost, также известны своей высокой дисперсией, т. е. тенденцией к переобучению на тестовых данных (G. Raetsch, T. Onoda and K. R. Mueller, «An Improvement of Adaboost to Avoid Overfitting». In Proc. of the Int. Conf. on Neural Information Processing. Citeseer, 1998 (*«Улучшение алгоритма Adaboost для недопущения переобучения»*)).

В отличие от описанной здесь исходной процедуры бустинга, алгоритм AdaBoost для тренировки слабых учеников использует полный тренировочный набор, где тренировочные образцы взвешиваются повторно в каждой итерации с целью построения сильного классификатора, который обучается на ошибках предыдущих слабых учеников в ансамбле. Прежде чем мы глубже погрузимся в конкретные подробности алгоритма AdaBoost, взглянем на следующий ниже рисунок, чтобы лучше уловить суть ключевой идеи, лежащей в основе алгоритма AdaBoost:



Анализируя иллюстрацию алгоритма AdaBoost в пошаговом режиме, начнем с подрисунка **1**, представляющего тренировочный набор для двуклассовой (бинарной) классификации, где всем тренировочным образцам назначены равные веса. На основе этого тренировочного набора мы тренируем пенек решения (показанный

как пунктирная линия), который пытается классифицировать образцы из этих двух классов (треугольники и круги), насколько возможно, путем минимизации функции стоимости (либо оценки неоднородности в частном случае ансамблей деревьев решений). В следующем раунде (подрисунок **2**) мы назначаем больший вес двум ранее ошибочно классифицированным образцам (круги). Кроме того, мы понижаем вес правильно классифицированных образцов. Следующий пенек решения теперь будет более сфокусирован на тренировочных образцах с самыми большими весами, т. е. тренировочных образцах, которые предположительно трудно классифицировать. Слабый ученик, показанный в подрисунке **2**, ошибочно классифицирует три разных образца из класса с кругами, которым затем присваивается больший вес, как показано на подрисунке **3**. Учитывая, что наш ансамбль AdaBoost состоит всего из трех раундов бустинга, мы затем объединяем трех слабых учеников, натренированных на разных повторно взвешенных тренировочных подмножествах, путем взвешенного мажоритарного голосования, как показано на подрисунке **4**.

Имея более глубокое понимание идеи, лежащей в основе алгоритма AdaBoost, теперь проанализируем алгоритм более детально при помощи псевдокода. Для ясности обозначим поэлементное умножение крестиком ($\times$) и соответственно скалярное произведение между двумя векторами – точкой ($\cdot$). Шаги следующие:

1. Назначить весовому вектору $\boldsymbol{w}$ равномерные веса, где $\sum_i \boldsymbol{w}_i = 1$.
2. Для j в m раундах бустинга сделать следующее.
3. Натренировать взвешенного слабого ученика: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{x})$.
4. Идентифицировать метки классов: $\hat{\boldsymbol{y}} = \text{predict}(C_j, \boldsymbol{X})$.
5. Вычислить взвешенную частоту появления ошибок: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq y)$.
6. Вычислить коэффициент: $a_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
7. Обновить веса: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-a_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
8. Нормализовать веса к сумме, равной 1: $\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i \boldsymbol{w}_i}$.
9. Вычислить окончательный прогноз: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} (\boldsymbol{a}_j \times \text{predict}(C_j, \boldsymbol{X})) > 0 \right)$.

Отметим, что выражение $(\hat{\boldsymbol{y}} == \boldsymbol{y})$ в шаге 5 обозначает вектор из 1 и 0, где 1 назначается, если прогноз ошибочный, и 0 – в противном случае.

Несмотря на то что алгоритм AdaBoost выглядит довольно прямолинейным, проанализируем более конкретный пример, воспользовавшись тренировочным набором, состоящим из 10 тренировочных образцов, как проиллюстрировано в следующей ниже таблице:

Индексы выборок	x	y	Веса	$\hat{y}(x <= 3.0)$?	Правильно?	Обновленные веса
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

Первый столбец таблицы показывает индексы 1–10 образцов тренировочного набора. Во втором столбце мы видим значения признаков индивидуальных образцов, принимая, что это одномерный набор данных. Третий столбец показывает истинную метку класса y_i для каждого тренировочного образца x_i, где $y_i \in \{1, -1\}$. Исходные веса показаны в четвертом столбце; мы инициализируем веса однородно и нормализуем к сумме, равной 1. Поэтому в случае тренировочного набора из 10 образцов мы назначаем 0.1 каждому весу w_i в весовом векторе $\boldsymbol{w}$. Идентифицированные метки классов $\hat{\boldsymbol{y}}$ показаны в пятом столбце, принимая, что наш критерий разделения равен $x \leq 3.0$. Последний столбец таблицы показывает веса, обновленные на основе правил обновления, которые мы определили в псевдокоде.

Учитывая, что вычисление обновлений весов может поначалу выглядеть немного сложным, теперь проследим за их вычислением в пошаговом режиме. Начнем с вычисления взвешенной частоты появления ошибок ε, как описано в шаге 5:

$$\varepsilon = 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 +$$
$$+ 0.1\times 0 + 0.1\times 1 + 0.1\times 1 + 0.1\times 1 + 0.1\times 0 = \frac{3}{10} = 0.3.$$

Затем вычислим коэффициент a_j (показанный в шаге 6), который позже используется в шаге 7 для обновления весов, а также для весов в прогнозе на основе мажоритарного голосования (шаг 10):

$$a_j = 0.5\log\frac{1-\varepsilon}{\varepsilon} \approx 0.424.$$

После того как мы вычислили коэффициент a_j, теперь можно обновить весовой вектор, используя для этого следующее равенство:

$$\boldsymbol{w} := \boldsymbol{w} \times \exp(-a_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}).$$

Здесь $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ – это поэлементное умножение соответственно между векторами спрогнозированных и истинных меток классов. Поэтому если прогноз $\hat{\boldsymbol{y}}$ будет правильным, то $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ будет иметь положительный знак, в результате чего мы уменьшим i-й вес, поскольку a_j – тоже положительное число:

$$0.1 \times \exp(-0.424 \times 1 \times 1) \approx 0.065.$$

Аналогичным образом мы увеличим i-й вес, в случае если $\hat{\boldsymbol{y}}$ спрогнозировал метку ошибочно, как здесь:

$$0.1 \times \exp(-0.424 \times 1 \times (-1)) \approx 0.153.$$

Или как здесь:

$$0.1 \times \exp(-0.424 \times (-1) \times (1)) \approx 0.153.$$

После обновления каждого веса в весовом векторе мы нормализуем веса так, чтобы они в сумме давали 1 (шаг 8):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i \boldsymbol{w}_i}.$$

Здесь $\sum_i \boldsymbol{w}_i = 7 \times 0.065 + 3 \times 0.153 = 0.914.$

Таким образом, каждый вес, который соответствует правильно классифицированному образцу, будет уменьшен с исходного значения 0.1 до $0.065/0.914 \approx 0.071$ для следующего раунда бустинга. Точно так же веса каждого ошибочно классифицированного образца увеличатся с 0.1 до $0.153/0.914 \approx 0.167$.

Это был алгоритм AdaBoost вкратце. Переходя к более практической части, теперь натренируем ансамблевый классификатор AdaBoost посредством библиотеки scikit-learn. Мы воспользуемся тем же самым подмножеством сортов вин, которое мы использовали ранее в предыдущем разделе для тренировки метаклассификатора на основе бэггинга. Используя атрибут base_estimator, натренируем классификатор AdaBoostClassifier на 500 пеньках деревьев решений:

```
>>>
from sklearn.ensemble import AdaBoostClassifier
tree = DecisionTreeClassifier(criterion='entropy',
                              max_depth=1,
                              random_state=0)
ada = AdaBoostClassifier(base_estimator=tree,
                         n_estimators=500,
                         learning_rate=0.1,
                         random_state=0)
tree = tree.fit(X_train, y_train)
y_train_pred = tree.predict(X_train)
y_test_pred = tree.predict(X_test)
tree_train = accuracy_score(y_train, y_train_pred)
tree_test = accuracy_score(y_test, y_test_pred)
print('Верность дерева решений на тренировочном/тестовом наборах %.3f/%.3f'
      % (tree_train, tree_test))
Верность дерева решений на тренировочном/тестовом наборах 0.845/0.854
```

Как видно, пенек дерева решений показывает тенденцию к недообучению под тренировочные данные, в отличие от неподрезанного дерева решений, которое мы видели в предыдущем разделе:

```
>>>
ada = ada.fit(X_train, y_train)
y_train_pred = ada.predict(X_train)
y_test_pred = ada.predict(X_test)
ada_train = accuracy_score(y_train, y_train_pred)
ada_test = accuracy_score(y_test, y_test_pred)
print('Верность AdaBoost на тренировочном/тестовом наборах %.3f/%.3f'
      % (ada_train, ada_test))
Верность AdaBoost на тренировочном/тестовом наборах 1.000/0.875
```

Как видно, модель AdaBoost правильно идентифицирует все метки классов тренировочного набора и также показывает слегка улучшенное качество на тестовом наборе, по сравнению с пеньком дерева решений. Однако мы также видим, что вместе с нашей попыткой уменьшить смещение модели мы привнесли дополнительную дисперсию.

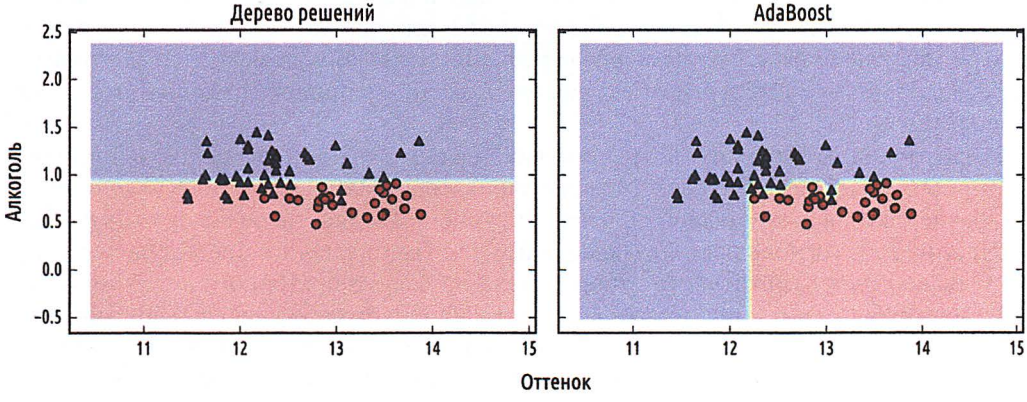
Несмотря на то что в демонстрационных целях мы использовали еще один простой пример, мы видим, что качество классификатора AdaBoost немного улучшилась, по сравнению с пеньком решения, и достигла оценок верности, очень схожих с классификатором на основе бэггинга, который мы натренировали в предыдущем разделе. Однако стоит отметить, что отбор модели на основе повторного использования тестового набора считается порочной практикой. Оценка обобщающей способности может быть слишком оптимистичной, что мы обсудили более подробно в *главе 6* «*Анализ наиболее успешных приемов оценивания моделей и тонкой настройки гиперпараметров*».

Наконец, проверим, как выглядят области решений:

```
x_min = X_train[:, 0].min() - 1
x_max = X_train[:, 0].max() + 1
y_min = X_train[:, 1].min() - 1
y_max = X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
f, axarr = plt.subplots(1, 2,
                        sharex='col',
                        sharey='row',
                        figsize=(8, 3))
for idx, clf, tt in zip([0, 1],
                        [tree, ada],
                        ['Decision Tree', 'AdaBoost']):
    clf.fit(X_train, y_train)
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
    axarr[idx].scatter(X_train[y_train==0, 0],
                       X_train[y_train==0, 1],
                       c='blue',
                       marker='^')
    axarr[idx].scatter(X_train[y_train==1, 0],
                       X_train[y_train==1, 1],
                       c='red',
                       marker='o')
```

```
    axarr[idx].set_title(tt)
    axarr[0].set_ylabel('Алкоголь', fontsize=12)
plt.text(10.2, -1.2,
         s='Оттенок',
         ha='center',
         va='center',
         fontsize=12)
plt.show()
```

Рассматривая области решений, мы видим, что граница решения модели AdaBoost существенно сложнее границы решения пенька решения. Кроме того, мы отмечаем, что модель AdaBoost разделяет пространство признаков очень похоже с классификатором на основе бэггинга, который мы натренировали в предыдущем разделе.



В заключительных комментариях по поводу ансамблевых методов стоит отметить, что ансамблевое обучение увеличивает вычислительную сложность, по сравнению с индивидуальными классификаторами. На практике следует тщательно подумать, хотим ли мы оплачивать высокую цену в виде увеличенных вычислительных затрат в обмен на часто относительно скромное улучшение предсказательной способности.

Часто цитируемым примером такого компромисса является известный *приз Netflix в 1 миллион долларов*, выигранный при помощи ансамблевых методов. Подробности об алгоритме были раскрыты в работе: A. Toescher, M. Jahrer and R.M. Bell, «The Bigchaos Solution to the Netflix Grand Prize». Netflix prize documentation, 2009 (*«Решение Bigchaos для главного приза Netflix»*), которая доступна по прямой ссылке http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf. Хотя победившая команда получила денежный приз в размере 1 миллион долл., Netflix так и не удалось осуществить их модель из-за ее сложности, которая сделала ее невыполнимой в реальных условиях. Процитируем их точные слова (http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html):

> «[...] дополнительная отдача за счет верности, которую мы измерили, как оказалось, не оправдает программу технических работ, необходимую для ее внедрения в производственную среду».

Резюме

В этой главе мы рассмотрели некоторые самые популярные и широко используемые методы ансамблевого обучения. Ансамблевые методы объединяют различные модели классификации для уравновешивания их индивидуальных слабых сторон, что нередко приводит к стабильным и качественно работающим моделям, которые выглядят очень привлекательными для промышленного применения и конкурсов по машинному обучению.

В начале этой главы мы реализовали на Python ансамблевый мажоритарный классификатор MajorityVoteClassifier, позволяющий объединять разные алгоритмы классификации данных. Затем мы рассмотрели бэггинг, широко применяемый метод снижения дисперсии модели путем извлечения случайных образцов из тренировочного набора и объединения отдельно натренированных классификаторов мажоритарным голосованием. Потом мы обсудили алгоритм AdaBoost, основанный на слабых учениках, которые в дальнейшем обучаются на ошибках.

На протяжении всех предыдущих глав мы обсуждали разные алгоритмы обучения, тонкую настройку их параметров и методы оценки их качества работы. В следующей главе мы рассмотрим отдельно взятое приложение машинного обучения, анализ мнений, которое, несомненно, стало интересной темой в эру Интернета и социальных медиа.

Применение алгоритмов машинного обучения в анализе мнений

Во времена и годы Интернета и социальных медиа мнения, отзывы и рекомендации людей стали ценным ресурсом для политологов и деловых кругов. Благодаря современным технологиям мы теперь в состоянии аккумулировать и анализировать такие данные наиболее эффективным образом. В этой главе мы углубимся в подобласть **обработки естественного языка** (natural language processing, NLP) под названием **анализ мнений**[1] и узнаем, как использовать алгоритмы машинного обучения для классификации документов, основываясь на их направленности: отношении автора. В последующих разделах мы охватим темы, которые включают:

- очистку и подготовку текстовых данных;
- создание признаковых векторов из текстовых документов;
- тренировку машиннообучаемой модели выполнять классификацию положительных и отрицательных киноотзывов;
- работу с большими текстовыми наборами данных с использованием обучения вне ядра.

Получение набора данных киноотзывов IMDb

Анализ мнений, иногда также именуемый **добычей мнений**, – популярная подотрасль более широкой области – обработки естественного языка (NLP), предназначенная для анализа **полярности** документов, т. е. их направленности. Популярной задачей в анализе мнений является классификация документов на основе выраженных мнений или эмоций авторов относительно определенной темы.

В этой главе мы будем работать с большим набором данных киноотзывов из **интернет-базы кинофильмов** (Internet Movie Database, сокращенно IMDb), составленным Маасом и др. (A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng and C. Potts, «Learning Word Vectors for Sentiment Analysis». In the proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, p. 142–150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics («*Извлечение словарных векторов для проведения анализа мнений*»)).

Набор данных киноотзывов состоит из 50 000 полярных отзывов о кинофильмах, помеченных как *положительные* либо как *отрицательные*; в данном случае положи-

[1] Анализ мнений (sentiment analysis, opinion mining), или сентимент-анализ, анализ тональности текста – численный анализ мнений, настроений, субъективности, оценок, отношения, эмоций и т. д., которые выражены в текстовом виде. – *Прим. перев.*

тельный означает, что в IMDb фильм было оценен более шестью звездами, и отрицательный означает, что в IMDb фильм получил менее пяти звезд. В последующих разделах мы узнаем, как выделять содержательную информацию из подмножества этих киноотзывов для построения машиннообучаемой модели, которая способна прогнозировать, понравился определенному рецензенту фильм или нет.

Сжатый архив набора данных киноотзывов (84.1 Мб) можно скачать с http://ai.stanford.edu/~amaas/data/sentiment/ как сжатый архив gzip в архиве tarball:

- если вы работаете в Linux или Mac OS X, то можете открыть новое окно терминала, перейти при помощи команды cd в каталог загрузки download и выполнить команду tar -zxf aclImdb_v1.tar.gz, чтобы разархивировать набор данных;
- если вы работаете в Windows, то для извлечения файлов из скачанного архива вы можете скачать бесплатный архиватор, такой как 7-Zip (http://www.7-zip.org).

После успешного извлечения набора данных теперь соберем отдельные текстовые документы из раскрытого архива в один CSV-файл. В следующем ниже фрагменте исходного кода мы прочитаем киноотзывы в объект библиотеки pandas, таблицу данных DataFrame; на стандартном настольном компьютере этот процесс может занять до 10 минут. Для визуализации прогресса и оценочного времени до завершения работы мы воспользуемся индикатором прогресса, библиотекой Python **PyPrind** (https://pypi.python.org/pypi/PyPrind/), которую я разработал несколько лет назад для таких целей. Библиотеку PyPrind можно установить, выполнив команду pip install pyprind.

```
>>>
import pyprind
import pandas as pd
import os
pbar = pyprind.ProgBar(50000)
labels = {'pos':1, 'neg':0}
df = pd.DataFrame()
for s in ('test', 'train'):
    for l in ('pos', 'neg'):
        path ='./aclImdb/%s/%s' % (s, l)
        for file in os.listdir(path):
            with open(os.path.join(path, file), 'r', encoding='utf-8') as infile:
                txt = infile.read()
            df = df.append([[txt, labels[l]]], ignore_index=True)
            pbar.update()
df.columns = ['review', 'sentiment']
0%                            100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 725.001 sec
```

Выполняя приведенный выше исходный код, мы сначала инициализировали новый объект индикатора выполнения pbar 50 000 итерациями, т. е. числом документов, которые мы собирались прочитать. При помощи вложенных циклов for мы выполнили итерации по подкаталогам train и test внутри основного каталога aclImdb и прочли отдельные текстовые файлы из подкаталогов pos и neg, которые мы в конце добавили в таблицу данных DataFrame, представленную объектной переменной df вместе с целочисленной меткой класса (1 = положительный и 0 = отрицательный).

Поскольку метки классов в собранном наборе данных отсортированы, мы теперь перемешаем DataFrame при помощи функции permutation из подмодуля np.random – это поможет разделить набор данных на тренировочный и тестовый наборы в более поздних разделах главы, когда мы будем передавать поток данных непосредственно с нашего локального диска. Для нашего собственного удобства мы также сохраним собранный и перемешанный набор данных киноотзывов в виде CSV-файла:

```
import numpy as np
np.random.seed(0)
df = df.reindex(np.random.permutation(df.index))
df.to_csv('./movie_data.csv', index=False)
```

Поскольку позднее в этой главе мы собираемся использовать этот набор данных, оперативно убедимся, что мы успешно сохранили данные в правильном формате. Для этого мы прочитаем CSV-файл и напечатаем фрагмент из первых трех образцов:

```
df = pd.read_csv('./movie_data.csv')
df.head(3)
```

Если вы выполняете примеры программ в блокнотах Jupyter, то теперь вы должны увидеть первые три образца из набора данных, как показано в следующей ниже таблице:

	Киноотзыв	Мнение
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Концепция модели мешка слов

Из *главы 4 «Создание хороших тренировочных наборов – предобработка данных»* мы помним, что, прежде чем мы сможем передать категориальные данные, такие как текст или слова, на вход алгоритма машинного обучения, нам нужно их преобразовать в числовую форму. В этом разделе мы представим модель **мешка слов**[1], позволяющую представлять текст как числовые векторы признаков. В основе модели мешка слов лежит довольно простая идея, которую можно резюмировать следующим образом:

1) создать из всего набора документов **вокабуляр** уникальных **лексем** (tokens) – например, слов;
2) построить из каждого документа вектор признаков, содержащий частотности вхождений каждого слова в определенный документ.

Учитывая, что уникальные слова в каждом документе представляют в вокабуляре модели мешка слов лишь малое подмножество всех слов, векторы признаков будут состоять главным образом из нулей, и поэтому мы называем их **разреженными**. Не

[1] Модель «мешок слов» (bag-of-words) – это упрощение, применяемое в области обработки ЕЯ. Текст представляется как неупорядоченная коллекция слов без учета грамматических правил и порядка слов в предложениях. – *Прим. перев.*

переживайте, если это представляеся слишком абстрактным; в следующих подразделах мы проанализируем процесс создания простой модели мешка слов в пошаговом режиме.

Преобразование слов в векторы признаков

Чтобы построить модель мешка слов на основе частотностей слов в соответствующих документах, можно воспользоваться классом векторизации количеств CountVectorizer, реализованным в библиотеке scikit-learn. Как мы видим в следующем ниже фрагменте исходного кода, класс CountVectorizer принимает массив текстовых данных, которые могут быть документами или просто предложениями, и строит для нас модель мешка слов:

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
count = CountVectorizer()
docs = np.array([
            'The sun is shining',
            'The weather is sweet',
            'The sun is shining and the weather is sweet, and one and one is two'])
bag = count.fit_transform(docs)
```

Вызвав метод fit_transform в объекте CountVectorizer, мы только что создали вокабуляр модели мешка слов и преобразовали следующие три предложения в разряженные векторы признаков:

1. The sun is shining (Солнце светит).
2. The weather is sweet (Погодка сладкая).
3. The Sun is shining and the weather is sweet (Солнце светит, погодка сладкая, и один плюс один равно два).

Теперь распечатаем содержимое вокабуляра, чтобы получше разобраться в основных принципах работы модели:

```
>>>
print(count.vocabulary_)
{'one': 2, 'the': 6, 'sweet': 5, 'and': 0, 'sun': 4, 'is': 1, 'shining': 3, 'two': 7,
'weather': 8}
```

Как видно из результата выполнения предыдущей команды, вокабуляр модели хранится в виде словаря Python, где уникальным словам поставлены в соответствие целочисленные индексы. Далее распечатаем векторы признаков, которые мы только что создали:

```
>>>
print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [3 3 2 1 1 1 2 1 1]]
```

Каждая индексная позиция в показанных тут векторах признаков соответствует целочисленным значениям, которые хранятся как элементы вокабуляра в CountVectorizer. Например, первый признак в индексной позиции 0 указывает на количество вхождений слова and, которое появляется лишь в последнем документе, а в индекс-

ной позиции 1 находится слово is (2-ой признак в векторах документов), которое появляется во всех трех предложениях. Значения в векторах признаков также называются **исходными частотами терминов** (raw term frequencies): $tf(t, d)$ – количество раз термин t появляется в документе d.

Последовательность элементов в модели мешка слов, которую мы только что создали, также называется **одноградмной**, или **униграммной**, моделью – каждый элемент или лексема в вокабуляре представляет одно слово. В более широком смысле в обработке ЕЯ непрерывные последовательности элементов – слова, буквы или символы – называются также ***n*-граммами**. Выбор числа n в n-граммной модели зависит от отдельно взятого приложения; например, исследование Канариса и др. показало, что n-граммы размером 3 и 4 дают хорошее качество антиспамной фильтрации электронных писем (Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas and Efstathios Stamatatos, «Words vs Character N-Grams for Anti-Spam Filtering». International Journal on Artificial Intelligence Tools, 16(06):1047-1067, 2007 (*«Слова и знаки в n-граммах для антиспамной фильтрации»*)). Резюмируя концепцию n-граммного представления, униграммное (одноградмное) и биграммное (двуграммное) представления нашего первого документа «the sun is shining» были бы сконструированы следующим образом:

- **одноградмное**: “the”, “sun”, “is”, “shining”;
- **двуграммное**: “the sun”, “sun is”, “is shining”.

Класс векторизации частотностей CountVectorizer в библиотеке scikit-learn при помощи своего параметра ngram_range позволяет использовать разные n-граммные модели. Хотя одноградмное представление используется по умолчанию, мы можем переключиться на двуграммное представление путем инициализации нового экземпляра класса CountVectorizer параметром ngram_range = (2,2).

Оценка релевантности слова методом tf-idf

Когда мы анализируем текстовые данные, мы часто сталкиваемся со словами из обоих классов, которые встречаются в двух и более документах. Такие часто встречающиеся слова, как правило, не содержат полезной или отличительной информации. В этом подразделе мы узнаем о широко используемом методе под названием **tf-idf** (term frequency – inverse document frequency), или **частота термина**, – **обратная частота документа**, который может использоваться для понижающего взвешивания этих часто встречающихся слов в векторах признаков. Коэффициент tf-idf определяется как произведение **частоты термина** и **обратной частоты документа**:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t, d).$$

Здесь $\text{tf}(t, d)$ – это частота термина, которую мы представили в предыдущем разделе, при этом обратную частоту документа $\text{idf}(t, d)$ можно вычислить как

$$\text{idf}(t,d) = \log \frac{n_d}{1 + \text{df}(d,t)},$$

где n_d – это общее число документов и $\text{df}(d, t)$ – число документов d, которые содержат термин t. Отметим, что добавление константы 1 в знаменатель является факультативным и служит для назначения ненулевого значения терминам, которые встречаются во всех тренировочных образцах; логарифм используется для того, чтобы низкие частоты документов гарантированно не получали слишком большого веса.

В библиотеке scikit-learn реализован еще один класс-преобразователь, преобразователь коэффициентов tf-idf TfidfTransformer, который в качестве входных данных принимает из векторизатора частотностей CountVectorizer исходные частоты терминов и преобразует их в серию tf-idf'ов:

```
>>>
from sklearn.feature_extraction.text import TfidfTransformer
tfidf = TfidfTransformer()
np.set_printoptions(precision=2)
print(tfidf.fit_transform(count.fit_transform(docs)).toarray())
[[ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.  ]
 [ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
 [ 0.66  0.39  0.44  0.17  0.17  0.17  0.26  0.22  0.17]]
```

Как мы увидели в предыдущем подразделе, слово is имело наибольшую частоту термина в 3-м документе, будучи наиболее часто встречающимся словом. Однако после преобразования того же вектора признаков в tf-idf'ы мы видим, что в документе 3 слову is теперь поставлен в соответствие относительно малый tf-idf (0.39), поскольку оно также содержится в документах 1 и 2 и поэтому вряд ли будет содержать какую-то полезную, отличительную информацию.

Однако если бы в наших векторах признаков мы вычислили tf-idf'ы отдельных терминов вручную, то заметили бы, что класс TfidfTransformer вычисляет tf-idf'ы немного по-другому, по сравнению со *стандартными* уравнениями из учебника, которые мы определили ранее. В библиотеке scikit-learn для idf и tf-idf реализованы несколько другие уравнения. Уравнение idf следующее:

$$\text{idf}(t,d) = \log\frac{1+n_d}{1+\text{df}(d,t)}.$$

Уравнение tf-idf в scikit-learn реализовано следующим образом:

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times (\text{idf}(t, d) - 1).$$

Хотя более типичный сценарий перед вычислением коэффициентов tf-idf также состоит в нормализации исходных частот терминов, класс TfidfTransformer непосредственно сам нормализует tf-idf'ы. По умолчанию (norm='l2') класс TfidfTransformer библиотеки scikit-learn применяет L2-нормализацию путем деления ненормализованного вектора признаков v на его L2-норму, в результате чего возвращается вектор длиной 1:

$$v_{\text{norm}} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \ldots + v_n^2}} = \frac{v}{\left(\sum_{i=1}^{n} v_i^2\right)^{1/2}}.$$

Чтобы удостовериться, что мы понимаем, как работает класс TfidfTransformer, разберем пример детально, вычислив коэффициент tf-idf слова is в 3-м документе.

Слово is в документе 3 имеет частоту термина, равную 3 (tf = 3), а частота документа этого термина равна 3, поскольку термин is встречается во всех трех документах (df = 3). Таким образом, мы можем вычислить idf следующим образом:

$$\text{idf}("is", d3) = \log\frac{1+3}{1+3} = 0.$$

Теперь, чтобы вычислить коэффициент tf-idf, нам просто нужно прибавить 1 к обратной частоте документа и умножить ее на частоту термина:

$$\text{tf-idf}(is, d3) = 3 \times (0 + 1) = 3.$$

Если бы мы повторили эти вычисления для всех терминов в 3-м документе, то получили бы следующие векторы tf-idf: [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69 и 1.29]. Однако мы замечаем, что значения в этом векторе признаков отличаются от значений, которые мы получили из примененного нами ранее класса TfidfTransformer. В этой калькуляции tf-idf нам недостает завершающего шага, которым является L2-нормализация и которую можно применить следующим образом:

$$\text{tf-idf}(is, d3) = 0.39.$$

Как видно, теперь наши результаты соответствуют результатам на выходе из класса TfidfTransformer библиотеки scikit-learn. Поскольку теперь мы понимаем, как вычисляются tf-idf'ы, перейдем к следующим разделам и применим эти принципы к набору данных киноотзывов.

Очистка текстовых данных

В предыдущих подразделах мы узнали о модели мешка слов, частотах терминов и коэффициентах tf-idf. Однако первым важным шагом – прежде чем мы построим нашу модель мешка слов – является очистка текстовых данных путем удаления из них всех нежелательных символов. В качестве иллюстрации того, почему это важно сделать, покажем последние 50 символов из первого документа в перемешанном наборе данных киноотзывов:

```
>>>
df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

Как здесь видно, текст содержит разметку HTML, а также пунктуацию и другие небуквенные символы. В то время как разметка HTML не несет какой-то большой полезной семантики, знаки препинания в определенных контекстах обработки ЕЯ (NLP) могут представлять полезную дополнительную информацию. Однако для простоты мы теперь удалим все знаки препинания, сохранив только символы эмоций, или **эмограммы**, такие как «:)», поскольку они определенно пригодятся для анализа мнений. Для выполнения этой задачи мы воспользуемся библиотекой **регулярных выражений** Python re, как показано ниже:

```
import re
def preprocessor(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)', text)
    text = re.sub('[\W]+', ' ', text.lower()) + \
                  ' '.join(emoticons).replace('-', '')
    return text
```

Первым регулярным выражением <[^>]*> в приведенном выше фрагменте исходного кода мы пытались удалить всю содержавшуюся в киноотзывах разметку HTML. Хотя многие программисты обычно отговаривают от использования ре-

гулярных выражений для разбора HTML, приведенного регулярного выражения должно быть достаточно, чтобы *очистить* этот отдельно взятый набор данных. После того как мы удалили разметку HTML, мы воспользовались регулярным выражением чуть посложнее, чтобы отыскать эмограммы, которые мы временно сохранили как emoticons. Затем регулярным выражением [\W]+ мы удалили из текста все несловарные символы, преобразовали текст в строчные буквы и в заключение добавили временно сохраненные в emoticons эмограммы в конец обработанной последовательности символов документа. Кроме того, в целях единообразия мы удалили из эмограмм символ *носа* (-).

Регулярные выражения предлагают эффективный и удобный подход к поиску символов в строковой последовательности, вместе с тем они также сопровождаются крутой кривой обучаемости. К сожалению, всестороннее обсуждение регулярных выражений выходит за рамки этой книги. Однако на веб-странице https://developers.google.com/edu/python/regular-expressions портала Google Developers вы можете найти замечательное пособие. Кроме того, можно обратиться к официальной документации по библиотеке Python на https://docs.python.org/3.4/library/re.html.

Несмотря на то что добавление эмограмм в конец очищенных строковых последовательностей документов, возможно, не выглядит как самый изящный подход, порядок слов в нашей модели мешка слов, с учетом того, что наш вокабуляр состоит лишь из однословных лексем, не имеет значения. Но прежде чем мы продолжим разговор о разбивке документов на отдельные термины, слова или лексемы, проверим правильность работы нашего препроцессора:

```
>>>
preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Наконец, учитывая, что в последующих разделах мы неоднократно будем пользоваться *очищенными* текстовыми данными, теперь применим нашу функцию preprocessor ко всем киноотзывам в нашей таблице данных DataFrame:

```
df['review'] = df['review'].apply(preprocessor)
```

Переработка документов в лексемы

Успешно подготовив набор данных с киноотзывами, теперь следует подумать о том, каким образом разделить текстовый корпус на отдельные элементы. Один из способов *разбивки* документов *на лексемы* состоит в том, чтобы разделить их на отдельные слова путем разбиения очищенного документа по пробельным символам:

```
>>>
def tokenizer(text):
    return text.split()
tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

В контексте лексемизации другой полезный метод состоит в **выделении основы слова** (word stemming), или стемминге, т. е. процессе редукции слова к его основе, что позволяет проецировать на одну и ту же основу связанные с ней слова. Ориги-

нальный алгоритм стемминга был разработан Мартином Портером в 1979 г. и отсюда получил название **стеммер Портера** (Martin F. Porter, «An algorithm for suffix stripping». Program: electronic library and information systems, 14(3):130-137, 1980 («*Алгоритм для удаления суффиксов*»)). В естественно-языковом инструментарии для Python – библиотеке NLTK (http://www.nltk.org) – реализован стеммер Портера, которым мы воспользуемся в следующем ниже фрагменте исходного кода. Чтобы установить библиотеку NLTK, просто выполните pip install nltk.

```
>>>
from nltk.stem.porter import PorterStemmer
porter = PorterStemmer()
def tokenizer_porter(text):
    return [porter.stem(word) for word in text.split()]
tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Несмотря на то что естественно-языковой инструментарий NLTK (natural language toolkit) не находится в центре внимания этой главы, настоятельно рекомендуем вам посетить веб-сайт библиотеки NLTK, а также почитать официальную книгу NLTK, которая находится в свободном доступе на http://www.nltk.org/book/, в случае если вы интересуетесь более продвинутыми приложениями в области обработки ЕЯ (NLP).

При помощи класса PorterStemmer из библиотеки nltk мы модифицировали нашу функцию лексемизации tokenizer, в результате чего она редуцирует слова к их основам, что проиллюстрировано на предыдущем простом примере, где из слова running была выделена его основа run.

Алгоритм выделения основ слов Портера является, вероятно, одним из старейших и простейших алгоритмов выделения основ слов. Другие популярные алгоритмы выделения основ слов включают более новый **стеммер Snowball** (Porter2, или «английский» стеммер) и **стеммер Ланкастерского университета** (стеммер Пэйс-Хаска), который быстрее, но и агрессивнее алгоритма Портера. Альтернативные алгоритмы выделения основ слов также имеются в библиотеке NLTK (http://www.nltk.org/api/nltk.stem.html)[1]. Следует учесть, что, как показано в предыдущем примере, алгоритм выделения основ слов может создавать несуществующие слова, такие как thu (из thus). Между тем метод под названием «**лемматизация**» стремится получать канонические (грамматически правильные) формы отдельных слов – так называемые **леммы**. Однако лемматизация в вычислительном плане более трудна и затратна, по сравнению со стеммингом, при этом на практике было отмечено, что стемминг и лемматизация оказывают мало влияния на качество выполнения классификации текстов (Michal Toman, Roman Tesar and Karel Jezek, «Influence of word normalization on text classification». Proceedings of InSciT, p. 354–358, 2006 («*Влияние нормализации слов на классификацию текстов*»)).

Прежде чем перейти к следующему разделу, где мы натренировали машиннообучаемую модель при помощи модели мешка слов, кратко остановимся на еще одной полезной теме, именуемой **удалением стоп-слов**. Стоп-слова – это просто слова, которые чрезвычайно распространены во всех видах текстов и, скорее всего, не несут никакой (либо имеют очень мало) полезной информации, которую можно использовать для различения разных классов документов. Примерами стоп-слов являются

[1] Основательная статья, посвященная анализаторам слов и алгоритмам выделения основ слов, включая ориентированные на русский язык, имеется в Википедии (https://ru.wikipedia.org/wiki/Стемминг). – *Прим. перев.*

is, *has*, *and* и т. п. В случае если мы работаем с исходными или нормализованными частотами терминов, а не tf-idf'ами, которые и без того понижают веса часто встречающихся слов, бывает полезно удалить стоп-слова.

Чтобы удалить стоп-слова из киноотзывов, мы воспользуемся набором из 127 английских стоп-слов, имеющимся в библиотеке NLTK, который можно получить путем вызова функции nltk.download:

```
import nltk
nltk.download('stopwords')
```

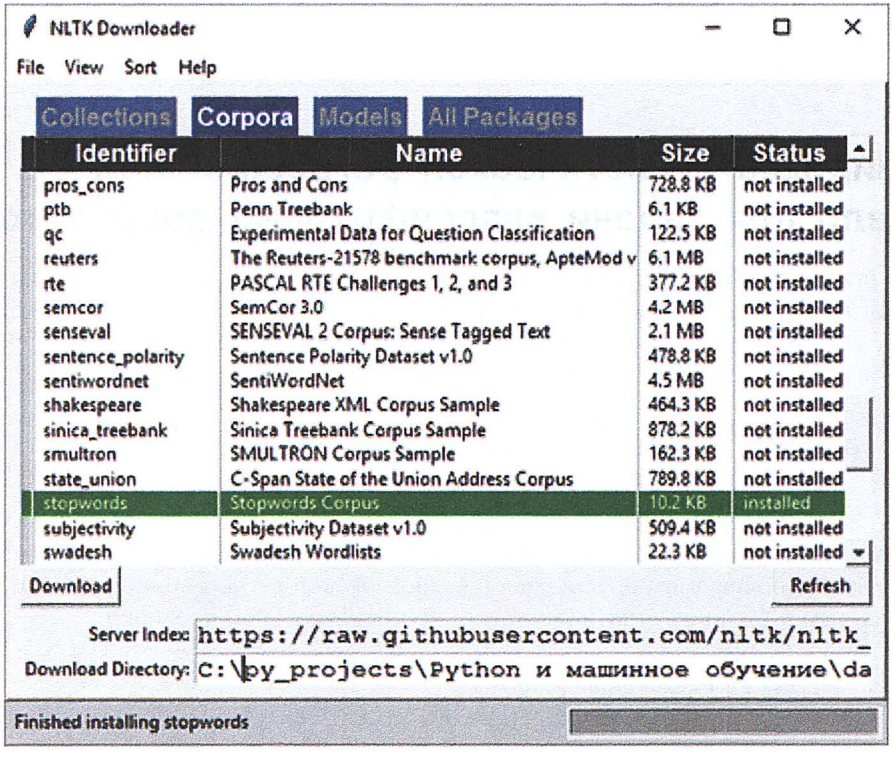
NLTK поставляется с большим корпусом текстовых данных, миниатюрных грамматик, натренированных моделей и т. д. Полный перечень размещен на http://nltk.org/nltk_data/. Чтобы установить эти данные, можно воспользоваться загрузчиком NLTK, как описано ниже.

Помимо отдельных пактов данных, можно скачать всю коллекцию (используя параметр «all») либо только данные, которые требуются для примеров и упражнений из книги по NLTK (используя параметр «book»), либо только корпус документов без грамматик и натренированных моделей (используя параметр «all-corpora»).

Чтобы вызвать окно загрузчика, нужно запустить интерпретатор Python и набрать следующие команды:

```
import nltk
nltk.download()
```

Появится новое окно загрузчика NLTK Downloader.



Нажмите на элементе меню **File** (Файл) и выберите **Change Download Directory** (Поменять каталог для скачивания). Для централизованной установки поменяйте его на C:\nltk_data (Windows), /usr/local/share/nltk_data (Mac) или /usr/share/nltk_data (Linux). Далее выберите пакеты и коллекции, которые вы хотите скачать. Например, на снимке экрана каталог для закачки данных установлен в каталог C:\py_projects\Python и машинное обучение\data, и если выбрать stopwords, то туда будет скачан пакет стоп-слов для разных языков, включая русский.

Если вы не устанавливаете данные в одно из этих центральных мест расположений, то вам нужно задать переменную окружения NLTK_DATA, в которой прописать месторасположение данных (на компьютере под управлением Windows правой кнопкой нажать на кнопке **Пуск** и затем выбрать **Система** ⇒ **Дополнительные параметры системы** ⇒ **Переменные среды** ⇒ **Переменные среды пользователя** ⇒ **Создать...**). Проверить, что данные были установлены, можно следующим образом (подразумевается, что был скачан лингвистический корпус Университета Брауна):

```
>>>
from nltk.corpus import brown
brown.words()
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
```

После того как мы скачали набор стоп-слов, загрузить и применить английский набор стоп-слов можно следующим образом:

```
>>>
from nltk.corpus import stopwords
stop = stopwords.words('english')
[w for w in tokenizer_porter('a runner likes running and runs a lot')[-10:]
 if w not in stop]

['runner', 'like', 'run', 'run', 'lot']
```

Тренировка логистической регрессионной модели для задачи классификации документов

В этом разделе мы натренируем логистическую регрессионную модель для классификации киноотзывов на положительные и отрицательные. Сначала мы разделим таблицу данных DataFrame с очищенными текстовыми документами на 25 000 документов для проведения тренировки и 25 000 документов для тестирования:

```
X_train = df.loc[:25000, 'review'].values
y_train = df.loc[:25000, 'sentiment'].values
X_test = df.loc[25000:, 'review'].values
y_test = df.loc[25000:, 'sentiment'].values
```

Затем воспользуемся объектом сеточного поиска GridSearchCV, чтобы найти оптимальный набор параметров для нашей логистической регрессионной модели с использованием 5-блочной стратифицированной перекрестной проверки:

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
tfidf = TfidfVectorizer(strip_accents=None,
                        lowercase=False,
                        preprocessor=None)
param_grid = [{'vect__ngram_range': [(1,1)],
               'vect__stop_words': [stop, None],
               'vect__tokenizer': [tokenizer,
                                   tokenizer_porter],
                'clf__penalty': ['l1', 'l2'],
                'clf__C': [1.0, 10.0, 100.0]},
               {'vect__ngram_range': [(1,1)],
                'vect__stop_words': [stop, None],
                'vect__tokenizer': [tokenizer,
                                    tokenizer_porter],
                'vect__use_idf':[False],
                'vect__norm':[None],
                'clf__penalty': ['l1', 'l2'],
                'clf__C': [1.0, 10.0, 100.0]}
               ]
lr_tfidf = Pipeline([('vect', tfidf),
                     ('clf',
                      LogisticRegression(random_state=0))])
gs_lr_tfidf = GridSearchCV(lr_tfidf, param_grid,
                           scoring='accuracy',
                           cv=5, verbose=1,
                           n_jobs=-1)
gs_lr_tfidf.fit(X_train, y_train)
```

Инициализировав при помощи приведенного выше исходного кода объект GridSearchCV и его сетку параметров, мы ограничились небольшим числом комбинаций параметров, поскольку число векторов признаков и большой вокабуляр могут сделать поиск по сетке параметров в вычислительном отношении довольно затратным; на стандартном настольном компьютере выполнение сеточного поиска может занять до 40 минут.

В приведенном выше примере мы заменили векторизатор частотностей CountVectorizer и преобразователь tf-idf'ов TfidfTransformer из предыдущего подраздела на векторизатор TfidfVectorizer, который объединяет упомянутые объекты. Наша сетка параметров param_grid состояла из двух словарей параметров. В первом словаре мы использовали векторизатор TfidfVectorizer с его настройками по умолчанию (use_idf=True, smooth_idf=True и norm='l2'), чтобы вычислить tf-idf'ы; во втором словаре мы установили эти параметры в use_idf=False, smooth_idf=False и norm=None, чтобы натренировать модель на основе исходных частот терминов. Кроме того, для непосредственно самого логистического регрессионного классификатора мы натренировали модели с использованием L2- и L1-регуляризации и штрафного параметра clf__penalty и затем сравнили разные силы регуляризации, задав диапазон значений для параметра обратной регуляризации C.

После завершения поиска по сетке параметров можно распечатать наилучший набор параметров:

```
>>>
print('Наилучший набор параметров: %s ' % gs_lr_tfidf.best_params_)
Наилучший набор параметров: {<clf__C': 10.0, 'vect__stop_words': None,
'clf__penalty': 'l2', 'vect__tokenizer': <function tokenizer at
0x7f6c704948c8>, 'vect__ngram_range': (1, 1)}
```

Как здесь видно, мы получили наилучшие результаты поиска по сетке параметров с применением обычного анализатора лексем без стемминга Портера, без библиотеки стоп-слов и tf-idf'ов в сочетании с логистическим регрессионным классификатором, который использует регуляризацию L2 с силой регуляризации `C=10.0`.

На основе наилучшей модели, полученной в результате поиска по сетке параметров, распечатаем среднюю оценку верности по результатам 5-блочной перекрестной проверки на тренировочном наборе и верность классификации на тестовом наборе:

```
>>>
print('Перекрестно-проверочная верность: %.3f'
      % gs_lr_tfidf.best_score_)
Перекрестно-проверочная верность: 0.897
clf = gs_lr_tfidf.best_estimator_
print('Верность на тестовом наборе: %.3f'
      % clf.score(X_test, y_test))
Верность на тестовом наборе: 0.899
```

Результаты показывают, что наша машиннообучаемая модель может идентифицировать киноотзывы как положительные либо как отрицательные с верностью 90%.

Наивный байесовский классификатор продолжает оставаться очень популярным классификатором для классификации текстов, завоевавшим популярность в приложениях фильтрации спама в электронных сообщениях. Наивные байесовские классификаторы легко реализуемы, эффективны в вычислительном плане и демонстрируют тенденцию выполняться особенно хорошо на относительно малых наборах данных, по сравнению с другими алгоритмами. Несмотря на то что в этой книге мы не обсуждаем наивных байесовских классификаторов, заинтересованный читатель может найти мою статью о наивной классификации текстов, которая размещена в свободном доступе на arXiv (S. Raschka. *Naive Bayes and Text Classification I – introduction and Theory* («Наивный Байес и классификация текстов I – введение и теория»). Computing Research Repository (CoRR), abs/1410.5329, 2014, http://arxiv.org/pdf/1410.5329v3.pdf).

Работа с более крупными данными – динамические алгоритмы и обучение вне ядра

Если вы выполнили примеры исходного кода из предыдущего раздела, то, возможно, заметили, что построение векторов признаков для набора данных из 50 000 киноотзывов во время поиска по сетке параметров может в вычислительном плане оказаться довольно затратным. Во многих реальных приложениях очень часто приходится работать с еще более крупными наборами данных, которые могут даже превышать память используемого компьютера. Поскольку не у всех имеется доступ к суперкомпьютерному обеспечению, мы теперь применим метод, именуемый **обучением вне ядра** (out-of-core learning), т. е. с использованием внешней памяти, который позволяет работать с такими большими наборами данных.

Еще в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* мы представили принцип работы алгоритма **стохастического градиентного спуска** (СГС, SGD), т. е. алгоритма оптимизации, который обновляет модельные веса по одному образцу за один раз. В этом разделе мы воспользуемся функцией `partial_fit` классификатора на основе стохастического градиентного спуска `SGDClas-`

sifier библиотеки scikit-learn, чтобы передавать поток документов непосредственно из нашего локального диска и тренировать логистическую регрессионную модель с использованием небольших мини-пакетов документов.

Сначала определим функцию лексемизации tokenizer, которая очищает необработанные текстовые данные из нашего файла movie_data.csv, созданного нами в начале этой главы, и разделяет их на словарные лексемы, удаляя при этом стоп-слова.

```
import numpy as np
import re
from nltk.corpus import stopwords
stop = stopwords.words('english')
def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
    text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
               + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized
```

Далее определим генераторную функцию stream_docs, которая считывает и выдает по одному документу за раз:

```
def stream_docs(path):
    with open(path, 'r', encoding='utf-8') as csv:
        next(csv) # пропустить заголовок
        for line in csv:
            text, label = line[:-3], int(line[-2])
            yield text, label
```

Чтобы удостовериться, что наша функция stream_docs работает правильно, прочитаем из файла movie_data.csv первый документ, который должен вернуть кортеж, состоящий из текста отзыва и соответствующей метки класса:

```
>>>
next(stream_docs(path='./movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

Теперь определим функцию get_minibatch, которая принимает поток документов из функции stream_docs и возвращает отдельно взятое число документов, заданных в параметре size:

```
def get_minibatch(doc_stream, size):
    docs, y = [], []
    try:
        for _ in range(size):
            text, label = next(doc_stream)
            docs.append(text)
            y.append(label)

        except StopIteration:
            return None, None
    return docs, y
```

К сожалению, мы не сможем использовать векторизатор частотностей CountVectorizer для обучения вне ядра, поскольку он потребует наличия в памяти полного

словаря. Кроме того, векторизатор tf-idf-ов TfidfVectorizer должен поддерживать в памяти все векторы признаков тренировочного набора данных, для того чтобы вычислить обратные частоты документов. Однако в библиотеке scikit-learn реализован еще один полезный векторизатор для обработки текста – хэширующий векторизатор HashingVectorizer. Хэширующий векторизатор HashingVectorizer независим от данных и хэширует признаки (посредством хэш-трюка) на основе 32-разрядного алгоритма MurmurHash3 Остина Эпплби (https://sites.google.com/site/murmurhash/).

