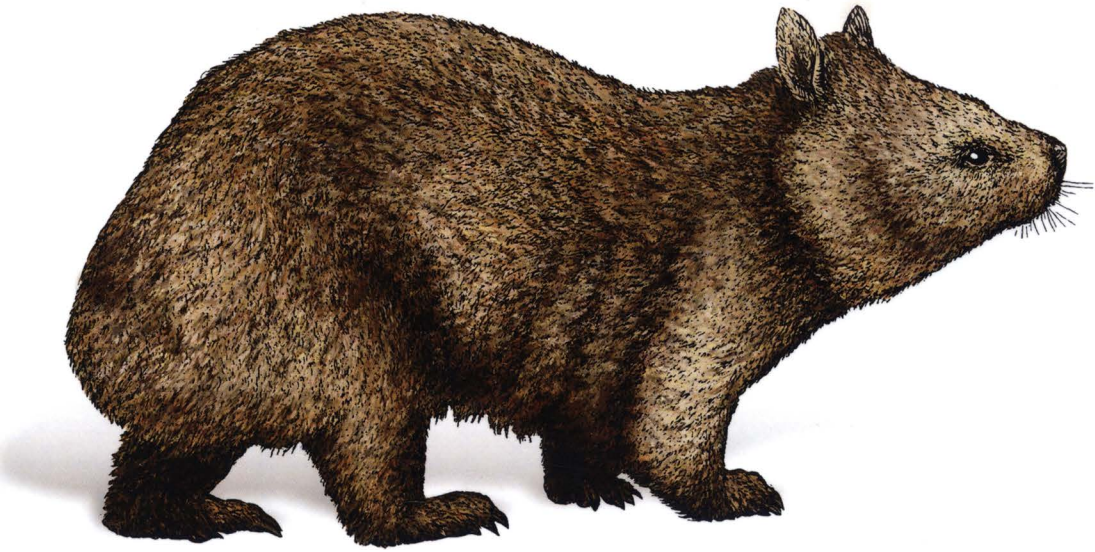
O'REILLY®

Прикладное машинное обучение без учителя

с использованием Python



Анкур Пател

Прикладное машинное обучение без учителя

с использованием Python

Анкур Пател

Москва · Санкт-Петербург
2020

ББК 32.973.26-018.2.75
П20
УДК 681.3.07

Компьютерное издательство "Диалектика"
Перевод с английского канд. хим. наук *А.Г. Гузикевича*
Под редакцией *В.Р. Гинзбурга*
По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info.dialektika@gmail.com, http://www.williamspublishing.com

Пател, Анкур

П20 Прикладное машинное обучение без учителя с использованием Python. : Пер. с англ. — СПб. : ООО "Диалектика", 2020. — 432 с. — Парал. тит. англ.

ISBN 978-5-907144-99-6 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Анкур Пател

Прикладное машинное обучение без учителя с использованием Python

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

ISBN 978-5-907144-99-6 (рус.)
ISBN 978-1-492-03564-0 (англ.)

Оглавление

Содержание

Об авторе

Анкур Пател — вице-президент компании 7Park Data, входящей в портфель активов инвестиционной компании Vista Equity Partners. Вместе со своей командой разрабатывает программные продукты по обработке данных для хедж-фондов, а также систему MLaaS (машинное обучение как услуга), предназначенную для корпоративных клиентов. В системе MLaaS реализованы такие услуги, как обработка естественного языка, обнаружение аномалий, кластеризация и прогнозирование временных рядов. До того как начать работать в компании 7Park Data, Анкур занимался исследованиями для израильской компании ThetaRay, одного из пионеров в области прикладного машинного обучения без учителя.

Анкур начинал свою карьеру в качестве аналитика в J.P. Morgan, после чего работал ведущим трейдером по кредитным операциям с развивающимися странами в инвестиционной компании Bridgewater Associates, являющейся крупнейшим хедж-фондом в мире. Впоследствии учредил хедж-фонд R-Squared Macro, разрабатывавший для клиентов стратегии глобальных макроинвестиций на основе методов машинного обучения. Выпускник Школы общественных и международных отношений имени Вудро Вильсона при Принстонском университете, где был удостоен премии имени Джона Ларкина за лучшую магистерскую диссертацию в области политической экономии.

Об изображении на обложке

Животное на обложке книги — *короткошерстный, или обыкновенный, вомбат* (лат. *Vombatus ursinus*). Несмотря на то что его видовое название включает слово "ursinus" (медведь), вомбат — сумчатое животное, близкий родственник коал и кенгуру. Дикие вомбаты водятся лишь в Австралии и Тасмании. Их естественной средой обитания служат прибрежные леса, редколесья и кустарниковые степи, где они роют норы своими острыми когтями.

Тело вомбата покрыто короткой густой шерстью, его конечности короткие и сильные, нос голый, а уши небольшие. Как и у любого сумчатого, у вомбата имеется сумка (внешний мешочек), в которой живет детеныш, но она повернута назад. Таким образом, мордочка малыша обращена в сторону задних конечностей матери. Благодаря этому, когда вомбат роет землю, грязь не попадает в сумку. Детеныши рождаются лысыми. Беременность длится месяц, но в течение примерно еще одного года мать продолжает кормить и согревать малыша.

Длина тела взрослого вомбата в среднем достигает одного метра, а вес — примерно 20 кг. В дикой природе вомбаты живут около 15 лет и производят потомство раз в два года. В ночное время они грызут всевозможные травы и коренья своими непрерывно растущими передними резцами. В целом вомбаты — ночные животные, тем не менее они не прочь понежиться на солнышке, когда наступают холода.

Недавно было обнаружено, что самка вомбата, готовая к спариванию, покусывает самца сзади. Такие укусы ничем не вредят самцу, поскольку его шкура в этом месте достаточно грубая. Более того, когда вомбат отражает нападение хищника, он поворачивается к нему задом и залазит в нору, оставляя незащищенной самую прочную часть своего тела. Внешне неуклюжий, перед лицом угрозы вомбат способен развивать скорость до 40 км/час.

Многие из животных, изображаемых на обложках книг издательства O'Reilly, находятся под угрозой вымирания, и все они представляют ценность для нашего мира. Чтобы узнать о том, каким может быть ваш личный вклад в их спасение, посетите сайт `animals.oreilly.com`.

Изображение на обложке книги — копия черно-белой гравюры XIX столетия из многотомного издания *Royal Natural History* Лидеккера.

Введение

Краткая история машинного обучения

Машинное обучение — класс методов искусственного интеллекта (ИИ), позволяющих компьютерам обучаться на основе данных (обычно в ходе решения узкоспециализированных задач) без явного программирования. Термин *машинное обучение* был введен Артуром Сэмюэлем, легендой в области ИИ, еще в 1959 году, однако до конца XX века лишь немногим проектам удалось достигнуть коммерческого успеха в области машинного обучения, которая оставалась не более чем нишей для академических исследований.

Поначалу (в 1960-е годы) многие члены сообщества ИИ были полны оптимизма. Например, Герберт Саймон и Марвин Минский полагали, что достижение искусственным интеллектом уровня человеческого разума — дело ближайших десятилетий[1]:

> Через двадцать лет машины будут способны выполнять любую работу, которая под силу человеку.
>
> *Герберт Саймон, 1965 г.*

> Через 3–8 лет у нас появятся машины, обладающие общим интеллектом на уровне типичного человека.
>
> *Марвин Минский, 1970 г.*

Ослепленные оптимизмом, исследователи сфокусировались на развитии проектов так называемого *сильного* ИИ. Их целью было создание интеллектуальных агентов, способных решать задачи, формировать знания, обучаться и планировать свои действия, понимать естественный язык, воспринимать окружающую среду и контролировать моторику. Всеобщий энтузиазм способствовал привлечению серьезного финансирования со стороны таких организаций, как Министерство обороны США, однако задачи, стоявшие перед исследователями, оказались настолько амбициозными, что попытки решить их в ту пору были заведомо обречены на провал.

[1] Эти идеи вдохновили Стэнли Кубрика на создание в 1968 году образа разумного компьютера HAL 9000 в фильме "Космическая одиссея 2001 года".

Академические исследования редко доводились до уровня промышленных разработок, что проявило себя в серии так называемых "зим искусственного интеллекта". Эти "зимние" периоды (аллюзия на выражение "ядерная зима", бытовавшее в эпоху холодной войны) характеризовались угасанием интереса к ИИ и соответствующим сокращением финансирования. Временами вокруг ИИ поднималась рекламная шумиха, однако никакого реального эффекта это не приносило. К началу 1990-х годов интерес к ИИ и его финансирование были почти сведены на нет.

ИИ снова на пике популярности, но почему именно сейчас?

Тем не менее за последние два десятилетия отрасль возродилась — сначала в академических кругах, а затем в виде массового феномена, в который вовлечены лучшие умы из университетской и корпоративной среды. Решающую роль в возрождении ИИ сыграли три фактора: прорыв в разработке алгоритмов машинного обучения, доступность больших объемов данных и появление сверхбыстрых компьютеров.

Во-первых, вместо того, чтобы фокусироваться на чересчур амбициозных проектах сильного ИИ, исследователи сосредоточились на более узких задачах, известных как *слабый*, или *ограниченный*, ИИ. Результатом акцентирования внимания на улучшении решений для узких задач стал прорыв в разработке алгоритмов, проложивший путь к успешным коммерческим приложениям. Многие из этих алгоритмов — часто из числа тех, которые первоначально разрабатывались в упиверситетах или частных исследовательских лабораториях, — быстро превратились в проекты с открытым исходным кодом, что ускорило внедрение этих технологий в промышленной среде.

Во-вторых, большинство организаций занялось накоплением данных, а стоимость их хранения резко упала благодаря прогрессу в создании эффективных компьютерных хранилищ, в то время как Интернет обеспечил доступность больших объемов данных в невиданных ранее масштабах.

В-третьих, благодаря облачным технологиям стали доступны сверхмощные вычислительные ресурсы, что позволяет исследователям легко и недорого масштабировать IT-инфраструктуру, не делая огромных инвестиций в оборудование.

Появление прикладного ИИ

Под влиянием трех вышеперечисленных факторов исследования в области ИИ переместились из академической сферы в промышленную, что способствовало повышению интереса к ИИ и привлечению более существенного финансирования, которое из года в год увеличивается. Теперь искусственный интеллект — не только предмет теоретических исследований, но и полноценная прикладная область. На рис. 1 показан график из Google Trends, свидетельствующий о росте интереса к машинному обучению за последние годы.

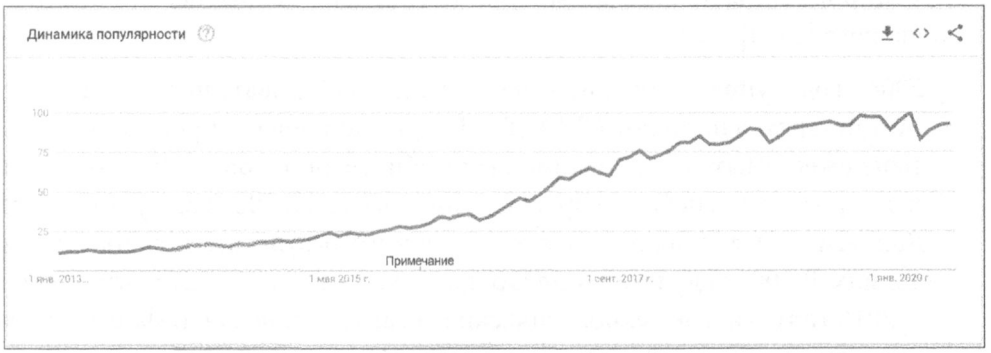


Рис. 1. *Рост интереса к машинному обучению*

В настоящее время ИИ рассматривается как революционная технология, сродни появлению компьютеров и смартфонов, которая на протяжении последующего десятилетия окажет значительное влияние на все отрасли промышленности[2].

Успешные коммерческие применения машинного обучения включают оптическое распознавание текста, фильтрацию спама электронной почты, компьютерное зрение, распознавание речи, машинный перевод, генерирование синтетических данных, обнаружение аномалий, предотвращение киберпреступлений, выявление мошеннических операций с банковскими картами, прогнозирование временных рядов, классификацию документов, рекомендательные и поисковые системы, робототехнику, онлайн-рекламу, сентимент-анализ, анализ финансовых рынков и многое другое.

[2] Согласно прогнозам международной консалтинговой компании McKinsey больше половины видов профессиональной деятельности, осуществляемых в настоящее время людьми, могут быть автоматизированы к 2055 г.

Основные этапы развития прикладного ИИ за последние 20 лет

Нижеперечисленные вехи развития ИИ способствовали тому, что эта область исследований, первоначально представлявшая лишь академический интерес, превратилась в передовой край современной науки.

- 1997 год: шахматный суперкомпьютер Deep Blue под управлением ИИ, находившийся в разработке с середины 1980-х годов, выиграл матч у чемпиона мира по шахматам Гарри Каспарова — событие, широко освещавшееся в прессе.
- 2004 год: управление перспективных исследовательских проектов Министерства обороны США (DARPA) организовало проведение в пустыне ежегодных соревнований автомобилей-роботов. В 2005 году главный приз завоевал Стэнфордский университет. В 2007 году университет Карнеги — Меллона добился аналогичного результата в городских условиях. В 2009 году компания Google создала беспилотный автомобиль. К 2015 году многие технологические гиганты, включая Tesla и Uber, запустили щедро финансируемые программы по созданию беспилотных машин.
- 2006 год: Джеффри Хинтон из университета Торонто предложил быстрый алгоритм обучения многослойных нейронных сетей, что положило начало революции глубокого обучения.
- 2006 год: компания Netflix организовала соревнования Netflix Prize с призом в один миллион долларов, предложив командам разработчиков использовать машинное обучение для повышения точности своей рекомендательной системы по крайней мере на 10%. Одна из команд завоевала этот приз в 2009 году.
- 2007 год: ИИ достиг уровня человеческих возможностей при игре в шашки; соответствующее решение было предложено командой разработчиков из университета Альберты.
- 2010 год: в рамках проекта ImageNet были организованы ежегодные соревнования ILSVRC (ImageNet Large Scale Visual Recognition Challenge), в которых команды-участники должны использовать алгоритмы машинного обучения для корректного распознавания и классификации объектов, хранящихся в виде крупного и хорошо детализированного набора.

Эти соревнования привлекли значительное внимание со стороны как академических, так и технологических гигантов. Благодаря прогрессу в разработке глубоких сверточных нейронных сетей ошибку классификации, которая в 2011 году составила 25%, удалось снизить всего лишь до нескольких процентов в 2015 году, что сделало возможным создание коммерческих приложений для компьютерного зрения и распознавания объектов.

- 2010 год: компания Microsoft выпустила бесконтактный сенсорный игровой контроллер Kinect для консоли Xbox 360, способный отслеживать движения человеческого тела и транслировать их для управления игрой.
- 2010 год: компания Apple купила Siri, один из первых голосовых помощников, и в октябре 2011 года включила его в состав программного обеспечения для iPhone 4S. В конечном счете голосовой помощник Siri стал составной частью всех продуктов Apple. Благодаря использованию сверточных нейронных сетей и рекуррентных сетей с долгой краткосрочной памятью,Siri способен распознавать речь и обрабатывать естественный язык. Вскоре в эту гонку включились компании Amazon, Microsoft и Google, выпустив аналогичные программные продукты Alexa (2014), Cortana (2014) и Google Assistant (2016) соответственно.
- 2011 год: IBM Watson, интеллектуальный агент вопросно-ответной системы, разработанный командой под руководством Дейвида Ферруччи, побил результаты предыдущих победителей американской телевикторины *Jeopardy!* Брэда Раттера и Кена Дженнингса. В настоящее время IBM Watson используется в ряде отраслей, включая здравоохранение и розничную торговлю.
- 2012 год: группа, работавшая над проектом Google Brain под руководством Эндрю Ына и Джеффа Дина, обучила нейронную сеть распознаванию котов на неразмеченных изображениях, взятых из видеороликов YouTube.
- 2013 год: компания Google выиграла соревнования Robotics Challenge, проводимые DARPA, в которых полуавтономные роботы решают сложные задачи в условиях изменчивого окружения: управление автомобилем, перешагивание через препятствия, освобождение входа от загромождающих его предметов, открытие дверей и подъем по лестнице.

- 2014 год: компания Facebook опубликовала результаты разработки нейросистемы DeepFace, способной распознавать лица людей на цифровых изображениях с точностью 97%, что более чем на 27% превосходило результаты предыдущих систем и приближалось к уровню человеческих возможностей.
- 2015 год: компания DeepMind разработала программу AlphaGo для игры в го, которая выиграла матч у профессионала мирового класса Фань Хуэя. В 2016 году AlphaGo победила Ли Седоля, а в 2017 году — Кэ Дзе. В 2017 году новая версия программы под названием AlphaGo Zero одержала победу над предыдущей версией AlphaGo со счетом 100:0. Программа AlphaGo Zero использует методы обучения без учителя и овладевает мастерством игры в го, играя сама с собой.
- 2016 год: компания Google выпустила кардинально переработанную версию своей системы машинного перевода Google Translate, заменив существующую программу, основанную на переводе фраз, нейросистемой глубокого обучения, что позволило снизить число ошибок перевода до 87%, приблизив ее к уровню человеческих возможностей.
- 2017 год: программа Libratus, разработанная в университете Карнеги — Меллона, стала победителем в турнире по безлимитному техасскому холдему.
- 2017 год: бот, обученный некоммерческой исследовательской компанией OpenAI, обыграл профессионального игрока на турнире по Dota 2.

От слабого ИИ к сильному

Конечно, перечисленные успехи в применении ИИ для решения узких задач — это всего лишь отправная точка. В среде сообщества ИИ крепнет вера в то, что объединение нескольких *слабых* систем ИИ откроет путь к разработке *сильного* ИИ (artificial general intelligence — AGI). Такой AGI-агент сможет функционировать на уровне человеческих возможностей для решения широкого круга задач.

Некоторые исследователи прогнозируют, что вслед за тем, как ИИ сравняется по своим возможностям с человеком, столь сильный ИИ превзойдет человеческий интеллект и достигнет уровня так называемого *суперинтеллекта*. Согласно различным оценкам для этого потребуется как минимум 15, а то и 100 лет, но большинство исследователей сходятся в том, что ИИ сможет достаточно

продвинуться в этом направлении на протяжении нескольких поколений. Что это: результат вновь раздутой рекламной шумихи (с чем мы уже сталкивались на предыдущих циклах развития ИИ), или же теперь все будет по-другому?

Время покажет.

Цели и подходы

На сегодняшний день в большинстве успешных коммерческих приложений — а это такие области, как компьютерное зрение, распознавание речи, машинный перевод и обработка естественного языка, — применяется обучение с учителем, позволяющее использовать преимущества размеченных наборов данных. Однако подавляющая часть накопленных в мире данных не размечена.

Эта книга посвящена обсуждению методов *обучения без учителя* (unsupervised learning) — направления машинного обучения, применяемого для выявления скрытых закономерностей в неразмеченных данных. По мнению многих экспертов, например Яна Лекуна, директора подразделения AI Research в компании Facebook и профессора Нью-Йоркского университета, обучение без учителя — передовой рубеж технологий искусственного интеллекта и, возможно, ключ к созданию AGI. В силу этой и ряда других причин обучение без учителя в настоящее время является одной из самых востребованных тем в области ИИ.

Цель книги — сформировать у читателей общее представление об основных концепциях и инструментах обучения без учителя, что должно способствовать развитию интуиции, которая необходима для практического применения данной технологии. Будет также показано, как эффективная разметка немаркированных данных позволяет превращать задачи обучения без учителя в задачи с частичным привлечением учителя.

В книге мы будем следовать прикладному подходу. Предварительно мы ознакомимся с необходимыми теоретическими основами, но затем уделим основное внимание решению практических задач. Используемые наборы данных и код доступны в виде блокнотов Jupyter на сайте GitHub (`http://bit.ly/2Gd4v7e`).

Вооружившись пониманием основных концепций и опытом их практического использования, вы сможете применять метод обучения без учителя к крупным наборам неразмеченных данных для выявления скрытых закономерностей, более глубокого анализа деловых данных, обнаружения аномалий, кластеризации на основе сходства, автоматического конструирования признаков, генерирования синтетических наборов данных и решения многих других задач.

Исходные предположения

В книге предполагается, что вы уже обладаете опытом программирования на языке Python, а также знакомы с библиотеками NumPy и Pandas.

Дополнительную информацию о Python можно найти на официальном сайте (`www.python.org`). Для получения информации о среде Jupyter Notebook посетите официальный сайт проекта (`http://jupyter.org/index.html`). Чтобы освежить свои знания по высшей математике, прочитайте часть I книги *Глубокое обучение* (`www.deeplearningbook.org`). Книга *Основы статистического обучения: интеллектуальный анализ данных, логический вывод и прогнозирование, 2-е издание* (`https://stanford.io/2Tju4al`) позволит вам освежить знания по машинному обучению.

Структура книги

Книга разбита на четыре части, охватывающие следующие темы.

Часть I. Основы обучения без учителя

Различия между обучением с учителем и без учителя, обзор популярных алгоритмов и готовый проект машинного обучения.

Часть II. Обучение без учителя с использованием библиотеки Scikit-learn

Снижение размерности, обнаружение аномалий, а также кластеризация и сегментация групп.

Для получения более полной информации, касающейся материала, изложенного в частях I и II, обратитесь к документации по библиотеке Scikit-learn (`https://scikit-learn.org/stable/modules/classes.html`).

Часть III. Обучение без учителя с использованием библиотек TensorFlow и Keras

Обучение представлениям, автоматическое извлечение признаков, автокодировщики и обучение с частичным привлечением учителя.

Часть IV. Глубокое обучение без учителя с использованием библиотек TensorFlow и Keras

Ограниченные машины Больцмана, глубокие сети доверия и генеративно-состязательные сети.

Соглашения, принятые в книге

В книге используются следующие типографские соглашения.

Курсив

Курсивом выделены новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный шрифт`

Используется в листингах программ, а также в основном тексте для выделения таких программных элементов, как переменные, имена функций, типы данных и ключевые слова.

`Полужирный моноширинный шрифт`

Используется для выделения команд, которые должен вводить пользователь.

`Курсивный моноширинный шрифт`

Используется для выделения текста, который должен быть заменен пользовательскими значениями или значениями, определяемыми контекстом.

Этой пиктограммой помечены советы и рекомендации.

Этой пиктограммой помечены замечания общего характера.

Этой пиктограммой помечены предупреждения и предостережения.

Файлы примеров и цветные иллюстрации

Все примеры программ, используемые в книге, доступны для загрузки на сайте GitHub:

`http://bit.ly/2Gd4v7e`

Также архив локализованных файлов доступен на сайте издательства:

`http://www.williamspublishing.com/Books/978-5-907144-99-6.html`

Все иллюстрации к книге в цветном варианте доступны по адресу:

`http://go.dialektika.com/unsupervised`

Ждем ваших отзывов!

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

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

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

Наши электронные адреса:

E-mail: `info.dialektika@gmail.com`

WWW: `http://www.williamspublishing.com`

Основы обучения без учителя

Мы начнем первую часть книги с обсуждения текущего положения дел в области машинного обучения и способов внедрения методов обучения без учителя. Мы реализуем готовый проект с нуля, что позволит понять, как настроить программную среду, загрузить и подготовить данные, выбрать алгоритмы машинного обучения и функцию потерь, а также оценить полученные результаты.

ГЛАВА 1

Обучение без учителя как один из видов машинного обучения

Большую часть знаний люди и животные приобретают в ходе самообучения. Обучение без учителя можно представить как торт, глазурь на котором — обучение с учителем, а вишенка на торте — обучение с подкреплением. Мы знаем, как подготовить глазурь и вишенку, но не знаем, как испечь торт. Прежде чем задумываться о том, как приблизиться к созданию истинного ИИ, нам нужно справиться с задачей обучения без учителя.

Ян Лекун

Данная глава посвящена изучению различий между обучением с учителем и без учителя, а также сильных и слабых сторон каждого из этих подходов. Кроме того, мы обсудим многие популярные алгоритмы обучения с учителем и без учителя и кратко рассмотрим такие методы, как обучение с частичным привлечением учителя и обучение с подкреплением.

Базовая терминология машинного обучения

Прежде чем углубиться в изучение различных типов машинного обучения, рассмотрим простой пример, который поможет лучше понять вводимые понятия: фильтрация спама. Предположим, требуется создать простую программу, которая получает сообщения электронной почты и классифицирует их либо как "спам", либо как "не спам". Это типичная задача классификации.

Освежим в памяти ключевые термины машинного обучения. В этой задаче *входными переменными* служат тексты сообщений электронной почты. Входные переменные также называют *независимыми переменными*, *признаками* или *предикторами*. В нашем случае *выходная переменная* — т.е. то, что мы пытаемся предсказать, — имеет значения "спам" или "не спам". Такую переменную также называют *целевой*, *зависимой* или *ответной* (а еще *классом*, поскольку это задача классификации).

Набор примеров, на которых обучается ИИ, называют *обучающим* (*тренировочным*) *набором*, а каждый отдельный пример называют *обучающим* (*тренировочным*) *примером* или *образцом* (выборкой). В процессе обучения ИИ пытается минимизировать свою *функцию потерь* (cost function), или *частоту ошибок* (error rate), или же (в более позитивной формулировке) максимизировать свою *функцию значения* (value function), в данном случае — процент корректной классификации электронных сообщений. Частоту ошибок вычисляют путем сравнения предсказанной метки с истинной.

Однако нас больше всего интересует то, насколько хорошо ИИ способен обобщать опыт, приобретенный в процессе обучения, на данные, которые ему прежде не встречались. Истинным тестом будет следующий: способен ли ИИ корректно классифицировать сообщения, которые не вошли в состав примеров, образующих тренировочный набор? Именно *ошибка обобщения* (generalization error), или *ошибка за пределами выборки* (out-of-sample error), является основной величиной, которую мы используем для оценки эффективности систем машинного обучения.

Набор не предоставленных ранее примеров известен как *тестовый* (*отложенный*) *набор*, поскольку эти данные не привлекаются для обучения ИИ. Если мы решим использовать несколько тестовых наборов (возможно, с целью калибровки ошибки обобщения в процессе обучения, что рекомендуется делать), то у нас могут быть промежуточные наборы, которые применяются для оценки прогресса еще до того, как будет задействован финальный тестовый набор. Такие промежуточные наборы называются *валидационными*.

Подведем итоги. Итак, ИИ обучается на тренировочном наборе данных (*опыт*) для снижения частоты ошибок (*производительность*) при маркировании спама (*задача*), а окончательным критерием успеха служит то, насколько хорошо опыт, приобретенный ИИ, обобщается на новые, еще не встречавшиеся данные (*ошибка обобщения*).

Обучение, основанное на наборе правил, и машинное обучение

Используя подход на основе правил, мы могли бы спроектировать фильтр с помощью явно сформулированных правил, помечающих как спам такие, например, сообщения электронной почты, в которых встречаются фразы наподобие "купите прямо сейчас" и т.п. Однако обеспечить долговременную поддержку подобной системы будет трудно, поскольку спамеры, изменив со

временем свою тактику, смогут обойти установленные нами правила фильтрации спама. Применяя систему на основе правил, мы будем вынуждены часто изменять их вручную, приспосабливаясь к новым обстоятельствам. К тому же настройка такой системы займет очень много времени. Подумайте только, сколько правил придется создать, чтобы все работало эффективно.

Вместо описанного подхода мы можем использовать машинное обучение, тренируя систему на наборе сообщений электронной почты и автоматически конструируя правила для пометки спама. Такая система обеспечит автоматическую подстройку с течением времени, а ее обучение и сопровождение обойдутся гораздо дешевле.

Если с отсевом спама, представляющим собой сравнительно простую задачу, мы еще могли бы справиться, задавая правила вручную, то во многих других случаях реализация такого подхода вообще невозможна. В качестве примера рассмотрим беспилотные автомобили. Представьте, какое количество правил нужно продумать, чтобы охватить все возможные ситуации, с которыми может столкнуться такой автомобиль. Решить подобную задачу практически нереально, если только не наделить автомобиль способностью обучаться и адаптироваться к окружению, основываясь на собственном опыте.

Системы машинного обучения также можно использовать в качестве инструмента для исследования данных с целью более глубокого понимания сути задачи, которую мы пытаемся решить. Так, в примере с электронной почтой можно изучить, какие слова или фразы чаще всего оказываются характерными признаками нежелательных сообщений, и использовать эту информацию для распознавания новых шаблонов спама.

Обучение с учителем и обучение без учителя

Существуют две основные методологии машинного обучения: *обучение с учителем* (supervised learning), или *контролируемое обучение*, и *обучение без учителя* (unsupervised learning), или *неконтролируемое обучение* (*самообучение*). Есть еще множество методик, которые являются своеобразными мостиками между ними.

При обучении с учителем интеллектуальный агент имеет доступ к *меткам*, или *маркерам*, которые могут быть использованы для улучшения производительности в ряде задач. В задаче фильтрации спама у нас имеется набор сообщений электронной почты с полными текстами каждого из них. Нам также известно (посредством меток), какие сообщения являются спамом, а

какие — нет. Метки ценны тем, что они помогают ИИ отделять спам от остальных сообщений при обучении с учителем.

В случае обучения без учителя метки отсутствуют. Поэтому задача ИИ не является четко определенной, что осложняет точное измерение эффективности обучения. Вновь обратимся к задаче фильтрации спама, но на этот раз без использования меток. Теперь интеллектуальный агент будет пытаться понять базовую структуру электронных сообщений, разбивая данные на группы, в каждой из которых сообщения сходны между собой, но отличаются от сообщений из других групп.

Задача обучения без учителя формулируется менее четко по сравнению с обучением с учителем, и интеллектуальному агенту труднее ее решать, и в то же время хорошо продуманный план действий позволяет получать более мощные решения. Это обусловлено следующим обстоятельством. ИИ может найти несколько групп, которые он пометит как "спам", но при этом могут быть обнаружены также группы, которые впоследствии будут помечены как "важное" или категоризированы как "семья", "работа", "новости", "покупки" и т.п. Другими словами, поскольку задача не задана строго, интеллектуальный агент может обнаружить новые интересные закономерности помимо тех, которые мы первоначально пытались найти.

Более того, система обучения без учителя лучше справляется с обнаружением новых закономерностей в предоставляемых ей неизвестных данных, чем система, основанная на обучении с учителем, что делает ее в перспективе куда более эффективной. В этом и заключается преимущество обучения без учителя.

Сильные и слабые стороны обучения с учителем

Обучение с учителем отлично справляется с оптимизацией в случае достаточно четко определенных задач с множеством меток. Предположим, имеется очень большой набор изображений, каждое из которых снабжено меткой. Если этот набор достаточно велик, а тренировка осуществляется с использованием алгоритмов машинного обучения (например, с помощью сверточных нейронных сетей) на достаточно мощных компьютерах, то мы сможем получить весьма неплохую систему классификации изображений на основе обучения с учителем.

Поскольку в обучении с учителем ИИ тренируется на данных, он будет способен измерить свою эффективность (посредством фупкции потерь), сравнивая предсказанную метку изображения с истинной, которая хранится в файле. ИИ будет пытаться минимизировать функцию потерь таким образом,

чтобы ошибка классификации изображений, которые еще не предоставлялись системе (например, изображений из тестового набора), была как можно меньшей.

Вот почему метки играют столь важную роль — они обеспечивают возможность измерения ошибки интеллектуальным агентом. ИИ использует эту информацию для повышения производительности с течением времени. В отсутствие меток ИИ не будет знать, насколько успешно (или неуспешно) он справляется с классификацией изображений.

Однако стоимость ручной разметки изображений довольно высокая. Даже предварительно подготовленные наборы данных содержат всего несколько тысяч меток. Это становится источником проблем, поскольку системы обучения хорошо справляются с классификацией изображений, для которых имеются метки, и плохо — с классификацией тех изображений, для которых метки отсутствуют.

Какими бы мощными ни были системы обучения с учителем, они ограничены в своих возможностях обобщения полученных знаний на изображения помимо тех, которые были включены в размеченный тренировочный набор. Поскольку большая часть доступных в мире данных не размечена, в случае обучения с учителем возможности ИИ по эффективному использованию приобретенного опыта применительно к новым данным довольно ограничены.

Иначе говоря, обучение с учителем отлично подходит для решения задач слабого ИИ, но хуже справляется с решением более амбициозных, но менее четко заданных задач ИИ сильного типа.

Сильные и слабые стороны обучения без учителя

Обучение с учителем превосходит обучение без учителя при решении четко сформулированных задач, для которых имеются четко определенные шаблоны, не сильно изменяющиеся с течением времени, и при условии, что у нас есть доступ к достаточно большим наборам размеченных данных.

Но в тех случаях, когда шаблоны неизвестны, постоянно меняются или же мы не имеем доступа к достаточно большим наборам размеченных данных, на помощь приходит обучение без учителя.

Вместо того чтобы руководствоваться метками, система обучения без учителя изучает базовую структуру данных, используемых в процессе тренировки модели. Это достигается за счет попыток представить тренировочные данные с помощью набора параметров, размер которого значительно меньше количества примеров, доступных в наборе данных. Благодаря этому

обучение без учителя позволяет идентифицировать различные шаблоны в наборе данных.

В примере с набором изображений (на этот раз не снабженных метками) ИИ может идентифицировать и группировать изображения, исходя из того, насколько они схожи между собой и отличаются от остальных изображений. Например, будут объединены в отдельные группы все изображения, похожие на стул, все изображения, похожие на собаку, и т.п.

Разумеется, ИИ не пометит эти группы как "стулья" или "собаки", но теперь, когда сходные изображения сгруппированы, человеку будет намного проще расставить метки. Вместо того чтобы помечать миллионы изображений вручную, специалисты могут вручную присваивать метки различным группам, после чего эти метки будут автоматически применены ко всем элементам группы.

Если по завершении начальной тренировки интеллектуальный агент обнаружит изображения, которые не принадлежат ни к одной из помеченных групп, то он создаст отдельные группы для таких неклассифицированных изображений, переложив на человека задачу последующей разметки новых групп.

Обучение без учителя превращает задачи, которые ранее не удавалось решать, в такие, которые допускают возможность их решения, и позволяет намного быстрее находить скрытые закономерности (шаблоны поведения) в исторических данных, как доступных в процессе тренировки, так и будущих, с которыми система до сих пор не сталкивалась. Более того, тем самым ИИ прокладывает путь к обработке огромных хранилищ неразмеченных данных, существующих в мире.

Несмотря на то что обучение без учителя менее пригодно для решения специфических задач со строгой формулировкой по сравнению с обучением с учителем, оно лучше справляется с "размытыми" задачами, стоящими перед сильным ИИ, и обобщением приобретенных знаний.

Также немаловажен тот факт, что обучение без учителя способно оказать помощь при решении многих обычных задач, с которыми часто приходится сталкиваться аналитикам при создании приложений машинного обучения.

Использование обучения без учителя для улучшения систем машинного обучения

Недавние успехи в области машинного обучения обусловлены расширением возможностей доступа к большим объемам данных, резким повышением вычислительных мощностей, появлением облачных ресурсов, а также прорывом в разработке соответствующих алгоритмов. Но эти успехи главным образом достигнуты при решении задач узкого ИИ, таких как классификация изображений, компьютерное зрение, распознавание речи, обработка естественного языка и машинный перевод.

Решение более амбициозных задач ИИ требует привлечения обучения без учителя. Рассмотрим наиболее распространенные проблемы, с которыми приходится сталкиваться исследователям при построении приложений, и попытаемся понять, какую помощь в этом может оказать обучение без учителя.

Нехватка размеченных данных

> Построение ИИ можно сравнить с созданием космического корабля. Располагая мощным двигателем, но мизерным запасом топлива, вы не сможете вывести ракету на орбиту. В то же время ракета с тоннами топлива, но маломощным двигателем не сможет оторваться от земли. Чтобы запустить ракету в космос, необходим мощный двигатель и достаточно большой запас топлива.
>
> *Эндрю Ын*

Если провести аналогию между машинным обучением и космическим кораблем, то данные можно уподобить топливу — без них наш аппарат не смог бы взлететь. Но не все данные равноценны. Чтобы использовать алгоритмы обучения с учителем, нам требуется множество размеченных данных, генерирование которых — трудоемкий и затратный процесс.

В случае обучения без учителя мы можем автоматически помечать неразмеченные образцы. Вот как это делается. Мы кластеризуем все образцы, а затем применяем метки из размеченных примеров к неразмеченным, принадлежащим к одному кластеру. Неразмеченные образцы получат метки тех образцов, с которыми они имеют наибольшее сходство. (Кластеризация подробно исследуется в главе 5.)

Переобучение

Если тренировочные данные используются для того, чтобы обучить алгоритм машинного обучения чрезмерно сложной функции, то он может плохо работать на примерах, которые ему еще не предоставлялись, скажем, на примерах из наборов, зарезервированных в качестве валидационных или тестовых. В этом случае в результате извлечения чересчур большого объема информации из шума, содержащегося в данных, велика вероятность проявления эффектов *переобучения* (overfitting), приводящих к значительному ухудшению обобщающей способности модели. Иными словами, алгоритм *запоминает* обучающие данные, а не учится тому, как обобщать приобретенные знания на другие случаи[1].

Для преодоления этой проблемы можно использовать обучение без учителя в качестве регуляризатора. *Регуляризация* — это процесс снижения сложности алгоритма машинного обучения, способствующий извлечению сигнала из зашумленных данных. Одной из форм регуляризации служит предварительное обучение без учителя. Вместо того чтобы передавать оригинальные входные данные алгоритму обучения с учителем, мы можем генерировать их новое представление, которое и будет передаваться алгоритму.

Это новое представление захватывает наиболее существенную составляющую исходных данных — их истинную базовую структуру, одновременно избавляя нас от не содержащего значимой информации шума. Алгоритму обучения с учителем, которому передается новое представление, будет легче справляться с оставшимися шумами и извлекать полезный сигнал, что приведет к улучшению его обобщающей способности. (Более подробному обсуждению этой темы посвящена глава 7.)

Проклятие размерности

Несмотря на невиданный рост доступных вычислительных мощностей обработка больших данных все еще доставляет трудности алгоритмам машинного обучения. Вообще говоря, добавление большего количества примеров не

[1] Другая проблема, с которой можно столкнуться в приложениях машинного обучения, — *недообучение*, но она решается легче. Причиной недообучения становится излишняя простота модели — алгоритму не удается построить достаточно сложную аппроксимирующую функцию, позволяющую принимать решения, которые хорошо подходили бы для текущей задачи. Избавиться от этой проблемы можно либо за счет расширения алгоритма (путем ввода дополнительных параметров, увеличения количества итераций обучения и т.п.), либо за счет применения более сложного алгоритма обучения.

является проблемой, поскольку, используя такие современные модели распределенных вычислений, как Spark, мы можем распараллеливать выполнение операций. В то же время, чем больше имеется признаков, тем более трудоемким становится обучение.

В случае пространств очень большой размерности алгоритмы обучения с учителем должны обучаться тому, как разделять точки и строить функцию-аппроксиматор, обеспечивающую принятие правильных решений. Если признаков очень много, то эта процедура оказывается дорогостоящей как в отношении времени, так и в отношении требуемых вычислительных ресурсов. В некоторых случаях данный фактор вообще делает невозможным получение достаточно эффективных и быстрых решений.

Эта проблема известна как *проклятие размерности*, и для ее решения хорошо подходит обучение без учителя. Используя инструменты снижения размерности, мы можем найти наиболее значимые признаки в исходном наборе и снизить, пусть даже за счет потери незначительной части важной информации, число измерений до приемлемого уровня, а затем применить алгоритмы обучения с учителем для более эффективного поиска подходящей аппроксимирующей функции. (Снижение размерности обсуждается в главе 3.)

Конструирование признаков

Конструирование признаков (feature engineering) — одна из самых важных задач, которые должны решать исследователи. Не располагая достоверными признаками, алгоритм машинного обучения окажется неспособным достаточно надежно разделять точки в пространстве для того, чтобы принимать корректные решения относительно незнакомых ему примеров. Однако, как правило, конструирование признаков — весьма трудоемкая задача, требующая креативного вмешательства человека для создания признаков нужного типа в ручном режиме. Вместо этого мы можем использовать представление, полученное с помощью алгоритмов обучения без учителя, для автоматического обучения признакам подходящего типа, помогающим решить конкретную задачу. (Автоматическое извлечение признаков обсуждается в главе 7.)

Выбросы

Важную роль играет также качество данных. Если алгоритмы машинного обучения тренируются на данных, содержащих редкие *выбросы*, которые несколько искажают истинную картину, то ошибка обобщения будет меньше, чем если бы мы игнорировали их или обрабатывали отдельно. В случае

обучения без учителя мы можем обнаруживать выбросы, используя механизмы снижения размерности, и создать два приложения: одно, специально предназначенное для обработки выбросов, и другое, предназначенное для обработки нормальных данных. (Построением системы обнаружения аномалий мы займемся в главе 4.)

Дрейф данных

Модели машинного обучения также должны считаться с *дрейфом данных* (data drift). Если данные, которые используются моделью в целях прогнозирования, статистически отличаются от данных, на которых она обучалась, то может потребоваться повторно обучить ее на тех данных, которые лучше соответствуют текущей задаче. Если модель не подвергается повторной тренировке или ей не удается распознать дрейф, то ее предсказательная способность на текущих данных от этого пострадает.

Создавая распределения вероятностей с помощью обучения без учителя, мы можем оценить, насколько текущие данные отличаются от данных тренировочного набора. И если эти различия существенны, то мы можем автоматически запускать повторную тренировку модели. (Построению такого типа систем посвящена глава 12.)

Краткий обзор алгоритмов машинного обучения с учителем

Прежде чем углубиться в изучение систем обучения без учителя, целесообразно кратко ознакомиться с алгоритмами обучения с учителем и тем, как они работают. Это поможет нам очертить границы, в которые вписывается обучение без учителя.

Существуют два основных типа задач, относящихся к сфере обучения с учителем: *классификация* и *регрессия*. В случае классификации ИИ должен корректно относить элементы к одному из двух или нескольких классов. Если есть всего два класса, то мы имеем дело с так называемой *бинарной классификацией*. Если же классов три или более, то задачу такого рода называют *многоклассовой* (*мультиклассовой*) *классификацией*.

Задачи классификации также называются *дискретными предсказаниями*, поскольку каждый класс образует отдельную группу. Другое их название — *качественный*, или *категориальный*, анализ.

В случае регрессии ИИ должен предсказывать значения *непрерывной* переменной, а не дискретной. Другое название регрессионных задач — *количественный анализ.*

Алгоритмы машинного обучения с учителем расширяют этот спектр задач, но все они нацелены на минимизацию некоторой функции потерь (либо максимизации функции значения), которая ассоциирована с метками, имеющимися в наборе данных.

Как уже упоминалось ранее, больше всего нас должно заботить то, насколько хорошо система машинного обучения обобщается на ранее не предоставленные образцы. Для минимизации ошибки обобщения очень важно сделать правильный выбор алгоритма обучения с учителем.

Для обеспечения как можно меньшей ошибки обобщения необходимо, чтобы сложность алгоритмической модели соответствовала сложности истинной функции, заложенной в данных. Мы не знаем, что в действительности представляет собой эта функция. Если бы это было известно, то нам не потребовалось бы задействовать машинное обучение для создания модели — мы могли бы просто использовать эту функцию для получения корректного ответа. Но поскольку истинная функция неизвестна, мы выбираем алгоритм машинного обучения для тестирования гипотез и находим модель, которая наилучшим образом аппроксимирует истинную функцию (т.е. приводит к наименьшей возможной ошибке обобщения).

Если сложность того, что моделирует алгоритм, меньше сложности истинной функции, то мы столкнемся с проблемой *недообучения* (underfitting). В таком случае ошибку обобщения можно уменьшить посредством выбора алгоритма, способного моделировать более сложную функцию. Но если алгоритм порождает чересчур сложную модель, то мы столкнемся с *переобучением* (overfitting) и получим плохие результаты для ранее не встречавшихся данных, увеличив ошибку обобщения.

Другими словами, отдавать предпочтение более сложным алгоритмам по сравнению с более простыми не всегда будет правильным выбором — иногда простое лучше сложного. Каждому алгоритму свойствен свой набор сильных и слабых сторон, а также допущений, и знание того, какие из них следует применять для решения конкретной задачи с имеющимися данными, является очень важным условием овладения искусством машинного обучения на профессиональном уровне.

Далее мы рассмотрим наиболее популярные алгоритмы машинного обучения с учителем (включая их практические применения), а затем — алгоритмы обучения без учителя[2].

Линейные методы

Самые простые алгоритмы обучения с учителем моделируют линейное соотношение между входными признаками и выходной переменной, значение которой мы хотим предсказать.

Линейная регрессия

Простейшим из всех алгоритмов является *линейная регрессия*, в которой используется модель, предполагающая линейное соотношение между входными переменными (X) и единственным выходным значением (y). Если соотношение между входами и выходом действительно линейное, а между входными переменными отсутствует значительная корреляция (ситуация, известная как *коллинеарность*), то линейная регрессия станет неплохим выбором. Если же истинное соотношение носит более сложный или нелинейный характер, то использование линейной регрессии приведет к недообучению[3].

Учитывая простоту самого алгоритма, интерпретация моделируемого им соотношения также не представляет никакого труда. *Интерпретируемость* результатов — очень важный фактор машинного обучения, поскольку решение должно быть понято и воплощено в жизнь как техническими, так и нетехническими отраслевыми специалистами. В отсутствие возможности интерпретации решения оно превращается в загадочный черный ящик.

Сильные стороны

Линейная регрессия отличается простотой, интерпретируемостью и несклонностью к переобучению, поскольку она не в состоянии моделировать чрезмерно сложные отношения. Она великолепно подходит для тех случаев, когда соотношение между входными и выходными переменными в основном является линейным.

[2] Этот список не является исчерпывающим, но включает большинство алгоритмов, которые обычно используются в машинном обучении.

[3] К числу других потенциальных проблем, делающих линейную регрессию неудачным вариантом выбора, относятся выбросы, а также наличие корреляции между составляющими ошибки и непостоянство их дисперсии.

Слабые стороны

Линейная регрессия будет недообучаться на данных, если соотношение между входными и выходными переменными нелинейное.

Применение

Поскольку вес человека в целом линейно зависит от его роста, линейная регрессия будет хорошо предсказывать вес человека, используя его рост в качестве входных данных, и наоборот.

Логистическая регрессия

Простейшим алгоритмом классификации является *логистическая регрессия*, которая также относится к линейным методам, но ее предсказания преобразуются с помощью логистической функции. Выходом такого преобразования являются *вероятности классов* — иначе говоря, вероятности принадлежности образца к тому или иному классу, причем сумма всех вероятностей по всем образцам должна равняться единице. Затем каждому примеру приписывается класс, вероятность принадлежности к которому максимальная.

Сильные стороны

Как и линейная регрессия, логистическая регрессия отличается простотой и интерпретируемостью. Если классы, которые мы пытаемся предсказать, не перекрываются и линейно разделимы, то логистическая регрессия — отличный выбор.

Слабые стороны

Логистическая регрессия не работает в случае линейно неразделимых классов.

Применение

Если классы — например, рост ребенка и рост взрослого человека — в целом не перекрываются, то логистическая регрессия даст хорошие результаты.

Методы на основе соседства точек

Другую группу очень простых алгоритмов образуют методы, основанные на понятии *соседства точек* (neighborhood). Методы этой группы — *ленивые ученики*, поскольку в данном случае обучение разметке новых точек основывается на их близости к уже размеченным точкам. В отличие от линейной или

логистической регрессии модели, основанные на соседстве точек, не обучаются предсказанию меток для новых точек. Вместо этого они предсказывают метки для новых точек, исключительно исходя из того, насколько новые точки удалены от существующих размеченных точек. *Ленивое обучение* (lazy learning) также относится к методам *обучения на примерах* (instance-based learning), или *непараметрическим* методам.

Метод *k*-ближайших соседей

Наиболее популярный метод этой категории — *k-ближайших соседей* (k-nearest neighbors — KNN). Для пометки каждой новой точки алгоритм осуществляет поиск k ближайших соседних помеченных точек (где k — целое число) и наделяет уже размеченных соседей правом голоса при принятии решения относительно того, какую метку следует присвоить новой точке. В качестве меры удаленности новой точки от ближайших соседей в KNN по умолчанию используется евклидово расстояние.

От выбора значения k зависит очень многое. Если оно установлено слишком низким, то метод приобретает повышенную гибкость, очерчивая границы, учитывающие множество нюансов, и переобучаясь на данных. Если же для k выбрано слишком большое значение, то метод теряет гибкость, очерчивая слишком грубые границы и потенциально недообучаясь на данных.

Сильные стороны

KNN, в отличие от линейных методов, обладает значительной гибкостью и приспособлен для обучения более сложным нелинейным соотношениям. При этом он сохраняет простоту и интерпретируемость.

Слабые стороны

KNN плохо работает в тех случаях, когда количество наблюдений и признаков достигает критической величины. В условиях заполненного большим количеством точек многомерного пространства метод становится неэффективным с вычислительной точки зрения, поскольку для предсказания признаков приходится рассчитывать расстояния от новой точки до уже размеченных многочисленных соседних точек. Он не позволяет использовать эффективную модель с уменьшенным количеством параметров для генерирования необходимых предсказаний. Кроме того, метод весьма чувствителен к выбору параметра k. При слишком низких значениях k метод может переобучаться, а при слишком высоких — недообучаться.

Применение

KNN широко применяется в рекомендательных системах, например тех, которые используются для прогнозирования пользовательских предпочтений: фильмов (Netflix), музыки (Spotify), друзей (Facebook), фотографий (Instagram), поисковых запросов (Google) и потребительских товаров (Amazon). В частности, метод может оказаться полезным при предсказании предпочтений отдельного пользователя на основании известных предпочтений пользователей с аналогичными вкусами (так называемая *коллаборативная фильтрация*) или на основании собственных предпочтений пользователя в прошлом (так называемая *фильтрация по содержимому*).

Методы на основе деревьев решений

Вместо того чтобы использовать линейный метод, мы можем построить дерево решений, в котором все примеры *сегментируются*, или *стратифицируются*, по отдельным областям на основании имеющихся меток. По завершении сегментирования каждая область соответствует определенному классу меток (для задач классификации) или диапазону предсказанных значений (для задач регрессии). Этот процесс аналогичен тому, как если бы мы поручили ИИ автоматически создать правила, ориентированные на получение наилучших решений или предсказаний.

Одиночное дерево решений

Простейший вариант — метод *одиночного дерева решений*, в котором ИИ выполняет один проход по тренировочным данным, создает правила для сегментирования данных на основании имеющихся меток и использует результирующее дерево решений для создания предсказаний в отношении не представленных ранее данных, включенных в валидационный или тестовый набор. Однако одиночное дерево решений обычно плохо обобщает то, чему обучилось в процессе тренировки, на незнакомые примеры ввиду переобучения на одной-единственной тренировочной итерации.

Бэггинг

В качестве улучшения одиночного дерева решений можно воспользоваться *бутстрэп-агрегированием*, или *бэггингом*, когда мы формируем из тренировочных данных *несколько случайных выборок примеров*, а затем создаем дерево решений для каждой выборки и предсказываем выход для каждого примера

путем усреднения предсказаний отдельных деревьев. За счет использования *рандомизации* выборок и усреднения результатов нескольких деревьев (подход, известный как *метод ансамблей*) бэггинг помогает частично справиться с переобучением, свойственным одиночному дереву решений.

Случайные леса

Мы можем дополнительно ослабить эффекты переобучения, семплируя не только примеры, но и сами предикторы. В методе *случайных лесов* мы отбираем несколько случайных выборок примеров из тренировочных данных, как это делается в бэггинге, но в каждом дереве решений мы создаем расщепление, основанное не на всех предикторах, а на *случайной выборке предикторов*. Количество предикторов, устанавливаемых для каждого дерева, обычно выбирается равным квадратному корню из общего их количества.

Семплируя предикторы описанным способом, алгоритм случайных лесов создает деревья, еще в меньшей степени скоррелированные между собой (по сравнению с деревьями бэггинга), тем самым ослабляя эффекты переобучения и уменьшая ошибку обобщения.

Бустинг

В другом подходе, который называется *бустинг*, также создается несколько деревьев, но при этом деревья формируются *последовательно*, что позволяет использовать знания, которым ИИ обучился на предыдущем дереве, для улучшения результатов, получаемых с помощью следующего дерева. Глубина каждого дерева поддерживается на довольно мелком уровне, с небольшим количеством расщеплений, и обучение происходит последовательно, дерево за деревом. Из всех методов, основанных на деревьях решений, наилучшую производительность продемонстрировали *машины градиентного бустинга*, которые часто одерживают победы на соревнованиях по машинному обучению[4].

Сильные стороны

Методы на основе деревьев решений относятся к числу наиболее производительных алгоритмов обучения с учителем, предназначенных для решения задач прогнозирования. Эти методы способны выявлять сложные соотношения в данных путем обучения многим простым правилам,

[4] Подробнее о градиентном бустинге можно прочитать в блоге Бена Гормана (`http://bit.ly/2S1C8Qy`).

по одному правилу за раз. Кроме того, они способны обрабатывать отсутствующие данные и категориальные признаки.

Слабые стороны

Методы на основе деревьев решений с трудом поддаются интерпретации, особенно в тех случаях, когда для получения надежных предсказаний требуется использовать большое количество признаков. Кроме того, по мере увеличения количества признаков производительность также становится проблемой.

Применение

Градиентный бустинг и случайные леса отлично подходят для решения задач прогнозирования.

Метод опорных векторов

Вместо того чтобы строить деревья для разделения данных, мы можем использовать алгоритмы, предназначенные для создания гиперплоскостей, разделяющих данные на основании имеющихся признаков. Соответствующий подход получил название *метод опорных векторов* (support vector machine — SVM). SVM не гарантирует идеального разделения (не все точки в определенной области гиперпространства обязаны иметь одну и ту же метку), но расстояние между пограничными точками, с которыми ассоциирована некоторая метка, и пограничными точками, с которыми ассоциирована другая метка, должны быть как можно большими. Кроме того, границы не обязаны быть линейными — мы можем обеспечить более гибкое разделение данных, используя нелинейные ядра.

Нейронные сети

Обучение представлениям данных можно проводить с использованием *нейронных сетей*, которые состоят из входного слоя, нескольких скрытых слоев и выходного слоя[5]. Входной слой использует признаки, тогда как выходной слой пытается добиться соответствия переменной отклика (ответной переменной). Скрытые слои представляют собой вложенную иерархию

[5] Для получения более подробной информации о нейронных сетях обратитесь к книге *Глубокое обучение* Яна Гудфеллоу, Иошуа Бенджио и Аарона Курвилля (`http://www.deeplearningbook.org/`).

абстрактных понятий — каждый слой (или понятие) пытается понять, как предыдущий слой соотносится с выходным слоем.

Опираясь на эту иерархию, нейронная сеть может обучаться сложным понятиям путем их формирования на основе более простых понятий. Нейронные сети — один из наиболее мощных подходов к аппроксимации функций, однако с ним сопряжены такие проблемы, как переобучение и трудность интерпретации — недостатки, которые мы будем подробно обсуждать далее.

Краткий обзор алгоритмов машинного обучения без учителя

А теперь переключимся на рассмотрение задач, в которых мы имеем дело с неразмеченными данными. Применяемые в таких случаях алгоритмы обучения без учителя не создают предсказания, а пытаются обучиться базовой структуре данных.

Снижение размерности

Алгоритмы снижения размерности транслируют оригинальное многомерное пространство входных данных в пространство более низкой размерности, фильтруя наименее релевантные признаки и сохраняя как можно большее количество признаков, представляющих интерес. Снижение размерности позволяет эффективнее выявлять шаблоны и решать крупномасштабные, вычислительно трудоемкие задачи (чаще всего связанные с обработкой изображений, видео, речи и текста).

Линейная проекция

Существуют две основные разновидности алгоритмов снижения размерности: *линейная проекция* и *нелинейное снижение размерности*.

Анализ главных компонент

Один из подходов к изучению базовой структуры данных заключается в идентификации наиболее важных признаков, объясняющих изменчивость примеров. Не все признаки равноценны. Одни из них испытывают лишь незначительные изменения и поэтому менее полезны в плане объяснения данных. В то же время другие признаки могут варьироваться в определенных пределах, и именно они заслуживают более пристального внимания, поскольку с их помощью легче построить модель, предназначенную для разделения данных.

В методе PCA (principal component analysis — анализ главных компонент) алгоритм находит такое представление данных низкой размерности, которое обеспечивает сохранение максимально возможной доли их вариативности. Размерность этого представления значительно меньше размерности полного набора данных (т.е. полного числа признаков). Переход к пространству пониженной размерности сопровождается потерей определенной части дисперсии, но упрощает идентификацию базовой структуры данных, что обеспечивает более эффективное решение таких задач, как кластеризация.

Существует несколько вариантов метода PCA, которые мы более подробно обсудим в последующих главах. В их число входят мини-пакетные версии, такие как *инкрементный PCA*, нелинейные версии, такие как *ядерный PCA*, и разреженные версии, такие как *разреженный PCA*.

Сингулярное разложение

Суть другого подхода к изучению базовой структуры данных заключается в снижении ранга исходной матрицы признаков до меньшего значения с сохранением возможности восстановления исходной матрицы путем образования линейной комбинации некоторых из векторов полученпой матрицы более низкого ранга. Это и есть метод SVD (singular value decomposition — сингулярное разложение). Генерируя матрицу меньшего ранга, метод SVD оставляет те из векторов исходной матрицы, которые содержат наибольшую часть информации (обеспечивают наибольшее сингулярное значение). Эта матрица пониженного ранга захватывает наиболее важные элементы исходного пространства признаков.

Случайная проекция

Аналогичный алгоритм снижения размерности включает проецирование точек из многомерного пространства в пространство более низкой размерности таким образом, чтобы сохранить масштаб расстояний между точками. Для этого используют либо *случайную гауссовскую* либо *случайную разреженную* матрицу.

Обучение на многообразиях

Как PCA, так и случайная проекция подразумевают линейное проецирование данных из многомерного пространства в пространство низкой размерности. Результаты можно улучшить за счет применения нелинейного преобразования вместо линейного — такой подход известен как *обучение на многообразиях* (manifold learning), или *нелинейное снижение размерности*.

Isomap

Этот алгоритм обучается внутренней геометрии данных путем оценки *геодезических* (*искривленных*) *расстояний* между каждой точкой и ее ближайшими соседями, а не евклидовых. Isomap использует эту информацию для последующего вложения многомерного пространства в пространство пониженной размерности.

Стохастическое вложение соседей с t-распределением

Другой подход к снижению размерности, называемый *t-SNE* (t-distributed stochastic neighbor embedding), основан на вложении многомерных данных в пространство, имеющее всего лишь два-три измерения, что позволяет визуализировать преобразованные данные. В этом двух- или трехмерном пространстве схожие примеры располагаются на небольших, а несхожие — на больших расстояниях друг от друга.

Словарное обучение

Подход, известный как *словарное обучение*, подразумевает обучение разреженному представлению базовых данных. Этими репрезентативными элементами служат простые бинарные векторы (состоящие из нулей и единиц), и каждый пример в наборе данных может быть реконструирован в виде взвешенной суммы репрезентативных элементов. Матрица (*словарь*), генерируемая в процессе обучения без учителя, заполнена главным образом пулями и содержит лишь небольшое количество ненулевых весов.

Благодаря созданию подобного словаря алгоритм способен эффективно идентифицировать наиболее существенные репрезентативные элементы исходного пространства признаков — те, которые содержат наибольшее количество ненулевых весов. Менее важные элементы содержат меньшее количество ненулевых весов. Как и PCA, словарное обучение отлично подходит для разделения данных и идентификации интересующих нас шаблонов.

Анализ независимых компонент

Одной из общих проблем, порождаемых неразмеченными данными, становится наличие множества независимых сигналов, которые вложены в имеющиеся в нашем распоряжении признаки. Используя *анализ независимых компонент* (independent component analysis — ICA), мы можем разделить эту смесь сигналов на независимые компоненты. После такого разделения можно реконструировать любой из оригинальных признаков, образуя линейные комбинации сгенерированных индивидуальных компонент. Метод ICA широко

применяется для обработки сигналов (например, для идентификации отдельных голосов в аудиозаписи из шумного кафетерия).

Латентное размещение Дирихле

Обучение без учителя также может быть использовано для объяснения набора данных путем изучения факторов, которые обусловливают сходство отдельных частей набора между собой. Это требует обучения ненаблюдаемым элементам набора данных — подход, получивший название *латентное размещение Дирихле* (latent Dirichlet allocation — LDA). Предположим, имеется текстовый документ, содержащий множество слов. Эти слова не являются случайными, они подчиняются определенной структуре.

Такую структуру можно смоделировать как ненаблюдаемые элементы — так называемые *темы*. После проведения соответствующей тренировки метод LDA может объяснить данный документ с помощью ограниченного набора тем, для каждой из которых имеется небольшой набор часто используемых слов. Это и есть скрытая структура, которую LDA способен захватывать, что позволяет лучше объяснять ранее неструктурированный текстовый корпус.

Снижение размерности сводит оригинальный набор признаков к меньшему набору, включающему лишь наиболее важные признаки. Далее мы можем запускать другие алгоритмы обучения без учителя на этом меньшем наборе признаков с целью поиска шаблонов, представляющих интерес (см. следующий раздел, посвященный кластеризации), или, при наличии меток, для ускорения цикла обучения алгоритмов обучения с учителем путем передачи им меньшей матрицы признаков вместо использования оригинальной матрицы.

Кластеризация

После редуцирования набора оригинальных признаков до набора меньшего размера мы можем приступить к поиску интересующих нас закономерностей путем группирования схожих примеров. Этот процесс, называемый *кластеризацией*, можно реализовать с помощью целого ряда алгоритмов обучения без учителя и использовать в таких задачах, как сегментирование рынка.

Метод *k*-средних

Чтобы успешно справиться с кластеризацией, мы должны идентифицировать отдельные группы, примеры в пределах которых схожи между собой, но

отличаются от примеров, относящихся к другим группам. Одним из таких алгоритмов является *кластеризация методом* k-*средних* (*k*-means clustering). В данном алгоритме мы задаем желаемое количество кластеров, *k*, и алгоритм относит каждый пример ровно к одному из этих *k* кластеров. Алгоритм оптимизирует группирование, минимизируя внутрикластерную вариацию (называемую *инерцией*) так, чтобы сумма внутрикластерных вариаций по всем *k* кластерам была как можно меньшей.

С целью ускорения процесса кластеризации метод *k*-средних случайным образом относит каждое наблюдение к одному из *k* кластеров, а затем переназначает наблюдения для минимизации евклидовых расстояний между каждым наблюдением и центральной точкой кластера, или *центроидом*. Как следствие, различные запуски алгоритма *k*-средних — каждый со своей начальной точкой — будут приводить к несколько различающимся отнесениям наблюдений к тем или иным кластерам. Из этой серии различных запусков мы выбираем тот, который характеризуется наилучшим разделением, дающим наименьшую общую сумму внутрикластерных вариаций по всем *k* кластерам[6].

Иерархическая кластеризация

Альтернативой обычной кластеризации служит так называемая *иерархическая кластеризация*, не требующая предварительного задания количества кластеров. В одном из вариантов иерархической кластеризации, известном как *агломеративная кластеризация*, применяется кластеризация на основе деревьев и создается так называемая *дендрограмма*. Последняя графически отображается в виде перевернутого дерева, в котором листья находятся внизу, а ствол дерева — вверху.

Самые нижние листья — это индивидуальные примеры, содержащиеся в наборе данных. Затем, по мере перемещения вверх по перевернутому дереву, иерархическая кластеризация объединяет листья на основании их взаимного сходства. В первую очередь объединяются наиболее схожие примеры (или группы примеров), в последнюю очередь — наименее схожие. В конечном счете описанный итеративный процесс приводит к тому, что все примеры оказываются связанными, образуя единый ствол дерева.

Такое графическое представление весьма информативно. Как только алгоритм иерархической кластеризации завершит свою работу, мы сможем проанализировать дендрограмму и определить, в каком месте мы хотим обрезать дерево. Чем ниже линия обреза, тем больше останется индивидуальных

[6] Существуют более быстрые варианты кластеризации методом *k*-средних, которые мы обсудим в последующих главах.

ветвей (т.е. кластеров). Если мы хотим уменьшить количество кластеров, то линия обреза должна располагаться высоко на дендрограмме, ближе к стволу, находящемуся на самом верху перевернутого дерева. Размещение линии обреза аналогично выбору количества k кластеров в алгоритме кластеризации методом k-средних[7].

DBSCAN

DBSCAN (density-based spatial clustering of applications with noise — основанная на плотности пространственная кластеризация для приложений с шумами) — еще более мощный алгоритм кластеризации. DBSCAN группирует вместе тесно расположенные примеры. Теснота расположения определяется как минимальное количество примеров, взаимные расстояния между которыми не превышают заданной величины. При этом мы задаем как требуемое минимальное количество таких примеров, так и определенное расстояние, ограничивающее пределы близости.

Если пример находится в пределах указанного расстояния от нескольких кластеров, то он будет группироваться с тем из них, для которого теснота расположения оказывается большей. Примеры, расположенные в областях малой плотности, помечаются как выбросы.

В отличие от метода k-средних мы не должны предварительно задавать количество кластеров. Кластеры могут иметь произвольную форму. Метод DBSCAN гораздо меньше подвержен искажениям, которые обычно создаются выбросами.

Извлечение признаков

Обучение без учителя позволяет обучаться новым представлениям оригинальных признаков данных — это называется *извлечением признаков* (feature extraction). Данный подход может применяться для эффективного снижения размерности данных путем создания редуцированного подмножества оригинальных признаков. А кроме того, это позволяет генерировать новые признаки с целью повышения производительности в задачах обучения с учителем.

[7] В случае иерархической кластеризации по умолчанию используются евклидовы расстояния, но можно задействовать и другие аналогичные метрики, например *расстояние, исчисляемое на основе корреляции*, которое мы рассмотрим в последующих главах.

Автокодировщики

Для генерирования новых представлений признаков можно использовать нерекуррентную нейронную сеть прямого распространения, в которой количество узлов во входном и выходном слоях совпадает. Эта нейронная сеть, так называемый *автокодировщик* (autoencoder), способна эффективно реконструировать исходные признаки, обучаясь новому представлению с помощью промежуточных скрытых слоев[8].

Каждый скрытый слой автокодировщика обучается представлению оригинальных признаков, причем каждый последующий слой достраивает представление, которому обучились предыдущие слои. Благодаря этому автокодировщик обучается все более сложным представлениям на основе более простых.

На выходе автокодировщика мы получаем окончательное новое представление, которому он обучился. Затем это представление может быть использовано в качестве входа для модели обучения с учителем с целью уменьшения ошибки обобщения.

Извлечение признаков путем контролируемого обучения сетей прямого распространения

Если в нашем распоряжении имеются метки, то возможной альтернативой подхода, основанного на извлечении признаков, является использование нерекуррентной сети прямого распространения, в которой выходной слой пытается предсказать правильную метку. Как и в случае автокодировщиков, каждый скрытый слой обучается представлению оригинальных признаков.