```
from sklearn.feature_extraction.text import HashingVectorizer
from sklearn.linear_model import SGDClassifier
vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
clf = SGDClassifier(loss='log', random_state=1, n_iter=1)
doc_stream = stream_docs(path='./movie_data.csv')
```

При помощи приведенного выше исходного кода мы инициализировали хэширующий векторизатор HashingVectorizer нашей функцией tokenizer и определили число признаков равным 2^{21}. Кроме того, мы повторно инициализировали логистический регрессионный классификатор, задав параметр потерь loss классификатора SGDClassifier равным log, – отметим, что, выбирая большое число признаков в хэширующем векторизаторе HashingVectorizer, мы уменьшаем шанс вызвать хэш-коллизию[1], но мы также увеличиваем число коэффициентов в нашей логистической регрессионной модели.

Теперь наступает по-настоящему интересная часть. Настроив все вспомогательные функции, мы теперь можем приступить к обучению вне ядра, воспользовавшись для этого следующим ниже исходным кодом:

```
>>>
import pyprind
pbar = pyprind.ProgBar(45)
classes = np.array([0, 1])
for _ in range(45):
    X_train, y_train = get_minibatch(doc_stream, size=1000)
    if not X_train:
        break
    X_train = vect.transform(X_train)
    clf.partial_fit(X_train, y_train, classes=classes)
    pbar.update()

0%                          100%
[##############################] | ETA[sec]: 0.000
Total time elapsed: 50.063 sec
```

Мы снова воспользовались библиотекой PyPrind, чтобы проследить за ходом выполнения алгоритма обучения. Мы инициализировали объект индикатора выполнения 45 итерациями и в следующем цикле for выполнили итерации по 45 мини-пакетам документов, где каждый мини-пакет состоит из 1000 документов.

---

[1] Хэш-коллизия (хэш-конфликт) – ситуация, когда два различных ключа выбирают («хэшируют») одно и то же значение, указывая на одну и ту же ячейку в хэш-таблице. – *Прим. перев.*

Закончив инкрементный процесс обучения, мы воспользуемся последними 5000 документами, чтобы оценить качество нашей модели:

```
>>>
X_test, y_test = get_minibatch(doc_stream, size=5000)
X_test = vect.transform(X_test)
print('Верность: %.3f' % clf.score(X_test, y_test))
Верность: 0.868
```

Как видно, верность модели составляет 87%, слегка ниже верности, которую мы достигли в предыдущем разделе в результате поиска по сетке параметров с целью тонкой настройки гиперпараметров. Вместе с тем обучение вне ядра очень эффективно использует память и заняло менее минуты на выполнение всей работы. В заключение мы можем задействовать последние 5000 документов для обновления нашей модели:

```
clf = clf.partial_fit(X_test, y_test)
```

Если вы планируете перейти прямиком к *главе 9 «Встраивание алгоритма машинного обучения в веб-приложение»*, то рекомендуем вам оставить текущий сеанс Python открытым. В следующей главе мы будем использовать модель, которую мы только что натренировали, чтобы научиться сохранять ее на диск для более позднего использования и встраивать ее в веб-приложение.

Несмотря на то что модель мешка слов продолжает оставаться наиболее широко используемой моделью для классификации текстов, она не рассматривает структуру предложений и грамматику. Популярным расширением модели мешка слов является тематическая модель **латентного распределения Дирихле**, которая рассматривает скрытую семантику слов (D. M. Blei, A. Y. Ng and M. I. Jordan, «Latent Dirichlet allocation». The Journal of machine Learning research, 3:993-1022, 2003 (*«Латентное распределение Дирихле»*)).

Более современной альтернативой модели мешка слов является алгоритм **word2vec**, который был опубликован компанией Google в 2013 г. (T. Mikolov, K. Chen, G. Corrado and J. Dean, «Efficient Estimation of Word Representations in Vector Space». arXiv preprint arXiv:1301.3781, 2013 (*«Эффективная оценка представлений слов в векторном пространстве»*)). Алгоритм word2vec – это алгоритм обучения без учителя на основе нейронных сетей, который пытается автоматически извлечь связи между словами. В основе алгоритма word2vec лежит идея, которая состоит в том, чтобы помещать слова, имеющие подобные значения, в подобные группы; благодаря умному распределению векторного пространства (vector-spacing) модель может воспроизводить определенные слова при помощи простой векторной математики, например *король – мужчина + женщина = королева*.

Исходную реализацию алгоритма на C вместе с полезными ссылками на соответствующие публикации и альтернативные реализации можно найти на https://code.google.com/p/word2vec/.

Резюме

В этой главе мы научились использовать алгоритмы машинного обучения для выполнения задачи классификации текстовых документов на основе их полярности, составляющей базовую задачу в анализе мнений в области обработки естественного

языка. Мы не только научились кодировать документ как вектор признаков, используя для этого модель мешка слов, но и взвешивать частоту термина по релевантности, используя для этого метод частоты термина – обратной частоты документа (tf-idf).

Ввиду больших векторов признаков, создаваемых во время этого процесса, работа с текстовыми данными может быть в вычислительном плане довольно затратной; в последнем разделе мы научились использовать инкрементное обучение (вне ядра) для обучения алгоритма машинного обучения без загрузки всего набора данных в оперативную память компьютера.

В следующей главе мы воспользуемся нашим классификатором документов и научимся внедрять его в веб-приложение.

Глава 9

Встраивание алгоритма машинного обучения в веб-приложение

В предыдущих главах вы познакомились с большим количеством разнообразных принципов и алгоритмов машинного обучения, которые способны помочь принимать более успешные и более эффективные решения. Однако методы машинного обучения не ограничиваются офлайновыми приложениями и анализом и могут стать прогнозным движком ваших веб-служб. Например, популярные и полезные применения машиннообучаемых моделей в веб-приложениях включают в себя обнаружение спама в отправляемых формах, поисковые движки, рекомендательные системы для СМИ или порталов покупок и еще много другого.

В этой главе вы узнаете, как интегрировать машиннообучаемую модель в веб-приложение, которая не только сможет классифицировать данные, но и обучаться на данных в режиме реального времени. В этой главе мы охватим следующие темы:

- сохранение текущего состояния натренированной машиннообучаемой модели;
- применение баз данных SQLite для хранения данных;
- разработка веб-приложения при помощи популярной веб-платформы Flask;
- развертывание приложения машинного обучения на публичном сервере.

Сериализация подогнанных оценщиков библиотеки scikit-learn

Тренировка машиннообучаемой модели может быть в вычислительном плане довольно затратной, как мы убедились в *главе 8 «Применение алгоритмов машинного обучения в анализе мнений»*. Разумеется, мы же не хотим тренировать нашу модель всякий раз, когда мы закрываем наш интерпретатор Python и хотим сделать новый прогноз или перезагрузить наше веб-приложение? Одним из вариантов обеспечения **перманентности моделей**, или их живучести, персистентности[1], является встроенный модуль Python консервации данных pickle (https://docs.python.org/3.4/library/pickle.html), который позволяет выполнять сериализацию и десериализацию объектных структур Python в виде компактного байт-кода, в результате чего мы можем сохранять наш классификатор в его текущем состоянии и перезагружать его, если мы

[1] Персистентность (persistance) – в программировании означает способность состояния существовать дольше, чем процесс, который его создал. Без этой возможности состояние может существовать только в оперативной памяти и теряется, когда оперативная память выключается, например при выключении компьютера. – *Прим. перев.*

хотим выполнить классификацию новых образцов без необходимости снова и снова извлекать модель из тренировочных данных. Перед выполнением следующего ниже исходного кода удостоверьтесь, что вы обучили логистическую регрессионную модель из последнего раздела *главы 8 «Применение алгоритмов машинного обучения в анализе мнений»* вне ядра и имеете ее в готовом виде в текущем сеансе Python:

```
import pickle
import os
dest = os.path.join('movieclassifier', 'pkl_objects')
if not os.path.exists(dest):
    os.makedirs(dest)
pickle.dump(stop,
            open(os.path.join(dest, 'stopwords.pkl'),'wb'),
            protocol=4)
pickle.dump(clf,
            open(os.path.join(dest, 'classifier.pkl'),'wb'),
            protocol=4)
```

Используя приведенный выше исходный код, мы создали каталог классификатора movieclassifier, куда мы позже будем сохранять файлы и данные для нашего веб-приложения. В каталоге movieclassifier мы создали подкаталог pkl_objects для сохранения на нашем локальном диске сериализованных объектов Python. При помощи метода dump библиотеки pickle мы затем сериализовали натренированную логистическую регрессионную модель, а также набор стоп-слов из библиотеки NLTK, в результате чего нам не нужно на нашем сервере устанавливать словарь NLTK. Метод dump в качестве своего первого аргумента принимает объект, который мы хотим законсервировать, и в качестве второго аргумента мы предоставили открытый файловый объект, куда объект Python будет записан. При помощи аргумента wb в функции open мы открыли файл в двоичном режиме для выполнения консервации и задали параметр протокола protocol=4, чтобы выбрать последний и самый эффективный протокол консервации, который был добавлен в Python 3.4 (если у вас имеются проблемы с протоколом 4, то убедитесь, что вы используете последнюю версию Python 3. Как вариант можете рассмотреть применение протокола с более низким числом).

Наша логистическая регрессионная модель содержит несколько массивов библиотеки NumPy, таких как весовой вектор, при этом более эффективный способ сериализации массивов NumPy состоит в том, чтобы использовать альтернативную библиотеку joblib. Для обеспечения совместимости с серверной средой, которую мы будем использовать в более поздних разделах главы, мы будем использовать стандартный подход консервации. Если вы интересуетесь, то по библиотеке joblib вы можете найти дополнительную информацию на https://pypi.python.org/pypi/joblib.

Нам не нужно консервировать хэширующий векторизатор HashingVectorizer, поскольку он не требует выполнения подгонки. Вместо этого можно создать новый сценарный файл Python, из которого можно импортировать векторизатор в наш текущий сеанс Python. Теперь скопируем следующий ниже исходный код и сохраним его как vectorizer.py в каталоге классификатора movieclassifier:

```
from sklearn.feature_extraction.text import HashingVectorizer
import re
import os
import pickle
```

```
cur_dir = os.path.dirname(__file__)
stop = pickle.load(open(
                    os.path.join(cur_dir,
                                 'pkl_objects',
                                 'stopwords.pkl'), 'rb'))
def tokenizer(text):
    text = re.sub('<[^>]*>', '', text)
    emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
                           text.lower())
    text = re.sub('[\W]+', ' ', text.lower()) \
               + ' '.join(emoticons).replace('-', '')
    tokenized = [w for w in text.split() if w not in stop]
    return tokenized

vect = HashingVectorizer(decode_error='ignore',
                         n_features=2**21,
                         preprocessor=None,
                         tokenizer=tokenizer)
```

После того как мы законсервировали объекты Python и создали файл vectorizer.py, теперь неплохо перезапустить наш интерпретатор Python или ядро блокнотов Jupyter, с тем чтобы проверить, сможем ли мы без ошибки десериализовать объекты. Однако стоит отметить, что расконсервация данных из источника, не заслуживающего доверия, может представлять потенциальную угрозу безопасности, поскольку библиотека pickle не обеспечивает защиту от вредоносного программного кода. В окне терминала перейдем в каталог movieclassifier, запустим новый сеанс Python и выполним следующий ниже исходный код, чтобы проверить, что вы можете импортировать векторизатор и расконсервировать классификатор:

```
import pickle
import re
import os
from vectorizer import vect
clf = pickle.load(open(
                    os.path.join('pkl_objects',
                                 'classifier.pkl'), 'rb'))
```

После того как мы успешно загрузили векторизатор и расконсервировали классификатор, теперь можно использовать эти объекты для предобработки образцов документов и прогнозирования мнений:

```
>>>
import numpy as np
label = {0:'negative', 1:'positive'}
example = ['I love this movie']
X = vect.transform(example)
print('Прогноз: %s\nВероятность: %.2f%%' %\
       (label[clf.predict(X)[0]],
        np.max(clf.predict_proba(X))*100))

Прогноз: positive
Вероятность: 91.56%
```

С учетом того, что наш классификатор возвращает метки классов в виде целых чисел, мы определили простой словарь Python, в котором целым числам поставлены в соответствие мнения. Затем мы применили хэширующий векторизатор Hashing-

Vectorizer для преобразования простого примера документа в словарный вектор X. В заключение мы использовали метод predict логистического регрессионного классификатора для прогнозирования метки класса, а также метод predict_proba для возврата соответствующей вероятности нашего прогноза. Отметим, что вызов метода predict_proba возвращает массив со значением вероятности для каждой уникальной метки класса. Поскольку метка класса с самой большой вероятностью соответствует метке класса, полученной в результате вызова метода predict, мы использовали функцию np.max для возврата вероятности спрогнозированного класса.

Настройка базы данных SQLite для хранения данных

В этом разделе мы настроим простую **базу данных SQLite** для аккумулирования дополнительной информации с отзывами о выполненных прогнозах от пользователей веб-приложения. Мы можем использовать эту обратную связь для обновления нашей модели классификации. СУБД SQLite представляет собой общедоступное ядро базы данных SQL, не требующее работы отдельного сервера, что делает ее идеальным вариантом для малых проектов и простых веб-приложений. В сущности, база данных SQLite может пониматься как одиночный автономный файл базы данных с прямым доступом к хранящимся данным. Кроме того, СУБД SQLite не требует никакого системно-специфичного конфигурирования и поддерживается всеми распространенными операционными системами. Она заслуженно заработала репутацию за свою надежность, т. к. используется такими популярными компаниями, как Google, Mozilla, Adobe, Apple, Microsoft, и еще множеством других. Если вы хотите узнать о SQLite больше, то рекомендуем посетить ее официальный веб-сайт на http://www.sqlite.org.

К счастью, следуя философии Python *«батарейки в комплекте»*, в стандартной библиотеке Python уже имеется программный интерфейс (API) **sqlite3**, который позволяет работать с базами данных SQLite (для получения дополнительной информации о sqlite3, пожалуйста, посетите https://docs.python. org/3.4/library/sqlite3.html).

Выполнив следующий ниже исходный код, мы создадим в каталоге movieclassifier новую базу данных SQLite и сохраним два киноотзыва в качестве примера:

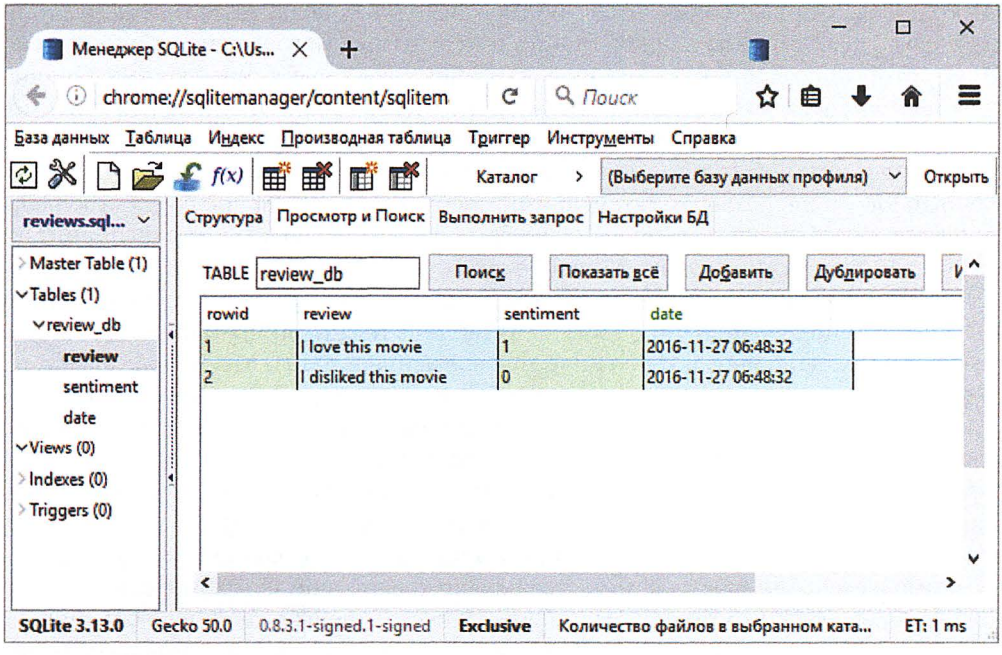
```
import sqlite3
import os
conn = sqlite3.connect('reviews.sqlite')
c = conn.cursor()
c.execute('CREATE TABLE review_db'\
          ' (review TEXT, sentiment INTEGER, date TEXT)')
example1 = 'I love this movie'
c.execute("INSERT INTO review_db"\
          " (review, sentiment, date) VALUES"\
          " (?, ?, DATETIME('now'))", (example1, 1))
example2 = 'I disliked this movie'
c.execute("INSERT INTO review_db"\
          " (review, sentiment, date) VALUES"\
          " (?, ?, DATETIME('now'))", (example2, 0))
conn.commit()
conn.close()
```

Прослеживая приведенный выше пример исходного кода, мы создали соединение (conn) с файлом базы данных SQLite путем вызова метода connect библиотеки sqlite3, который создал в каталоге movieclassifier новый файл базы данных киноотзывов reviews.sqlite, в случае если его еще не существовало. Отметим, что в SQLite функция замены существующих таблиц не реализована; если хотите выполнить исходный код во второй раз, вы должны удалить файл базы данных вручную из вашего файлового диспетчера. Затем при помощи метода cursor мы создали курсор, позволяющий перемещаться по записям базы данных, используя мощный синтаксис SQL. В результате первого вызова execute мы потом создали новую таблицу базы данных киноотзывов review_db. Мы использовали ее для хранения записей базы данных и получения к ним доступа. В таблице review_db мы также создали три столбца: review, sentiment и date (киноотзыв, мнение и дата). Мы использовали их для хранения двух примеров киноотзывов и соответствующих меток классов (мнений). При помощи команды SQL DATETIME('now') мы также добавили в наши записи метки даты и времени. В дополнение к меткам времени мы применили символы вопросительного знака (?) для передачи в виде членов кортежа текста киноотзывов (example1 и example2) и соответствующих меток классов (1 и 0) как позиционных аргументов метода execute. В заключение мы вызвали метод commit, чтобы записать внесенные изменения в базу данных, и закрыли соединение методом close.

Чтобы проверить, что записи были корректно сохранены в таблице базы данных, мы теперь вновь откроем соединение с базой данных и применим команду SQL SELECT, чтобы выбрать все строки в таблице базы данных, зафиксированных между началом 2015 г. и текущей датой:

```
>>>
conn = sqlite3.connect('reviews.sqlite')
c = conn.cursor()
c.execute("SELECT * FROM review_db WHERE date"\
          " BETWEEN '2015-01-01 00:00:00' AND DATETIME('now')")
results = c.fetchall()
conn.close()
print(results)

[('I love this movie', 1, '2015-06-02 16:02:12'), ('I disliked this
movie', 0, '2015-06-02 16:02:12')]
```

Как вариант можно было воспользоваться бесплатным плагином браузера Firefox **менеджер SQLite** (доступным по ссылке https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/), который предлагает удобный графический интерфейс для работы с базами данных SQLite, как показано на следующем ниже снимке экрана:



Разработка веб-приложения в веб-платформе Flask

После того как в предыдущем подразделе мы подготовили исходный код для выполнения классификации киноотзывов, обсудим основы веб-платформы Flask, требующейся для разработки нашего веб-приложения. После первого выпуска Армином Ронэкэром в 2010 г. за прошедшие годы веб-платформа Flask получила огромную популярность, и примеры популярных приложений с использованием Flask включают LinkedIn и Pinterest. Поскольку веб-платформа Flask написана на Python, она предоставляет нам, программистам на Python, удобный интерфейс для встраивания существующего исходного кода на Python, такого как наш классификатор кинофильмов.

Веб-платформу Flask также называют микроплатформой, что означает, что ее ядро остается компактным и простым и одновременно может быть легко расширено другими библиотеками. Несмотря на то что кривая обучаемости легковесному программному интерфейсу веб-платформы Flask даже отдаленно не напоминает крутую кривую других популярных веб-платформ Python, таких как Django, все же призываем вас обратиться к официальной документации по Flask на http://flask.pocoo.org/docs/0.10/, чтобы узнать о его функциональности больше.

Если библиотека Flask еще не установлена в вашей среде Python, вы можете легко установить ее при помощи менеджера пакетов pip прямо из вашего терминала (во время написания книги последним стабильным выпуском была версия 0.10.1, во время перевода данной книги – 0.11.1): pip install flask.

Наше первое веб-приложение Flask

В этом подразделе мы разработаем очень простое веб-приложение, для того чтобы познакомиться поближе с программным интерфейсом веб-платформы Flask, прежде чем приступим к реализации классификатора кинофильмов. Сначала создадим дерево каталогов:

```
1st_flask_app_1/
    app.py
    templates/
        first_app.html
```

Файл приложения app.py будет содержать основной исходный код, который будет исполняться интерпретатором Python для выполнения веб-приложения на основе Flask. Каталог templates – это место, где Flask будет искать статические файлы HTML для вывода в окне веб-браузера. Теперь посмотрим на содержимое файла app.py:

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('first_app.html')

if __name__ == '__main__':
    app.run()
```

В данном случае мы выполняем наше приложение как единственный модуль, поэтому мы инициализировали новый экземпляр Flask с аргументом __name__, чтобы дать веб-платформе Flask знать, что она может найти папку HTML-шаблонов в том же каталоге, где расположена сама веб-платформа. Далее мы использовали декоратор маршрута (@app.route('/')) для указания URL-адреса, который должен активировать запуск функции index. Здесь наша функция index просто выводит в браузере файл HTML first_app.html, расположенный в папке шаблонов templates. В заключение мы использовали функцию run, просто чтобы выполнить приложение на сервере, когда этот сценарий выполняется непосредственно интерпретатором Python; мы это обеспечили при помощи условного оператора if __name__ == '__main__'.

Теперь посмотрим на содержимое файла first_app.html. Если вы еще не знакомы с синтаксисом HTML, то рекомендуем посетить веб-страницу https://developer.mozilla.org/en-US/docs/Web/HTML по поводу полезных учебных пособий для изучения основ HTML.

```
<!doctype html>
<html>
  <head>
    <title>Первое приложение</title>
  </head>
  <body>
    <div>Привет! Это мое первое веб-приложение Flask!</div>
  </body>
</html>
```

Здесь мы просто заполнили пустой файл HTML-шаблона элементом div (элементом блочного уровня), который содержит предложение: Привет, это мое первое веб-приложение на основе Flask!. Удобным образом веб-платформа Flask позволяет нам управлять нашими приложениями локально, что оказывается полезным для разработки и тестирования веб-приложений перед их развертыванием на публичном веб-сервере. Теперь запустим наше веб-приложение, исполнив команду из терминала внутри каталога 1st_flask_app_1:

```
python3 app.py
```

Мы должны теперь увидеть в терминале строку, такую как показано ниже:

```
* Running on http://127.0.0.1:5000/
```

Эта строка содержит адрес нашего локального сервера. Теперь можно ввести этот адрес в наш веб-браузер и в результате увидеть веб-приложение в действии. Если все выполнено правильно, то мы должны увидеть простой веб-сайт с приветственным сообщением: «**Привет! Это мое первое веб-приложение Flask!**»

Протокол запуска веб-приложения Flask в Windows:
Открыть окно терминала/командной строки и набрать:

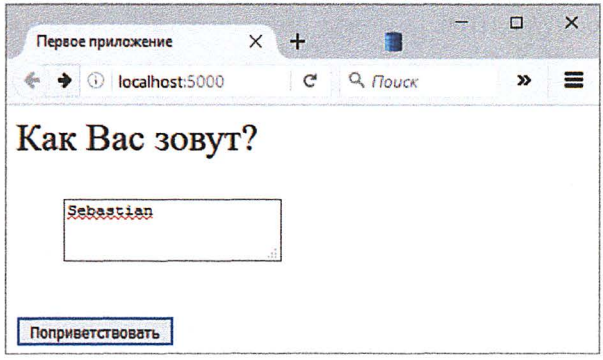
```
C:\> cd C:\Users\путь\к\приложению-Flask\1st_flask_app_1
C:\Users\путь\к\приложению-Flask\1st_flask_app_1> python app.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 281-870-455
```

Открыть веб-браузер и набрать в адресной строке: localhost:5000.

Валидация и отображение формы

В этом подразделе мы расширим наше простое веб-приложение Flask элементами HTML-формы, чтобы научиться собирать данные от пользователя, используя для этого библиотеку **WTForms** (https://wtforms.readthedocs.org/en/latest/), которую можно установить при помощи менеджера пакетов pip: pip install wtforms.

Это веб-приложение пригласит пользователя набрать в текстовом поле свое имя, как показано в следующем ниже снимке экрана:



После того как кнопка отправки данных на сервер (**Поприветствовать**) была нажата и форма была проверена, будет сгенерирована новая страница HTML, которая покажет имя пользователя.

Для такого приложения нам нужно настроить новую структуру каталогов, которая выглядит следующим образом:

```
1st_flask_app_2/
    app.py
    static/
        style.css
    templates/
        _formhelpers.html
        first_app.html
        hello.html
```

Ниже приведено содержимое модифицированного файла приложения app.py:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators

app = Flask(__name__)

class HelloForm(Form):
    sayhello = TextAreaField('',[validators.DataRequired()])

@app.route('/')
def index():
    form = HelloForm(request.form)
    return render_template('first_app.html', form=form)

@app.route('/hello', methods=['POST'])
def hello():
    form = HelloForm(request.form)
    if request.method == 'POST' and form.validate():
        name = request.form['sayhello']
        return render_template('hello.html', name=name)
    return render_template('first_app.html', form=form)

if __name__ == '__main__':
    app.run(debug=True)
```

При помощи библиотеки wtforms мы расширили функцию index текстовым полем, которое мы включим в нашу стартовую страницу, пользуясь классом TextAreaField, который автоматически проверит, предоставил пользователь допустимый входной текст или нет. Кроме того, мы определили новую функцию hello, которая сгенерирует страницу HTML hello.html, в случае если форма была подтверждена. Здесь в теле

сообщения для транспортировки данных формы на сервер мы использовали метод POST. В заключение, задав внутри метода app.run аргумент debug=True, мы далее активировали отладчик веб-платформы Flask. Этот полезный функционал используется для разработки новых веб-приложений.

Теперь в файле _formhelpers.html мы реализуем универсальную макрокоманду, воспользовавшись для этого движком шаблонной обработки (шаблонизатором) **Jinja2**, который мы позже импортируем в наш файл first_app.html для генерирования текстового поля:

```
{% macro render_field(field) %}
  <dt>{{ field.label }}
  <dd>{{ field(**kwargs)|safe }}
  {% if field.errors %}
    <ul class=errors>
    {% for error in field.errors %}
      <li>{{ error }}</li>
    {% endfor %}
    </ul>
  {% endif %}
  </dd>
  </dt>
{% endmacro %}
```

Всестороннее обсуждение языка шаблонной обработки Jinja2 выходит за рамки этой книги. Однако вы можете найти подробную документацию по синтаксису Jinja2 на http://jinja.pocoo.org.

Далее настраиваем простой файл **каскадных таблиц стилей** (CSS) style.css, чтобы продемонстрировать, каким образом можно изменять внешний вид и поведение документов HTML. Нам нужно сохранить следующий ниже файл CSS, который просто удвоит размер шрифта наших элементов HTML внутри элемента body, в подкаталоге с именем static, где веб-платформа Flask по умолчанию ищет статические файлы, такие как CSS. Содержимое каскадной таблицы стилей следующее:

```
body {
  font-size: 2em;
}
```

Ниже приведено содержимое модифицированного файла first_app.html, который теперь сгенерирует текстовую форму, где пользователь может ввести имя:

```
<!doctype html>
<html>
  <head>
    <title>Первое приложение</title>
    <link rel="stylesheet"
          href="{{ url_for('static', filename='style.css') }}">
  </head>

  <body>

{% from "_formhelpers.html" import render_field %}

    <div>Как вас зовут?</div>
    <form method=post action="/hello">
      <dl>
```

```
        {{ render_field(form.sayhello) }}
      </dl>
      <input type=submit value='Поприветствовать' name='submit_btn'>
    </form>
  </body>
</html>
```

В заголовочной части файла first_app.html мы загрузили файл CSS. Он теперь должен изменить размер всех текстовых элементов в основной части HTML. В основной части HTML мы импортировали макрокоманду формы из файла _formhelpers.html и сгенерировали форму sayhello, которую мы определили в файле app.py. Кроме того, к тому же элементу формы мы добавили кнопку, в результате чего пользователь может отправить на сервер данные текстового поля.

В заключение мы создаем файл hello.html, который будет сгенерирован строкой return render_template('hello.html', name=name) внутри функции hello, которую мы определили в сценарии app.py для отображения текста, который пользователь отправил в текстовом поле. Соответствующий исходный код выглядит следующим образом:

```
<!doctype html>
<html>
  <head>
    <title>Первое приложение</title>
    <link rel="stylesheet"
          href="{{ url_for('static', filename='style.css') }}">
  </head>

  <body>
    <div>Hello {{ name }}</div>
  </body>
</html>
```

Настроив наше модифицированное веб-приложение на основе веб-платформы Flask, мы можем выполнить его локально, исполнив следующую ниже команду из главного каталога приложения, при этом мы сможем увидеть результат в нашем веб-браузере по адресу http://127.0.0.1:5000/:

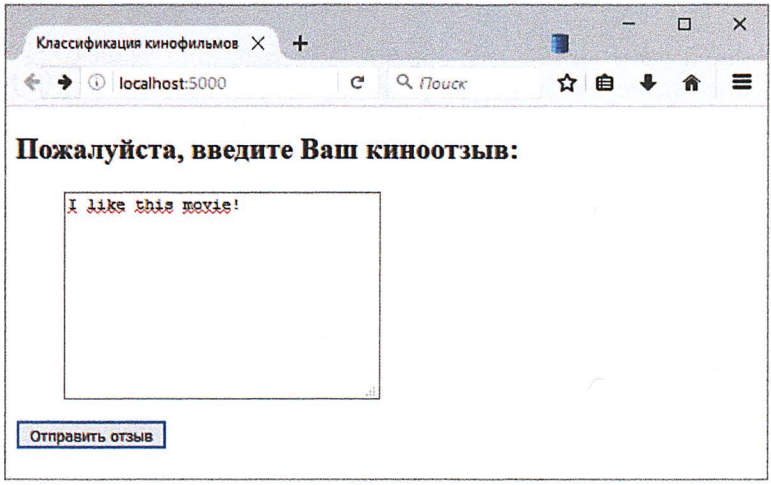
```
python3 app.py
```

Если вы новичок в веб-разработке, то некоторые из представленных понятий могут на первый взгляд показаться очень сложными. В этом случае призываем вас просто выполнить настройку приведенных выше файлов в каталоге на вашем жестком диске и тщательно их проанализировать. Вы увидите, что веб-платформа Flask является на самом деле довольно прямолинейной и намного проще, чем это могло показаться с самого начала! Кроме того, за дополнительной помощью приглашаем обратиться к превосходной документации с примерами по веб-платформе Flask на http://flask.pocoo.org/docs/0.10/.

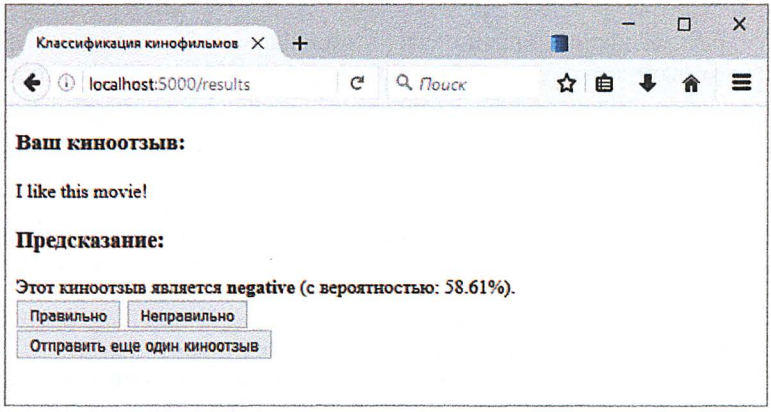
Превращение классификатора кинофильмов в веб-приложение

Несколько ознакомившись с основами веб-разработки на основе веб-платформы Flask, теперь продвинемся дальше и внедрим наш классификатор кинофильмов

в веб-приложение. В этом разделе мы разработаем веб-приложение, которое сначала пригласит пользователя ввести отзыв о кинофильме, как показано на следующем ниже снимке экрана:

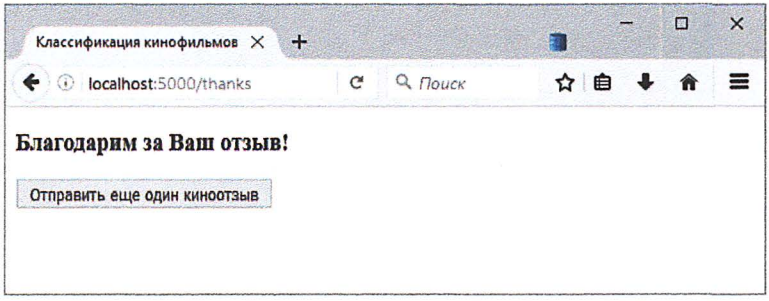


После того как отзыв будет отправлен на сервер, пользователь увидит новую страницу, демонстрирующую спрогнозированную метку класса и вероятность прогноза. Кроме того, пользователь сможет предоставить отклик об этом прогнозе путем нажатия на кнопке **Правильно** или **Неправильно**, как показано на следующем ниже снимке экрана:



Если пользователь нажал на одной из кнопок: **Правильно** или **Неправильно**, наша классификационная модель будет обновлена относительно отклика пользователя. Кроме того, мы также сохраним в базе данных SQLite для дальнейшего использования предоставленный пользователем текст киноотзыва и предложенную метку класса, о которой можно заключить, исходя из нажатия кнопки. Третьей страницей, которую пользователь увидит после нажатия на одну из кнопок обратной связи, будет простое сообщение с благодарностью за предоставленную информацию (*спаси-*

бо) и кнопкой **Отправить еще один киноотзыв**, которая перенаправит пользователя назад к стартовой странице. Это показано на следующем ниже снимке экрана:



Прежде чем мы поближе посмотрим на реализацию исходного кода этого веб-приложения, приглашаем вас посмотреть действующий прототип, выгруженный по URL-адресу http://raschkas.pythonanywhere.com, для того чтобы получше разобраться в том, что мы пытаемся выполнить в этом разделе.

Начав с общей картины, взглянем на приведенное ниже дерево каталогов, которое мы собираемся создать для этого приложения классификации кинофильмов:

В предыдущем разделе этой главы мы уже создали файл vectorizer.py, базу данных SQLite reviews.sqlite и подкаталог pkl_objects с законсервированными объектами Python.

Файл app.py в главном каталоге – это сценарий Python с нашим исходным кодом для веб-платформы Flask. Файл базы данных review.sqlite (который мы создали ранее в этой главе) мы будем использовать для хранения киноотзывов, которые отправляются в наше веб-приложение. Подкаталог шаблонов templates содержит HTML-шаблоны, которые будут использоваться веб-платформой Flask для генерирования страниц и их показа в браузере, и подкаталог static будет содержать простой файл CSS для корректировки внешнего вида генерируемого HTML-кода.

Учитывая, что файл app.py довольно длинный, мы освоим его в два этапа. Первая часть файла app.py импортирует библиотеки, модули и объекты Python, которые

нам потребуются, а также исходный код, который выполняет расконсервацию и настройку нашей классификационной модели:

```
from flask import Flask, render_template, request
from wtforms import Form, TextAreaField, validators
import pickle
import sqlite3
import os
import numpy as np

# импортировать HashingVectorizer из локального каталога
from vectorizer import vect

app = Flask(__name__)

######## Подготовка классификатора
cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                    'pkl_objects/classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

def classify(document):
    label = {0: 'negative', 1: 'positive'}
    X = vect.transform([document])
    y = clf.predict(X)[0]
    proba = clf.predict_proba(X).max()
    return label[y], proba

def train(document, y):
    X = vect.transform([document])
    clf.partial_fit(X, [y])

def sqlite_entry(path, document, y):
    conn = sqlite3.connect(path)
    c = conn.cursor()
    c.execute("INSERT INTO review_db (review, sentiment, date)"\
              " VALUES (?, ?, DATETIME('now'))", (document, y))
    conn.commit()
    conn.close()
```

Первая часть сценария app.py к настоящему времени должна выглядеть очень знакомой. Мы просто импортировали хэширующий векторизатор HashingVectorizer и расконсервировали логистический регрессионный классификатор. Далее мы определили функцию классификации classify, которая для заданного текстового документа возвращает его спрогнозированную метку класса и соответствующий ей вероятностный прогноз. Функция тренировки train может использоваться для обновления классификатора, при условии что документ и метка класса предоставлены. При помощи функции sqlite_entry мы можем сохранить в нашей базе данных SQLite отправленный на сервер киноотзыв, его метку класса и метку времени, которую мы используем в наших личных целях. Отметим, что объект clf будет сброшен в свое первоначальное законсервированное состояние, в случае если мы перезапустим веб-приложение. В конце этой главы вы узнаете, как использовать данные, которые мы аккумулируем в базе данных SQLite для непрерывного обновления классификатора.

Принципы работы во второй части сценария app.py должны также выглядеть довольно знакомыми:

```
app = Flask(__name__)
class ReviewForm(Form):
    moviereview = TextAreaField('',
                                [validators.DataRequired(),
                                 validators.length(min=15)])
@app.route('/')
def index():
    form = ReviewForm(request.form)
    return render_template('reviewform.html', form=form)

@app.route('/results', methods=['POST'])
def results():
    form = ReviewForm(request.form)
    if request.method == 'POST' and form.validate():
        review = request.form['moviereview']
        y, proba = classify(review)
        return render_template('results.html',
                               content=review,
                               prediction=y,
                               probability=round(proba*100, 2))
    return render_template('reviewform.html', form=form)

@app.route('/thanks', methods=['POST'])
def feedback():
    feedback = request.form['feedback_button']
    review = request.form['review']
    prediction = request.form['prediction']

    inv_label = {'negative': 0, 'positive': 1}
    y = inv_label[prediction]
    if feedback == 'Неправильно':
        y = int(not(y))
    train(review, y)
    sqlite_entry(db, review, y)
    return render_template('thanks.html')

if __name__ == '__main__':
    app.run(debug=True)
```

Мы определили класс ReviewForm для формы с киноотзывом, конструирующий экземпляр текстового поля TextAreaField, который будет сгенерирован в файле шаблона reviewform.html (целевая страница нашего веб-приложения). Он, в свою очередь, генерируется функцией index. Параметр validators.length(min=15) задает длину отзыва в символах (не более 15), который пользователь должен ввести. Внутри функции results мы несем содержимое отправленной веб-формы и передаем его в наш классификатор, чтобы выполнить прогноз мнения о кинофильме, который затем будет показан в сгенерированном шаблоне results.html.

Функция отклика пользователя feedback может на первый взгляд выглядеть немного сложной. Она, по сути, приносит из шаблона results.html спрогнозированную метку класса, в случае если пользователь нажал кнопку обратной связи **Правильно** или **Неправильно**, и преобразует спрогнозированное мнение назад в целочисленную метку класса, которая будет использована для обновления классификатора функцией тренировки train, реализованной нами в первой части сценария app.py. Кроме того, функцией sqlite_entry будет сделана новая запись в базе данных SQLite, в случае если был предоставлен отклик пользователя, и в конце будет сгенерирован

шаблон thanks.html, чтобы выразить благодарность пользователю за предоставленный отклик.

Далее посмотрим на шаблон reviewform.html, который соответствует стартовой странице нашего приложения:

```
<!doctype html>
<html>
  <head>
    <title>Классификация кинофильмов</title>
  </head>

  <body>
    <h2>Пожалуйста, введите ваш киноотзыв:</h2>

    {% from "_formhelpers.html" import render_field %}

    <form method=post action="/results">
      <dl>
        {{ render_field(form.moviereview, cols='30', rows='10') }}
      </dl>
      <div>
        <input type=submit value='Отправить отзыв' name='submit_btn'>
      </div>
    </form>

  </body>
</html>
```

Здесь мы просто импортировали тот же самый шаблон _formhelpers.html, который мы определили в разделе *Валидация и отображение формы* ранее в этой главе. Функция render_field этой макрокоманды используется для генерирования текстового поля TextAreaField, где пользователь может предоставить киноотзыв и отправить его кнопкой **Отправить отзыв**, выводимой внизу страницы. Поле текстовой области TextAreaField имеет ширину 30 колонок и высоту 10 строк.

Наш следующий шаблон results.html выглядит немного интереснее:

```
<!doctype html>
<html>
  <head>
    <title>Классификация кинофильмов</title>
    <link rel="stylesheet"
          href="{{ url_for('static',
          filename='style.css') }}">
  </head>

  <body>
    <h3>ваш киноотзыв:</h3>
    <div>{{ content }}</div>

    <h3>Прогноз:</h3>
    <div>Этот киноотзыв является <strong>{{ prediction }}</strong>
      (с вероятностью: {{ probability }}%).</div>

    <div class='button'>
      <form action="/thanks" method="post">
        <input type=submit value='Правильно' name='feedback_button'>
        <input type=submit value='Неправильно' name='feedback_button'>
```

```
        <input type=hidden value='{{ prediction }}' name='prediction'>
        <input type=hidden value='{{ content }}' name='review'>
      </form>
    </div>

    <div class='button'>
      <form action="/">
        <input type=submit value='Отправить еще один отзыв'>
      </form>
    </div>

  </body>
</html>
```

Вначале в соответствующие поля {{ content }}, {{ prediction }} и {{ probability }} мы вставили предложенный отзыв и результаты прогнозирования. Вы можете заметить, что мы использовали замещающие переменные {{ content }} и {{ prediction }} второй раз в форме с кнопками **Правильно** или **Неправильно**. Это обходное решение вместо отправки этих значений методом POST назад на сервер, чтобы выполнить обновление классификатора и сохранить отзыв в случае, если пользователь нажимает на одну из этих двух кнопок. Кроме того, в начале файла results.html мы импортировали файл CSS (style.css). Конфигурация этого файла довольно простая; он ограничивает ширину содержимого этого веб-приложения 600 пикселами и перемещает кнопки **Правильно** или **Неправильно**, помеченные на странице как div class='button', вниз на 20 пикселов:

```
body{
  width:600px;
}
.button{
  padding-top: 20px;
}
```

Этот CSS-файл – всего лишь заготовка, поэтому не стесняйтесь по вашему усмотрению корректировать его для изменения внешнего вида и поведения веб-приложения.

Последним файлом HTML, который мы создадим для нашего веб-приложения, будет шаблон thanks.html. Как видно из названия, он просто предоставляет пользователю подкрепляющее сообщение с благодарностью за предоставленный отклик, отправленный нажатием кнопки **Правильно** или **Неправильно**. Кроме того, в основание этой страницы мы помещаем кнопку **Отправить еще один отзыв**, которая перенаправит пользователя к стартовой странице. Содержимое файла thanks.html приведено ниже:

```
<!doctype html>
<html>
  <head>
    <title>Классифиция кинофильмов</title>
  </head>

  <body>
    <h3>Благодарим за ваш отзыв!</h3>

    <div id='button'>
```

```
      <form action="/">
        <input type=submit value='Отправить еще один киноотзыв'>
      </form>
    </div>

  </body>
</html>
```

Теперь, прежде чем мы перейдем к следующему подразделу и развернем веб-приложение на публичном сервере, неплохо запустить веб-приложение локально из нашего терминала следующей ниже командой:

```
python3 app.py
```

После того как будет завершено тестирование веб-приложения, напоминаем о необходимости удалить аргумент debug=True в команде app.run() нашего сценария app.py.

Развертывание веб-приложения на публичном сервере

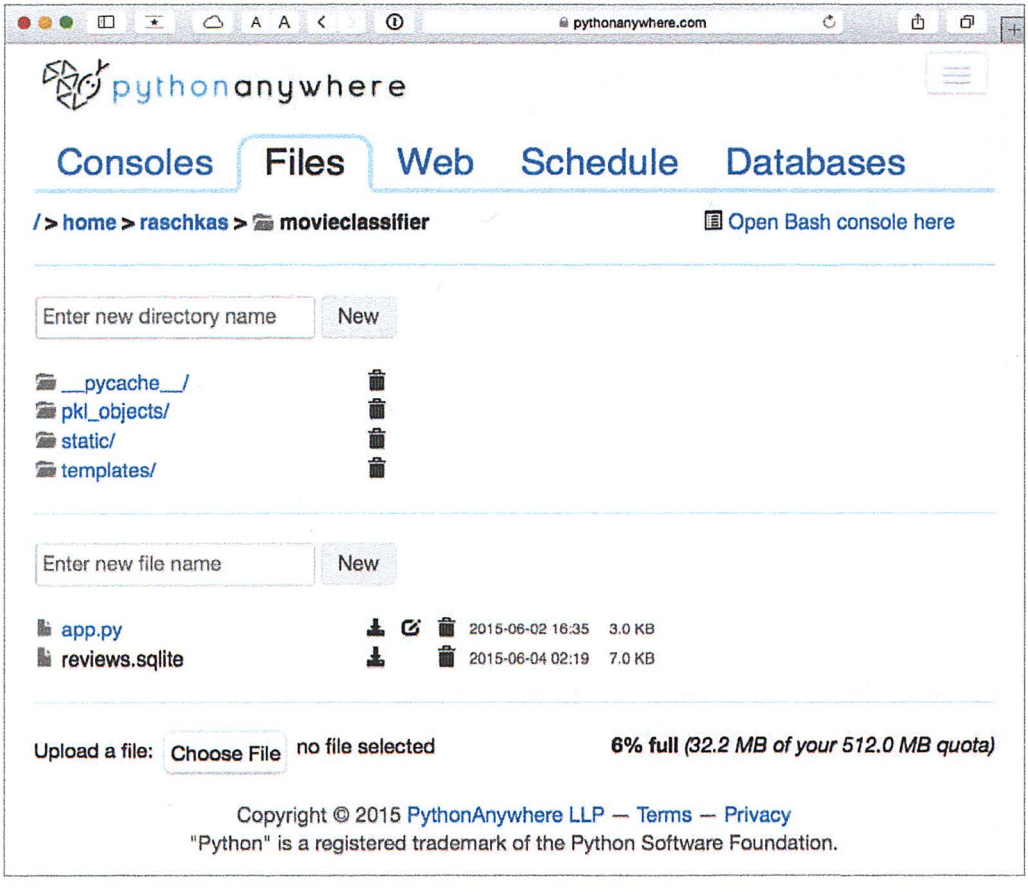
После того как мы проверили работу веб-приложения локально, мы теперь готовы развернуть наше веб-приложение на публичном сервере. В целях данного практического руководства мы воспользуемся службой веб-хостинга **PythonAnywhere**, которая специализируется на хостинге веб-приложений Python и делает его чрезвычайно простым и беспроблемным. Кроме того, веб-служба PythonAnywhere предлагает возможность создания аккаунта новичка, который позволяет бесплатно запускать одно веб-приложение.

Для того чтобы создать новый аккаунт веб-службы PythonAnywhere, надо зайти на веб-сайт https://www.pythonanywhere.com и нажать на ссылке **Pricing & signup** (Стоимость услуг и регистрация), расположенной в верхнем правом углу. Далее нажать на кнопке **Create a Beginner account** (Создать аккаунт новичка), где нужно ввести имя пользователя, пароль и действительный адрес электронной почты. После прочтения правил и ограничений и согласия с ними у нас должен быть новый аккаунт.

К сожалению, бесплатный аккаунт новичка не позволяет получать из нашего терминала или командной строки доступ к удаленному серверу по протоколу SSH. Поэтому для управления веб-приложением нам придется использовать веб-интерфейс веб-службы PythonAnywhere. Но прежде чем мы сможем выгрузить наши локальные файлы приложения на сервер, мы должны создать новое веб-приложение для нашего аккаунта PythonAnywhere. После нажатия на кнопке **Dashboard** (Личный кабинет) в верхнем правом углу мы получим доступ к панели управления, показанной в верхней части страницы. Затем нажимаем на вкладку **Web** (Веб-приложение), которая теперь видима в верхней части страницы. Далее нажимаем на кнопку **Add a new web app** (Добавить новое веб-приложение) слева, которая позволяет создать новое веб-приложение Flask на Python 3.5, которое мы назовем классификатором кинофильмов movieclassifier.

После создания нового приложения для нашего аккаунта PythonAnywhere мы направляемся на вкладку **Files** (Файлы) для выгрузки файлов из нашего локаль-

ного каталога movieclassifier на сервер с помощью веб-интерфейса веб-службы PythonAnywhere. После выгрузки файлов веб-приложения, которые мы создали локально на нашем компьютере, у нас должен быть каталог movieclassifier в нашем аккаунте PythonAnywhere. Он содержит те же каталоги и файлы, что и наш локальный каталог movieclassifier, как показано на следующем ниже снимке экрана:



В конце мы направляемся на вкладку **Web** еще раз и нажимаем на кнопку **Reload <username>.pythonanywhere.com** (Перезагрузить), чтобы распространить все изменения и освежить наше веб-приложение. В заключение наше веб-приложение должно теперь находиться в состоянии полной готовности к работе и быть общедоступным по адресу <username>.pythonanywhere.com.

К сожалению, веб-серверы могут быть довольно чувствительны к самым крошечным проблемам в веб-приложении. Если вы испытываете проблемы с управлением веб-приложением на веб-хостинге PythonAnywhere и получаете сообщения об ошибках в браузере, то можете проверить сервер и журналы ошибок, к которым можно получить доступ из вкладки **Web** в вашем аккаунте PythonAnywhere, с тем чтобы детально диагностировать проблему.

Обновление классификатора киноотзывов

Несмотря на то что наша прогнозная модель обновляется на лету каждый раз, когда пользователь предоставляет отклик о результате классификации, обновления в объекте clf будут сброшены в случае аварийного прекращения работы веб-сервера или его перезапуска. Если мы перезагрузим веб-приложение, то объект clf будет повторно инициализирован из файла консервации classifier.pkl. Один из вариантов перманентного закрепления обновлений состоит в том, чтобы консервировать объект clf повторно после каждого обновления. Однако это стало бы в вычислительном плане очень неэффективно с растущим числом пользователей и может повредить файл консервации, в случае если пользователи предоставят отклик о результате прогноза одновременно. Альтернативное решение состоит в том, чтобы обновлять прогнозную модель на основе данных откликов, накапливающихся в базе данных SQLite. Как вариант можно скачать базу данных SQLite с сервера PythonAnywhere, локально обновить объект clf на нашем компьютере и выгрузить новый файл консервации на веб-хостинг PythonAnywhere. Для обновления классификатора локально на нашем компьютере мы создаем в каталоге movieclassifier сценарный файл update.py со следующим ниже содержимым:

```
import pickle
import sqlite3
import numpy as np
import os

# импортировать HashingVectorizer из локального каталога
from vectorizer import vect

def update_model(db_path, model, batch_size=10000):
    conn = sqlite3.connect(db_path)
    c = conn.cursor()
    c.execute('SELECT * from review_db')

    results = c.fetchmany(batch_size)
    while results:
        data = np.array(results)
        X = data[:, 0]
        y = data[:, 1].astype(int)

        classes = np.array([0, 1])
        X_train = vect.transform(X)
        model.partial_fit(X_train, y, classes=classes)
        results = c.fetchmany(batch_size)

    conn.close()
    return model

cur_dir = os.path.dirname(__file__)
clf = pickle.load(open(os.path.join(cur_dir,
                                    'pkl_objects',
                                    'classifier.pkl'), 'rb'))
db = os.path.join(cur_dir, 'reviews.sqlite')

clf = update_model(db_path=db, model=clf, batch_size=10000)

# Раскомментировать следующие строки, если уверены, что
# желаете постоянно обновлять законсервированный файл
```

```
# classifier.pkl.
# pickle.dump(clf, open(os.path.join(cur_dir,
# 'pkl_objects', 'classifier.pkl'), 'wb')
# , protocol=4)
```

Функция update_model будет приносить записи из базы данных SQLite партиями по 10 000 записей за один раз, в случае если база данных не содержит меньше записей. Как вариант можно также приносить по одной записи за один раз, используя fetchone вместо fetchmany, что в вычислительном плане будет очень неэффективно. Использование альтернативного метода fetchall может стать проблематичным, если мы работаем с большими наборами данных, превышающими объем памяти компьютера или сервера.

Создав сценарий update.py, теперь можно тоже выгрузить его в каталог movieclassifier на веб-хостинге PythonAnywhere и импортировать функцию update_model в главном сценарии приложения app.py для обновления классификатора из базы данных SQLite всякий раз, когда мы перезапускаем веб-приложение. Для того чтобы это выполнить, нам просто нужно добавить в начало сценария app.py строку исходного кода с импортом функции update_model из сценария update.py:

```
# import update function from local dir
from update import update_model
```

Затем нужно вызвать функцию update_model в теле главного приложения:

```
...
if __name__ == '__main__':
    clf = update_model(db_path="db", model=clf, batch_size=10000)
...
```

Резюме

В этой главе вы изучили много полезных и практических тем, расширяющих наши познания теории машинного обучения. Вы узнали, как выполнять сериализацию модели после ее тренировки и как загружать ее для более поздних случаев использования. Кроме того, мы создали базу данных SQLite для эффективного хранения данных и создали веб-приложение, позволяющее сделать наш классификатор кинофильмов доступным для внешнего мира.

На протяжении всей книги мы подробно обсудили много идей и принципов машинного обучения, наиболее успешных практических методов и моделей с учителем для классификации данных. В следующей главе мы рассмотрим еще одну подкатегорию обучения с учителем – регрессионный анализ, который позволяет прогнозировать результирующие переменные в непрерывной шкале, в отличие от категориальных меток классов классификационных моделей, с которыми мы работали до сих пор.

Прогнозирование значений непрерывной целевой переменной на основе регрессионного анализа

В предыдущих главах вы много узнали об основных принципах, лежащих в основе обучения *с учителем*, и натренировали множество разных моделей для выполнения задач классификации с целью идентификации принадлежности группе или категориальных переменных. В этой главе мы окунемся в еще одну подкатегорию обучения с учителем: *регрессионный анализ*.

Регрессионные модели используются для предсказания целевых переменных в *непрерывной* шкале, что делает их привлекательными для решения многих вопросов в науке и для приложений в информационной отрасли, таких как понимание связей между переменными, оценивание тенденций или создание прогнозов. Одним из примеров может быть предсказание продаж компании в будущие месяцы.

В этой главе мы обсудим основные понятия регрессионных моделей и затронем следующие темы:

- разведочный анализ и визуализация наборов данных;
- рассмотрение разных подходов к реализации линейных регрессионных моделей;
- тренировка устойчивых к выбросам регрессионных моделей;
- оценивание регрессионных моделей и диагностирование типичных проблем;
- подгонка регрессионных моделей под нелинейные данные.

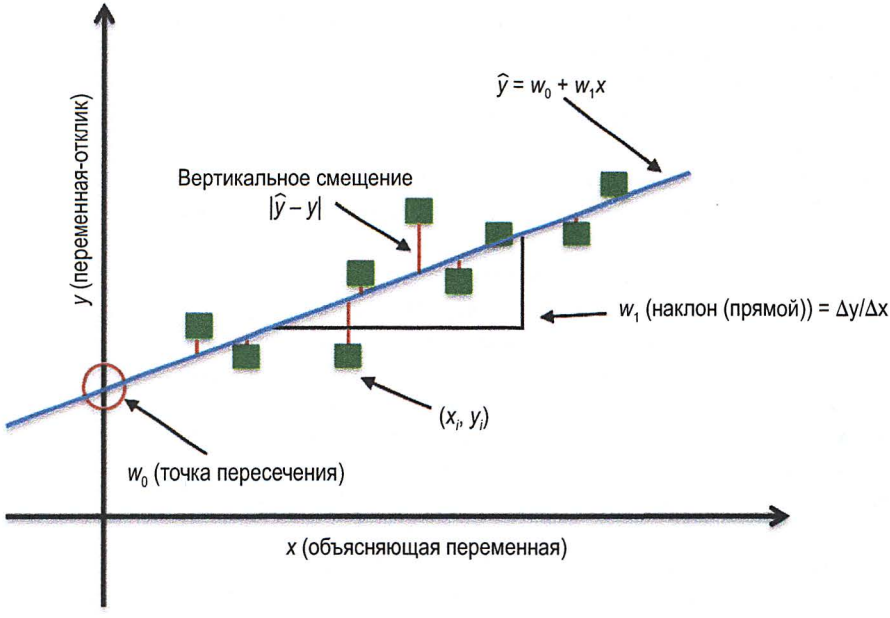
Введение в простую линейную регрессионную модель

Задача простой (одномерной) линейной регрессии состоит в том, чтобы смоделировать связь между единственным признаком (объясняющей переменной x) и непрерывнозначным *откликом* (целевой переменной y). Уравнение линейной модели с одной объясняющей переменной определяется следующим образом:

$$y = w_0 + w_1 x.$$

Здесь вес w_0 – это точка пересечения оси Y и w_1 – коэффициент объясняющей переменной. Наша задача – извлечь веса линейного уравнения, чтобы описать связь между объясняющей и целевой переменными, которую затем можно использовать для предсказания откликов новых объясняющих переменных, не бывших частью тренировочного набора данных.

Основываясь на линейном уравнении, определенном нами выше, линейная регрессия может пониматься как нахождение оптимально подогнанной прямой линии, проходящей через точки образцов данных, как показано на нижеследующем рисунке:



Оптимально подогнанная прямая линия также называется **линией регрессии**, а вертикальные прямые от линии регрессии до точек данных – это так называемые **смещения**, или **остатки**, – ошибки нашего предсказания.

Частный случай, состоящий из одной объясняющей переменной, называется **простой линейной регрессией**, но, разумеется, мы также можем обобщить линейную регрессионную модель на две и более объясняющих переменных. Отсюда этот процесс называется **множественной линейной регрессией**[1]:

$$y = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^T x.$$

Здесь w_0 – это точка пересечения оси Y при $x_0 = 1$.

Разведочный анализ набора данных Housing

Прежде чем реализовать нашу первую линейную регрессионную модель, введем новый набор данных – **набор данных жилищного фонда** Housing, содержащий ин-

[1] Для справки: уравнение простой регрессии имеет на правой стороне точку пересечения (w_0) и объясняющую переменную с коэффициентом наклона ($w_0 x$). В отличие от него, уравнение множественной регрессии имеет на правой стороне две или более объясняющих переменных, каждая со своим коэффициентом наклона. – *Прим. перев.*

формацию о зданиях в пригородах Бостона, собранную Д. Харрисоном и Д. Л. Рубинфельдом в 1978 г. для Бюро переписи населения США. *Набор данных Housing* находится в открытом доступе, и его можно скачать из *репозитория машинного обучения UCI* на https://archive.ics.uci.edu/ml/datasets/Housing.

Признаки (характеристики) 506 образцов резюмированы в следующей ниже выдержке из описания набора данных:

- **CRIM**: уровень преступности на душу населения по городу;
- **ZN**: доля жилых земельных участков, предназначенных для лотов свыше 25 000 кв. м футов;
- **INDUS**: доля акров нерозничного бизнеса в расчете на город;
- **CHAS**: фиктивная переменная реки Чарльз (= 1, если участок ограничивает реку; 0 в противном случае);
- **NOX**: концентрация окислов азота (частей на 10 млн);
- **RM**: среднее число комнат в жилом помещении;
- **AGE**: доля занимаемых владельцами единиц, построенных до 1940 г.;
- **DIS**: взвешенные расстояния до пяти Бостонских центров занятости;
- **RAD**: индекс доступности к радиальным шоссе;
- **TAX**: полная ставка налога на имущество на 10 тыс. долл.;
- **PTRATIO**: соотношение ученик-учитель по городу;
- **B:** вычисляется как *1000 (Bk – 0.63)^2*, где Bk – доля людей афроамериканского происхождения по городу;
- **LSTAT**: процент населения с более низким статусом;
- **MEDV**: медианная стоимость занимаемых владельцами домов в 1 тыс. долл.

В оставшейся части этой главы в качестве нашей целевой переменной – переменной, которую мы хотим предсказать с использованием одной или нескольких из этих 13 объясняющих переменных, мы рассмотрим цены на жилье (MEDV). Прежде чем мы продолжим разведку этого набора данных, занесем его в таблицу данных DataFrame библиотеки pandas прямо из репозитория UCI:

```
import pandas as pd
url = 'https://archive.ics.uci.edu/ml/machine-learningdatabases/housing/housing.data'
df = pd.read_csv(url, header=None, sep='\s+')
df.columns = ['CRIM', 'ZN', 'INDUS', 'CHAS',
              'NOX', 'RM', 'AGE', 'DIS', 'RAD',
              'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
df.head()
```

Чтобы удостовериться, что набор данных был загружен успешно, мы показали первые пять строк набора данных, как проиллюстрировано на следующем ниже снимке экрана:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	MEDV
0	0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

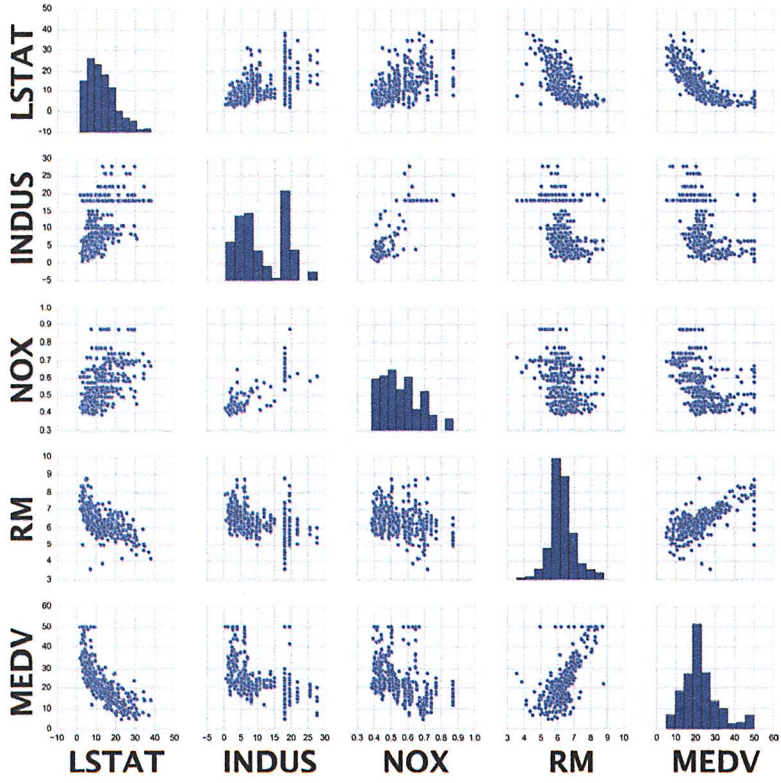
Визуализация важных характеристик набора данных

Разведочный анализ данных (exploratory data analysis, EDA) – это существенный первый шаг, рекомендуемый для выполнения перед тренировкой машиннообучаемой модели. В остальной части этого раздела мы воспользуемся несколькими простыми и одновременно полезными методами из графического инструментария разведочного анализа данных (EDA), которые способны помочь визуально установить наличие выбросов, распределение данных и связей между признаками.

Сначала создадим *матрицу точечных графиков*, позволяющую в одном месте визуализировать в этом наборе данных попарные корреляции между разными признаками. Для подготовки матрицы точечных графиков воспользуемся функцией pairplot из библиотеки Python seaborn (http://stanford.edu/~mwaskom/software/seaborn/), которая разработана на основе библиотеки matplotlib для построения статистических графиков:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style='whitegrid', context='notebook')
cols = ['LSTAT', 'INDUS', 'NOX', 'RM', 'MEDV']
sns.pairplot(df[cols], size=2.5)
plt.show()
```

Как видно на следующей ниже диаграмме, матрица точечных графиков предоставляет полезное графическое резюме связей, присутствующих в наборе данных:



Импортирование библиотеки seaborn изменит в текущем сеансе Python эстетику, применяющуюся в matplotlib по умолчанию. Если вы не хотите использовать стилевые настройки библиотеки seaborn, то можете вернуться к настройкам библиотеки matplotlib, выполнив следующую команду:

```
sns.reset_orig()
```

Ввиду ограниченности свободного пространства и в целях удобочитаемости мы построили графики лишь пяти столбцов набора данных: **LSTAT**, **INDUS**, **NOX**, **RM** и **MEDV**. Впрочем, в целях дальнейшего разведочного анализа вы вполне можете сами создать матрицу точечных графиков всей таблицы данных DataFrame.

Используя построенную матрицу точечных графиков, теперь можно быстро прикинуть на глаз, каким образом данные распределены и содержат ли они выбросы. Например, мы видим, что существует линейная связь между количеством комнат **RM** и ценами на жилье **MEDV** (пятый столбец четвертой строки). Кроме того, на гистограмме видно (нижний правый подграфик в матрице точечных графиков), что переменная **MEDV**, похоже, нормально распределена, но содержит несколько выбросов.

Отметим, что вопреки общепринятому мнению во время тренировки линейной регрессионной модели не требуется, чтобы объясняющие либо целевые переменные были нормально распределены. Допущение о нормальности распределения является необходимым условием для определенных статистических тестов и статистических проверок гипотез, рассмотрение которых выходит за рамки этой книги (Montgomery D. C., Peck E. A. and Vining G. G., «Introduction to linear regression analysis». John Wiley and Sons, 2012, p. 318–319 (*«Введение в линейный регрессионный анализ»*)).

Для определения количества линейной связи между признаками теперь создадим корреляционную матрицу. Корреляционная матрица тесно связана с ковариационной матрицей, которую мы видели в разделе об **анализе главных компонент** (PCA) в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»*. Корреляционную матрицу можно интуитивно интерпретировать как приведенную версию ковариационной матрицы. Фактически корреляционная матрица идентична ковариационной матрице, вычисленной на основе стандартизированных данных.

Корреляционная матрица – это квадратная матрица, которая содержит **линейные коэффициенты корреляции Пирсона** (Pearson product-moment correlation coefficients, или PPMCC, часто сокращенно **r Пирсона**), которые измеряют линейную зависимость между парами признаков. Коэффициенты корреляции ограничены диапазоном [−1, 1]. Два признака имеют соответственно абсолютно положительную корреляцию, если $r = 1$, никакой корреляции, если $r = 0$, и абсолютно отрицательную корреляцию, если $r = -1$. Как упомянуто ранее, коэффициент корреляции Пирсона можно вычислить просто как ковариацию между двумя признаками x и y – числитель, деленный на произведение их стандартных отклонений (знаменатель):

$$r = \frac{\sum_{i=1}^{n}\left[(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)\right]}{\sqrt{\sum_{i=1}^{n}(x^{(i)} - \mu_x)^2}\sqrt{\sum_{i=1}^{n}(y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}.$$

Здесь μ обозначает эмпирическое среднее соответствующего признака, σ_{xy} – ковариацию между признаками x и y и соответственно σ_x и σ_y – стандартные отклонения признаков.