Но когда генерируются новые представления, сеть работает непосредственно под управлением признаков. Для извлечения окончательного вновь изученного представления оригинальных признаков мы извлекаем предпоследний скрытый слой, непосредственно предшествующий выходному. Далее этот предпоследний слой можно использовать в качестве входа в любой модели обучения с учителем.

[8] Существует несколько разновидностей автокодировщиков, каждый из которых обучается отдельному типу представлений. В их число входят *обесшумливающие автокодировщики*, *разреженные автокодировщики* и *вариационные автокодировщики*. Все они будут подробно рассмотрены в последующих главах.

Глубокое обучение без учителя

Обучение без учителя находит целый ряд важных применений в технологии глубокого обучения; с некоторыми из них мы познакомимся в последующих главах. Соответствующая область исследований называется *глубоким обучением без учителя* (unsupervised deep learning).

Еще совсем недавно обучение глубоких сетей было практически неосуществимым из-за трудоемкости вычислений. В таких нейронных сетях скрытые слои обучаются внутренним представлениям для решения текущей задачи. Представления постепенно улучшаются за счет обновления весов различных узлов с использованием *градиента функции ошибки* на каждой итерации тренировки.

Подобное обновление весов требует интенсивных вычислений, в процессе которых могут возникать две основные трудности. Во-первых, градиент функции ошибки может уменьшиться до очень небольших значений, а поскольку обратное распространение ошибки основано на перемножении этих значений, веса сети будут обновляться очень медленно или вообще не обновляться, препятствуя ее обучению[9]. Это явление известно как *проблема затухающих градиентов*.

Суть другой проблемы, противоположной той, которая была только что описана, заключается в том, что градиент функции ошибки также может принимать очень большие значения. В таком случае веса сети могут обновляться с использованием огромных приращений, что делает процесс обновления крайне нестабильным. Это явление известно как *проблема взрывных градиентов*.

Предварительное обучение без учителя

Для преодоления описанных выше проблем, возникающих в процессе обучения очень глубоких, многослойных нейронных сетей, исследователи, занимающиеся машинным обучением, тренируют нейронные сети на протяжении нескольких последовательных этапов, каждый из которых включает мелкую нейронную сеть. Выход одной мелкой сети используется в качестве входа следующей нейронной сети. В типичных случаях первая мелкая сеть в этом конвейере включает неконтролируемую, т.е. обучаемую без учителя, сеть, тогда как более поздние сети обучаются с учителем.

[9] *Обратное распространение ошибки* (backpropagation) — это алгоритм градиентного спуска, применяемый в нейронных сетях для обновления весов. При этом сначала вычисляются веса последнего слоя, которые затем используются для обновления весов предыдущих слоев. Процесс продолжается до тех пор, пока не будут обновлены веса первого слоя.

Неконтролируемая часть сети известна как *жадное послойное предварительное обучение без учителя* (greedy layer-wise unsupervised pretraining). В 2006 году Джеффри Хинтон продемонстрировал успешное применение предварительного обучения без учителя для инициализации обучения конвейера глубокой нейронной сети, тем самым дав старт нынешней революции в области глубокого обучения. Предварительное обучение без учителя позволяет ИИ захватывать улучшенное представление оригинальных входных данных, преимуществами которого обучаемая с учителем часть сети может воспользоваться для решения текущей задачи.

Этот подход называют "жадным", поскольку каждая часть нейронной сети обучается независимо от других ее частей, а не совместно с ними. Для большинства современных нейронных сетей предварительное обучение обычно не требуется. Вместо этого все слои обучаются совместно посредством обратного распространения ошибки. Благодаря прогрессу в области компьютерных технологий справляться с проблемами затухающих и взрывных градиентов стало намного проще.

Предварительное обучение без учителя упростило решение не только задач обучения с учителем, но и задач *переносимого обучения* (transfer learning), в котором алгоритмы машинного обучения используются для сохранения знаний, полученных в процессе решения одной задачи, чтобы ускорить решение другой родственной задачи, причем на основе значительно меньшего объема данных.

Ограиичeнные машиины Больцмана

Одним из прикладных примеров предварительного обучения без учителя может служить *ограниченная машина Больцмана* (restricted Boltzmann machine — RBM), представляющая собой мелкую двухслойную нейронную сеть. Ее первым слоем является входной слой, а вторым — скрытый. Каждый узел связан с каждым узлом другого слоя, но узлы в пределах одного и того же слоя не связаны между собой — это и есть то ограничение, которое фигурирует в названии данного метода.

RBM способна решать такие задачи обучения без учителя, как снижение размерности и извлечение признаков, а кроме того, подходит для использования в качестве составной части систем обучения с учителем, обеспечивающей предварительное обучение без учителя. RBM аналогичны автокодировщикам, но отличаются от них в ряде важных аспектов. Например, если в автокодировщиках предусмотрен выходной слой, то в RBM он отсутствует. К обсуждению этого и других отличий мы вернемся в последующих главах.

Глубокие сети доверия

RBM могут связываться между собой, образуя многозвенную нейронную сеть — так называемую *глубокую сеть доверия* (deep belief network — DBN). Скрытый слой каждой RBM используется в качестве входа для следующей RBM. Другими словами, каждая RBM генерирует представление данных, от которого затем отталкивается очередная RBM. Связывая между собой последовательные этапы обучения представлениям такого типа, глубокая сеть доверия способна обучаться более сложным представлениям, которые часто используются в качестве *детекторов признаков*[10].

Генеративно-состязательные сети

Одним из главных достижений в области глубокого обучения без учителя стали *генеративно-состязательные сети* (generative adversarial network — GAN), разработанные Яном Гудфеллоу с коллегами из Монреальского университета в 2014 году. Генеративно-состязательные сети имеют многочисленные применения; например, их можно использовать для создания высокореалистичных синтетических данных, таких как изображения или речь, либо для обнаружения аномалий.

GAN состоит из двух нейронных сетей. Одна из них, называемая *генератором*, генерирует данные на основе модельного распределения, которое она создает, используя выборки из реально полученных данных. Другая сеть, называемая *дискриминатором*, пытается отличить данные, созданные генератором, от данных, подчиняющихся истинному распределению.

Если воспользоваться простой аналогией и отвести генератору роль фальшивомонетчика, то дискриминатор можно уподобить криминалисту, пытающемуся отличить подделку. Эти две сети вовлечены в *игру с нулевой суммой* (zero-sum game). Генератор пытается ввести дискриминатор в заблуждение, чтобы тот думал, будто синтетические данные постунают из источника с истинным распределением данных, тогда как дискриминатор пытается выявить, что данные в действительности синтетические.

Генеративно-состязательные сети — это алгоритмы обучения без учителя, поскольку генератор способен изучить базовую структуру истинного распределения данных, даже если они не снабжены метками. Генеративно-состяза-

[10] Детекторы признаков обучаются подходящим представлениям исходных данных, способствующим выделению различающихся элементов. Например, в случае изображений детекторы признаков облегчают выделение таких элементов, как нос, глаза, рот и т.п.

тельные сети достигают этого в процессе тренировки, эффективно выявляя структуру с использованием умеренного количества параметров.

Этот процесс аналогичен процессу обучения представлениям посредством глубокого обучения. Каждый скрытый слой нейронной сети генератора захватывает представление базовых данных. Первоначальные представления очень простые, но каждый последующий слой усложняет представление, полученное в предыдущем слое.

Совместно используя все эти слои, генератор обучается базовой структуре данных и, опираясь на приобретенный опыт, пытается создать синтетические данные, которые были бы почти идентичны настоящим. Если генератору удается уловить суть истинного распределения данных, то синтетические данные будут казаться реальными.

Обработка последовательных данных с помощью обучения без учителя

Обучение без учителя позволяет также обрабатывать последовательные данные, например временны*е* ряды. Один из таких подходов предполагает обучение скрытым состояниям *марковской модели*. В *простой марковской модели* состояния полностью наблюдаемы и изменяются стохастически (другими словами — случайным образом). Будущие состояния зависят от текущего состояния и не зависят от предыдущих.

В *скрытой марковской модели* состояния являются лишь частично наблюдаемыми, но, подобно простым марковским моделям, выходы этих частично наблюдаемых состояний являются полностью наблюдаемыми. Поскольку имеющихся наблюдений недостаточно для полного определения состояния, приходится привлекать обучение без учителя, обеспечивающее более полное обнаружение всех скрытых состояний.

Алгоритмы скрытой марковской модели включают обучение вероятному следующему состоянию при условии, что известна последовательность ранее встречавшихся частично наблюдаемых состояний и полностью наблюдаемых выходов. Эти алгоритмы широко применяются для решения задач, связанных с обработкой речи, текста и временны*х* рядов.

Обучение с подкреплением с использованием обучения без учителя

Обучение с подкреплением (reinforcement learning) — третья из основных методологий машинного обучения, в соответствии с которой *агент* определяет свое оптимальное поведение (*действия*) в условиях *окружения* на основе обратной связи (получаемого *вознаграждения*). Эта обратная связь называется *сигналом подкрепления*. Целью агента является максимизация накапливаемого вознаграждения.

Несмотря на то что обучение с подкреплением было на слуху еще с 1950-х годов, в заголовках новостей оно стало фигурировать лишь в последние годы. В 2013 году компания DeepMind (в настоящее время куплена компанией Google) применила обучение с подкреплением для достижения результатов, превосходящих возможности человека, во многих играх на платформе Atari. Разработанной компанией DeepMind системе удалось добиться этого, используя в качестве входа лишь данные сенсорных датчиков без предварительного знания правил игры.

В 2016 году компания DeepMind вновь стала объектом внимания сообщества машинного обучения, на этот раз благодаря программе AlphaGo, победившей Ли Седоля, одного из лучших игроков в го. Эти успехи укрепили позиции обучения с подкреплением как одного из главных направлений в разработке ИИ.

Сегодня исследователи, работающие в области машинного обучения, применяют обучение с подкреплением для решения многих задач, включая следующие.

- Биржевая торговля, в ходе которой агент покупает и продает акции (действия), получая взамен прибыль или убытки (вознаграждение).
- Видео- и настольные игры, в которых агент принимает решения относительно ходов в игре (действия), в конечном счете приводящие к выигрышу или проигрышу (вознаграждение).
- Беспилотные автомобили, в которых агент управляет транспортным средством (действия) и либо нормально следует по своему маршруту, либо совершает дорожно-транспортное происшествие (вознаграждение).
- Управление роботом, когда агент передвигается среди объектов окружения (действия) и либо достигает конечного пункта, либо ему это не удается (вознаграждение).

В простейших случаях мы имеем дело с конечной задачей, т.е. с задачей, в которой имеется конечное число состояний окружения, конечное число действий, доступных при том или ином состоянии окружения, и конечное число вознаграждений. Действия, совершаемые агентом при заданном текущем состоянии окружения, определяют следующее действие, а целью агента является максимизации долгосрочного вознаграждения. Это семейство задач известно как *марковские процессы принятия решений с конечным числом состояний.*

Однако на практике не все так просто. Вознаграждение неизвестно и по своему характеру является динамической, а не статической величиной. Нашим помощником в установлении этой неизвестной функции вознаграждения и ее наилучшей аппроксимации может выступать обучение без учителя. Используя приближенную функцию вознаграждения, мы можем применить решения на основе обучения с подкреплением для увеличения накапливаемой величины вознаграждения.

Обучение с частичным привлечением учителя

Несмотря на то что обучение с учителем и обучение без учителя — две различные методологии машинного обучения, оба алгоритма могут совместно применяться в качестве звеньев конвейера машинного обучения[11]. В типичных случаях смесь алгоритмов обучения с учителем и без учителя используется в тех случаях, когда мы хотим в полной мере извлечь выгоду из немногих имеющихся меток или найти новые, еще неизвестные закономерности, исходя из неразмеченных данных, в дополнение к тем, которые были получены на основе размеченных данных. Задачи этого типа решаются путем использования гибридного варианта обучения с учителем и без учителя, получившего название *обучение с частичным привлечением учителя* (semisupervised learning). В последующих главах мы вернемся к более подробному обсуждению этой темы.

[11] Термин *конвейер* (pipeline) обозначает систему приложений машинного обучения, применяемых последовательно для достижения общей цели.

Успешные примеры обучения без учителя

За последние десять лет большинство наиболее успешных коммерческих применений машинного обучения было связано с использованием обучения с учителем, однако в настоящее время ситуация начала меняться. Все большее распространение получает обучение без учителя. В одних случаях обучение без учителя служит всего лишь средством, позволяющим улучшить приложения на основе обучения с учителем. В других случаях обучение без учителя уже само по себе выступает в качестве основы для построения коммерческих приложений. Ниже кратко описаны две наибольшие области применения обучения без учителя по состоянию на сегодняшний день.

Обнаружение аномалий

Снижение размерности позволяет редуцировать исходное многомерное пространство признаков в преобразованное пространство более низкой размерности, в котором мы находим области с высокой плотностью точек. Эти области образуют нормальное пространство. Точки, расположенные на гораздо больших расстояниях, называются *выбросами*, или *аномалиями*, и заслуживают более пристального рассмотрения.

Как правило, системы обнаружения аномалий служат для выявления попыток мошенничества, связанных с использованием банковских карт, средств коммуникации и страховых полисов. Кроме того, обнаружение аномалий применяется для идентификации редких опасных событий, таких как взлом устройств, подключенных к Интернету, сбои в работе критического оборудования, например самолетов и поездов, и появление брешей в системах кибербезопасности из-за действий вредоносных программ.

Обнаружение аномалий можно использовать для распознавания спама, что уже обсуждалось нами ранее. К числу других применений относится обнаружение фактов финансирования терроризма, отмывания денег, торговли людьми, наркотиками и оружием, идентификация рискованных финансовых операций и диагностирование онкологических заболеваний.

Чтобы облегчить выявление аномалий, можно задействовать алгоритм кластеризации для группирования сходных аномалий с последующим назначением меток этим кластерам вручную на основании типов представляемого ими поведения. В состав такой системы можно включить интеллектуальный агент на основе обучения без учителя, способный обнаруживать аномалии, кластеризовать их в соответствующие группы и, используя метки,

присвоенные человеком, рекомендовать соответствующий порядок действий для бизнес-анализа.

В случае систем обнаружения аномалий мы можем воспользоваться описанным подходом на основе меток кластеров и, отталкиваясь от задачи, предназначенной для обучения без учителя, создать в конечном счете задачу для обучения с частичным привлечением учителя. Уже потом мы можем запустить алгоритмы обучения с учителем наряду с алгоритмами обучения без учителя. Чтобы приложения машинного обучения были успешными, системы обучения без учителя и с учителем должны применяться совместно, взаимно дополняя друг друга.

Система обучения с учителем находит известные закономерности с высокой степенью точности, тогда как система обучения без учителя обнаруживает новые закономерности, которые могут представлять для нас интерес. Выявив эти закономерности с помощью обучения без учителя, можно разметить данные вручную, переведя их из категории неразмеченных в категорию размеченных.

Сегментирование групп

С помощью кластеризации мы можем сегментировать группы на основании сходства их поведения в таких областях, как маркетинг, диагностика заболеваний, онлайн-покупки, прослушивание музыки, просмотр видеороликов, службы онлайн-знакомств, социальные сети и классификация документов. В каждом из этих случаев приходится иметь дело с огромными объемами данных, и эти данные размечены лишь частично.

Для уточнения уже известных закономерностей можно применять обучение с учителем. Однако зачастую мы хотим обнаруживать новые закономерности и группы, представляющие для нас интерес, и использование для этой цели обучения без учителя является вполне естественным вариантом выбора. Опять-таки, все дело в синергии. Залогом построения более надежных решений машинного обучения является совместное использование обучения с учителем и без учителя.

Резюме

В этой главе были рассмотрены следующие темы:

- различия между системами на основе правил и системами на основе машинного обучения;
- различия между обучением с учителем и без учителя;
- как обучение без учителя помогает справиться с проблемами, с которыми часто приходится сталкиваться в процессе тренировки моделей машинного обучения;
- распространенные алгоритмы обучения с учителем и без учителя, а также алгоритмы обучения с подкреплением и обучения с частичным привлечением учителя;
- две основные области применения обучения без учителя — обнаружение аномалий и сегментирование групп.

В главе 2 мы поговорим о том, как создавать приложения машинного обучения. Затем мы подробно рассмотрим методы снижения размерности и кластеризации, попутно создав систему обнаружения аномалий и систему сегментирования групп.

Готовый проект машинного обучения

Прежде чем приступить к исследованию алгоритмов обучения без учителя, необходимо узнать, как управлять проектами машинного обучения, начиная от получения данных и заканчивая реализацией готового решения. В этой главе мы поработаем с моделями обучения с учителем, которые должны быть знакомы большинству читателей, а к моделям обучения без учителя перейдем в следующей главе.

Настройка среды

Для начала необходимо настроить среду обработки данных. Мы будем использовать ее как для обучения с учителем, так и для обучения без учителя.

Приведенные ниже инструкции ориентированы на Windows, однако соответствующие установочные пакеты доступны также для Mac и Linux.

Git: система управления версиями

Если вы еще этого не сделали, вам потребуется установить Git (`https://git-scm.com/`). Git — это система управления версиями, и все примеры, приведенные в книге, доступны в виде блокнотов Jupyter в репозитории GitHub (`http://bit.ly/2Gd4v7e`). Ознакомьтесь с руководством по работе с Git (`http://rogerdudler.github.io/git-guide/index.ru.html`), чтобы узнать, как клонировать репозитории, добавлять, фиксировать и распространять изменения, а также контролировать ветки.

Клонируйте репозиторий данной книги

Откройте интерфейс командной строки и перейдите в каталог, в котором хотите хранить свои проекты обучения без учителя. Чтобы клонировать репозиторий GitHub, связанный с данной книгой, выполните следующие команды.

```
$ git clone
  https://github.com/aapatel09/handson-unsupervised-learning.git
$ git lfs pull
```

То же самое можно сделать иначе: посетите сайт GitHub (`http://bit.ly/2Gd4v7e`) и загрузите репозиторий вручную. Можете задать режим наблюдения за данным репозиторием, чтобы не пропустить изменения, которые могут вноситься в связи с обновлением кода.

Загрузив репозиторий посредством клонирования или вручную, перейдите в каталог *handson-unsupervised-learning* из командной строки:

```
$ cd handson-unsupervised-learning
```

Все последующие установки будут делаться в командной строке.

Библиотеки для научных вычислений: дистрибутив Anaconda для Python

Чтобы установить Python и библиотеки для научных вычислений, которые потребуются для наших проектов машинного обучения, загрузите дистрибутив Python под названием Anaconda (`https://www.anaconda.com/download/`)[1].

Создайте изолированную среду Python, в которую можно будет импортировать различные библиотеки для каждого проекта по отдельности:

```
$ conda create -n unsupervisedLearning python=3.6 anaconda
```

Эта команда создает в Python 3.6 изолированную среду `unsupervisedLearning`, которая содержит все научные библиотеки, поставляемые вместе с дистрибутивом Anaconda.

Далее необходимо активировать среду:

```
$ activate unsupervisedLearning
```

Нейронные сети: TensorFlow и Keras

Следующий шаг — установка пакетов TensorFlow и Keras для работы с нейронными сетями. *TensorFlow* — открытый проект компании Google, не являющийся частью дистрибутива Anaconda.

```
$ pip install tensorflow
```

[1] Авторские примеры созданы в версии 3.6, локализованные иллюстрации — в версии 3.7. — *Примеч. ред.*

Keras — библиотека с открытым исходным кодом, которая предлагает высокоуровневый API, использующий низкоуровневые функции TensorFlow. Другими словами, мы будем использовать Keras поверх TensorFlow, чтобы получить доступ к интуитивно более понятному набору API-вызовов для разработки моделей глубокого обучения.

```
$ pip install keras
```

Градиентный бустинг, версия 1: `XGBoost`

Далее необходимо установить библиотеку градиентного бустинга XGBoost. Чтобы унростить этот шаг (по крайней мере, для пользователей Windows), перейдите в папку *xgboost* репозитория *handson-unsupervised-learning*, в которой вы найдете указанный пакет.

Для установки пакета используйте команду `pip install`.

```
$ cd xgboost
$ pip install xgboost-0.6+20171121-cp36-cp36m-win_amd64.whl
```

Можете поступить по-другому, загрузив более актуальную 32- или 64-битную версию XGBoost (`http://bit.ly/2G1jBxs`), в зависимости от разрядности используемой вами операционной системы.

Имя загруженного вами WHL-файла XGBoost может отличаться от приведенного выше, поскольку новые версии публикуются регулярно.

Когда файл XGBoost будет успешно установлен, вернитесь в папку *handson-unsupervised-learning*.

Градиентный бустинг, версия 2: `LightGBM`

Установите другую библиотеку градиентного бустинга — LightGBM, разработанную компанией Microsoft:

```
$ pip install lightgbm
```

Алгоритмы кластеризации

Сейчас мы установим несколько алгоритмов кластеризации, которые будут использоваться в последующих главах. Один из пакетов, `fastcluster`, представляет собой библиотеку C++ с интерфейсом Python/SciPy[2].

Пакет `fastcluster` можно установить с помощью следующей команды:

```
$ pip install fastcluster
```

Другой пакет кластеризации, `hdbscan`, можно установить с помощью программы `conda`:

```
$ conda install -c conda-forge hdbscan
```

Наконец, установите пакет `tslearn`, предназначенный для работы с временны*ы*ми рядами:

```
$ pip install tslearn
```

Интерактивная вычислительная среда: Jupyter Notebook

Блокнот Jupyter является частью дистрибутива Anaconda, поэтому сейчас мы активизируем его, чтобы запустить среду, которую только что настроили. Прежде чем ввести следующую команду, убедитесь в том, что находитесь в папке репозитория *handson-unsupervised-learning*:

```
$ jupyter notebook
```

В открывшемся окне браузера отобразится страница с адресом `http://localhost:8888/` (если этого не произошло, скопируйте указанный в командной строке адрес страницы в окно браузера). Для доступа к странице должны быть разрешены файлы "cookie".

Теперь мы готовы к созданию нашего первого проекта машинного обучения.

Обзор данных

В этой главе мы будем использовать реальный набор данных об анонимных операциях с банковскими картами, совершенных европейскими клиентами

[2] Для получения более подробной информации о библиотеке *fastcluster* обратитесь к документации (`https://pypi.org/project/fastcluster/`).

начиная с сентября 2013 года[3]. Все операции помечены как поддельные или подлинные, и мы построим приложение на основе машинного обучения, позволяющее выявлять мошенничество и предсказывать корректные метки для примеров, которые ранее не предоставлялись.

Указанный набор данных отличается крайней несбалансированностью. Из 284 807 банковских операций лишь 492 (0,172%) являются мошенническими. Столь низкий процент подделок вполне типичен для операций с банковскими картами.

Общее количество признаков равно 28, причем все они числовые, тогда как категориальные переменные отсутствуют[4]. Эти признаки не являются оригинальными и представляют собой результат анализа главных компонент (PCA), который мы будем исследовать в главе 3. В данном случае количество оригинальных признаков было уменьшено до 28 главных компонент путем снижения размерности.

В дополнение к 28 главным компонентам имеются три другие переменные: время выполнения транзакции, сумма транзакции и истинный класс транзакции (1 — поддельная, 0 — подлинная).

Подготовка данных

Прежде чем использовать машинное обучение для организации процесса тренировки на данных и разработки приложения, обнаруживающего факты мошенничества, необходимо подготовить данные, которые будут обрабатываться алгоритмами.

Получение данных

Первым шагом в любом проекте машинного обучения является получение данных.

Загрузка данных

Загрузите набор данных и поместите результирующий CSV-файл в папку */datasets/credit_card_data/* каталога *handson-unsupervised-learning*. Если ранее вы

[3] Этот набор данных, доступный в сети Kaggle (`www.kaggle.com`), был собран в ходе исследования, проведенного компанией Worldline и сообществом Machine Learning Group из Брюссельского свободного университета.

[4] Категориальные переменные имеют одно из возможных качественных значений, количество которых ограничено. Их часто приходится кодироваться в алгоритмах машинного обучения.

загрузили репозиторий GitHub, то этот файл уже находится в указанной папке. (Убедитесь, что файл полностью загружен, так как он имеет огромный размер.)

Импорт необходимых библиотек

Импортируйте библиотеки Python, которые потребуются для построения приложения, обнаруживающего факты мошенничества.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline

'''Подготовка данных'''
from sklearn import preprocessing as pp
from scipy.stats import pearsonr
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score
from sklearn.metrics import confusion_matrix, classification_report

'''Алгоритмы'''
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
import lightgbm as lgb
```

Чтение данных

```
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'credit_card_data', \
                         'credit_card.csv'])
data = pd.read_csv(current_path + file)
```

Предварительный просмотр данных

Следующая команда выводит первые пять строк набора:

```
data.head()
```

Как видите, требуемый набор данных успешно загружен (табл. 2.1; здесь и далее показаны только начальные столбцы).

Таблица 2.1. Предварительный просмотр данных

	Time	V1	V2	V3	V4	V5
0	0.0	−1.359807	−0.072781	2.536347	1.378155	−0.338321
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018
2	1.0	−1.358354	−1.340163	1.773209	0.379780	−0.503198
3	1.0	−0.966272	−0.185226	1.792993	−0.863291	−0.010309
4	2.0	1.158233	0.877737	1.548718	0.403034	−0.407193

5 rows x 31 columns

Исследование данных

Нашим следующим шагом будет более глубокое исследование данных. Для этого мы сгенерируем итоговую сводку, выявим отсутствующие данные или категориальные признаки и подсчитаем количество различных значений по признакам.

Генерирование итоговой сводки

Следующая команда выводит сводку по столбцам (табл. 2.2):

```
data.describe()
```

Таблица 2.2. Простая итоговая сводка

	Time	V1	V2	V3	V4
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	3.919560e−15	5.688174e−16	−8.769071e−15	2.782312e−15
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00
min	0.000000	−5.640751e+01	−7.271573e+01	−4.832559e+01	−5.683171e+00
25%	54201.500000	−9.203734e−01	−5.985499e−01	−8.903648e−01	−8.486401e−01
50%	84692.000000	1.810880e−02	6.548556e−02	1.798463e−01	−1.984653e−02

Окончание табл. 2.2

	Time	V1	V2	V3	V4
75%	139320.500000	1.315642e+00	8.037239e–01	1.027196e+00	7.433413e–01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01

8 rows x 31 columns

```
data.columns
```

```
Index(['Time', 'V1,' 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', \
    'V9', 'V10', 'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', \
    'V18', 'V19', 'V20', 'V21', 'V22', 'V23', 'V24', 'V25', 'V26', \
    'V27', 'V28', 'Amount', 'Class'], dtype='object')
```

```
data['Class'].sum()
```

Общее количество положительных меток, или мошеннических операций, составило 492. Как и следовало ожидать, всего имеется 284 807 примеров и 31 столбец: 28 числовых признаков (`V1`–`V28`), а также столбцы `Time`, `Amount` и `Class`.

Значения временны*х* меток (`Time`) находятся в диапазоне 0–172792, суммы операций (`Amount`) — в диапазоне 0–25691,16; всего имеется 492 мошеннические операции. На них мы будем ссылаться как на позитивные случаи, или позитивные метки (они помечены единицами); нормальные операции соответствуют отрицательным случаям, или отрицательным меткам (они помечены нулями).

Все 28 числовых признаков пока еще не стандартизированы, но вскоре мы стандартизируем данные. В процессе *стандартизации* данные масштабируются таким образом, чтобы среднее значение было равно нулю, а стандартное отклонение — единице.

Некоторые приложения машинного обучения весьма чувствительны к масштабированию данных, поэтому приведение данных к одной и той же относительной шкале — стандартизация — считается хорошей практикой.

Еще одним общепринятым методом масштабирования данных является их *нормализация*, при которой данные приводятся к диапазону значений от нуля до единицы. В отличие от стандартизированных данных все нормализованные данные располагаются вдоль шкалы в области положительных значений.

Выявление нечисловых и отсутствующих значений

Некоторые алгоритмы машинного обучения не могут обрабатывать нечисловые и отсутствующие значения. Поэтому оптимальное решение заключается в использовании особого способа идентификации нечисловых значений (также известных как *NaN* от англ. "not a number").

В случае отсутствующих значений мы можем моделировать их, например путем замены отсутствующей точки данных средним значением, медианой или модой соответствующего признака, либо подставлять значения, определяемые пользователем. В случае категориальных значений можно кодировать данные с помощью разреженной матрицы, представляющей все категориальные значения. Далее эта матрица комбинируется с числовыми признаками. Полученный комбинированный набор признаков используется для тренировки алгоритма машинного обучения.

Выполнив следующую команду, мы видим, что в наборе отсутствуют наблюдения со значением NaN, поэтому нет никакой необходимости моделировать или кодировать какие-либо значения.

```
nanCounter = np.isnan(data).sum()

Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
```

```
V21        0
V22        0
V23        0
V24        0
V25        0
V26        0
V27        0
V28        0
Amount     0
Class      0
dtype: int64
```

Выявление различающихся зиачений признаков

Чтобы получше разобраться с особенностями набора транзакционных данных, подсчитаем количество различных значений признаков.

Выполнив следующую команду, мы видим, что набор данных содержит 124 592 различные временны*е* метки. Но мы уже знаем, что всего имеется 284 807 наблюдений. Это означает, что некоторым временны*м* меткам соответствует несколько транзакций.

И, как и следовало ожидать, существуют всего лишь два класса: 1 для мошеннических транзакций и 0 — для подлинных.

```
distinctCounter = data.apply(lambda x: len(x.unique()))
```

```
Time       124592
V1         275663
V2         275663
V3         275663
V4         275663
V5         275663
V6         275663
V7         275663
V8         275663
V9         275663
V10        275663
V11        275663
V12        275663
V13        275663
V14        275663
V15        275663
V16        275663
```

```
V17        275663
V18        275663
V19        275663
V20        275663
V21        275663
V22        275663
V23        275663
V24        275663
V25        275663
V26        275663
V27        275663
V28        275663
Amount      32767
Class           2
dtype: int64
```

Генерирование матрицы признаков и массива меток

Давайте создадим и стандартизируем матрицу признаков `X` и выделим массив меток `Y` (1 для мошеннических транзакций, 0 — для подлинных). Впоследствии мы будем передавать эту информацию алгоритмам машинного обучения в процессе тренировки.

Создание матрицы признаков X и матрицы меток Y

```
dataX = data.copy().drop(['Class'], axis=1)
dataY = data['Class'].copy()
```

Стандартизация матрицы признаков X

Изменим масштаб матрицы признаков таким образом, чтобы каждый признак, за исключением времени, имел среднее значение, равное 0, и стандартное отклонение, равное 1.

```
featuresToScale = dataX.drop(['Time'], axis=1).columns
sX = pp.StandardScaler(copy=True)
dataX.loc[:, featuresToScale] = \
  sX.fit_transform(dataX[featuresToScale])
```

Как показано в табл. 2.3, теперь стандартизированные признаки соответствуют указанным условиям.

Таблица 2.3. Итоговые масштабированные признаки

	Time	V1	V2	V3	V4
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	−8.157366e−16	3.154853e−17	−4.409878e−15	−6.734811e−16
std	47488.145955	1.000002e+00	1.000002e+00	1.000002e+00	1.000002e+00
min	0.000000	−2.879855e+01	−4.403529e+01	−3.187173e+01	−4.013919e+00
25%	4201.500000	−4.698918e−01	−3.624707e−01	−5.872142e−01	−5.993788e−01
50%	84692.000000	9.245351e−03	3.965683e−02	1.186124e−02	−1.401724e−01
75%	139320.500000	6.716939e−01	4.867202e−01	6.774569e−01	5.250082e−01
max	172792.000000	1.253351e+00	1.335775e+01	6.187993e+00	1.191874e+01

8 rows x 30 columns

Конструирование и отбор признаков

В большинстве проектов машинного обучения мы должны рассматривать вопросы конструирования и выбора признаков как часть решения. *Конструирование признаков* (feature engineering) подразумевает создание новых признаков — например, путем вычисления отношений, значений счетчиков или сумм на основе исходных признаков, — чтобы оказать помощь алгоритму машинного обучения в извлечении более сильного сигнала из набора данных.

Отбор признаков (feature selection) подразумевает выбор поднабора признаков для тренировки, что в конечном счете означает исключение некоторых менее релевантных признаков из рассмотрения. Целью этого шага является снижение вероятности переобучения алгоритма машинного обучения на шуме в наборе данных.

Для нашего набора данных отсутствуют исходные признаки. Мы располагаем лишь главными компонентами, полученными с использованием метода PCA (один из вариантов снижения размерности), о чем пойдет речь в главе 3. Поскольку мы не знаем, что именно представляет собой любой из признаков, мы не можем применить какую-либо разумную методику конструирования признаков.

В нашем случае отбор признаков также не требуется, поскольку количество наблюдений (284 807) намного превосходит количество признаков (30), что резко снижает вероятность переобучения. Как следует из рис. 2.1, признаки лишь в незначительной степени коррелируют между собой. Другими словами, у нас нет избыточных признаков. Если бы они были, мы могли бы устранить

или уменьшить избыточность данных путем снижения размерности. Разумеется, в этом нет ничего удивительного, поскольку к рассматриваемому набору данных о банковских картах уже был применен метод PCA, который выполнил всю работу за нас.

Проверка корреляции признаков

```
correlationMatrix = pd.DataFrame(data=[], index=dataX.columns, \
                                 columns=dataX.columns)
for i in dataX.columns:
    for j in dataX.columns:
        correlationMatrix.loc[i, j] = \
            np.round(pearsonr(dataX.loc[:, i], \
                              dataX.loc[:, j])[0], 2)
```

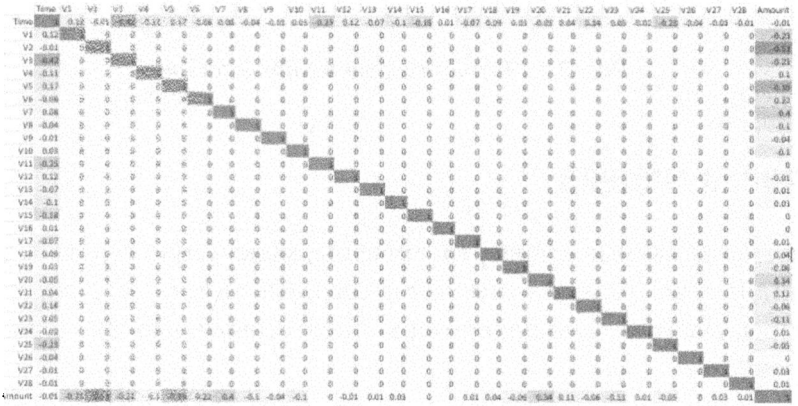


Рис. 2.1. Корреляционная матрица

Визуализация данных

Последний шаг — визуализация данных, позволяющая оценить степень несбалансированности набора примеров (рис. 2.2). Ввиду малочисленности примеров мошеннических операций в наборе, с которым мы работаем, решение этой задачи представляет значительные трудности; к счастью, в нашем распоряжении имеются метки для всего набора данных.