Можно продемонстрировать, что ковариация между стандартизированными признаками фактически равна их коэффициенту линейной корреляции.
Сначала стандартизируем признаки x и y, чтобы получить их z-оценки[1], которые мы обозначим соответственно как x' и y':

$$x' = \frac{x - \mu_x}{\sigma_x},\ y' = \frac{y - \mu_y}{\sigma_y}.$$

Напомним, что мы вычисляем (генеральную) ковариацию между двумя признаками следующим образом:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}(x^{(i)} - \mu_x)(y^{(i)} - \mu_y).$$

Поскольку стандартизация центрирует переменную признака в нулевом среднем значении, мы можем теперь вычислить ковариацию между масштабированными признаками следующим образом:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}(x' - 0)(y' - 0).$$

За счет повторной подстановки получаем следующий результат:

$$\frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right);$$

$$\frac{1}{n \cdot \sigma_x \sigma_y}\sum_{i}^{n}(x^{(i)} - \mu_x)(y^{(i)} - \mu_y).$$

Его можно упростить таким образом:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}.$$

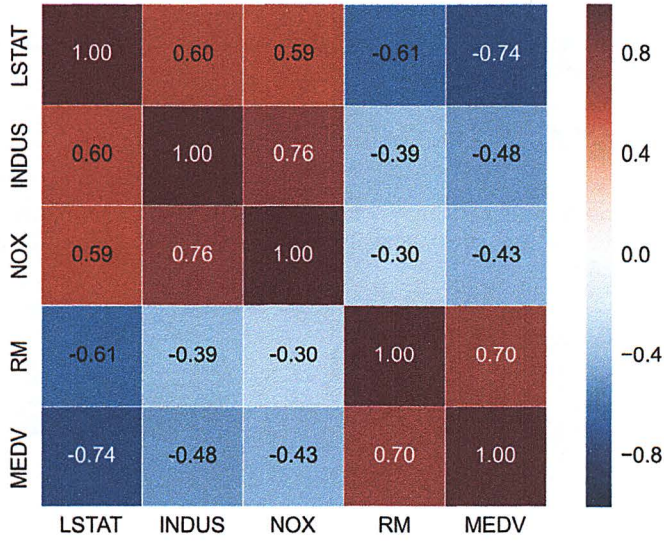
В следующем ниже примере мы воспользуемся функцией corrcoef библиотеки NumPy на пяти столбцах признаков, которые мы ранее визуализировали в матрице точечных графиков, при этом воспользуемся функцией heatmap библиотеки seaborn для подготовки массива корреляционных матриц в виде теплокарты:

```
import numpy as np
cm = np.corrcoef(df[cols].values.T)
sns.set(font_scale=1.5)
hm = sns.heatmap(cm,
                 cbar=True,
                 annot=True,
```

[1] Для справки: z-оценка, или стандартная оценка, позволяет определить меру отклонения величины от среднего, т. е. показываает, на сколько стандартных отклонений от среднего лежит выбранное значение переменной. – *Прим. перев.*

```
                    square=True,
                    fmt='.2f',
                    annot_kws={'size': 15},
                    yticklabels=cols,
                    xticklabels=cols)
plt.show()
```

Как видно на получившемся рисунке, корреляционная матрица предоставляет нам еще одну итоговую диаграмму, которая способна помочь отобрать признаки, основываясь на соответствующих им линейных корреляциях:



Для того чтобы выполнить подгонку линейной регрессионной модели, нас интересуют те признаки, которые имеют высокую корреляцию с нашей целевой переменной **MEDV**. Глядя на приведенную выше корреляционную матрицу, мы видим, что наша целевая переменная **MEDV** показывает наибольшую корреляцию с переменной **LSTAT** (–0.74). Однако, как вы помните из матрицы точечных графиков, имеется явная нелинейная связь между **LSTAT** и **MEDV**. С другой стороны, корреляция между **RM** и **MEDV** тоже относительно высока (0.70), и при наличии линейной связи между этими двумя переменными, которые мы наблюдали в матрице точечных графиков, переменная **RM** кажется хорошим вариантом в качестве объясняющей переменной, которая понадобится в следующем разделе во время ознакомления с принципами работы простой линейной регрессионной модели.

Реализация линейной регрессионной модели обычным методом наименьших квадратов

В начале этой главы мы обсудили, что линейная регрессия может пониматься как нахождение оптимально подогнанной прямой линии, проходящей через точки образцов тренировочных данных. Однако мы не определили ни термина *оптимальная*

подгонка, ни обсудили различную методику подгонки такой модели. В следующих подразделах мы заполним недостающие части этой головоломки при помощи **обычного метода наименьших квадратов** (обычного МНК, ordinary least squares, OLS), применяемого для оценки параметров линии регрессии, минимизирующих сумму квадратичных вертикальных расстояний (остатков или ошибок) до точек образцов.

Решение уравнения регрессии для параметров регрессии методом градиентного спуска

Рассмотрим нашу реализацию **адаптивного линейного нейрона** (ADALINE) из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*; мы помним, что искусственный нейрон использует линейную функцию активации, и мы определили функцию стоимости $J(\cdot)$, которую мы минимизировали для извлечения весов благодаря алгоритмам оптимизации, таким как **градиентный спуск** (GD) и **стохастический градиентный спуск** (SGD). Этой функцией стоимости в ADALINE является **сумма квадратичных ошибок** (sum of squared errors, SSE). Она идентична функции стоимости по МНК, которую мы определили:

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2.$$

Здесь $\hat{y}$ – это предсказанное значение $\hat{y} = w^T x$ (отметим, что член 1/2 используется просто для удобства получения правила обновления при градиентном спуске). По существу, линейная регрессия по МНК может пониматься как ADALINE без единичной ступенчатой функции, в результате чего вместо меток классов –1 и 1 мы получаем непрерывные целевые значения. В качестве демонстрации подобия возьмем реализацию градиентного спуска для ADALINE из *главы 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* и удалим единичную ступенчатую функцию, чтобы реализовать нашу первую линейную регрессионную модель:

```
class LinearRegressionGD(object):

    def __init__(self, eta=0.001, n_iter=20):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

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

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

    def predict(self, X):
        return self.net_input(X)
```

Если вам нужно освежить память по поводу того, каким образом обновляются веса, – делая шаг в противоположном от градиента направлении, – пожалуйста, обратитесь к разделу об ADALINE в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*.

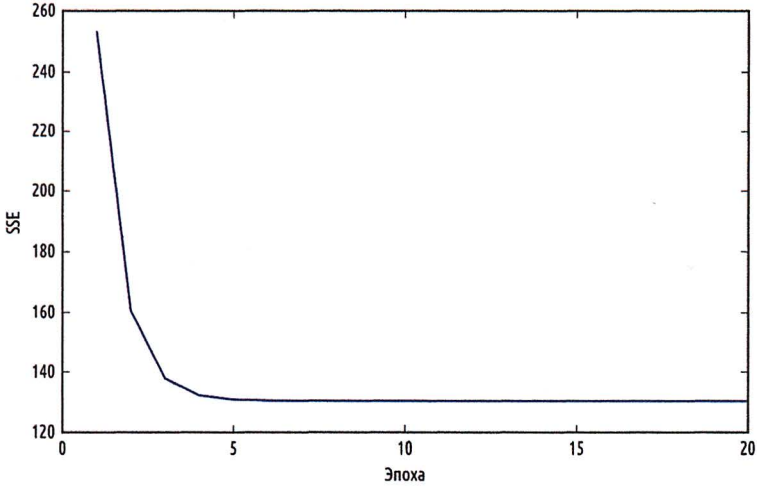
Чтобы увидеть наш линейный регрессор LinearRegressionGD в действии, воспользуемся переменной RM (число комнат) из набора данных жилищного фонда в качестве объясняющей переменной для тренировки модели, которая может предсказывать MEDV (цены на жилье). Более того, для лучшей сходимости алгоритма градиентного спуска мы стандартизируем переменные. Соответствующий исходный код выглядит следующим образом:

```
X = df[['RM']].values
y = df['MEDV'].values

from sklearn.preprocessing import StandardScaler
sc_x = StandardScaler()
sc_y = StandardScaler()
X_std = sc_x.fit_transform(X)
y_std = sc_y.fit_transform(y)
lr = LinearRegressionGD()
lr.fit(X_std, y_std)
```

В *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* мы упомянули, что всегда, когда мы используем алгоритмы оптимизации, такие как градиентный спуск, неплохо построить график стоимости как функции от числа эпох (проходов по тренировочному набору данных) для проверки на сходимость. Иными словами, теперь построим график стоимости в сопоставлении с числом эпох, чтобы проверить сходимость линейной регрессии:

```
plt.plot(range(1, lr.n_iter+1), lr.cost_)
plt.ylabel('SSE')  # сумма квадратичных ошибок
plt.xlabel('Эпоха')
plt.show()
```

Как видно на следующем ниже графике, алгоритм ГС сходился после пятой эпохи:



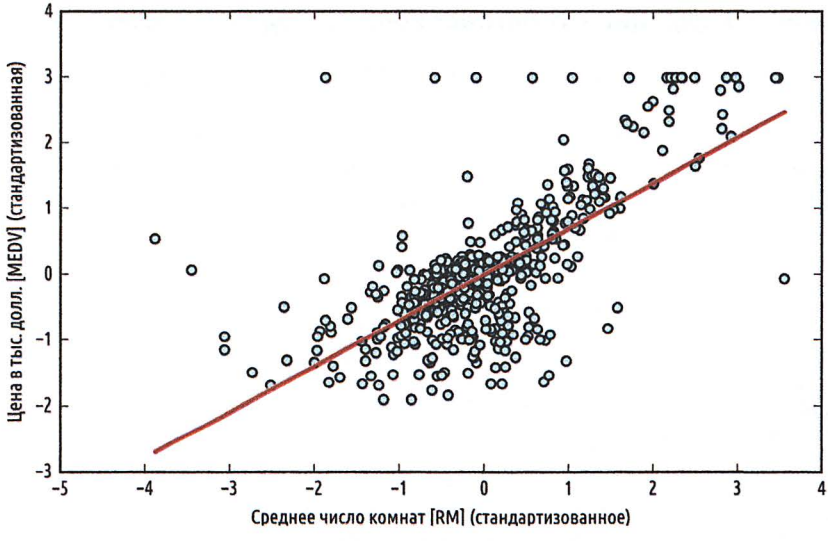
Далее посмотрим, насколько хорошо линия линейной регрессии подогнана под тренировочные данные. Для этого определим простую вспомогательную функцию, которая построит точечный график тренировочных образцов и добавит линию регрессии:

```
def lin_regplot(X, y, model):
    plt.scatter(X, y, c='blue')
    plt.plot(X, model.predict(X), color='red')
    return None
```

Теперь воспользуемся функцией lin_regplot, чтобы построить график числа комнат в сопоставлении с ценами на жилье:

```
lin_regplot(X_std, y_std, lr)
plt.xlabel('Среднее число комнат [RM] (стандартизованное)')
plt.ylabel('Цена в тыс.долл. [MEDV] (стандартизованная)')
plt.show()
```

Как видно на следующем ниже графике, линия линейной регрессии отражает общий тренд, что цены на жилье имеют тенденцию увеличиваться вместе с числом комнат:



Несмотря на то что это наблюдение имеет интуитивный смысл, во многих случаях данные также говорят о том, что число комнат не объясняет цены на жилье очень хорошо. Позже в этой главе мы обсудим, каким образом количественно определить качество работы регрессионной модели. Примечательно, что мы также наблюдаем любопытную линию $y = 3$, которая предполагает, что цены, возможно, были подрезаны. В определенных приложениях бывает важным также сообщить о предсказанных результирующих переменных в их первоначальной шкале. Чтобы прошкалировать назад предсказанный ценовой исход на осях **Цена в тыс. долл.**, можно просто применить метод обратной трансформации inverse_transform класса-преобразователя стандартных шкал StandardScaler:

```
>>>
num_rooms_std = sc_x.transform([5.0])
price_std = lr.predict(num_rooms_std)
print("Цена в тыс. долл.: %.3f" % sc_y.inverse_transform(price_std))

Цена в тыс.долл.: 10.840
```

В приведенном выше примере исходного кода мы использовали ранее натренированную линейную регрессионную модель для предсказания цены дома с пятью комнатами. Согласно нашей модели, такой дом стоит 10 840 долл.

Попутно также стоит заметить, что, строго говоря, нам не обязательно обновлять веса точки пересечения, если мы работаем со стандартизированными переменными, поскольку в этих случаях точка пересечения оси *Y* всегда равна 0. Мы можем быстро подтвердить это, распечатав веса:

```
>>>
print('Наклон: %.3f' % lr.w_[1])
Наклон: 0.695

print('Пересечение: %.3f' % lr.w_[0])
Пересечение: -0.000
```

Оценивание коэффициента регрессионной модели в scikit-learn

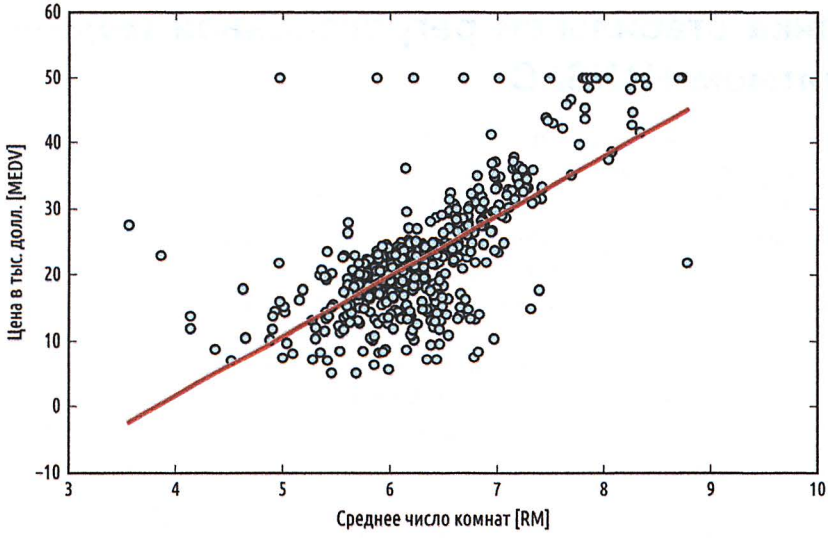
В предыдущем разделе мы реализовали действующую модель для регрессионного анализа. Однако в реальном приложении мы можем быть заинтересованы в более эффективных реализациях, например в объекте линейной регрессии LinearRegression библиотеки scikit-learn, в котором используются динамическая библиотека **LIBLINEAR** и продвинутые алгоритмы оптимизации, которые лучше работают с нестандартизированными переменными. Для определенных приложений иногда это желательно:

```
>>>
from sklearn.linear_model import LinearRegression
slr = LinearRegression()
slr.fit(X, y)
print('Наклон: %.3f' % slr.coef_[0])
Наклон: 9.102
print('Пересечение: %.3f' % slr.intercept_)
Пересечение: -34.671
```

Как видно в результате выполнения предыдущего исходного кода, модель LinearRegression библиотеки scikit-learn, подогнанная нестандартизированными переменными **RM** и **MEDV**, выдала другие модельные коэффициенты. Сравним ее с нашей собственной реализацией на основе градиентного спуска, построив график MEDV в сопоставлении с RM:

```
lin_regplot(X, y, slr)
plt.xlabel('Среднее число комнат [RM]')
plt.ylabel('Цена в тыс. долл. [MEDV]')
plt.show()
```

Построив график тренировочных данных и выполнив подгонку модели путем выполнения вышеприведенного исходного кода, теперь видно, что общий результат выглядит идентично нашей реализации на основе градиентного спуска:



Как альтернатива использованию библиотек машинного обучения также существует решение в замкнутой форме (аналитическое) для решения МНК с применением системы линейных уравнений, которое можно найти в большинстве учебников по основам математической статистики:

$$w = (X^T X)^{-1} X^T y.$$

Мы можем реализовать его на Python следующим образом:

```
# добавляем вектор-столбец из "единиц"
Xb = np.hstack((np.ones((X.shape[0], 1)), X))
w = np.zeros(X.shape[1])
z = np.linalg.inv(np.dot(Xb.T, Xb))
w = np.dot(z, np.dot(Xb.T, y))

print('Наклон: %.3f' % w[1])
Наклон: 9.102

print('Пересечение: %.3f' % w[0])
Пересечение: -34.671
```

Преимущество этого метода состоит в том, что он гарантированно находит оптимальное решение аналитически. Однако если мы работаем с очень большими наборами данных, инвертирование матрицы в этой формуле (иногда также называемой **нормальным уравнением**) может быть в вычислительном плане слишком затратным, либо матрица образцов может быть сингулярной (необратимой), вследствие этого в определенных случаях мы можем предпочесть итеративные методы.

Если вы интересуетесь дополнительной информацией о том, как получить нормальные уравнения, рекомендуем взглянуть на главу из лекций доктора Стивена Поллока «*Классическая линейная регрессионная модель*» в Лестерском университете, которые доступны бесплатно по прямой ссылке http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf.

Подгонка стабильной регрессионной модели алгоритмом RANSAC

Выбросы могут оказывать сильное воздействие на линейные регрессионные модели. В определенных ситуациях незначительное подмножество наших данных может иметь большой эффект на оцениваемые модельные коэффициенты. Для обнаружения выбросов используются разнообразные статистические тесты, рассмотрение которых выходит за рамки данной книги. Однако во время удаления выбросов всегда требуется наше собственное суждение как аналитика данных, а также наше знание предметной области.

В качестве альтернативы исключению выбросов рассмотрим устойчивый метод регрессии с использованием алгоритма **RANSAC** (RANdom SAmple Consensus, т. е. консенсус на основе случайных образцов), который выполняет подгонку регрессионной модели на подмножестве данных, так называемых *не-выбросах* (inliers), т. е. хороших точках данных.

Итеративный алгоритм RANSAC можно резюмировать следующим образом.

1. Выбрать случайное число образцов в качестве не-выбросов и выполнить подгонку модели.
2. Проверить все остальные точки данных на подогнанной модели и добавить те точки, которые попадают в пределы заданного пользователями допуска для не-выбросов.
3. Выполнить повторную подгонку модели с использованием всех не-выбросов.
4. Оценить ошибку подогнанной модели относительно не-выбросов.
5. Завершить алгоритм, в случае если качество соответствует определенному заданному пользователем порогу либо если было достигнуто фиксированное число итераций; в противном случае вернуться к шагу 1.

Теперь обернем нашу линейную модель в алгоритм RANSAC, воспользовавшись для этого объектом RANSACRegressor библиотеки scikit-learn:

```
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor(LinearRegression(),
                         max_trials=100,
                         min_samples=50,
                         residual_metric=lambda x: np.sum(np.abs(x), axis=1),
                         residual_threshold=5.0,
                         random_state=0)
ransac.fit(X, y)
```

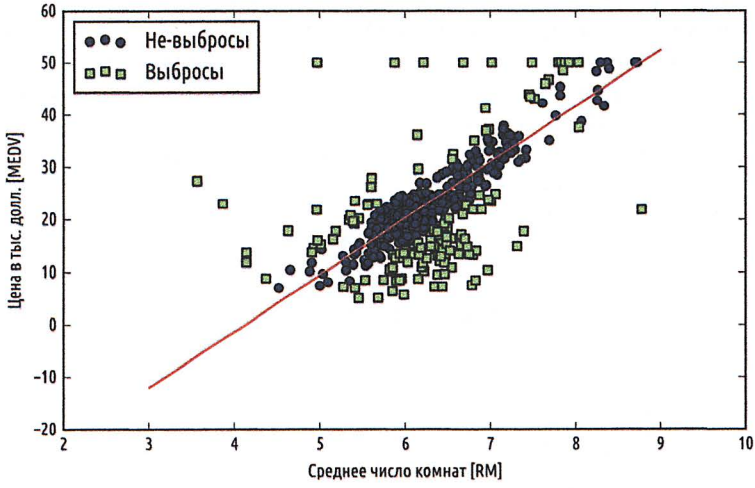
Мы задаем максимальное число итераций RANSACRegressor равным 100 и параметром min_samples=50 устанавливаем минимальное число случайно отобранных образцов равным как минимум 50. Используя параметр метрики остатков residual_metric, мы передали вызываемую лямбда-функцию (lambda), которая просто рассчитывает абсолютные вертикальные расстояния между подогнанной линией и точками образцов. Задав параметр residual_threshold равным 5.0, мы разрешили включать в подмножество не-выбросов образцы с вертикальным расстоянием до подогнанной линии только в пределах 5 единиц расстояния, что хорошо работает на этом отдельно взятом наборе данных. По умолчанию в библиотеке scikit-learn для отбора порога не-выбросов используется оценка на основе медианного абсолютно-

го отклонения (оценка MAD)[1] целевых значений y, где MAD – это аббревиатура для Median Absolute Deviation. Однако выбор надлежащего значения для порога не-выбросов зависит от конкретной задачи, в чем кроется один из недостатков алгоритма RANSAC. За последние годы для автоматического отбора хорошего порога не-выбросов было разработано множество разных подходов. Вы можете найти детальное обсуждение данной темы в: R. Toldo and A. Fusiello's, «Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting». In Image Analysis and Processing-ICIAP 2009, p. 123–131. Springer, 2009 (*«Автоматическая оценка порога не-выбросов для робастной подгонки множественных структур»*).

После подгонки модели RANSAC получим не-выбросы и выбросы из подогнанной линейной регрессионной модели RANSAC и построим совместный график с линейной подгонкой:

```
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)
line_X = np.arange(3, 10, 1)
line_y_ransac = ransac.predict(line_X[:, np.newaxis])
plt.scatter(X[inlier_mask], y[inlier_mask],
            c='blue', marker='o', label='Не-выбросы')
plt.scatter(X[outlier_mask], y[outlier_mask],
            c='lightgreen', marker='s', label='Выбросы')
plt.plot(line_X, line_y_ransac, color='red')
plt.xlabel('Среднее число комнат [RM]')
plt.ylabel('Цена в тыс. долл. [MEDV]')
plt.legend(loc='upper left')
plt.show()
```

Как видно на следующем ниже точечном графике, линейная регрессионная модель была подогнана на обнаруженном подмножестве не-выбросов, показанных как круги:



[1] Медианное абсолютное отклонение вычисляется как MAD = median($|x_i - \tilde{x}|$), где x_i – это абсолютное значение точки данных, $\tilde{x}$ – медианное значение группы. Используется вместо среднего отклонения, когда крайние значения из области отклонений должны оказывать меньшее влияние на величину отклонения в силу того, что медиана затрагивается крайними значениями области отклонений в меньшей степени, чем среднее. – *Прим. перев.*

Когда мы распечатаем наклон (угловой коэффициент) и точку пересечения модели, выполнив нижеследующий исходный код, то увидим, что линия линейной регрессии немного отличается от подгонки, которую мы получили в предыдущем разделе без модели RANSAC:

```
>>>
print('Наклон: %.3f' % ransac.estimator_.coef_[0])
Наклон: 9.621

print('Пересечение: %.3f' % ransac.estimator_.intercept_)
Пересечение: -37.137
```

Используя модель RANSAC, мы уменьшили в этом наборе данных потенциальное влияние выбросов, но мы не знаем, имеет ли этот подход положительное влияние на предсказательную способность на ранее не встречавшихся данных. Поэтому в следующем разделе мы обсудим задачу оценивания регрессионной модели для разных подходов, являющуюся ключевой составной частью конструирования систем, предназначенных для прогнозного моделирования.

Оценивание качества работы линейных регрессионных моделей

В предыдущем разделе мы рассмотрели вопрос подгонки регрессионной модели на тренировочных данных. Однако в предыдущих главах вы узнали, что крайне важно протестировать модель на данных, которые она не видела во время тренировки, с целью получить несмещенную оценку качества ее работы.

Как мы помним из *главы 6 «Анализ наиболее успешных приемов оценивания моделей и тонкой настройки гиперпараметров»*, мы делим наш набор данных на раздельные тренировочный и тестовый наборы данных, где первый мы используем для подгонки модели и второй – для оценки ее качества по обобщению на ранее не встречавшиеся данные. Вместо того чтобы продолжить работу с простой регрессионной моделью, теперь задействуем все переменные набора данных и натренируем множественную регрессионную модель:

```
from sklearn.model_selection import train_test_split
X = df.iloc[:, :-1].values
y = df['MEDV'].values
X_train, X_test, y_train, y_test = train_test_split(
      X, y, test_size=0.3, random_state=0)
slr = LinearRegression()
slr.fit(X_train, y_train)
y_train_pred = slr.predict(X_train)
y_test_pred = slr.predict(X_test)
```

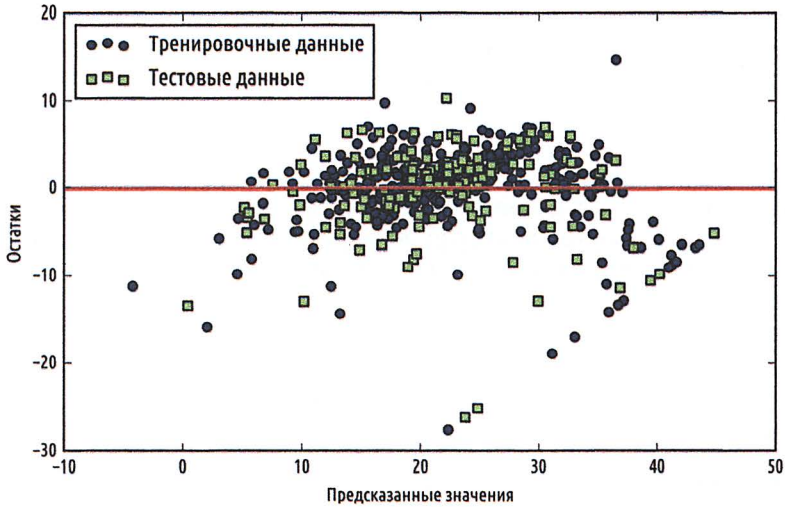
Учитывая, что наша модель использует две и более объясняющих переменных, мы не можем визуально представить линию линейной регрессии (или гиперплоскость, если быть точным) на двумерном графике, но мы можем вывести на график остатки (разницы или вертикальные расстояния между фактическими и предсказанными значениями) в сопоставлении с предсказанными значениями, чтобы диагностировать нашу регрессионную модель. Такие **графики остатков** широко применяются в графи-

ческом анализе для диагностирования регрессионных моделей с целью обнаружения нелинейности и выбросов, а также проверки случайности распределения ошибок.

При помощи приведенного ниже исходного кода теперь построим график остатков, где мы просто вычтем истинные целевые переменные из наших предсказанных откликов:

```
plt.scatter(y_train_pred, y_train_pred - y_train,
            c='blue', marker='o', label='Тренировочные данные')
plt.scatter(y_test_pred, y_test_pred - y_test,
            c='lightgreen', marker='s', label='Тестовые данные')
plt.xlabel('Предсказанные значения')
plt.ylabel('Остатки')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
plt.xlim([-10, 50])
plt.show()
```

После выполнения исходного кода мы должны увидеть график остатков с линией, проходящей через начало отсчета оси *X*, как показано ниже:



В случае идеального предсказания остатки были бы строго нулями, с чем в реальных и практических приложениях мы, вероятно, никогда не столкнемся. Однако от хорошей регрессионной модели мы ожидаем, что ошибки распределены случайно, а остатки случайно разбросаны вокруг средней линии. Если мы видим на графике остатков повторяющиеся образы, то, значит, наша модель неспособна захватить некоторую объяснительную информацию, что проявилось в остатках, как мы можем едва заметить на нашем предыдущем графике остатков. Более того, мы также можем использовать графики остатков для обнаружения выбросов, которые представлены точками с большим отклонением от средней линии.

Еще одной полезной количественной мерой оценки качества модели является так называемая **средневзвешенная квадратичная ошибка** (mean squared error, MSE), т. е. просто усредненное значение функции стоимости SSE, которую мы минимизируем для подгонки линейной регрессионной модели. MSE полезна для сравнения

разных регрессионных моделей или для тонкой настройки их параметров путем поиска по сетке параметров и перекрестной проверки:

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2.$$

Выполним следующий ниже исходный код:

```
>>>
from sklearn.metrics import mean_squared_error
print('MSE тренировка: %.3f, тестирование: %.3f' % (
    mean_squared_error(y_train, y_train_pred),
    mean_squared_error(y_test, y_test_pred)))
```

Мы увидим, что MSE на тренировочном наборе равна 19.96, а MSE тестового набора намного больше со значением 27.20, что указывает на то, что наша модель переподогнана под тренировочные данные.

Иногда может быть более целесообразным сообщить о коэффициенте детерминации (R-квадрате, или R^2), который может пониматься как стандартизированная версия MSE в целях получения лучшей интерпретируемости качества модели. Другими словами, R^2 – это доля дисперсии отклика, которая охвачена моделью. Значение R^2 определяется следующим образом:

$$R^2 = 1 - \frac{SSE}{SST}.$$

Здесь SSE – это сумма квадратичных ошибок, SST – полная сумма квадратов, или, другими словами, это просто дисперсия переменной отклика. Быстро покажем, что R^2 – это на самом деле просто приведенная версия MSE:

$$R^2 = 1 - \frac{SSE}{SST};$$

$$1 - \frac{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \mu_y)^2};$$

$$1 - \frac{MSE}{\mathrm{Var}(y)}.$$

Для тренировочного набора данных коэффициент детерминации R^2 ограничен значениями между 0 и 1, но может стать отрицательным для тестового набора. Если $R^2 = 1$, то модель идеально аппроксимирует данные с соответствующим $MSE = 0$.

После оценивания на тренировочных данных коэффициент детерминации R^2 нашей модели составляет 0.765, что является неплохим результатом. Однако R^2 на тестовом наборе данных составил всего 0.673, который можно рассчитать, выполнив следующий ниже фрагмент исходного кода:

```
>>>
from sklearn.metrics import r2_score
print('R^2 тренировка: %.3f, тестирование: %.3f' %
      (r2_score(y_train, y_train_pred),
       r2_score(y_test, y_test_pred)))
```

Применение регуляризованных методов для регрессии

Как осуждалось в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, регуляризация – это один из подходов к решению проблемы переобучения путем добавления дополнительной информации и тем самым сжатия значений параметров модели, чтобы вызвать штраф за сложность. Самыми популярными подходами к регуляризованной линейной регрессии являются так называемый метод **гребневой регрессии** (ridge regression), метод **lasso** (оператор наименьшего абсолютного стягивания и отбора, least absolute shrinkage and selection operator, lasso) и метод **эластичной сети** (elastic net)[1].

Гребневая регрессия – это модель с L2-штрафом, где к нашей функции стоимости на основе МНК мы просто добавляем квадратичную сумму весов:

$$J(w)_{\text{Гребень}} = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda\|w\|_2^2.$$

Здесь

$$L2: \quad \lambda\|w\|_2^2 = \lambda\sum_{j=1}^{m} w_j^2.$$

Увеличивая значение гиперпараметра λ, мы увеличиваем силу регуляризации и стягиваем веса нашей модели. Отметим, что мы не регуляризуем член уравнения для точки пересечения w_0.

Альтернативный подход, который может привести к разреженным моделям, представлен методом lasso. В зависимости от силы регуляризации определенные веса могут стать нулевыми, что делает метод lasso пригодным для применения в качестве метода отбора признаков с учителем:

$$J(w)_{\text{Лассо}} = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda w_1.$$

Здесь

$$L1: \quad \lambda\|w\|_1 = \lambda\sum_{j=1}^{m}|w_j|.$$

Однако ограничение метода lasso состоит в том, что он отбирает не более n переменных, если $m > n$. Компромиссом между гребневой регрессией и методом lasso является эластичная сеть, при которой имеются L1-штраф для генерирования разреженности и L2-штраф для преодоления некоторых ограничений метода lasso, таких как число отобранных переменных.

---

[1] Метод эластичной сети – это метод регуляризованной регрессии, компенсирующий недостатки метода lasso добавлением L1- и L2-штрафов. – *Прим. перев.*

$$J(w)_{\text{ЭластичнаяСеть}} = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda_1 \sum_{j=1}^{m} w_j^2 + \lambda_2 \sum_{j=1}^{m} |w_j|.$$

Все эти модели регуляризованной регрессии имеются в библиотеке scikit-learn, и их использование подобно обычной регрессионной модели, за исключением того, что нам нужно указать силу регуляризации в параметре λ, например оптимизированным методом *k*-блочной перекрестной проверки.

Модель гребневой регрессии можно инициализировать следующим образом:

```
from sklearn.linear_model import Ridge
ridge = Ridge(alpha=1.0)
```

Отметим, что сила регуляризации регулируется параметром alpha, который аналогичен параметру λ. Схожим образом можно инициализировать lasso-регрессор из подмодуля linear_model:

```
from sklearn.linear_model import Lasso
lasso = Lasso(alpha=1.0)
```

И наконец, реализация регрессии по методу эластичной сети ElasticNet позволяет варьировать соотношение L1 к L2:

```
from sklearn.linear_model import ElasticNet
lasso = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Например, если установить l1_ratio равным 1.0, то регрессор эластичной сети ElasticNet будет равен lasso. По поводу более подробной информации о разных реализациях линейной регрессии, пожалуйста, обратитесь к документации на http://scikit-learn.org/stable/modules/linear_model.html.

Превращение линейной регрессионной модели в криволинейную – полиномиальная регрессия

В предыдущих разделах мы допустили наличие между объясняющей переменной и переменной отклика линейной связи. Один из способов объяснить нарушение допущения о линейности связи состоит в том, чтобы использовать модель полиномиальной регрессии, добавив в уравнение полиномиальные члены:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_d x^d.$$

Здесь d обозначает степень полинома. Несмотря на то что мы можем использовать полиномиальную регрессию для моделирования нелинейных связей, она по-прежнему рассматривается как модель множественной линейной регрессии, ввиду линейных коэффициентов регрессии w.

Теперь обсудим вопрос использования класса-преобразователя для полиномиальных признаков PolynomialFeatures библиотеки scikit-learn с целью добавления в задачу простой регрессии с одной объясняющей переменной квадратичного члена ($d = 2$) и сравнения полинома с линейной подгонкой. Соответствующие шаги показаны ниже.

1. Добавить член с полиномом второй степени:

```
from sklearn.preprocessing import PolynomialFeatures
X = np.array([258.0, 270.0, 294.0,
              320.0, 342.0, 368.0,
              396.0, 446.0, 480.0,
              586.0])[:, np.newaxis]
y = np.array([236.4, 234.4, 252.8,
              298.6, 314.2, 342.2,
              360.8, 368.0, 391.2,
              390.8])
lr = LinearRegression()
pr = LinearRegression()
quadratic = PolynomialFeatures(degree=2)
X_quad = quadratic.fit_transform(X)
```

2. Для сравнения выполнить подгонку простой линейной регрессионной модели:

```
lr.fit(X, y)
X_fit = np.arange(250,600,10)[:, np.newaxis]
y_lin_fit = lr.predict(X_fit)
```

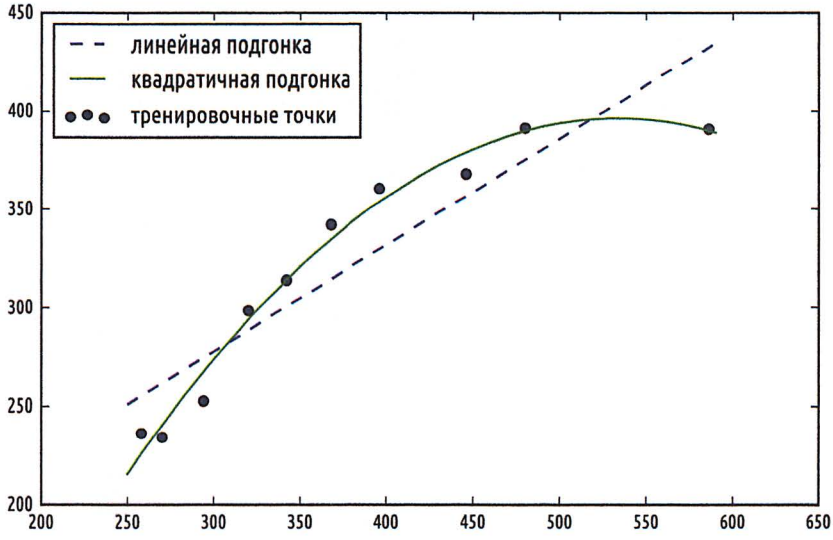
3. Выполнить подгонку множественной регрессионной модели на преобразованных признаках для полиномиальной регрессии:

```
pr.fit(X_quad, y)
y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Построить график результатов:

```
plt.scatter(X, y, label='тренировочные точки')
plt.plot(X_fit, y_lin_fit,
         label='линейная подгонка', linestyle='--')
plt.plot(X_fit, y_quad_fit,
         label='квадратичная подгонка')
plt.legend(loc='upper left')
plt.show()
```

На итоговом графике видно, что полиномиальная подгонка захватывает связь между переменной отклика и объясняющей переменной намного лучше, чем линейная подгонка:




```
>>>
y_lin_pred = lr.predict(X)
y_quad_pred = pr.predict(X_quad)
print('Тренировочная MSE линейная: %.3f, квадратичная: %.3f' % (
      mean_squared_error(y, y_lin_pred),
      mean_squared_error(y, y_quad_pred)))
Тренировочная MSE линейная: 569.780, квадратичная: 61.330

print('Тренировочная R^2 линейная: %.3f, квадратичная: %.3f' % (
      r2_score(y, y_lin_pred),
      r2_score(y, y_quad_pred)))
Тренировочная R^2 линейная: 0.832, квадратичная: 0.982
```

Как видно после выполнения приведенного выше исходного кода, средневзвешенная квадратичная ошибка (MSE) в этой отдельно взятой миниатюрной задаче уменьшилась с 570 (линейная подгонка) до 61 (квадратичная подгонка), при этом коэффициент детерминации отражает более тесную подгонку к квадратичной модели (R^2 = 0.982) в противоположность линейной подгонке (R^2 = 0.832).

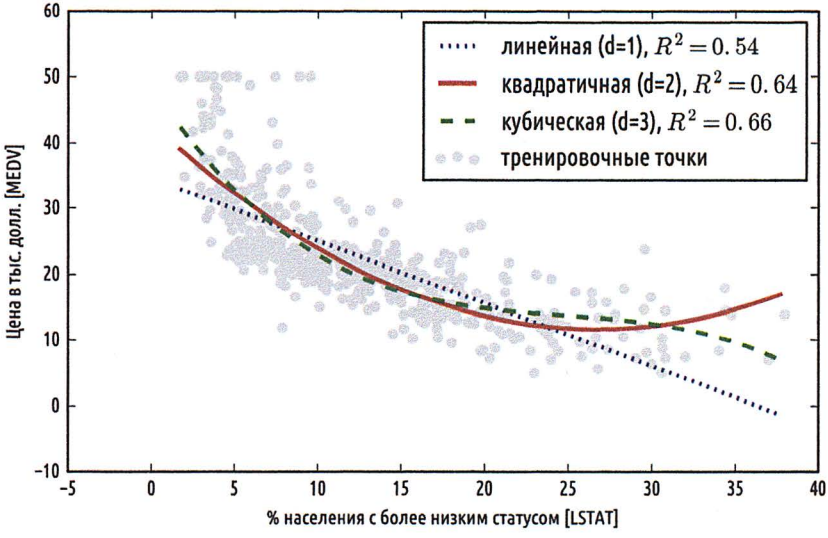
Моделирование нелинейных связей в наборе данных Housing

После того как мы обсудили вопрос конструирования полиномиальных признаков для подгонки нелинейных связей в миниатюрной задаче, теперь рассмотрим более конкретный пример и применим эти принципы работы к данным *Housing*. Выполнив приведенный ниже исходный код, мы смоделируем связь между ценами на жилье и LSTAT (процентом населения с более низким статусом) с использованием полинома второй степени (квадратичного) и третьей степени (кубического) и сравним его с линейной подгонкой. Соответствующий исходный код приведен ниже:

```
X = df[['LSTAT']].values
y = df['MEDV'].values
regr = LinearRegression()
```

```
# создать полиномиальные признаки
quadratic = PolynomialFeatures(degree=2)
cubic = PolynomialFeatures(degree=3)
X_quad = quadratic.fit_transform(X)
X_cubic = cubic.fit_transform(X)

# линейная подгонка
X_fit = np.arange(X.min(), X.max(), 1)[:, np.newaxis]
regr = regr.fit(X, y)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y, regr.predict(X))

# квадратичная подгонка
regr = regr.fit(X_quad, y)
y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
quadratic_r2 = r2_score(y, regr.predict(X_quad))

# кубическая подгонка
regr = regr.fit(X_cubic, y)
y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
cubic_r2 = r2_score(y, regr.predict(X_cubic))

# построить график с результатами
plt.scatter(X, y,
            label='тренировочные точки',
            color='lightgray')
plt.plot(X_fit, y_lin_fit,
         label='линейная (d=1), $R^2=%.2f$'
               % linear_r2,
         color='blue',
         lw=2,
         linestyle=':')
plt.plot(X_fit, y_quad_fit,
         label='квадратичная (d=2), $R^2=%.2f$'
                % quadratic_r2,
         color='red',
         lw=2,
         linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
             label='кубическая (d=3), $R^2=%.2f$'
                    % cubic_r2,
             color='green',
             lw=2,
             linestyle='--')
plt.xlabel('% населения с более низким статусом [LSTAT]')
plt.ylabel('Цена в тыс. долл. [MEDV]')
plt.legend(loc='upper right')
plt.show()
```

Как видно на получившемся графике, кубическая подгонка захватывает связь между ценами на жилье и **LSTAT** лучше, чем линейная и квадратичная подгонка. Однако мы должны осознавать, что добавление все большего количества полиномиальных признаков увеличивает сложность модели и поэтому увеличивает шанс ее переобучения. Поэтому на практике всегда рекомендуется выполнять оценку работоспособности модели на отдельном тестовом наборе данных для оценки обобщающей способности:

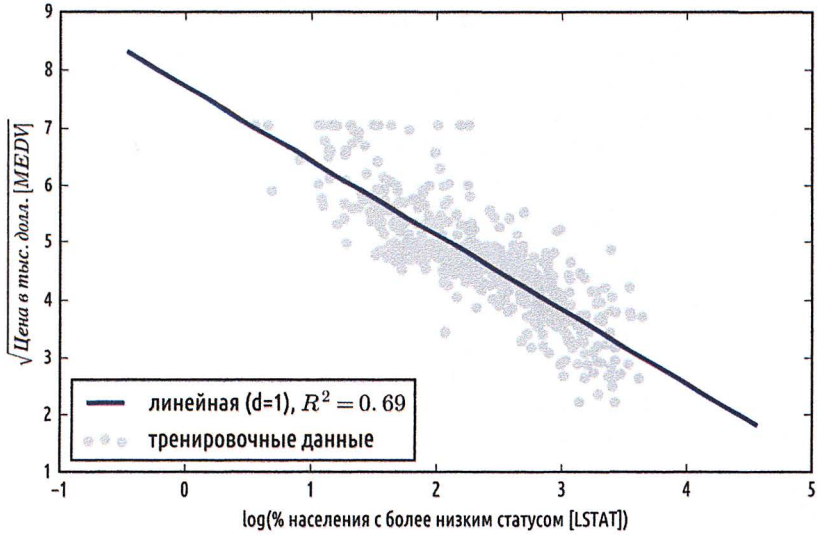


Кроме того, использование полиномиальных признаков для моделирования нелинейных связей не всегда является наилучшим вариантом. Например, просто посмотрев на точечный график **MEDV-LSTAT**, можно было бы предположить, что логарифмическое преобразование переменной признака **LSTAT** и квадратный корень переменной **MEDV** могут спроецировать данные на линейное пространство признаков, подходящее для линейной регрессионной подгонки. Проверим эту гипотезу, выполнив следующий ниже фрагмент исходного кода:

```
# преобразовать признаки
X_log = np.log(X)
y_sqrt = np.sqrt(y)

# выполнить подгонку признаков
X_fit = np.arange(X_log.min()-1,
                  X_log.max()+1, 1)[:, np.newaxis]
regr = regr.fit(X_log, y_sqrt)
y_lin_fit = regr.predict(X_fit)
linear_r2 = r2_score(y_sqrt, regr.predict(X_log))

# построить график с результатами
plt.scatter(X_log, y_sqrt,
            label='тренировочные точки',
            color='lightgray')
plt.plot(X_fit, y_lin_fit,
         label='линейная (d=1), $R^2=%.2f$' % linear_r2,
         color='blue',
         lw=2)
plt.xlabel('log(% населения с более низким статусом [LSTAT])')
plt.ylabel('$\sqrt{Цена \; в \; тыс. долл. [MEDV]}$')
plt.legend(loc='lower left')
plt.show()
```

После преобразования объясняющих в логарифмическое пространство и взятие квадратного корня целевых переменных мы смогли захватить связь между двумя

переменными с линией линейной регрессии, которая, похоже, аппроксимирует данные лучше ($R^2 = 0.69$), чем любое из предыдущих преобразований полиномиальных признаков:



Обработка нелинейных связей при помощи случайных лесов

В этом разделе мы рассмотрим регрессию на основе **случайного леса**, которая концептуально отличается от предыдущих моделей регрессии, рассмотренных в этой главе. Случайный лес, т. е. ансамбль из двух и более **деревьев решений**, может пониматься как сумма кусочно-линейных функций, в отличие от глобальных моделей линейной и полиномиальной регрессии, которые мы обсудили ранее. Другими словами, при помощи алгоритма дерева решений мы подразделяем входное пространство на области меньшего размера, которые становятся более *управляемыми*.

Регрессия на основе дерева решений

Преимущество алгоритма дерева решений состоит в том, что он не требует преобразования признаков, в случае если мы имеем дело с нелинейными данными. Из *главы 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* мы помним, что мы выращиваем дерево решений путем итеративного расщепления его узлов, пока листы не станут однородными, либо не будет удовлетворен критерий останова. Когда мы применяли деревья решений для классификации данных, мы определили энтропию как меру неоднородности для установления того, какое расщепление признака максимизирует **прирост информации** (information gain, IG), который для бинарного расщепления можно определить следующим образом:

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

Здесь x – это признак, на котором выполняется расщепление, N_p – число образцов в родительском узле, I – функция неоднородности, D_p – подмножество тренировочных образцов в родительском узле и $D_{\text{левый}}$ и $D_{\text{правый}}$ – подмножества тренировочных образцов в левом и правом дочерних узлах после расщепления. Напомним, что наша задача – найти расщепление признака, которое максимизирует прирост информации, или, другими словами, мы хотим найти расщепление признака, которое сокращает неоднородности в дочерних узлах. В *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* в качестве меры неоднородности мы использовали *энтропию*, критерий, который широко применяется в задачах классификации. Для использования дерева решений для регрессии мы заменим энтропию как меру неоднородности узла t на MSE:

$$I(t) = \text{MSE}(t) = \frac{1}{N_t}\sum_{i \in D_t}^{n}(y^{(i)} - \hat{y}_t)^2.$$

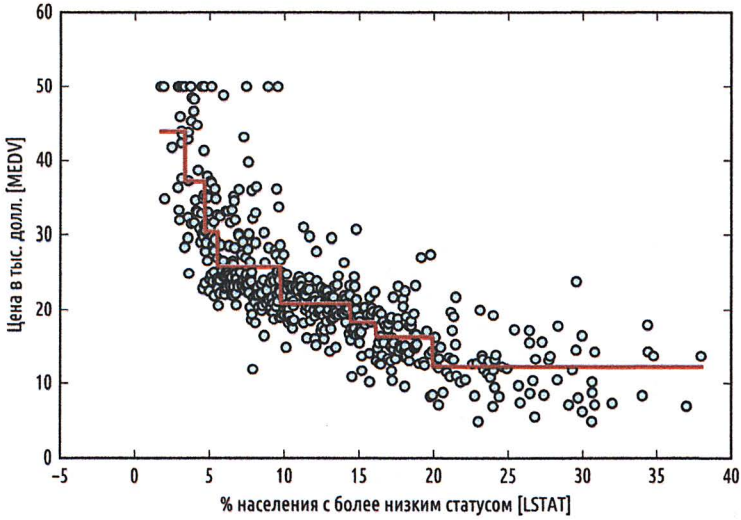
Здесь N_t – число тренировочных образцов в узле t, D_t – тренировочное подмножество в узле t, $y^{(i)}$ – истинное целевое значение и $\hat{y}_t$ – предсказанное целевое значение (эмпирическое среднее):

$$\hat{y}_t = \frac{1}{N}\sum_{i \in D_t} y^{(i)}.$$

В контексте регрессии на основе дерева решений показатель MSE часто также упоминается как внутриузловая дисперсия, и по этой причине критерий расщепления также более известен как *сокращение дисперсии*. Чтобы посмотреть, на что похожа линия подгонки дерева решений, воспользуемся для моделирования нелинейной связи между переменными **MEDV** и **LSTAT** регрессором дерева решений DecisionTreeRegressor, реализованным в библиотеке scikit-learn:

```
from sklearn.tree import DecisionTreeRegressor
X = df[['LSTAT']].values
y = df['MEDV'].values
tree = DecisionTreeRegressor(max_depth=3)
tree.fit(X, y)
sort_idx = X.flatten().argsort()
lin_regplot(X[sort_idx], y[sort_idx], tree)
plt.xlabel('% населения с более низким статусом [LSTAT]')
plt.ylabel('Цена в тыс. долл. [MEDV]')
plt.show()
```

Как видно на итоговом графике, дерево решений захватывает общий тренд в данных. Однако ограничение этой модели состоит в том, что она не захватывает непрерывность и дифференцируемость нужного предсказания. Кроме того, нам нужно быть осторожными относительно выбора надлежащего значения для глубины дерева, чтобы не было переобучения либо недообучения под данные; в данном случае глубина 3, похоже, представляет собой хороший выбор:



В следующем разделе мы обратимся к более устойчивому способу подгонки регрессионных деревьев: случайным лесам.

Регрессия на основе случайного леса

Как обсуждалось в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, алгоритм случайного леса представляет собой ансамблевый метод, который объединяет два и более деревьев решений. Случайный лес обычно имеет обобщающую способность лучше, чем у отдельного дерева решений, ввиду случайности, помогающей уменьшить дисперсию модели. Другие преимущества случайных лесов состоят в том, что они менее чувствительны к выбросам в наборе данных и не требуют большой настройки параметров. Единственный параметр в случайных лесах, с которым нам нужно экспериментировать, – это, как правило, число деревьев в ансамбле. Базовый алгоритм случайного леса для регрессии почти идентичен алгоритму случайного леса для классификации, который мы обсуждали в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*. Разница состоит в том, что для роста отдельных деревьев решений мы используем критерий MSE, и предсказанная целевая переменная вычисляется как усредненное предсказание по всем деревьям решений.

Теперь воспользуемся всеми признаками в наборе данных Housinf, чтобы выполнить подгонку регрессионной модели на основе случайного леса на 60% образцов, и оценим качество ее работы на оставшихся 40%. Соответствующий исходный код выглядит следующим образом:

```
>>>
X = df.iloc[:, :-1].values
y = df['MEDV'].values
X_train, X_test, y_train, y_test = \
      train_test_split(X, y,
                       test_size=0.4,
                       random_state=1)
```

```
from sklearn.ensemble import RandomForestRegressor
forest = RandomForestRegressor(n_estimators=1000,
                               criterion='mse',
                               random_state=1,
                               n_jobs=-1)
forest.fit(X_train, y_train)
y_train_pred = forest.predict(X_train)
y_test_pred = forest.predict(X_test)
print('MSE тренировка: %.3f, тестирование: %.3f' % (
      mean_squared_error(y_train, y_train_pred),
      mean_squared_error(y_test, y_test_pred)))
print('R^2 тренировка: %.3f, тестирование: %.3f' % (
      r2_score(y_train, y_train_pred),
      r2_score(y_test, y_test_pred)))
```

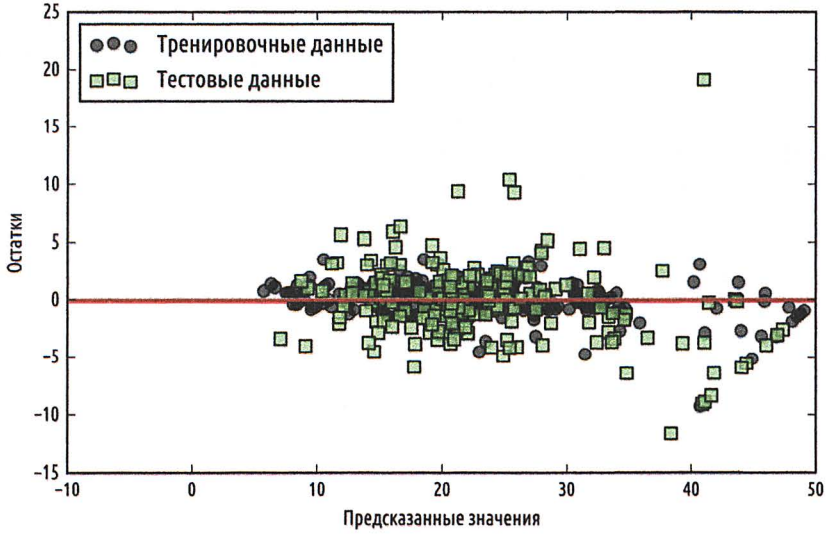
```
MSE тренировка: 1.642, тестирование: 11.635
R^2 тренировка: 0.960, тестирование: 0.871
```

К сожалению, мы видим, что случайный лес демонстрирует тенденцию к переподгонке под тренировочные данные. Однако он по-прежнему в состоянии относительно хорошо объяснить связь между целевой и объясняющими переменными (R^2 = 0.871 на тестовом наборе данных).

Наконец, также посмотрим на остатки предсказания:

```
plt.scatter(y_train_pred,
            y_train_pred - y_train,
            c='black',
            marker='o',
            s=35,
            alpha=0.5,
            label='Тренировочные данные')

plt.scatter(y_test_pred,
            y_test_pred - y_test,
            c='lightgreen',
            marker='s',
            s=35,
            alpha=0.7,
            label='Тестовые данные')
plt.xlabel('Предсказанные значения')
plt.ylabel('Остатки')
plt.legend(loc='upper left')
plt.hlines(y=0, xmin=-10, xmax=50, lw=2, color='red')
plt.xlim([-10, 50])
plt.show()
```

Как уже было резюмировано коэффициентом детерминации R^2, мы видим, что модель соответствует тренировочным данным лучше тестовых данных, на что указывают выбросы в направлении оси Y. Кроме того, распределение остатков вокруг нулевой отметки не выглядит абсолютно случайным, указывая, что модель не в состоянии захватить всю разведочную информацию. Однако график остатков показывает большое улучшение, по сравнению с графиком остатков линейной модели, который мы построили в этой главе ранее:



В *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* мы также рассмотрели ядерный трюк, который может использоваться для задачи классификации в сочетании с **методом опорных векторов** (SVM), что целесообразно делать, в случае если мы имеем дело с нелинейными задачами. Учитывая, что обсуждение этой темы выходит за пределы объема данной книги, просто отметим, что методы SVM могут также применяться в нелинейных регрессионных задачах. Заинтересованный читатель может найти дополнительную информацию о методах опорных векторов для задачи регрессии в превосходном техническом отчете С. Р. Ганна: S. R. Gunn et al., «Support Vector Machines for Classification and Regression», ISIS technical report, 14, 1998 (*«Методы опорных векторов для задач классификации и регрессии»*). Регрессор SVM также реализован в библиотеке scikit-learn, и дополнительную информацию о его применении можно найти на http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR.

Резюме

В начале этой главы вы научились использовать простой линейный регрессионный анализ для моделирования связи между единственной объясняющей переменной (независимой переменной) и непрерывной переменной отклика (зависимой переменной). Затем мы обсудили широко применяемый метод разведочного анализа данных для поиска повторяющихся образов и аномалий в данных, который является важным первым шагом в задачах прогнозного моделирования.

Мы построили нашу первую модель, реализовав линейную регрессию с использованием подхода на основе градиентной оптимизации. Затем мы увидели, каким образом применять на практике линейные модели библиотеки scikit-learn для задачи регрессии, а также реализовывать стабильный регрессионный метод RANSAC как подход для решения проблемы выбросов. Чтобы определить предсказательную способность регрессионных моделей, мы вычислили средневзвешенную сумму квадратичных ошибок и связанную с ней метрику R^2. Кроме того, мы также обсудили полезный графический прием, применяемый для диагностирования проблем регрессионных моделей: график остатков.

После обсуждения того, как применять регуляризацию к регрессионным моделям с целью уменьшить сложность модели и избежать ее переобучения, мы также представили несколько подходов для моделирования нелинейных связей, включая полиномиальное преобразование признаков и регрессоры на основе случайных лесов.

В предыдущих главах мы очень подробно обсудили принципы и методы обучения с учителем, задачу классификации и регрессионный анализ. В следующей главе мы обсудим еще одну интересную подобласть машинного обучения: обучение без учителя. В следующей главе вы научитесь применять кластерный анализ для нахождения скрытых структур в данных в отсутствие целевых переменных.

Глава 11

Работа с немаркированными данными – кластерный анализ

В предыдущих главах мы использовали методы обучения с учителем для конструирования машиннообучаемых моделей с использованием данных, где ответ был уже известен, – метки классов уже имелись в наших тренировочных данных. В этой главе мы сменим тему и займемся разбором кластерного анализа, категории методов **обучения без учителя**, позволяющей обнаруживать скрытые структуры в данных, где мы заранее не знаем правильного ответа. Задача кластеризации состоит в том, чтобы отыскать в данных естественное разбиение по группам, такое что элементы в том же кластере более подобны друг другу, чем из других кластеров.

Кластеризация данных, отличаясь разведочным характером процедуры, представляет собой захватывающую тему, и в этой главе вы узнаете следующие понятия, которые способны помочь вам организовать данные в содержательные структуры:

- нахождение центров подобия при помощи популярного алгоритма *k* средних;
- использование восходящего принципа для построения иерархических кластерных деревьев;
- идентификация произвольных фигур объектов при помощи подхода к кластеризации на основе плотности.

Группирование объектов по подобию методом *k* средних

В этом разделе мы обсудим один из самых популярных алгоритмов **кластеризации**, алгоритм ***k* средних** (*k*-means), который широко используется в научных кругах и в информационной отрасли. Кластеризация, или кластерный анализ, – это методология, которая позволяет находить группы подобных объектов, объекты, более связанные друг с другом, чем с объектами в других группах. Примеры бизнес-ориентированных приложений кластеризации включают группирование документов, музыки и фильмов по различным темам или нахождение клиентов, разделяющих схожие интересы, основываясь на общих проявлениях покупательского поведения как основы для рекомендательных движков.

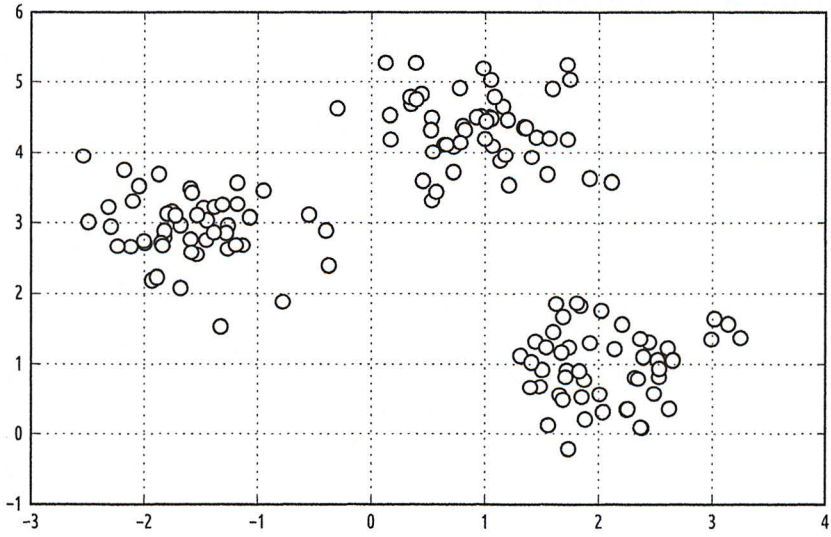
Как мы вскоре увидим, алгоритм *k* средних чрезвычайно прост в реализации и одновременно очень эффективен в вычислительном плане, по сравнению с другими алгоритмами кластеризации, что может объяснять его популярность. Алгоритм *k* средних принадлежит к категории кластеризации на основе прототипов. Далее в этой главе мы обсудим две другие категории кластеризации, **иерархическую** кластеризацию и кластеризацию **на основе плотности**. Кластеризация **на основе прото-**

типов означает, что каждый кластер представлен прототипом, который может быть либо **центроидом** (средним) подобных точек с непрерывными признаками, либо **медоидом** (наиболее представительной или наиболее часто встречающейся точкой) в случае категориальных признаков. В то время как алгоритм *k* средних очень хорошо выполняет идентификацию кластеров сферической формы, один из недостатков этого алгоритма кластеризации состоит в том, что нам нужно априорно указывать число кластеров *k*. Некорректный выбор числа *k* может привести к плохой кластеризующей способности. Позже в этой главе мы обсудим **метод локтя** и **силуэтные графики**, полезные методы оценки кластеризующей способности, помогающие определить оптимальное число кластеров *k*.

Несмотря на то что кластеризация по методу *k* средних может применяться к данным более высокой размерности, в целях визуализации последующие примеры будут проанализированы с использованием простого двумерного набора данных:

```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=150,
                  n_features=2,
                  centers=3,
                  cluster_std=0.5,
                  shuffle=True,
                  random_state=0)
import matplotlib.pyplot as plt
plt.scatter(X[:,0],
            X[:,1],
            c='white',
            marker='o',
            s=50)
plt.grid()
plt.show()
```

Мы только что создали набор данных, который состоит из 150 случайно сгенерированных точек, сгруппированных примерно на три области с более высокой плотностью, которые визуализируются на двумерном точечном графике:



В реальных приложениях кластеризации раз и навсегда заданная информация о категориях по этим образцам отсутствует; в противном случае они попали бы в категорию обучения с учителем. Поэтому наша задача состоит в том, чтобы сгруппировать образцы, основываясь на подобии их признаков, что можно достичь при помощи алгоритма k средних, который можно резюмировать в следующих четырех шагах.

1. Случайно выбрать из точек образцов k центроидов как исходных центров кластеров.
2. Назначить каждый образец самому ближайшему к ней центроиду $\mu^{(i)}, j \in \{1, ..., k\}$.
3. Переместить каждый центроид в центр образцов, которые были ему назначены.
4. Повторять шаги 2 и 3, пока назначения кластеров не перестанут изменяться либо не будет достигнут заданный пользователями допуск или максимальное число итераций.

Теперь следующий вопрос заключается в том, *каким образом измерять подобие между объектами*. Подобие можно определить как противоположность расстоянию, и для кластеризации образцов с непрерывными признаками обычно используется **квадратичное евклидово расстояние** между двумя точками x *и* y в m*-мерном* пространстве:

$$d(\boldsymbol{x},\boldsymbol{y})^2 = \sum_{j=1}^{m}(x_j - y_j)^2 = \|\boldsymbol{x}-\boldsymbol{y}\|_2^2.$$

Отметим, что в предыдущем уравнении индекс j относится к j-му измерению (признаковому столбцу) точек образцов x и y. В остальной части данного раздела мы будем использовать надстрочные индексы i и j для обозначения, соответственно, индекса образца и индекса кластера.

Основываясь на евклидовой метрике расстояния, можно описать алгоритм k средних как простую задачу оптимизации – итеративную минимизацию **внутрикластерной суммы квадратичных ошибок** (SSE), которая иногда также называется **инерцией кластера**:

$$\mathrm{SSE} = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)} \left\|\boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)}\right\|_2^2.$$

Здесь $\mu^{(j)}$ – это представительная точка (центроид) для кластера j и $w^{(i,j)} = 1$, в случае если образец $x^{(i)}$ находится в группе j, и $w^{(i,j)} = 0$ в противном случае.

Познакомившись с тем, как работает простой алгоритм k средних, теперь применим его к нашему модельному набору данных при помощи класса KMeans с реализацией алгоритма k средних из модуля sklearn.cluster библиотеки scikit-learn:

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=3,
            init='random',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
y_km = km.fit_predict(X)
```

Используя приведенный выше исходный код, мы задаем нужное число кластеров равным 3; априорное задание числа кластеров является одним из ограничений алгоритма *k* средних. Задаем параметр n_init=10 для 10-кратного независимого выполнения алгоритмов кластеризации методом *k* средних с разными случайными центроидами, чтобы в результате выбрать окончательную модель, которая имеет самую низкую SSE. В параметре max_iter мы указываем максимальное число итераций для каждого отдельного прохода (в данном случае 300). Отметим, что реализация алгоритма *k* средних в библиотеке scikit-learn останавливается рано, в случае если алгоритм сходится до достижения максимального числа итераций.

Однако существует возможность, что для отдельно взятого прохода алгоритм *k* средних не достигнет сходимости, что может быть проблематичным (затратным в вычислительном плане), если выбирать относительно большие значения для max_iter. Один из способов решения проблемы сходимости состоит в том, чтобы выбирать более крупные значения для tol, параметра, который управляет допустимом значении относительно изменений в показателе внутрикластерной суммы квадратичных ошибок и объявляет о сходимости. В приведенном выше примере исходного кода мы выбрали допуск равным 1e-04 (= 0.0001).

Алгоритм k-средних++

До настоящего момента мы обсуждали классический алгоритм *k* средних, в котором для размещения исходных центроидов используется случайное начальное число (random seed), что иногда может давать плохие результаты кластеризации либо замедление сходимости, в случае если исходные центроиды выбраны плохо. Один из способов решения этой проблемы состоит в кратном выполнении алгоритма *k* средних на наборе данных и выборе наиболее качественной модели с точки зрения SSE. Другая стратегия заключается в том, чтобы помещать исходные центроиды далеко друг от друга, используя для этого алгоритм ***k*-средних++** (*k*-means++), что в итоге приводит к более хорошим и более достоверным результатам, чем классический алгоритм *k* средних (D. Arthur, S. Vassilvitskii. *k-means++: The Advantages of Careful Seeding* («*k*-средних++: преимущества аккуратного обращения со случайным начальным числом ГПСЧ»). In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, p. 1027–1035. Society for Industrial and Applied Mathematics, 2007).

Инициализацию алгоритма *k*-средних++ можно резюмировать следующим образом:

1. Инициализировать пустое множество **M** для хранения *k* отбираемых центроидов.
2. Случайно выбрать из входных образцов первый центроид $\mu^{(j)}$ и назначить его множеству **M**.
3. Для каждого образца $x^{(i)}$, который не находится в **M**, найти минимальное квадратичное расстояние $d(x^{(i)}, \mathbf{M})^2$ до любого из центроидов в **M**.
4. Чтобы случайно отобрать следующий центроид $\mu^{(p)}$, использовать взвешенное вероятностное распределение, равное $\frac{d(\mu^{(p)},\mathbf{M})^2}{\sum_i d(x^{(i)},\mathbf{M})^2}$.
5. Повторить шаги 2 и 3, пока не будет выбрано *k* центроидов.
6. Продолжить работу с классическим алгоритмом *k* средних.

Чтобы воспользоваться алгоритмом *k*-средних++ на основе объекта KMeans библиотеки scikit-learn, нужно просто вместо random установить параметр init в k-means++ (конфигурация по умолчанию).

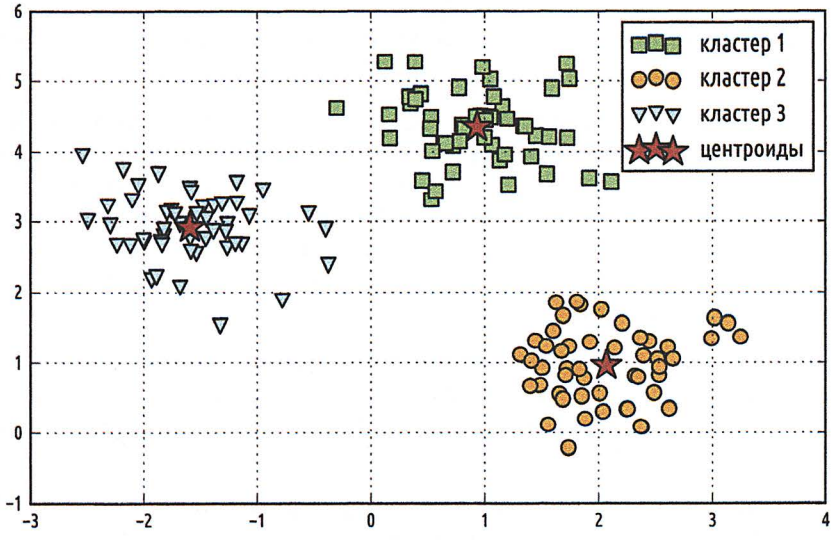
Еще одна проблема с алгоритмом *k* средних состоит в том, что один или более кластеров могут быть пустыми. Отметим, что эта проблема не существует для *k* медоидов или алгоритма нечетких С-средних, который мы обсудим в следующем подразделе. Впрочем, в текущей реализации *k* средних в библиотеке scikit-learn эта проблема учтена. Если кластер пуст, то алгоритм отыщет образец, находящийся на наибольшем удалении от центроида пустого кластера. Затем он сделает центроид равным этой самой дальней точке.

Когда мы применяем алгоритм *k* средних к реальным данным, при этом пользуясь евклидовой метрикой расстояния, мы должны убедиться, что признаки измеряются в той же самой шкале, и при необходимости применить стандартизацию на основе z-оценки либо минимаксное масштабирование.

После того как мы предсказали метки кластеров y_km и обсудили трудности алгоритма *k* средних, теперь в наглядной форме покажем кластеры, которые алгоритм *k* средних идентифицировал в наборе данных, вместе с кластерными центроидами. Они хранятся в атрибуте centers_ подогнанного объекта KMeans:

```
plt.scatter(X[y_km==0,0],
            X[y_km==0,1],
            s=50,
            c='lightgreen',
            marker='s',
            label='кластер 1')
plt.scatter(X[y_km==1,0],
            X[y_km==1,1],
            s=50,
            c='orange',
            marker='o',
            label='кластер 2')
plt.scatter(X[y_km==2,0],
            X[y_km==2,1],
            s=50,
            c='lightblue',
            marker='v',
            label='кластер 3')
plt.scatter(km.cluster_centers_[:,0],
            km.cluster_centers_[:,1],
            s=250,
            marker='*',
            c='red',
            label='центроиды')
plt.legend()
plt.grid()
plt.show()
```

На следующем ниже точечном графике мы видим, что алгоритм *k* средних поместил эти три центроида в центр каждой сферы, что выглядит как разумная группировка при наличии используемого набора данных:



Хотя алгоритм *k* средних хорошо показал себя на этом миниатюрном наборе данных, стоит отметить некоторые из основных трудностей алгоритма *k* средних. Один из недостатков алгоритма *k* средних состоит в том, что нам приходится указывать число кластеров *k* априорно, что не всегда может быть настолько очевидным в реальных приложениях, в особенности если мы работаем с набором данных более высокой размерности, который невозможно визуализировать. Другие свойства *k* средних состоят в том, что кластеры не накладываются и не иерархичны, и мы также исходим из допущения, что в каждом кластере имеется, по крайней мере, один элемент.

Жесткая кластеризация в сопоставлении с мягкой

Жесткая кластеризация описывает семейство алгоритмов, где каждый образец в наборе данных назначен строго одному кластеру, как в алгоритме *k* средних, который мы обсудили в предыдущем подразделе. В отличие от них, алгоритмы **мягкой кластеризации**, иногда также именуемые **нечеткой кластеризацией** (fuzzy clustering), назначают образец одному или более кластерам. Популярным примером мягкой кластеризации является алгоритм **нечетких C-средних** (fuzzy C-means, FCM), также именуемый алгоритмом **мягких *k* средних** или **нечетких *k* средних**. Первоначальная идея восходит к 1970-м, когда Джозеф К. Данн впервые предложил раннюю версию нечеткой кластеризации для улучшения алгоритма *k* средних (J. C. Dunn, «A Fuzzy Relative of the Isodata Process and its Use in Detecting Compact Well-separated Clusters», 1973 (*«Нечеткий родственник процесса изоданных и его использование для обнаружения компактных хорошо разделенных кластеров»*)). Почти десятилетие спустя Джеймс К. Бедзек опубликовал свою работу по усовершенствованию алгоритма нечеткой кластеризации, который теперь известен как алгоритм FCM (J. C. Bezdek, «Pattern Recognition with Fuzzy Objective Function Algorithms». Springer Science & Business Media, 2013 (*«Распознавание образов при помощи алгоритмов с нечеткой целевой функцией»*)).

Процедура алгоритма FCM очень похожа на алгоритм k средних. Однако вместо жесткого назначения кластеров каждой принадлежащей кластеру точке мы назначаем им вероятности. В k средних мы можем выразить принадлежность образца x кластеру разряженным вектором двоичных значений:

$$\begin{bmatrix} \mu^{(1)} \to 0 \\ \mu^{(2)} \to 1 \\ \mu^{(3)} \to 0 \end{bmatrix}.$$

Здесь позиция индекса со значением 1 указывает на кластерный центроид $\mu^{(j)}$, которому назначен образец (принимая, что $k = 3, j \in \{1, 2, 3\}$). Напротив, в алгоритме FCM вектор принадлежности может быть представлен следующим образом:

$$\begin{bmatrix} \mu^{(1)} \to 0.1 \\ \mu^{(2)} \to 0.85 \\ \mu^{(3)} \to 0.05 \end{bmatrix}.$$

Здесь каждое значение попадает в диапазон [0, 1] и представляет вероятность принадлежности соответствующему кластерному центроиду. Сумма принадлежностей для заданного образца равна 1. Аналогично алгоритму k средних, алгоритм FCM можно резюмировать в четырех ключевых шагах:

1. Указать число k центроидов и для каждой точки случайным образом их принадлежности кластерам.
2. Вычислить кластерные центроиды $\mu^{(j)}, j \in \{1, ..., k\}$.
3. По каждой точке обновить их принадлежности кластерам.
4. Повторить шаги 2 и 3, пока коэффициенты принадлежности не перестанут изменяться либо не будет достигнут заданный пользователями допуск или максимальное число итераций.

Целевая функция алгоритма FCM – кратко обозначим ее как J_m – выглядит очень похожей на **внутрикластерную сумму квадратичных ошибок**, которую мы минимизируем в алгоритме k средних:

$$J_m = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{m(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2, m \in [1, \infty].$$

Однако отметим, что индикатор принадлежности $w^{(i,j)}$ – это не бинарное значение, как в k средних ($w^{(i,j)} \in \{0, 1\}$), а действительное значение, обозначающее вероятность принадлежности кластеру ($w^{(i,j)} \in [0, 1]$) . Вы также, возможно, заметили, что мы добавили в $w^{(i,j)}$ дополнительную экспоненту; экспонента m, любое число ≥ 1 (как правило, $m = 2$) – это так называемый **коэффициент нечеткости** (или просто **фаззификатор**), который управляет степенью **нечеткости**. Чем больше значение m, тем меньше становится принадлежность $w^{(i,j)}$, приводя к более нечетким кластерам. Непосредственно сама вероятность принадлежности кластеру вычисляется следующим образом:

$$w^{(i,j)} = \left[\sum_{p=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}.$$

Например, если в приведенном выше примере k средних выбрать три кластерных центра, то принадлежность образца $\boldsymbol{x}^{(i)}$ кластеру $\mu^{(j)}$, можно вычислить следующим образом:

$$w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}.$$

Центр $\mu^{(j)}$ непосредственно самого кластера вычисляется как среднее всех образцов в кластере, взвешенного на степень принадлежности своему собственному кластеру:

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}.$$

Просто глядя на равенство для вычисления принадлежностей кластерам, интуитивно понятно, что каждая итерация в алгоритме FCM более затратна, чем итерация в k средних. Однако алгоритм FCM, как правило, в целом требует меньше итераций до достижения сходимости. К сожалению, алгоритм FCM в библиотеке scikit-learn в настоящее время не реализован. Тем не менее на практике было показано, что оба алгоритма генерируют очень похожие результаты кластеризации, как описано в исследовании С. Гоша и С. К. Дьюби (S. Ghosh and S. K. Dubey, «Comparative Analysis of k-means and Fuzzy c-means Algorithms», IJACSA, 4:35-38, 2013 («*Сравнительный анализ алгоритмов k средних и нечетких C-средних*»)).

Использование метода локтя для нахождения оптимального числа кластеров

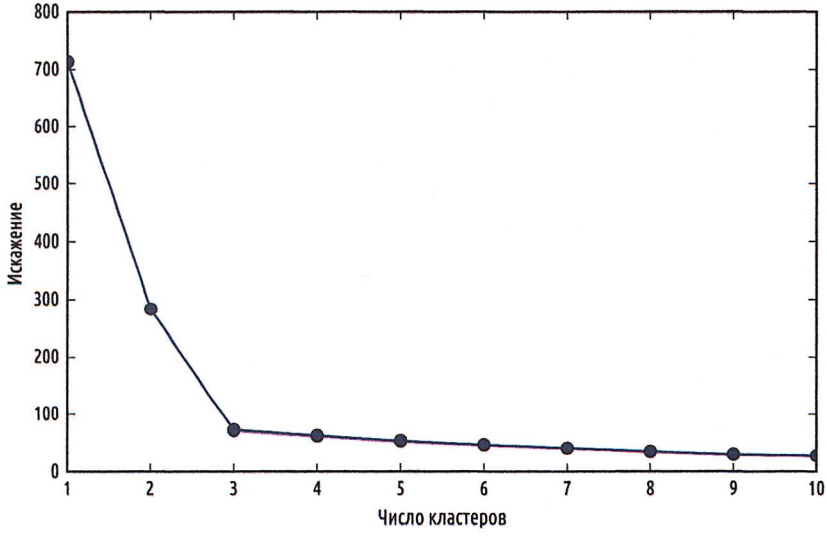
Одна из основных трудностей в обучении без учителя состоит в том, что мы не знаем точного ответа. В нашем наборе данных нет раз и навсегда установленных данных о метках классов, позволяющих применять методы, которые мы использовали в *главе 6 «Анализ наиболее успешных приемов оценивания моделей и тонкой настройки гиперпараметров»* для оценки качества модели с учителем. Поэтому для количественного определения качества кластеризации нам нужно использовать внутренние метрики – такие как внутрикластерная SSE (искажение или инерция), которую мы обсудили ранее в этой главе, – для сравнения качества разных кластеризаций по методу k средних. Удобно то, что нам не нужно вычислять внутрикластерную SSE явным образом, поскольку этот показатель уже доступен в атрибуте `inertia_` после подгонки модели `KMeans`:

```
>>>
print('Искажение: %.2f' % km.inertia_)
Искажение: 72.48
```

Основываясь на внутрикластерной SSE, мы можем применить графический инструмент, так называемый метод **локтя**, для оценки оптимального числа k кластеров для поставленной задачи. Интуитивно мы можем сказать, что если k увеличивается, то искажение уменьшается. Это вызвано тем, что образцы будут ближе к центроидам, которым они назначены. В основе метода локтя лежит идея, которая состоит в том, чтобы идентифицировать значение k в точке, где искажение начинает увеличиваться быстрее всего, что станет понятнее, если мы построим график искажения для разных значений k:

```
distortions = []
for i in range(1, 11):
    km = KMeans(n_clusters=i,
                init='k-means++',
                n_init=10,
                max_iter=300,
                random_state=0)
km.fit(X)
distortions.append(km.inertia_)
plt.plot(range(1,11), distortions, marker='o')
plt.xlabel('Число кластеров')
plt.ylabel('Искажение')
plt.show()
```

Как видно на следующем ниже графике, *локоть* расположен в $k = 3$, что свидетельствует о том, что $k = 3$ является действительно хорошим выбором для этого набора данных:



Количественная оценка качества кластеризации методом силуэтных графиков

Еще одна внутренняя метрика для оценки качества кластеризации представлена **силуэтным анализом**, который также может применяться к другим алгоритмам кластеризации, помимо k средних, которые мы обсудим далее в этой главе. Силуэтный анализ может использоваться в качестве графического инструмента для построения графика меры плотности группировки образцов в кластерах. Чтобы вычислить **силуэтный коэффициент** одиночного образца в нашем наборе данных, можно применить следующие три шага.

1. Вычислить внутрикластерную связность $a^{(i)}$ как среднее расстояние между образцом $x^{(i)}$ и всеми другими точками в том же самом кластере.
2. Вычислить межкластерное разделение $b^{(i)}$ от следующего ближайшего кластера как среднее расстояние между образцом $x^{(i)}$ и всеми образцами в ближайшем кластере.
3. Вычислить силуэт $s^{(i)}$ как разницу между внутрикластерной связностью и межкластерным разделением, деленную на наибольшее из этих двух значений, как показано ниже:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}.$$

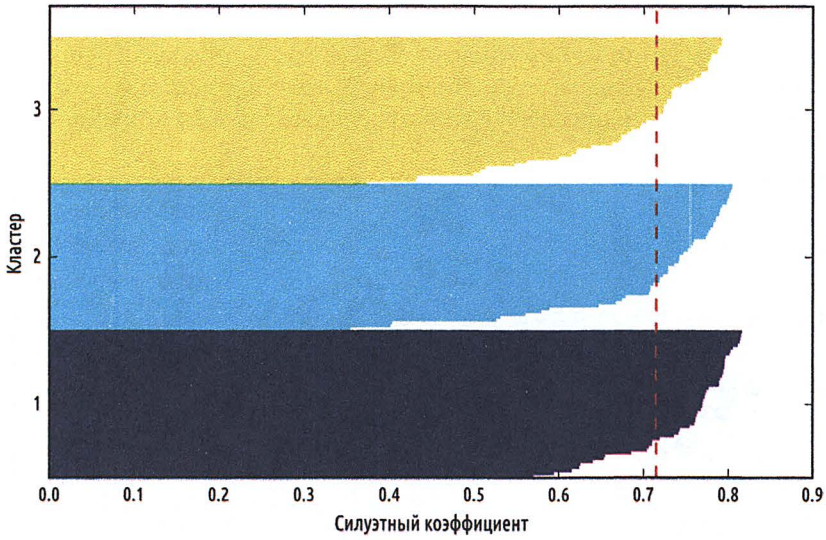
Силуэтный коэффициент ограничен диапазоном от –1 до 1. Основываясь на приведенной выше формуле, мы видим, что силуэтный коэффициент равен 0, если межкластерное разделение и внутрикластерная связность равны ($b^{(i)} = a^{(i)}$). Более того, мы приближаемся к идеальному силуэтному коэффициенту, равному 1, если $b^{(i)} >> a^{(i)}$, поскольку $b^{(i)}$ количественно определяет, насколько образец отличается от другого кластера, и соответственно $a^{(i)}$ говорит, насколько он подобен другим образцам в своем собственном кластере.

Силуэтный коэффициент доступен как функция silhouette_samples из модуля metric библиотеки scikit-learn, причем факультативно можно импортировать силуэтные оценки silhouette_scores. Эта функция вычисляет средний силуэтный коэффициент по всем образцам, являясь эквивалентом для numpy.mean(silhouette_samples(...)). Выполнив следующий ниже исходный код, теперь создадим график силуэтных коэффициентов для кластеризации по методу k средних с числом $k = 3$:

```
km = KMeans(n_clusters=3,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
y_km = km.fit_predict(X)
import numpy as np
from matplotlib import cm
from sklearn.metrics import silhouette_samples
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X,
                                     y_km,
                                     metric='euclidean')
```

```
y_ax_lower, y_ax_upper = 0, 0
yticks = []
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(float(i) / n_clusters) # float(i) для совместимости с Python 2.7
    plt.barh(range(y_ax_lower, y_ax_upper),
             c_silhouette_vals,
             height=1.0,
             edgecolor='none',
             color=color)
    yticks.append((y_ax_lower + y_ax_upper) / 2)
    y_ax_lower += len(c_silhouette_vals)
silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg,
            color="red",
            linestyle="--")
plt.yticks(yticks, cluster_labels + 1)
plt.ylabel('Кластер')
plt.xlabel('Силуэтный коэффициент')
plt.show()
```

Визуальный анализ силуэтного графика позволяет быстро рассмотреть размеры разных кластеров и идентифицировать кластеры, которые содержат *выбросы*:

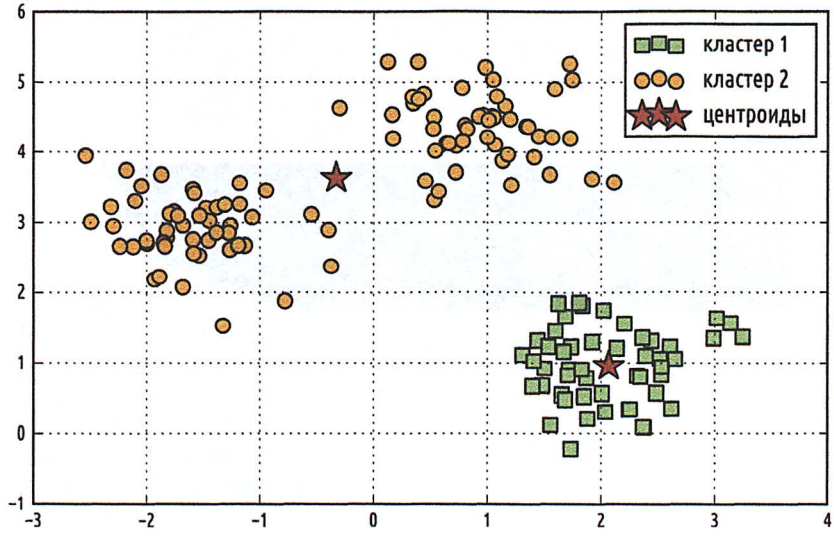


Как видно на приведенном выше силуэтном графике, наши силуэтные коэффициенты даже близко не находятся рядом с 0, что может служить индикатором хорошего объединения в кластеры. Более того, чтобы резюмировать качество нашей кластеризации, мы добавили в график средний силуэтный коэффициент (пунктирная линия).

Чтобы увидеть, как выглядит силуэтный график для относительно *плохой* кластеризации, инициализируем алгоритм *k* средних всего двумя центроидами:

```
km = KMeans(n_clusters=2,
            init='k-means++',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
y_km = km.fit_predict(X)
plt.scatter(X[y_km==0,0],
            X[y_km==0,1],
            s=50, c='lightgreen',
            marker='s',
            label='кластер 1')
plt.scatter(X[y_km==1,0],
            X[y_km==1,1],
            s=50,
            c='orange',
            marker='o',
            label='кластер 2')
plt.scatter(km.cluster_centers_[:,0],
            km.cluster_centers_[:,1],
            s=250,
            marker='*',
            c='red',
            label='центроиды')
plt.legend()
plt.grid()
plt.show()
```

Как видно на приведенном ниже точечном графике, один из центроидов попадает между двумя из трех сферических группировок точек (образцов). Хотя результат кластеризации и не выглядит совсем отвратительным, он – субоптимальный

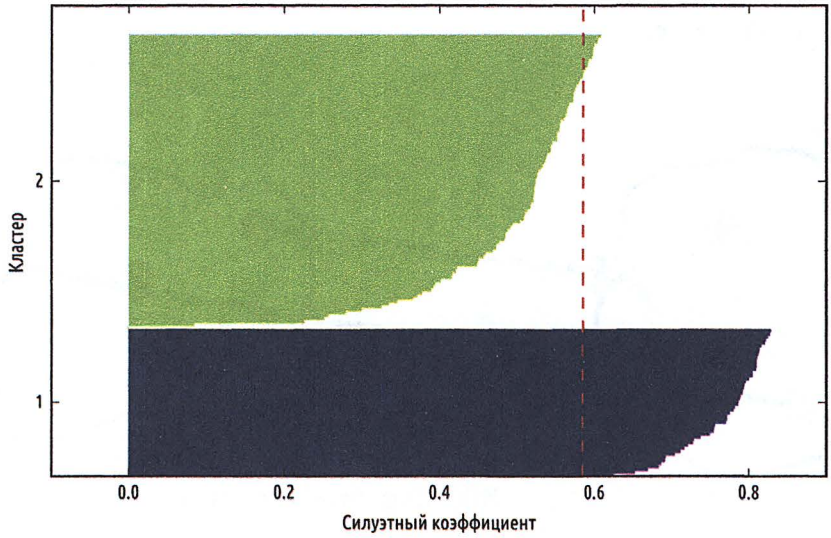


Далее создадим силуэтный график, чтобы оценить результаты. Следует иметь в виду, что в реальных задачах мы, как правило, не можем себе позволить роскошь

выполнять визуализацию наборов данных на двумерном точечном графике, поскольку мы, как правило, работаем с данными более высоких размерностей:

```
cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X,
                                     y_km,
                                     metric='euclidean')
y_ax_lower, y_ax_upper = 0, 0
yticks = []
for i, c in enumerate(cluster_labels):
    c_silhouette_vals = silhouette_vals[y_km == c]
    c_silhouette_vals.sort()
    y_ax_upper += len(c_silhouette_vals)
    color = cm.jet(i / n_clusters)
    plt.barh(range(y_ax_lower, y_ax_upper),
                   c_silhouette_vals,
                   height=1.0,
                   edgecolor='none',
                   color=color)
    yticks.append((y_ax_lower + y_ax_upper) / 2)
    y_ax_lower += len(c_silhouette_vals)
silhouette_avg = np.mean(silhouette_vals)
plt.axvline(silhouette_avg, color="red", linestyle="--")
plt.yticks(yticks, cluster_labels + 1)
plt.ylabel('Кластер')
plt.xlabel('Силуэтный коэффициент')
plt.show()
```

Как видно на итоговом графике, силуэты теперь имеют визуально различимые длину и ширину, предоставляя новые доказательства субоптимальной кластеризации:

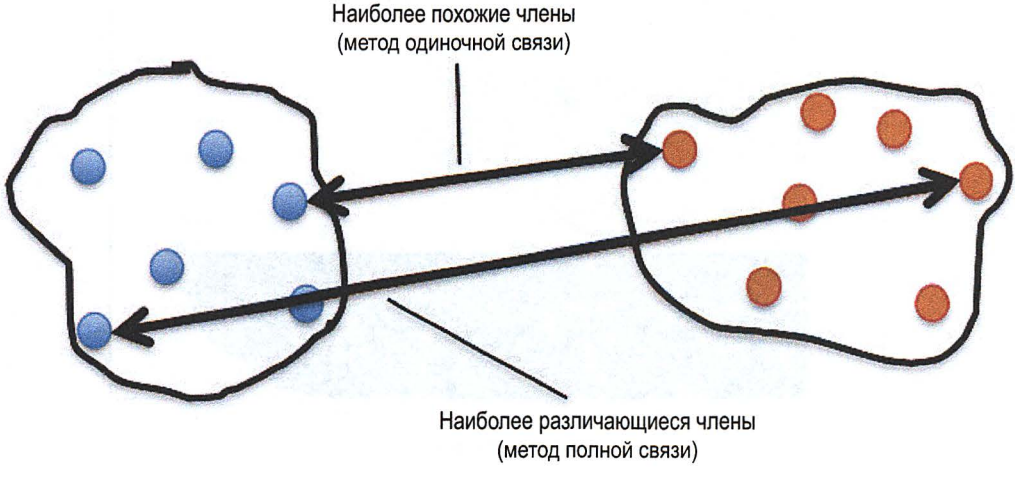


Организация кластеров в виде иерархического дерева

В этом разделе мы рассмотрим альтернативный подход к кластеризации на основе прототипов: **иерархическую кластеризацию**. Одно из преимуществ алгоритмов иерархической кластеризации состоит в том, что она позволяет строить **дендограммы** (древовидные визуализации бинарной иерархической кластеризации), которые способны помочь интерпретировать результаты путем создания содержательных таксономий. Другое полезное преимущество этого иерархического подхода состоит в том, что нам не нужно предварительно указывать число кластеров.

Существуют два основных подхода к иерархической кластеризации: **агломеративный** (объединительный) и **дивизивный** (разделяющий). В дивизивной иерархической кластеризации мы начинаем с единственного кластера, который охватывает все наши образцы, и мы итеративно расщепляем кластер на меньшие кластеры, пока каждый кластер не будет содержать всего один образец. В этом разделе мы сосредоточимся на агломеративной кластеризации, в которой принят противоположный подход. Мы начинаем с каждого образца как отдельного кластера и объединяем ближайшие пары кластеров, пока не останется всего один кластер.

Агломеративная иерархическая кластеризация представлена двумя стандартными алгоритмами: методом **одиночной связи** (single linkage, также известен как метод ближайшего соседа) и методом **полной связи** (complete linkage, также известен как метод дальнего соседа). Используя метод одиночной связи, для каждой пары кластеров мы вычисляем расстояния между самыми похожими членами и объединяем два кластера, для которых расстояние между самыми похожими членами наименьшее. Подход на основе полной связи подобен методу одиночной связи, но, вместо того чтобы в каждой паре кластеров сравнивать самых похожих членов, для выполнения объединения мы сравниваем наиболее различающихся членов. Это показано на приведенной ниже схеме:



Другие широко используемые алгоритмы агломеративной иерархической кластеризации представлены методом **средней связи** и **методом Уорда**. В методе средней связи мы объединяем пары кластеров, основываясь на минимальных средних расстояниях между всеми членами в двух кластерах. В методе Уорда объединяются те два кластера, которые приводят к минимальному увеличению общей внутрикластерной SSE.

В этом разделе мы сосредоточимся на агломеративной кластеризации на основе метода полной связи. Эту итеративную процедуру можно резюмировать следующими шагами.

1. Вычислить матрицу расстояний всех образцов.
2. Представить каждую точку данных как одноэлементный кластер.
3. Объединить два ближайших кластера, основываясь на расстоянии наиболее различающихся (дальних) членов.
4. Обновить матрицу расстояний.
5. Повторять шаги 2-4, пока не останется единственный кластер.

Теперь обсудим, как вычислять матрицу расстояний (шаг 1). Но сначала сгенерируем немного рабочих данных со случайными образцами. Строки представляют разные данные наблюдений (идентификаторы от 0 до 4); столбцы – разные признаки (X, Y, Z) этих образцов:

```
>>>
import pandas as pd
import numpy as np
np.random.seed(123)
variables = ['X', 'Y', 'Z']
labels = ['ID_0','ID_1','ID_2','ID_3','ID_4']
X = np.random.random_sample([5,3])*10
df = pd.DataFrame(X, columns=variables, index=labels)
df
```

После выполнения приведенного выше исходного кода мы должны теперь увидеть следующую ниже таблицу данных DataFrame со случайно сгенерированными образцами:

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Выполнение иерархической кластеризации на матрице расстояний

Чтобы вычислить матрицу расстояний в качестве входа в алгоритм иерархической кластеризации, мы воспользуемся функцией pdist из подмодуля spatial.distance библиотеки SciPy:

```
>>>
from scipy.spatial.distance import pdist, squareform
row_dist = pd.DataFrame(squareform(
                            pdist(df, metric='euclidean')),
                        columns=labels, index=labels)
row_dist
```

При помощи приведенного выше фрагмента исходного кода мы вычислили евклидово расстояние между каждой парой точек (образцов) в нашем наборе данных, взяв за основу признаки X, Y и Z, и передали сжатую матрицу расстояний, – возвращенную функцией pdist, – на вход функции squareform для создания симметричной матрицы попарных расстояний, которая показана ниже:

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Далее мы применим к нашим кластерам агломеративную кластеризацию на основе метода полной связи, воспользовавшись для этого функцией linkage из подмодуля cluster.hierarchy библиотеки SciPy, которая вернет так называемую **матрицу связей**.

Однако, прежде чем вызвать функцию linkage, внимательно изучим документацию по этой функции[1]:

```
>>>
from scipy.cluster.hierarchy import linkage
help(linkage)
[…]
Parameters:
  y : ndarray
    A condensed or redundant distance matrix. A condensed
    distance matrix is a flat array containing the upper
    triangular of the distance matrix. This is the form
    that pdist returns. Alternatively, a collection of m
    observation vectors in n dimensions may be passed as
    an m by n array.

  method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.

  metric : str, optional
```

[1] Перевод документации:

```
Параметры:
  y : ndarray
Матрица сжатых или избыточных расстояний. Матрица сжатых расстояний - это плоский мас-
сив, содержащий верхний треугольник матрицы расстояний. Это та форма, в которой функ-
ция pdist возвращает результат. Как вариант в качестве массива размера m на n можно
передать коллекцию m векторов наблюдений в n измерениях
  method : str, факультативно
Используемый алгоритм связи. См. раздел методов связи ниже по поводу полного описания
  metric : str, факультативно
Используемая метрика расстояния. См. функцию distance.pdist по поводу списка допусти-
мых метрик расстояния
  Возвращает
Иерархическая кластеризация, кодированная в виде матрицы связей. – Прим. перев.
```

```
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.

  Returns:
    Z : ndarray
      The hierarchical clustering encoded as a linkage matrix.
[…]
```

Основываясь на описании функции, мы заключаем, что можем использовать сжатую матрицу расстояний (верхний треугольник) из функции pdist как входной атрибут. Как вариант можно также передать массив исходных данных и использовать евклидову метрику euclidean в качестве аргумента функции связи linkage. Однако мы не должны использовать квадратную матрицу расстояний, которую мы определили ранее, поскольку она даст значения расстояний, отличающиеся от ожидаемых. Резюмируя, приведем ниже три возможных сценария.

- **Неправильный подход**: используется квадратная матрица расстояний. Исходный код следующий:

```
from scipy.cluster.hierarchy import linkage
row_clusters = linkage(row_dist,
                       method='complete',
                       metric='euclidean')
```

- **Правильный подход**: используется сжатая матрица расстояний. Исходный код следующий:

```
row_clusters = linkage(pdist(df, metric='euclidean'),
                            method='complete')
```

- **Правильный подход**: используется матрица входных образцов. Исходный код следующий:

```
row_clusters = linkage(df.values,
                       method='complete',
                       metric='euclidean')
```

Чтобы повнимательнее рассмотреть результаты кластеризации, можно преобразовать их в таблицу данных библиотеки pandas DataFrame (рекомендуем для просмотра блокнот Jupyter). Это делается следующим образом:

```
pd.DataFrame(row_clusters,
     columns=['метка строки 1',
              'метка строки 2',
              'расстояние',
              'число элементов в класт.'],
     index=['кластер %d' %(i+1) for i in
            range(row_clusters.shape[0])])
```

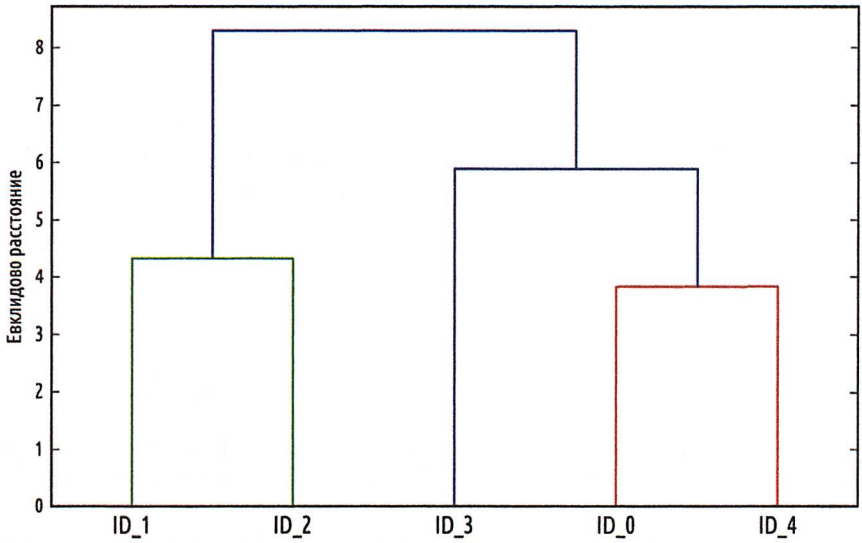
Как показано в следующей ниже таблице, матрица связей состоит из нескольких строк, где каждая строка представляет одно объединение. Первый и второй столбцы обозначают наиболее различающихся членов в каждом кластере, и третий столбец сообщает о расстоянии между этими членами. Последний столбец возвращает число членов в каждом кластере.

	Метка строки 1	Метка строки 2	Расстояние	Число элементов в кластере
Кластер 1	0.0	4.0	3.835396	2.0
Кластер 2	1.0	2.0	4.347073	2.0
Кластер 3	3.0	5.0	5.899885	3.0
Кластер 4	6.0	7.0	8.316594	5.0

Вычислив матрицу связей, теперь можно визуализировать результаты в форме дендограммы:

```
from scipy.cluster.hierarchy import dendrogram
# сделать дендограмму черной (часть 1/2)
# from scipy.cluster.hierarchy import set_link_color_palette
# set_link_color_palette(['black'])
row_dendr = dendrogram(row_clusters,
                       labels=labels,
                       # сделать дендограмму черной (часть 2/2)
                       # color_threshold=np.inf
                       )
plt.tight_layout()
plt.ylabel('Евклидово расстояние')
plt.show()
```

Если вы будете выполнять приведенный выше исходный код или читать электронную версию этой книги, то заметите, что ветки в итоговой дендограмме показаны в различных цветах. Окрашивающая схема получена на основании списка цветов библиотеки matplotlib, которые зациклены для порогов расстояний в дендограмме. Например, для показа дендограммы в черном цвете вы можете раскомментировать соответствующие участки, которые вставлены в приведенный выше исходный код.



Такая древовидная диаграмма резюмирует разные кластеры, сформированные во время агломеративной иерархической кластеризации; например, мы видим, что, ос-

новываясь на евклидовой метрике расстояния, образцы **ID_0** и **ID_4**, а затем **ID_1** и **ID_2** являются самыми похожими.

Прикрепление дендограмм к теплокарте

В практических приложениях дендограммы иерархической кластеризации часто используются в сочетании с **теплокартой**, позволяющей представлять отдельные значения в матрице образцов цветовым кодом. В этом разделе мы рассмотрим, как прикреплять дендограмму к графику теплокарты и соответствующим образом упорядочивать строки в теплокарте.

Однако во время выполнения процедуры прикрепления дендограммы к теплокарте могут появиться небольшие сложности, поэтому пройдем ее в пошаговом режиме:

1. При помощи атрибута add_axes создать новый объект figure и задать позицию осей X и Y, ширину и высоту дендограммы. Кроме того, повернуть дендограмму на 90° против часовой стрелки. Исходный код следующий:

```
fig = plt.figure(figsize=(8,8), facecolor='white')
axd = fig.add_axes([0.09,0.1,0.2,0.6])
row_dendr = dendrogram(row_clusters, orientation='right')
# примечание: для matplotlib >= v1.5.1, использовать orientation='left'
```

2. Далее переупорядочить данные в нашей исходной таблице данных DataFrame согласно меткам кластеризации, к которым можно обратиться из объекта дендограммы, являющегося по существу словарем Python, по ключу leaves. Исходный код следующий:

```
df_rowclust = df.ix[row_dendr['leaves'][::-1]]
```

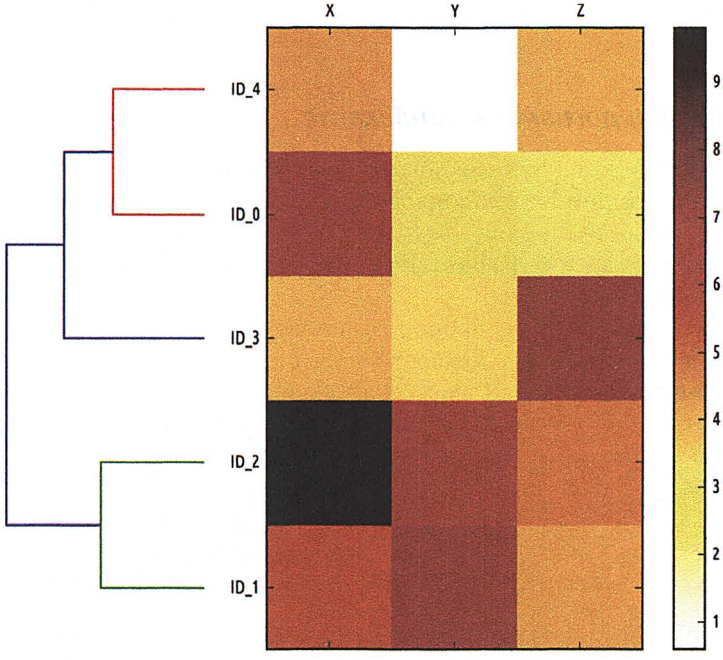
3. Теперь построить теплокарту из переупорядоченной таблицы данных DataFrame и расположить ее прямо напротив дендограммы:

```
axm = fig.add_axes([0.23,0.1,0.6,0.6])
cax = axm.matshow(df_rowclust,
                  interpolation='nearest', cmap='hot_r')
```

4. Наконец, изменить эстетику теплокарты, удалив деления осей и спрятав линии сетки. Кроме того, добавить цветную полосу и назначить меткам делений соответственно оси X и Y имена признаков и образцов. Исходный код следующий:

```
axd.set_xticks([])
axd.set_yticks([])
for i in axd.spines.values():
    i.set_visible(False)
fig.colorbar(cax)
axm.set_xticklabels([''] + list(df_rowclust.columns))
axm.set_yticklabels([''] + list(df_rowclust.index))
plt.show()
```

После выполнения предыдущих шагов теплокарта должна быть показана вместе с дендограммой:



Как видно, порядок следования строк в теплокарте отражает кластеризацию образцов в дендограмме. В дополнение к простой дендограмме цветокодированные значения каждого образца и признака на теплокарте предоставляют неплохое резюме набора данных.

Применение агломеративной кластеризации в scikit-learn

В этом разделе мы увидели, как выполнять агломеративную иерархическую кластеризацию, используя для этого библиотеку SciPy. Однако в библиотеке scikit-learn тоже имеется реализация агломеративной кластеризации – класс AgglomerativeClustering, позволяющий выбирать число кластеров, которые мы хотим получить. Он часто используется, в случае если надо подрезать иерархическое кластерное дерево. Установив параметр n_cluster равным 2, теперь сгруппируем образцы в две группы с использованием того же самого метода полной связи на основе евклидовой метрики расстояния, как и прежде:

```
>>>
from sklearn.cluster import AgglomerativeClustering
ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
labels = ac.fit_predict(X)
print('Метки кластеров: %s' % labels)
Метки кластеров: [0 1 1 0 0]]
```

Глядя на предсказанные метки кластеров, мы видим, что 1-й, 4-й и 5-й образцы (**ID_0**, **ID_3** и **ID_4**) были назначены кластеру 0, а образцы **ID_1** и **ID_2** – кластеру 1, что согласуется с результатами, которые мы можем наблюдать на дендограмме.

Локализация областей высокой плотности алгоритмом DBSCAN

Учитывая, что в этой главе мы не сможем охватить целый ряд разных алгоритмов кластеризации, по крайней мере, представим еще один подход, применяемый в кластеризации: **плотностная пространственная кластеризация приложений с присутствием шума** (density-based spatial clustering of applications with noise, DBSCAN). Понятие плотности в DBSCAN определяется как число точек внутри указанного радиуса ε.

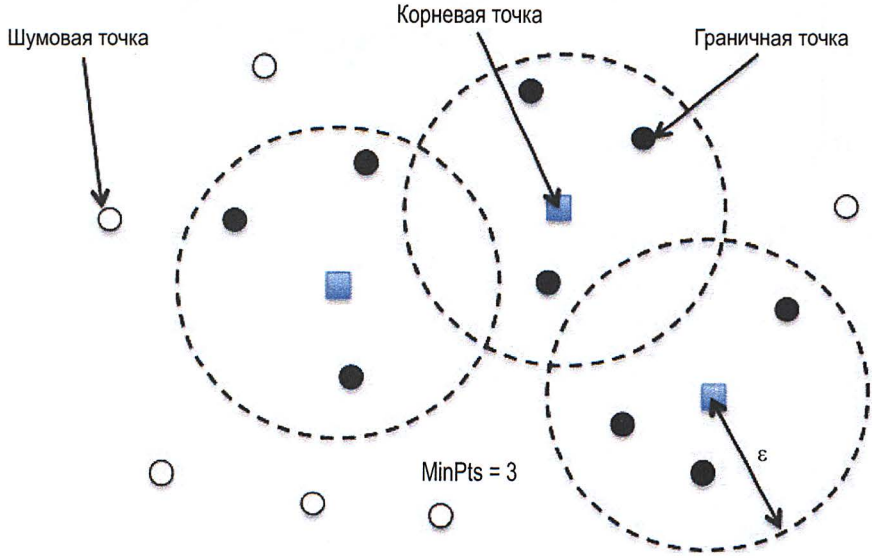
В DBSCAN каждому образцу (точке) назначается специальная метка, при этом используются следующие критерии:

- точка рассматривается как **корневая**, если, по меньшей мере, указанное число окрестных точек (MinPts) попадает в пределы указанного радиуса ε;
- **граничная точка** – это точка, имеющая соседей меньше, чем MinPts в пределах ε, но лежащая в пределах радиуса ε корневой точки;
- все остальные точки, которые не являются ни корневыми, ни граничными точками, рассматриваются как **шумовые точки**.

После маркировки точек как корневых, граничных либо шумовых алгоритм DBSCAN можно резюмировать в двух простых шагах:

1) сформировать для каждой корневой точки отдельный кластер либо связную группу корневых точек (корневые точки являются связными, в случае если они расположены не дальше, чем ε);
2) назначить каждую граничную точку кластеру соответствующей корневой точки.

Прежде чем приступить к реализации алгоритма DBSCAN и чтобы глубже разобраться, как может выглядеть результат его работы, резюмируем в нижеследующем рисунке то, что вы узнали о корневых, граничных и шумовых точках:

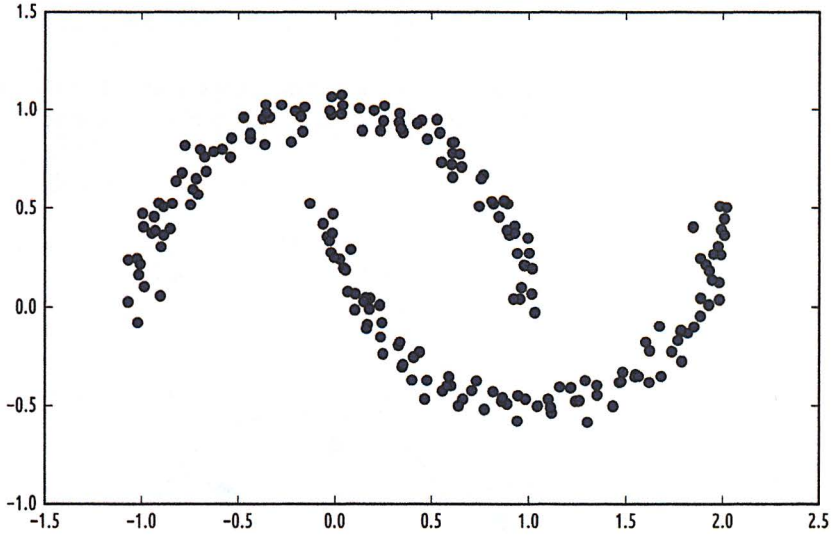


Одно из основных преимуществ использования алгоритма DBSCAN состоит в том, что он не делает допущения о сферичной форме кластеров, как в алгоритме *k* средних. Кроме того, алгоритм DBSCAN отличается от кластеризации по методу *k* средних и иерархической кластеризации тем, что он с необходимостью не назначает каждую точку кластеру и одновременно способен удалять шумовые точки.

В качестве иллюстрации создадим новый набор данных из структур в форме полумесяца, чтобы сравнить кластеризацию по методу *k* средних, иерархическую и DBSCAN:

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200,
                  noise=0.05,
                  random_state=0)
plt.scatter(X[:,0], X[:,1])
plt.show()
```

Как видно на получившемся графике, имеются две видимые группы в форме полумесяцев, состоящих из 100 точек образцов каждая:

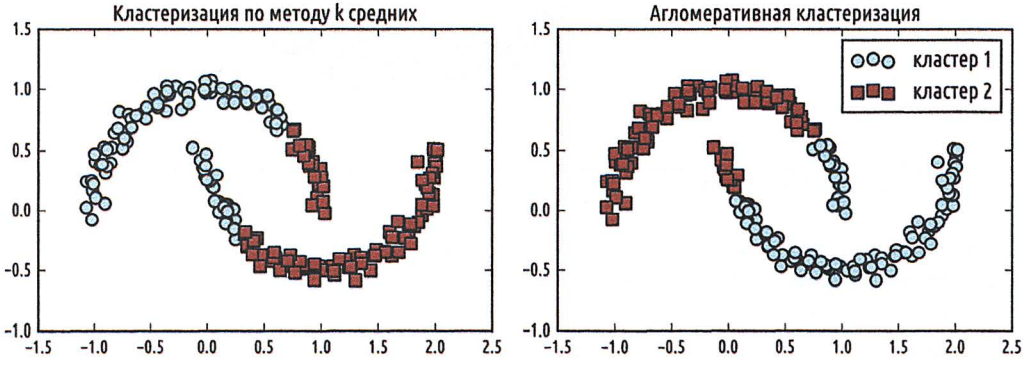


Начнем с применения алгоритма *k* средних и кластеризации с полной связью, чтобы увидеть, сможет ли один из этих ранее обсуждавшихся алгоритмов кластеризации успешно идентифицировать фигуры полумесяца как отдельные кластеры. Соответствующий исходный код выглядит следующим образом:

```
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8,3))
km = KMeans(n_clusters=2,
            random_state=0)
y_km = km.fit_predict(X)
ax1.scatter(X[y_km==0,0],
            X[y_km==0,1],
            c='lightblue',
            marker='o',
            s=40,
```

```
            label='кластер 1')
ax1.scatter(X[y_km==1,0],
            X[y_km==1,1],
            c='red',
            marker='s',
            s=40,
            label='кластер 2')
ax1.set_title('Кластеризация по методу k средних»)
ac = AgglomerativeClustering(n_clusters=2,
                             affinity='euclidean',
                             linkage='complete')
y_ac = ac.fit_predict(X)
ax2.scatter(X[y_ac==0,0],
            X[y_ac==0,1],
            c='lightblue',
            marker='o',
            s=40,
            label='кластер 1')
ax2.scatter(X[y_ac==1,0],
            X[y_ac==1,1],
            c='red',
            marker='s',
            s=40,
            label='кластер 2')
ax2.set_title('Агломеративная кластеризация')
plt.legend()
plt.show()
```

Основываясь на визуально представленных результатах кластеризации, мы видим, что алгоритм *k* средних неспособен разделить эти два кластера, а алгоритм иерархической кластеризации был поставлен этими сложными фигурами под сомнение:

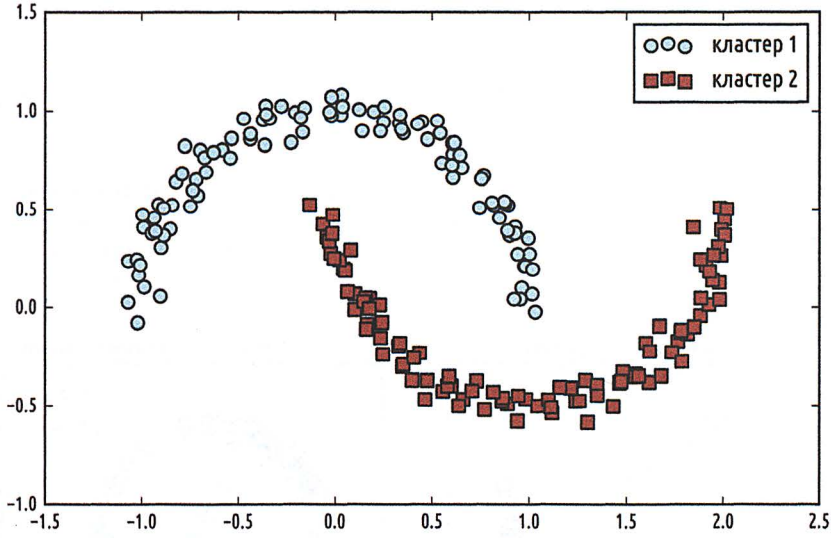


Наконец, на этом же наборе данных проверим работу алгоритма DBSCAN, чтобы увидеть, сможет ли он найти два кластера в форме полумесяцев при помощи плотностного подхода:

```
from sklearn.cluster import DBSCAN
db = DBSCAN(eps=0.2,
            min_samples=5,
            metric='euclidean')
```

```
y_db = db.fit_predict(X)
plt.scatter(X[y_db==0,0],
            X[y_db==0,1],
            c='lightblue',
            marker='o',
            s=40,
            label='кластер 1')
plt.scatter(X[y_db==1,0],
            X[y_db==1,1],
            c='red',
            marker='s',
            s=40,
            label='кластер 2')
plt.legend()
plt.show()
```

Алгоритм DBSCAN может успешно обнаруживать фигуры полумесяца, что подчеркивает одну из сильных сторон алгоритма DBSCAN (кластеризация данных произвольной формы):



Однако мы должны также отметить некоторые недостатки алгоритма DBSCAN. С растущим числом признаков в наборе данных – при заданном фиксированном размере тренировочного набора – отрицательный эффект *проклятия размерности* увеличивается. Эта проблема в особенности проявляется, если мы используем евклидову метрику расстояния. Однако проблема *проклятия размерности* не уникальна для DBSCAN; она влияет и на другие алгоритмы кластеризации, которые используют евклидову метрику расстояния, например алгоритм кластеризации по методу k средних и иерархические алгоритмы кластеризации. Вдобавок в DBSCAN имеются два гиперпараметра (MinPts и ε), которые должны быть оптимизированы с целью получения хороших результатов кластеризации. Нахождение хорошей комбинации параметров MinPts и ε может быть проблематичным, в случае если в наборе данных разницы в плотностях относительно большие.