```
count_classes = pd.value_counts(data['Class'], \
                                 sort=True).sort_index()
ax = sns.barplot(x=count_classes.index, \
                 y=tuple(count_classes/len(data)))
```

```
ax.set_title('Частотность меток классов')
ax.set_xlabel('Класс')
ax.set_ylabel('Частотность')
```

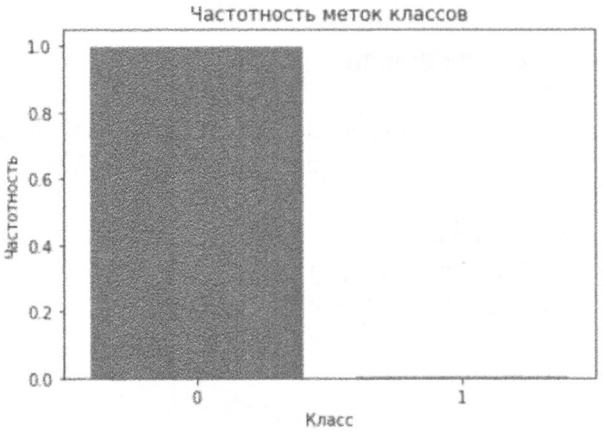


Рис. 2.2. Частотность меток классов

Подготовка модели

Теперь, когда данные стандартизированы, можем приступать к подготовке модели. Для этого мы должны разбить данные на тренировочный и тестовый наборы, выбрать функцию потерь и подготовить наборы для *k*-кратной кросс-проверки.

Разбиение данных на тренировочный и тестовый наборы

В главе 1 говорилось о том, что алгоритмы машинного обучения предварительно обучаются (тренируются) на некотором наборе данных, с тем чтобы обеспечить хорошую результативность (т.е. точность предсказаний) в отношении данных, которые ранее им не предоставлялись. Критерием результативности служит так называемая *ошибка обобщения* (generalization error) — самая важная из метрик, применяемых для оценки пригодности модели машинного обучения.

Прежде всего мы должны настроить наш проект машинного обучения для использования набора данных, на котором будет обучаться алгоритм. Нам также потребуется тестовый набор (не предоставлявшийся ранее), который будет задействоваться алгоритмом машинного обучения в целях прогнозирования.

Результативность, достигаемая на этом тестовом наборе, и будет конечной оценкой успешности модели.

Итак, разобьем набор данных об операциях с банковскими картами на два набора: тренировочный и тестовый.

```
X_train, X_test, y_train, y_test = train_test_split(dataX, dataY, \
    test_size=0.33, random_state=2018, stratify=dataY)
```

Теперь мы имеем тренировочный набор, содержащий 190 820 образцов (67% исходного количества), и тестовый набор, содержащий 93 987 образцов (оставшиеся 33%). Чтобы обеспечить одинаковую (~0,17%) долю мошеннических операций в тренировочном и тестовом наборах, мы устанавливаем параметр `stratify`. Мы также фиксируем значение `2018` для переменной `random_state`, чтобы обеспечить воспроизводимость результатов[5].

Для окончательной оценки ошибки обобщения, также называемой *ошибкой за пределами выборки* (out-of-sample error), мы будем использовать тестовый набор.

Выбор функции потерь

Для обучения модели на тренировочном наборе нам понадобится *функция потерь* (cost function), которую мы будем передавать алгоритму машинного обучения. Алгоритм будет пытаться минимизировать функцию стоимости, обучаясь на тренировочных примерах.

Поскольку в данном случае мы применяем обучение с учителем для решения задачи классификации (с двумя классами), мы используем логарифмическую функцию потерь бинарной классификации для вычисления *кросс-энтропии* между метками истинных транзакций и модельными предсказаниями:

$$\log\text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j}),$$

где N — количество наблюдений, M — количество меток классов (в данном случае равное 2); $y_{i,j}$ — 1, если наблюдение i относится к классу j, и 0 — в противном случае; $p_{i,j}$ — предсказанная вероятность того, что наблюдение i относится к классу j.

[5] Чтобы узнать, каким образом параметр `stratify` сохраняет постоянную долю положительных меток, обратитесь к документации (`http://bit.ly/2NiKWfi`). Если хотите воспроизвести аналогичное разбиение наборов в своих экспериментах, установите для параметра `random_state` значение `2018`. Если не задавать его или выбрать другое значение, то результаты будут другими.

Модель машинного обучения будет генерировать вероятность подделки для каждой операции с банковскими картами. Чем ближе вероятности подделки к истинным меткам (1 для поддельных транзакций и 0 для истинных), тем ниже значение функции потерь. Именно это значение и будет минимизировать алгоритм машинного обучения.

Создание наборов для *k*-кратной кросс-проверки

Чтобы помочь алгоритму машинного обучения в оценке того, насколько успешно он справится с примерами, которые ранее ему не предоставлялись (тестовый набор), принято дополнительно разбивать тренировочный набор на собственно тренировочный и валидационный наборы.

Например, если мы разделим тренировочный набор на пять равных частей, то можно будет выполнить тренировку модели на четырех пятых исходного тренировочного набора, а оставшуюся пятую часть (валидационный набор), использовать для оценки результативности предсказаний обученной модели.

Подобную тренировку и оценку модели можно выполнить пять раз, каждый раз меняя отложенную пятую часть исходного набора в качестве валидационного набора. Такой подход получил название *k-кратная кросс-проверка* (k-fold cross-validation — CV). В нашем случае *k* равно 5, так что для ошибки обобщения мы будем иметь не одну, а пять оценок.

Мы будем сохранять оценки тренировки и кросс-проверки для каждого из пяти прогонов, всякий раз сохраняя предсказания для кросс-проверочного набора. По завершении всех пяти прогонов мы будем располагать кросс-проверочными предсказаниями для всего набора данных. Это даст нам наилучшую общую оценку результативности для тестового набора.

Настройка *k*-кратной валидации, где *k* равно 5, осуществляется следующим образом:

```
k_fold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2018)
```

Модели машинного обучения (часть I)

Теперь мы полностью подготовлены к построению моделей машинного обучения. Для каждого рассматриваемого алгоритма машинного обучения мы будем устанавливать гиперпараметры, тренировать модель и оценивать результаты.

Модель №1: логистическая регрессия

Начнем с самого простого алгоритма классификации — логистической регрессии.

Настройка гиперпараметров

```
penalty = 'l2'
C = 1.0
class_weight = 'balanced'
random_state = 2018
solver = 'liblinear'
n_jobs = 1

logReg = LogisticRegression(penalty=penalty, C=C, \
    class_weight=class_weight, random_state=random_state, \
    solver=solver, n_jobs=n_jobs)
```

Установим для параметра `penalty` значение `l2`, а не `l1`. Дело в том, что *L2*-регуляризация менее чувствительна к выбросам по сравнению с *L1*-регуляризацией и назначает ненулевые значения весов почти всем признакам, что будет приводить к получению стабильных решений. *L1*-регуляризация назначает высокие значения весов наиболее важным признакам и почти нулевые веса остальным признакам, по сути, выполняя отбор признаков в процессе тренировки алгоритма. Однако ввиду существенной вариации весов от признака к признаку *L1*-решения, в отличие от *L2*-решений, не всегда ведут себя стабильно по отношению к изменениям в точках данных[6].

Параметр `C` определяет интенсивность регуляризации. Как вы помните из главы 1, регуляризация помогает бороться с переобучением путем наложения штрафов за сложность. Иными словами, чем интенсивнее регуляризация, тем больший штраф за сложность налагается алгоритмом машинного обучения. Регуляризация подталкивает алгоритм к тому, чтобы при прочих равных условиях предпочтение отдавалось более простым моделям по сравнению с более сложными.

В качестве значения константы регуляризации `C` необходимо задавать положительное вещественное число. Чем меньше это значение, тем интенсивнее регуляризация. Мы оставим заданное по умолчанию значение `1.0`.

[6] Более подробную информацию о различиях между L1- и L2-регуляризацией можно найти по адресу `http://bit.ly/2Bcx413`.

Наш набор данных о транзакциях с банковскими картами отличается крайней несбалансированностью: из общего количества примеров, равного 284 807, лишь 492 соответствуют поддельным транзакциям. Мы хотим, чтобы в процессе тренировки алгоритм машинного обучения уделял больше внимания обучению на транзакциях с положительными метками — другими словами, обучению на поддельных транзакциях, поскольку их количество в наборе данных очень невелико.

В этой модели логистической регрессии мы установим для переменной `class_weight` значение `'balanced'`. Тем самым мы сообщаем алгоритму логистической регрессии о том, что в данной задаче имеются проблемы, связанные с несбалансированностью классов. Алгоритму потребуется приписывать *б*ольшие веса положительным меткам в процессе тренировки. В данном случае веса будут обратно пропорциональны частотам встречаемости классов. Алгоритм будет приписывать *б*ольшие веса редко встречающимся положительным меткам (которые соответствуют поддельным транзакциям) и меньшие — более частым отрицательным меткам (соответствующим подлинным транзакциям).

Для переменной `random_state` мы фиксируем значение `2018`, чтобы упростить другим пользователям — в том числе всем читателям — воспроизведение результатов. Для переменной `solver` установлено стандартное значение `'liblinear'`.

Тренировка модели

Установив все гиперпараметры, мы можем приступить к тренировке модели логистической регрессии, используя по очереди каждый из пяти вариантов разбиения исходного набора в соответствии с методикой *k*-кратной кросс-проверки, т.е. тренируя модель каждый раз на четырех пятых тренировочного набора и оценивая ее результативность с помощью отложенной пятой части набора.

В процессе этой пятикратной тренировки и оценки модели мы будем вычислять фупкцию потерь — в нашем случае логарифмическую функцию — как во время обучения (на срезе размером четыре пятых исходного тренировочного набора), так и во время валидации (на срезе размером в одну пятую исходного тренировочного набора). Кроме того, мы будем сохранять предсказания также для каждого из пяти кросс-проверочных наборов, и к концу пятого прогона мы будем иметь пять предсказаний для всего тренировочного набора.

```
trainingScores = []
cvScores = []
```

```
predictionsBasedOnKFolds = pd.DataFrame(data=[], \
                                        index=y_train.index, \
                                        columns=[0, 1])

model = logReg

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
    X_train_fold, X_cv_fold = X_train.iloc[train_index, :], \
        X_train.iloc[cv_index, :]
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    model.fit(X_train_fold, y_train_fold)
    loglossTraining = log_loss(y_train_fold, \
        model.predict_proba(X_train_fold)[:, 1])
    trainingScores.append(loglossTraining)

    predictionsBasedOnKFolds.loc[X_cv_fold.index, :] = \
        model.predict_proba(X_cv_fold)
    loglossCV = log_loss(y_cv_fold, \
        predictionsBasedOnKFolds.loc[X_cv_fold.index, 1])
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения: ', loglossTraining)
    print('Логарифмические потери валидации: ', loglossCV)

loglossLogisticRegression = log_loss(y_train, \
    predictionsBasedOnKFolds.loc[:, 1])
print('Логарифмические потери логистической регрессии: ', \
      loglossLogisticRegression)
```

Оценка результатов

Ниже приведены тренировочные и кросс-проверочные значения функции потерь для каждого из пяти прогонов. Как правило (но не всегда) тренировочные значения ниже кросс-нроверочных. Поскольку алгоритм машинного обучения обучался непосредственно на тренировочных данных, его эффективность (характеризуемая логарифмическими потерями) должна быть выше на тренировочном наборе, а не на кросс-проверочном. Вспомните, что кросс-проверочный набор включает транзакции, которые были явным образом удержаны из тренировочных примеров.

```
Логарифмические потери обучения:    0.10080139188958696
Логарифмические потери валидации:   0.10490645274118293
Логарифмические потери обучения:    0.12098957040484648
Логарифмические потери валидации:   0.11634801169793386
Логарифмические потери обучения:    0.1074616029843435
Логарифмические потери валидации:   0.10845630232487576
Логарифмические потери обучения:    0.10228137039781758
Логарифмические потери валидации:   0.10321736161148198
Логарифмические потери обучения:    0.11476012373315266
Логарифмические потери валидации:   0.1160124452312548
```

Работая с набором данных о транзакциях с банковскими картами, важно не забывать, что мы строим приложение, предназначенное для выявления мошеннических операций. Всякий раз, когда речь идет об эффективности или результативности модели машинного обучения, имеется в виду, насколько она справляется с обнаружением подделок.

Модель машинного обучения выводит вероятности предсказаний для каждой транзакции, причем единица соответствует поддельной транзакции, а нуль — подлинной транзакции. Чем ближе значение вероятности к единице, тем больше шансов, что транзакция является мошеннической, и чем ближе это значение к нулю, тем вероятнее, что транзакция является законной. Сравнивая предсказанные моделью значения вероятности с метками, мы можем судить о пригодности модели.

На каждом из пяти прогонов тренировочные и кросс-проверочные логарифмические потери имеют близкие значения. Для данной модели логистической регрессии мы не наблюдаем сколь-нибудь значительных проявлений переобучения, так как в противном случае мы имели бы низкие тренировочные логарифмические потери и высокие кросс-проверочные потери.

Поскольку мы сохраняем предсказания для каждого из пяти кросс-проверочных наборов, мы можем скомбинировать их для получения оценки, относящейся к единому набору. Этот единый набор представляет собой то же самое, что и исходный тренировочный набор, что позволяет нам рассчитать общие логарифмические потери для всего исходного тренировочного набора. Наилучшая оценка логарифмических потерь для данной модели логистической регрессии, полученная на тестовом наборе, имеет следующее значение:

```
Логарифмические потери логистической регрессии: 0.10978811472134588
```

Оценочные метрики

Несмотря на то что логарифмические потери могут служить неплохой оценкой эффективности модели машинного обучения, было бы желательно иметь более интуитивный способ оценки, облегчающий понимание результатов. Например, нам хотелось бы знать, какую долю поддельных транзакций из числа тех, которые имеются в тренировочном наборе, удалось выявить. Этот показатель называется *полнота* (recall). Нас также интересует, какова действительная доля истинных транзакций среди тех, которые были помечены моделью логистической регрессии как истинные. Этот показатель характеризует *точность* (precision) модели.

Рассмотрим более детально, как эти и аналогичные оценочные метрики способствуют более интуитивному осмыслению результатов.

Роль этих оценочных метрик очень велика, поскольку они дают исследователям возможность на интуитивном уровне объяснять результаты менеджерам, менее знакомым с такими понятиями, как логарифмические потери, кросс-энтропия и другие функции стоимости. Умение доносить неспециалистам суть полученных результатов в наиболее простой форме служит одним из наиболее важных навыков, которыми должен обладать специалист по работе с данными.

Матрица неточностей

В типичных задачах классификации (в отсутствие дисбаланса классов) для оценки результатов можно использовать *матрицу неточностей* (confusion matrix), которая представляет собой таблицу, содержащую суммарные данные о количестве истинноположительных, истинноотрицательных, ложноположительных и ложноотрицательных результатов (рис. 2.3)[7].

[7] *Истинноположительные* результаты примеров — те, в которых как предсказание, так и фактическая метка имеют истинные значения. *Истинноотрицательные* результаты — те, в которых как предсказание, так и фактическая метка имеют ложные значения. *Ложноположительными* считаются результаты, в соответствии с которыми предсказание имеет истинное значение, а фактическая метка — ложное (такую ситуацию называют *ложной тревогой* или *ошибкой I рода*). *Ложноотрицательными* считаются результаты, в соответствии с которыми предсказание имеет ложное значение, а фактическая метка — истинное (такую ситуацию называют *пропуском события* или *ошибкой II рода*).

Рис. 2.3. Матрица неточностей

Учитывая высокую степень несбалансированности нашего набора данных относительно операций с банковскими картами, информация, предоставляемая матрицей неточностей, будет не слишком содержательной. Например, если бы мы предсказали, что ни одна из транзакций не является поддельной, то результаты были бы такими: 284 315 истинноотрицательных, 492 ложноотрицательных, 0 истинноположительных и 0 ложноположительных. Таким образом, при идентификации действительно поддельных транзакций мы получили бы точность 0%. В задачах с несбалансированными классами матрица неточностей плохо справляется с выявлением таких неоптимальных исходов.

В случае задач, включающих более сбалансированные классы (т.е. когда количество истинноположительных результатов в грубом приближении совпадает с количеством истинноотрицательных), матрица неточностей может успешно использоваться в качестве непосредственной оценочной метрики. В то же время для нашего несбалансированного набора данных мы должны подыскать более подходящую метрику.

Кривая "точность — полнота"

В контексте нашего случая, когда набор данных об операциях с банковскими картами не сбалансирован, для оценки результатов лучше всего использовать такие относительные показатели качества классификатора, как точность и полнота. *Точность* — это отношение количества истинноположительных предсказаний к общему фактическому количеству положительных результатов для набора данных. Другими словами, этот показатель позволяет судить о том, как много поддельных транзакций действительно удалось определить модели.

$$\text{Точность} = \text{Истинноположительные} / (\text{Истинноположительные} + \text{Ложноположительные})$$

Высокое значение точности означает, что из всех положительных предсказаний многие являются истинноположительными (т.е. доля ложноположительных предсказаний незначительна).

Полнота (чувствительность) — это отношение количества истинноположительных предсказаний к общему фактическому количеству правильно предсказанных результатов для набора данных[8].

$$\text{Полнота} = \text{Истинноположительные} / (\text{Истинноположительные} + \text{Ложноотрицательные})$$

Высокое значение полноты означает, что модели удалось обнаружить большую часть истинноположительных результатов (т.е. доля ложноотрицательных результатов невелика).

Решение с высокой полнотой, но низкой точностью возвращает большое количество результатов, захватывая значительную долю положительных результатов, но при этом также содержит много ложноположительных. Решение с высокой точностью, но низкой полнотой ведет себя прямо противоположно: оно возвращает небольшое количество результатов, захватывая лишь некоторую часть положительных результатов, содержащихся в наборе, но большинство его предсказаний корректны.

Итак, если наше решение характеризуется высокой точностью, но низкой полнотой, то число обнаруженных поддельных транзакций будет очень небольшим, зато большинство из них будут действительно поддельными.

В то же время решение с низкой точностью, но с высокой полнотой пометит многие транзакции как поддельные, фиксируя значительное количество подделок, хотя на самом деле такие транзакции в большинстве своем не будут поддельными.

Очевидно, что оба решения порождают существенные проблемы. В первом случае компания — эмитент банковских карт понесет ощутимые финансовые потери в связи с осуществлением мошеннических операций, но зато не восстановит против себя клиентов, необоснованно отвергая некоторые транзакции. Во втором случае компания перехватит значительную долю попыток мошенничества, но при этом, несомненно, вызовет гнев клиентов из-за необоснованных отказов в осуществлении многих нормальных транзакций, не являющихся мошенническими.

---

[8] Родственным по отношению к полноте понятием является *специфичность*, или доля истинноотрицательных предсказаний. Этот показатель определяется как отношение количества истинноотрицательных предсказаний к общему количеству отрицательных предсказаний для набора данных. См. также `https://ru.wikipedia.org/wiki/Двоичная_классификация`.

Оптимальное решение должно характеризоваться высокой точностью и полнотой, отвергая лишь действительно мошеннические операции (высокая точность) и перехватывая бо́льшую часть таких операций, представленных в наборе данных (высокая полнота).

Во многих случаях приходится соблюдать определенный компромисс между точностью и полнотой, что обычно достигается установкой порога решающего правила, используемого алгоритмом для разделения положительных и отрицательных случаев. В нашем примере положительные случаи соответствуют поддельным операциям, отрицательные — подлинным. Если установлен слишком высокий порог, то лишь очень небольшое количество случаев будут предсказаны как положительные, в результате чего мы получим высокую точность, но низкую полноту. По мере снижения порога все больше случаев будут предсказываться как положительные, что обычно будет приводить к уменьшению точности и увеличению полноты.

Применительно к нашему набору данных о транзакциях с банковскими картами можно считать, что пороговое значение характеризует чувствительность модели машинного обучения в отношении отклонения транзакций. В случае слишком высокого/строгого порога количество отклоненных моделью транзакций будет небольшим, но, вероятнее всего, они действительно окажутся поддельными.

По мере снижения порога (т.е. ослабления строгости) модель будет отклонять большее количество транзакций, относя их к случаям мошенничества, но при этом в их число попадут также нормальные транзакции.

График зависимости между точностью и полнотой классификации называется *кривая "точность — полнота"*. Для вычисления этой кривой мы должны рассчитать среднюю точность, являющуюся взвешенным средним значений точности при каждом значении порога. Чем выше средняя точность, тем лучше решение.

Выбор порогового значения играет очень важную роль и обычно подразумевает участие человека, принимающего решения. Исследователи могут предоставлять кривые "точность — полнота" менеджерам, чтобы те решали, каким должен быть порог.

При работе с нашим набором данных, содержащим сведения об операциях с банковскими картами, ключевым становится вопрос о том, как достичь разумного компромисса между комфортом клиентов (путем сведения к минимуму случаев ложного признания нормальных операций мошенническими) и пресечением попыток мошенничества (за счет отклонения поддельных транзакций). Мы

не можем дать ответ на этот вопрос без привлечения менеджеров соответствующего уровня, но можем найти модель, характеризующуюся наилучшей кривой "точность — полнота". После этого мы можем предъявить эту модель менеджерам, чтобы те дали свои рекомендации относительно того, какой порог следует установить.

Рабочая характеристика приемника

Еще одной неплохой метрикой, позволяющей оценить качество бинарной классификации, служит показатель *auROC* (area under ROC — площадь под ROC-кривой). Здесь ROC (receiver operating characteristic) — *рабочая характеристика приемника*. Кривая рабочей характеристики приемника (также называемая *кривой ошибок*) отображает в графическом виде соотношение между частотой (долей) истинноположительных (ось Y) и ложноположительных (ось X) результатов. Эти две переменные используются в качестве следующих двух характеристик алгоритма классификации: *чувствительность*, или *TPR* (true positive rate — частота истинноположительных результатов), и *специфичность*, или *FPR* (false positive rate — частота ложноположительных результатов). Чем ближе кривая ошибок к левому верхнему углу графика, тем лучше решение. Координата (`0.0, 1.0`) — это точка абсолютного оптимума, которой соответствуют 0% ложноположительных результатов и 100% истинноположительных результатов.

Для количественной интерпретации ROC можно использовать площадь области под кривой ошибок, или показатель auROC. Чем выше этот показатель, тем лучше решение.

Вычисление модели логистической регрессии

Теперь, когда вам уже понятен смысл некоторых из оценочных метрик, мы можем использовать их для интерпретации результатов логистической регрессии.

Прежде всего построим кривую "точность — полнота" и вычислим среднюю точность.

```
preds = pd.concat([y_train, predictionsBasedOnKFolds.loc[:, 1]], \
                  axis=1)
preds.columns = ['trueLabel', 'prediction']
predictionsBasedOnKFoldsLogisticRegression = preds.copy()

precision, recall, thresholds = \
    precision_recall_curve(preds['trueLabel'], preds['prediction'])
```

```
average_precision = \
    average_precision_score(preds['trueLabel'], preds['prediction'])

plt.step(recall, precision, color='k', alpha=0.7, where='post')
plt.fill_between(recall, precision, step='post', alpha=0.3, \
                 color='k')

plt.xlabel('Полнота')
plt.ylabel('Точность')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])

plt.title('Кривая "точность - полнота": средняя точность = \
    {0:0.2f}'.format(average_precision))
```

Кривая "точность — полнота" показана на рис. 2.4. Учитывая все вышесказанное, мы видим, что можно достичь приблизительно 80% полноты классификации (т.е. зафиксировать 80% поддельных транзакций) при приблизительно 70% точности (означающей, что из всех транзакций, помеченных моделью как поддельные, 70% действительно являются поддельными, тогда как оставшиеся 30% были ошибочно помечены как таковые).

Рис. 2.4. Кривая "точность — полнота" логистической регрессии

Эту кривую можно свести к единственному числу, рассчитав среднюю точность, значение которой для данной модели логистической регрессии составляет `0.74`. И все же, ввиду отсутствия сравнения с другими моделями, мы пока что ничего не можем сказать о том, насколько хороша такая точность.

А теперь давайте измерим показатель auROC.

```
fpr, tpr, thresholds = roc_curve(preds['trueLabel'], \
                                 preds['prediction'])

areaUnderROC = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color='r', lw=2, label='ROC-кривая')
plt.plot([0, 1], [0, 1], color='k', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Доля ложноположительных исходов')
plt.ylabel('Доля истинноположительных исходов')
plt.title('    Рабочая характеристика приемника: \n \
          площадь под кривой = {0:0.2f}'.format(areaUnderROC))
plt.legend(loc="lower right")
plt.show()
```

В соответствии с рис. 2.5 значение показателя auROC для данной кривой составляет 0.97. Эта метрика — еще один способ оценить приемлемость модели логистической регрессии, позволяющий определить, какую долю поддельных операций можно выявить при сохранении доли ложноположительных результатов на минимально возможном уровне. Как и в случае средней точности, мы не знаем, является ли значение показателя auROC для данной кривой, равное 0.97, хорошим или плохим, но сможем выяснить это, как только сравним данный результат с аналогичными результатами других моделей.

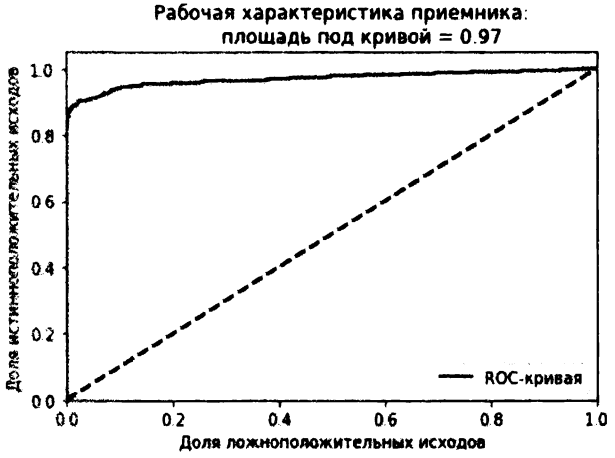


Рис. 2.5. Кривая auROC логистической регрессии

Модели машинного обучения (часть II)

Чтобы сравнить пригодность модели логистической регрессии, построим несколько других моделей, использующих иные алгоритмы обучения с учителем.

Модель №2: случайные леса

Начнем с модели случайных лесов.

Как и в случае логистической регрессии, установим гиперпараметры, обучим модель и оценим результаты, используя кривую "точность — полнота" и показатель auROC.

Настройка гиперпараметров

```
n_estimators = 10
max_features = 'auto'
max_depth = None
min_samples_split = 2
min_samples_leaf = 1
min_weight_fraction_leaf = 0.0
max_leaf_nodes = None
bootstrap = True
oob_score = False
n_jobs = -1
random_state = 2018
class_weight = 'balanced'

RFC = RandomForestClassifier(n_estimators=n_estimators,
    max_features=max_features, max_depth=max_depth,
    min_samples_split=min_samples_split,
    min_samples_leaf=min_samples_leaf,
    min_weight_fraction_leaf=min_weight_fraction_leaf,
    max_leaf_nodes=max_leaf_nodes, bootstrap=bootstrap,
    oob_score=oob_score, n_jobs=n_jobs, random_state=random_state,
    class_weight=class_weight)
```

Начнем с установки значений гиперпараметров. Количество *оценщиков* (estimators) задается равным 10. Иначе говоря, мы построим 10 деревьев и усредним полученные для них результаты. Для каждого дерева будет учитываться квадратный корень из общего количества признаков (в данном случае квадратный корень из 30, равный 5 при округлении вниз).

В результате задания для гиперпараметра `max_depth` значения `None` деревья будут стремиться к максимально возможному росту, разветвляясь в как можно большей степени при заданном подмножестве признаков. По аналогии с тем, как мы поступали в отношении логистической регрессии, установим для параметра `random_state` значение `2018`, чтобы обеспечить воспроизводимость результатов, и, учитывая несбалансированность нашего набора данных, значение `'balanced'` для параметра `class_weight`.

Тренировка модели

Мы будем запускать *k*-мерную кросс-проверку пять раз и сохранять предсказания, каждый раз обучая модель на четырех пятых тренировочного набора данных и используя для предсказаний оставшуюся пятую часть.

```
trainingScores = []
cvScores = []
predictionsBasedOnKFolds = pd.DataFrame(data=[], \
                                        index=y_train.index, \
                                        columns=[0, 1])

model = RFC

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
    X_train_fold, X_cv_fold = X_train.iloc[train_index, :], \
        X_train.iloc[cv_index, :]
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    model.fit(X_train_fold, y_train_fold)
    loglossTraining = log_loss(y_train_fold, \
        model.predict_proba(X_train_fold)[:, 1])
    trainingScores.append(loglossTraining)

    predictionsBasedOnKFolds.loc[X_cv_fold.index, :] = \
        model.predict_proba(X_cv_fold)
    loglossCV = log_loss(y_cv_fold, \
        predictionsBasedOnKFolds.loc[X_cv_fold.index, 1])
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения: ', loglossTraining)
    print('Логарифмические потери валидации: ', loglossCV)
```

```
loglossRandomForestsClassifier = log_loss(y_train, \
    predictionsBasedOnKFolds.loc[:, 1])
print('Логарифмические потери случайных лесов: ', \
    loglossRandomForestsClassifier)
```

Оценка результатов

Логарифмические потери в процессе тренировки и кросс-проверки приведены ниже.

```
Логарифмические потери обучения:    0.0003951763883952557
Логарифмические потери валидации:   0.014479198936303003
Логарифмические потери обучения:    0.0004501221178398935
Логарифмические потери валидации:   0.005712702421375242
Логарифмические потери обучения:    0.00043128813023860164
Логарифмические потери валидации:   0.00908372752510077
Логарифмические потери обучения:    0.0004341676022058672
Логарифмические потери валидации:   0.013491161736979267
Логарифмические потери обучения:    0.0004275530435950083
Логарифмические потери валидации:   0.009963232439211515
```

Обратите внимание на то, что логарифмические потери обучения значительно меньше логарифмических потерь кросс-проверки, что указывает на определенную степень переобучения классификатора на основе случайных лесов в процессе тренировки.

Логарифмические потери по всему тренировочному набору (полученные с использованием кросс-проверочных предсказаний) приведены ниже:

```
Логарифмические потери случайных лесов: 0.010546004611793962
```

Несмотря на наличие эффектов переобучения, валидационные логарифмические потери случайных лесов составляют примерно одну десятую потерь логистической регрессии — значительное улучшение по сравнению с предыдущим решением. Модель случайных лесов проявляет себя с лучшей стороны в отношении корректного выявления попыток мошенничества при выполнении операций с банковскими картами.

Кривая "точность — полнота" для случайных лесов представлена на рис. 2.6. Как видите, модель в состоянии обнаруживать примерно 80% всех попыток мошенничества с приблизительно 80%-ной точностью. Это более впечатляющий результат по сравнению с аналогичными показателями для логистической регрессии, значения которых составляют 80% и 70% соответственно.

Рис. 2.6. Кривая "точность — полнота" модели случайных лесов

Средняя точность для модели случайных лесов, равная `0.79`, представляет собой явное улучшение по сравнению со средней точностью модели логистической регрессии, равной `0.74`. В то же время кривая auROC (рис. 2.7) ведет себя несколько хуже, на что указывает сравнение показателей `0.93` и `0.97` для моделей случайных лесов и логистической регрессии соответственно.

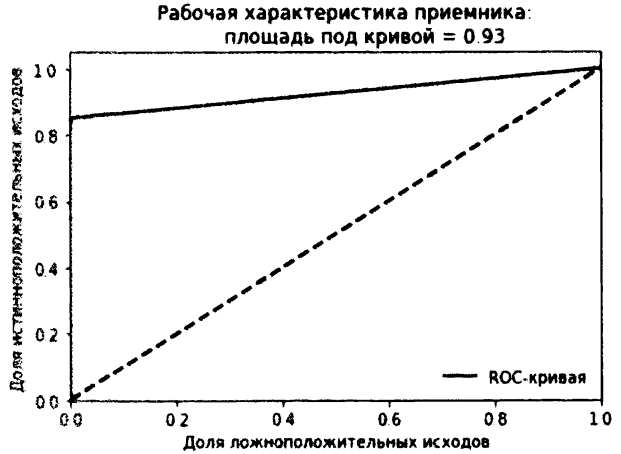


Рис. 2.7. Кривая auROC модели случайных лесов

Модель №3: машина градиентного бустинга XGBoost

Следующее, чем мы займемся, — это тренировка модели с помощью градиентного бустинга и оценка полученных результатов. Существуют два популярных варианта градиентного бустинга: XGBoost и более быстрая версия под

названием LightGBM, разработанная компанией Microsoft. Оба этих варианта будут испытаны нами для построения модели. Начнем с рассмотрения градиентного бустинга XGBoost[9].

Настройка гиперпараметров

Настроим гиперпараметры для задачи бинарной классификации, используя логарифмические потери в качестве функции потерь. Установим для параметра `max_depth` значение 6. Для каждого дерева мы используем все наблюдения и признаки; эти настройки задаются по умолчанию. Чтобы обеспечить воспроизводимость результатов, установим для параметра `random_state` значение 2018.