До настоящего момента мы увидели три самые фундаментальные категории алгоритмов кластеризации: алгоритм k средних на основе прототипов, алгоритм агломеративной иерархической кластеризации и плотностная кластеризация на основе алгоритма DBSCAN. Однако стоит упомянуть и четвертый класс более продвинутых алгоритмов кластеризации, которые мы не затронули в этой главе: **графовые алгоритмы кластеризации**. Вероятно, наиболее выдающиеся представители семейства графовых алгоритмов кластеризации представлены **алгоритмами спектральной кластеризации**. Несмотря на то что существует много разных реализаций спектральной кластеризации, все они объединены тем, что используют собственные векторы из матрицы подобия для получения кластерных связей. Поскольку спектральная кластеризация выходит за рамки этой книги, вы можете прочитать превосходное учебное пособие Ульрике фон Люксбург, чтобы узнать об этой теме больше (U. Von Luxburg, «A Tutorial on Spectral Clustering», Statistics and computing, 17(4):395–416, 2007 («*Учебное пособие по спектральной кластеризации*»)). Оно находится в свободном доступе на arXiv по прямой ссылке http://arxiv.org/pdf/0711.0189v1.pdf.

Отметим, что на практике не всегда очевидно, который алгоритм на заданном наборе данных будет выполняться лучше всего, в особенности если данные поступают в многочисленных размерностях, затрудняя визуализацию либо делая ее невозможной. Более того, важно подчеркнуть, что успешная кластеризация не зависит только от алгоритма и его гиперпараметров. Скорее, более важными могут быть выбор надлежащей метрики расстояния и использование знаний в предметной области, которые помогают направить экспериментальный поиск наилучшей конфигурации в нужное русло.

Резюме

В этой главе вы узнали о трех разных алгоритмах кластеризации, которые способны помочь в обнаружении скрытых структур или информации в данных. Мы начали эту главу с подхода на основе прототипов, алгоритма k средних, который кластеризует образцы в сферические формы на основе заданного числа кластерных центроидов. Учитывая, что кластеризация является методом обучения без учителя, мы не можем себе позволить роскошь раз и навсегда определенных данных о метках классов с целью оценки качества модели. Поэтому мы обратились к полезным внутренним метрикам оценки качества, таким как метод локтя или силуэтный анализ, в качестве попытки количественно определить качество кластеризации.

Затем мы проанализировали другой подход к кластеризации: агломеративную иерархическую кластеризацию. Иерархическая кластеризация не требует предварительно указывать число кластеров, и ее результат может визуализироваться в виде дендограммы, которая помогает интерпретировать результаты. Последним алгоритмом кластеризации, который мы рассмотрели в этой главе, был алгоритм DBSCAN, который группирует точки, основываясь на локальных плотностях, и способен обрабатывать выбросы и идентифицировать нешаровидные фигуры.

После этой экскурсии в область обучения без учителя теперь пора представить некоторые наиболее захватывающие алгоритмы машинного обучения для обучения с учителем: многослойные искусственные нейронные сети. После недавнего всплес-

ка интереса нейронные сети еще раз стали самой горячей темой в научных исследованиях в области машинного обучения. Благодаря разработанным в последние годы алгоритмам глубокого обучения нейронные сети воспринимаются как передовая технология для многих сложных задач, таких как классификация изображений и распознавание речи. В *главе 12 «Тренировка искусственных нейронных сетей для распознавания изображений»* мы построим с нуля нашу собственную многослойную нейронную сеть. В *главе 13 «Распараллеливание тренировки нейронных сетей при помощи Theano»* мы представим мощные библиотеки, которые способны помочь наиболее эффективно тренировать сложные сетевые архитектуры.

Глава 12

Тренировка искусственных нейронных сетей для распознавания изображений

Как вы, возможно, знаете, **глубокое обучение** сейчас привлекает к себе много внимания СМИ и является, вне всякого сомнения, самой горячей темой в сфере машинного обучения. Глубокое обучение может пониматься как набор алгоритмов, разработанных для наиболее эффективной тренировки **искусственных нейронных сетей**, состоящих из множества слоев. В этой главе вы изучите основы икусственных нейронных сетей, благодаря чему будете хорошо подготовлены для дальнейшего изучения самых захватывающих областей научного исследования в сфере машинного обучения, а также продвинутых программных библиотек Python для работы с глубоким обучением.

Мы затронем следующие темы:

- получение концептуального представления о многослойных нейронных сетях;
- тренировка нейронных сетей для задачи классификации изображений;
- реализация мощного алгоритма обратного распространения ошибки;
- отладка реализаций нейронных сетей.

Моделирование сложных функций искусственными нейронными сетями

В начале этой книги наше путешествие началось в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* обсуждением алгоритмов машинного обучения с искусственными нейронами. Искусственные нейроны составляют базовые элементы многослойных искусственных нейронных сетей и станут темой обсуждения данной главы. Ключевая идея, лежащая в основе искусственных нейронных сетей, опирается на гипотезы и модели того, каким образом человеческий мозг работает для решения сложных практических задач. Несмотря на то что в последние годы искусственные нейронные сети получили большую популярность, ранние исследования нейронных сетей восходят к 1940-м, когда Уоррен Маккалок и Уолтер Питт впервые дали описание того, как могли бы работать нейроны. Однако в последующие десятилетия вслед за первой реализацией модели **нейрона Маккалока–Питта**, персептрона Розенблатта в 1950-х, многие научные исследователи и практики в области машинного обучения постепенно стали терять интерес к нейронным сетям, поскольку никто не мог предложить хорошего решения задачи тренировки многослойной нейронной сети. Впоследствии интерес к нейронным

сетям вспыхнул вновь, когда в 1986 г. Д. Румелхарт, Г. Хинтон и Р. Дж. Уильямс повторно открыли и популяризировали **алгоритм обратного распространения ошибки** (backpropagation) для более эффективной тренировки нейронных сетей (Rumelhart D. E., Hinton G. E., Williams R. J. (1986). *Learning Representations by Back-propagating Errors* («Обучение представлений методом обратного распространения ошибки»). Nature 323 (6088): 533-536), и мы обсудим этот алгоритм более подробно далее в этой главе.

В течение предыдущего десятилетия многие другие важные прорывные работы привели к тому, что теперь мы называем алгоритмами глубокого обучения; они используются для создания из немаркированных данных **детекторов признаков** (feature detectors), которые применяются для предварительной тренировки глубоких нейронных сетей – нейронных сетей, скомпонованных из многих слоев. Нейронные сети – горячая тема не только в научных кругах, но и в крупных технологических компаниях, таких как Facebook, Microsoft и Google, которые выделяют огромные инвестиции на искусственные нейронные сети и научные исследования в области глубокого обучения. На сегодняшний день сложные нейронные сети, приводимые в действие алгоритмами глубокого обучения, рассматриваются как передовая технология, когда дело доходит до решения таких сложных задач, как распознавание изображений и голоса. Популярные примеры продуктов из нашей повседневной жизни, которые приводятся в действие технологией глубокого обучения, представлены службами Google по поиску изображений и переводу (Google Перевод), приложением для смартфонов, которое способно автоматически распознавать текст в изображениях с его последующим переводом на 20 языков в реальном времени (http://googleresearch.blogspot.com/2015/07/how-google-translate-squeezes-deep.html).

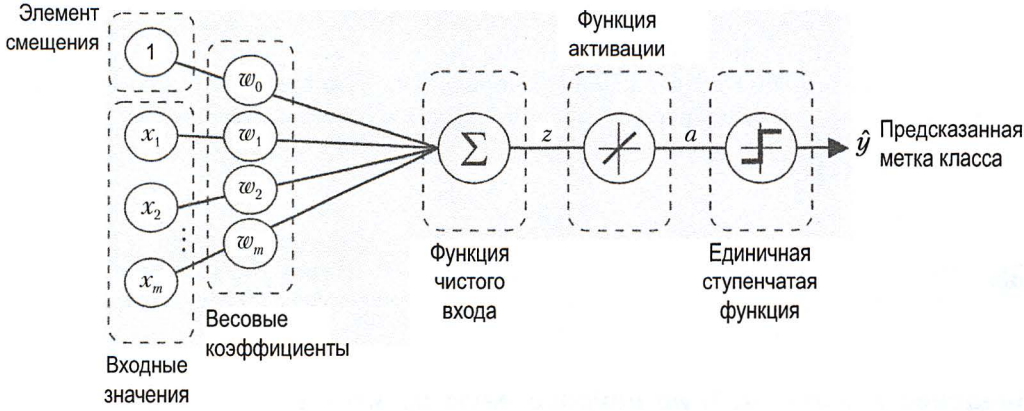
В крупнейших технологических компаниях в процессе активной разработки находится много других увлекательных приложений с использованием глубоких нейронных сетей, например приложение DeepFace компании Facebook для маркировки изображений (Y. Taigman, M. Yang, M. Ranzato and L. Wolf, «DeepFace: Closing the gap to human-level performance in face verification». In Computer Vision and Pattern Recognition CVPR, 2014 IEEE Conference, p. 1701–1708 (*«DeepFace: преодоление отрыва от человеческого уровня качества в идентификации лиц»*)) и приложение DeepSpeech компании Baidu, которое способно обрабатывать голосовые запросы на китайском языке (Hannun A., Case C., Casper J., Catanzaro B., Diamos G., Elsen E., Prenger R., Satheesh S., Sengupta S., Coates A. et al., «DeepSpeech: Scaling up end-to-end speech recognition». arXiv preprint arXiv:1412.5567, 2014 (*«DeepSpeech: масштабирование сквозного распознавания речи»*)). Кроме того, фармацевтическая промышленность недавно начала использовать методы глубокого обучения для изобретения лекарственных препаратов и предсказания токсичности; исследование показало, что эти новые методы существенно превышают качество традиционных методов виртуального скрининга[1] (Unterthiner T., Mayr A., Klambauer G., Hochreiter S. *Toxicity prediction using deep learning* («Предсказание токсичности при помощи глубокого обучения»). arXiv preprint arXiv:1503.01445, 2015).

---

[1] В химической отрасли *виртуальный скрининг* (virtual screening) – выбор соединений путем оценки их желательности в вычислительной модели. – *Прим. перев.*

Краткое резюме однослойных нейронных сетей

Эта глава полностью посвящена многослойным нейронным сетям: тому, как они работают и как их тренировать для решения сложных задач. Однако, прежде чем мы углубимся в архитектуру отдельно взятой многослойной нейронной сети, кратко повторим некоторые понятия однослойных нейронных сетей, которые мы представили в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*, в частности алгоритм **адаптивного линейного нейрона** (ADALINE), который показан на нижеследующем рисунке:



В *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* мы реализовали алгоритм ADALINE для выполнения задачи бинарной классификации и применили алгоритм оптимизации на основе **градиентного спуска**, чтобы извлечь весовые коэффициенты модели. В каждой **эпохе** (проходе по тренировочному набору) мы обновляли весовой вектор $\boldsymbol{w}$, используя следующее ниже правило обновления:

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

где $\Delta\boldsymbol{w} = -\eta\nabla(\boldsymbol{w})$.

Другими словами, мы вычисляли градиент, основываясь на всем тренировочном наборе, и обновляли веса модели, делая шаг в противоположном от градиента направлении $\nabla J(\boldsymbol{w})$. Для того чтобы найти оптимальные веса модели, мы оптимизировали целевую функцию, которую определили как функцию стоимости на основе **суммы квадратичных ошибок** $J(\boldsymbol{w})$ (sum of squared errors, SSE). Кроме того, мы умножали градиент на **темп обучения** η, коэффициент, который мы выбрали тщательно, чтобы сбалансировать скорость обучения против риска непопадания по глобальному минимуму функции стоимости.

В оптимизации на основе градиентного спуска мы после каждой эпохи обновляли все веса одновременно, при этом мы определили частную производную для каждого веса w_j в векторе весов $\boldsymbol{w}$ следующим образом:

$$\frac{\delta}{\delta w_j} J(\boldsymbol{w}) = -\sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}.$$

Здесь $y^{(i)}$ – это целевая метка класса отдельно взятого образца $x^{(i)}$ и $a^{(i)}$ — **активация** нейрона, т. е. линейная функция в частном случае ADALINE. Кроме того, мы определили *функцию активации* $\phi(\cdot)$ как:

$$\phi(z) = z = a.$$

Здесь чистый вход z – это линейная комбинация весов, которая соединяет входной слой с выходным:

$$z = \sum_j w_j x_j = \boldsymbol{w}^T \boldsymbol{x}.$$

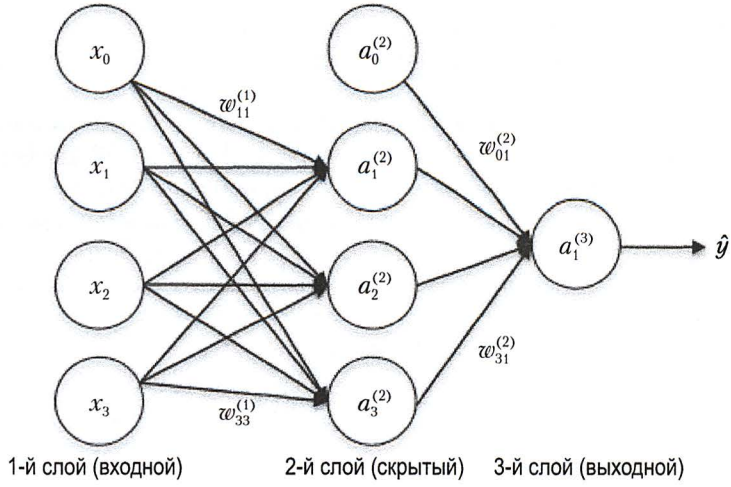
Для вычисления градиентного обновления мы использовали активацию $\phi(z)$, тогда как для предсказания мы реализовали **пороговую функцию** (функцию Хевисайда), которая запихивает непрерывнозначный выход в бинарные метки классов:

$$\hat{y} = \begin{cases} 1, & \text{если } g(z) \geq 0 \\ -1, & \text{иначе} \end{cases}$$

Отметим, что, несмотря на то что ADALINE состоит из двух слоев, одного входного и одного выходного слоя, он называется однослойной сетью вследствие его единственной связи между входным и выходным слоями.

Введение в многослойную нейросетевую архитектуру

В этом разделе мы увидим, как соединять два и более единичных нейрона в **многослойную нейронную сеть с прямым распространением сигналов**; этот особый тип сети также называется **многослойным персептроном** (МСП, MLP). Следующий ниже рисунок объясняет принцип работы MLP, состоящего из трех слоев: один входной слой, один **скрытый слой** и один выходной слой. Узлы (units) в скрытом слое полностью связаны с входным слоем, и соответственно, выходной слой полностью связан со скрытым слоем. Если такая сеть имеет более одного скрытого слоя, то мы также называем ее *глубокой* искусственной нейронной сетью.



Для создания более глубокой сетевой архитектуры мы можем добавить в MLP произвольное число скрытых слоев. Практически число слоев и узлов в нейронной сети можно представить как дополнительные **гиперпараметры**, которые мы хотим оптимизировать для данной практической задачи, используя для этого перекрестную проверку, которую мы обсудили в *главе 6 «Анализ наиболее успешных приемов оценивания моделей и тонкой настройки гиперпараметров»*.

Однако градиенты ошибки, которые мы вычислим позже методом обратного распространения ошибки, стали бы по мере добавления в сеть новых слоев во все возрастающей степени уменьшаться. Эта проблема *исчезновения градиента* еще более усложняет извлечение модели. Поэтому, чтобы предварительно натренировать такие глубокие нейросетевые структуры, были разработаны специальные алгоритмы под общим названием *глубокого обучения*.

Как показано на приведенном выше рисунке, мы обозначаем i-й узел активации в l-ом слое как $a_i^{(l)}$, узлы активации $a_0^{(1)}$ и $a_0^{(2)}$ – это соответственно **узлы смещения** (узлы постоянного смещения), которые мы установили равными 1. Активация узлов во входном слое – это просто вход плюс узел смещения:

$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}.$$

Каждый узел в слое l соединен весовым коэффициентом со всеми узлами в слое $l + 1$. Например, связь между k-ым узлом в слое l с j-ым узлом в слое $l + 1$ была бы записана как $w_{j,k}^{(l)}$. Отметим, что надстрочный индекс i в $x_m^{(i)}$ обозначает i-й образец, а не i-й слой. В следующих абзацах для ясности надстрочный индекс i мы часто будем опускать.

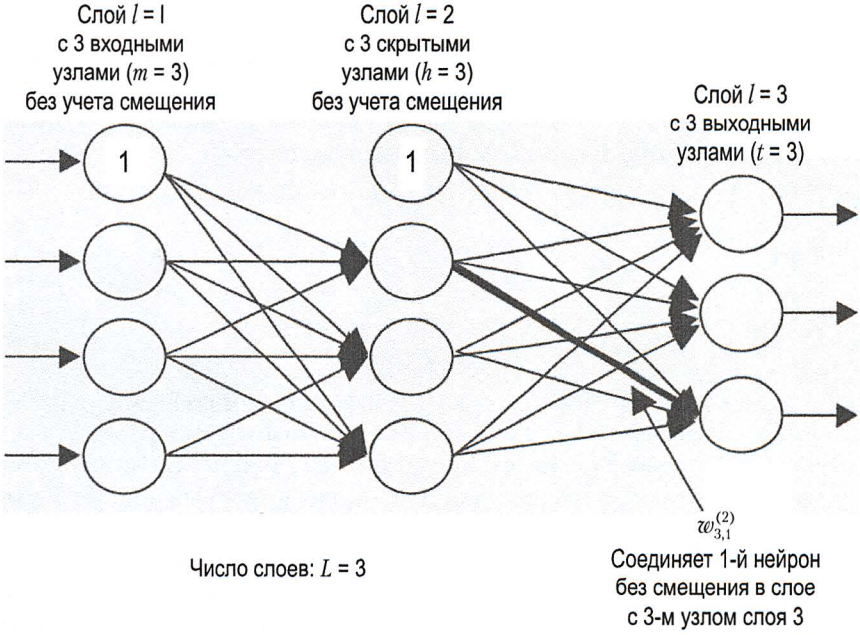
Хотя одного узла в выходном слое было бы достаточно, чтобы выполнить задачу бинарной классификации, на приведенном выше рисунке мы увидели более общую форму нейронной сети, позволяющую выполнять многоклассовую классификацию обобщенным методом «**один против всех**» (OvA). Чтобы лучше понять, как она работает, вспомним представление категориальных переменных в виде **прямого кода**, с которым мы познакомились в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»*. Например, мы бы закодировали три метки классов в хорошо знакомом наборе данных цветков ириса (`0=щетинистый, 1=разноцветный, 2=виргинский`) следующим образом:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Представление вектора в виде такого прямого кода позволяет решать задачи классификации с произвольным числом имеющихся в тренировочном наборе уникальных меток классов.

Если вы плохо знакомы со способами представления нейронных сетей, то терминология, связанная с (над- и подстрочными) индексами, поначалу может выглядеть немного озадачивающей. Вам, наверное, интересно, почему для обозначения весо-

вого коэффициента, который соединяет k-ый узел в слое l с j-ым узлом в слое $l + 1$, мы записали $w_{j,k}^{(l)}$, а не $w_{k,j}^{(l)}$. То, что поначалу может выглядеть немного необычным, обретет больший смысл в более поздних разделах, где мы займемся векторизацией представления нейронной сети. Например, мы обобщаем веса, которые соединяют входной и скрытый слои, матрицей $w^{(1)} \in \mathbb{R}^{h \times [m+1]}$, где h – это число скрытых узлов и $m + 1$ – число входных узлов плюс узел смещения. Поскольку эту форму записи важно усвоить, чтобы понимать смысл упоминаемых далее в этой главе понятий, подытожим то, что мы только что обсудили, описательной иллюстрацией упрощенного многослойного персептрона в конфигурации 3-4-3:



Активация нейронной сети методом прямого распространения сигналов

В этом разделе мы опишем процесс **прямого распространения сигналов** для вычисления выхода модели многослойного персептрона (MLP). Чтобы понять, как он вписывается в контекст обучения модели MLP, резюмируем процедуру обучения MLP тремя простыми шагами:

1) начиная с входного слоя, распространить по сети вперед образы (сигналы) тренировочных данных для генерирования выхода;
2) основываясь на выходе сети, рассчитать ошибку, подлежащую минимизации с использованием функции стоимости, которую мы опишем позже;
3) распространить ошибку назад, найти ее производную относительно каждого веса в сети и обновить модель.

В конце, после того как шаги были выполнены повторно для двух и более эпох и были извлечены веса MLP, мы используем прямое распространение сигналов, чтобы вычислить выход сети, и применяем пороговую функцию, чтобы получить

предсказанные метки классов в виде прямого кода, который мы описали в предыдущем разделе.

Теперь пройдемся по этим отдельным шагам метода прямого распространения, чтобы сгенерировать выход из образов в тренировочных данных. Поскольку каждый узел в скрытом слое соединен со всеми узлами во входном слое, сначала мы вычисляем активацию $a_1^{(2)}$. Она определяется следующим образом:

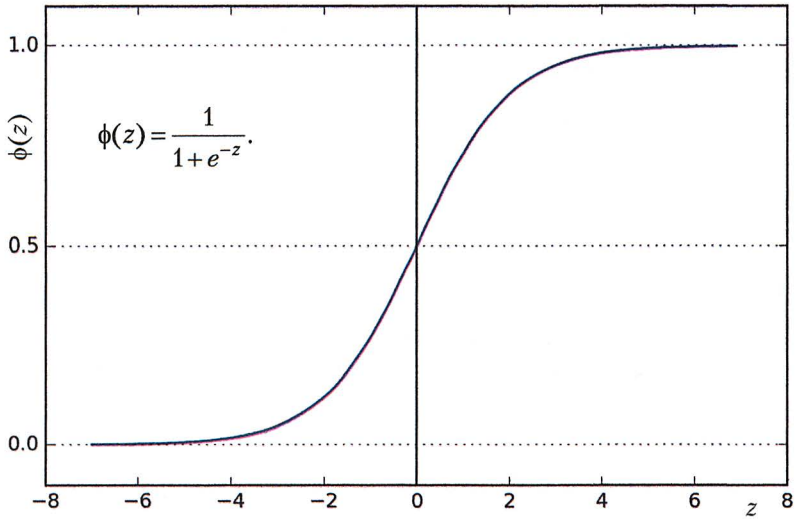
$$z_1^{(2)} = a_0^{(1)} w_{1,0}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + \ldots + a_m^{(1)} w_{1,m}^{(1)};$$

$$a_1^{(2)} = \phi(z_1^{(2)}).$$

Здесь $z_1^{(2)}$ – это чистый вход и $\phi(\cdot)$ – функция активации, которая для извлечения связывающих нейроны весов должна быть дифференцируемой на основе градиентного подхода. Чтобы суметь решать такие сложные задачи, как классификация изображений, в нашей модели MLP нужны нелинейные функции активации, например **сигмоидальная** (логистическая) функция активации, которую мы использовали в **логистической регрессии** в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*:

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

Как мы помним, сигмоидальная функция – это S-образная кривая, которая проецирует чистый вход z на логистическое распределение в диапазоне от 0 до 1, рассекая ось Y в $z = 0$, как показано на следующем ниже графике:



MLP является типичным примером искусственной нейронной сети *прямого распространения* сигналов. Термин «*прямое распространение сигналов*» обозначает, что каждый слой служит входом в следующий слой без циклов, в отличие от *рекуррентных нейронных сетей*, архитектуры, которую мы обсудим далее в этой главе. Термин «многослойный персептрон» может показаться немного озадачивающим, поскольку

искусственные нейроны в такой сетевой архитектуре – это, как правило, сигмоидальные узлы, а не *персептроны*. Нейроны в MLP можно интуитивно представить как узлы логистической регрессии, которые возвращают значения в непрерывном диапазоне между 0 и 1.

В целях эффективности и удобочитаемости исходного кода теперь напишем активацию в более компактной форме. Для этого воспользуемся понятиями из основ линейной алгебры, которые позволят **векторизовать** нашу программную реализацию в библиотеке NumPy без привлечения многократно вложенных и затратных циклов `for` языка Python:

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)};$$
$$\boldsymbol{a}^{(2)} = \phi(\boldsymbol{z}^{(2)}).$$

На этой странице везде, где встречается h, h можно представить как $h + 1$, чтобы учесть узел смещения (и скорректировать размерность).

Здесь $\boldsymbol{a}^{(1)}$ – это наш $[m + 1]\times 1$-мерный вектор признаков образца $\boldsymbol{x}^{(1)}$ плюс узел смещения. $\boldsymbol{W}^{(1)}$ – это $h\times[m + 1]$-мерная весовая матрица, где h – число скрытых узлов в нейронной сети. После матрично-векторного умножения получаем $h\times 1$-мерный вектор чистого входа $\boldsymbol{z}^{(2)}$ для вычисления активации $\boldsymbol{a}^{(2)}$ (где $\boldsymbol{a}^{(2)} \in \mathbb{R}^{h\times 1}$). Более того, мы можем обобщить это вычисление на все n образцов в тренировочном наборе:

$$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)}[\boldsymbol{A}^{(1)}]^T.$$

Здесь $\boldsymbol{A}^{(1)}$ – это теперь $n\times[m + 1]$-матрица, при этом матрично-матричное умножение в результате даст $h\times n$-мерную матрицу $\boldsymbol{Z}^{(2)}$ чистых входов. В конце к каждому значению в матрице чистых входов мы применяем функцию активации $\phi(\cdot)$ и получаем $h\times n$-матрицу активации $\boldsymbol{A}^{(2)}$ для следующего слоя (в данном случае выходного слоя):

$$\boldsymbol{A}^{(2)} = \phi(\boldsymbol{Z}^{(2)}).$$

Точно так же можно переписать активацию выходного слоя в векторизованной форме:

$$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{A}^{(2)}.$$

Здесь мы умножаем $t\times h$-матрицу $\boldsymbol{W}^{(2)}$ (t – число выходных узлов) на $h\times n$-мерную матрицу $\boldsymbol{A}^{(2)}$ для получения $t\times n$-мерной матрицы $\boldsymbol{Z}^{(3)}$ (столбцы в этой матрице – это выходы для каждого образца).

Наконец, мы применяем сигмоидальную функцию активации и получаем непрерывнозначный выход из нашей сети:

$$\boldsymbol{A}^{(3)} = \phi(\boldsymbol{Z}^{(3)}),\ \boldsymbol{A}^{(3)} \in \mathbb{R}^{t\times n}.$$

Классификация рукописных цифр

В предыдущем разделе мы затронули много теоретических положений касательно нейронных сетей, которые могут немного подавлять своей сложностью, в случае

если вы плохо знакомы с этой темой. Прежде чем продолжить обсуждение алгоритма обратного распространения ошибки, который используется для извлечения весов модели MLP, немного отдохнем от теории и взглянем на нейронную сеть в действии.

Теория нейронных сетей может быть довольно сложной для понимания. В связи с этим предлагаем два дополнительных ресурса, где некоторые понятия, которые мы обсуждаем в этой главе, разбираются более подробно:

- Hastie T., Friedman J., Tibshirani R. *The Elements of Statistical Learning* («Элементы статистического обучения»). Vol. 2. Springer, 2009;
- Bishop C. M. et al. *Pattern Recognition and Machine Learning* («Распознавание образов и машинное обучение»). Vol. 1. Springer New York, 2006.

В этом разделе мы натренируем нашу первую многослойную нейронную сеть задаче классификации рукописных цифр из популярного набора данных **MNIST** – сокращенно от Mixed National Institute of Standards and Technology, смешанная база данных Национального института стандартов и технологий, созданная Я. ЛеКуном и др. и служащая популярным эталонным набором данных для алгоритмов машинного обучения (LeCun Y., Bottou L., Bengio Y., Haffner P. *Gradient-based Learning Applied to Document Recognition* («Обучение на основе градиентных методов применительно к распознаванию документов»). Proceedings of the IEEE, 86(11):2278-2324, November 1998).

Получение набора данных MNIST

Набор данных MNIST находится в общем доступе на http://yann.lecun.com/exdb/mnist/ и состоит из следующих четырех частей:

- **тренировочный набор изображений**: train-images-idx3-ubyte.gz (9.9 Мб, 47 Мб в разархивированном виде и 60 000 образцов);
- **тренировочный набор меток**: train-labels-idx1-ubyte.gz (29 Кб, 60 Кб в разархивированном виде и 60 000 меток);
- **тестовый набор изображений**: t10k-images-idx3-ubyte.gz (1.6 Мб, 7.8 Мб в разархивированном виде и 10 000 образцов);
- **тестовый набор меток**: t10k-labels-idx1-ubyte.gz (5 Кб, 10 Кб в разархивированном виде и 10 000 меток).

Набор данных MNIST был составлен из двух наборов данных **Национального института стандартов и технологий,** США (NIST). Учебный набор состоит из рукописных цифр, взятых у 250 разных людей, 5% из которых были учениками средней школы и остальные 50% – служащими Бюро переписи населения. Отметим, что тестовый набор содержит рукописные цифры, взятые у разных людей в тех же самых пропорциях.

После скачивания файлов рекомендуем, находясь в локальном каталоге загрузки набора данных mnist, разархивировать файлы утилитой gzip под Unix/Linux из терминала, используя для эффективности следующую ниже команду:

```
gzip *ubyte.gz -d
```

Как вариант вы можете воспользоваться своим предпочитаемым архиватором, в случае если работаете на компьютере под управлением Microsoft Windows. Изображения хранятся в байтовом формате, и далее мы прочитаем их в массивы библиотеки NumPy, которые мы применим для тренировки и тестирования нашей

реализации MLP:

```
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Загрузить данные MNIST из пути `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                               % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)

    return images, labels
```

Функция load_mnist возвращает два массива, первый – это $n \times m$-мерный массив NumPy (images), где n – число образцов и m – число признаков. Тренировочный набор данных состоит из 60 000 тренировочных цифр; тестовый набор соответственно содержит 10 000 образцов. Изображения в наборе данных MNIST представляют собой растры размером 28×28 пикселов, в котором каждый пиксел представлен значением интенсивности полутоновой шкалы. Здесь мы разворачиваем 28×28 пикселов в одномерные векторы-строки, соответствующие строкам в нашем массиве изображений (784 пиксела в расчете на строку или изображение). Второй возвращаемой функцией load_mnist (labels) массив содержит соответствующую целевую переменную, метки классов (целые числа 0–9) рукописных цифр.

Способ считывания изображения может сперва показаться немного необычным:

```
magic, n = struct.unpack('>II', lbpath.read(8))
labels = np.fromfile(lbpath, dtype=np.int8)
```

Чтобы разобраться в том, как эти две строки исходного кода работают, посмотрим на описание набора данных на веб-сайте MNIST:

[смещение]	[тип]	[значение]	[описание]
0000	32-разрадное целое	0x00000801(2049)	магическое число (сначала MSB)
0004	32-разрядное целое	60000	число элементов
0008	беззнаковый байт	??	метка
0009	беззнаковый байт	??	метка
....			

xxxx	беззнаковый байт	??	метка

При помощи двух строк приведенного выше исходного кода мы сначала считываем *магическое число*, которое представляет собой описание формата файла или файлового протокола, а также *число элементов* (*n*) из файлового буфера и только потом методом fromfile считываем последующие байты в массив NumPy. Значение параметра fmt (>II), которое мы передали в качестве аргумента в struct.unpack, имеет две части:

- >: обратный порядок байтов (определяет порядок, в котором хранится последовательность байтов); если термины *прямой* и *обратный порядок байтов* вам незнакомы, то превосходную статью по данной теме можно найти в Википедии (https://ru.wikipedia.org/wiki/Порядок_байтов);
- I: беззнаковое целое число.

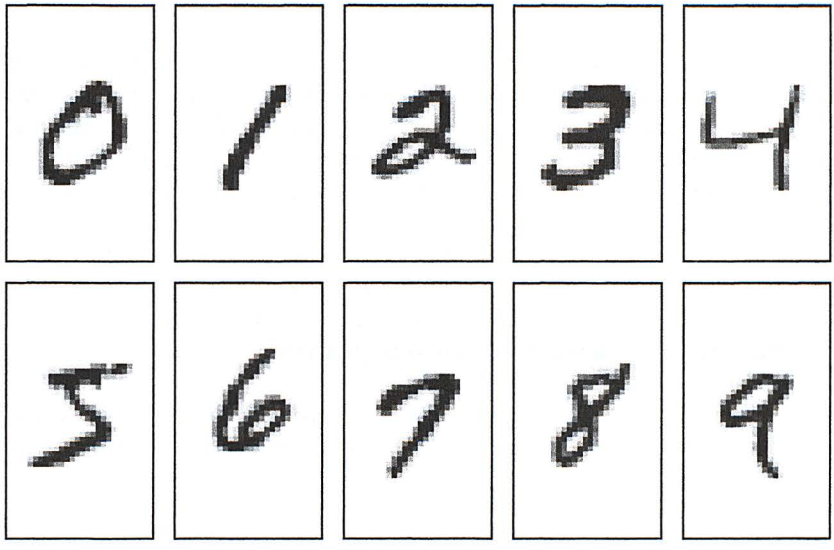
Выполнив следующий ниже фрагмент исходного кода, теперь загрузим 60 000 тренировочных экземпляров и 10 000 тестовых образцов из каталога mnist, куда мы разархивировали набор данных MNIST:

```
>>>
X_train, y_train = load_mnist('mnist', kind='train')
print('Тренировка - строки: %d, столбцы: %d'
      % (X_train.shape[0], X_train.shape[1]))
Тренировка - строки: 60 000, столбцы: 784

X_test, y_test = load_mnist('mnist', kind='t10k')
print('Тестирование - строки: %d, Столбцы: %d'
      % (X_test.shape[0], X_test.shape[1]))
Тестирование - строки: 10 000, столбцы: 784
```

Чтобы получить представление о том, как выглядят изображения в MNIST, покажем примеры цифр 0–9 в наглядном виде после приведения 784-пиксельных векторов из нашей матрицы признаков в исходные изображения в формате 28×28, которые можно вывести на экран при помощи imshow библиотеки matplotlib:

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()
for i in range(10):
    img = X_train[y_train == i][0].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```

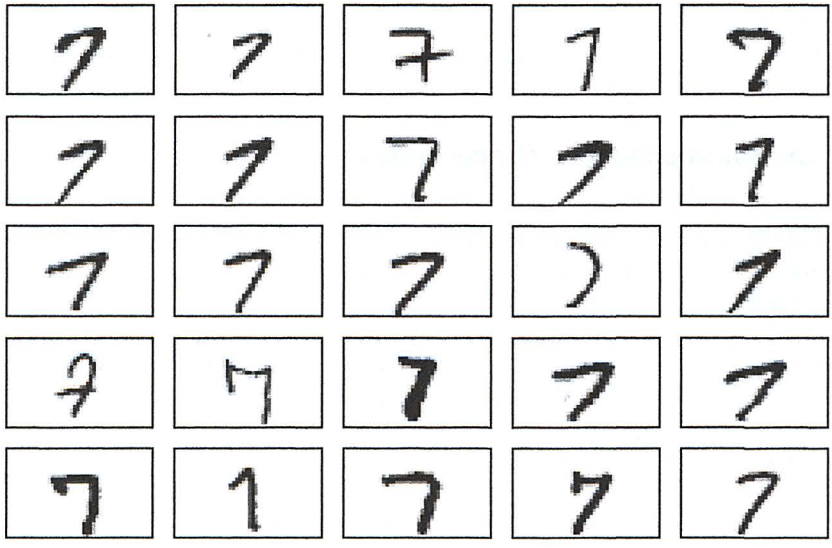
В результате мы должны увидеть составной график из 2×5 частей с репрезентативными изображениями каждой уникальной цифры:



В дополнение к этому также построим график с повторными примерами одной и той же цифры, чтобы увидеть, насколько в действительности эти примеры почерка друг от друга отличаются:

```
fig, ax = plt.subplots(nrows=5,
                       ncols=5,
                       sharex=True,
                       sharey=True,)
 ax = ax.flatten()
for i in range(25):
    img = X_train[y_train == 7][i].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```

В результате выполнения исходного кода мы должны видеть первые 25 вариантов цифры 7.



Как вариант можно сохранить данные изображений и метки набора данных MNIST в CSV-файлы, с тем чтобы их можно было открывать в программах, которые не поддерживают используемого в них специального байтового формата. Однако стоит напомнить, что файловый формат CSV займет значительный объем вашего дискового пространства, как показано в списке ниже:

- train_img.csv: 109.5 Мб;
- train_labels.csv: 120 Кб;
- test_img.csv: 18.3 Мб;
- test_labels.csv: 20 Кб.

Если мы решаем сохранить данные в CSV-файлах, то после загрузки данных MNIST в массивы NumPy в нашем сеансе Python мы можем выполнить следующий ниже фрагмент исходного кода:

```
np.savetxt('train_img.csv', X_train,    fmt='%i', delimiter=',')
np.savetxt('train_labels.csv', y_train, fmt='%i', delimiter=',')
np.savetxt('test_img.csv', X_test,      fmt='%i', delimiter=',')
np.savetxt('test_labels.csv', y_test,   fmt='%i', delimiter=',')
```

Сохранив CSV-файлы, мы можем загрузить их назад в Python, воспользовавшись функцией NumPy genfromtxt:

```
X_train = np.genfromtxt('train_img.csv',
                        dtype=int, delimiter=',')
y_train = np.genfromtxt('train_labels.csv',
                        dtype=int, delimiter=',')
X_test = np.genfromtxt('test_img.csv',
                       dtype=int, delimiter=',')
y_test = np.genfromtxt('test_labels.csv',
                       dtype=int, delimiter=',')
```

Однако на загрузку данных MNIST из CSV-файлов уйдет существенно больше времени, поэтому, если это возможно, рекомендуем придерживаться исходного байтового формата.

Реализация многослойного персептрона

В этом подразделе мы займемся реализацией исходного кода многослойного персептрона (MLP) с одним входным, одним скрытым и одним выходным слоями для выполнения задачи классификации изображений на наборе данных MNIST. Исходный код оставлен максимально простым. Правда, поначалу он может показаться немного сложным, и поэтому рекомендуем вам скачать исходный код примеров для этой главы с веб-сайта издательства «Packt Publishing», где вы можете найти эту реализацию MLP, аннотированную комментариями и с подсветкой синтаксиса для лучшей удобочитаемости. Если вы не выполняете исходный код как блокнот Jupyter, то рекомендуем скопировать его и сохранить как скриптовый файл Python, например neuralnet.py, в вашем текущем рабочем каталоге и затем импортировать в свой текущий сеанс Python, выполнив следующую ниже команду:

```
from neuralnet import NeuralNetMLP
```

Исходный код будет содержать фрагменты, которых мы еще не обсуждали, такие как алгоритм обратного распространения ошибки, но если исходить из реализации ADALINE в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* и обсуждения прямого распространения сигналов в более ранних разделах этой главы, то бóльшая часть исходного кода должна выглядеть достаточно для вас знакомой. Не переживайте, если исходный код не сразу стал понятен; далее в этой главе мы продолжим рассмотрение отдельных частей. Однако поверхностный осмотр исходного кода на данном этапе может упростить понимание излагаемых позже теоретических положений[1].

```
import numpy as np
from scipy.special import expit
import sys

class NeuralNetMLP(object):
    """ Нейронная сеть с прямым распространением сигналов /
    Классификатор на основе многослойного персептрона.

    Параметры
    ---------
    n_output : int
      Число выходных узлов.
      Должно равняться числу уникальных меток классов.

    n_features : int
      Число признаков (размерностей) в целевом наборе данных.
      Должно равняться числу столбцов в массиве X.

    n_hidden : int (по умолчанию: 30)
      Число скрытых узлов.
```

[1] В приведенном ниже листинге приведены все комментарии из прилагаемого к книге примера исходного кода, о котором говорит автор. – *Прим. перев.*

```
l1 : float (по умолчанию: 0.0)
  Значение лямбды для L1-регуляризации.
  Регуляризация отсутствует, если l1=0.0 (по умолчанию)

l2 : float (по умолчанию: 0.0)
  Значения лямбды для L2-регуляризации.
  Регуляризация отсутствует, если l2=0.0 (по умолчанию)

epochs : int (по умолчанию: 500)
  Число проходов по тренировочному набору.

eta : float (по умолчанию: 0.001)
  Темп обучения.

alpha : float (по умолчанию: 0.0)
  Константа импульса. Фактор, помноженный на градиент
  предыдущей эпохи t-1, с целью улучшить скорость обучения
  w(t) := w(t) - (grad(t) + alpha*grad(t-1))

decrease_const : float (по умолчанию: 0.0)
  Константа уменьшения. Сокращает (стягивает) темп обучения
  после каждой эпохи посредством eta / (1 + эпоха*константа_уменьшения)

shuffle : bool (по умолчанию: True)
  Перемешивает тренировочные данные в каждой эпохе
  для предотвращения зацикливания, если True.

minibatches : int (по умолчанию: 1)
  Разбивает тренировочные данные на k мини-пакетов для эффективности.
  Обучение с нормальным градиентным спуском, если k=1 (по умолчанию).

random_state : int (по умолчанию: None)
  Инициализирует генератор случайных чисел
  для перемешивания и инициализации весов.


Атрибуты
--------
cost_ : list/список
  Сумма квадратичных ошибок после каждой эпохи.

"""
def __init__(self, n_output, n_features, n_hidden=30,
             l1=0.0, l2=0.0, epochs=500, eta=0.001,
             alpha=0.0, decrease_const=0.0, shuffle=True,
             minibatches=1, random_state=None):
    np.random.seed(random_state)
    self.n_output = n_output
    self.n_features = n_features
    self.n_hidden = n_hidden
    self.w1, self.w2 = self._initialize_weights()
    self.l1 = l1
    self.l2 = l2
    self.epochs = epochs
    self.eta = eta
    self.alpha = alpha
    self.decrease_const = decrease_const
    self.shuffle = shuffle
    self.minibatches = minibatches
```

```
    def _encode_labels(self, y, k):
        """Преобразовать метки в прямую кодировку one-hot

        Параметры
        ----------
        y : массив, форма = [n_samples]
            Целевые значения.

        Возвращает
        ----------
        onehot : массив, форма = (n_labels, n_samples)

        """
        onehot = np.zeros((k, y.shape[0]))
        for idx, val in enumerate(y):
            onehot[val, idx] = 1.0
        return onehot

    def _initialize_weights(self):
        """Инициализировать веса малыми случайными числами."""
        w1 = np.random.uniform(-1.0, 1.0,
                               size=self.n_hidden*(self.n_features + 1))
        w1 = w1.reshape(self.n_hidden, self.n_features + 1)
        w2 = np.random.uniform(-1.0, 1.0,
                               size=self.n_output*(self.n_hidden + 1))
        w2 = w2.reshape(self.n_output, self.n_hidden + 1)
        return w1, w2

    def _sigmoid(self, z):
        """Вычислить логистическую функцию (сигмоиду)

        Использует функцию scipy.special.expit, чтобы избежать ошибки
        переполнения для очень малых входных значений z.

        """
        # expit эквивалентна 1.0 / (1.0 + np.exp(-z))
        return expit(z)

    def _sigmoid_gradient(self, z):
        """Вычислить градиент логистической функции"""
        sg = self._sigmoid(z)
        return sg * (1 - sg)

    def _add_bias_unit(self, X, how='column'):
        """Добавить в массив узел смещения
        (столбец или строку из единиц) в нулевом индексе"""
        if how == 'column':
            X_new = np.ones((X.shape[0], X.shape[1]+1))
            X_new[:, 1:] = X
        elif how == 'row':
            X_new = np.ones((X.shape[0]+1, X.shape[1]))
            X_new[1:, :] = X
        else:
            raise AttributeError('`how` должен быть `column` либо `row`')
        return X_new

    def _feedforward(self, X, w1, w2):
        """Вычислить шаг прямого распространения сигнала

        Параметры
        ---------
```

```
        X : массив, форма = [n_samples, n_features]
          Входной слой с исходными признаками.

        w1 : массив, форма = [n_hidden_units, n_features]
          Матрица весовых коэффициентов для "входной слой -> скрытый слой".

        w2 : массив, форма = [n_output_units, n_hidden_units]
          Матрица весовых коэффициентов для "скрытый слой -> выходной слой".

        Возвращает
        ----------
        a1 : массив, форма = [n_samples, n_features+1]
          Входные значения с узлом смещения.

        z2 : массив, форма = [n_hidden, n_samples]
          Чистый вход скрытого слоя.

        a2 : массив, форма = [n_hidden+1, n_samples]
          Активация скрытого слоя.

        z3 : массив, форма = [n_output_units, n_samples]
          Чистый вход выходного слоя.

        a3 : массив, форма = [n_output_units, n_samples]
          Активация выходного слоя.

        """
        a1 = self._add_bias_unit(X, how='column')
        z2 = w1.dot(a1.T)
        a2 = self._sigmoid(z2)
        a2 = self._add_bias_unit(a2, how='row')
        z3 = w2.dot(a2)
        a3 = self._sigmoid(z3)
        return a1, z2, a2, z3, a3

    def _L2_reg(self, lambda_, w1, w2):
        """Вычислить L2-регуляризованную стоимость"""
        return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2)\
                + np.sum(w2[:, 1:] ** 2))

    def _L1_reg(self, lambda_, w1, w2):
        """Вычислить L1-регуляризованную стоимость"""
        return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum()\
                + np.abs(w2[:, 1:]).sum())

    def _get_cost(self, y_enc, output, w1, w2):
        """Вычислить функцию  стоимости.

        y_enc : массив, форма = (n_labels, n_samples)
          Прямокодированные метки классов.

        output : массив, форма = [n_output_units, n_samples]
          Активация выходного слоя (прямое распространение)

        w1 : массив, форма = [n_hidden_units, n_features]
          Матрица весовых коэффициентов для "входной слой -> скрытый слой".

        w2 : массив, форма = [n_output_units, n_hidden_units]
          Матрица весовых коэффициентов для "скрытый слой -> выходной слой".

        Возвращает
        ----------
```

```
        cost : float
          Регуляризованная стоимость.

        """
        term1 = -y_enc * (np.log(output))
        term2 = (1 - y_enc) * np.log(1 - output)
        cost = np.sum(term1 - term2)
        L1_term = self._L1_reg(self.l1, w1, w2)
        L2_term = self._L2_reg(self.l2, w1, w2)
        cost = cost + L1_term + L2_term
        return cost

    def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
        """ Вычислить шаг градиента, используя обратное распространение.

        Параметры
        ---------
        a1 : массив, форма = [n_samples, n_features+1]
          Входные значения с узлом смещения.

        a2 : массив, форма = [n_hidden+1, n_samples]
          Активация скрытого слоя.

        a3 : массив, форма = [n_output_units, n_samples]
          Активация выходного слоя.

        z2 : массив, форма = [n_hidden, n_samples]
          Чистый вход скрытого слоя.

        y_enc : массив, форма = (n_labels, n_samples)
          Прямокодированные метки классов.

        w1 : массив, форма = [n_hidden_units, n_features]
          Матрица весовых коэффициентов для "входной слой -> скрытый слой".

        w2 : массив, форма = [n_output_units, n_hidden_units]
          Матрица весовых коэффициентов для "скрытый слой -> выходной слой".

        Возвращает
        ----------

        grad1 : массив, форма = [n_hidden_units, n_features]
          Градиент матрицы весовых коэффициентов w1.

        grad2 : массив, форма = [n_output_units, n_hidden_units]
          Градиент матрицы весовых коэффициентов w2.

        """
        # обратное распространение
        sigma3 = a3 - y_enc
        z2 = self._add_bias_unit(z2, how='row')
        sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
        sigma2 = sigma2[1:, :]
        grad1 = sigma2.dot(a1)
        grad2 = sigma3.dot(a2.T)

        # регуляризовать
        grad1[:, 1:] += self.l2 * w1[:, 1:]
        grad1[:, 1:] += self.l1 * np.sign(w1[:, 1:])
        grad2[:, 1:] += self.l2 * w2[:, 1:]
```

```
        grad2[:, 1:] += self.l1 * np.sign(w2[:, 1:])

        return grad1, grad2

    def predict(self, X):
        """Идентифицировать (предсказать) метки классов

        Параметры
        ---------
        X : массив, форма = [n_samples, n_features]
          Входной слой с исходными признаками.

        Возвращает
        ----------
        y_pred : массив, форма = [n_samples]
          Идентифицированные метки классов.

        """
        if len(X.shape) != 2:
            raise AttributeError(
                 'X должен быть массивом [n_samples, n_features].\n'
                 'Используйте X[:,None] для 1-признаковой классификации,'
                 '\nлибо X[[i]] для 1-точечной классификации')

        a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
        y_pred = np.argmax(z3, axis=0)
        return y_pred

    def fit(self, X, y, print_progress=False):
        """ Извлечь веса из тренировочных данных.

        Параметры
        ---------
        X : массив, форма = [n_samples, n_features]
          Входной слой с исходными признаками.

        y : массив, форма = [n_samples]
          Целевые метки классов.

        print_progress : bool (default: False)
          Распечатывает ход работы в виде числа эпох
          на устройстве stderr.

        Возвращает
        ----------
        self

        """
        self.cost_ = []
        X_data, y_data = X.copy(), y.copy()
        y_enc = self._encode_labels(y, self.n_output)

        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)

        for i in range(self.epochs):

            # адаптивный темп обучения
            self.eta /= (1 + self.decrease_const*i)

            if print_progress:
```

```
            sys.stderr.write(
                    '\rЭпоха: %d/%d' % (i+1, self.epochs))
            sys.stderr.flush()

        if self.shuffle:
            idx = np.random.permutation(y_data.shape[0])
            X_data, y_enc = X_data[idx], y_enc[:,idx]

        mini = np.array_split(range(
                    y_data.shape[0]), self.minibatches)
        for idx in mini:

            # прямое распространение
            a1, z2, a2, z3, a3 = self._feedforward(
                                    X_data[idx], self.w1, self.w2)
            cost = self._get_cost(y_enc=y_enc[:, idx],
                                  output=a3,
                                  w1=self.w1,
                                  w2=self.w2)
            self.cost_.append(cost)

            # вычислить градиент методом обратного распространения
            grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                              a3=a3, z2=z2,
                                              y_enc=y_enc[:, idx],
                                              w1=self.w1,
                                              w2=self.w2)
            # обновить веса
            delta_w1, delta_w2 = self.eta * grad1, self.eta * grad2
            self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
            self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
            delta_w1_prev, delta_w2_prev = delta_w1, delta_w2

    return self
```

Теперь инициализируем новый MLP с конфигурацией 784-50-10, т. е. нейронную сеть с 784 входными узлами (n_features), 50 скрытыми узлами (n_hidden) и 10 выходными узлами (n_output):

```
nn = NeuralNetMLP(n_output=10,
                  n_features=X_train.shape[1],
                  n_hidden=50,
                  l2=0.1,
                  l1=0.0,
                  epochs=1000,
                  eta=0.001,
                  alpha=0.001,
                  decrease_const=0.00001,
                  shuffle=True,
                  minibatches=50,
                  random_state=1)
```

Просматривая приведенную выше реализацию MLP, вы, возможно, отметили, что мы также реализовали некоторый дополнительный функционал, который резюмирован ниже:

- l2: лямбда-параметр λ для L2-регуляризации, уменьшающий степень переобучения, а также l1 – лямбда-параметр λ для L1-регуляризации;
- epochs: число проходов по тренировочному набору;
- eta: темп обучения η;
- alpha: параметр импульса (или инерции) обучения, добавляющий фактор предыдущего градиента в обновление веса в целях ускорения обучения $\Delta \boldsymbol{w}_t = \eta \nabla J(\boldsymbol{w}_t) + \alpha \Delta \boldsymbol{w}_{t-1}$ (где t – это текущий квант времени или эпоха)[1];
- decrease_const: константа уменьшения d для адаптивного темпа обучения n, который со временем уменьшается для более хорошей сходимости $\eta/1 + t \times d$;
- shuffle: перемешивание тренировочного набора перед началом каждой эпохи для предотвращения зацикливания алгоритма;
- Minibatches: разделение тренировочных данных на k мини-пакетов в каждой эпохе. В целях ускорения обучения градиент вычисляется отдельно для каждого мини-пакета вместо его вычисления для всех тренировочных данных.

Далее мы тренируем MLP, используя 60 000 образцов из уже перемешанного тренировочного набора данных MNIST. Перед выполнением следующего ниже исходного кода отметим, что тренировка нейронной сети на стандартных настольных компьютерах может занять 10–30 минут:

```
>>>
nn.fit(X_train, y_train, print_progress=True)
Эпоха: 1000/1000
```

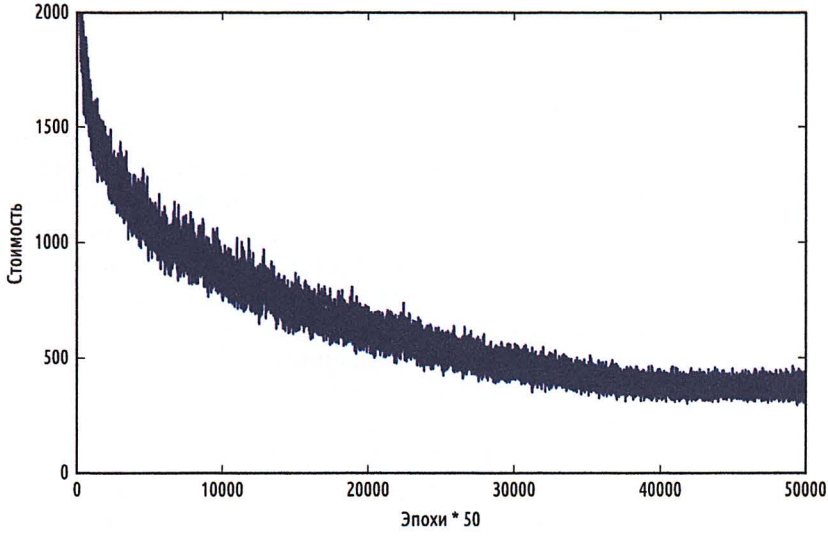
Аналогично нашей предыдущей реализации ADALINE мы сохраняем стоимость для каждой эпохи в списке cost_, который теперь можно визуализировать и получить подтверждение, что алгоритм оптимизации достиг сходимости. Здесь мы показываем только каждый 50-й шаг, чтобы учесть 50 мини-пакетов (50 мини-пакетов × 1000 эпох). Исходный код следующий:

```
plt.plot(range(len(nn.cost_)), nn.cost_)
plt.ylim([0, 2000])
plt.ylabel('Стоимость')
plt.xlabel('Эпохи * 50')
plt.tight_layout()
plt.show()
```

Как видно на нижеследующем рисунке, график функции стоимости выглядит очень зашумленным. Это вызвано тем, что мы тренировали нейронную сеть, используя мини-пакетное обучение, вариант стохастического градиентного спуска.

---

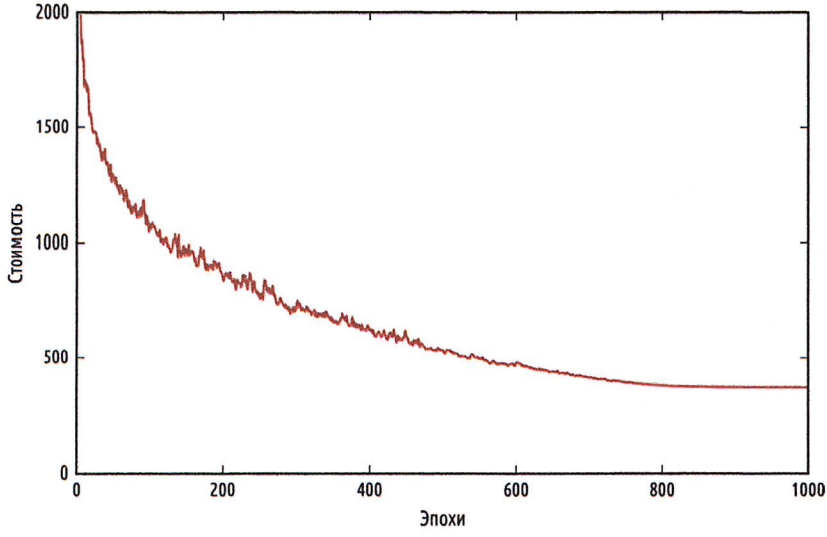
[1] Импульс обучения (momentum learning) – это слагаемое, которое способствует продвижению в фиксированном направлении, поэтому если было сделано несколько шагов в одном и том же направлении, то алгоритм «увеличивает скорость», что (иногда) позволяет избежать локального минимума, а также быстрее проходить плоские участки. – *Прим. перев.*



Хотя на графике уже можно увидеть, что алгоритм оптимизации сходился приблизительно после 800 эпох (40 000/50 = 800), мы построим график более гладкой версии функции стоимости в сопоставлении с числом эпох путем усреднения по мини-пакетным интервалам. Исходный код следующий:

```
batches = np.array_split(range(len(nn.cost_)), 1000)
cost_ary = np.array(nn.cost_)
cost_avgs = [np.mean(cost_ary[i]) for i in batches]
plt.plot(range(len(cost_avgs)),
         cost_avgs,
         color='red')
plt.ylim([0, 2000])
plt.ylabel('Стоимость')
plt.xlabel('Эпохи')
plt.tight_layout()
plt.show()
```

Следующий ниже график дает нам более ясную картину, указывающую на то, что алгоритм обучения сходился вскоре после 800-й эпохи:



Теперь выполним оценку качества работы модели, вычислив верность предсказания на тренировочном наборе:

```
>>>
y_train_pred = nn.predict(X_train)
acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
print('Верность на тренировочном наборе: %.2f%%' % (acc * 100))
Верность на тренировочном наборе: 97.59%
```

Как видно, модель правильно классифицирует большинство тренировочных цифр. Осталось ответить на вопрос: как хорошо она обобщается на данные, которые она не видела прежде? Вычислим верность на 10 000 изображениях в тестовом наборе данных:

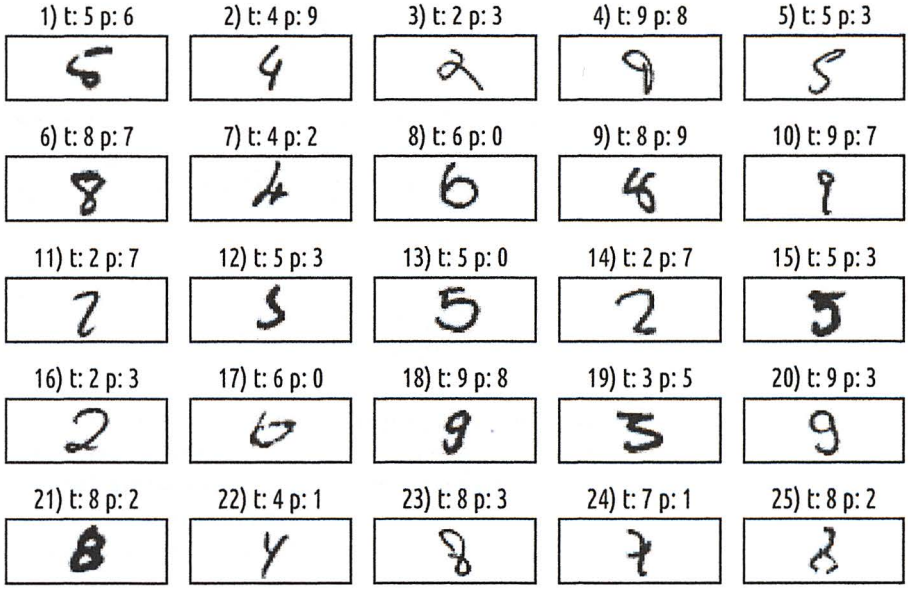
```
>>>
y_test_pred = nn.predict(X_test)
acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
print('Верность на тестовом наборе: %.2f%%' % (acc * 100))
Верность на тестовом наборе: 95.62%
```

Основываясь на небольшом разрыве между верностью на тренировочных и тестовых данных, мы можем прийти к заключению, что модель лишь слегка переподогнана под тренировочные данные. Чтобы продолжить тонкую настройку модели, можно было бы изменить число скрытых узлов, значения параметров регуляризации, темпа обучения, значений константы уменьшения или адаптивного темпа обучения, используя методы, которые мы обсудили в *главе 6 «Анализ наиболее успешных приемов оценивания моделей и тонкой настройки гиперпараметров»* (оставляем это в качестве упражнения для читателя).

Теперь посмотрим на несколько изображений, с которыми борется наш MLP:

```
miscl_img = X_test[y_test != y_test_pred][:25]
correct_lab = y_test[y_test != y_test_pred][:25]
miscl_lab= y_test_pred[y_test != y_test_pred][:25]
fig, ax = plt.subplots(nrows=5,
                       ncols=5,
                       sharex=True,
                       sharey=True,)
ax = ax.flatten()
for i in range(25):
    img = miscl_img[i].reshape(28, 28)
    ax[i].imshow(img,
                 cmap='Greys',
                 interpolation='nearest')
    ax[i].set_title('%d) t: %d p: %d'
                    % (i+1, correct_lab[i], miscl_lab[i]))
ax[0].set_xticks([])
ax[0].set_yticks([])
plt.tight_layout()
plt.show()
```

Сейчас мы должны увидеть составной график в виде матрицы 5×5, где первое число в подзаголовках обозначает индекс графика, второе – истинную метку класса (t) и третье – идентифицированную метку класса (p).



Как видно на приведенном выше рисунке, некоторые из этих изображений сложно классифицировать правильно даже людям. Например, мы видим, что цифра 9 классифицируется как 3 или 8, в случае если нижняя часть цифры имеет крюкообразное искривление (подграфики 4, 18 и 20).

Тренировка искусственной нейронной сети

Увидев нейронную сеть в действии и получив на основе беглого просмотра исходного кода общее представление о том, как она работает, теперь углубимся в некоторые понятия, такие как логистическая функция стоимости и алгоритм обратного распространения ошибки, который мы реализовали для извлечения весов.

Вычисление логистической функции стоимости

Логистическая функция стоимости, которую мы реализовали как метод `_get_cost`, фактически довольно простая, поскольку это та же самая функция стоимости, которую мы описали в разделе, посвященном логистической регрессии, в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*.

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}).$$

Здесь $a^{(i)}$ – это сигмоидальная активация i-го узла в одном из слоев, который мы вычисляем на шаге прямого распространения сигналов:

$$a^{(i)} = \phi(z^{(i)}).$$

Теперь добавим члены **регуляризации**, которые позволят уменьшить степень переподгонки. Как вы помните из более ранних глав, члены L2- и L1-регуляризации задаются следующим образом (напомним, что узлы смещения мы не упорядочиваем):

$$L2: \lambda \|\boldsymbol{w}\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \quad \text{и} \quad L1: \lambda \|\boldsymbol{w}\|_1^1 = \lambda \sum_{j=1}^{m} |w_j|.$$

Хотя наша реализация многослойного персептрона (MLP) поддерживает обе разновидности регуляризации – L1- и L2-регуляризацию, теперь для простоты мы сосредоточимся только на члене L2-регуляризации. Однако те же принципы работы применимы к члену L1-регуляризации. Добавив в нашу логистическую функцию стоимости член L2-регуляризации, мы получаем следующее уравнение:

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n} y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}) \right] + \frac{\lambda}{2} \|\boldsymbol{w}\|_2^2.$$

Учитывая, что мы реализовали MLP для многоклассовой классификации, он возвращает выходной вектор из t элементов, которые мы должны сравнить с $t \times 1$-мерным целевым вектором, представленным в виде прямого кода. Например, активация третьего слоя и целевой класс (в данном случае класс 2) для отдельно взятого образца может выглядеть следующим образом:

$$a^{(3)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \; y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}.$$

Следовательно, мы должны обобщить логистическую функцию стоимости на все узлы активации j в нашей сети. Поэтому наша функция стоимости (без члена регуляризации) становится:

$$J(\boldsymbol{w}) = -\sum_{i=1}^{n}\sum_{j=1}^{t} y_j^{(i)} \log(a_j^{(i)}) + (1 - y_j^{(i)}) \log(1 - a_j^{(i)}).$$

Здесь надстрочный индекс i – это индекс отдельно взятого образца в нашем тренировочном наборе.

Следующий ниже обобщенный член регуляризации поначалу выглядит довольно сложным, но здесь мы просто вычисляем сумму всех весов слоя l (без члена смещения), которую мы добавили к первому столбцу:

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{m} y_j^{(i)} \log(\phi(z_j^{(i)})) + (1 - y_j^{(i)}) \log(1 - \phi(z_j^{(i)}))\right] +$$
$$+ \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_l+1} (w_{j,i}^{(i)})^2.$$

Следующее ниже выражение представляет собой слагаемое (член) со штрафом за счет L2-регуляризации:

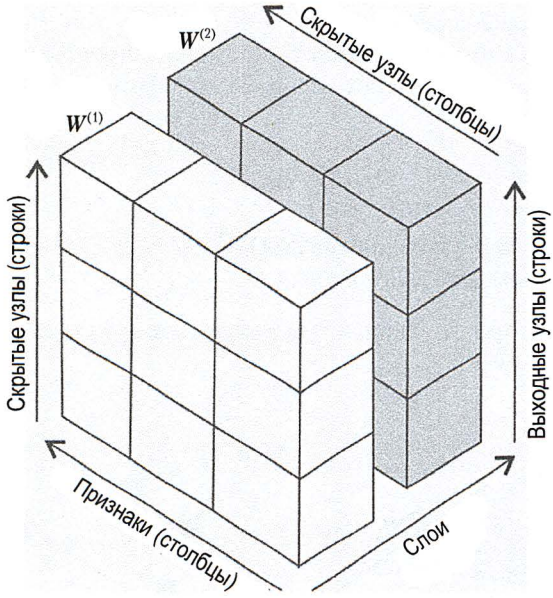
$$\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_l+1} (w_{j,i}^{(i)})^2.$$

Напомним, что наша задача – минимизировать функцию стоимости $J(\boldsymbol{w})$. Вследствие этого мы должны вычислить частную производную матрицы $\boldsymbol{W}$ относительно каждого веса для каждого слоя в сети:

$$\frac{\delta}{\delta w_{j,i}^{(l)}} J(\boldsymbol{W}).$$

В следующем разделе мы поговорим об алгоритме обратного распространения ошибки, который позволяет вычислять эти частные производные с целью минимизации функции стоимости.

Отметим, что $\boldsymbol{W}$ состоит из нескольких матриц. В многослойном персептроне с одним скрытым слоем у нас есть весовая матрица $\boldsymbol{W}^{(1)}$, которая соединяет вход со скрытым слоем, и весовая матрица $\boldsymbol{W}^{(2)}$, которая соединяет скрытый слой с выходным слоем. Интуитивная визуализация матрицы $\boldsymbol{W}$ приведена на нижеследующем рисунке:



На этом упрощенном рисунке может показаться, что $\boldsymbol{W}^{(1)}$ и $\boldsymbol{W}^{(2)}$ имеют одно и то же число строк и столбцов, что, как правило, это не так, в случае если мы не инициализируем MLP одинаковым числом скрытых узлов, выходных узлов и входных признаков.

Если в связи с этим возникают вопросы, то не переключайтесь, т. к. в следующем разделе мы обсудим размерность $\boldsymbol{W}^{(1)}$ и $\boldsymbol{W}^{(2)}$ более подробно в контексте алгоритма обратного распространения ошибки.

Тренировка нейронных сетей методом обратного распространения ошибки

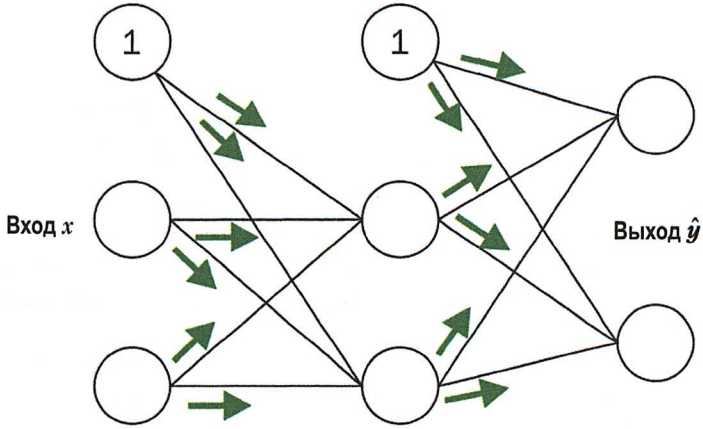
В этом разделе мы коснемся математики метода обратного распространения ошибки, чтобы разобраться в том, каким образом можно наиболее эффективным образом извлекать веса в нейронной сети. В зависимости от того, насколько вы удобно себя чувствуете с математическими представлениями, следующие ниже равенства поначалу могут казаться относительно сложными. Многие предпочитают восходящий подход (снизу вверх) и пройтись по уравнениям постепенно, чтобы развить интуитивное понимание алгоритмов. Однако если вы предпочитаете нисходящий подход и хотите узнать об обратном распространении ошибки без каких-либо математических обозначений, то рекомендуем сначала прочитать следующий далее раздел «*Развитие интуитивного понимания обратного распространения ошибки*» и позже вернуться к этому разделу.

В предыдущем разделе мы узнали, каким образом вычисляется стоимость, как разница между активацией последнего слоя и целевой меткой класса. Теперь мы увидим, как алгоритм обратного распространения обновляет веса в нашей модели

MLP, которую мы реализовали в методе _get_gradient. Как мы помним из начала этой главы, сначала нам нужно применить прямое распространение сигналов, с тем чтобы получить активацию выходного слоя. Ее мы сформулировали следующим образом:

$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)}[\boldsymbol{A}^{(1)}]^T$ (чистый вход скрытого слоя);
$\boldsymbol{A}^{(2)} = \phi(\boldsymbol{Z}^{(2)})$ (активация скрытого слоя);
$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{A}^{(2)}$ (чистый вход выходного слоя);
$\boldsymbol{A}^{(3)} = \phi(\boldsymbol{Z}^{(3)})$ (активация выходного слоя).

Если кратко, то мы просто распространяем входные признаки вперед по связям в сети, как показано ниже:



При обратном распространении мы распространяем ошибку справа налево. Мы начинаем с вычисления вектора ошибки выходного слоя:

$\Delta^{(3)} = a^{(3)} - y.$

Здесь y – это вектор истинных меток классов.

Далее мы вычисляем член ошибки скрытого слоя:

$$\delta^{(2)} = (\boldsymbol{W}^{(2)})^T \delta^{(3)} * \frac{\delta\phi(z^{(2)})}{\delta z^{(2)}}.$$

Здесь $\frac{\delta\phi(z^{(2)})}{\delta z^{(2)}}$ – это просто производная сигмоидальной функции активации, которую мы реализовали как _sigmoid_gradient:

$$\frac{\delta\phi(z^{(2)})}{\delta z^{(2)}} = (a^{(2)} * (1 - a^{(2)})).$$

Отметим, что символ звездочки (*) в этом контексте обозначает поэлементное умножение.

Несмотря на то что понимание следующих ниже уравнений не требуется, вам может быть любопытно знать, каким образом была получена производная функции активации. Операция взятия производной пошагово резюмирована ниже:

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

Чтобы лучше разобраться в том, как происходит вычисление частной производной $\Delta^{(3)}$, пройдемся по нему более подробно. В предыдущем уравнении мы умножили транспонирование $(\boldsymbol{W}^{(2)})^T$ $t \times h$-мерной матрицы $\boldsymbol{W}^{(2)}$; t – это число выходных меток классов и h – число скрытых узлов. Теперь $(\boldsymbol{W}^{(2)})^T$ становится $h \times t$-мерной матрицей с $\Delta^{(3)}$, который является $t \times 1$-мерным вектором. Затем мы выполнили попарное умножение $(\boldsymbol{W}^{(2)})^T\Delta^{(3)}$ на $(a^{(2)} * (1 - a^{(2)}))$, который тоже является $t \times 1$-мерным вектором. В заключение после получения членов Δ теперь можем записать взятие функции стоимости следующим образом:

$$\frac{\delta}{\delta w_{i,j}^{(l)}} J(\boldsymbol{W}) = a_j^{(l)} \delta_i^{(l+1)}.$$

Далее нам нужно накопить частную производную каждого j-го узла в слое l и i-ю ошибку узла в слое $l + 1$:

$$\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}.$$

Напомним, что для каждого образца в тренировочном наборе нам нужно вычислить обновление $\Delta_{i,j}^{(l)}$. Следовательно, будет проще, если реализовать его в векторизованной версии, как в нашей предыдущей программной реализации MLP:

$$\Delta^{(l)} = \Delta^{(l)} + \Delta^{(l+1)}(\boldsymbol{A}^{(l)})^T.$$

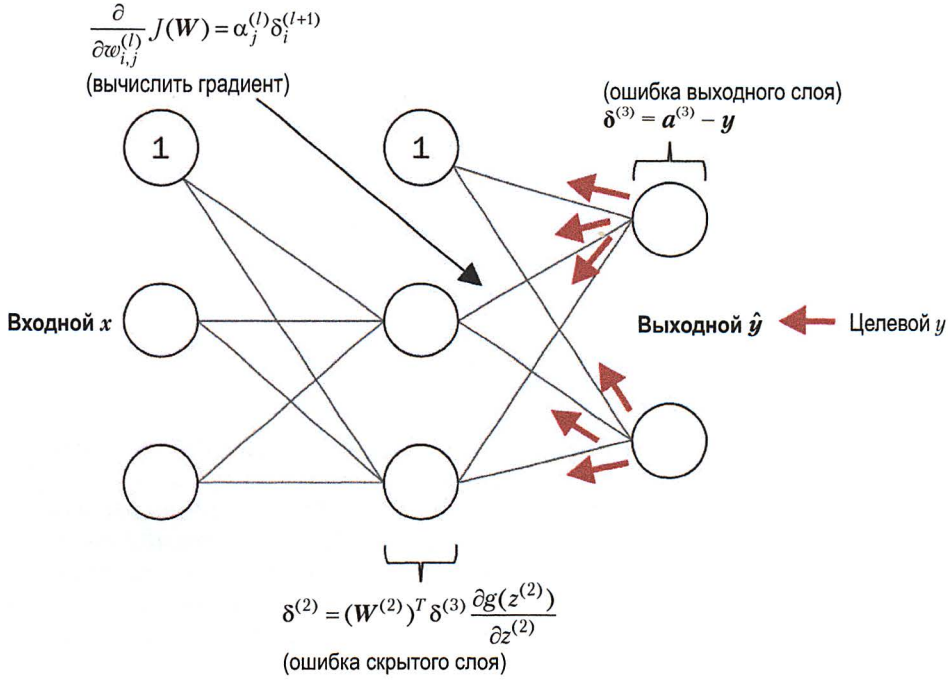
Накопив частные производные, мы можем добавить член регуляризации следующим образом:

$\Delta^{(l)} = \Delta^{(l)} + \Delta^{(l)}$ (за исключением члена смещения).

Наконец, после того как мы вычислили градиенты, мы теперь можем обновить веса, сделав шаг в противоположную от градиента сторону (в направлении антиградиента):

$$\boldsymbol{W}^{(l)} := \boldsymbol{W}^{(l)} - \eta\Delta^{(l)}.$$

Чтобы свести все воедино, резюмируем метод обратного распространения ошибки на нижеследующем рисунке:



Развитие интуитивного понимания алгоритма обратного распространения ошибки

Несмотря на то что алгоритм обратного распространения ошибки был повторно открыт и популяризирован почти 30 лет назад, он все еще остается одним из наиболее широко используемых алгоритмов наиболее эффективной тренировки искусственной нейронной сети. В этом разделе мы увидим более интуитивное резюме и более широкую картину того, как работает этот замечательный алгоритм.

В сущности, обратное распространение ошибки – это просто очень эффективный в вычислительном плане подход к вычислению производных сложной функции стоимости. Наша задача – использовать эти производные для извлечения весовых коэффициентов с целью параметризации многослойной искусственной нейронной сети. Сложность параметризации нейронных сетей состоит в том, что мы, как правило, имеем дело с очень большим количеством весовых коэффициентов в высокоразмерном пространстве признаков. В отличие от других функций стоимости, которые мы увидели в предыдущих главах, функция стоимости поверхности ошибки нейронной сети не является выпуклой или гладкой. На этой высокоразмерной поверхности стоимости имеется много ям (локальных минимумов), которые мы должны преодолеть для нахождения глобального минимума функции стоимости.

Вы можете вспомнить понятие цепного правила из ваших уроков по основам дифференциального исчисления. Цепное правило – это подход к получению сложной, вложенной функции, например $f(g(x)) = y$, которая разлагается на составляющие компоненты:

$$\frac{\delta y}{\delta x} = \frac{\delta f}{\delta g}\frac{\delta g}{\delta x}.$$

В контексте вычислительной алгебры был разработан ряд методов для очень эффективного решения таких задач под общим названием методов *автоматического дифференцирования*. Если вам интересно больше узнать об автоматическом дифференцировании в приложениях с использованием машинного обучения, то рекомендуем обратиться к следующему ресурсу: *Baydin A. G., Pearlmutter B. A.* Automatic Differentiation of Algorithms for Machine Learning. arXiv preprint arXiv:1404.7456, 2014 (Бейдин А. Г., Перлматтер А. «*Автоматическое дифференцирование алгоритмов для машинного обучения*»), который имеется в свободном доступе по прямой ссылке на arXiv по http://arxiv.org/pdf/1404.7456.pdf.

Автоматическое дифференцирование имеет два режима, соответственно *прямой* и *обратный* режимы. Обратное распространение ошибки – это просто частный случай обратного режима автоматического дифференцирования. Ключевой момент заключается в том, что применение цепного правила в прямом режиме может быть довольно затратным, поскольку нам пришлось бы умножать большие матрицы для каждого слоя (якобианы), которые мы в конечном счете умножаем на вектор для получения выхода. Трюк обратного режима состоит в том, что мы начинаем справа налево: мы умножаем матрицу на вектор и в результате получаем другой вектор, который умножается на следующую матрицу, и т. д. Матрично-векторное умножение является в вычислительном плане намного более дешевым, чем матрично-матричное умножение, и именно по этой причине алгоритм обратного распространения ошибки является одним из самых популярных алгоритмов, используемых для тренировки нейронных сетей.

Отладка нейронных сетей процедурой проверки градиента

Реализации искусственных нейронных сетей могут быть довольно сложными, и всегда стоит *вручную* проверить, что мы правильно реализовали алгоритм обратного распространения ошибки. В этом разделе мы поговорим о простой процедуре, именуемой *проверкой градиента* (gradient checking), которая, по существу, представляет собой сравнение наших аналитических градиентов в сети с числовыми градиентами. Проверка градиента не ограничивается исключительно нейронными сетями с прямым распространением сигналов и может применяться к любой другой нейросетевой архитектуре, в которой используется градиентная оптимизация. Даже если вы планируете реализовать такие более тривиальные алгоритмы с использованием градиентной оптимизации, как линейная регрессия, логистическая регрессия и метод опорных векторов, то обычно неплохо проверить, вычислены ли градиенты правильно.

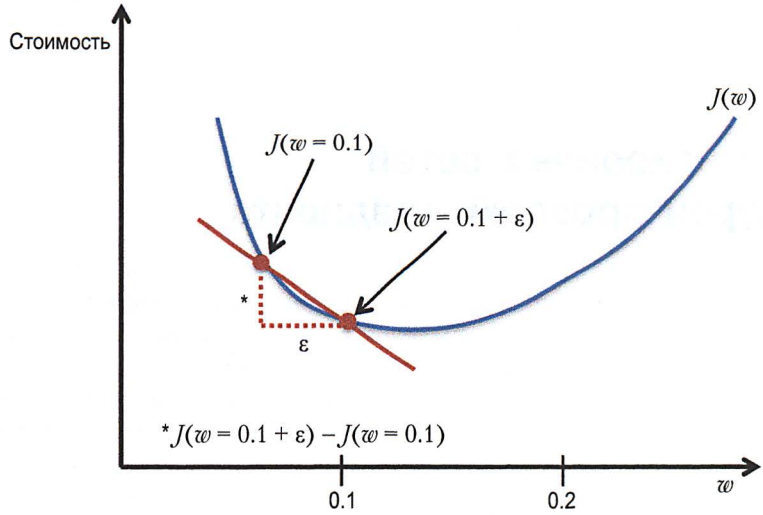
В предыдущих разделах мы определили функцию стоимости $J(\boldsymbol{W})$, где $\boldsymbol{W}$ – это матрица весовых коэффициентов искусственной сети. Отметим, что $J(\boldsymbol{W})$ – грубо говоря – «многоярусная» (каскадная, stacked) матрица, состоящая из матриц $\boldsymbol{W}^{(1)}$ и $\boldsymbol{W}^{(2)}$ в многослойном персептроне с одним скрытым узлом. Мы определили $\boldsymbol{W}^{(1)}$ как $h \times [m + 1]$-мерную матрицу, которая соединяет входной слой со скрытым слоем, где h – это число скрытых узлов и m – число признаков (входных узлов). Матрица $\boldsymbol{W}^{(2)}$, которая соединяет скрытый слой с выходным слоем, имеет размерность $t \times h$, где t – число выходных узлов. Затем мы вычислили производную функции стоимости для веса $w_{i,j}^{(l)}$ следующим образом:

$$\frac{\delta}{\delta w_{i,j}^{(l)}} J(\boldsymbol{W}).$$

Напомним, что мы обновляем веса, делая шаг в противоположную от градиента сторону. При проверке градиента мы сравниваем это аналитическое решение с численно аппроксимированным градиентом:

$$\frac{\delta}{\delta w_{i,j}^{(l)}} J(\boldsymbol{W}) \approx \frac{J(w_{i,j}^{(l)} + \varepsilon) - J(w_{i,j}^{(l)})}{\varepsilon}.$$

Здесь ε – это, как правило, очень небольшое число, например 1e–5 (отметим, что 1e–5 – это просто более удобная запись числа 0.00001). Эта конечная разностная аппроксимация может быть интуитивно представлена как наклон секущей, соединяющей точки функции стоимости для двух весов w и $w + \varepsilon$ (оба скалярные величины), как показано на нижеследующем рисунке. Для простоты надстрочные и подстрочные индексы мы опускаем.



Еще лучше подход с более точной аппроксимацией градиента состоит в вычислении симметричного (или центрированного) отношения приращений, заданного двуточечной формулой:

$$\frac{J(w_{i,j}^{(l)}+\varepsilon)-J(w_{i,j}^{(l)}-\varepsilon)}{2\varepsilon}.$$

Как правило, аппроксимированная разница между числовым градиентом J'_n и аналитическим градиентом J'_a затем вычисляется как норма L2-вектора. В силу практических соображений мы разворачиваем вычисленные градиентные матрицы в плоские векторы, в результате чего можем более удобно вычислить ошибку (разницу между градиентными векторами):

$$\text{Ошибка} = \|J'_n - J'_a\|_2.$$

Одна из проблем состоит в том, что ошибка не инвариантна по шкале (небольшие ошибки более значительны, если нормы весовых векторов тоже небольшие). Поэтому рекомендуется вычислять нормализованную разность:

$$\text{Относительная ошибка} = \frac{\|J'_n - J'_a\|_2}{\|J'_n\|_2 + \|J'_a\|_2}.$$

Теперь нам нужно, чтобы относительная ошибка (погрешность) между числовым градиентом и аналитическим градиентом была как можно меньше. Прежде чем реализовать проверку градиента, мы должны обсудить еще одну деталь: каков должен быть приемлемый порог ошибки, чтобы проверка градиента прошла успешно? Порог относительной ошибки для прохождения проверки градиента зависит от сложности сетевой архитектуры. Как показывает опыт, чем больше добавляется скрытых слоев, тем больше может становиться разница между числовым и аналитическим градиентами, в случае если алгоритм обратного распространения реализован правильно. Поскольку в этой главе мы реализовали архитектуру относительно простой нейронной сети, будем достаточно строгими в отношении порога и определим следующие правила:

- относительная ошибка <= 1e−7 означает, что все хорошо!
- относительная ошибка <= 1e−4 означает, что условие проблематично и мы должны его изучить;
- относительная ошибка > 1e−4 означает, что в нашем исходном коде, вероятно, что-то не так.

Установив основные правила, теперь реализуем проверку градиента. Для этого мы можем просто взять класс NeuralNetMLP, который мы реализовали ранее, и добавить в тело класса следующий метод:

```
def _gradient_checking(self, X, y_enc, w1,
                       w2, epsilon, grad1, grad2):
    """ Применить проверку градиента (исключительно для отладки)

    Возвращает
    ----------
    relative_error : float

      Относительная ошибка между численно аппроксимированными
      градиентами и градиентами обратного распространения.
    """
```

```
num_grad1 = np.zeros(np.shape(w1))
epsilon_ary1 = np.zeros(np.shape(w1))
for i in range(w1.shape[0]):
    for j in range(w1.shape[1]):
        epsilon_ary1[i, j] = epsilon
        a1, z2, a2, z3, a3 = self._feedforward(
                                    X,
                                    w1 - epsilon_ary1,
                                    w2)
        cost1 = self._get_cost(y_enc,
                               a3,
                               w1-epsilon_ary1,
                               w2)
        a1, z2, a2, z3, a3 = self._feedforward(
                                      X,
                                      w1 + epsilon_ary1,
                                      w2)
        cost2 = self._get_cost(y_enc,
                               a3,
                               w1 + epsilon_ary1,
                               w2)
        num_grad1[i, j] = (cost2 - cost1) / (2 * epsilon)
        epsilon_ary1[i, j] = 0

num_grad2 = np.zeros(np.shape(w2))
epsilon_ary2 = np.zeros(np.shape(w2))
for i in range(w2.shape[0]):
    for j in range(w2.shape[1]):
        epsilon_ary2[i, j] = epsilon
        a1, z2, a2, z3, a3 = self._feedforward(
                                         X,
                                         w1,
                                         w2 - epsilon_ary2)
        cost1 = self._get_cost(y_enc,
                               a3,
                               w1,
                               w2 - epsilon_ary2)
        a1, z2, a2, z3, a3 = self._feedforward(
                                      X,
                                      w1,
                                      w2 + epsilon_ary2)
        cost2 = self._get_cost(y_enc,
                               a3,
                               w1,
                               w2 + epsilon_ary2)
        num_grad2[i, j] = (cost2 - cost1) / (2 * epsilon)
        epsilon_ary2[i, j] = 0

num_grad = np.hstack((num_grad1.flatten(),
                      num_grad2.flatten()))
grad = np.hstack((grad1.flatten(), grad2.flatten()))
norm1 = np.linalg.norm(num_grad - grad)
norm2 = np.linalg.norm(num_grad)
norm3 = np.linalg.norm(grad)
relative_error = norm1 / (norm2 + norm3)
return relative_error
```

Исходный код _gradient_checking кажется довольно простым. Однако моя личная рекомендация состоит в том, чтобы таким же его и оставить. Наша задача – перепроверить вычисление градиента, и поэтому мы хотим быть уверенными в том, что не вносим дополнительных ошибок в проверку градиента написанием эффективного, но сложного исходного кода. Далее нам нужно всего лишь сделать небольшое изменение в методе fit. В приведенном ниже фрагменте для ясности пропущен исходный код в начале функции fit, и единственные строки, которые нам нужно добавить в метод, помещены между комментариями ## начало проверки градиента и ## конец проверки градиента:

```
class MLPGradientCheck(object):
    [...]
    def fit(self, X, y, print_progress=False):
        [...]
                # вычислить градиент методом обратного распространения
                grad1, grad2 = self._get_gradient(
                                        a1=a1,
                                        a2=a2,
                                        a3=a3,
                                        z2=z2,
                                        y_enc=y_enc[:, idx],
                                        w1=self.w1,
                                        w2=self.w2)
                ## начало проверки градиента
                grad_diff = self._gradient_checking(
                                    X=X[idx],
                                    y_enc=y_enc[:, idx],
                                    w1=self.w1,
                                    w2=self.w2,
                                    epsilon=1e-5,
                                    grad1=grad1,
                                    grad2=grad2)
                if grad_diff <= 1e-7:
                    print('Успешно: %s' % grad_diff)
                elif grad_diff <= 1e-4:
                    print('Предупреждение: %s' % grad_diff)
                else:
                    print('ПРОБЛЕМА: %s' % grad_diff)

                ## конец проверки градиента

                # обновить веса; [alpha * delta_w_prev] для импульса обучения
                delta_w1 = self.eta * grad1
                delta_w2 = self.eta * grad2
                self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
                self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
                delta_w1_prev = delta_w1
                delta_w2_prev = delta_w2

        return self
```

Если предположить, что мы назвали наш класс модифицированного многослойного персептрона MLPGradientCheck, то теперь мы можем инициализировать новый MLP 10 скрытыми слоями. Мы также отключаем регуляризацию, адаптивное обучение и импульс обучения. Кроме того, мы используем регулярный градиентный спуск, установив мини-пакеты в 1. Исходный код следующий:

```
nn_check = MLPGradientCheck(n_output=10,
                            n_features=X_train.shape[1],
                            n_hidden=10,
                            l2=0.0,
                            l1=0.0,
                            epochs=10,
                            eta=0.001,
                            alpha=0.0,
                            decrease_const=0.0,
                            minibatches=1,
                            random_state=1)
```

Оборотной стороной проверки градиента является то, что он в вычислительном плане очень и очень затратный. Тренировка нейронной сети с включенной проверкой градиента настолько медленная, что на самом деле ее следует использовать только для целей отладки. По этой причине весьма распространено выполнять проверку градиента исключительно на небольшой серии тренировочных образцов (в данном случае мы взяли 5). Исходный код следующий:

```
>>>
nn_check.fit(X_train[:5], y_train[:5], print_progress=False)
Успешно: 2.56712936241e-10
Успешно: 2.94603251069e-10
Успешно: 2.37615620231e-10
Успешно: 2.43469423226e-10
Успешно: 3.37872073158e-10
Успешно: 3.63466384861e-10
Успешно: 2.22472120785e-10
Успешно: 2.33163708438e-10
Успешно: 3.44653686551e-10
Успешно: 2.17161707211e-10
```

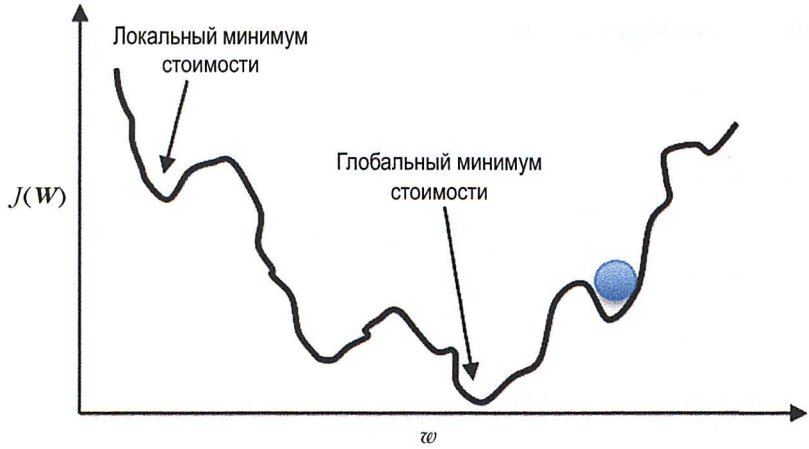
Как видно из результата выполнения исходного кода, наш многослойный персептрон проходит этот тест с превосходными результатами.

Сходимость в нейронных сетях

Вам, наверное, интересно, почему для тренировки нашей нейронной сети с целью выполнения задачи классификации рукописных цифр мы вместо регулярного градиентного спуска использовали мини-пакетное обучение. Если вы припоминаете наше обсуждение стохастического градиентного спуска, то мы его использовали для реализации динамического обучения. В динамическом обучении для обновления весов мы вычисляем градиент, основываясь каждый раз на единственном тренировочном примере ($k = 1$). Несмотря на то что этот подход стохастический, он часто приводит к очень точным решениям с гораздо более быстрой сходимостью, чем регулярный градиентный спуск. Мини-пакетное обучение – это специальная форма стохастического градиентного спуска, где мы вычисляем градиент на основе подмножества k из n тренировочных образцов при $1 < k < n$. Преимущество мини-пакетного обучения перед динамическим обучением состоит в том, что мы можем использовать наши векторизованные реализации для повышения вычислительной эффективности. С другой стороны, обновлять веса можно намного быстрее, чем в регулярном

градиентном спуске. Мини-пакетное обучение интуитивно можно представить как предсказание явки избирателей на президентских выборах, исходя из опроса только представительного подмножества граждан вместо опроса всех граждан.

Кроме того, мы добавили больше настроечных параметров, таких как константа уменьшения или параметр для адаптивного темпа обучения. Причина состоит в том, что тренировать нейронные сети намного тяжелее, чем более простые алгоритмы, такие как ADALINE, логистическая регрессия или метод опорных векторов. В многослойных нейронных сетях, как правило, имеются сотни, тысячи или даже миллиарды весовых коэффициентов, которые мы должны оптимизировать. К сожалению, функция выходов имеет грубую поверхность, и алгоритм оптимизации может легко застрять в локальных минимумах, как показано на нижеследующем рисунке:



Отметим, что это представление чрезвычайно упрощенное, поскольку наша нейронная сеть имеет много размерностей; это делает невозможной визуализацию реальной поверхности стоимости для человеческого глаза. Здесь мы показываем поверхность стоимости только для единственного веса на оси X. Однако главный посыл заключается в том, что мы не хотим, чтобы наш алгоритм застревал в локальных минимумах. Увеличивая темп обучения, мы можем с большей готовностью избежать таких локальных минимумов. С другой стороны, мы также увеличиваем шанс промаха по глобальному оптимуму, в случае если темп обучения слишком большой. Поскольку мы инициализируем веса случайными значениями, мы начинаем с решения задачи оптимизации, которая, как правило, безнадежно неверна. Заданная нами ранее константа уменьшения способна помочь быстрее спуститься по поверхности стоимости в самом начале, а адаптивный темп обучения позволяет лучше отжигать к глобальному минимуму.

Другие нейросетевые архитектуры

В этой главе мы обсудили одно из самых популярных представлений нейронной сети с прямым распространением сигналов – многослойный персептрон. Нейронные сети в настоящее время являются одной из самых активных тем научных ис-

следований в области машинного обучения, и существует много других нейросетевых архитектур, которые выходят далеко за рамки объема этой книги. Если вы интересуетесь нейронными сетями и алгоритмами для глубокого обучения и хотите узнать о них больше, то рекомендуем почитать введение и обзор данной темы: *Bengio Y.* Learning Deep Architectures for A». Foundations and Trends in Machine Learning, 2(1):1-127, 2009 («*Изучение глубоких архитектур для ИИ*»). Книга Й. Бенджо в настоящее время находится в общем доступе по прямой ссылке на http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf.

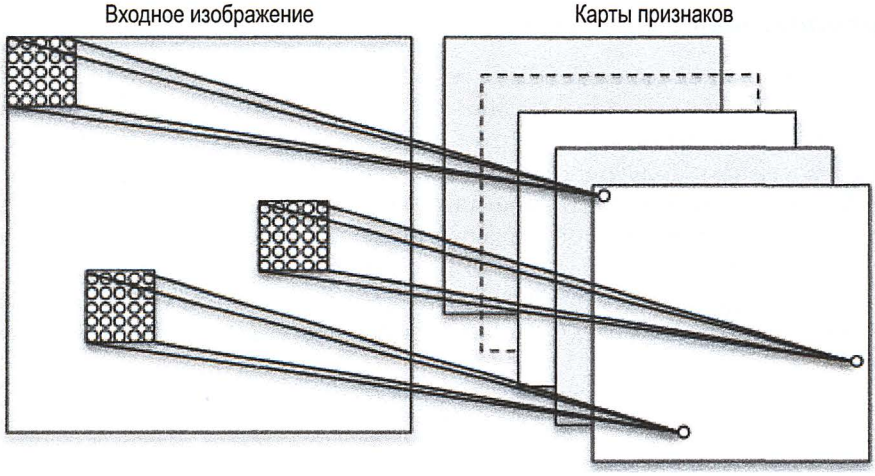
Несмотря на то что нейронные сети на самом деле являются темой для еще одной книги, проведем, по крайней мере, беглый обзор двух других популярных архитектур, **сверточных** и **рекуррентных нейронных сетей**.

Сверточные нейронные сети

Сверточные нейронные сети (convolutional neural network, CNN или ConvNet) завоевали популярность в компьютерном зрении из-за их экстраординарно хорошего качества работы на задачах классификации изображений. На сегодняшний день CNN являются одной из самых популярных архитектур нейронных сетей в глубоком обучении. Лежащая в основе сверточных нейронных сетей ключевая идея состоит в создании множества слоев **детекторов признаков** (feature detectors), чтобы учитывать пространственное расположение пикселов во входном изображении. Отметим, что есть много разных вариантов нейронных сетей CNN. В этом разделе мы обсудим только общую идею, лежащую в основе этой архитектуры. Если вам интересно узнать о сверточных нейронных сетях больше, то рекомендуем посмотреть публикации (http://yann.lecun.com) Янна Лекуна, одного из соавторов нейронных сетей CNN. В частности, для начала работы с нейронными сетями CNN можно порекомендовать следующую литературу:

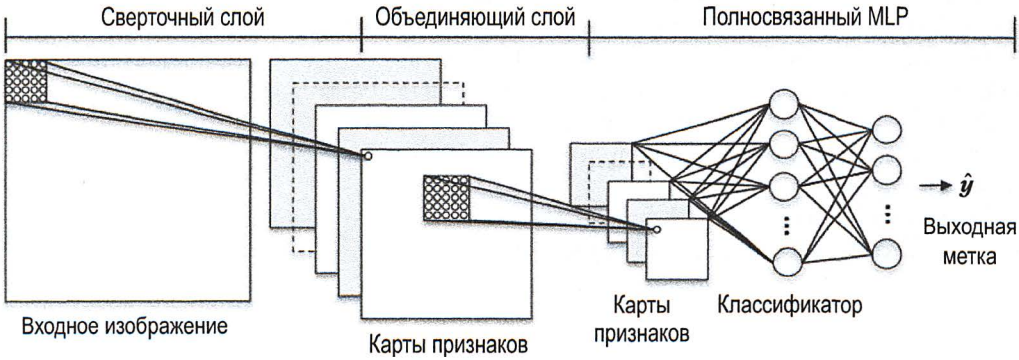
- *LeCun Y., Bottou L., Bengio Y., Haffner P.* Gradient-based Learning Applied to Document Recognition. Proceedings of the IEEE, 86(11):2278-2324, 1998, («*Градиентное обучение применительно к распознаванию документов*»);
- *Simard P. Y., Steinkraus D., Platt J. C.* Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis. IEEE, 2003, p. 958 («*Наиболее успешные практические методы для сверточных нейронных сетей применительно к анализу визуальных документов*»).

Как вы помните из нашей реализации многослойного персептрона, мы развернули изображения в векторы признаков, и эти входы были полностью связаны со скрытым слоем – пространственная информация в эту сетевую архитектуру не была закодирована. В нейронных сетях CNN мы используем **рецепторные поля** (receptive fields) для соединения входного слоя с картой признаков. Эти рецепторные поля могут пониматься как перекрывающиеся окна, которые мы двигаем по пикселям входного изображения для создания карты признаков. Длины шагов скольжения окна и размер окна являются дополнительными гиперпараметрами модели, которую мы должны задать *априорно*. Процесс создания **карты признаков** также называется **сверткой**. Пример такого **сверточного слоя**, соединяющего входные пикселы с каждым узлом в карте признаков, показан на нижеследующем рисунке:



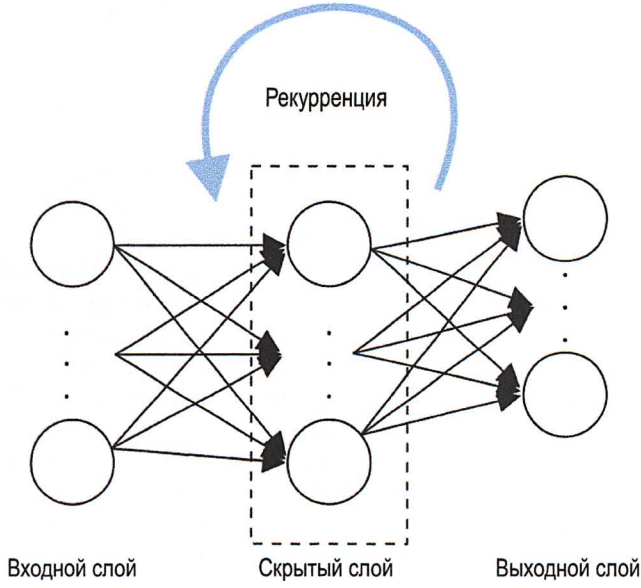
Важно отметить, что детекторы признаков дублируются, т. е. рецепторные поля, которые ставят в соответствие признакам узлы в следующем слое, имеют те же веса. Здесь ключевая идея состоит в том, что если детектор признаков полезен в одной части изображения, то он может оказаться полезным и в другой части тоже. Хороший побочный эффект этого подхода состоит в том, что он значительно сокращает количество параметров, которые должны быть извлечены. Поскольку различным участкам изображения разрешено быть представленными по-разному, нейронные сети CNN в особенности хороши в распознавании объектов на изображении различных размеров и различных положений. Нам особо не нужно беспокоиться о нормализации и центрировании изображений, так как это делалось в MNIST.

В нейронных сетях CNN за сверточным слоем следует **объединяющий слой** (pooling layer) (иногда также именуемый **подвыборкой**). При объединении мы обобщаем окрестные детекторы признаков для сокращения количества признаков для следующего слоя. Объединение может пониматься как простой метод выделения признаков, где мы берем среднее или максимальное значение участка окрестных признаков и передаем его в следующий слой. Для создания глубокой сверточной нейронной сети мы складываем два и более слоев в ярусы, чередуя сверточные и объединяющие слои, после чего мы соединяем их с многослойным персептроном для выполнения классификации. Это показано на нижеследующем рисунке:



Рекуррентные нейронные сети

Рекуррентные нейронные сети (РНС, recurrent neural network, RNN) можно рассматривать как нейронные сети прямого распространения сигналов с циклами обратной связи или обратным распространением по времени. В нейронных сетях RNN нейроны срабатывают только в течение ограниченного количества времени, прежде чем они будут (временно) деактивированы. В свою очередь, эти нейроны активируют другие нейроны, которые срабатывают в более поздний момент времени. В своей основе рекуррентные нейронные сети можно представить как MLP с дополнительной переменной времени. Временная составляющая и динамическая структура дает сети возможность использовать не только текущие входы, но и входы, которые она уже встречала ранее.



Несмотря на то что нейронные сети RNN достигли замечательных результатов в распознавании речи, машинном переводе и распознавании связного почерка, эту сетевую архитектуру, как правило, намного труднее тренировать. Это вызвано тем, что мы не можем просто распространять ошибку назад от слоя к слою; мы должны учитывать дополнительный компонент времени, который усиливает проблему исчезновения и взрыва градиента. В 1997 г. Юрген Шмидхубер и его коллеги представили так называемые **долгие кратковременные элементы памяти** для преодоления этой проблемы: Long Short Term Memory (LSTM); *Hochreiter S., Schmidhuber J.* Long Short-term Memory. Neural Computation, 9(8):1735–1780, 1997 (*«Долгая кратковременная память»*).

Однако мы должны отметить, что существует много разных вариантов нейронных сетей RNN, и их детальное обсуждение выходит за рамки этой книги.

Несколько последних замечаний по реализации нейронной сети

Вам, наверное, интересно, почему мы прошлись по всей этой теории только для того, чтобы в итоге реализовать простую многослойную искусственную сеть, которая может классифицировать рукописные цифры, вместо того чтобы воспользоваться библиотекой Python машинного обучения с открытым исходным кодом. Одна из причин кроется в том, что во время написания этой книги в библиотеке scikit-learn еще не было реализации многослойного персептрона (MLP). Что еще более важно, мы (практики машинного обучения) должны иметь, по крайней мере, базовое понимание алгоритмов, которые мы используем для применения методов машинного обучения корректным и успешным образом.

Зная, как работают нейронные сети прямого распространения сигнала, мы теперь готовы разобрать более современные библиотеки Python, построенные поверх NumPy, такие как Theano (http://deeplearning.net/software/theano/), которая позволяет более эффективно строить нейронные сети. Мы увидим это в *главе 13 «Распараллеливание тренировки нейронных сетей при помощи Theano»*. За последние несколько лет библиотека Theano приобрела большую популярность среди исследователей в области машинного обучения, которые используют ее для построения глубоких нейронных сетей, по причине ее способности оптимизировать математические выражения для вычислений на многомерных массивах с использованием **графических процессоров** (ГП, GPU).

Большую коллекцию практических руководств по Theano можно найти на http://deeplearning.net/software/theano/tutorial/index.html#tutorial.

Существует также целый ряд интересных библиотек, активно разрабатываемых для тренировки нейронных сетей в Theano, которые вы должны взять на заметку:

- **Pylearn2** (http://deeplearning.net/software/pylearn2/);
- **Lasagne** (https://lasagne.readthedocs.org/en/latest/);
- **Keras** (http://keras.io).

Резюме

В этой главе вы узнали о самых важных идеях, лежащих в основе многослойных искусственных нейронных сетей, которые в настоящее время являются самой горячей темой исследований в области машинного обучения. В *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»* мы начали наше путешествие с простых однослойных нейросетевых структур, затем мы соединили многослойные нейроны в мощную нейросетевую архитектуру для решения таких сложных задач, как распознавание рукописных цифр. Мы сняли покров загадочности с популярного алгоритма обратного распространения ошибки, который является одним из базовых элементов многих моделей нейронных сетей, использующихся в глубоком обучении. Изучив алгоритм обратного распространения ошибки, мы смогли обновить веса сложной нейронной сети. Мы также добавили полезные видоизменения, такие как мини-пакетное обучение и адаптивный темп обучения, которые позволяют тренировать нейронную сеть более эффективно.

Распараллеливание тренировки нейронных сетей при помощи Theano

В предыдущей главе мы прошлись по большому количеству математических понятий, чтобы разобраться в том, как работают искусственные нейронные сети прямого распространения сигналов, в частности в работе многослойных персептронов. Прежде всего наличие хорошего понимания математической подоплеки алгоритмов машинного обучения очень важно, поскольку оно помогает нам использовать эти мощные алгоритмы наиболее эффективно и *правильно*. В предыдущих главах вы посвятили много времени изучению наиболее успешных практических методов машинного обучения и даже попрактиковались в самостоятельной реализации алгоритмов с нуля. В этой главе вы можете почивать на лаврах, откинувшись на спинку кресла, и наслаждаться увлекательной поездкой по одной из самых мощных библиотек, которая используется исследователями в области машинного обучения для экспериментирования с глубокими нейронными сетями и очень эффективной их тренировки. Значительная часть современных научных исследований использует компьютеры с мощными **графическими процессорами** (ГП, GPU). Если вам интересно окунуться в глубокое обучение, которое в настоящее время является самой горячей темой в исследованиях в области машинного обучения, то эта глава определенно для вас. Однако не переживайте, если у вас нет доступа к ГП; в этой главе использование ГП будет факультативным.

Прежде чем мы начнем, сделаем краткий обзор тем, которые мы затронем в этой главе:

- написание оптимизированного исходного кода с использованием методов машинного обучения при помощи библиотеки Theano;
- выбор функций активации для искусственных нейронных сетей;
- использование библиотеки глубокого обучения Keras с целью проведения быстрых и простых экспериментов.

Сборка, компиляция и выполнение выражений в Theano

В этом разделе мы проанализируем мощный инструмент – программную библиотеку Theano, разработанную для наиболее эффективной тренировки машиннообучаемых моделей с использованием языка Python. Разработка библиотеки Theano началась еще в 2008 г. в лаборатории **LISA** (сокращенно от **Laboratoire d'Informatique des Systèmes Adaptatifs**, Монреаль, Канада, фр. Лаборатория адаптивных систем (http://lisa.iro.umontreal.ca)) под руководством Й. Бенджо (Yoshua Bengio).

Прежде чем мы обсудим, что из себя представляет библиотека Theano в действительности и что она может сделать для нас, чтобы ускорить наши задачи машинного обучения, обсудим некоторые трудности, когда мы выполняем затратные вычисления на наших аппаратных средствах. К счастью, производительность компьютерных процессоров с течением времени продолжает постоянно улучшаться. Это позволяет нам тренировать более мощные и сложные системы обучения с улучшенной предсказательной способностью наших машиннообучаемых моделей. Даже самые дешевые из имеющихся сегодня настольных компьютеров поставляются с процессорами, имеющими два и более ядер. В предыдущих главах мы увидели, что многие функции в библиотеке scikit-learn позволяют распределять вычисления по нескольким процессорам. Однако по причине **глобальной блокировки интерпретатора** (global interpreter lock, GIL) по умолчанию Python ограничивается выполнением на одном ядре. Однако, несмотря на то что для распределения вычислений по двум и более ядрам мы используем в своих интересах его библиотеку multiprocessing, мы должны учитывать, что даже продвинутые настольные компьютеры редко имеют больше 8 или 16 таких ядер.

Если мы вспомним предыдущую главу, где мы реализовали очень простой многослойный персептрон всего с одним скрытым слоем, состоящим из 50 узлов, то там нам уже пришлось выполнять оптимизацию приблизительно 1000 весовых коэффициентов, чтобы извлечь модель для очень простой задачи классификации изображений. Изображения в MNIST довольно маленькие (28×28 пикселов), и мы можем только вообразить взрывной рост числа параметров, если мы захотим добавить дополнительные скрытые слои или работать с изображениями, имеющими более высокую плотность пикселов. Такая задача быстро стала бы невыполнимой для одного-единственного процессора. Теперь вопрос состоит в том, каким образом можно преодолевать такие проблемы эффективнее. Очевидное решение этой проблемы состоит в том, чтобы использовать графические процессоры (ГП). Графический процессор – это настоящий энергетик. Видеокарту можно представить как небольшой компьютерный кластер внутри вашей машины. Еще одно его преимущество состоит в том, что, как видно из следующей ниже обзорной таблицы, современные ГП относительно дешевы, по сравнению с современным ЦП:

Спецификации	Intel® Core™ i7-5960X Processor Extreme Edition	NVIDIA GeForce® GTX™ 980 Ti
Базовая частота процессора	3,0 ГГц	1,0 ГГц
Количество ядер	8	2816
Пропускная способность памяти	68 Гб/с	336,5 Гб/с
Вычисления с плавающей запятой	354 флоп/с	5632 флоп/с
Стоимость, долл.	1000	700

Источник приведенной выше информации можно найти на следующих веб-сайтах:

- http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980-ti/specifications;
- http://ark.intel.com/products/82930/Intel-Core-i7-5960XProcessor-Extreme-Edition-20M-Cache-up-to-3_50-GHz (дата: 20 августа 2015 г.).

За 70% цены современного ЦП мы можем получить ГП, который имеет в 450 раз больше ядер и способен выполнять приблизительно в 15 раз больше вычислений с плавающей точкой в секунду. Так что же сдерживает нас от использования гэпов

в наших задачах машинного обучения? Трудность состоит в том, что написание исходного кода, предназначенного для гэпов, не так тривиально, как выполнение исходного кода Python в нашем интерпретаторе. Для этого предназначены специальные пакеты, такие как CUDA и OpenCL, позволяющие перенацеливать исходный код для ГП. Однако программные среды CUDA или OpenCL – это, вероятно, не самые удобные среды для реализации и выполнения алгоритмов машинного обучения. Хорошая новость состоит в том, что именно для этого и была разработана библиотека Theano!

Что такое Theano?

Что конкретно из себя представляет Theano – язык программирования, компилятор или программная библиотека Python? Оказывается, что она соответствует всем этим описаниям. Библиотека Theano была разработана для того, чтобы очень эффективно реализовывать, компилировать и вычислять математические выражения с повышенным вниманием на многомерных массивах (тензорах). По выбору исходный код может выполняться на центральном процессоре (процессорах). Однако его настоящая мощность заключается в использовании гэпов, чтобы воспользоваться преимуществом большой пропускной способности памяти и больших возможностей математических вычислений с плавающей точкой. Используя библиотеку Theano, можно также легко выполнять исходный код параллельно по всей общей (совместно используемой) памяти. В 2010 г. разработчики библиотеки Theano сообщили о 1.8-кратном превышении производительности, по сравнению с библиотекой NumPy, когда исходный код выполнялся на ЦП, и если Theano перенацеливался на ГП, то его быстродействие было в 11 раз выше, чем в NumPy (*Bergstra J., Breuleux O., Bastien F., Lamblin P., Pascanu R., Desjardins G., Turian J., Warde-Farley D., Bengio Y.* Theano: A CPU and GPU Math Compiler in Python. In Proc. 9th Python in Science Conf, p. 1–7, 2010 (*«Theano: математический компилятор на основе ЦП и ГП на Python»*)). При этом стоит иметь в виду, что это тестирование проводилось в 2010 г., и за эти годы библиотека Theano была значительно усовершенствована, так же как и возможности современных видеокарт.

Итак, каким образом библиотека Theano связана с библиотекой NumPy? Библиотека Theano опирается на библиотеку NumPy и имеет очень похожий синтаксис, что делает его использование очень удобным для людей, которые уже знакомы с последней. Ради справедливости библиотека Theano – это не просто «NumPy на стероидах», как многие ее бы описали, ввиду того, что она также имеет много общего с библиотекой Python SymPy (http://www.sympy.org) для символических вычислений (или символической алгебры). Как мы уже видели в предыдущих главах, в NumPy мы описываем то, чем являются наши переменные, и как мы хотим их объединить; и затем исходный код выполняется построчно, строка за строкой. В Theano же мы сначала записываем задачу и описываем то, как мы хотим ее проанализировать. Затем Theano оптимизирует и компилирует исходный код за нас, используя для этого C/C++ либо CUDA/OpenCL, если мы хотим выполнять его на ГП. Для того чтобы сгенерировать для нас оптимизированный код, библиотека Theano должна знать объем нашей задачи; ее можно представить как дерево операций (либо граф символических выражений). Отметим, что библиотека Theano все еще находится в процессе активной разработки, и много нового функционала добавляет-

ся, и улучшения вносятся на регулярной основе. В этой главе мы разведаем основные принципы работы библиотеки Theano и научимся использовать ее для задач машинного обучения. Поскольку Theano – большая библиотека с большим числом продвинутого функционала, было бы невозможно охватить их все в этой книге. Однако если вы захотите узнать об этой библиотеке больше, то ниже предоставлены полезные ссылки на ее превосходную онлайновую документацию (http://deeplearning.net/software/theano/).

Первые шаги с библиотекой Theano

В этом разделе мы сделаем наши первые шаги с библиотекой Theano. В зависимости от того, как настроена ваша система, как правило, можно просто использовать инсталлятор pip библиотек Python и установить Theano из каталога пакетов PyPI, выполнив следующую ниже команду из командной строки терминала:

```
pip install Theano
```

В случае если вы испытываете проблемы с процессом установки, рекомендуем прочитать больше о системе и платформозависимых рекомендациях, предоставленных на http://deeplearning.net/software/theano/install.html. Отметим, что весь исходный код в этой главе может выполняться на вашем ЦП; использование ГП совершенно факультативное, но рекомендовано, если вы хотите полностью пользоваться преимуществами Theano. Если у вас есть видеокарта, поддерживающая CUDA либо OpenCL, пожалуйста, обратитесь к обновляемому пособию на http://deeplearning.net/software/theano/tutorial/using_gpu.html#using-gpu, чтобы настроить ее соответствующим образом.

В своей основе библиотека Theano построена вокруг так называемых тензоров для вычисления символьных математических выражений. Тензоры могут пониматься как обобщение скаляров, векторов, матриц и т. д. Говоря более конкретно, скаляр может быть определен как тензор нулевого ранга, вектор – как тензор 1-го ранга, матрица – как тензор 2-го ранга, и матрицы, сложенные ярусами в третьем измерении, – как тензоры 3-го ранга. В качестве разминочного упражнения мы начнем с использования простых скаляров из модуля tensor библиотеки Theano и вычислим чистый вход z точки образца x в одномерном наборе данных с весом w_1 и смещением w_0:

$$z = x_1 \times w_1 + w_0.$$

Исходный код следующий:

```
>>>
import theano
from theano import tensor as T

# инициализировать
x1 = T.scalar()
w1 = T.scalar()
w0 = T.scalar()
z1 = w1 * x1 + w0

# скомпилировать
```

```
net_input = theano.function(inputs=[w1, x1, w0],
                            outputs=z1)

# исполнить
print('Чистый вход: %.2f' % net_input(2.0, 1.0, 0.5))
Чистый вход: 2.50
```

Это было довольно прямолинейно, согласитесь? Если мы пишем исходный код в Theano, мы просто должны выполнить три простых шага: определить *символы* (объекты Variable), скомпилировать исходный код и выполнить его. На шаге инициализации мы определили три символа, x1, w1 и w0, для вычисления z1. Затем мы скомпилировали функцию net_input, чтобы вычислить чистый вход z1.

Однако существует одна отдельно взятая деталь, заслуживающая особого внимания, в случае если мы пишем исходный код Theano: тип переменных (dtype). Можно рассматривать это как бремя или проклятие, но в Theano мы должны выбирать между 64-разрядными либо 32-разрядными целочисленными либо числами с плавающей точкой; этот выбор в значительной мере влияет на производительность исходного кода. Обсудим эти типы переменных более подробно в следующем разделе.

Конфигурирование библиотеки Theano

В наше время независимо от того, работаем мы в Mac OS X, Linux или Microsoft Windows, мы в основном используем программное обеспечение и приложения с 64-разрядными адресами памяти. Однако если мы хотим ускорить вычисление математических выражений на ГП, то мы все еще часто полагаемся на более старые 32-разрядные адреса памяти. В настоящее время это единственная поддерживаемая вычислительная архитектура в Theano. В этом разделе мы увидим, как правильно сконфигурировать Theano. Если вы интересуетесь подробностями по поводу конфигурирования Theano, пожалуйста, обратитесь к онлайновой документации на http://deeplearning.net/software/theano/library/config.html.

Когда мы реализуем алгоритмы машинного обучения, мы главным образом работаем с числами с плавающей точкой. По умолчанию обе библиотеки – и NumPy, и Theano – используют формат с плавающей точкой двойной точности (float64). Однако было бы действительно полезно переключаться между float64 (ЦП) и float32 (ГП), когда мы разрабатываем исходный код Theano для прототипирования на ЦП и исполнения на ГП. Например, для доступа к настройкам по умолчанию для переменных с плавающей точкой библиотеки Theano мы можем выполнить в нашем интерпретаторе Python следующий ниже фрагмент исходного кода:

```
>>>
print(theano.config.floatX)
float64
```

Если вы не изменили настроек после установки Theano, то плавающая точка по умолчанию должна иметь разрядность float64. Однако следующей ниже командой мы можем просто поменять ее на float32 в нашем текущем сеансе Python:

```
theano.config.floatX = 'float32'
```

Отметим, что если в настоящее время использование ГП в Theano требует типов с разрядностью float32, то на наших ЦП мы можем использовать оба типа: float64

и float32. Таким образом, если вы хотите глобально поменять настройки по умолчанию, то вы можете изменить настройки в своей переменной THEANO_FLAGS при помощи команды (Bash) в терминале:

```
export THEANO_FLAGS=floatX=float32
```

Как вариант вы можете применить эти настройки только к определенному сценарию Python, выполнив их следующим образом:

```
THEANO_FLAGS=floatX=float32 python your_script.py
```

До сих пор мы обсуждали, как устанавливать типы с плавающей запятой, используемые по умолчанию, чтобы получить наилучшую отдачу на нашем ГП с использованием Theano. Далее обсудим варианты настроек для переключения между выполнением на ЦП и ГП. Если выполнить следующую ниже команду, то мы сможем проверить, что мы используем – ЦП или ГП:

```
>>>
print(theano.config.device)
cpu
```

Моя личная рекомендация состоит в том, чтобы использовать cpu по умолчанию. Это позволяет упростить прототипирование и отладку кода. Например, можно выполнить исходный код Theano на своем ЦП, выполнив его из командной строки в окне терминала как сценарий:

```
THEANO_FLAGS=device=cpu,floatX=float64 python your_script.py
```

Однако, после того как мы реализовали программный код и хотим выполнить его самым эффективным образом с использованием наших аппаратных средств ГП, тогда мы можем выполнить его с помощью следующего ниже кода, не внося дополнительных модификаций в наш исходный код:

```
THEANO_FLAGS=device=gpu,floatX=float32 python your_script.py
```

Кроме того, может быть удобным создать в своем корневом (домашнем) каталоге файл .theanorc, чтобы сделать эти настройки конфигурации постоянными. Например, чтобы всегда использовать float32 и ГП, вы можете создать такой файл .theanorc, включив приведенные ниже настройки. Команда следующая:

```
echo -e "\n[global]\nfloatX=float32\ndevice=gpu\n" >> ~/.theanorc
```

Если вы не работаете в терминале Mac OS X или Linux, то можете создать файл .theanorc вручную с помощью предпочитаемого текстового редактора и добавить в него следующее содержимое:

```
[global]
floatX=float32
device=gpu
```

Зная, как правильно конфигурировать Theano в соответствии с имеющимися аппаратными средствами, теперь в следующем разделе мы можем обсудить, как использовать более сложные матричные структуры.

Работа с матричными структурами

В этом разделе мы обсудим вопрос использования матричных структур в Theano при помощи его модуля tensor. Выполнив следующий ниже исходный код, мы создадим простую матрицу размера 2×3 и вычислим суммы столбцов, используя для этого оптимизированные тензорные выражения Theano:

```
>>>
import numpy as np

# инициализировать
# если вы выполняете Theano в 64-разрядном режиме, то
# вам нужно использовать dmatrix вместо fmatrix
x = T.fmatrix(name='x')
x_sum = T.sum(x, axis=0)

# скомпилировать
calc_sum = theano.function(inputs=[x], outputs=x_sum)

# выполнить (сначала Python)
ary = [[1, 2, 3], [1, 2, 3]]
print('Сумма столбца:', calc_sum(ary))
Сумма столбца: [ 2. 4. 6.]