```
params_xGB = {
    'nthread': 16,              # количество основных потоков
    'gamma': 0,                 # диапазон от 0 до бесконечности,
                                # по умолчанию 0; при увеличении
                                # снижается сложность (растет
                                # смещение, уменьшается дисперсия)
    'max_depth': 6,             # диапазон от 1 до бесконечности,
                                # по умолчанию 6
    'min_child_weight': 1,      # диапазон от 0 до бесконечности,
                                # по умолчанию 1
    'max_delta_step': 0,        # диапазон от 0 до бесконечности,
                                # по умолчанию 0
    'subsample': 1.0,           # диапазон от 0 до 1, по умолчанию 1;
                                # степень субдискретизации
                                # тренировочных примеров
    'colsample_bytree': 1.0,    # диапазон от 0 до 1, по умолчанию 1;
                                # степень субдискретизации признаков
    'objective': 'binary:logistic',
    'num_class': 1,
    'eval_metric': 'logloss',
    'seed': 2018
}
```

[9] Более подробную информацию о градиентном бустинге XGBoost можно найти на сайте GitHub (`https://github.com/dmlc/xgboost`).

Тренировка модели

Как и прежде, используем *k*-кратную кросс-проверку, выполняя тренировку на различных частях исходного набора данных размером четыре пятых каждая и используя для предсказаний оставшуюся пятую часть.

На каждом из прогонов тренировка модели градиентного бустинга циклически выполняется две тысячи раз с оценкой того, уменьшаются ли при этом кросс-проверочные логарифмические потери. Если дальнейшего улучшения данного показателя (по сравнению с предыдущими двумястами итерациями) не наблюдается, то процесс обучения прекращается во избежание переобучения. Результаты тренировочного процесса слишком длинные, чтобы приводить их здесь, но вы сможете ознакомиться с ними на сайте GitHub (`http://bit.ly/2Gd4v7e`).

```
trainingScores = []
cvScores = []
predictionsBasedOnKFolds = pd.DataFrame(data=[], \
    index=y_train.index, columns=['prediction'])

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
    X_train_fold, X_cv_fold = X_train.iloc[train_index, :], \
        X_train.iloc[cv_index, :]
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    dtrain = xgb.DMatrix(data=X_train_fold, label=y_train_fold)
    dCV = xgb.DMatrix(data=X_cv_fold)

    bst = xgb.cv(params_xGB, dtrain, num_boost_round=2000, \
        nfold=5, early_stopping_rounds=200, verbose_eval=50)

    best_rounds = np.argmin(bst['test-logloss-mean'])
    bst = xgb.train(params_xGB, dtrain, best_rounds)

    loglossTraining = log_loss(y_train_fold, bst.predict(dtrain))
    trainingScores.append(loglossTraining)

    predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction']= \
        bst.predict(dCV)
    loglossCV = log_loss(y_cv_fold, \
        predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'])
```

```
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения: ', loglossTraining)
    print('Логарифмические потери валидации: ', loglossCV)

loglossXGBoostGradientBoosting = \
    log_loss(y_train, predictionsBasedOnKFolds.loc[:, 'prediction'])
print('Логарифмические потери градиентного бустинга XGBoost: ',\
      loglossXGBoostGradientBoosting)
```

Оценка результатов

Приведенные ниже результаты свидетельствуют о том, что логарифмические потери, вычисленные по всему тренировочному набору (с использованием кросс-проверочных предсказаний), составляют одну пятую от потерь модели случайных лесов и одну пятидесятую от потерь модели логистической регрессии. Это существенное улучшение по сравнению с двумя предыдущими моделями.

```
Логарифмические потери градиентного бустинга XGBoost:
0.0029566906288156715
```

Как видно из рис. 2.8, средняя точность равна `0.83`. Это близко к точности модели случайных лесов (`0.79`) и представляет собой значительное улучшение по сравнению с логистической регрессией (`0.74`).

Рис. 2.8. Кривая "точность — полнота" градиентного бустинга XGBoost

Значение показателя auROC для ROC-кривой (рис. 2.9), равное `0.97`, то же, что и для логистической регрессии (`0.97`), но демонстрирует улучшение по

сравнению со значением аналогичного показателя (`0.93`) для модели случайных лесов. Пока что, судя по логарифмическим потерям, поведению кривой "точность — полнота" и показателю auROC, модель на основе градиентного бустинга оказалась наилучшей из трех рассмотренных до сих пор моделей.

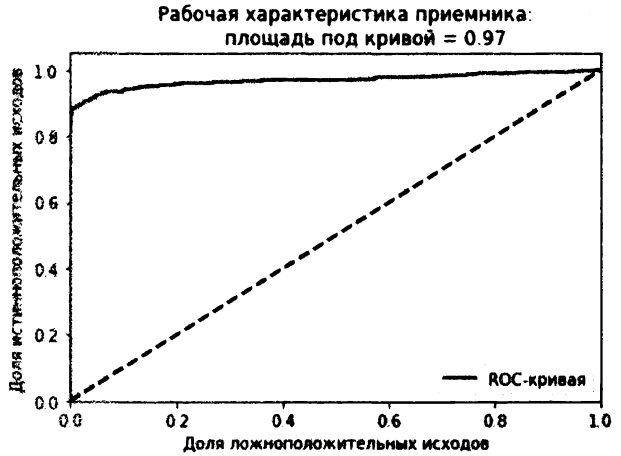


Рис. 2.9. Кривая auROC градиентного бустинга XGBoost

Модель №4: машина градиентного бустинга LightGBM

А теперь проведем тренировку модели, используя другую версию градиентного бустинга: LightGBM[10].

Настройка гиперпараметров

Настроим гиперпараметры для задачи бинарной классификации, используя логарифмические потери в качестве функции потерь. Установим для параметров `max_depth` и `learning_rate` значения `4` и `0.01` соответственно. Для каждого дерева мы используем все наблюдения и признаки; эти настройки задаются по умолчанию. Кроме того, мы зададим количество листьев в одном дереве равным `31` и установим для параметра `bagging_seed` значение `2018`, чтобы обеспечить воспроизводимость результатов.

```
params_lightGB = {
    'task': 'train',
    'application': 'binary',
    'num_class': 1,
```

[10] Более подробную информацию о градиентном бустинге LightGBM компании Microsoft можно найти на сайте GitHub (`https://github.com/Microsoft/LightGBM`).

```
    'boosting': 'gbdt',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'metric_freq': 50,
    'is_training_metric': False,
    'max_depth': 4,
    'num_leaves': 31,
    'learning_rate': 0.01,
    'feature_fraction': 1.0,
    'bagging_fraction': 1.0,
    'bagging_freq': 0,
    'bagging_seed': 2018,
    'verbose': 0,
    'num_threads': 16
}
```

Тренировка модели

Как и прежде, используем *k*-кратную кросс-проверку в пяти прогонах программы, сохраняя предсказания для валидационных наборов.

```
trainingScores = []
cvScores = []
predictionsBasedOnKFolds = pd.DataFrame(data=[], \
    index=y_train.index, columns=['prediction'])

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
    X_train_fold, X_cv_fold = X_train.iloc[train_index, :], \
        X_train.iloc[cv_index, :]
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    lgb_train = lgb.Dataset(X_train_fold, y_train_fold)
    lgb_eval = lgb.Dataset(X_cv_fold, y_cv_fold, reference=lgb_train)
    gbm = lgb.train(params_lightGB, lgb_train, \
                    num_boost_round=2000, valid_sets=lgb_eval, \
                    early_stopping_rounds=200)

    loglossTraining = log_loss(y_train_fold, \
        gbm.predict(X_train_fold, num_iteration=gbm.best_iteration))
    trainingScores.append(loglossTraining)
```

```
    predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'] = \
        gbm.predict(X_cv_fold, num_iteration=gbm.best_iteration)
    loglossCV = log_loss(y_cv_fold, \
       predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'])
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения: ', loglossTraining)
    print('Логарифмические потери валидации: ', loglossCV)

loglossLightGBMGradientBoosting = \
    log_loss(y_train, predictionsBasedOnKFolds.loc[:, 'prediction'])
print('Логарифмические потери градиентного бустинга LightGBM: ', \
    loglossLightGBMGradientBoosting)
```

На каждом из прогонов тренировка модели градиентного бустинга циклически выполняется две тысячи раз с оценкой того, уменьшаются ли при этом кросс-проверочные логарифмические потери. Если дальнейшего улучшения данного показателя (по сравнению с предыдущими двумястами итерациями) не наблюдается, то процесс обучения прекращается во избежание переобучения. Результаты тренировочного процесса слишком длинные, чтобы приводить их здесь, но вы сможете ознакомиться с ними на сайте GitHub (`http://bit.ly/2Gd4v7e`).

Оценка результатов

Приведенные ниже результаты свидетельствуют о том, что логарифмические потери, вычисленные по всему тренировочному набору (с использованием кросс-проверочных предсказаний), близки к тем, которые были получены с помощью модели XGBoost, и составляют одну пятую от потерь модели случайных лесов и одну пятидесятую от потерь модели логистической регрессии. В то же время версия LightGBM работает намного быстрее версии XGBoost:

```
Логарифмические потери градиентного бустинга LightGBM:
0.0029732268054261826
```

В соответствии с рис. 2.10 средняя точность составляет `0.83`, что совпадает с точностью XGBoost (`0.83`), превышает точность модели случайных лесов (`0.79`) и представляет собой значительное улучшение по сравнению с логистической регрессией (`0.74`).

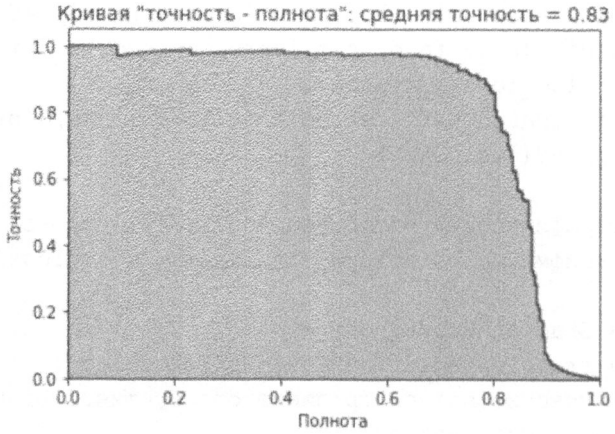


Рис. 2.10. Кривая "точность — полнота" градиентного бустинга LightGBM

В соответствии с рис. 2.11 показатель auROC для ROC-кривой равен `0.98`, что представляет собой улучшение по сравнению с моделями XGBoost (`0.97`), логистической регрессии (`0.97`) и случайных лесов (`0.93`).

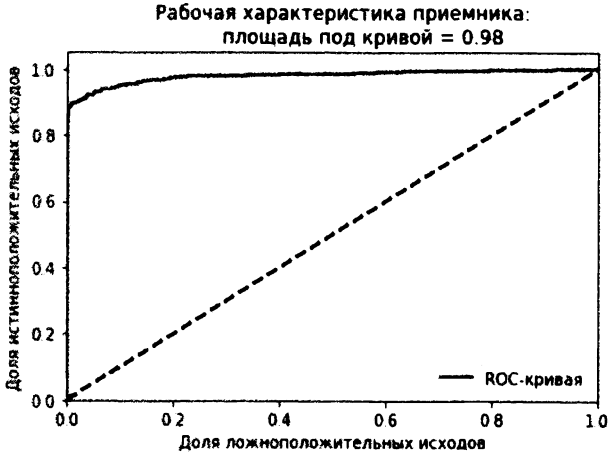


Рис. 2.11. Кривая auROC градиентного бустинга LightGBM

Оценка четырех моделей с помощью тестового набора

К этому моменту мы научились:

- настраивать среду для проектов машинного обучения;
- получать, загружать, исследовать, очищать и визуализировать данные;
- разбивать набор данных на тренировочный и тестовый наборы и формировать наборы для *k*-кратных валидационных тестов;
- выбирать подходящую функцию потерь;
- задавать гиперпараметры, обучать модель и выполнять кросс-проверочные тесты;
- оценивать результаты.

Мы не исследовали вопрос о том, как регулировать гиперпараметры для улучшения результатов каждого из решений и уменьшения эффектов недо/переобучения, однако вам будет совсем несложно самостоятельно выполнить соответствующие эксперименты, воспользовавшись кодом, доступным на сайте GitHub (`http://bit.ly/2Gd4v7e`).

Но даже без тонкой настройки гиперпараметров полученные результаты красноречиво свидетельствуют о том, что наилучшим решением оказывается градиентный бустинг LightGBM, от которого ненамного отстает XGBoost. Худшими оказались решения на основе случайных лесов и логистической регрессии.

Для окончательной оценки каждой из четырех моделей воспользуемся тестовым набором.

Мы будем предсказывать вероятности подделок на тестовом наборе транзакций, используя каждую из обученных моделей. Затем мы вычислим логарифмические потери для каждой модели, сравнивая предсказанные вероятности подделок с истинными метками.

```
predictionsTestSetLogisticRegression = \
    pd.DataFrame(data=[], index=y_test.index, columns=['prediction'])
predictionsTestSetLogisticRegression.loc[:, 'prediction'] = \
    logReg.predict_proba(X_test)[:, 1]
logLossTestSetLogisticRegression = \
    log_loss(y_test, predictionsTestSetLogisticRegression)
```

```
predictionsTestSetRandomForests = \
   pd.DataFrame(data=[], index=y_test.index, columns=['prediction'])
predictionsTestSetRandomForests.loc[:, 'prediction'] = \
    RFC.predict_proba(X_test)[:, 1]
logLossTestSetRandomForests = \
    log_loss(y_test, predictionsTestSetRandomForests)

predictionsTestSetXGBoostGradientBoosting = \
   pd.DataFrame(data=[], index=y_test.index, columns=['prediction'])
dtest = xgb.DMatrix(data=X_test)
predictionsTestSetXGBoostGradientBoosting.loc[:, 'prediction'] = \
    bst.predict(dtest)
logLossTestSetXGBoostGradientBoosting = \
    log_loss(y_test, predictionsTestSetXGBoostGradientBoosting)

predictionsTestSetLightGBMGradientBoosting = \
   pd.DataFrame(data=[], index=y_test.index, columns=['prediction'])
predictionsTestSetLightGBMGradientBoosting.loc[:, 'prediction'] = \
    gbm.predict(X_test, num_iteration=gbm.best_iteration)
logLossTestSetLightGBMGradientBoosting = \
    log_loss(y_test, predictionsTestSetLightGBMGradientBoosting)
```

Приведенная ниже сводка логарифмических потерь не содержит сюрпризов: наименьшие логарифмические потери продемонстрировал градиентный бустинг LightGBM, оставивший позади всех остальных.

```
Логарифмические потери логистической регрессии на тестовом наборе:
0.123732961313
Логарифмические потери случайных лесов на тестовом наборе:
0.00918192757674
Логарифмические потери градиентного бустинга XGBoost на тестовом
наборе: 0.00249116807943
Логарифмические потери градиентного бустинга LightGBM на тестовом
наборе: 0.002376320092424
```

На рис. 2.12–2.19 представлены кривые "точность — полнота", средние значения точности и ROC-кривые для всех четырех моделей, построенные на основании полученных выше результатов.

Рис. 2.12. Кривая "точность — полнота" логистической регрессии на тестовом наборе

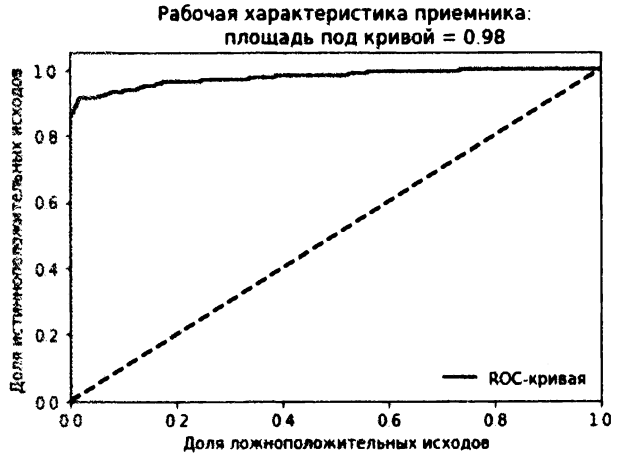


Рис. 2.13. Кривая auROC логистической регрессии на тестовом наборе

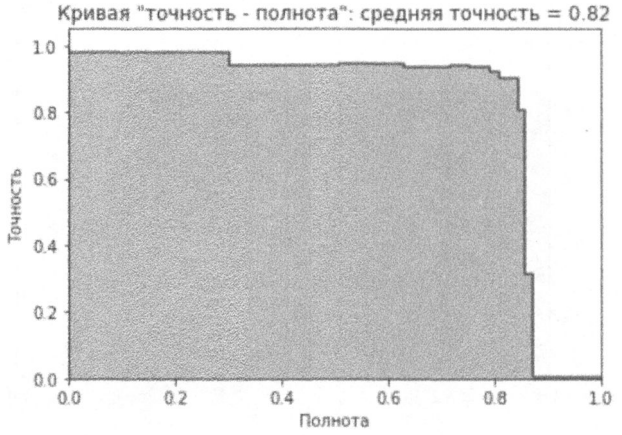


Рис. 2.14. Кривая "точность — полнота" модели случайных лесов на тестовом наборе

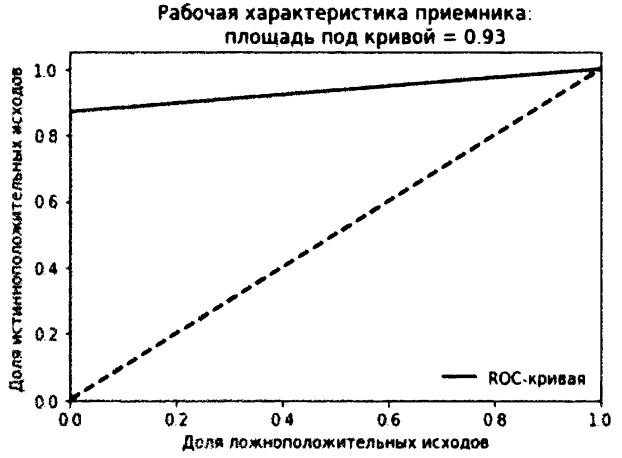


Рис. 2.15. Кривая auROC модели случайных лесов на тестовом наборе

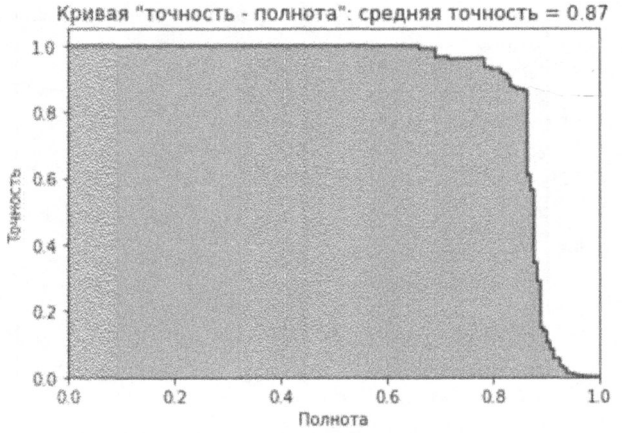


Рис. 2.16. Кривая "точность — полнота" градиентного бустинга XGBoost на тестовом наборе

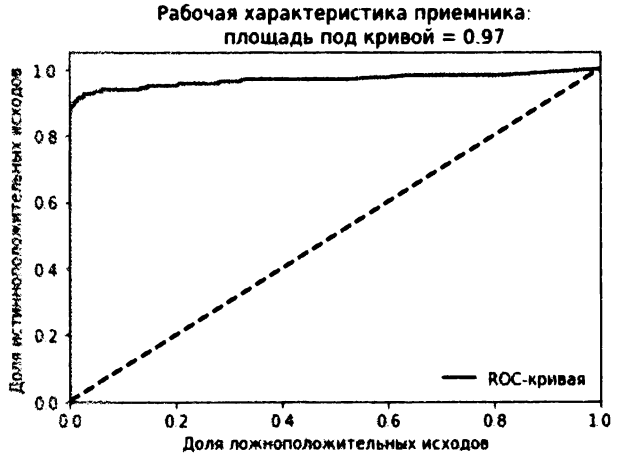


Рис. 2.17. Кривая auROC градиентного бустинга XGBoost на тестовом наборе

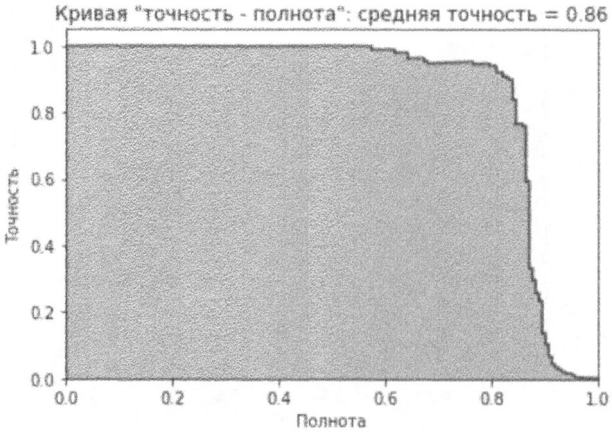


Рис. 2.18. Кривая "точность — полнота" градиентного бустинга LightGBM на тестовом наборе

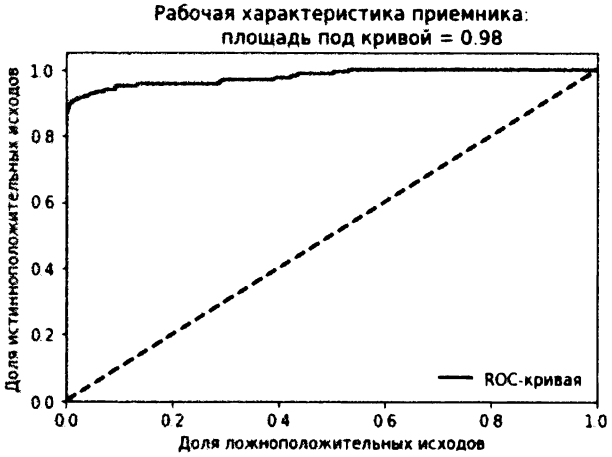


Рис. 2.19. Кривая auROC градиентного бустинга LightGBM на тестовом наборе

Результаты градиентного бустинга LightGBM впечатляют: нам удалось выявить свыше 80% поддельных транзакций с почти 90%-ной точностью (другими словами, наряду с фиксацией 80% общего количества поддельных транзакций модель LightGBM ошиблась лишь в 10% случаев).

Учитывая малочисленность примеров поддельных транзакций в нашем наборе данных, такой результат является замечательным достижением.

Ансамбли

Вместо того чтобы задействовать лишь один из алгоритмов машинного обучения, мы можем оценить, приведет ли использование ансамбля моделей к повышению эффективности обнаружения подделок[11].

В общем случае образование ансамбля одинаково сильных решений из различных семейств алгоритмов машинного обучения (например, одного из семейства случайных лесов и одного из семейства нейронных сетей) должно приводить к улучшению результата по сравнению с отдельными решениями. Это объясняется тем, что каждое из автономных решений имеет свои сильные и слабые стороны. Объединяя автономные решения в ансамбль, мы добиваемся того, что сильные стороны одних моделей компенсируют слабые стороны других.

Однако не следует забывать об одном важном моменте. В случае решений примерно равной силы производительность ансамбля будет превышать производительность любого из них. Но если одно из решений намного эффективнее других, то производительность ансамбля будет определяться производительностью наиболее эффективного автономного решения; остальные решения не будут вносить в производительность ансамбля никакого вклада.

Кроме того, между автономными решениями должна (в разумной степени) отсутствовать корреляция. При наличии значительной корреляции между ними сильные стороны одного решения будут зеркальным отражением сильных сторон других, и то же самое можно сказать об их слабых сторонах. Диверсификация сильных и слабых сторон автономных решений в рамках ансамбля принесет лишь минимальную выгоду.

Стекинг

В нашей задаче две модели (градиентный бустинг LightGBM и XGBoost) значительно превосходят две другие (модели случайных лесов и логистической регрессии). Но две наиболее сильные модели относятся к одному семейству, а это означает, что их сильные и слабые стороны демонстрируют сильную корреляцию.

[11] Более подробную информацию об ансамблевом обучении можно найти в статьях *Kaggle Ensembling Guide* (`https://mlwave.com/kaggle-ensembling-guide/`) и *Introduction to Ensembling/Stacking in Python* (`http://bit.ly/2RYV4iF`).

Чтобы выяснить, возможно ли добиться улучшения по сравнению с изученными ранее автономными моделями, мы можем использовать стекинг (разновидность ансамбля). При этом мы берем предсказания, полученные посредством *k*-кратной кросс-проверки каждой из четырех автономных моделей (*предсказания первого слоя*), и присоединяем их к исходному тренировочному набору данных. После этого мы проводим тренировку на наборе данных, содержащем исходные признаки плюс предсказания первого слоя, используя *k*-кратную кросс-проверку.

Это приведет к новому набору предсказаний, полученных с помощью *k*-кратной кросс-проверки (*предсказания второго слоя*), который мы будем оценивать для выяснения того, достигается ли улучшение по сравнению с любой из автономных моделей.

Комбинирование предсказаний первого слоя с оригинальным тренировочным набором

Прежде всего объединим предсказания каждой из четырех моделей машинного обучения, которые мы построили с использованием исходного тренировочного набора данных.

```
predictionsBasedOnKFoldsFourModels = pd.DataFrame(data=[], \
    index=y_train.index)
predictionsBasedOnKFoldsFourModels = \
    predictionsBasedOnKFoldsFourModels.join(
        predictionsBasedOnKFoldsLogisticRegression['prediction'] \
        .astype(float), how='left').join( \
        predictionsBasedOnKFoldsRandomForests['prediction'] \
        .astype(float), how='left', rsuffix="2").join( \
        predictionsBasedOnKFoldsXGBoostGradientBoosting[ \
        'prediction'].astype(float), how='left', rsuffix="3").join( \
        predictionsBasedOnKFoldsLightGBMGradientBoosting[ \
        'prediction'].astype(float), how='left', rsuffix="4")
predictionsBasedOnKFoldsFourModels.columns = ['predsLR', \
    'predsRF', 'predsXGB', 'predsLightGBM']

X_trainWithPredictions = \
    X_train.merge(predictionsBasedOnKFoldsFourModels, \
                  left_index=True, right_index=True)
```

Настройка гиперпараметров

Далее мы задействуем градиентный бустинг LightGBM — наилучший из используемых в предыдущих упражнениях алгоритмов — для обучения на наборе, содержащем оригинальные признаки плюс предсказания первого слоя. Гиперпараметры остаются прежними.

```
params_lightGB = {
    'task': 'train',
    'application': 'binary',
    'num_class': 1,
    'boosting': 'gbdt',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'metric_freq': 50,
    'is_training_metric': False,
    'max_depth': 4,
    'num_leaves': 31,
    'learning_rate': 0.01,
    'feature_fraction': 1.0,
    'bagging_fraction': 1.0,
    'bagging_freq': 0,
    'bagging_seed': 2018,
    'verbose': 0,
    'num_threads': 16
}
```

Тренировка модели

Как и раньше, мы будем использовать *k*-кратную кросс-проверку и генерировать вероятности поддельных транзакций для пяти различных кросс-проверочных наборов.

```
trainingScores = []
cvScores = []
predictionsBasedOnKFoldsEnsemble = pd.DataFrame(data=[], \
    index=y_train.index, columns=['prediction'])

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
    X_train_fold, X_cv_fold = \
        X_trainWithPredictions.iloc[train_index, :], \
        X_trainWithPredictions.iloc[cv_index, :]
```

```
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    lgb_train = lgb.Dataset(X_train_fold, y_train_fold)
    lgb_eval = lgb.Dataset(X_cv_fold, y_cv_fold, reference=lgb_train)
    gbm = lgb.train(params_lightGB, lgb_train, \
                    num_boost_round=2000, valid_sets=lgb_eval, \
                    early_stopping_rounds=200)

    loglossTraining = log_loss(y_train_fold, \
        gbm.predict(X_train_fold, num_iteration=gbm.best_iteration))
    trainingScores.append(loglossTraining)

    predictionsBasedOnKFoldsEnsemble.loc[X_cv_fold.index, \
        'prediction'] = gbm.predict(X_cv_fold, \
        num_iteration=gbm.best_iteration)
    loglossCV = log_loss(y_cv_fold, \
        predictionsBasedOnKFoldsEnsemble.loc[X_cv_fold.index, \
        'prediction'])
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения: ', loglossTraining)
    print('Логарифмические потери валидации: ', loglossCV)

loglossEnsemble = log_loss(y_train, \
    predictionsBasedOnKFoldsEnsemble.loc[:, 'prediction'])
print('Логарифмические потери ансамбля: ', loglossEnsemble)
```

Оценка результатов

Приведенные ниже результаты не демонстрируют никакого улучшения по сравнению с предыдущими. Логарифмические потери ансамбля весьма близки к логарифмическим потерям градиентного бустинга. Улучшение результатов не наблюдается, поскольку наилучшие автономные решения принадлежат к одному и тому же семейству (градиентный бустинг). Их сильные и слабые стороны характеризуются высокой степенью корреляции, поэтому диверсификация по моделям не приносит никакого выигрыша:

```
Логарифмические потери ансамбля: 0.002885415974220497
```

Приведенные на рис. 2.20 и 2.21 кривая "точность — полнота", средняя точность и кривая auROC также подтверждают отсутствие улучшения.

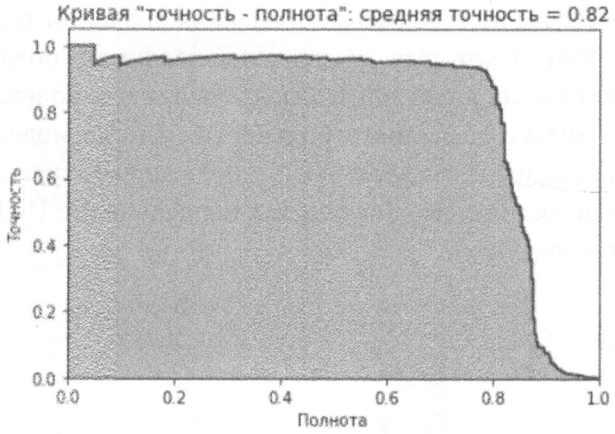


Рис. 2.20. Кривая "точность — полнота" для ансамбля

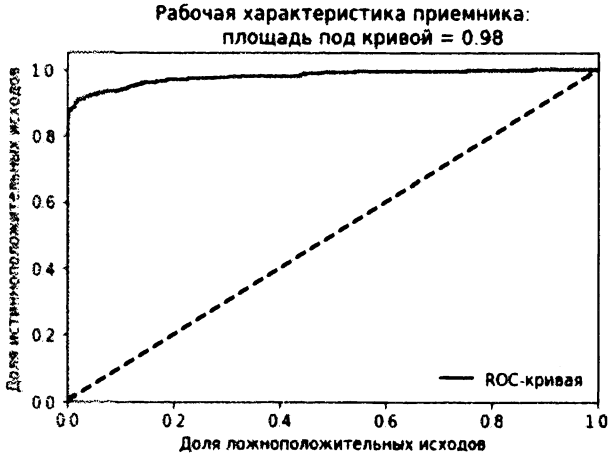


Рис. 2.21. Кривая auROC для ансамбля

Выбор окончательной модели

Поскольку привлечение ансамбля не приводит к улучшению результатов, мы отдаем предпочтение простоте автономной модели градиентного бустинга LightGBM и будем использовать ее в производственных целях.

Прежде чем приступить к организации конвейера для работы с новыми транзакциями, визуально проанализируем, насколько хорошо модель LightGBM отличает поддельные транзакции от нормальных для тестового набора.

На рис. 2.22 значения предсказанных вероятностей отложены вдоль оси X. Приведенный график дает основания утверждать, что модель довольно

хорошо справляется с приписыванием высоких значений вероятности подделки тем транзакциям, которые действительно являются поддельными. И наоборот, транзакциям, не являющимся поддельными, в большинстве случаев приписываются низкие значения вероятности. Иногда модель ошибается и приписывает низкие значения вероятности поддельным транзакциям и высокие значения транзакциям, не являющимся поддельными. Но в целом результаты весьма впечатляющие.

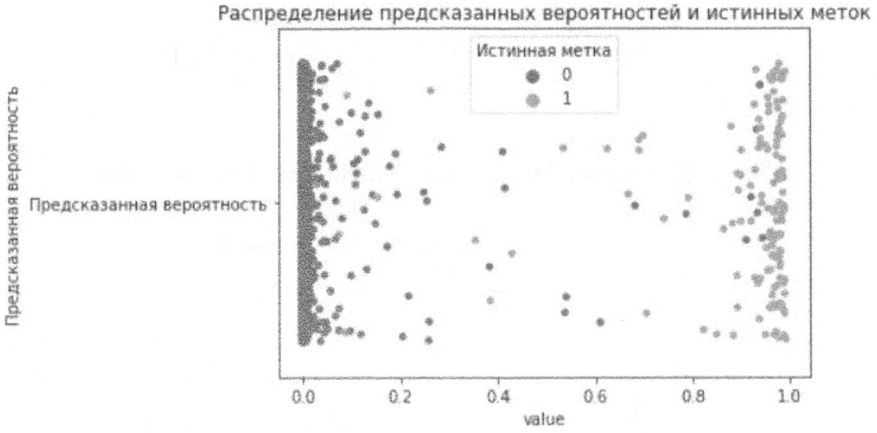


Рис. 2.22. График распределения предсказанных вероятностей и истинных меток

Производственный конвейер

Выбрав модель, которая будет применяться на практике, наметим простой производственный конвейер, включающий три стадии обработки новых данных: загрузку, масштабирование признаков и генерирование предсказаний с помощью модели LightGBM, которую мы уже обучили и выбрали для использования в производственной среде.

```
'''Конвейер для новых данных'''
# Во-первых, импортируем новые данные во фрейм 'newData'.
# Во-вторых, масштабируем данные.
# newData.loc[:, featuresToScale] = \
#     sX.transform(newData[featuresToScale])
# В-третьих, создаем предсказания, используя LightGBM.
# gbm.predict(newData, num_iteration=gbm.best_iteration)
```

Сгенерировав предсказания, аналитик сможет сфокусировать свое внимание на транзакциях, для которых получена наиболее высокая вероятность

того, что они являются поддельными, и приступить к работе с соответствующим списком. Или же, если целью служит автоматизация этого процесса, аналитик сможет использовать систему, автоматически отклоняющую транзакции, для которых предсказанная вероятность мошенничества превышает заданное пороговое значение.

Например, если, опираясь на рис. 2.18, мы будем автоматически отклонять транзакции с предсказанной вероятностью `0.9`, то среди них почти наверняка будут только поддельные транзакции, а риск случайного отклонения нормальной транзакции будет сведен к минимуму.

Резюме

Примите поздравления! Используя обучение с учителем, вы создали систему, позволяющую выявлять попытки выполнения мошеннических операций с банковскими картами.

Для этого вам пришлось настроить среду машинного обучения, получить и подготовить данные, обучить и оценить ряд моделей, сделать окончательный выбор производственной модели и организовать конвейер для обработки новых транзакций. Таким образом, вы успешно разработали приложение машинного обучения.

Далее мы используем аналогичный подход для разработки приложений на основе обучения без учителя.

Приведенное выше решение со временем потребует повторного обучения в связи с появлением новых способов выполнения мошеннических операций с банковскими картами. Кроме того, нам следовало бы подыскать другие алгоритмы машинного обучения, принадлежащие к другим семействам и сравнимые по производительности с градиентным бустингом, и включить их в ансамбль для повышения общей эффективности системы в отношении обнаружения попыток мошенничества.

Наконец, в реальных приложениях машинного обучения большую роль играет легкость интерпретации результатов. Поскольку в рассмотренном нами наборе данных признаки являются выходом PCA (один из методов снижения размерности, который исследуется в главе 3), мы не можем дать словесное объяснение тому, почему некоторые транзакции помечены как потенциально мошеннические. Для улучшения интерпретируемости результатов нам нужен доступ к исходным признакам, подающимся на вход PCA, которые отсутствуют в нашем простом наборе данных.

Обучение без учителя с использованием библиотеки Scikit-learn

В следующих главах мы познакомимся с двумя основными концепциями обучения без учителя — снижение размерности и кластеризация — и задействуем их для обнаружения аномалий и сегментирования групп.

Как обнаружение аномалий, так и сегментирование групп находят широкое применение во многих областях.

Обнаружение аномалий применяется для эффективного выявления редких событий, таких как мошеннические транзакции, попытки взлома компьютерных систем, терроризм, торговля людьми, оружием и наркотиками, отмывание денег, необычная торговая активность, вспышки болезней или отказы в обслуживании критического оборудования.

Сегментирование групп позволяет изучать поведение пользователей в таких областях, как маркетинг, онлайн-покупки, прослушивание музыки, просмотр видео, сайты знакомств и социальные сети.

ГЛАВА 3

Снижение размерности

В этой главе мы сосредоточимся на рассмотрении одного из главных препятствий на пути к успешному созданию приложений машинного обучения, имя которому — *проклятие размерности*. В методе обучения без учителя реализуется замечательная контрмера: *снижение размерности*. Мы ознакомимся с данной концепцией и реализуем соответствующие решения, чтобы у читателей выработалось понимание того, как это работает.

В главе 4 мы создадим собственное приложение, реализовав обучение без учителя на основе снижения размерности. Это будет система обнаружения мошеннических операций с банковскими картами (в главе 2 мы применили обучение с учителем). Выявление случаев мошенничества подобного типа является частным случаем *обнаружения аномалий* — быстро развивающейся области прикладного машинного обучения без учителя.

Но прежде чем приступить к построению системы обнаружения аномалий, подробно рассмотрим саму концепцию снижения размерности.

Причины снижения размерности

Как отмечалось в главе 1, снижение размерности помогает бороться с наиболее распространенной проблемой машинного обучения — так называемым *проклятием размерности*, когда алгоритмы не могут эффективно и рационально обучаться на данных исключительно ввиду большой размерности пространства признаков.

Алгоритмы снижения размерности проецируют многомерные данные на пространство низкой размерности, удерживая как можно больше существенно важной информации и удаляя избыточную. Как только данные переведены в пространство низкой размерности, алгоритмы машинного обучения получают возможность более эффективно и рационально выявлять шаблоны, представляющие интерес, за счет значительного снижения уровня шумов.

Иногда снижение размерности служит самоцелью — например, для построения системы обнаружения аномалий, что будет продемонстрировано в следующей главе.

Однако в иных ситуациях снижение размерности выступает, скорее, не конечной целью, а средством для достижения другой цели. Например, его часто применяют в конвейере машинного обучения для решения крупномасштабных, вычислительно трудоемких задач, связанных с обработкой изображений, видео и текста.

База данных рукописных цифр MNIST

Прежде чем начать знакомство с алгоритмами снижения размерности, необходимо исследовать набор данных, с которым нам предстоит работать. Это база рукописных цифр MNIST (Mixed National Institute of Standards and Technology) — один из наиболее известных наборов данных в области машинного обучения, применяемый в компьютерном зрении. База находится в свободном доступе на сайте Яна Лекуна (`http://yann.lecun.com/exdb/mnist/`). Для простоты мы будем использовать сериализованную версию, предоставленную на сайте *Deeplearning.net* (`http://deeplearning.net/tutorial/gettingstarted.html`).

База MNIST разбита на три набора: тренировочный (50 000 примеров), валидационный (10 000 примеров) и тестовый (10 000 примеров). Для всех примеров имеются метки.

База содержит изображения рукописных цифр размером 28 × 28 пикселей. Каждая точка данных (т.е. каждое изображение) может передаваться в виде массива чисел, в котором каждое число описывает интенсивность каждого пикселя. Другими словами, изображению размером 28 × 28 пикселей соответствует массив чисел размерностью 28 × 28.

Для упрощения задачи мы можем развернуть каждый массив в 784-мерный (28 × 28) вектор. Каждая компонента вектора — вещественное число в диапазоне 0–1, представляющее интенсивность отдельного пикселя. Нулю соответствует черный цвет пикселя, единице — белый. Метками служат числа от 0 до 9, указывающие, какую именно цифру представляет изображение.

Импорт библиотек

Для начала загрузим необходимые библиотеки.

```
# Импорт библиотек

'''Основные библиотеки'''
import numpy as np
import pandas as pd
```

```
import os, time
import pickle, gzip

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
```

Загрузка наборов данных MNIST

А теперь загрузим наборы данных MNIST.

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'mnist_data', \
                         'mnist.pkl.gz'])
f = gzip.open(current_path+file, 'rb')
train_set, validation_set, test_set = \
    pickle.load(f, encoding='latin1')
f.close()

X_train, y_train = train_set[0], train_set[1]
X_validation, y_validation = validation_set[0], validation_set[1]
X_test, y_test = test_set[0], test_set[1]
```

Верификация формы наборов данных

Верифицируем форму наборов данных. Это даст уверенность в том, что они были загружены правильно.

```
# Верификация формы наборов данных
print("Форма X_train: ", X_train.shape)
print("Форма y_train: ", y_train.shape)
print("Форма X_validation: ", X_validation.shape)
print("Форма y_validation: ", y_validation.shape)
print("Форма X_test: ", X_test.shape)
print("Форма y_test: ", y_test.shape)
```

Результаты подтверждают, что формы наборов данных соответствуют ожидаемым.

```
Форма X_train:       (50000, 784)
Форма y_train:       (50000,)
Форма X_validation:  (10000, 784)
Форма y_validation:  (10000,)
Форма X_test:        (10000, 784)
Форма y_test:        (10000,)
```

Создание структур `DataFrame` библиотеки Pandas на основе наборов данных

Преобразуем массивы `numpy` в структуры данных `DataFrame` библиотеки Pandas, которые более удобны для исследования и работы.

```
# Создадим структуры DataFrame библиотеки Pandas
# на основе наборов данных
train_index = range(0, len(X_train))
validation_index = range(len(X_train), \
                         len(X_train)+len(X_validation))
test_index = range(len(X_train)+len(X_validation), \
                   len(X_train)+len(X_validation)+len(X_test))

X_train = pd.DataFrame(data=X_train, index=train_index)
y_train = pd.Series(data=y_train, index=train_index)

X_validation = pd.DataFrame(data=X_validation, \
                            index=validation_index)
y_validation = pd.Series(data=y_validation, index=validation_index)

X_test = pd.DataFrame(data=X_test, index=test_index)
y_test = pd.Series(data=y_test, index=test_index)
```

Исследование данных

Сгенерируем статистическую сводку по набору.

```
# Описание тренировочной матрицы
X_train.describe()
```

В табл. 3.1 представлена сводка по загруженным данным (показаны только первые столбцы). Многие значения оказались нулевыми, а значит, большинство пикселей в изображениях черные. В этом есть смысл, поскольку все цифры белого цвета и отображаются на черном фоне.

Таблица 3.1. Исследование данных

	0	1	2	3	4	5	6
count	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0	50000.0
mean	0.0	0.0	0.0	0.0	0.0	0.0	0.0
std	0.0	0.0	0.0	0.0	0.0	0.0	0.0
min	0.0	0.0	0.0	0.0	0.0	0.0	0.0
25%	0.0	0.0	0.0	0.0	0.0	0.0	0.0
50%	0.0	0.0	0.0	0.0	0.0	0.0	0.0
75%	0.0	0.0	0.0	0.0	0.0	0.0	0.0
max	0.0	0.0	0.0	0.0	0.0	0.0	0.0

8 rows x 784 columns

Данные меток — это одномериый вектор, представляющий фактическое содержимое изображения. Ниже приведеиы метки для первых нескольких изображений.

```
# Отобразить метки
y_train.head()
```

```
0    5
1    0
2    4
3    1
4    9
dtype: int64
```

Вывод изображений

Определим функцию для просмотра изображения вместе с его меткой.

```
def view_digit(example):
    label = y_train.loc[0]
    image = X_train.loc[example, :].values.reshape([28, 28])
    plt.title('Пример: %d Метка: %d' % (example, label))
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.show()
```

Первое изображение, полученное после преобразования 784-мерного вектора в матрицу размером 28 × 28 пикселей, представляет цифру 5 (рис. 3.1).

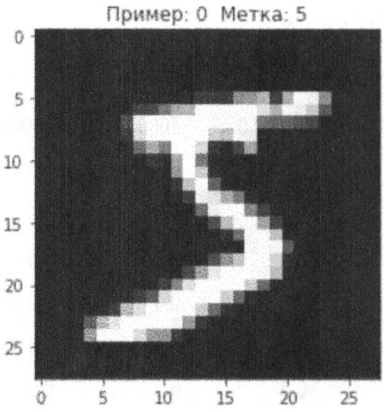


Рис. 3.1. *Вид первой цифры*

Алгоритмы снижения размерности

Теперь, когда набор данных MNIST загружен и исследован, пора перейти к рассмотрению алгоритмов снижения размерности. Сначала мы познакомимся с концепцией каждого алгоритма, а затем применим его к набору данных MNIST, чтобы понять, как работает алгоритм.

Линейное проецирование и многократное обучение

Существуют две основные группы методов снижения размерности. К одной из них относятся методы *линейного проецирования* (linear projection), создающие линейную проекцию данных из высокоразмерного пространства в низкоразмерное. Сюда входят такие методы, как *анализ главных компонент* (principal component analysis — PCA), *сингулярное разложение* (singular value decomposition — SVD) и *случайное проецирование* (random projection).

Ко второй группе относятся методы так называемого *многократного обучения* (manifold learning), *или нелинейного снижения размерности* (nonlinear dimensionality reduction). Сюда, в частности, входит метод *Isomap*, который измеряет расстояния между точками вдоль кривой, определяя *геодезическое*, а не евклидово, расстояние. Среди других методов можно назвать *многомерное масштабирование* (multidimensional scaling — MDS), *локально-линейное вложение* (locally linear embedding — LLE), *стохастическое вложение соседей с t-распределением* (t-distributed stochastic neighbor embedding — t-SNE), *словарное обучение* (dictionary learning), *вложение случайных деревьев* (random trees embedding) и *анализ независимых компонент* (independent component analysis).

Анализ главных компонент

Мы рассмотрим несколько вариантов PCA: *стандартный*, *инкрементный*, *разреженный* и *ядерный*.

Концепция PCA

Начнем с рассмотрения стандартного варианта PCA — одного из наиболее распространенных линейных методов снижения размерности. В PCA алгоритм находит низкоразмерное представление данных, одновременно сохраняя как можно большую долю дисперсии (представляющей наиболее существенную информацию).

PCA достигает этого, беря в расчет корреляцию между признаками. Если внутри некоторого подмножества признаков существует высокая корреляция, то алгоритм будет пытаться объединить коррелирующие признаки и представить эти данные, используя меньшее количество нризнаков, между которыми отсутствует линейная корреляция. Алгоритм продолжает нроцесс сведения корреляции, находя направления максимальной дисперсии среди исходных высокоразмерных данных и проецируя их на пространство меньшей размерности. Эти вновь извлеченные компоненты называются *главными компонентами* (principal components).

С помощью главных компонент возможна реконструкция исходных признаков — пусть неточная, но достаточно близкая к ним. Алгоритм PCA предпринимает активные попытки минимизировать ошибку реконструкции в ходе поиска оптимальных компонент.

В нашем примере с базой данных MNIST пространство исходных признаков имеет 784 измерения, называемых d-измерениями. Алгоритм PCA проецирует данные на меньшее пространство, имеющее k измерений (где $k < d$), в то же время сохраняя как можно больше существенно важной информации. Эти k измерений и есть главные компоненты.

Количество оставляемых нами главных компонент значительно меньше числа измерений в исходном наборе данных. Переходя к этому пространству низкой размерности, мы теряем некоторую долю дисперсии (т.е. часть информации), но зато идентификация базовой структуры данных упрощается, что позволяет более эффективно и рационально решать такие задачи, как обнаружение аномалий и кластеризация.

Более того, уменьшая размерность данных, PCA уменьшает и объем данных, что приводит к повышению производительности алгоритмов машинного

обучения, запускаемых далее в конвейерной цепочке (например, при решении таких задач, как классификация изображений).

Прежде чем запускать алгоритм PCA, очень важно предварительно масштабировать признаки. PCA весьма чувствителен к соотношению диапазонов значений исходных признаков. Как правило, необходимо масштабировать данные, приводя их к одному и тому же диапазону относительных значений. Однако в нашем наборе данных рукописных цифр MNIST признаки уже масштабированы и приведены к диапазону значений от нуля до единицы, так что этот этап можно опустить.

Реализация PCA

Теперь, когда вы лучше поняли принципы работы алгоритма PCA, можно применить этот метод к набору данных MNIST и посмотреть, насколько хорошо он справляется с захватом наиболее существенной информации о цифрах в процессе проецирования данных из исходного 784-мерного пространства в пространство меньшей размерности.

Настройка гиперпараметров

Настроим параметры для алгоритма PCA.

```
from sklearn.decomposition import PCA

n_components = 784
whiten = False
random_state = 2018

pca = PCA(n_components=n_components, whiten=whiten, \
          random_state=random_state)
```

Применение PCA

Мы установим количество главных компонент, равное исходному числу измерений (т.е. 784). PCA захватит существенную информацию из исходного пространства данных и начнет генерировать главные компоненты. Как только они будут получены, мы определим, какое их количество необходимо для того, чтобы эффективно захватывать большую часть дисперсии/информации из оригинального набора признаков.

Подстроим и преобразуем наши тренировочные данные, сгенерировав главные компоненты.

```
X_train_PCA = pca.fit_transform(X_train)
X_train_PCA = pd.DataFrame(data=X_train_PCA, index=train_index)
```

Оценка PCA

Поскольку мы пока еще не уменьшили размерность данных, а лишь преобразовали их, доля дисперсии исходных данных, захваченная 784 главными компонентами, должна составить 100%.

```
# Доля дисперсии, захваченной 784 главными компонентами
print("Дисперсия, объясненная всеми 784 главными компонентами: ", \
      sum(pca.explained_variance_ratio_))

Дисперсия, объясненная всеми 784 главными компонентами:
0.9999999999999997
```

Следует, однако, отметить, что значимость 784 главных компонент варьируется в заметных пределах. Показатели значимости первых *X* главных компонент приведены ниже.

```
# Доля дисперсии, захваченной X главными компонентами
importanceOfPrincipalComponents = \
    pd.DataFrame(data=pca.explained_variance_ratio_)
importanceOfPrincipalComponents = importanceOfPrincipalComponents.T

print('Дисперсия, захваченная первыми 10 главными компонентами: ', \
      importanceOfPrincipalComponents.loc[:, \
      0:9].sum(axis=1).values)
print('Дисперсия, захваченная первыми 20 главными компонентами: ', \
      importanceOfPrincipalComponents.loc[:, \
      0:19].sum(axis=1).values)
print('Дисперсия, захваченная первыми 50 главными компонентами: ', \
      importanceOfPrincipalComponents.loc[:, \
      0:49].sum(axis=1).values)
print('Дисперсия, захваченная первыми 100 главными компонентами: ', \
      importanceOfPrincipalComponents.loc[:, \
      0:99].sum(axis=1).values)
print('Дисперсия, захваченная первыми 200 главными компонентами:', \
      importanceOfPrincipalComponents.loc[:, \
      0:199].sum(axis=1).values)
```

```
print('Дисперсия, захваченная первыми 300 главными компонентами:', \
      importanceOfPrincipalComponents.loc[:, \
      0:299].sum(axis=1).values)

Дисперсия, захваченная первыми 10 главными компонентами:
 [0.48876238]
Дисперсия, захваченная первыми 20 главными компонентами:
 [0.64398025]
Дисперсия, захваченная первыми 50 главными компонентами:
 [0.8248609]
Дисперсия, захваченная первыми 100 главными компонентами:
 [0.91465857]
Дисперсия, захваченная первыми 200 главными компонентами:
 [0.96650076]
Дисперсия, захваченная первыми 300 главными компонентами:
 [0.9862489]
```

Первые 10 компонент суммарно захватывают приблизительно 50% дисперсии, первые 100 компонент — 90%, а первые 300 компонент — 99%. Доля информации, захваченной оставшимися главными компонентами, пренебрежимо мала.

Мы можем отобразить значимость каждой главной компоненты в виде диаграммы, ранжируя компоненты от первой до последней. Чтобы упростить просмотр такой диаграммы, на рис. 3.2 представлены первые 10 компонент.

Теперь эффективность PCA должна стать для вас более очевидной. С помощью всего лишь первых 200 главных компонент (что гораздо меньше исходного числа измерений, равного 784), нам удалось захватить более 96% дисперсии/информации.

PCA позволяет ощутимо снизить размерность исходных данных, одновременно сохраняя бо́льшую часть существенной информации. На уменьшенном с помощью PCA подмножестве признаков другим алгоритмам машинного обучения, запускаемым далее в конвейерной цепочке, будет легче разделять точки данных в пространстве (для решения таких задач, как обнаружение аномалий и кластеризация), и для этого им потребуется меньше вычислительных ресурсов.

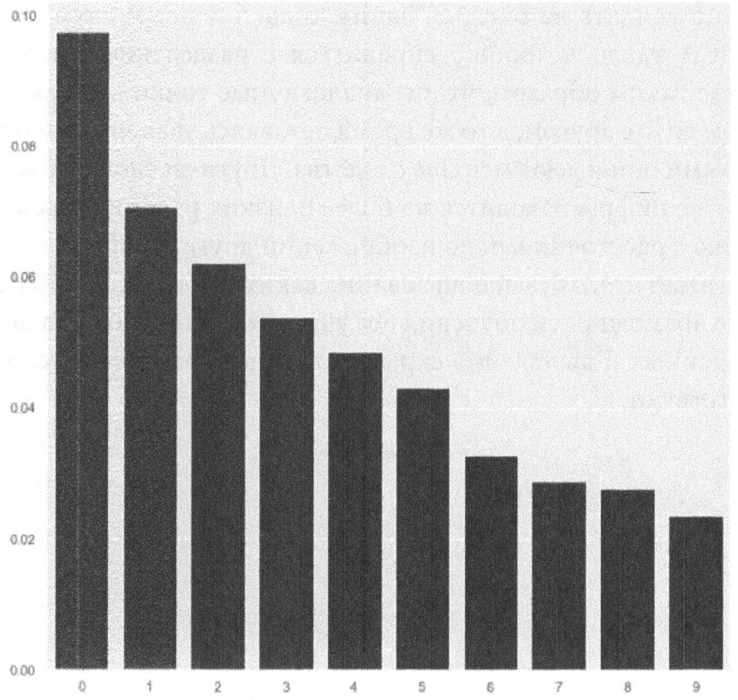


Рис. 3.2. *Значимость PCA-компонент*

Визуализация распределения точек данных в пространстве

Чтобы продемонстрировать, насколько эффективен метод PCA в отношении компактного захвата дисперсии, содержащейся в исходных данных, отобразим с помощью точечной диаграммы наблюдения в двух измерениях. В частности, мы представим на диаграмме первую и вторую главные компоненты и пометим наблюдения меткой истинности. Для этого создадим функцию `scatterPlot`, поскольку впоследствии нам потребуется создавать аналогичные визуализации для других алгоритмов снижения размерности.

```
def scatterPlot(xDF, yDF, algoName):
    tempDF = pd.DataFrame(data=xDF.loc[:, 0:1], index=xDF.index)
    tempDF = pd.concat((tempDF, yDF), axis=1, join="inner")
    tempDF.columns = ["Первый вектор", "Второй вектор", "Метка"]
    sns.lmplot(x="Первый вектор", y="Второй вектор", hue="Метка", \
               data=tempDF, fit_reg=False)
    ax = plt.gca()
    ax.set_title("Разделение наблюдений: " + algoName)

scatterPlot(X_train_PCA, y_train, "PCA")
```

Как можно увидеть на рис. 3.3, одним только первым двум главным компонентам PCA удалось хорошо справиться с разделением точек данных в пространстве таким образом, чтобы аналогичные точки компактно располагались рядом друг с другом, в то же время оставаясь удаленными от других точек, с которыми они имеют меньше сходства. Другими словами, изображения одной и той же цифры находятся на более близких расстояниях между собой по сравнению с расстояниями до изображений других цифр.

PCA достигает этого без использования каких-либо меток, что подтверждает огромные возможности обучения без учителя в отношении захвата базовой структуры данных и выявления скрытых закономерностей в условиях, когда метки отсутствуют.

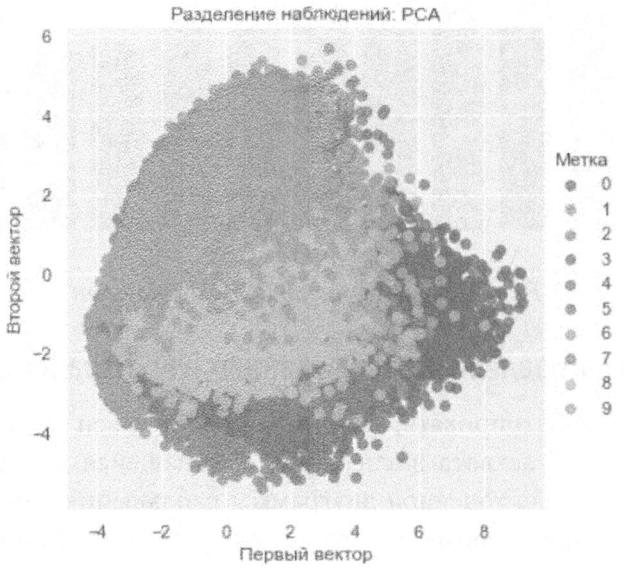


Рис. 3.3. *Разделение наблюдений с использованием PCA*

Аналогичная двумерная точечная диаграмма, но построенная с использованием двух наиболее важных признаков, выбранных из исходного полного набора (784 признака) исходя из результатов тренировки по методике обучения с учителем, демонстрирует плохое разделение точек (рис. 3.4).

Сравнение рис. 3.3 и 3.4 лишний раз свидетельствует о больших возможностях PCA в отношении изучения базовой структуры набора данных без привлечения каких-либо меток. Даже в случае использования только двух измерений мы получаем разумное разделение изображений в соответствии с представляемыми ими цифрами.

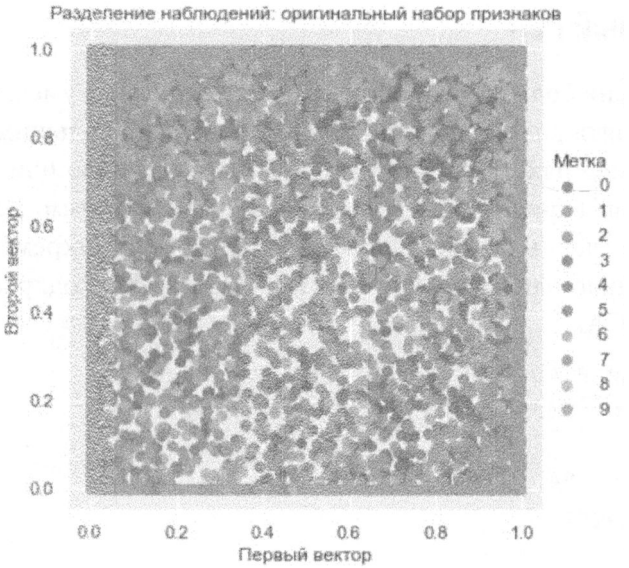


***Рис. 3.4.** Разделение наблюдений без использования PCA*

PCA не только облегчает выявление скрытых закономерностей в данных в процессе их разделения, но и помогает уменьшить размеры набора признаков, тем самым способствуя снижению затратности машинного обучения как по времени, так и по необходимым вычислительным ресурсам.

В примере с набором данных MNIST сокращение времени обучения будет умеренным ввиду малого размера самого набора: всего лишь 784 признака и 50 000 наблюдений. Но в случае наборов данных, насчитывающих миллионы признаков и миллиарды наблюдений, снижение размерности приведет к резкому сокращению времени обучения алгоритмов, используемых далее в конвейерной цепочке машинного обучения.

И последнее замечание. Обычно PCA отбрасывает часть информации, доступной в оригинальном наборе признаков, но делает это разумно, захватывая наиболее важные элементы и избавляясь от тех, которые представляют наименьшую ценность. Модель, обученная на наборе признаков, который был уменьшен с помощью PCA, может работать не настолько хорошо в смысле точности, как модель, обученная на полном наборе, но зато тренировка и предсказания будут выполняться значительно быстрее. Это один из наиболее важных компромиссов, на который приходится идти, принимая решение о том, стоит ли выполнять снижение размерности в создаваемом приложении машинного обучения.

Инкрементный PCA

В случае очень больших наборов данных, которые не умещаются в памяти, можно выполнять анализ инкрементно, работая с небольшими порциями — пакетами. Размер пакета можно устанавливать вручную или автоматически. Такой пакетный вариант PCA называется *инкрементным*. Как правило, результирующие главные компоненты стандартного и инкрементного вариантов PCA довольно близки между собой (рис. 3.5). Вот код, реализующий инкрементный PCA.

```
# Инкрементный PCA
from sklearn.decomposition import IncrementalPCA

n_components = 784
batch_size = None

incrementalPCA = IncrementalPCA(n_components=n_components, \
                                batch_size=batch_size)

X_train_incrementalPCA = incrementalPCA.fit_transform(X_train)
X_train_incrementalPCA = pd.DataFrame(data=X_train_incrementalPCA, \
    index=train_index)

X_validation_incrementalPCA = incrementalPCA.transform(X_validation)
X_validation_incrementalPCA = \
    pd.DataFrame(data=X_validation_incrementalPCA, \
    index=validation_index)

scatterPlot(X_train_incrementalPCA, y_train, "инкрементный PCA")
```

Разреженный PCA

Обычный алгоритм PCA осуществляет поиск линейных комбинаций среди всех входных переменных, уменьшая пространство исходных признаков в максимально возможной степени. Но в некоторых задачах машинного обучения может оказаться выгодной небольшая разреженность признаков. Разновидность PCA, сохраняющая определенный уровень разреженности (контролируемый гиперпараметром `alpha`), называется *разреженный PCA*. Алгоритм разреженного PCA осуществляет поиск линейных комбинаций лишь в некоторых входных переменных, сужая пространство исходных признаков, но не так компактно, как стандартный PCA.

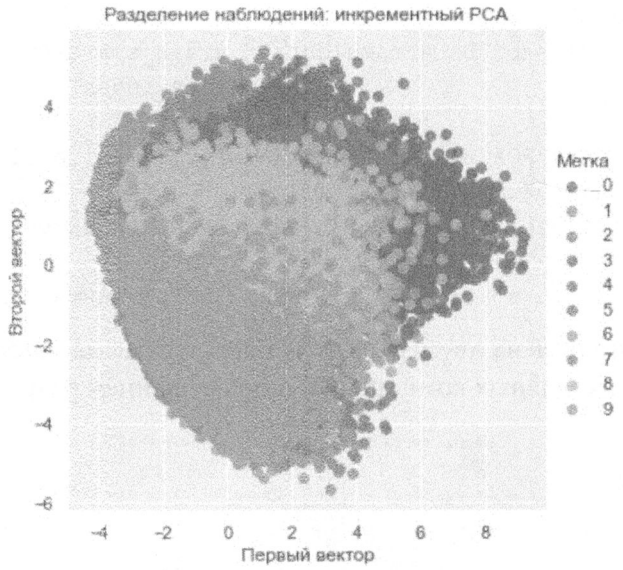


***Рис. 3.5.** Разделение наблюдений с использованием инкрементного PCA*

Поскольку этот алгоритм обучается немного медленнее, чем обычный PCA, мы будем использовать для тренировки лишь первые 10 000 примеров из полного тренировочного набора (включающего в общей сложности 50 000 примеров). Мы будем придерживаться этой практики обучения на поднаборах наблюдений в тех случаях, когда обучение алгоритма на полном наборе замедляется.

Сокращение процесса тренировки вполне подходит для нашей цели, ведь мы лишь хотим понять, как работают алгоритмы снижения размерности. Для получения оптимальных результатов лучше проводить обучение модели на полном тренировочном наборе.

```
# Разреженный PCA
from sklearn.decomposition import SparsePCA

n_components = 100
alpha = 0.0001
random_state = 2018
n_jobs = -1

sparsePCA = SparsePCA(n_components=n_components, alpha=alpha, \
                      random_state=random_state, n_jobs=n_jobs)

sparsePCA.fit(X_train.loc[:10000, :])
```

```
X_train_sparsePCA = sparsePCA.transform(X_train)
X_train_sparsePCA = pd.DataFrame(data=X_train_sparsePCA, \
                                 index=train_index)

X_validation_sparsePCA = sparsePCA.transform(X_validation)
X_validation_sparsePCA = \
    pd.DataFrame(data=X_validation_sparsePCA, index=validation_index)

scatterPlot(X_train_sparsePCA, y_train, "разреженный PCA")
```

На рис. 3.6 приведена двумерная точечная диаграмма, построенная на основе первых двух главных компонент с использованием разреженного PCA.

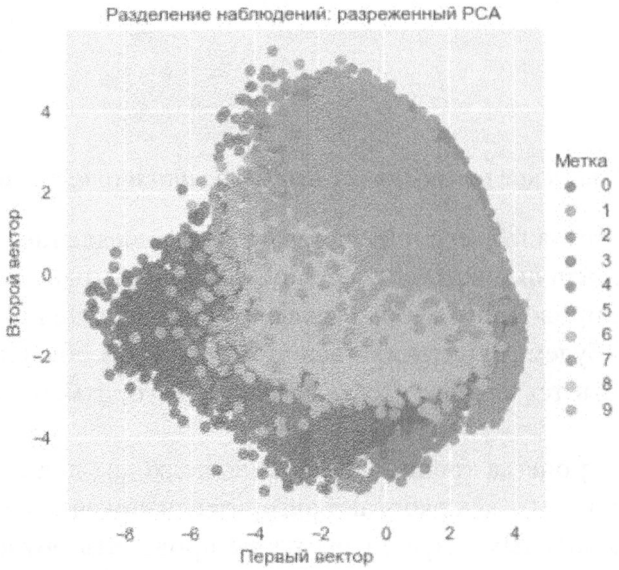


Рис. 3.6. *Разделение наблюдений с использованием разреженного PCA*

Заметьте, что эта диаграмма отличается от той, которая была получена с использованием стандартного PCA, чего и следовало ожидать. Стандартная и разреженная разновидности PCA по-разному генерируют главные компоненты, что и приводит к различиям в разделении точек данных.