# выполнить (массив NumPy)
ary = np.array([[1, 2, 3], [1, 2, 3]],
               dtype=theano.config.floatX)
print('Сумма столбца:', calc_sum(ary))
Сумма столбца: [ 2. 4. 6.]
```

Как мы видели ранее, работая с библиотекой Theano, нам нужно выполнить всего три основных шага: определить переменную, скомпилировать код и выполнить его. Предыдущий пример показывает, что Theano может работать как с типами Python, так и с типами NumPy: list и numpy.ndarray.

Отметим, что когда мы создавали тензорную переменную fmatrix, которая может пригодиться во время отладки нашего кода либо распечатки графа Theano, мы использовали дополнительный именованный аргумент (в данном случае x). Например, если распечатать символ x тензорной переменной fmatrix, не давая ему имени name, функция печати print возвратит его тип тензора TensorType:

```
>>>
print(x)
<TensorType(float32, matrix)>
```

Однако если бы тензорная переменная TensorVariable была инициализирована с именованным аргументом x, как в нашем предыдущем примере, то функция печати вернула бы его:

```
>>>
print(x)
x
```

Доступ к типу тензора можно получить при помощи метода type:

```
>>>
print(x.type())
<TensorType(float32, matrix)>
```

Theano также имеет очень умную систему управления памятью, которая для ускорения работы использует память повторно. Если более конкретно, то Theano распределяет объем памяти по нескольким устройствам, ЦП и ГП; чтобы отслеживать изменения в объеме памяти, она назначает псевдонимы соответствующим буферам. Далее мы посмотрим на переменную shared, позволяющую распределять большие объекты (массивы) и предоставлять нескольким функциям чтения и записи право доступа, в результате чего после компиляции мы также можем выполнять обновления на этих объектах. Подробное описание обработки памяти в Theano выходит за рамки этой книги. Поэтому призываем вас отслеживать обновление информации о Theano и об управлении памятью на http://deeplearning.net/software/theano/tutorial/aliasing.html.

```
>>>
# инициализировать
x = T.fmatrix('x')
w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
                             dtype=theano.config.floatX))
z = x.dot(w.T)
update = [[w, w + 1.0]]

# скомпилировать
net_input = theano.function(inputs=[x],
                            updates=update,
                            outputs=z)

# исполнить
data = np.array([[1, 2, 3]],
                dtype=theano.config.floatX)
for i in range(5):
    print('z%d:' % i, net_input(data))

z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

Как видно, совместно использовать память в библиотеке Theano действительно просто: в приведенном выше примере мы определили переменную update, где мы объявили, что хотим обновлять массив w значением 1.0 после каждой итерации цикла for. После того как мы определили, какой объект мы хотим обновлять и каким образом, мы передали эту информацию параметру update компилятора theano.function.

Еще один ловкий трюк в библиотеке Theano состоит в том, чтобы использовать переменную givens для вставки значений в граф перед его компиляцией. Используя этот подход, мы можем сократить количество перемещений из оперативной памяти (RAM) через ЦП в ГП, ускоряя работу алгоритмов обучения, в которых используются совместно используемые (общие) переменные. Если в компиляторе theano.function использовать параметр inputs, то данные будут перемещаться из ЦП в ГП многократно, например если мы многократно (эпохи) выполняем итерации по набору данных во время градиентного спуска. Используя givens, мы можем держать набор данных на ГП, если он помещается в его памяти (например, если мы выполняем тренировку мини-пакетами). Исходный код следующий:

```
>>>
# инициализировать
data = np.array([[1, 2, 3]],
                dtype=theano.config.floatX)
x = T.fmatrix('x')
w = theano.shared(np.asarray([[0.0, 0.0, 0.0]],
                  dtype=theano.config.floatX))
z = x.dot(w.T)
update = [[w, w + 1.0]]

# скомпилировать
net_input = theano.function(inputs=[],
                            updates=update,
                            givens={x: data},
                            outputs=z)

# исполнить
for i in range(5):
    print('z:', net_input())

z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

Глядя на приведенный выше пример исходного кода, мы также видим, что атрибут givens – это словарь Python, который ставит в соответствие имени переменной фактический объект Python. Здесь мы назначили это имя, когда определяли fmatrix.

Завершающий пример – линейная регрессия

Ознакомившись с Theano, теперь обратимся к действительно практическому примеру и реализуем регрессию по методу **наименьших квадратов** (МНК, OLS). Чтобы быстро освежить память в отношении регрессионного анализа, пожалуйста, обратитесь к *главе 10 «Предсказание непрерывных целевых величин при помощи регрессионного анализа»*.

Начнем с создания небольшого одномерного миниатюрного набора данных с 10 тренировочными образцами:

```
X_train = np.asarray([[0.0], [1.0],
                      [2.0], [3.0],
                      [4.0], [5.0],
                      [6.0], [7.0],
                      [8.0], [9.0]],
                     dtype=theano.config.floatX)
y_train = np.asarray([1.0, 1.3,
                      3.1, 2.0,
                      5.0, 6.3,
                      6.6, 7.4,
                      8.0, 9.0],
                     dtype=theano.config.floatX)
```

Отметим, что когда мы конструируем массивы NumPy, мы используем theano.config.floatX, с тем чтобы произвольно переключаться туда и обратно между ЦП и ГП, если понадобится.

Далее реализуем тренировочную функцию для извлечения весов линейной регрессионной модели с использованием функции стоимости в виде суммы квадратичных ошибок. Отметим, что w_0 – это узел смещения (точка пересечения оси Y при $x = 0$). Исходный код следующий:

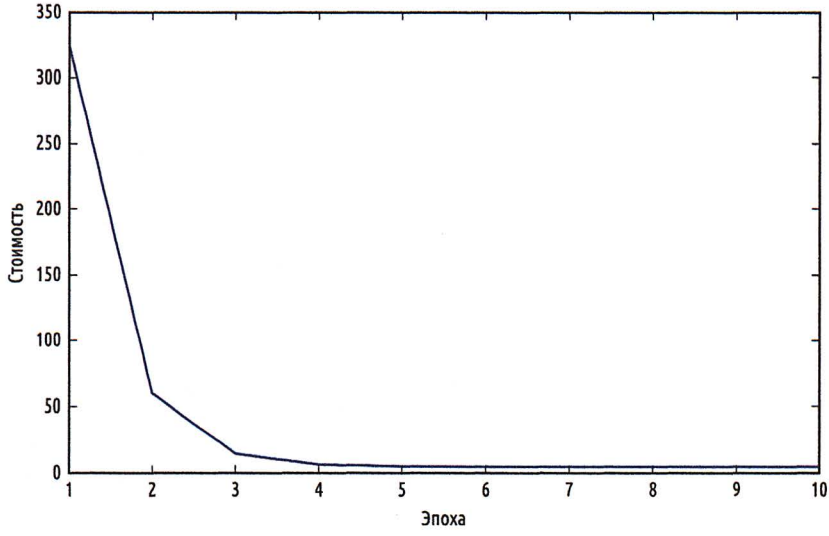
```
import theano
from theano import tensor as T
import numpy as np

def train_linreg(X_train, y_train, eta, epochs):

    costs = []

    # Инициализировать массивы
    eta0 = T.fscalar('eta0')
    y = T.fvector(name='y')
    X = T.fmatrix(name='X')
    w = theano.shared(np.zeros(
                      shape=(X_train.shape[1] + 1),
                      dtype=theano.config.floatX),
                      name='w')

    # Вычислить стоимость
    net_input = T.dot(X, w[1:]) + w[0]
    errors = y - net_input
    cost = T.sum(T.pow(errors, 2))

    # Выполнить корректировку градиента
    gradient = T.grad(cost, wrt=w)
    update = [(w, w - eta0 * gradient)]

    # Скомпилировать модель
    train = theano.function(inputs=[eta0],
                            outputs=cost,
                            updates=update,
                            givens={X: X_train,
                                    y: y_train,})
    for _ in range(epochs):
        costs.append(train(eta))

    return costs, w
```

Действительно приятной особенностью в Theano является функция grad, которую мы использовали в предыдущем примере исходного кода. Функция grad автоматически вычисляет производную выражения *относительно* его параметров, которые мы передали в функцию в качестве аргумента wrt.

После того как мы реализовали тренировочную функцию, натренируем нашу линейную регрессионную модель и посмотрим на значения функции стоимости в виде **суммы квадратичных ошибок** (SSE), чтобы проверить ее сходимость:

```
import matplotlib.pyplot as plt
costs, w = train_linreg(X_train, y_train, eta=0.001, epochs=10)
plt.plot(range(1, len(costs)+1), costs)
plt.tight_layout()
plt.xlabel('Эпохи')
plt.ylabel('Стоимость')
plt.show()
```

Как видно на следующем ниже графике, алгоритм обучения сходился уже после пятой эпохи:



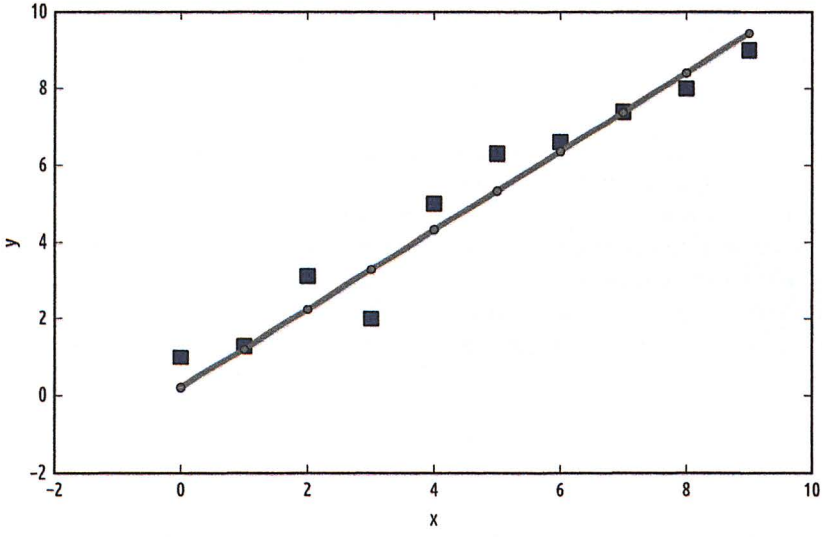
Пока неплохо; глядя на функцию стоимости, кажется, что мы построили рабочую регрессионную модель из этого отдельно взятого набора данных. Теперь скомпилируем новую функцию для выполнения предсказаний на основе входных признаков:

```
def predict_linreg(X, w):
    Xt = T.matrix(name='X')
    net_input = T.dot(Xt, w[1:]) + w[0]
    predict = theano.function(inputs=[Xt],
                              givens={w: w},
                              outputs=net_input)
    return predict(X)
```

Реализация функции predict была довольно прямолинейной, если следовать трехшаговой процедуре библиотеки Theano: определить, скомпилировать и исполнить. Далее построим график подгонки линейной регрессии на тренировочных данных:

```
plt.scatter(X_train,
            y_train,
            marker='s',
            s=50)
plt.plot(range(X_train.shape[0]),
         predict_linreg(X_train, w),
         color='gray',
         marker='o',
         markersize=4,
         linewidth=3)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Как видно на получившемся графике, наша модель должным образом аппроксимирует точки данных:



Реализация простой регрессионной модели была хорошим упражнением, позволяющим познакомиться с API Theano. Однако наша конечная цель – проявить преимущества библиотеки Theano, а именно реализация мощных искусственных нейронных сетей. Теперь мы должны быть вооружены всеми инструментами, которые понадобятся для реализации многослойного персептрона из *главы 12 «Тренировка искусственных нейронных сетей для распознавания изображений»* в Theano. Однако это было бы довольно скучным занятием, согласитесь? Поэтому мы обратимся к одной из моих любимых библиотек глубокого обучения, построенных поверх Theano, чтобы сделать экспериментирование с нейронными сетями максимально удобным. Впрочем, прежде чем представить вам библиотеку Keras, в следующем разделе сначала обсудим варианты функций активации в нейронных сетях.

Выбор функций активации для нейронных сетей с прямым распространением сигналов

Для простоты до настоящего момента мы обсудили только сигмоидальную функцию активации в контексте многослойных нейронных сетей с прямым распространением сигналов; мы использовали ее в скрытом и выходном слоях в реализации многослойного персептрона в *главе 12 «Тренировка искусственных нейронных сетей для распознавания изображений»*. Несмотря на то что мы обозначили эту функцию активации как *сигмоидальную* – как ее обычно называют в литературе, – более точным определением было бы *«логистическая функция»* или *«отрицательная логарифмическая функция правдоподобия»*. В следующих подразделах вы узнаете об альтернативных сигмоидальных функциях, которые широко используются для реализации многослойных нейронных сетей.

Строго говоря, в многослойных нейронных сетях в качестве функции активации можно использовать любую функцию, пока она дифференцируема. Можно было бы использовать даже линейные функции активации, например как в ADALINE (*гла-*

ва 2 «Тренировка алгоритмов машинного обучения для задачи классификации»). Однако на практике использовать линейные функции активации для скрытых и выходных слоев нецелесообразно, поскольку мы хотим внести в типичную искусственную нейронную сеть нелинейность, чтобы быть в состоянии решать сложные практические задачи. Сумма линейных функций в конечном счете дает линейную функцию.

Логистическая функция активации, которую мы использовали в предыдущей главе, вероятно, наиболее близко имитирует концепцию нейрона в головном мозгу человека: ее можно представить как вероятность того, сработает нейрон или нет. Однако логистические функции активации могут быть проблематичными, в случае если у нас очень отрицательные входы, поскольку в этом случае выход из сигмоидальной функции будет близким к нулю. Если сигмоидальная функция дает на выходе значения, которые близки к нулю, то нейронная сеть будет тренироваться очень медленно, и повышается вероятность, что во время тренировки она застрянет в локальных минимумах. По этой причине в скрытых слоях часто в качестве функции активации предпочитают использовать функцию **гиперболического тангенса**. Прежде чем мы обсудим, что из себя представляет функция гиперболического тангенса, кратко пройдемся по основным пунктам логистической функции и посмотрим на обобщение, которое делает ее более полезной для задач многоклассовой классификации.

Краткое резюме логистической функции

Как мы уже упоминали во введении в этот раздел, логистическая функция, часто именуемая просто *сигмоидой*, фактически представляет собой частный случай сигмоидальной функции. Из раздела о логистической регрессии в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»* мы помним, что логистическую функцию можно использовать для моделирования вероятности, что образец x принадлежит положительному классу (класс 1) в задаче бинарной классификации:

$$\phi_{\text{логистическая}}(z) = \frac{1}{1+e^{-z}}.$$

Здесь скалярная переменная z определяется как чистый вход:

$$z = w_0 x_0 + \ldots + w_m x_m = \sum_{j=0}^{m} x_j w_j = \boldsymbol{w}^T \boldsymbol{x}.$$

Отметим, что w_0 – это узел смещения (точка пересечения оси Y, $x_0 = 1$). В качестве более конкретного примера примем модель для двумерной точки данных x и модель со следующими весовыми коэффициентами, присвоенными вектору w:

```
>>>
X = np.array([[1, 1.4, 1.5]])
w = np.array([0.0, 0.2, 0.4])

def net_input(X, w):
    z = X.dot(w)
    return z

def logistic(z):
```

```
    return 1.0 / (1.0 + np.exp(-z))

def logistic_activation(X, w):
    z = net_input(X, w)
    return logistic(z)

print('P(y=1|x) = %.3f'
      % logistic_activation(X, w)[0])

P(y=1|x) = 0.707
```

Если вычислить чистый вход и использовать его для активации логистического нейрона с этими отдельно взятыми значениями признаков и весовыми коэффициентами, то мы в результате получим значение 0.707, которое можно интерпретировать как вероятность в размере 70.7%, что эта отдельно взятый образец x принадлежит положительному классу. В *главе 12 «Тренировка искусственных нейронных сетей для распознавания изображений»* мы использовали метод прямого кодирования для вычисления значений в выходном слое, состоящем из двух и более узлов логистической активации. Однако, как будет показано на следующем ниже примере исходного кода, выходной слой, состоящий из двух и более узлов логистической активации, не производит содержательных, поддающихся интерпретации значений вероятности:

```
>>>
# W : массив, форма = [n_output_units, n_hidden_units+1]
# Весовая матрица для скрытого слоя -> выходной слой.
# Примечание: первый столбец (W[:][0]) содержит узлы смещения
W = np.array([[1.1, 1.2, 1.3, 0.5],
              [0.1, 0.2, 0.4, 0.1],
              [0.2, 0.5, 2.1, 1.9]])

# A : массив, форма = [n_hidden+1, n_samples]
# Активация скрытого слоя.
# Примечание: первый узел (A[0][0] = 1) - узел смещения
A = np.array([[1.0],
              [0.1],
              [0.3],
              [0.7]])

# Z : массив, форма = [n_output_units, n_samples]
# Чистый вход выходного слоя.
Z = W.dot(A)
y_probas = logistic(Z)
print('Вероятности:\n', y_probas)

Вероятности:
[[ 0.87653295]
[ 0.57688526]
[ 0.90114393]]
```

Как видно по результату на выходе, вероятность, что отдельно взятый образец принадлежит первому классу, составляет почти 88%, соответственно, вероятность, что отдельно взятый образец принадлежит второму классу, составляет почти 58%, и вероятность, что отдельно взятый образец принадлежит третьему классу, составляет 90%. Это, несомненно, сбивает с толку, поскольку всем известно, что процент должен интуитивно выражаться как доля от 100. Но фактически не вызывает особо-

го беспокойства, если мы используем нашу модель только для предсказания меток классов, а не вероятностей принадлежности классу.

```
>>>
y_class = np.argmax(Z, axis=0)
print('Предсказанная метка класса: %d' % y_class[0])
Предсказанная метка класса: 2
```

Однако в определенных контекстах может быть полезным вернуть содержательные вероятности классов, подходящих для многоклассовых предсказаний. В следующем разделе мы посмотрим на обобщение логистической функции, функцию **softmax**, которая способна помочь нам с этой задачей.

Оценивание вероятностей в многоклассовой классификации функцией softmax

Функция **softmax**, или нормализованная экспоненциальная функция, – это обобщение логистической функции, которая позволяет вычислять содержательные вероятности классов в многоклассовых конфигурациях (мультиномиальная логистическая регрессия). В softmax вероятность отдельно взятого образца с чистым входом z, принадлежащую i-му классу, можно вычислить с нормализующим членом в знаменателе, т. е. суммой всех линейных функций M:

$$P(y=i\,|\,z)=\phi_{\text{softmax}}(z)=\frac{e_i^z}{\sum_{m=1}^{M} e_m^z}.$$

Чтобы посмотреть на функцию softmax в действии, реализуем ее исходный код на Python:

```
>>>
def softmax(z):
    return np.exp(z) / np.sum(np.exp(z))

def softmax_activation(X, w):
    z = net_input(X, w)
    return softmax(z)

y_probas = softmax(Z)
print('Вероятности:\n', y_probas)

Вероятности:
[[ 0.40386493]
[ 0.07756222]
[ 0.51857284]]

y_probas.sum()
1.0
```

Как видно, предсказанные вероятности классов теперь в сумме равны единице, как мы и ожидали. Также стоит отметить, что вероятность для второго класса близка к нулю, поскольку имеется большой разрыв между z_1 и $\max(z)$. Однако отметим, что предсказанная метка класса такая же, что и в логистической функции. Интуитивно функция softmax может представляться как *нормализованная* логистическая

функция, которая в многоклассовых конфигурациях часто используется для получения содержательных предсказаний принадлежности классу.

```
>>>
y_class = np.argmax(Z, axis=0)
print('Предсказанная метка класса:
      %d' % y_class[0])

Предсказанная метка класса: 2
```

Расширение выходного спектра при помощи гиперболического тангенса

В скрытых слоях искусственных нейронных сетей часто используется еще одна сигмоидальная функция – функция **гиперболического тангенса** (tanh), которую можно интерпретировать как приведенную версию логистической функции:

$$\phi_{\tanh}(z) = \phi_{\text{логистическая}}(2 \times z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}};$$

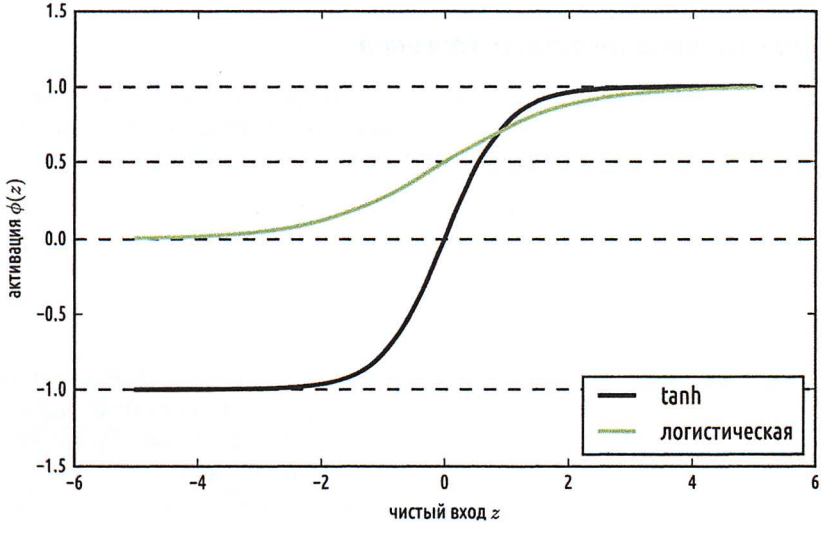
$$\phi_{\text{логистическая}}(z) = \frac{1}{1 + e^{-z}}.$$

Преимущество функции гиперболического тангенса над логистической функцией состоит в том, что она имеет более широкий выходной спектр и располагается в интервале (–1, 1), что может улучшить сходимость алгоритма обратного распространения (*Bishop C. M.* Neural networks for pattern recognition. Oxford university press, 1995, p. 500–501 («*Нейронные сети для распознавания образов*»)). Напротив, логистическая функция возвращает выходной сигнал, располагающийся в интервале (0, 1). Для интуитивного сравнения логистической функции и гиперболического тангенса построим график двух сигмоидальных функций:

```
import matplotlib.pyplot as plt

def tanh(z):
    e_p = np.exp(z)
    e_m = np.exp(-z)
    return (e_p - e_m) / (e_p + e_m)

z = np.arange(-5, 5, 0.005)
log_act = logistic(z)
tanh_act = tanh(z)
plt.ylim([-1.5, 1.5])
plt.xlabel('чистый вход $z$')
plt.ylabel('активация $\phi(z)$')
plt.axhline(1, color='black', linestyle='--')
plt.axhline(0.5, color='black', linestyle='--')
plt.axhline(0, color='black', linestyle='--')
plt.axhline(-1, color='black', linestyle='--')
plt.plot(z, tanh_act,
         linewidth=2,
         color='black',
         label='tanh')
plt.plot(z, log_act,
         linewidth=2,
```

```
        color='lightgreen',
        label='логистическая')
plt.legend(loc='lower right')
plt.tight_layout()
plt.show()
```

Как видно, формы двух сигмоидальных кривых выглядят очень похожими; однако функция tanh имеет выходное пространство в 2 раза больше, чем в логистической функции:



Отметим, что мы реализовали функции logistic и tanh многословно в целях иллюстрации. На практике мы можем использовать функцию tanh библиотеки NumPy и получить те же результаты:

```
tanh_act = np.tanh(z)
```

Кроме того, логистическая функция logistic имеется в специальном модуле библиотеки SciPy:

```
from scipy.special import expit
log_act = expit(z)
```

Зная больше о разных функциях активации, которые широко используются в искусственных нейронных сетях, теперь завершим этот раздел обзором различных функций активации, с которыми мы столкнулись в этой книге.

Функция активации	Формула	Пример	Изображение
Единичная ступенчатая (Хевисайд)	$\phi(z)=\begin{cases}0, & z<0,\\ 0.5, & z=0,\\ 1, & z>0\end{cases}$	Вариант персептрона	
Кусочно-постоянная (сигнум)	$\phi(z)=\begin{cases}-1, & z<0,\\ 0, & z=0,\\ 1, & z>0\end{cases}$	Вариант персептрона	
Линейная	$\phi(z)=z$	ADALINE, линейная регрессия	
Кусочно-линейная	$\phi(z)=\begin{cases}1, & z\geq\frac{1}{2},\\ z+\frac{1}{2}, & -\frac{1}{2}<z<\frac{1}{2},\\ 0, & z\leq-\frac{1}{2}\end{cases}$	Метод опорных векторов	
Логистическая (сигмоида)	$\phi(z)=\frac{1}{1+e^{-z}}$	Логистическая регрессия, многослойная НС	
Гиперболический тангенс	$\phi(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	Многослойная НС	

Эффективная тренировка нейронных сетей при помощи библиотеки Keras

В этом разделе мы рассмотрим библиотеку Keras, одну из последних разработок, предназначенных для упрощения тренировки нейронных сетей. Разработка библиотеки Keras началась в самом начале 2015 г.; и на сегодняшний день она эволюционировала в одну из самых популярных и широко используемых библиотек, построенных поверх библиотеки Theano, которая позволяет для ускорения тренировки нейронных сетей использовать графический процессор (ГП). Одна из ее ярких черт состоит в ее очень интуитивном API, позволяющем реализовывать нейронные сети всего в нескольких строках исходного кода. Установив библиотеку Theano, вы можете установить Keras из каталога пакетов PyPI при помощи следующей команды из командной строки в окне терминала:

```
pip install Keras
```

Для получения дополнительной информации о библиотеке Keras, пожалуйста, посетите ее официальный веб-сайт: http://keras.io.

Чтобы увидеть, как выглядит тренировка нейронной сети при помощи библиотеки Keras, реализуем многослойный персептрон для классификации рукописных

цифр из набора данных MNIST, который мы представили в предыдущей главе. Набор данных MNIST можно скачать с http://yann.lecun.com/exdb/mnist/ в четырех частях, как показано ниже:

- train-images-idx3-ubyte.gz: тренировочный набор изображений (9 912 422 байта);
- train-labels-idx1-ubyte.gz: тренировочный набор меток (28 881 байт);
- t10k-images-idx3-ubyte.gz: тестовый набор изображений (1 648 877 байт);
- t10k-labels-idx1-ubyte.gz: тестовый набор меток (4542 байта).

После скачивания и разархивирования мы помещаем файлы в каталог mnist в нашем текущем рабочем каталоге, чтобы можно было загружать тренировочный и тестовый наборы данных при помощи следующей ниже функции:

```
>>>
import os
import struct
import numpy as np

def load_mnist(path, kind='train'):
    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                                % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                                % kind)

    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II',
                                 lbpath.read(8))
        labels = np.fromfile(lbpath,
                             dtype=np.uint8)

    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
                                               imgpath.read(16))
        images = np.fromfile(imgpath,
                             dtype=np.uint8).reshape(len(labels), 784)
    return images, labels

X_train, y_train = load_mnist('mnist', kind='train')
print('Тренировка - строки: %d, столбцы: %d' % (X_train.shape[0], X_train.shape[1]))
Тренировка - строки: 60 000, столбцы: 784

X_test, y_test = load_mnist('mnist', kind='t10k')
print('Тестирование - строки: %d, столбцы: %d' % (X_test.shape[0], X_test.shape[1]))
Тестирование - строки: 10 000, столбцы: 784
```

На следующих далее страницах мы пошагово пройдемся по примерам программирования с использованием библиотеки Keras, которые можно запускать прямо из своего интерпретатора Python. Однако если вы интересуетесь тренировкой нейронной сети на своем ГП, то можете поместить его в сценарий Python либо скачать соответствующий код с веб-сайта Packt Publishing. Для того чтобы выполнить сценарий Python на ГП, следует из каталога, где расположен файл mnist_keras_mlp.py, выполнить следующую команду:

```
THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python mnist_keras_mlp.py
```

Чтобы продолжить подготовку тренировочных данных, приведем массив изображений MNIST в 32-разрядный формат:

```
import theano
theano.config.floatX = 'float32'
X_train = X_train.astype(theano.config.floatX)
X_test = X_test.astype(theano.config.floatX)
```

Далее нам нужно преобразовать метки классов (целые числа 0–9) в формат прямого кода. К счастью, библиотека Keras обеспечивает для этого удобный инструмент:

```
>>>
from keras.utils import np_utils
print('Первые 3 метки: ', y_train[:3])
Первые 3 метки: [5 0 4]

y_train_ohe = np_utils.to_categorical(y_train)
print('\nПервые 3 метки (прямой код):\n', y_train_ohe[:3])
Первые 3 метки (прямой код):
[[ 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [ 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

Теперь можно приступить к самой интересной части и реализовать нейронную сеть. Здесь мы будем использовать ту же самую архитектуру, что и в *главе 12 «Тренировка искусственных нейронных сетей для распознавания изображений»*. Однако мы заменим логистические узлы в скрытом слое на функции активации в виде гиперболического тангенса, заменим логистическую функцию в выходном слое на функцию softmax и добавим дополнительный скрытый слой. Как видно из приведенной ниже программной реализации, библиотека Keras очень упрощает эти задачи:

```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD

np.random.seed(1)

model = Sequential()
model.add(Dense(input_dim=X_train.shape[1],
                output_dim=50,
                init='uniform',
                activation='tanh'))

model.add(Dense(input_dim=50,
                output_dim=50,
                init='uniform',
                activation='tanh'))

model.add(Dense(input_dim=50,
                output_dim=y_train_ohe.shape[1],
                init='uniform',
                activation='softmax'))

sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)
model.compile(loss='categorical_crossentropy', optimizer=sgd)
```

Сначала при помощи класса Sequential мы инициализируем новую модель для реализации нейронной сети с прямым распространением сигналов. Затем мы можем добавить в нее столько слоев, сколько нам нравится. Однако, учитывая, что первый добавленный нами слой является входным, мы должны удостовериться, что атрибут input_dim соответствует числу признаков (столбцов) в тренировочном наборе (в данном случае 768). Кроме того, мы должны удостовериться, что число выходных узлов (output_dim) и число входных узлов (input_dim) двух последовательных слоев соответствуют друг другу. В предыдущем примере мы добавили два скрытых слоя с 50 скрытыми узлами плюс 1 узел смещения каждый. Отметим, что в библиотеке Keras узел смещения в полносвязных сетях инициализируется нулем. Это контрастирует с реализацией MLP в *главе 12 «Тренировка искусственных нейронных сетей для распознавания изображений»*, где мы инициализировали узлы смещения единицами, что является общепринятым (и не обязательно лучшим) соглашением.

Наконец, число узлов в выходном слое должно быть равно числу уникальных меток классов – числу столбцов в массиве меток классов в прямой кодировке. Прежде чем мы сможем скомпилировать нашу модель, мы также должны задать оптимизатор. В предыдущем примере мы выбрали оптимизацию на основе стохастического градиентного спуска, с которой мы уже знакомы из предыдущих глав. Кроме того, мы можем установить значения для константы снижения весов и импульса обучения, чтобы скорректировать темп обучения в каждой эпохе, как обсуждалось в *главе 12 «Тренировка искусственных нейронных сетей для распознавания изображений»*. Наконец, мы устанавливаем функцию стоимости (или потерь) в categorical_crossentropy. (Бинарная) перекрестная энтропия – это просто технический термин для функции стоимости в логистической регрессии, и категориальная перекрестная энтропия – это ее обобщение для многоклассовых прогнозов с использованием функции softmax. После компиляции модели теперь можно натренировать ее путем вызова метода fit. Здесь мы используем мини-пакетный стохастический градиент с размером пакета 300 тренировочных образцов из расчета на пакет. Мы тренируем MLP на 50 эпохах и можем проследить за оптимизацией функции стоимости во время тренировки, установив verbose=1. Параметр validation_split особенно удобен, поскольку он резервирует 10% тренировочных данных (в данном случае 6000 образцов) для проверки после каждой эпохи, с тем чтобы во время тренировки мы могли удостовериться, что модель не находится в переподогнанном состоянии.

```
>>>
model.fit(X_train,
          y_train_ohe,
          nb_epoch=50,
          batch_size=300,
          verbose=1,
          validation_split=0.1,
          show_accuracy=True)

Train on 54000 samples, validate on 6000 samples
Epoch 0
54000/54000 [==============================] - 1s - loss: 2.2290 -
acc: 0.3592 - val_loss: 2.1094 - val_acc: 0.5342
```

```
Epoch 1
54000/54000 [==============================] - 1s - loss: 1.8850 -
acc: 0.5279 - val_loss: 1.6098 - val_acc: 0.5617
Epoch 2
54000/54000 [==============================] - 1s - loss: 1.3903 -
acc: 0.5884 - val_loss: 1.1666 - val_acc: 0.6707
Epoch 3
54000/54000 [==============================] - 1s - loss: 1.0592 -
acc: 0.6936 - val_loss: 0.8961 - val_acc: 0.7615
[…]
Epoch 49
54000/54000 [==============================] - 1s - loss: 0.1907 -
acc: 0.9432 - val_loss: 0.1749 - val_acc: 0.9482
```

Распечатка значения функции стоимости чрезвычайно полезна во время тренировки, поскольку мы можем быстро установить, уменьшается ли стоимость во время тренировки, и остановить алгоритм раньше, чтобы настроить значения гиперпараметров.

Чтобы предсказать метки классов, мы затем можем использовать метод predict_classes для возврата меток классов непосредственно в виде целых чисел:

```
>>>
y_train_pred = model.predict_classes(X_train, verbose=0)
print('Первые 3 предсказания: ', y_train_pred[:3])

Первые 3 предсказания: [5 0 4]
```

В заключение распечатаем верность модели на тренировочном и тестовом наборах:

```
train_acc = np.sum(
        y_train == y_train_pred, axis=0) / X_train.shape[0]

print('Верность на тренировочном наборе: %.2f%%' % (train_acc * 100))
Верность на тренировочном наборе: 94.51%

y_test_pred = model.predict_classes(X_test, verbose=0)
test_acc = np.sum(y_test == y_test_pred,
                  axis=0) / X_test.shape[0]
print('Верность на тестовом наборе: %.2f%%' % (test_acc * 100))
Верность на тестовом наборе: 94.39%
```

Отметим, что это всего лишь очень простая нейронная сеть без оптимизированных настроечных параметров. Если вам интересно поэкспериментировать с Keras дальше, то вы можете свободно попробовать поменять темп обучения, импульс, снижение весов и число скрытых узлов.

Несмотря на то что Keras – великолепная библиотека для реализации и экспериментирования с нейронными сетями, существует много других оберточных библиотек вокруг Theano, о которых стоит упомянуть. Видным примером является библиотека Pylearn2 (http://deeplearning.net/software/pylearn2/), которая была разработана в лаборатории LISA в Монреале. Также вас может заинтересовать библиотека Lasagne (https://github.com/Lasagne/Lasagne), в случае если вы предпочитаете более минималистскую библиотеку расширения, предлагающую больше контроля над базовым исходным кодом Theano.

Резюме

Я надеюсь, что вы получили удовольствие от этой последней главы захватывающего путешествия по машинному обучению. В этой книге мы затронули все существенные темы, которые эта область может предложить, и теперь вы должны быть хорошо подготовлены для приведения этих методов в действие для решения реальных задач.

Мы начали наше путешествие с краткого обзора разных типов обучения выполнять задачи: обучение с учителем, обучение с подкреплением и обучение без учителя. Мы обсудили несколько различных алгоритмов обучения, которые могут использоваться для классификации данных, начиная с простых однослойных нейронных сетей в *главе 2 «Тренировка алгоритмов машинного обучения для задачи классификации»*. Затем мы обсудили более усовершенствованные алгоритмы классификации в *главе 3 «Обзор классификаторов с использованием библиотеки scikit-learn»*, а в *главе 4 «Создание хороших тренировочных наборов – предобработка данных»* и *главе 5 «Сжатие данных путем снижения размерности»* вы узнали о самых важных аспектах конвейера машинного обучения. Напомним, что даже самый продвинутый алгоритм ограничен информацией в тренировочных данных, на которых он обучается. В *главе 6 «Анализ наиболее успешных приемов оценивания моделей и тонкой настройки гиперпараметров»* вы изучили наиболее успешные практические методы построения и оценивания прогнозных моделей, т. е. еще одного важного аспекта в приложениях с использованием машинного обучения. Если один-единственный алгоритм обучения не достигает необходимого нам качества, то для выполнения прогноза иногда целесообразно создать ансамбль экспертов. Мы обсудили это в *главе 7 «Объединение моделей для методов ансамблевого обучения»*. В *главе 8 «Применение алгоритмов машинного обучения в анализе мнений»* мы применили машинное обучение для анализа, вероятно, самой интересной формы данных в наше время, которая занимает доминирующее положение в платформах социальных медиа в Интернете: текстовых документов. Однако машинные методы обучения не ограничиваются офлайновым анализом данных, и в *главе 9 «Встраивание алгоритма машинного обучения в веб-приложение»* мы увидели, как встраивать машиннообучаемую модель в веб-приложение, с тем чтобы поделиться им с внешним миром. По большей части наше внимание концентрировалось на алгоритмах для классификации данных, вероятно, наиболее популярном приложении методов машинного обучения. Однако на этом они не заканчиваются! В *главе 10 «Предсказание непрерывных целевых величин при помощи регрессионного анализа»* мы разобрали несколько алгоритмов для задач регрессионного анализа с предсказанием выходных значений непрерывной целевой переменной. Еще одной захватывающей подобластью машинного обучения является кластерный анализ, который способен помочь найти в данных скрытые структуры, даже если наши тренировочные данные не содержат правильных ответов, на которых можно учиться. Мы обсудили это в *главе 11 «Работа с немаркированными данными – кластерный анализ»*.

В последних двух главах этой книги мы мельком просмотрели самые красивые и самые захватывающие алгоритмы во всей области машинного обучения: искусственные нейронные сети. Несмотря на то что глубокое обучение на самом деле выходит за рамки этой книги, я надеюсь, что смог, по крайней мере, разжечь в вас интерес настолько, чтобы побудить следить за новыми успехами в этой области. Если вы рассматриваете карьеру исследователя в области машинного обучения или же

просто желаете поддерживать свою осведомленность в этой области на современном уровне, то я могу рекомендовать вам проследить за работами ведущих экспертов в этой области, таких как Джефф Хинтон (Geoff Hinton, http://www.cs.toronto.edu/~hinton/), Эндрю Нг (Andrew Ng, http://www.andrewng.org), Янн ЛеКун (Yann LeCun, http://yann.lecun.com), Юрген Шмидтхубер (Juergen Schmidhuber, http://people.idsia.ch/~juergen/) и Йошуа Бенджо (Yoshua Bengio, http://www.iro.umontreal.ca/~bengioy), и это только некоторые из них. Кроме того, присоединяйтесь к списку рассылки библиотек scikit-learn, Theano и Keras, чтобы участвовать в интересных дискуссиях по поводу этих библиотек и машинного обучения в целом. Надеюсь встретить вас там! Вы всегда можете связаться со мной, если у вас имеются какие-либо вопросы по этой книге или вы нуждаетесь в некоторых общих советах по поводу машинного обучения.

Надеюсь, что это путешествие по различным аспектам машинного обучения действительно стоило того, и вы освоили много новых и полезных навыков, чтобы продвинуться по своей карьере и применять их к решению реальных задач.

Приложение *A*

Данное приложение составлено по материалам раздела часто задаваемых вопросов, касающихся книги и машинного обучения в целом (https://github.com/rasbt/python-machine-learning-book/tree/master/faq).

Оценка моделей

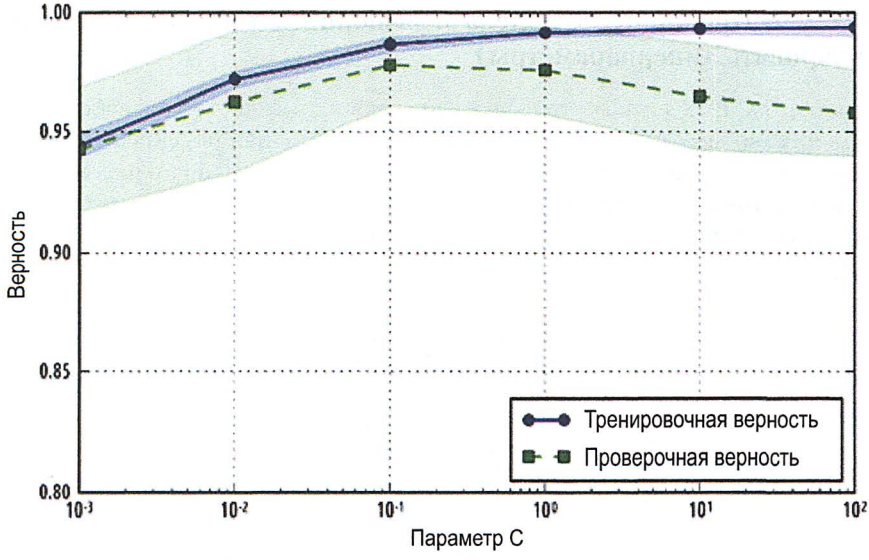
Что такое переобучение?

Пусть имеется гипотеза или модель m, которую мы адаптируем на наших тренировочных данных. В машинном обучении во время тренировки модели мы измеряем и оптимизируем качество модели – например, показатель верности (accuracy). Назовем его верностью на тренировочных данных $ACC_{train}(m)$.

С другой стороны, в машинном обучении мы на самом деле заинтересованы в построении модели, которая хорошо обобщается на ранее не встречавшиеся данные, т. е. мы хотим построить модель, которая имеет высокую верность на всем распределении данных; назовем ее генеральной верностью $ACC_{population}(m)$. При этом, чтобы оценить обобщающую способность, как правило, мы применяем методы перекрестной проверки и раздельный независимый тестовый набор данных.

Переобучение выполняется, если имеется альтернативная модель m' из пространста гипотез алгоритма, где точность на тренировочном наборе лучше и обобщающая способность хуже, по сравнению с моделью m, – мы говорим, что m переобучена (излишне адаптирована или аппроксимирована) под тренировочные данные.

Исходя из эмпирического опыта, модель, скорее всего, переобучена, если она слишком сложна при наличии фиксированного числа тренировочных образцов. Приведенный ниже рисунок показывает тренировочную и перекрестно-проверочную кривые модели SVM на некотором наборе данных. Здесь верность показана как функция обратного параметра регуляризации C – чем больше значение параметра C, тем крупнее штраф за сложность.



Мы замечаем более крупную разницу между верностью на тренировочном и тестовом наборах для увеличивающихся значений C (более сложных моделей). Основываясь на графике, можно сказать, что модель в диапазоне $< 10^{-1}$ недостаточно адаптирована к тренировочным данным (недообучена), тогда как в $> 10^{-1}$ модель излишне к ним адаптирована (переобучена).

Способы устранения переобучения включают в себя:

1) выбор более простой модели путем добавления смещения и/или сокращения числа параметров:
   а) добавление регуляризационных штрафов;
   б) снижение размерности пространства признаков;
2) сбор дополнительных тренировочных данных.

Как оценивать модель?

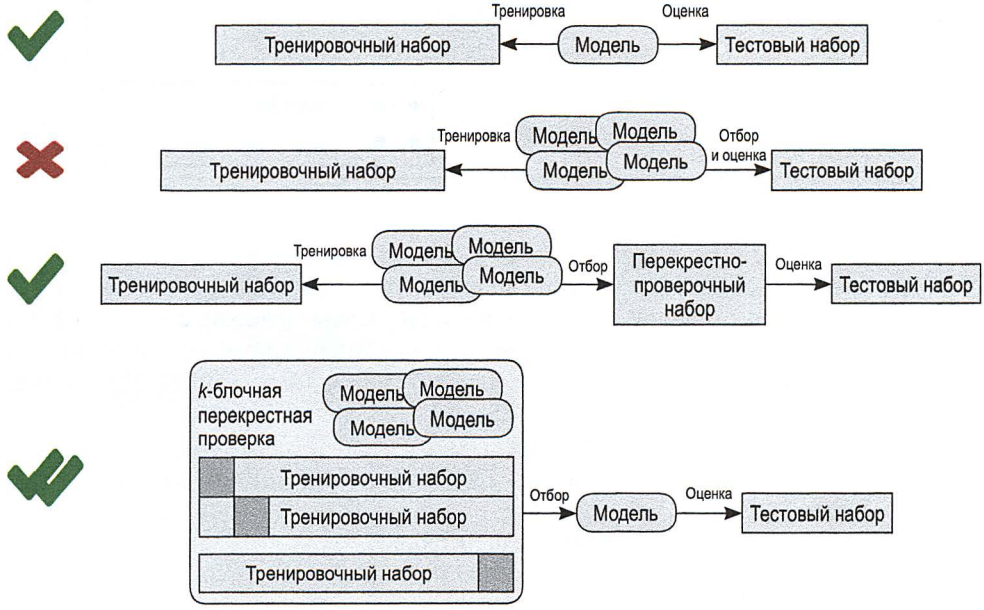
Если коротко, то следует сохранять независимый тестовый набор для окончательной модели – это должны быть данные, которые ваша модель ранее не видела. Впрочем, все зависит от ваших задач и принятого подхода.

Сценарий 1. Элементарно обучить простую модель

Разбить набор данных на раздельные тренировочный и тестовый наборы. Натренировать модель на первом, оценить модель на втором. Под глаголом «оценить» подразумевается вычисление метрик качества работы, таких как ошибка, точность (precision), полнота, площадь под ROC-кривой (ROC AUC) и пр.

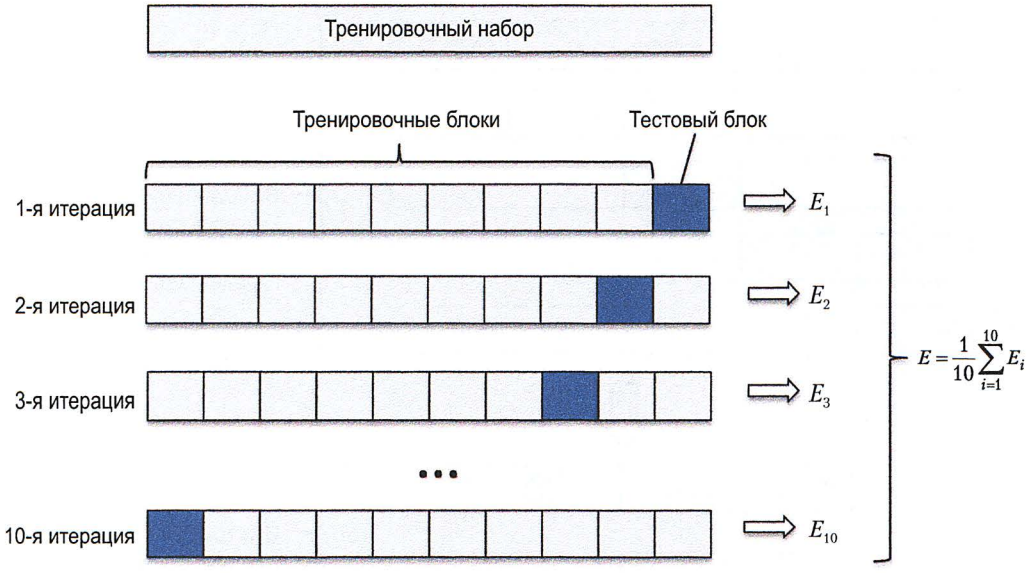
Сценарий 2. Натренировать модель и выполнить тонкую настройку (оптимизировать гиперпараметры)

Разбить набор данных на раздельные тренировочный и тестовый наборы. Применить на тренировочном наборе такие методы, как k-блочная перекрестная проверка, чтобы отыскать для вашей модели «оптимальный» набор гиперпараметров. Когда вы закончите с тонкой настройкой гиперпараметров, используйте тестовый набор, чтобы получить несмещенную оценку ее качества. На приведенном ниже рисунке иллюстрируется разница:



Перый ряд обозначает сценарий 1; третий ряд описывает более «классический» подход, когда вы далее подразделяете ваши тренировочные данные на тренировочное подмножество и перекрестно-проверочный набор. Затем вы тренируете вашу модель на тренировочном подмножестве и оцениваете модель на перекрестно-проверочном наборе, например чтобы оптимизировать ее гиперпараметры. В заключение вы тестируете ее на независимом тестовом наборе. Четвертый ряд описывает «наилучший» (более несмещенный) подход с использованием k-блочной перекрестной проверки, как описано в сценарии 2.

В качестве напоминания покажем общую схему k-блочной перекрестной проверки, в случае если вы с ней еще незнакомы:

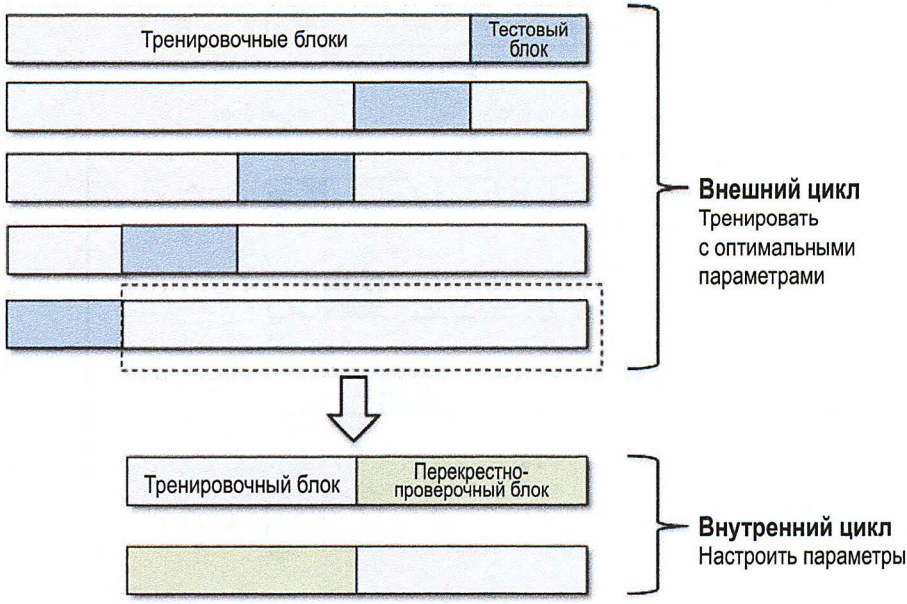


Здесь E – это ошибка прогнозирования, но ее можно поменять на точность, полноту, оценку F1, оценку площади под ROC-кривой (ROC AUC) либо любую другую метрику, которую вы предпочитаете использовать для решаемой задачи.

Сценарий 3. Построить разные модели и сравнить разные алгоритмы (например, SVM против логистической регрессии против случайных лесов и т. д.)

Здесь нам потребуется использование вложенной перекрестной проверки. Во вложенной перекрестной проверке мы имеем внешний цикл k-блочной перекрестной проверки, который разбивает данные на тренировочные и тестовый блоки, и внутренний цикл, который используется для отбора модели методом k-блочной перекрестной проверки на тренировочном блоке. После отбора модели затем для оценки ее качества используется тестовый блок. После того как мы идентифицировали наш «предпочтительный» алгоритм, мы можем продолжить с «регулярным» подходом на основе k-блочной перекрестной проверки (на полном тренировочном наборе), с тем чтобы найти его «оптимальные» гиперпараметры и оценить его на независимом тестовом наборе.

Чтобы пояснить, рассмотрим логистическую регрессионную модель. Используя вложенную перекрестную проверку, вы натренируете m разных логистических регрессионных моделей, одну для каждого из m внешних блоков, при этом внутренние блоки используются для оптимизации гиперпараметров каждой модели (например, с использованием поиска по сетке параметров в сочетании с k-блочной перекрестной проверкой. Если ваша модель стабильна, то все эти m моделей должны иметь одинаковые значения гиперпараметров, и вы сообщаете о среднем качестве работы модели, основываясь на внешних тестовых блоках. Затем вы переходите к следующему алгоритму, например SVM, и т. д.



Перекрестная проверка. Оценка качества оценщика

Извлечение параметров функции предсказания и их тестирование на тех же самых данных являются методологической ошибкой: модель, которой повторно предъявляют метки образцов, которые она только что видела, будет иметь идеальную оценку, но ей не удастся предсказать что-то полезное на ранее не встречавшихся ей данных. Такая ситуация называется переобучением. Чтобы избежать ее, во время выполнения эксперимента в области машинного обучения с учителем широко распространена практика, которая состоит в том, чтобы отложить часть имеющихся данных в качестве тестового набора X_test, y_test.

В библиотеке scikit-learn случайная разбивка на тренировочный и тестовый наборы быстро вычисляется при помощи вспомогательной функции train_test_split. Загрузим набор данных Iris, чтобы выполнить на нем подгонку линейного метода опорных векторов (SVM):

```
>>>
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import datasets
from sklearn import svm

iris = datasets.load_iris()
iris.data.shape, iris.target.shape
((150, 4), (150,))
```

Теперь можно быстро извлечь тренировочный набор, отложив 40% данных для тестирования (оценки) нашего классификатора:

```
>>>
X_train, X_test, y_train, y_test = train_test_split(
                                    iris.data, iris.target, test_size=0.4, random_state=0)
```

```
X_train.shape, y_train.shape
((90, 4), (90,))
X_test.shape, y_test.shape
((60, 4), (60,))

clf = svm.SVC(kernel='linear', C=1).fit(X_train, y_train)
clf.score(X_test, y_test)
0.96...
```

Во время оценки различных конфигураций (или «гиперпараметров») для оценщиков, как, например, настройка параметра C, который для модели на основе SVM нужно устанавливать вручную, все еще сохраняется риск переобучения на тестовом наборе, т. к. параметры можно подправлять, пока оценщик не будет показывать оптимальное качество работы. Информация о тестовом наборе таким путем может «проникнуть» в модель, и оценочные метрики перестанут сообщать правильные сведения об обобщающей способности. Для того чтобы решить эту проблему, можно отложить еще одну часть набора данных в качестве так называемого «проверочного набора» (или валидационного): тренировка проходит на тренировочном наборе, после чего оценка выполняется на проверочном наборе, и когда эксперимент оказывается успешным, окончательная оценка выполняется на тестовом наборе.

Однако, подразделив имеющиеся данные на три набора, мы кардинально сокращаем число образцов, которые можно использовать для извлечения модели, и результаты могут зависеть от отдельно взятого случайного выбора пары (тренировочного, проверочного) наборов.

Решением этой проблемы является процедура, именуемая перекрестной проверкой (cross-validation, CV). Тестовый набор все еще должен быть отложен в сторону для окончательной оценки, но при проведении перекрестной проверки проверочный набор больше не нужен. В условиях базового подхода, именуемого k-блочной перекрестной проверкой, тренировочный набор разбивается на k более мелких наборов. Следующая ниже процедура выполняется для каждого из k «блоков»:

- модель тренируется с использованием $k - 1$ блоков в качестве тренировочных данных;
- результирующая модель проверяется на соответствие качественным характеристикам на оставшейся части данных (т. е. эта часть используется в качестве тестового набора, чтобы вычислить меру качества работы, такую как точность).

Мера оценки качества, полученная в результате k-блочной перекрестной проверки, представляет собой среднее арифметическое значений, вычисленных в цикле. Такой подход может оказаться вычислительно затратным, но не расходует слишком много данных (как это происходит во время фиксации произвольного тестового набора). В этом главное преимущество, в частности для таких задач, где число образцов очень мало.

Самый простой способ применения перекрестной проверки состоит в вызове вспомогательной функции `cross_val_score` на оценщике и наборе данных.

Следующий ниже пример демонстрирует, как оценивать точность линейного метода опорных векторов на наборе данных Iris, разбив данные, выполнив подгонку модели и вычислив оценку 5 раз подряд (каждый раз с разными разбивками).