Ядерный PCA

Стандартный, инкрементный и разреженный варианты PCA генерируют линейную проекцию исходных данных на пространство более низкой размерности, но существует и нелинейная разновидность — *ядерный PCA*,

применяющий функцию сходства к парам исходных точек данных для нелинейного снижения размерности.

Обучаясь этой функции сходства (подход, получивший название *ядерный метод*), ядерный PCA выявляет скрытое пространство признаков, в котором располагается большинство точек данных, и создает это скрытое пространство с намного меньшим количеством измерений по сравнению с оригинальным набором признаков. Такой метод особенно эффективен, когда исходный набор признаков не является линейно разделимым.

Используя алгоритм ядерного PCA, мы должны задать требуемое количество компонент, тип ядра и ядерный коэффициент (gamma). Наиболее популярным ядром является *радиально-базисная функция* (radial basis function — RBF), известная как *RBF-ядро*. Соответствующий код приведен ниже.

```
# Ядерный PCA
from sklearn.decomposition import KernelPCA

n_components = 100
kernel = 'rbf'
gamma = None
random_state = 2018
n_jobs = 1

kernelPCA = KernelPCA(n_components=n_components, kernel=kernel, \
    gamma=gamma, n_jobs=n_jobs, random_state=random_state)

kernelPCA.fit(X_train.loc[:10000, :])
X_train_kernelPCA = kernelPCA.transform(X_train)
X_train_kernelPCA = pd.DataFrame(data=X_train_kernelPCA, \
    index=train_index)

X_validation_kernelPCA = kernelPCA.transform(X_validation)
X_validation_kernelPCA = pd.DataFrame(data=X_validation_kernelPCA, \
    index=validation_index)

scatterPlot(X_train_kernelPCA, y_train, "ядерный PCA")
```

В случае нашего набора данных MNIST двумерная точечная диаграмма для ядерного PCA почти идентична диаграмме, полученной в стандартном варианте PCA (рис. 3.7). Обучение RBF-ядра не приводит к улучшению результатов при снижении размерности.

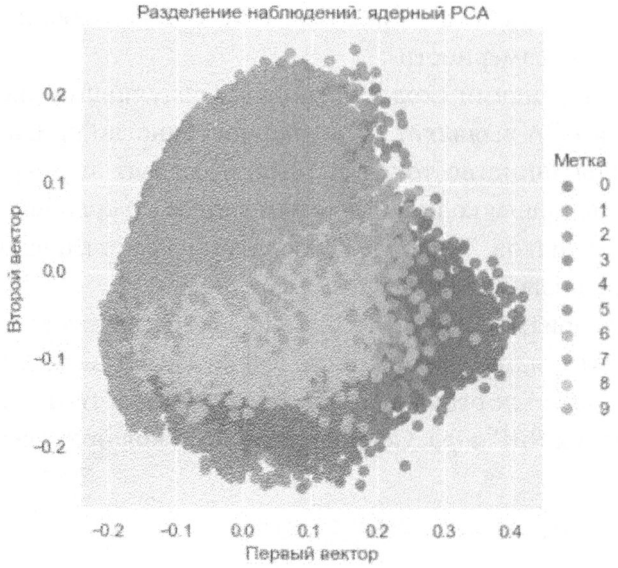


***Рис. 3.7.** Разделение наблюдений с использованием ядерного PCA*

Сингулярное разложение

Другой подход к обучению базовой структуре данных заключается в понижении ранга исходной матрицы признаков способом, допускающим возможность ее восстановления путем использования линейной комбинации некоторых из векторов, принадлежащих матрице меньшего ранга. Этот подход известен как *сингулярное разложение* (singular value decomposition — SVD).

Для генерирования матрицы меньшего ранга в методе SVD удерживаются векторы исходной матрицы, содержащие большую часть информации (т.е. те, которые имеют наибольшие сингулярные значения). Матрица пониженного ранга захватывает наиболее важные элементы в оригинальном пространстве признаков.

Все это очень напоминает метод PCA, в котором снижение размерности данных достигается за счет разложения ковариационной матрицы по собственным значениям. Фактически вычисления по методу PCA включают использование сингулярного разложения, но эта тема выходит за рамки книги.

Вот как работает метод SVD.

```
# Сингулярное разложение
from sklearn.decomposition import TruncatedSVD
```

```
n_components = 200
algorithm = 'randomized'
n_iter = 5
random_state = 2018

svd = TruncatedSVD(n_components=n_components, algorithm=algorithm, \
                   n_iter=n_iter, random_state=random_state)

X_train_svd = svd.fit_transform(X_train)
X_train_svd = pd.DataFrame(data=X_train_svd, index=train_index)

X_validation_svd = svd.transform(X_validation)
X_validation_svd = pd.DataFrame(data=X_validation_svd, \
                                index=validation_index)

scatterPlot(X_train_svd, y_train, "сингулярное разложение")
```

На рис. 3.8 показано разделение точек, достигнутое с использованием двух наиболее важных векторов SVD.

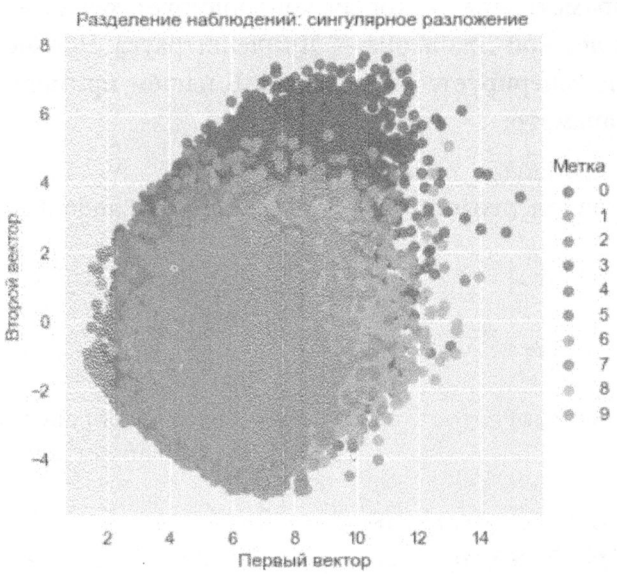


Рис. 3.8. *Разделение наблюдений с использованием сингулярного разложения*

Случайное проецирование

Еще одной разновидностью линейного снижения размерности является метод случайных проекций, в основу которого положена лемма о малом искажении, известная как *лемма Джонсона — Линденштрауса*. Согласно этой лемме точки многомерного пространства можно отобразить в пространство гораздо меньшей размерности таким образом, что расстояния между точками почти не изменятся. Другими словами, даже если мы перейдем от пространства высокой размерности к пространству низкой размерности, это не приведет к изменению структуры оригинального набора признаков.

Гауссовская случайная проекция

Существуют две разновидности случайного проецирования: стандартная (*гауссовская*) и разреженная.

В случае гауссовской случайной проекции мы можем либо указать требуемое количество компонент в уменьшенном пространстве признаков, либо задать гиперпараметр `eps`, который контролирует качество отображения в соответствии с леммой Джонсона — Линденштрауса. Чем меньше его значение, тем больше генерируется измерений. В нашем примере мы используем именно гиперпараметр.

```
# Гауссовская случайная проекция
from sklearn.random_projection import GaussianRandomProjection

n_components = 'auto'
eps = 0.5
random_state = 2018

GRP = GaussianRandomProjection(n_components=n_components, eps=eps, \
                               random_state=random_state)

X_train_GRP = GRP.fit_transform(X_train)
X_train_GRP = pd.DataFrame(data=X_train_GRP, index=train_index)

X_validation_GRP = GRP.transform(X_validation)
X_validation_GRP = pd.DataFrame(data=X_validation_GRP, \
                                index=validation_index)

scatterPlot(X_train_GRP, y_train, "гауссовская случайная проекция")
```

На рис. 3.9 показана двумерная точечная диаграмма, полученная с использованием гауссовской случайной проекции.

Рис. 3.9. *Разделение наблюдений с использованием гауссовской случайной проекции*

Несмотря на то что данный метод является одной из разновидностей линейного проецирования наподобие PCA, случайная проекция относится к совершенно другому семейству методов снижения размерности. Поэтому соответствующая точечная диаграмма заметно отличается от точечных диаграмм стандартного, инкрементного, разреженного и ядерного PCA.

Разреженная случайная проекция

Точно так же, как существует разреженная версия PCA, существует и разреженная версия случайного проецирования, известная как *разреженное случайное проецирование* (sparse random projection). Этот алгоритм сохраняет определенный уровень разреженности в преобразованном множестве признаков и в целом работает более эффективно, преобразуя оригинальные данные в редуцированное пространство гораздо быстрее, чем стандартная гауссовская проекция.

```
# Разреженная случайная проекция
from sklearn.random_projection import SparseRandomProjection
```

```
n_components = 'auto'
density = 'auto'
eps = 0.5
dense_output = False
random_state = 2018

SRP = SparseRandomProjection(n_components=n_components, \
    density=density, eps=eps, dense_output=dense_output, \
    random_state=random_state)

X_train_SRP = SRP.fit_transform(X_train)
X_train_SRP = pd.DataFrame(data=X_train_SRP, index=train_index)

X_validation_SRP = SRP.transform(X_validation)
X_validation_SRP = pd.DataFrame(data=X_validation_SRP, \
                                index=validation_index)

scatterPlot(X_train_SRP, y_train, "разреженная случайная проекция")
```

На рис. 3.10 показана двумерная точечная диаграмма, полученная с использованием разреженной случайной проекции.

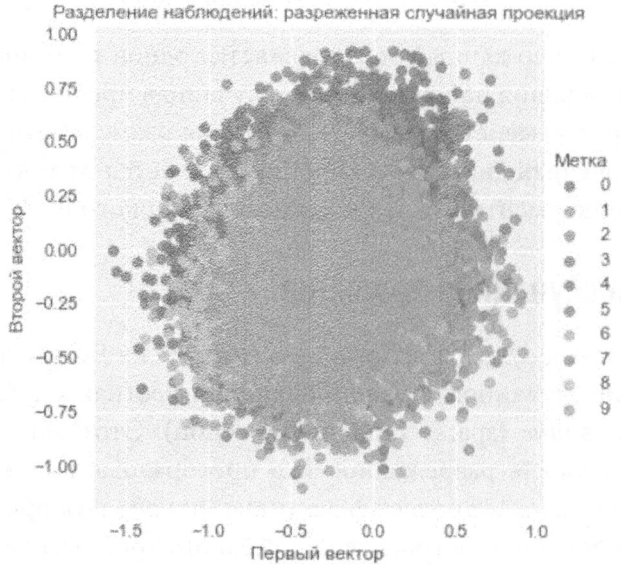


***Рис. 3.10.** Разделение наблюдений с использованием разреженной случайной проекции*

Метод Isomap

Вместо линейного проецирования данных из многомерного пространства в пространство низкой размерности можно использовать методы нелинейного снижения размерности. Эти методы называют *многократным обучением* (manifold learning).

Простейшая разновидность многократного обучения — *изометрическое отображение* (isometric mapping — сокр. *Isomap*). Подобно ядерному PCA метод Isomap обучается новому, низкоразмерному отображению оригинального набора признаков путем вычисления для всех точек попарных криволинейных расстояний, т.е. *геодезических кривых*, а не евклидовых линий. Другими словами, он обучается внутренней геометрии исходных данных на основании того, где именно расположена точка относительно своих соседей в пределах многообразия.

```
# Isomap

from sklearn.manifold import Isomap

n_neighbors = 5
n_components = 10
n_jobs = 4

isomap = Isomap(n_neighbors=n_neighbors, n_components=n_components, \
                n_jobs=n_jobs)

isomap.fit(X_train.loc[0:5000, :])
X_train_isomap = isomap.transform(X_train)
X_train_isomap = pd.DataFrame(data=X_train_isomap, index=train_index)

X_validation_isomap = isomap.transform(X_validation)
X_validation_isomap = pd.DataFrame(data=X_validation_isomap, \
                                   index=validation_index)

scatterPlot(X_train_isomap, y_train, "Isomap")
```

На рис. 3.11 показана двумерная точечная диаграмма, полученная с использованием метода Isomap.

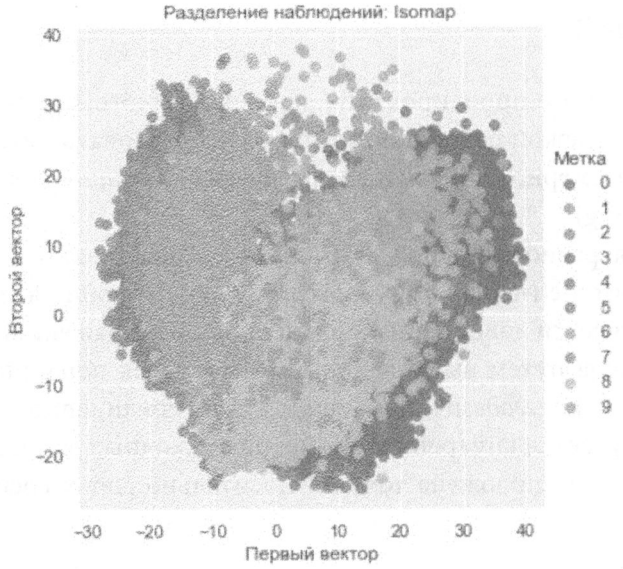


Рис. 3.11. *Разделение наблюдений с использованием метода Isomap*

Многомерное масштабирование

Многомерное масштабирование (multidimensional scaling — MDS) — это разновидность нелинейного снижения размерности, которая обучается сходству точек оригинального набора данных и моделирует это сходство в пространстве меньшей размерности.

```
# Многомерное масштабирование
from sklearn.manifold import MDS

n_components = 2
n_init = 12
max_iter = 1200
metric = True
n_jobs = 4
random_state = 2018

mds = MDS(n_components=n_components, n_init=n_init, \
          max_iter=max_iter, metric=metric, n_jobs=n_jobs, \
          random_state=random_state)

X_train_mds = mds.fit_transform(X_train.loc[0:1000, :])
```

```
X_train_mds = pd.DataFrame(data=X_train_mds, \
                           index=train_index[0:1001])

scatterPlot(X_train_mds, y_train, "многомерное масштабирование")
```

На рис. 3.12 показана двумерная точечная диаграмма, полученная с использованием метода MDS.

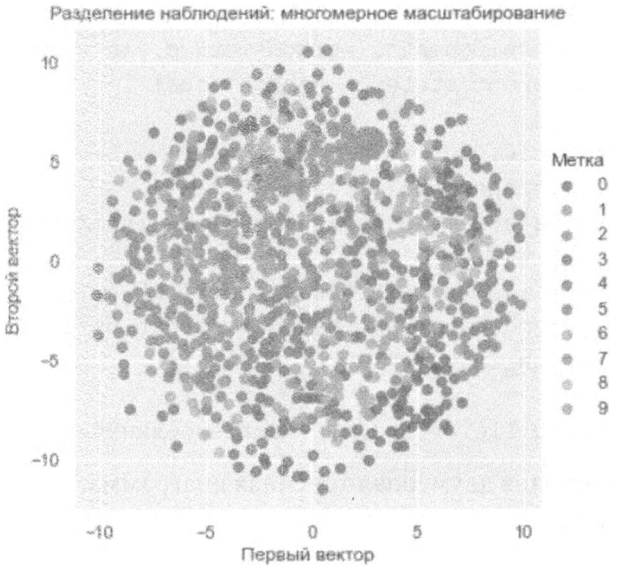


***Рис. 3.12.** Разделение наблюдений с использованием метода MDS*

Локально-линейное вложение

Другой популярный метод нелинейного снижения размерности — *локально-линейное вложение* (locally linear embedding — LLE). Идея метода заключается в сохранении расстояний между точками в пределах локального соседства при проецировании данных из исходного пространства признаков в пространство меньшей размерности. Метод LLE выявляет нелинейную структуру в исходном многомерном пространстве путем сегментирования данных на меньшие компоненты (локальные области соседних точек) и моделирует каждую компоненту как линейное вложение.

Для этого алгоритма мы задаем требуемое число компонент и количество точек, учитываемых в заданной области соседства.

```
# Локально-линейное вложение (LLE)
from sklearn.manifold import LocallyLinearEmbedding
```

```
n_neighbors = 10
n_components = 2
method = 'modified'
n_jobs = 4
random_state = 2018

lle = LocallyLinearEmbedding(n_neighbors=n_neighbors, \
    n_components=n_components, method=method, \
    random_state=random_state, n_jobs=n_jobs)

lle.fit(X_train.loc[0:5000, :])
X_train_lle = lle.transform(X_train)
X_train_lle = pd.DataFrame(data=X_train_lle, index=train_index)

X_validation_lle = lle.transform(X_validation)
X_validation_lle = pd.DataFrame(data=X_validation_lle, \
    index=validation_index)

scatterPlot(X_train_lle, y_train, "локально-линейное вложение")
```

На рис. 3.13 показана двумерная точечная диаграмма, полученная с использованием метода LLE.

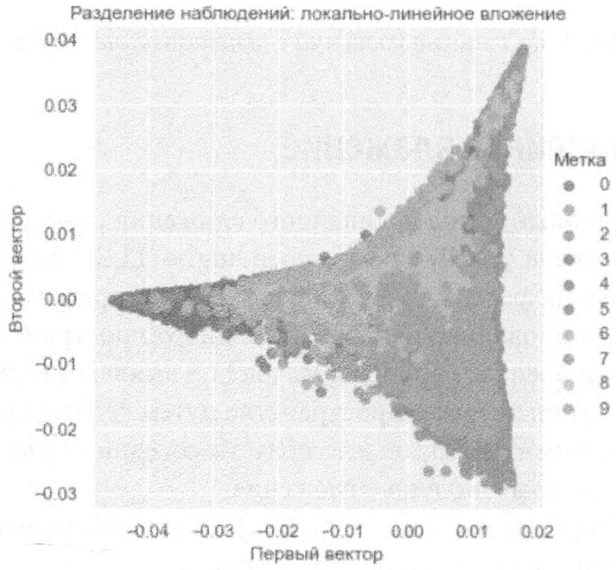


Рис. 3.13. *Разделение наблюдений с использованием метода LLE*

Стохастическое вложение соседей с t-распределением

Стохастическое вложение соседей с t-распределением (t-distributed stochastic neighbor embedding — t-SNE) — это метод нелинейного снижения размерности, применяемый для визуализации многомерных данных. В методе t-SNE каждая многомерная точка моделируется в 2- или 3-мерном пространстве таким образом, что сходные точки располагаются в нем по соседству друг с другом, а несходные точки — на удалении. Это достигается за счет учета двух распределений вероятности — по парам точек в многомерном пространстве и по парам точек в пространстве низкой размерности, — причем таким образом, чтобы сходные точки имели высокую вероятность, а несходные — низкую. В частности, метод t-SNE минимизирует расстояние Кульбака — Лейблера между двумя распределениями.

На практике, прежде чем применять метод t-SNE, лучше предварительно использовать какой-то другой метод снижения размерности (например, PCA, как сделано в данном случае). Это позволяет снизить уровень шумов в признаках, которые передаются методу t-SNE, что ускоряет работу алгоритма.

```
# t-SNE
from sklearn.manifold import TSNE

n_components = 2
learning_rate = 300
perplexity = 30
early_exaggeration = 12
init = 'random'
random_state = 2018

tSNE = TSNE(n_components=n_components, learning_rate=learning_rate, \
    perplexity=perplexity, early_exaggeration=early_exaggeration, \
    init=init, random_state=random_state)

X_train_tSNE = tSNE.fit_transform(X_train_PCA.loc[:5000, :9])
X_train_tSNE = pd.DataFrame(data=X_train_tSNE, \
                            index=train_index[:5001])

scatterPlot(X_train_tSNE, y_train, "t-SNE")
```

В методе t-SNE используется невыпуклая функция потерь, а это означает, что разные способы инициализации алгоритма будут приводить к разным результатам. Стабильного решения не существует.

Двумерная точечная диаграмма, полученная с использованием метода t-SNE, приведена на рис. 3.14.

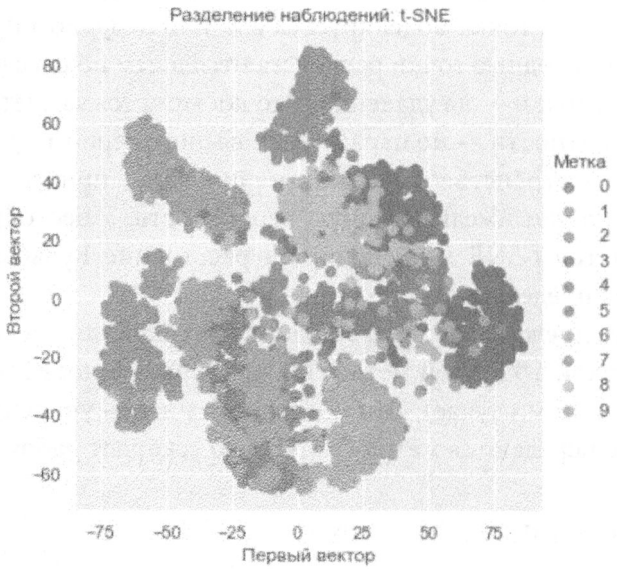


***Рис. 3.14.** Разделение наблюдений с использованием метода t-SNE*

Другие методы снижения размерности

Мы рассмотрели как линейные, так и нелинейные варианты снижения размерности. Обсудим теперь методы, которые не зависят от геометрии или пространственной метрики.

Словарное обучение

Один из таких методов — *словарное обучение* (dictionary learning), основанное на обучении разреженному представлению исходных данных. Результирующая матрица называется *словарем*, а результирующие векторы в словаре — *атомы*. Последние представляют собой простые бинарные векторы, компонентами которых служат нули и единицы. Каждый экземпляр исходных данных может быть реконструирован в виде взвешенной суммы атомов.

Если предположить, что исходный набор данных содержит d признаков, а словарь — n атомов, то возможны два варианта: словарь является либо неполным ($n < d$), либо переполненным ($n > d$). Неполный словарь обеспечивает снижение размерности, представляя исходные данные меньшим количеством векторов, на чем мы и сфокусируем свое внимание[1].

Существует *мини-пакетная* версия словарного обучения, которую мы применим к нашему набору данных рукописных цифр. Как и в случае других методов снижения размерности, мы зададим количество компонент. Кроме того, необходимо задать размер пакета и количество итераций обучения.

Поскольку мы хотим визуализировать изображения с помощью двумерной точечной диаграммы, нам для обучения необходим очень плотный словарь, но на практике следовало бы использовать более разреженную версию.

```
# Мини-пакетное словарное обучение
from sklearn.decomposition import MiniBatchDictionaryLearning

n_components = 50
alpha = 1
batch_size = 200
n_iter = 25
random_state = 2018

miniBatchDictLearning = \
    MiniBatchDictionaryLearning(n_components=n_components, \
        alpha=alpha, batch_size=batch_size, n_iter=n_iter, \
        random_state=random_state)

miniBatchDictLearning.fit(X_train.loc[:, :10000])
X_train_miniBatchDictLearning = \
    miniBatchDictLearning.fit_transform(X_train)
X_train_miniBatchDictLearning = \
    pd.DataFrame(data=X_train_miniBatchDictLearning, \
                 index=train_index)

X_validation_miniBatchDictLearning = \
    miniBatchDictLearning.transform(X_validation)
X_validation_miniBatchDictLearning = \
    pd.DataFrame(data=X_validation_miniBatchDictLearning, \
                 index=validation_index)
```

[1] Переполненный словарь служит другим целям и находит применение в таких областях, как сжатие изображений.

```
scatterPlot(X_train_miniBatchDictLearning, y_train, \
            "мини-пакетное словарное обучение")
```

На рис. 3.15 показана двумерная точечная диаграмма, полученная с использованием словарного обучения.

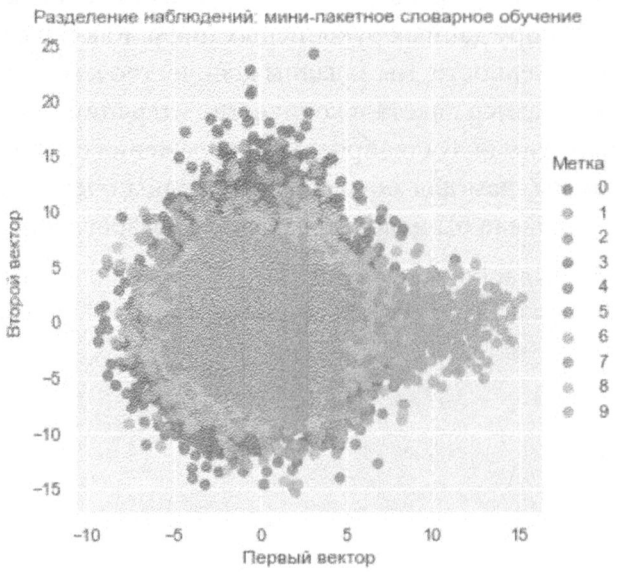


Рис. 3.15. *Разделение наблюдений с использованием словарного обучения*

Анализ независимых компонент

Одна из проблем, с которой часто приходится сталкиваться при работе с неразмеченными данными, заключается в том, что в имеющиеся признаки внедрены многочисленные независимые сигналы. Используя *анализ независимых компонент* (independent component analysis — ICA), можно выделять эти примеси в отдельные компоненты. По завершении такого разделения мы сможем реконструировать любой из оригинальных признаков, суммируя определенные комбинации индивидуальных компонент, которые генерируются. Метод ICA широко применяется при обработке сигналов (например, для идентификации голосов отдельных людей в аудиоклипе, записанном в шумном кафетерии).

Вот как работает метод ICA.

```
# Анализ независимых компонент
from sklearn.decomposition import FastICA
```

```
n_components = 25
algorithm = 'parallel'
whiten = True
max_iter = 100
random_state = 2018

fastICA = FastICA(n_components=n_components, algorithm=algorithm, \
    whiten=whiten, max_iter=max_iter, random_state=random_state)

X_train_fastICA = fastICA.fit_transform(X_train)
X_train_fastICA = pd.DataFrame(data=X_train_fastICA, \
                               index=train_index)

X_validation_fastICA = fastICA.transform(X_validation)
X_validation_fastICA = pd.DataFrame(data=X_validation_fastICA, \
                                    index=validation_index)

scatterPlot(X_train_fastICA, y_train, "анализ независимых компонент")
```

На рис. 3.16 показана двумерная точечная диаграмма, полученная с использованием метода ICA.

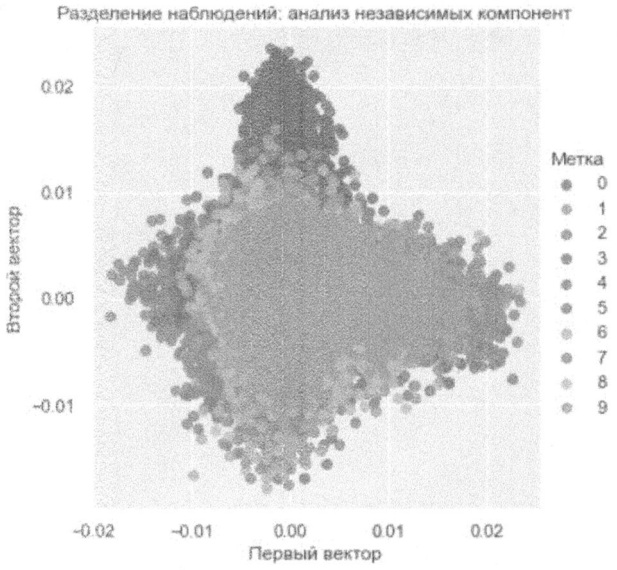


***Рис. 3.16.** Разделение наблюдений с использованием анализа независимых компонент*

Резюме

В этой главе мы ввели и исследовали ряд алгоритмов снижения размерности, начав с линейных методов, таких как PCA и метод случайных проекций. Далее мы обсудили нелинейные методы, такие как Isomap, многомерное масштабирование, LLE и t-SNE. Также были рассмотрены методы, не основанные на пространственных метриках, в частности, словарное обучение и ICA.

Снижение размерности позволяет переносить наиболее существенную информацию в пространство с меньшим количеством измерений путем обучения базовой структуре данных, причем без использования каких-либо меток. Применяя эти алгоритмы к набору рукописных цифр MNIST, мы смогли добиться разумного разделения изображений на основании представляемых ими цифр, используя всего лишь два измерения.

Это позволило вам убедиться в том, насколько огромны возможности подходов, основанных на снижении размерности данных.

В главе 4 мы построим приложение на основе обучения без учителя, используя алгоритмы снижения размерности. В частности, мы вернемся к задаче обнаружения поддельных операций с банковскими картами (см. гл. 2) и попытаемся отделить поддельные транзакции от нормальных, не используя метки.

ГЛАВА 4

Обнаружение аномалий

В главе 3 мы рассмотрели основные алгоритмы снижения размерности и исследовали их способность захватывать наиболее существенную часть информации из базы данных рукописных цифр MNIST, значительно снижая размерность по сравнению с первоначальными 784 измерениями. Даже если оставить всего два измерения, алгоритмы обеспечивают разумное разделение цифр без использования меток. Это служит свидетельством эффективности алгоритмов обучения без учителя — они могут обучаться базовой структуре данных и помогают выявлять скрытые закономерности в отсутствие меток.

В этой главе мы будем создавать приложение машинного обучения, используя методы снижения размерности, рассмотренные в предыдущей главе. Мы вернемся к задаче из главы 2 и реализуем систему, позволяющую обнаруживать попытки мошенничества с банковскими картами, только на этот раз без использования меток.

В реальности фальсификация часто остается незамеченной, и лишь обнаруженные факты мошенничества позволяют добавить метки в набор данных. Более того, схемы мошенничества со временем меняются, поэтому системы на основе обучения с учителем, подобные той, которую мы создали в главе 2, теряют эффективность. Они позволяют выявлять уже встречавшиеся типы подделок, но не способны адаптироваться к новым схемам.

В силу указанных причин (отсутствие достаточных меток и необходимость как можно быстрее адаптироваться к новым видам мошеннических операций) все большую популярность приобретают системы выявления мошенничества, основанные на обучении без учителя.

Обнаружение попыток мошенничества с банковскими картами

Вернемся к задаче выявления незаконных операций с банковскими картами, которая рассматривалась в главе 2.

Подготовка данных

Подобно тому, как это делалось в главе 2, загрузим набор данных с информацией об операциях с банковскими картами, сгенерируем матрицу признаков и массив меток и разобьем метки на тренировочный и тестовый наборы. Метки будут использоваться не для обнаружения аномалий, а для того, чтобы облегчить оценку возможностей создаваемой системы.

Как вы помните, всего имеется 284 807 транзакций с банковскими картами, из которых 492 поддельные, и им присвоены положительные (подтверждающие их мошеннический статус) метки, равные единице. Остальные (нормальные) транзакции снабжены отрицательными (указывающими на отсутствие мошенничества) метками, равными 0.

Мы располагаем 30 признаками, пригодными для обнаружения аномалий: время транзакции, сумма транзакции, а также 28 главных компонент. Разобьем набор данных на тренировочный (содержащий 190 820 транзакций и 330 примеров подделки) и тестовый (содержащий оставшиеся 93 987 транзакций и 162 примера подделки).

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'credit_card_data', \
                         'credit_card.csv'])
data = pd.read_csv(current_path + file)

dataX = data.copy().drop(['Class'], axis=1)
dataY = data['Class'].copy()

featuresToScale = dataX.columns
sX = pp.StandardScaler(copy=True)
dataX.loc[:,featuresToScale] = \
    sX.fit_transform(dataX[featuresToScale])

X_train, X_test, y_train, y_test = train_test_split(dataX, dataY, \
    test_size=0.33, random_state=2018, stratify=dataY)
```

Определение функции для оценки аномалий

Далее нам нужно написать функцию, определяющую, насколько необычна каждая транзакция. Если исходить из того, что попытки мошенничества встречаются редко и в чем-то отличаются от большинства транзакций, являющихся

законными, то можно предположить, что чем выше степень анормальности транзакции, тем выше вероятность того, что она является мошеннической.

Как обсуждалось в предыдущей главе, алгоритмы снижения размерности уменьшают размерность данных, одновременно пытаясь минимизировать ошибку их реконструкции. Другими словами, эти алгоритмы пытаются захватить наиболее существенные признаки так, чтобы с их помощью можно было обеспечить как можно более полное восстановление исходного набора признаков по редуцированному набору. Однако при переходе к пространству более низкой размерности алгоритмы не в состоянии захватить всю информацию об исходном наборе. Следовательно, обратная реконструкция всегда будет выполняться с определенной погрешностью.

В контексте нашего набора данных наибольшая ошибка реконструкции будет генерироваться на тех транзакциях, которые сложнее всего смоделировать, т.е. на транзакциях, которые встречаются реже всего и являются наиболее аномальными. Поскольку мошеннические транзакции возникают нечасто и предположительно отличаются от нормальных транзакций, именно они будут давать наибольшую ошибку реконструкции. Поэтому мы определим оценку аномалии как ошибку реконструкции. Для каждой транзакции ошибка реконструкции представляет собой сумму квадратов разниц между исходной матрицей признаков и реконструированной матрицей, полученной с использованием алгоритма снижения размерности. Мы будем масштабировать эту сумму квадратов в интервале от минимальной до максимальной суммы квадратов, вычисленной для всего набора данных, чтобы все значения ошибки реконструкции лежали в диапазоне от нуля до единицы.

Значения ошибки реконструкции для транзакций с наибольшей суммой квадратов разниц будут близки к единице, тогда как в случае транзакций с наименьшей суммой квадратов разниц эти значения будут близки к нулю.

Все это уже должно быть вам знакомым. Как и в случае созданного в главе 2 решения на основе обучения с учителем, алгоритм снижения размерности в конечном счете будет назначать каждой транзакции оценку аномалии в диапазоне от нуля до единицы. Нуль соответствует обычным транзакциям, а единица — аномальным (которые, вероятнее всего, являются мошенническими).

Вот как выглядит функция оценки.