```
>>>
from sklearn.model_selection import cross_val_score
clf = svm.SVC(kernel='linear', C=1)
scores = cross_val_score(clf, iris.data, iris.target, cv=5)
scores
array([ 0.96...,  1.  ...,  0.96...,  0.96...,  1.        ])
```

Отсюда выводится средняя оценка и 95%-ный доверительный интервал оценки:

```
>>>
print("Точность: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
Точность: 0.98 (+/- 0.03)
```

По умолчанию оценка, вычисляемая на каждой итерации перекрестной проверки, – это оценочный метод оценщика. Его можно поменять, воспользовавшись параметром scoring:

```
>>>
from sklearn import metrics
scores = cross_val_score(
            clf, iris.data, iris.target, cv=5, scoring='f1_macro')
scores
array([ 0.96...,  1.  ...,  0.96...,  0.96...,  1.        ])
```

В случае с набором данных Iris образцы сбалансированы по всем целевым классам, отсюда точность и оценка F1 почти одинаковые.

Когда аргумент cv является целочисленным, функция cross_val_score по умолчанию использует стратегии KFold либо StratifiedKFold, причем последняя используется, если оценщик является производным от ClassifierMixin.

Кроме того, имеется возможность применить другие стратегии перекрестной проверки, передав вместо этого итератор перекрестной проверки, например:

```
>>>
from sklearn.model_selection import ShuffleSplit
n_samples = iris.data.shape[0]
cv = ShuffleSplit(n_splits=3, test_size=0.3, random_state=0)
cross_val_score(clf, iris.data, iris.target, cv=cv)

array([ 0.97...,  0.97...,  1.        ])
```

Перекрестная проверка с исключением по одному

Перекрестная проверка с исключением по одному LeaveOneOut (LOO) – это простая перекрестная проверка, где каждый обучающий набор создается с использованием всех образцов, за исключением одного, и отложенный образец является тестовым набором. Так, для n образцов будет иметься n разных тренировочных и n разных тестовых наборов. Эта процедура перекрестной проверки не тратит слишком много данных, т. к. из тренировочного набора удаляется всего один образец:

```
>>>
from sklearn.model_selection import LeaveOneOut

X = [1, 2, 3, 4]
loo = LeaveOneOut()
for train, test in loo.split(X):
    print("%s %s" % (train, test))
[1 2 3] [0]
[0 2 3] [1]
[0 1 3] [2]
[0 1 2] [3]
```

Пример стратифицированной k-блочной перекрестной проверки

Стратифицированная StratifiedKFold (с разбивкой по классам) – это вариант *k*-блочной перекрестной проверки, которая возвращает стратифицированные блоки: каждый набор содержит приблизительно тот же процент образцов каждого целевого класса, что и полный набор.

Пример стратифицированной 3-блочной перекрестной проверки на наборе данных из 10 образцов из двух слегка разбалансированных классов:

```
>>>
from sklearn.model_selection import StratifiedKFold

X = np.ones(10)
y = [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
skf = StratifiedKFold(n_splits=3)
for train, test in skf.split(X, y):
    print("%s %s" % (train, test))
[2 3 6 7 8 9] [0 1 4 5]
[0 1 3 4 5 8 9] [2 6 7]
[0 1 2 4 5 6 7] [3 8 9]
```

Расширенный пример вложенной перекрестной проверки

Подробное объяснение вложенной перекрестной проверки приведено в:

- главе 6, раздел «*Отбор алгоритма методом вложенной перекрестной проверки*» (открыть пример исходного кода через nbviewer – http://nbviewer.jupyter.org/github/rasbt/python-machine-learning-book/blob/master/code/ch06/ch06.ipynb#Algorithm-selection-with-nested-cross-validation);
- разделе часто задаваемых вопросов (FAQ), вопрос «Как оценивать модель?» (https://github.com/rasbt/python-machine-learning-book/blob/master/faq/evaluate-a-model.md) подраздела «Оценка моделей».

Подготовка набора данных и конфигурирование оценщика:

```
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.datasets import load_iris

# загрузить и разделить данные
iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

# настроить конвейер
cls = SVC(C=10.0, kernel='rbf', gamma=0.1, decision_function_shape='ovr')
kernel_svm = Pipeline([('std', StandardScaler()),
                       ('svc', cls)])

# настроить сеточный поиск
param_grid = [
  {'svc__C': [1, 10, 100, 1000],
   'svc__gamma': [0.001, 0.0001],
   'svc__kernel': ['rbf']},
 ]
```

```
# настроить несколько объектов GridSearchCV, 1 для каждого алгоритма

gs_svm = GridSearchCV(estimator=kernel_svm,
                      param_grid=param_grid,
                      scoring='accuracy',
                      n_jobs=-1,
                      cv=5,
                      verbose=0,
                      refit=True,
                      pre_dispatch='2*n_jobs')
```

А. Вложенная кросс-валидация: быстрая версия

Здесь функция cross_val_score выполяет 5 внешних циклов, и объект GridSearch (gs) выполняет оптимизацию гиперпараметров во время 5 внутренних циклов.

```
>>>
import numpy as np

from sklearn.cross_validation import cross_val_score
scores = cross_val_score(gs_svm, X_train, y_train, scoring='accuracy', cv=5)
print('\nСредняя точность %.2f +/- %.2f' % (np.mean(scores), np.std(scores)))
Средняя точность 0.95 +/- 0.06
```

Б. Вложенная кросс-валидация: ручной подход с распечаткой модельных параметров

```
>>>
from sklearn.cross_validation import StratifiedKFold
from sklearn.metrics import accuracy_score
import numpy as np

params = []
scores = []

skfold = StratifiedKFold(y=y_train, n_folds=5, shuffle=False, random_state=1)
for train_idx, test_idx in skfold:
    gs_svm.fit(X_train[train_idx], y_train[train_idx])
    y_pred = gs_svm.predict(X_train[test_idx])
    acc = accuracy_score(y_true=y_train[test_idx], y_pred=y_pred)
    params.append(gs_svm.best_params_)
    scores.append(acc)

print('Модели SVM:')
for idx, m in enumerate(zip(params, scores)):
    print('%s. Точность: %.2f, параметры: %s' % (idx+1, m[1], m[0]))
print('\nСредняя точность %.2f +/- %.2f' % (np.mean(scores), np.std(scores)))

Модели SVM:
1. Точность:0.96, параметры:{<svc__C>:100,>svc__kernel>:>rbf>,>svc__gamma>: 0.001}
2. Точность:1.00, параметры:{<svc__C>:100,>svc__kernel>:>rbf>,>svc__gamma>: 0.001}
3. Точность:0.83, параметры:{<svc__C>:1000,>svc__kernel>:>rbf>,>svc__gamma>: 0.001}
4. Точность:1.00, параметры:{<svc__C>:100,>svc__kernel>:>rbf>,>svc__gamma>: 0.001}
5. Точность:0.96, параметры:{<svc__C>:100,>svc__kernel>:>rbf>,>svc__gamma>: 0.001}

Средняя точность 0.95 +/- 0.06
```

B. Регулярная k-блочная кросс-валидация для оптимизации модели на полном наборе тренировочных данных

Повторить вложенную кросс-валидацию для разных алгоритмов. Затем выбрать «наилучший» алгоритм (не лучшую модель!). Далее использовать полный набор тренировочных данных, чтобы тонко настроить наилучший алгоритм методом сеточного поиска (поиска по стеку параметров):

```
>>>
gs_svm.fit(X_train, y_train)
print('Наилучшие параметры %s' % gs_svm.best_params_)
Наилучшие параметры {<svc__C>: 100, <svc__kernel>: <rbf>, <svc__gamma>: 0.001}

train_acc = accuracy_score(y_true=y_train, y_pred=gs_svm.predict(X_train))
test_acc = accuracy_score(y_true=y_test, y_pred=gs_svm.predict(X_test))
print('Точность на тренировочном наборе: %.2f' % train_acc)
print('Точность на тестовом наборе %.2f' % test_acc)
print('Параметры: %s' % gs_svm.best_params_)
Точность на тренировочном наборе: 0.97
Точность на тестовом наборе: 0.97
Параметры: {<svc__C': 100, 'svc__kernel': 'rbf', 'svc__gamma': 0.001}
```

График проверочной (валидационной) кривой

Чтобы выполнить проверку качества модели, нам нужна оценочная функция (количественная оценка качества предсказаний), например точность/правильность для классификаторов. Надлежащий способ выбора гиперпараметров оценщика – это, разумеется, сеточный поиск либо подобные методы тонкой настройки гиперпараметров, которые отбирают гиперпараметр с максимальной оценкой на одном либо двух и более проверочных наборах. Отметим, что если мы оптимизировали гиперпараметры, основываясь на проверочной (валидационной) оценке, то проверочная оценка смещена и больше не является хорошим оценочным показателем качества обобщения. Чтобы получить надлежащий показатель качества обобщения, мы должны вычислить оценку на еще одном тестовом наборе.

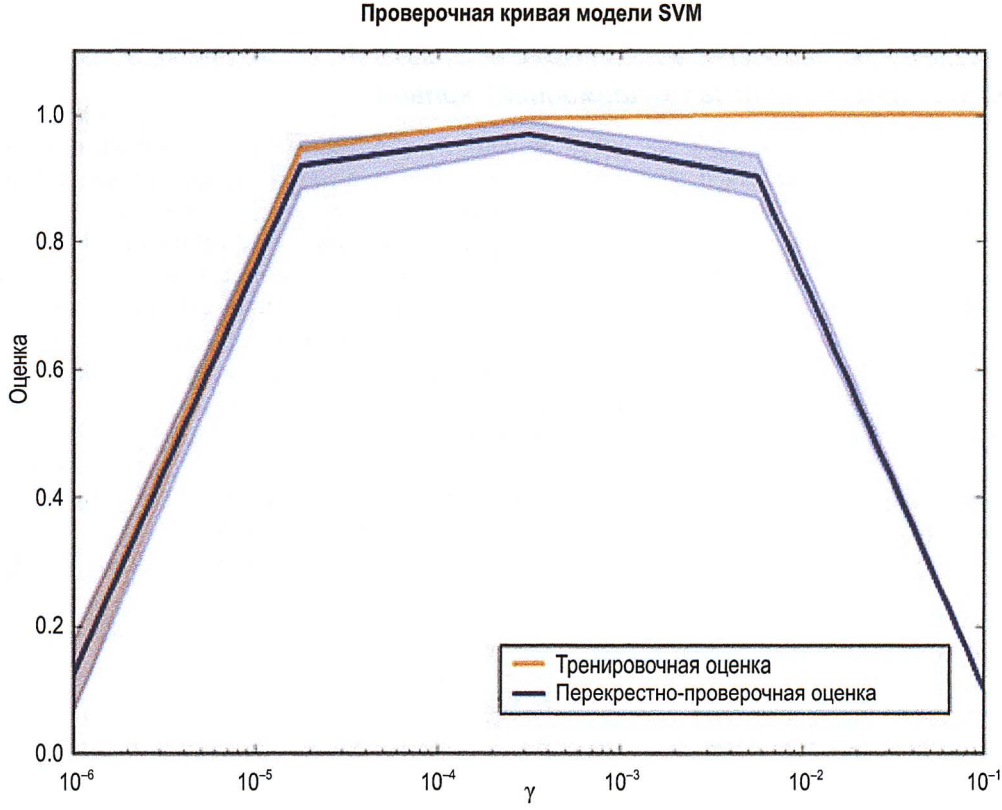
Однако иногда, чтобы выяснить, переобучен или недообучен оценщик для нескольких значений гиперпараметров, помогает построение графика влияния одиночного гиперпараметра на тренировочную и проверочную оценки.

В этом способна помочь функция перекрестно-проверочной кривой validation_curve:

```
>>>
import numpy as np
from sklearn.model_selection import validation_curve
from sklearn.datasets import load_iris
from sklearn.linear_model import Ridge

np.random.seed(0)
iris = load_iris()
X, y = iris.data, iris.target
indices = np.arange(y.shape[0])
np.random.shuffle(indices)
X, y = X[indices], y[indices]
```

```
train_scores, valid_scores = validation_curve(Ridge(), X, y, "alpha",
                                              np.logspace(-7, 3, 3))
train_scores
array([[ 0.94...,  0.92...,  0.92...],
       [ 0.94...,  0.92...,  0.92...],
       [ 0.47...,  0.45...,  0.42...]])

valid_scores
array([[ 0.90...,  0.92...,  0.94...],
       [ 0.90...,  0.92...,  0.94...],
       [ 0.44...,  0.39...,  0.45...]])
```

Если и тренировочная, и проверочная оценки низкие, то оценщик будет недоподогнан. Если тренировочная оценка высокая, а проверочная оценка низкая, то оценщик переобучен, в противном случае он работает очень хорошо. Низкая тренировочная оценка и высокая проверочная оценка обычно не возможны. Все эти случаи можно найти на графике проверочной кривой, приведенном ниже, где мы варьируем параметр γ модели SVM на наборе данных digits.



Настройка типового конвейера и сеточного поиска

Выполним настройку конвейера и поиска по сетке параметров:

```
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.datasets import load_iris

# загрузить и разделить данные
iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

# настроить конвейер
cls = SVC(C=10.0,
          kernel='rbf',
          gamma=0.1,
          decision_function_shape='ovr')

kernel_svm = Pipeline([('std', StandardScaler()),
                       ('svc', cls)])

# настроить сеточный поиск
param_grid = [
  {'svc__C': [1, 10, 100, 1000],
   'svc__gamma': [0.001, 0.0001],
   'svc__kernel': ['rbf']},
 ]

gs = GridSearchCV(estimator=kernel_svm,
                  param_grid=param_grid,
                  scoring='accuracy',
                  n_jobs=-1,
                  cv=5,
                  verbose=1,
                  refit=True,
                  pre_dispatch='2*n_jobs')
```

Выполним сеточный поиск:

```
>>>
gs.fit(X_train, y_train)

Fitting 5 folds for each of 8 candidates, totalling 40 fits
GridSearchCV(cv=5, error_score='raise',
       estimator=Pipeline(steps=[('std', StandardScaler(copy=True, with_mean=True, with_
std=True)), ('svc', SVC(C=10.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False))]),
       fit_params={}, iid=True, n_jobs=-1,
       param_grid=[{'svc__C': [1, 10, 100, 1000], 'svc__kernel': ['rbf'], 'svc__gamma':
[0.001, 0.0001]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='accuracy', verbose=1)
```

Распечатаем результаты:

```
>>>
print('Лучшая оценка в результате сеточного поиска %.2f' % gs.best_score_)
print('Лучшие параметры в результате сеточного поиска %s' % gs.best_params_)

# предсказание на тренировочном наборе
y_pred = gs.predict(X_train)
train_acc = (y_train == y_pred).sum()/len(y_train)
print('\nТочность на тренировочном наборе: %.2f' % (train_acc))

# оценка на тестовом наборе
y_pred = gs.predict(X_test)
test_acc = (y_test == y_pred).sum()/len(y_test)
print('\nТочность на тестовом наборе: %.2f' % (test_acc))

Лучшая оценка в результате сеточного поиска 0.96
Лучшие параметры в результате сеточного поиска {‹svc__C›: 100, ‹svc__kernel›: ‹rbf›,
‹svc__gamma›: 0.001}
Точность на тренировочном наборе: 0.97
Точность на тестовом наборе: 0.97
```

По поводу атрибута best_score_ объекта GridSearchCV заметим, что gs.best_score_ – это средняя оценка на k-блочных перекрестно-проверочных данных. Иными словами, если мы имеем объект GridSearchCV с 5-блочными перекрестно-проверочными данными (как тот, который был приведен выше), атрибут best_score_ возвращает среднюю оценку по 5 блокам лучшей модели. Чтобы это проиллюстрировать, приведем пример:

```
>>>
from sklearn.cross_validation import StratifiedKFold, cross_val_score
from sklearn.linear_model import LogisticRegression
import numpy as np

np.random.seed(0)
np.set_printoptions(precision=6)
y = [np.random.randint(3) for i in range(25)]
X = (y + np.random.randn(25)).reshape(-1, 1)

cv5_idx = list(StratifiedKFold(y, n_folds=5, shuffle=False, random_state=0))
cross_val_score(LogisticRegression(random_state=123), X, y, cv=cv5_idx)

array([ 0.6,  0.4,  0.6,  0.2,  0.6])
```

Выполнив приведенный выше исходный код, мы создали простой набор данных из случайных целочисленных, которые должны представлять наши метки классов. Далее передадим индексы 5 перекрестно-проверочных блоков (cv3_idx) в регистратор оценок cross_val_score, который вернул 5 оценок точности, – это 5 значений точности для 5 тестовых блоков.

Далее воспользуемся объектом GridSearchCV и подадим ему те же 5 перекрестно-проверочных наборов (посредством предварительно сгенерированных индексов cv3_idx):

```
>>>
from sklearn.grid_search import GridSearchCV
gs = GridSearchCV(LogisticRegression(), {}, cv=cv5_idx, verbose=3).fit(X, y)
```

```
Fitting 5 folds for each of 1 candidates, totalling 5 fits
[CV] ..............................................................
[CV] ..................................... , score=0.600000 -   0.0s
[CV] ..............................................................
[CV] ..................................... , score=0.400000 -   0.0s
[CV] ..............................................................
[CV] ..................................... , score=0.600000 -   0.0s
[CV] ..............................................................
[CV] ..................................... , score=0.200000 -   0.0s
[CV] ..............................................................
[CV] ..................................... , score=0.600000 -   0.0s
```

Как видно, оценки для 5 блоков точно такие же, что и те, которые были из cross_val_score ранее.

Теперь атрибут bestscore объекта GridSearchCV, который становится доступным после подгонки методом fit, возвращает среднюю оценку точности лучшей модели:

```
>>>
gs.best_score_
0.48
```

Как видно, полученный выше результат согласуется со средней оценкой, вычисленной cross_val_score.

```
>>>
cross_val_score(LogisticRegression(), X, y, cv=cv5_idx).mean()
0.47999999999999998
```

Машинное обучение

В чем разница между классификатором и моделью?

В принципе, термины «классификатор» и «модель» в определенных контекстах синонимичны; однако иногда, говоря о «классификаторе», имеют в виду алгоритм обучения, который извлекает модель из тренировочных данных. Чтобы внести ясность, дадим определение некоторых ключевых терминов:

Тренировочный образец – это точка данных x в имеющемся наборе тренировочных данных, использующаяся для решения задачи прогнозного моделирования. Например, если мы заинтересованы в классифицировании сообщений электронной почты, то одно почтовое сообщение в нашем наборе данных будет одним тренировочным образцом. Иногда импользуются синонимичные термины «тренировочный экземпляр» или «пример».

Целевая функция: в прогнозном моделировании мы, как правило, заинтересованы в моделировании отдельно взятых процессов; мы хотим извлечь или аппроксимировать отдельно взятую функцию, которая, например, позволяет различать спамные почтовые сообщения от неспамных. Целевая функция $f(x) = y$ – это истинная фунция f, которую мы хотим смоделировать.

Гипотеза – это некая функция, которая, как мы считаем (или надеемся), похожа на истинную функцию, целевую функцию, которую мы хотим смоделировать. В контексте классификации спамных почтовых сообщений она будет правилом,

к которому мы придем и которое позволит отделять спамные почтовые сообщения от неспамных.

Модель: в машинном обучении термины гипотеза и модель часто используются взаимозаменяемо. В других науках они могут иметь иные значения, т. е. гипотеза будет «эмпирическим предположением», высказанным ученым, а модель – проявлением предположения, которую можно использовать для проверки гипотезы.

Алгоритм обучения: наша задача – найти или аппроксимировать целевую функцию, и алгорим обучения – это набор инструкций, которые пытаются смоделировать целевую функцию, используя наш тренировочный набор данных. Алгоритм обучения сопровождается пространством гипотез, множеством возможных гипотез, к которым он может прийти, для того чтобы смоделировать неизвестную целевую функцию, сформулировав окончательную гипотезу.

Классификатор – это частный случай гипотезы (в наши дни часто извлекаемой алгоритмом машинного обучения). Классификатор – это гипотеза или дискретнозначная функция, которая используется для назначения (категориальных) меток классов отдельно взятым точкам данных. В примере с классификацией почтовых сообщений этот классификатор мог быть гипотезой для маркировки почтовых сообщений как спам или неспам. Однако гипотеза не обязательно должна быть синонимичной классификатору. В другом приложении наша гипотеза может быть функцией, которая ставит время, проведенное за занятиями, и общеобразовательную подготовку студентов в соответствие будущим оценкам за SAT-экзамен.

Таким образом, можно сказать, что классификатор – это частный случай гипотезы или модели: классификатор – это функция, которая назначает метку класса точке данных.

В чем разница между функцией стоимости и функцией потерь?

Термины «функция стоимости» и «функция потерь» синонимичны (иногда их называют функцией ошибок). Более обший сценарий состоит сначала в определении целевой функции, которую мы хотим оптимизировать. Эта целевая функция может заключаться в:

- максимизации апостериорных вероятностей (например, в наивном байесовском классификаторе);
- максимизации функции приспособленности или пригодности (генетическое программирование);
- максимизации общего вознаграждения/функции ценности (обучение с подкреплением);
- максимизации прироста информации/минимизации неоднородностей дочерних узлов (CART-классификация путем построения дерева решений);
- минимизации функции стоимости (потерь) с использованием средневзвешенной квадратичной ошибки (CART-классификация и регрессия путем построения дерева решений, регрессия на основе деревьев решений, линейная регрессия, адаптивные линейные нейроны и пр.);
- максимизации функции логарифмического правдоподобия или минимизации функции потерь (стоимости) перекрестной энтропии;
- минимизации кусочно-линейной функции потерь (метода опорных векторов) и пр.

Обеспечение персистентности моделей scikit-learn на основе JSON

Во многих ситуациях желательно сохранить натренированную модель для использования в будущем. Ситуации, когда мы хотим зафиксировать модель, например, представлены развернутым в Интернете веб-приложением или научной репродуцируемостью наших экспериментов.

В главе 9 даны подробные описания сериализации моделей scikit-learn с использованием библиотеки pickle в контексте веб-приложения, которое мы разработали в главе 8. Кроме того, на веб-сайте scikit-learn имеется превосходное руководство (http://scikit-learn.org/stable/modules/model_persistence.html) по теме персистентности моделей. Честно говоря, консервация объектов Python посредством модулей pickle (https://docs.python.org/3.5/library/pickle.html), dill (https://pypi.python.org/pypi/dill) или joblib (https://pythonhosted.org/joblib/), вероятно, представляет самый удобный подход к персистентности моделей. Однако консервация объектов Python иногда может быть немного проблематичной, например десериализация модели в Python 3.x, которая была изначально законсервирована в Python 2.7x, и наоборот. К тому же библиотека pickle предлагает разные протоколы (действующий протокол 0–4), которые не обязательно обратно совместимы. Поэтому, чтобы быть готовыми к наихудшему сценарию – поврежденным файлам консервации pickle или несовместимости версий, есть, по меньшей мере, еще один (немногим более трудоемкий) способ обеспечения живучести моделей – использование JSON.

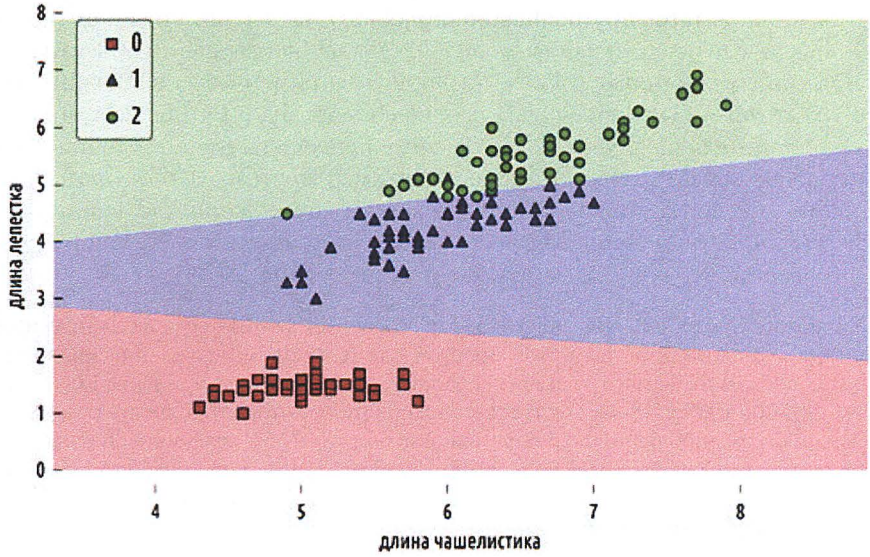
JSON (JavaScript Object Notation) – это легковесный формат обмена данными, легкий для чтения и написания людьми и простой для разбора и порождения машинами. Он основан на подмножестве языка программирования JavaScript стандарта ECMA-262, 3-е изд. – декабрь 1999 г. JSON – это текстовый формат, который полностью независим от языков программирования и одновременно использует обозначения, понятные для программистов семейства языков C, в том числе C, C++, C#, Java, JavaScript, Perl, Python и многих других. Эти особенности делают JSON идеальным языком обмена данными [источник: http://www.json.org].

Одно из преимуществ JSON состоит в том, что этот формат удобен для восприятия человеком. Поэтому когда дело доходит до крайностей, мы все равно способны прочитать файлы параметров и модельных коэффициентов «в ручном» режиме, присвоить эти значения соответствующему оценщику scikit-learn или построить нашу собственную модель, которая воспроизведет научные результаты.

Посмотрим, как это работает. Сначала натренируем простой логистический регрессионный классификатор на данных цветков ириса Iris:

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from mlxtend.plotting import plot_decision_regions

iris = load_iris()
y, X = iris.target, iris.data[:, [0, 2]]  # использовать всего 2 признака
lr = LogisticRegression(C=100.0,
                        class_weight=None,
                        dual=False,
                        fit_intercept=True,
                        intercept_scaling=1,
                        max_iter=100,
```

```
                        multi_class='multinomial',
                        n_jobs=1,
                        penalty='l2',
                        random_state=1,
                        solver='newton-cg',
                        tol=0.0001,
                        verbose=0,
                        warm_start=False)
lr.fit(X, y)
plot_decision_regions(X=X, y=y, clf=lr, legend=2)
plt.xlabel('длина чашелистика')
plt.ylabel('длина лепестка')
plt.show()
```



Удачным образом, в случае если мы хотим сохранить параметры оценщика в другом месте, нам не нужно их все перенабирать или копировать и вставлять вручную. Чтобы получить словарь этих параметров, можно просто применить вспомогательный метод get_params:

```
lr.get_params()
{'C': 100.0,
 'class_weight': None,
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'max_iter': 100,
 'multi_class': 'multinomial',
 'n_jobs': 1,
 'penalty': 'l2',
 'random_state': 1,
 'solver': 'newton-cg',
 'tol': 0.0001,
 'verbose': 0,
 'warm_start': False}
```

Сохранить их в формате JSON очень легко – из стандартной библиотеки Python мы просто импортируем модуль json и сбрасываем словарь в файл:

```
import json
path = './sckit-model-to-json/params.json'
with open(path, 'w', encoding='utf-8') as outfile:
   json.dump(lr.get_params(), outfile)
```

При чтении файла можно увидеть, что файл JSON – это просто копия один-в-один нашего словаря Python в текстовом формате:

```
>>>
path = './sckit-model-to-json/params.json'
with open(path, 'r', encoding='utf-8') as infile:
    print(infile.read())

{"n_jobs": 1, "random_state": 1, "intercept_scaling": 1, "penalty": "l2", "fit_intercept":
true, "solver": "newton-cg", "verbose": 0, "warm_start": false, "max_iter": 100, "tol":
0.0001, "dual": false, "C": 100.0, "class_weight": null, "multi_class": "multinomial"}
```

Теперь более подковыристая часть заключается в идентификации параметров «подгонки» оценщика, т. е. параметров нашей логистической регрессионной модели. Однако на практике на самом деле выяснить это достаточно просто, если перейти к соответствующей странице документации (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html): ищите атрибуты в разделе **Attribute**, в которых есть завершающий символ подчеркивания (спасибо команде scikit-learn за прекрасно продуманный API!). В случае с логистической регрессией нас интересуют веса .coef_, узел смещения (точка пересечения) .intercept_ и атрибуты classes_ и n_iter_.

```
>>>
attrs = [i for i in dir(lr) if i.endswith('_') and not i.endswith('__')]
print(attrs)
['classes_', 'coef_', 'intercept_', 'n_iter_']

attr_dict = {i: getattr(lr, i) for i in attrs}
```

Чтобы сериализовать массивы NumPy в объекты JSON, нам сначала нужно преобразовать массивы в (вложенные) списки Python. Однако благодаря методу tolist это не особо хлопотно (помимо этого, для ясности можно подумать о сохранении атрибутов в отдельных файлах JSON, например intercept.json и coef.json).

```
import numpy as np
for k in attr_dict:
    if isinstance(attr_dict[k], np.ndarray):
        attr_dict[k] = attr_dict[k].tolist()
```

Теперь мы готовы сбросить «словарь атрибутов» в файл JSON:

```
path = './sckit-model-to-json/attributes.json'
with open(path, 'w', encoding='utf-8') as outfile:
   json.dump(attr_dict,
             outfile,
             separators=(',', ':'),
             sort_keys=True,
             indent=4)
```

Если все прошло прекрасно, то наш файл JSON в простом текстовом формате должен выглядеть следующим образом:

```
>>>
path = './sckit-model-to-json/attributes.json'
with open(path, 'r', encoding='utf-8') as infile:
    print(infile.read())

{
    "classes_":[
        0,
        1,
        2
    ],
    "coef_":[
        [
            0.426252363761105,
            -8.557501546501133
        ],
        [
            1.564423133644856,
            -1.678365901997047
        ],
        [
            -1.9906754973974157,
            10.235867448237142
        ]
    ],
    "intercept_":[
        27.5333848531875,
        4.185099109107042,
        -31.718483962281997
    ],
    "n_iter_":[
        27
    ]
}
```

С такой же легкостью теперь можно воспользоваться методом loads библиотеки json, чтобы прочесть данные назад из файлов «.json» и заново присвоить их объектам Python (представим, что это происходит в другом сеансе Python).

```
import codecs
import json

path_param = './sckit-model-to-json/params.json'
path_attr = './sckit-model-to-json/attributes.json'
obj_text = codecs.open(path_param, 'r', encoding='utf-8').read()
params = json.loads(obj_text)
obj_text = codecs.open(path_attr, 'r', encoding='utf-8').read()
attributes = json.loads(obj_text)
```

В заключение нам просто нужно инициализировать используемый по умолчанию оценщик LogisticRegression, подать в него посредством метода set_params нужные параметры и переназначить другие атрибуты, воспользовавшись встроенным в Python методом setattr (правда, не забудьте заново преобразовать списки Python в массивы NumPy!):

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from mlxtend.plotting import plot_decision_regions
import numpy as np

iris = load_iris()
y, X = iris.target, iris.data[:, [0, 2]]  # использовать всего 2 признака
lr = LogisticRegression()

lr.set_params(**params)
for k in attributes:
    if isinstance(attributes[k], list):
        setattr(lr, k, np.array(attributes[k]))
    else:
        setattr(lr, k, attributes[k])

plot_decision_regions(X=X, y=y, clf=lr, legend=2)
plt.xlabel('длина чашелистика')
plt.ylabel('длина лепестка')
plt.show()
```

В результате выполнения вышеприведенного исходного кода мы получим точно такой же график, что и после тренировки простого логистического регрессионного классификатора на данных цветков ириса Iris ранее.

Глоссарий основных терминов и сокращений

Термины

ADALINE (adaptive linear element, адаптивный линейный элемент) – частный случай линейного классификатора или искусственной нейронной сети с одним слоем.

Активационная (передаточная) функция (activation function), или функция активации – функция линейного или нелинейного преобразования состояния нейрона. Эта функция преобразует сумму взвешенных входов нейрона в его выход.

Анализ мнений (sentiment analysis, opinion mining), или сентимент-анализ, анализ тональности текста – численный анализ мнений, настроений, субъективности, оценок, отношения, эмоций и т. д., которые выражены в текстовом виде.

Биноминальное распределение (binomial distribution) – распределение количества «успехов» в последовательности из n независимых случайных экспериментов, таких что вероятность «успеха» в каждом из них равна p.

Бустрап-выборка (bootstrap sample) – синтетический набор данных, полученный в результате генерирования повторных выборок (с возвратом) из исследуемой выборки, используемой в качестве «суррогатной генеральной совокупности», в целях аппроксимации выборочного распределения статистики (такой как среднее, медиана и др.).

Бутстрапирование (bootstrapping) – метод размножения выборок, когда из имеющихся данных генерируют новые случайные выборки размером n точек с возвратом, а затем вычисляют статистику (среднее, медиану и пр.) этих синтетических наборов данных.

Векторизация (vectorization) – автоматическое применение элементарной арифметической операции ко всем элементам в массиве.

Величина вектора (magnitude), или длина вектора – евклидов вектор представляет собой положение точки P в евклидовом пространстве. Геометрически его можно описать как стрелку, идущую из начала координат (хвост вектора) к этой точке (вершина вектора). Математически вектор **x** в N-мерном евклидовом пространстве можно определить как упорядоченный список из N вещественных чисел (прямоугольные координаты точки P): $X = [x_1, x_2, ..., x_n]$. Его величина или длина чаще всего определяется как его евклидова норма (или евклидова длина), т. е. квадратный корень из суммы квадратов элементов вектора.

Верность (accuracy) модели – это отношение количества правильно классифицированных исходов (истинно положительных и истинно отрицательных), к общему количеству исходов.

Выборка (sample) – это совокупность признаков, которым поставлены в соответствие конкретные значения.

Выборочное распределение (sampling distribution) – распределение вероятностей заданной статистики (например, выборочное распределение среднего), полученное с помощью большого числа образцов, отобранных из конкретной генеральной совокупности.

Гипотезы (hypotheses) – это предположения или теории, которые исследователь выдвигает относительно некоторых характеристик генеральной совокупности, подлежащей обследованию.

Градиентный (наискорейший) подъем (gradient ascent) – алгоритм градиентного подъема, инкрементный алгоритм оптимизации, или поиска оптимального решения, где приближение к локальному максимуму функции идет шагами, пропорциональными величине градиента этой функции в текущей точке.

Градиентный (наискорейший) спуск (gradient ascent) – алгоритм градиентного спуска, инкрементный алгоритм оптимизации, или поиска оптимального решения, где приближение к локальному минимуму функции идёт шагами, пропорциональными обратной величине градиента этой функции в текущей точке.

Граница решения (decision boundary), или поверхность решения – это гиперповерхность, которая разделяет базовое векторное пространство на два множества, одно для каждого класса; это часть пространства задачи, в которой выходная метка классификатора не определена.

Задача (problem) – проблемная ситуация с явно заданной целью, которую необходимо достичь; в более узком смысле задачей также называют саму эту цель, данную в рамках проблемной ситуации, то есть то, что требуется сделать.

Имитация отжига (simulated annealing) – общий алгоритмический метод аппроксимации глобального оптимума, похожий на градиентный спуск, но который, в отличие от него, за счет случайности выбора промежуточной точки реже попадает в локальные минимумы.

Импульс обучения (momentum learning) – это слагаемое, которое способствует продвижению в фиксированном направлении, поэтому если было сделано несколько шагов в одном и том же направлении, то алгоритм «увеличивает скорость», что (иногда) позволяет избежать локального минимума, а также быстрее проходить плоские участки.

Импутация (imputation) – процесс замещения пропущенных, некорректных или несостоятельных значений другими значениями.

Импутация простым средним (mean imputation) – замещение пропущенных данных средним арифметическим наблюдаемых значений данного признака.

Качество (performance) модели – измеренный количественно показатель качества работы модели, позволяющий судить о том, насколько результаты работы модели согласуются с тем, что ожидалось от нее получить.

Класс (class) – это множество всех объектов с данным значением метки.

Классификатор (classifier) – это функция, которая назначает метку класса точке данных; частный случай гипотезы и модели.

Конвейер (conveyor) – в информатике набор элементов обработки данных, соединенных последовательно, где выход одного элемента является входом следующего. Элементы конвейера часто выполняются параллельно по принципу квантования по времени; благодаря этому некоторое количество буферного хранилища часто вставляется между элементами.

Масштабирование признаков (feature scaling) – это метод, применяемый для стандартизации независимых переменных либо признаков данных. В сфере обра-

ботки данных этот метод известен как нормализация данных, которая обычно выполняется на этапе предобработки.

Матрично-векторное умножение (matrix-vector multiplication) – это последовательность вычисления скалярных произведений.

Мера неоднородности, или мера Джини (Gini impurity) – это вероятность, что с случайный образец будет классифицирован правильно, если произвольно выбрать метку согласно распределению в ветви дерева. Не следует путать с коэффициентом Джини в статистике, где он представляет собой показатель степени расслоения общества по отношению к какому-либо признаку.

Метка класса (class label) – выходная (независимая) переменная классификационной модели. Метка класса всегда является дискретной и принимает значения из некоторого ограниченного набора категорий – наименований классов. Например, если в задаче классификации требуется определить степень риска, связанного с выдачей кредита клиенту, то для метки класса могут быть определены 3 значения: «Высокий», «Средний» и «Низкий».

Метод эластичной сети (elastic net) – это метод регуляризованной регрессии, компенсирующий недостатки метода LASSO добавлением L1- и L2-штрафа..

Метрические данные (metric data) – измеряются по интервальной или относительной шкале.

Многомерное шкалирование (scaling) – метод анализа и визуализации данных с помощью расположения точек, соответствующих изучаемым (шкалируемым) объектам, в пространстве меньшей размерности, чем пространство признаков объектов. Точки размещаются так, чтобы попарные расстояния между ними в новом пространстве как можно меньше отличались от эмпирически измеренных расстояний в пространстве признаков изучаемых объектов.

Многослойный перцептрон (multilayered perceptron, MLP) – сеть с одним входным, одним выходным и одним или более скрытыми слоями нейронов.

Мода (mode) – значение во множестве наблюдений, которое встречается наиболее часто, т. е. значение признака, имеющее наибольшую частоту в статистическом ряду распределения.

Модель «мешок слов» (bag-of-words) – это упрощение, применяемое в области обработки ЕЯ. Текст представляется как неупорядоченная коллекция слов без учета грамматических правил и порядка слов в предложениях.

Набор данных (dataset) – совокупность экземпляров данных.

Недообучение (underfitting) – недостаточная аппроксимация целевой функции, когда модели не удается вычленить даже базовые данные.

Нейрон (neuron) – базовый элемент нейронных сетей. Имеет входы, снабженные весами, смещение, суммирующий элемент и выходную активационную функцию; выполняет функцию адаптивного сумматора с варьируемыми входными весовыми коэффициентами, суммарный выходной сигнал которого подвергается линейной или нелинейной обработке, образуя итоговый выходной сигнал.

Нейронная сеть (neural network) – структура соединенных между собой нейронов, которая характеризуется топологией, свойствами узлов, а также правилами обучения или тренировки для получения желаемого выходного сигнала.

Неметрические данные (nonmetric data) – оцениваются по номинальной или порядковой шкале.

Область задачи (problem domain), или пространство задачи (problem space) – сфера профессиональной деятельности или применения, которая подлежит исследованию в целях решения задачи.

Обобщение (generalization) – способность модели обобщать результат на новые данные, не участвовавшие в обучении.

Обратное распространение (backpropagation) – наиболее применяемый способ управляемого обучения, при котором сигнал ошибки на выходе нейронной сети распространяется в обратном направлении: от нейронов выходного слоя к нейронам входного слоя с последующей корректировкой синаптических весов нейронной сети для достижения минимальной выходной погрешности.

Объясненная вариация (explained variation) – статистический показатель, который измеряет долю вариации (дисперсии) в неком наборе данных, приходящуюся на математическую модель. Нередко вариация измеряется в виде дисперсии, и тогда используется более узкий термин «объясненная дисперсия» (explained variance).

Одномерные статистические методы (univariate techniques) – методы статистического анализа данных в случаях, если существует единый измеритель для оценки каждого элемента выборки либо если этих измерителей несколько, но каждая переменная анализируется отдельно от всех остальных.

Отношение шансов (odds ratio) – отношение вероятности наступления интересующего события к вероятности его ненаступления.

Оценка максимального правдоподобия (maximum likelihood estimation) – метод оценивания неизвестного параметра путем максимизации функции вероятности, в результате которой приобретаются значения параметров модели, которые делают данные «ближе» к реальным.

Оценщик (estimator) – оценивающее правило или функция, используемые для оценки заданного параметра, основываясь на наблюдаемых данных. Различают правило (оценщика), количественный параметр (оцениваемое) и результат (оценку). В математической статистике используется для оценки выборки, основываясь на имеющихся эмпирических данных, и в результате для получения информации о неизвестных параметрах генеральной совокупности.

Пенек решения (decision stump) – одноуровневое дерево решений, которое представляет собой дерево с одним внутренним узлом (корнем), который непосредственно подключен к терминальным узлам (своим листьям). Пенек решения выполняет прогноз, исходя из значения всего одного входного объекта.

Перекрестная проверка (cross validation) качества модели – это метод подтверждения качества моделей, который используется для оценки того, как результаты статистического анализа будут распространяться на независимый набор данных. В машинном обучении перекрестная проверка используется с целью оценить, насколько точно прогнозная модель будет работать на практике. В задаче прогнозирования модель обычно получает набор известных данных, на котором выполняется обучение (тренировочный набор), и набор неизвестных данных (ранее не встречавшихся), на которых модель тестируется (тестовый набор). Задача перекрестной проверки состоит в определении набора данных, который используется с целью «протестировать» модель на этапе ее тренировки (так называемый перекрестно-проверочный набор), с тем чтобы ограничить такие проблемы, как переобучение, дать представ-

ление о том, как модель будет обобщаться на независимый набор данных (т. е. ранее не встречавшихся данных, например из реальной задачи).

Перекрестная проверка с исключением по одному (leave-one-out CV) – это простая перекрестная проверка, где каждый обучающий набор создается с использованием всех образцов, за исключением одного, и этот отложенный образец становится тестовым набором. Так, для n образцов будет иметься n разных тренировочных и n разных тестовых наборов. Эта процедура перекрестной проверки не тратит слишком много данных, т. к. из тренировочного набора удаляется всего один образец.

Переобучение (overfitting) – чрезмерная аппроксимация целевой функции (чрезмерно близкая подгонка), которая приводит к тому, что модель вычленяет шум.

Персептрон (perceptron) – простейшая форма нейронной сети, под которой в основном понимается элементарный нейрон, представляющий собой линейный сумматор, каждый из входных сигналов которого умножается на некоторый весовой множитель, а выходной суммарный сигнал является ненулевым, если сумма превышает некоторое пороговое значение.

Персистентность (persistance), или живучесть – в программировании означает способность состояния существовать дольше, чем процесс, который его создал. Без этой возможности состояние может существовать только в оперативной памяти и теряется, когда оперативная память выключается, например при выключении компьютера.

Площадь под ROC-кривой (area under ROC-curve, AUC) – площадь, ограниченная ROC-кривой и осью доли ложноположительных классификаций. Чем выше оценка AUC, тем качественнее классификатор, при этом значение 0.5 демонстрирует непригодность выбранного метода классификации (соответствует случайному гаданию).

Подгонка (fitting) – аппроксимация целевой функции; также адаптация (подстройка) модели под тренировочные данные.

Правдоподобие (likelihood), или функция правдоподобия – это совместное распределение выборки из параметрического распределения, рассматриваемое как функция параметра. При этом используется совместная функция плотности (в случае выборки из непрерывного распределения) либо совместная вероятность (в случае выборки из дискретного распределения), вычисленные для значений выборочных данных. В статистике значения понятий вероятности и правдоподобия различаются по роли параметра и результата. Вероятность используется для описания результата функции при наличии фиксированного значения параметра (если сделать 10 бросков уравновешенной монеты, какова вероятность, что она повернется 10 раз орлом?). Правдоподобие используется для описания параметра функции при наличии ее результата (если брошенная 10 раз монета повернулась 10 раз орлом, насколько правдоподобна уравновешенность монеты?).

Правило обучения (learning rule), или процесс обучения – это применяемый многократно метод (математическая формула) обновления весов и узла смещения модели, который улучшает ее эффективность. В нейронных сетях это метод (математическая логика), который улучшает эффективность искусственной нейронной сети и обычно применяется по сети многократно. Это делается путем обновления весов и узла смещения сети при моделировании сети в конкретной среде данных.

Прямое кодирование (one-hot encoding, OHE) – представление конечного множества значений признака в виде двоичного кода фиксированной длины, содержа-

щего только одну 1, т. е. где i-е значение признака равно 1, а все остальные равны 0. При обратном кодировании имеется только один 0. Длина кода определяется количеством кодируемых значений признака, или объектов.

Сеточный поиск (grid search), или поиск по сетке параметров – это просто исчерпывающий поиск в заданном вручную подмножестве гиперпараметрического пространства.

Слой (layer) – множество нейронов (узлов), имеющих общие входные или выходные сигналы.

Специфичность (specificity) – отношение количества истинно отрицательных исходов к общему количеству отрицательных исходов (например, доля здоровых людей, которые правильно определены как не имеющие заболевания).

Собственная пара (число, вектор) матрицы (eigenpair) – это вещественное число λ и вектор z матрицы A, если они удовлетворяют условию, что $Az = \lambda z$.

Стандартная оценка (z-score), или z-оценка – статистический показатель, который позволяет определить меру отклонения величины от среднего, т. е. показываает, на сколько стандартных отклонений от среднего лежит выбранное значение переменной.

Статистический критерий (statistical criterion) – строгое математическое правило, на основе которого принимается или отвергается та или иная статистическая гипотеза.

Стратифицированная *k*-блочная перекрестная проверка (т. е. с разбивкой по классам) (stratified k-fold validation test) – это вариант *k*-блочной перекрестной проверки, которая возвращает стратифицированные блоки: каждый набор содержит приблизительно тот же процент образцов каждого целевого класса, что и полный набор.

Тестирование (testing) – этап проверки качества модели, в течение которого на вход модели подаются данные, которые не были использованы в процессе тренировки.

Тестовый набор данных (testing dataset) – подмножество данных, которое мы используем для установления точности модели, но которое не используется для тренировки модели.

Точка пересечения (intercept) – геометрически это точка, в которой линия регрессии пересекает ось Y. В формуле линейной регрессии пересечение – это свободный коэффициент, которому равна зависимая переменная, если предиктор, или независимая переменная, равен нулю.

Точность (precision) – отношение количества истинно положительных исходов к общему количеству положительных исходов.

Тренировка нейронной сети (neural network training) – этап функционирования нейронной сети, в процессе которого на ее вход поочередно поступают данные из тренировочного набора с целью корректировки весовых коэффициентов синаптических связей для получения наиболее адекватного сигнала на выходе нейронной сети.

Тренировочный набор данных (training dataset) – подмножество данных, которое подается на вход алгоритма машинного обучения для тренировки модели.

Тренировочный образец (training sample) – это точка данных x в имеющемся наборе тренировочных данных, который используется для решения задачи прогнозного моделирования.

Узел (unit) – отдельный нейрон в нейронной сети, имеющий, как правило, несколько сигналов на входе и лишь один выходной сигнал. Термин «узел» исполь-

зуется обычно для обозначения и представления единичного нейрона в структуре нейронной сети; в рамках графовой модели нейронной сети эквивалентен вершине графа.

Узел смещения (bias unit), или просто смещение – параметр нейрона, который суммируется со взвешенными входами нейрона, образуя текущее состояние нейрона – входную величину (аргумент) для функции активации нейрона.

Функция массы вероятности (probability mass function, PMF) – функция распределение вероятностей дискретной случайной величины.

Хэш-коллизия (hash collision), или хэш-конфликт – ситуация, когда два различных ключа выбирают («хэшируют») одно и то же значение, указывая на одну и ту же ячейку в хэш-таблице.

Чувствительность (sensitivity) – отношение количества истинно положительных исходов к количеству всех положительных исходов в выборке. Чувствительность еще называют полнотой (recall) (например, доля больных людей, которые правильно определены как имеющие заболевание).

Ядерная функция (kernel function), или просто ядро – функция известного типа (как правило, гауссова), которая размещается в известных точках данных и затем суммируется, таким образом строится аппроксимация выборочного распределения; скалярное произведение в преобразованном пространстве.

Сокращения

API – интерфейс программирования приложений, программный интерфейс.

BLAS – динамическая библиотека базовых подпрограмм линейной алгебры.

Breast Cancer Wisconsin – экспериментальный набор данных о раке груди по шт. Висконсин, США.

C – язык программирования.

C++ – язык программирования.

CSV – файл значений, разделенных запятыми.

Facebook – социальная сеть.

Flask – библиотека Python, реализующая легковесную веб-платформу.

Fortran – язык программирования.

GLM (Generalized Linear Model) – обобщенная линейная модель (ОЛМ).

Housing – экспериментальный набор данных жилищного фонда с информацией о зданиях в пригородах Бостона, США.

Iris – экспериментальный набор данных цветков ириса (ирисы Фишера).

JavaScript – язык программирования.

JSON (JavaScript Object Notation) – объектная форма записи JavaScript, текстовый формат обмена данными.

Jupyter – интерактивная вычислительная среда.

Keras – высокоуровневая библиотека Python для работы с глубокими нейронными сетями.

LAPACK (Linear Algebra PACKage) – динамическая библиотека с методами решения основных задач линейной алгебры.

LIBLINEAR – динамическая библиотека, предоставляющая оптимизированный классификатор для данных с миллионами экземпляров и признаков.

LIBSVM – динамическая библиотека для работы с методами (машинами) опорных векторов.

Matplotlib – библиотека Python для работы с двумерными графиками.

Markdown – файлы с упрощенной разметкой.

NLTK (Natural Language Toolkit) – лидирующая платформа для создания программ на Python по обработке данных на естественном языке.

MNIST (Mixed National Institute of Standards and Technology) – смешанная база данных Национального института стандартов и технологий, США, эталонный набор данных рукописных цифр для тестирования алгоритмов машинного обучения.

NumPy – основополагающая библиотека Python, необходимая для научных вычислений на Python.

OpenCL – параллельная вычислительная платформа.

SIMD (single instruction, multiple data) – процессорная архитектура с одиночным потоком команд и множественным потоком данных (ОКМД).

Pandas – библиотека Python, инструмент для анализа структурных данных и временных рядов.

Python – язык программирования.

R – язык программирования.

Seaborn – библиотека Python для визуализации, основанная на matplotlib.

Scikit-learn – библиотека Python, интегратор классических алгоритмов машинного обучения.

SciPy – библиотека Python, используемая в математике, естественных науках и инженерном деле.

SQLite – простая реляционная СУБД.

Theano – библиотека Python, позволяющая определять, оптимизировать и вычислять математические выражения с использованием многомерных массивов.

TSV – файл значений, разделенных табуляцией.

Twitter – социальная сеть.

UCI (University of California, Irvine) – Калифорнийский университет в Ирвайне, США.

URL (Uniform Resource Locator) – единообразный определитель местонахождения ресурса в сети.

Wine – экспериментальный набор данных сортов вин.

Предметный указатель

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ч

Ш

Э

Я

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью), по которому должны быть
высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Себастьян Рашка

Python и машинное обучение

**Наука и искусство построения алгоритмов,
которые извлекают знания из данных**

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16 .
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 39,1875. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Машинное обучение и прогнозная аналитика преобразуют традиционную схему функционирования предприятий!

Данная книга предоставит вам доступ в мир прогнозной аналитики и продемонстрирует, почему Python является одним из ведущих языков науки о данных. Если вы хотите более глубоко и точно анализировать данные либо нуждаетесь в усовершенствовании и расширении систем машинного обучения, эта книга окажет вам неоценимую помощь. Ознакомившись с широким кругом мощных программных библиотек Python, в том числе scikit-learn, Theano и Keras, а также получив советы по всем вопросам, начиная с анализа мнений и заканчивая нейронными сетями, вы сможете принять важные решения, во многом определяющие деятельность вашей организации.

Чему вы научитесь, прочитав эту книгу:

- исследовать, как используются разные машиннообучаемые модели, которые формулируют те или иные вопросы в отношении данных;
- конструировать нейронные сети при помощи библиотек Theano и Keras;
- писать красивый и лаконичный программный код на Python с оптимальным использованием созданных вами алгоритмов;
- встраивать вашу машиннообучаемую модель в веб-приложение для повышения ее общедоступности;
- предсказывать непрерывнозначные результаты при помощи регрессионного анализа;
- обнаруживать скрытые повторяющиеся образы и структуры в данных посредством кластерного анализа;
- организовывать данные с помощью эффективных методов предобработки и использовать передовые практические подходы к оценке машиннообучаемых моделей;
- осуществлять анализ мнений, позволяющий подробнее интерпретировать текстовые данные и информацию из социальных сетей.

Если вы хотите узнать, как использовать Python, чтобы начать отвечать на критические вопросы в отношении ваших данных, возьмите книгу "Python и машинное обучение" — и неважно, приступаете ли вы к изучению науки о данных с нуля или просто намереваетесь расширить по ней свои знания.

Наука и искусство построения алгоритмов, которые извлекают знания из данных!

Инетенет-магазин:
www.dmkpress.com
Книга – почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 782-3889
e-mail: books@alians-kniga.ru

[PACKT] PUBLISHING

ДМК ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-409-0
9 785970 604090 >