```
def anomalyScores(originalDF, reducedDF):
    loss = np.sum((np.array(originalDF)-np.array(reducedDF))**2, \
                  axis=1)
    loss = pd.Series(data=loss, index=originalDF.index)
    loss = (loss-np.min(loss))/(np.max(loss)-np.min(loss))
    return loss
```

Определение метрик оценки

Несмотря на то что при обнаружении попыток мошенничества метки не задействуются, мы будем применять их для оценки разрабатываемых решений. Метки помогут нам понять, насколько хорошо приложения справляются с выявлением известных схем мошенничества.

Как и в главе 2, мы будем использовать кривую "точность — полнота", среднюю точность и показатель auROC в качестве метрик оценки.

Вот функция, которая будет выводить результаты в графическом виде.

```
def plotResults(trueLabels, anomalyScores, returnPreds = False):
    preds = pd.concat([trueLabels, anomalyScores], axis=1)
    preds.columns = ['trueLabel', 'anomalyScore']
    precision, recall, thresholds = \
        precision_recall_curve(preds['trueLabel'], \
                               preds['anomalyScore'])
    average_precision = \
        average_precision_score(preds['trueLabel'], \
                                preds['anomalyScore'])
    plt.step(recall, precision, color='k', alpha=0.7, where='post')
    plt.fill_between(recall, precision, step='post', alpha=0.3, \
                     color='k')

    plt.xlabel('Полнота')
    plt.ylabel('Точность')
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])

    plt.title('Кривая "точность - полнота": средняя точность = \
              {0:0.2f}'.format(average_precision))

    fpr, tpr, thresholds = roc_curve(preds['trueLabel'], \
                                     preds['anomalyScore'])
    areaUnderROC = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='r', lw=2, label='ROC-кривая')
    plt.plot([0, 1], [0, 1], color='k', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Доля ложноположительных исходов')
    plt.ylabel('Доля истинноположительных исходов')
```

```
plt.title('    Рабочая характеристика приемника: \n \
      площадь под кривой = {0:0.2f}'.format(areaUnderROC))
plt.legend(loc="lower right")
plt.show()

if returnPreds==True:
    return preds
```

Метки подделок и оценочные метрики помогут нам понять, насколько хорошо система на основе обучения без учителя справляется с распознаванием известных схем мошенничества, т.е. типов мошенничества, которые были установлены ранее и для которых имеются метки.

Однако мы не сможем оценить эффективность системы в отношении распознавания неизвестных схем мошенничества. Другими словами, в наборе данных могут присутствовать образцы, некорректно помеченные как поддельные в силу того, что финансовая компания ранее не сталкивалась с ними.

Сразу же стоит отметить, что оценивать системы на основе обучения без учителя намного труднее, чем системы на основе обучения с учителем. О качестве систем на основе обучения без учителя часто судят по их способности выявлять известные схемы мошенничества. Но эта оценка неполная. Было бы лучше, если бы метрика позволяла оценить способность системы идентифицировать неизвестные схемы мошенничества, как в ранее известных, так и в будущих данных.

Поскольку мы не можем вновь обратиться к финансовой компании с просьбой оценить ранее неизвестные образцы мошенничества, идентифицированные нами, мы вынуждены оценивать решения на основе обучения без учителя исходя исключительно из того, насколько хорошо они обнаруживают известные образцы подделок. Оценивая результаты, очень важно не забывать об этом ограничении.

Определение функции для построения графика

Чтобы отобразить разделение точек, достигнутое с помощью алгоритма снижения размерности для двух первых измерений, мы воспользуемся функцией построения точечных диаграмм из главы 3.

```
def scatterPlot(xDF, yDF, algoName):
    tempDF = pd.DataFrame(data=xDF.loc[:, 0:1], index=xDF.index)
    tempDF = pd.concat((tempDF, yDF), axis=1, join="inner")
    tempDF.columns = ["Первый вектор", "Второй вектор", "Метка"]
    sns.lmplot(x="Первый вектор", y="Второй вектор", hue="Метка", \
               data=tempDF, fit_reg=False)
    ax = plt.gca()
    ax.set_title("Разделение наблюдений: " + algoName)
```

Обнаружение аномалий с помощью стандартного метода PCA

В главе 3 было продемонстрировано, как метод PCA захватывает бо́льшую часть информации, содержащейся в наборе данных MNIST, используя всего лишь несколько компонент, количество которых гораздо меньше количества первоначальных измерений. В действительности всего лишь двух измерений хватило для того, чтобы визуально разделить изображения на отчетливо различные группы, исходя из представляемых ими цифр.

Теперь, отталкиваясь от этой концепции, мы используем метод PCA для изучения базовой структуры набора данных, описывающего транзакции с банковскими картами. После этого мы используем обученную модель для реконструкции транзакций и рассчитаем, насколько реконструированные транзакции отличаются от первоначальных. Те транзакции, с восстановлением которых метод PCA справился хуже всего, являются наиболее аномальными (и, вероятнее всего, им соответствуют мошеннические операции).

Вспомните, что имеющиеся в нашем распоряжении признаки, соответствующие транзакциям из набора данных, уже являются выходом метода PCA и были предоставлены нам финансовой компанией. Однако не будет ничего необычного и в том, что мы выполним анализ для обнаружения аномалий, используя набор данных с уже уменьшенной размерностью. Мы просто будем обрабатывать исходные главные компоненты так, как если бы они были предоставлены нам в качестве исходных признаков.

Впредь мы будем трактовать исходные главные компоненты именно как исходные признаки. Любое упоминание главных компонент будет относиться к главным компонентам, полученным в результате анализа PCA, а не к оригинальным признакам, которые были нам предоставлены.

Прежде всего, попытаемся понять, почему PCA — и снижение размерности в целом — помогает нам обнаруживать аномалии. Процесс снижения размерности, в соответствии с тем, как мы его определили, основан на оценке ошибки реконструкции данных. Мы хотим, чтобы ошибка реконструкции для транзакций редкого типа — тех, которые, вероятнее всего, являются поддельными, — была как можно большей, а для остальных транзакций — как можно меньшей.

В случае PCA ошибка реконструкции будет в значительной степени зависеть от количества главных компонент, которые мы удерживаем и используем для реконструкции оригинальных транзакций. Чем больше главных компонент мы оставим, тем успешнее метод PCA обучится базовой структуре исходных данных.

Однако при этом необходимо обеспечить достижение определенного баланса. Если мы оставим слишком много главных компонент, то алгоритму PCA удастся настолько хорошо реконструировать исходные транзакции, что ошибка реконструкции будет минимальной для всех транзакций. Если же количество удержанных главных компонент будет слишком маленьким, то метод PCA не сможет достаточно хорошо реконструировать любую из исходных транзакций, даже если она нормальная, а не поддельная.

Ниже мы постараемся определить, каким должно быть оптимальное количество главных компонент, чтобы обеспечить построение эффективной системы обнаружения мошеннических транзакций.

Количество PCA-компонент совпадает с числом исходных размерностей

Для начала зададимся следующим вопросом: если мы используем PCA для генерирования главных компонент в количестве, совпадающем с исходным числом признаков, то сможет ли система обнаруживать аномалии?

Если хорошенько над этим подумать, то ответ должен быть очевидным. Вспомните приведенный в предыдущей главе пример использования метода PCA для работы с базой данных MNIST.

Когда количество главных компонент совпадает с исходным числом признаков, PCA захватывает почти 100% дисперсии/информации в процессе генерирования главных компонент. Поэтому, когда PCA реконструирует транзакции из главных компонент, ошибка реконструкции будет малой для всех транзакций, как поддельных, так и всех остальных. Мы не сможем проводить различие между транзакциями редко встречающегося типа и

обычными транзакциями. Иными словами, обнаружение аномалий будет неэффективным.

Чтобы пояснить вышесказанное, применим PCA для генерирования главных компонент в том же количестве, что и оригинальные признаки (30 для нашего набора транзакций с банковскими картами). Это осуществляется с помощью функции `fit_transform` из библиотеки Scikit-learn.

Для восстановления оригинальных транзакций из генерируемых главных компонент мы используем функцию `inverse_transform` из библиотеки Scikit-learn.

```
# 30 главных компонент
from sklearn.decomposition import PCA

n_components = 30
whiten = False
random_state = 2018

pca = PCA(n_components=n_components, whiten=whiten, \
          random_state=random_state)

X_train_PCA = pca.fit_transform(X_train)
X_train_PCA = pd.DataFrame(data=X_train_PCA, index=X_train.index)

X_train_PCA_inverse = pca.inverse_transform(X_train_PCA)
X_train_PCA_inverse = pd.DataFrame(data=X_train_PCA_inverse, \
                                   index=X_train.index)

scatterPlot(X_train_PCA, y_train, "PCA")
```

На рис. 4.1 приведена диаграмма разделения транзакций с использованием первых двух главных компонент метода PCA.

Рассчитаем кривую "точность — полнота" и auROC-кривую.

```
anomalyScoresPCA = anomalyScores(X_train, X_train_PCA_inverse)
preds = plotResults(y_train, anomalyScoresPCA, True)
```

Средняя точность, равная `0.07`, указывает на то, что данное решение не является удачным (рис. 4.2). Оно захватывает лишь небольшую часть поддельных транзакций.

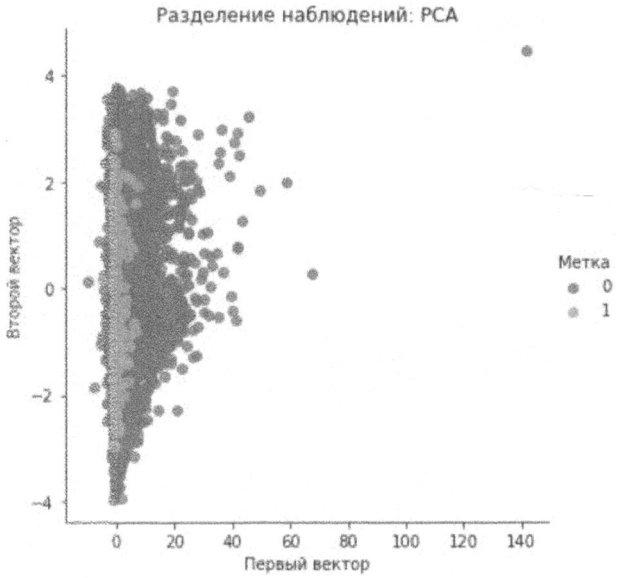


***Рис. 4.1.** Разделение наблюдений с использованием обычного метода PCA и 30 главных компонент*

Поиск оптимального количества главных компонент

А теперь давайте проведем ряд экспериментов, уменьшая количество генерируемых главных компонент PCA и вычисляя результаты, касающиеся обнаружения фактов мошенничества. Нам нужно, чтобы решение, обеспечивающее обнаружение подделок с помощью метода PCA, приводило к достаточно большой ошибке в тех редких случаях, когда удается разумно отделить мошеннические транзакции от законных. Однако ошибка не может быть настолько малой или настолько большой для всех транзакций, чтобы редкие и обычные транзакции были практически неразличимыми.

Выполнив ряд экспериментов с использованием кода, доступного на сайте GitHub (`http://bit.ly/2Gd4v7e`), мы найдем, что оптимальное количество главных компонент для данного набора равно 27.

На рис. 4.3 приведена диаграмма разделения транзакций с использованием первых двух главных компонент PCA.

На рис. 4.4 приведены кривая "точность — полнота", средняя точность и auROC-кривая.

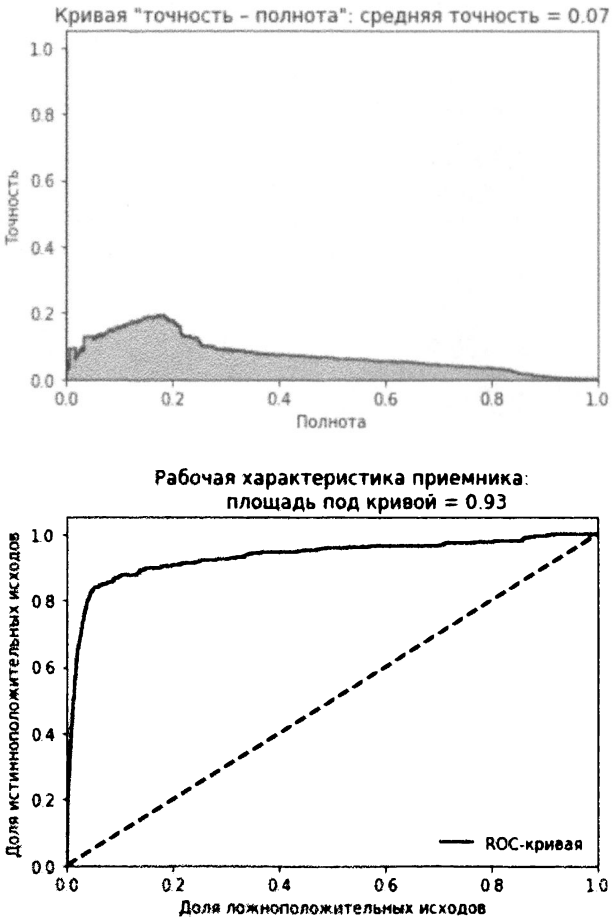


Рис. 4.2. *Результаты, полученные с использованием 30 главных компонент*

Как видите, мы смогли выявить 80% подделок с 75%-ной точностью. Это впечатляющий результат, если учесть, что мы не использовали никаких меток. Чтобы еще лучше это прочувствовать, вспомните, что тренировочный набор содержит 190 820 транзакций, из которых всего лишь 330 мошеннические.

Используя метод PCA, мы вычислили ошибку реконструкции для каждой из этих 190 820 транзакций. Если мы отсортируем эти транзакции по убыванию наибольшей ошибки реконструкции (также известной как оценка аномальности) и извлечем из списка первые 350 транзакций, то увидим, что 264 из них являются мошенническими.

Это соответствует точности 75%. Кроме того, 264 транзакции, извлеченные из отобранных нами 350, представляют 80% от общего количества подделок в

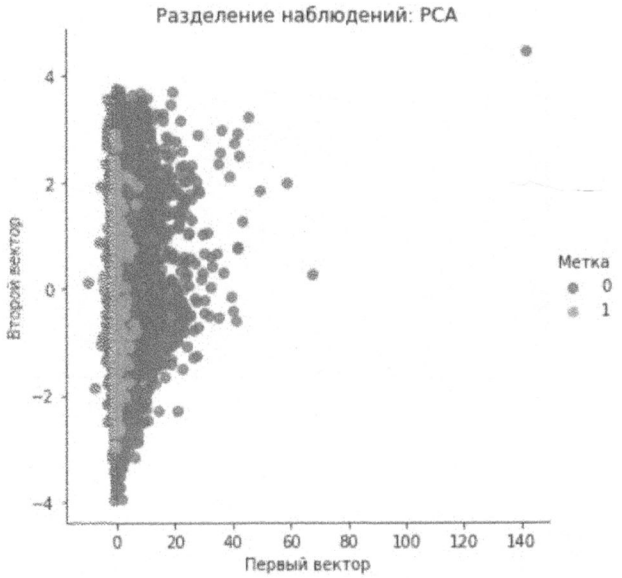


***Рис. 4.3.** Разделение наблюдений с использованием обычного метода PCA и 27 главных компонент*

тренировочном наборе (264 из 330). И не забывайте, что мы не использовали никаких меток. Это истинный пример обучения без учителя.

Ниже приведен код, обеспечивающий вывод соответствующей информации.

```
preds.sort_values(by="anomalyScore", ascending=False, inplace=True)
cutoff = 350
predsTop = preds[:cutoff]
print("Точность:", \
      np.round(predsTop.anomalyScore[predsTop.trueLabel == \
      1].count()/cutoff, 2))
print("Полнота:", \
      np.round(predsTop.anomalyScore[predsTop.trueLabel == \
      1].count()/y_train.sum(), 2))
print("Выявлено подделок из 330 случаев:", predsTop.trueLabel.sum())
```

Результаты подытожены ниже:

```
Точность: 0.75
Полнота: 0.8
Выявлено подделок из 330 случаев: 264
```

Несмотря на то что данное решение уже само по себе неплохое, мы попытаемся разработать систему обнаружения мошеннических транзакций, основанную на использовании других методов снижения размерности.

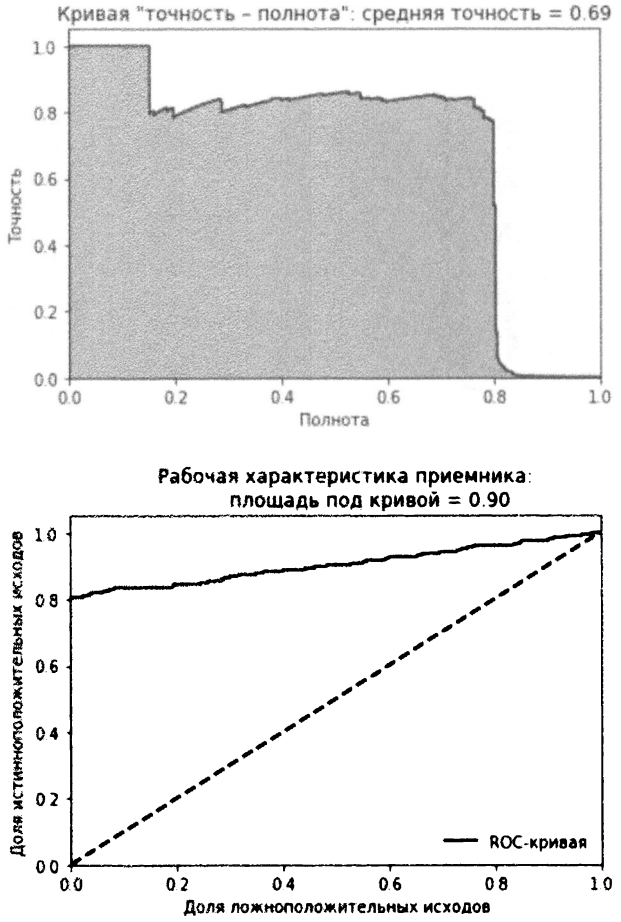


Рис. 4.4. *Результаты, полученные с использованием стандартного метода PCA и 27 главных компонент*

Обнаружение аномалий с помощью разреженного метода PCA

Для начала попробуем применить разреженный метод PCA, который напоминает стандартный метод PCA, но обеспечивает получение разреженного представления главных компонент.

Нам нужно указать требуемое количество главных компонент, а также задать гиперпараметр `alpha`, управляющий степенью разреженности. В процессе поиска оптимального решения мы будем экспериментировать с различными значениями этих параметров.

Следует отметить, что для стандартного метода PCA в библиотеке Scikit-learn предусмотрены функции `fit_transform`, генерирующая главные компоненты, и `inverse_transform`, реконструнрующая оригинальные измерения по главным компонентам. Используя эти две функции, мы смогли рассчитать ошибку реконструкции между оригинальным набором признаков и реконструированным набором, полученным с помощью PCA.

К сожалению, библиотека Scikit-learn не содержит функцию `inverse_transform` для разреженного метода PCA. Поэтому мы сами должны написать код для реконструирования оригинальных измерений после применения разреженного метода PCA.

Начнем с генерирования разреженной матрицы PCA с 27 главными компонентами и гиперпараметром `alpha`, равным `0.0001`.

```
# Разреженный PCA
from sklearn.decomposition import SparsePCA

n_components = 27
alpha = 0.0001
random_state = 2018
n_jobs = -1

sparsePCA = SparsePCA(n_components=n_components, \
    alpha=alpha, random_state=random_state, n_jobs=n_jobs)

sparsePCA.fit(X_train.loc[:, :])
X_train_sparsePCA = sparsePCA.transform(X_train)
X_train_sparsePCA = pd.DataFrame(data=X_train_sparsePCA, \
                                 index=X_train.index)

scatterPlot(X_train_sparsePCA, y_train, "разреженный PCA")
```

Точечная диаграмма для разреженного метода PCA приведена на рис. 4.5.

Далее мы сгенерируем оригинальные измерения путем простого умножения матрицы разреженного PCA (насчитывающей 190 820 образцов и 27 измерений) на компоненты разреженного PCA (матрица 27 × 30), используя библиотеку Scikit-learn. В результате мы получим матрицу оригинального размера (190 820 × 30). Кроме того, мы должны прибавить средние значения каждого оригинального признака к новой матрице.

Получив эту обратную матрицу, мы можем вычислить ошибки реконструкции (оценки аномальности), как это делалось в случае стандартного метода PCA.

```
X_train_sparsePCA_inverse = \
    np.array(X_train_sparsePCA).dot(sparsePCA.components_) + \
    np.array(X_train.mean(axis=0))
X_train_sparsePCA_inverse = \
    pd.DataFrame(data=X_train_sparsePCA_inverse, index=X_train.index)

anomalyScoresSparsePCA = anomalyScores(X_train, \
    X_train_sparsePCA_inverse)
preds = plotResults(y_train, anomalyScoresSparsePCA, True)
```

Сгенерируем кривую "точность — полнота" и ROC-кривую.

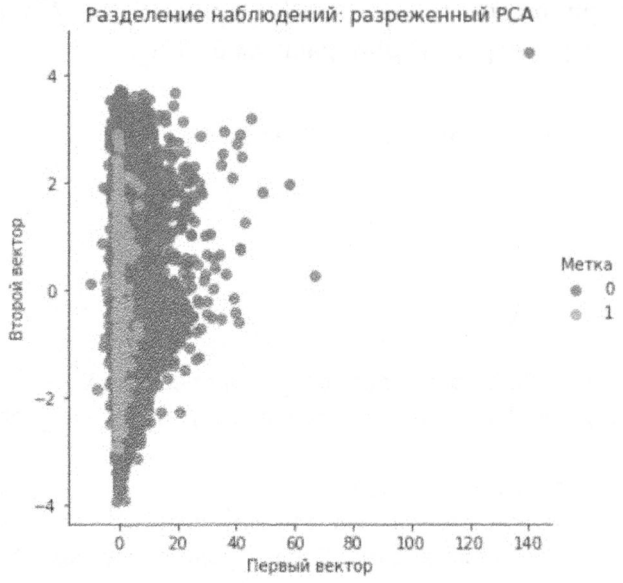


Рис. 4.5. *Разделение наблюдений с использованием разреженного метода PCA и 27 главных компонент*

Как следует из рис. 4.6, данные результаты аналогичны тем, которые были получены с использованием стандартного метода PCA. В этом нет ничего неожиданного, поскольку стандартный и разреженный методы PCA очень схожи между собой: последний — просто разреженное представление первого.

Используя код, приведенный на сайте GitHub (`http://bit.ly/2Gd4v7e`), вы сможете провести собственные эксперименты, изменяя количество генерируемых главных компонент и гиперпараметр `alpha`, однако наши эксперименты указывают на то, что данное решение на основе разреженного метода PCA является наилучшим.

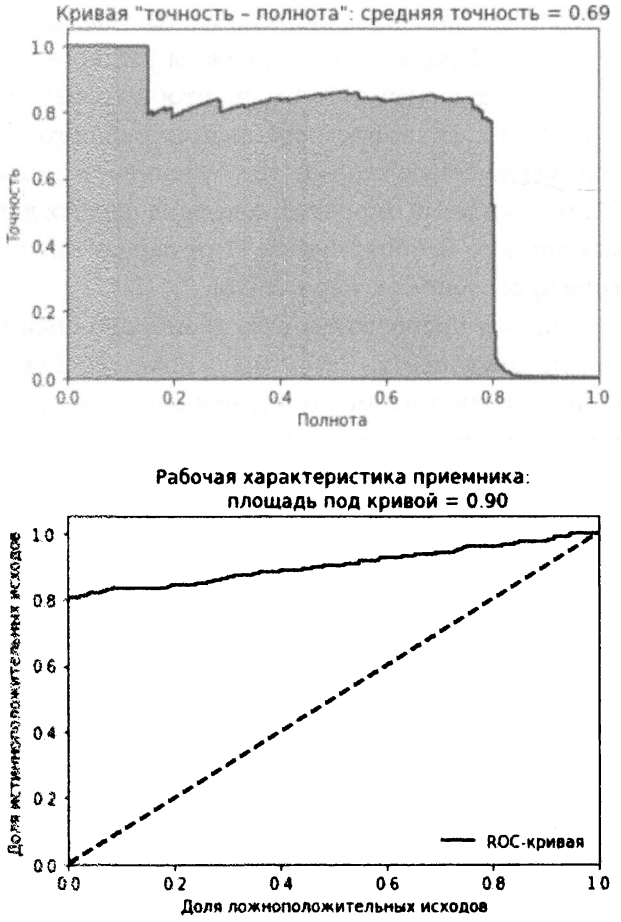


Рис. 4.6. *Результаты, полученные с использованием разреженного метода PCA и 27 главных компонент*

Обнаружение аномалий с помощью ядерного метода PCA

Следующим шагом будет применение ядерного метода PCA, который представляет собой нелинейную разновидность PCA, используемую в тех случаях, когда поддельные и подлинные транзакции не являются линейно разделимыми.

Нам нужно указать требуемое количество генерируемых компонент, ядро (мы используем RBF-ядро, как это делали в предыдущей главе) и гиперпараметр

gamma (для которого по умолчанию устанавливается значение 1/n_features, т.е. 1/30 в нашем случае). Кроме того, мы должны установить для переменной fit_inverse_transform значение true, чтобы применить встроенную функцию inverse_transform, предоставляемую библиотекой Scikit-learn.

Наконец, учитывая высокую стоимость обучения с использованием ядерного метода PCA, мы будем тренировать модель на первых двух тысячах примеров из транзакционного набора данных. Этот вариант не идеальный, но он обеспечит быстрое проведение экспериментов.

В ходе тренировки мы преобразуем весь тренировочный набор данных и сгенерируем главные компоненты. Затем мы используем функцию inverse_transform для воссоздания набора оригинальной размерности из главных компонент, полученных с помощью ядерного метода PCA.

```
# Ядерный PCA
from sklearn.decomposition import KernelPCA

n_components = 27
kernel = 'rbf'
gamma = None
fit_inverse_transform = True
random_state = 2018
n_jobs = 1

kernelPCA = KernelPCA(n_components=n_components, kernel=kernel, \
    gamma=gamma, fit_inverse_transform=fit_inverse_transform, \
    n_jobs=n_jobs, random_state=random_state)

kernelPCA.fit(X_train.iloc[:2000])
X_train_kernelPCA = kernelPCA.transform(X_train)
X_train_kernelPCA = pd.DataFrame(data=X_train_kernelPCA, \
                                 index=X_train.index)

X_train_kernelPCA_inverse =
    kernelPCA.inverse_transform(X_train_kernelPCA)
X_train_kernelPCA_inverse =
    pd.DataFrame(data=X_train_kernelPCA_inverse, index=X_train.index)

scatterPlot(X_train_kernelPCA, y_train, "ядерный PCA")
```

Точечная диаграмма для ядерного метода PCA приведена на рис. 4.7.

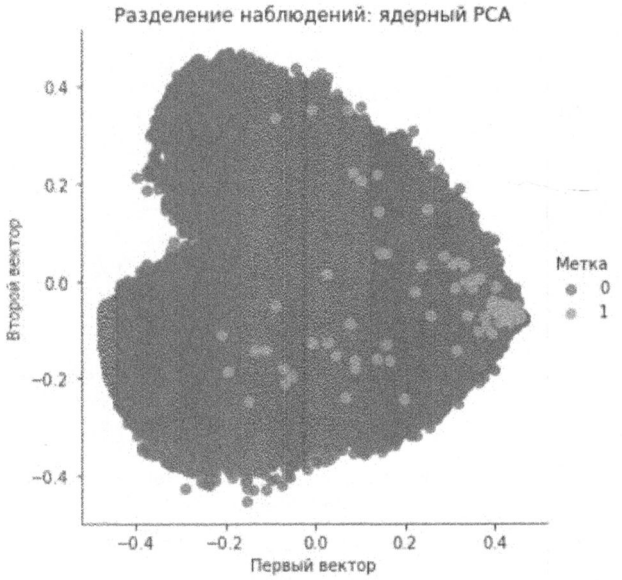


***Рис. 4.7.** Разделение наблюдений с использованием ядерного метода PCA и 27 главных компонент*

Вычислим оценки аномальности и выведем результаты.

Как следует из рис. 4.8, данные результаты намного хуже тех, которые были получены с помощью стандартного и разреженного методов PCA. С ядерным методом PCA стоит поэкспериментировать, но мы не будем применять его для обнаружения фактов мошенничества, поскольку предыдущие решения продемонстрировали лучшие результаты.

Мы не будем создавать решение на основе сингулярного разложения, поскольку результат очень схож с решением на основе стандартного метода PCA. Этого и следовало ожидать, ведь методы PCA и SVD тесно связаны между собой.

Обнаружение аномалий с помощью гауссовской случайной проекции

Теперь попытаемся разработать решение с использованием гауссовской случайной проекции. Вспомните, что в данном случае можно задать либо требуемое количество компонент, либо значение гиперпараметра `eps`, контролирующего качество вложения, полученного на основании леммы Джонсона — Линденштрауса.

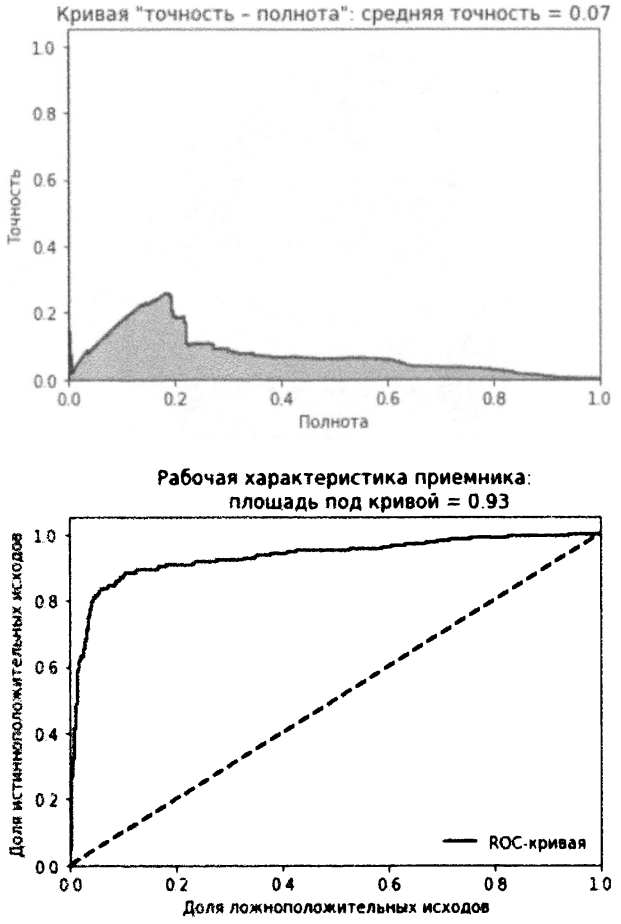


Рис. 4.8. *Результаты, полученные с использованием ядерного метода PCA и 27 главных компонент*

Мы остановимся на варианте, предполагающем явное задание количества компонент. Гауссовские случайные проекции обучаются очень быстро, поэтому мы сможем проводить обучение, используя полный тренировочный набор.

Как и в случае разреженного метода PCA, нам потребуется собственная функция `inverse_transform`, поскольку такая функция не предоставляется библиотекой Scikit-learn.

```
# Гауссовская случайная проекция
from sklearn.random_projection import GaussianRandomProjection

n_components = 27
eps = None
```

```
random_state = 2018

GRP = GaussianRandomProjection(n_components=n_components, eps=eps, \
                               random_state=random_state)

X_train_GRP = GRP.fit_transform(X_train)
X_train_GRP = pd.DataFrame(data=X_train_GRP, index=X_train.index)

scatterPlot(X_train_GRP, y_train, "гауссовская случайная проекция")
```

Точечная диаграмма для гауссовской случайной проекции приведена на рис. 4.9, а соответствующие результаты — на рис. 4.10.

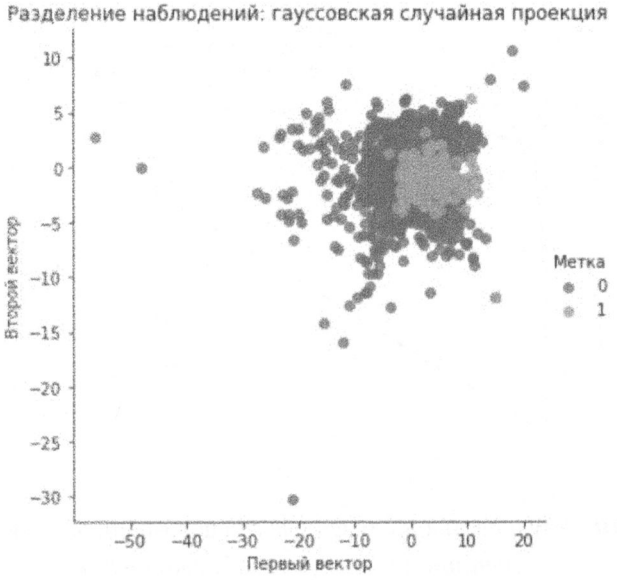


Рис. 4.9. *Разделение наблюдений с использованием гауссовской случайной проекции и 27 компонент*

Такие результаты нас не устраивают, поэтому мы не будем применять гауссовские случайные проекции для обнаружения фальсификаций.

Обнаружение аномалий с помощью разреженной случайной проекции

Попробуем спроектировать решение с использованием разреженной случайной проекции.

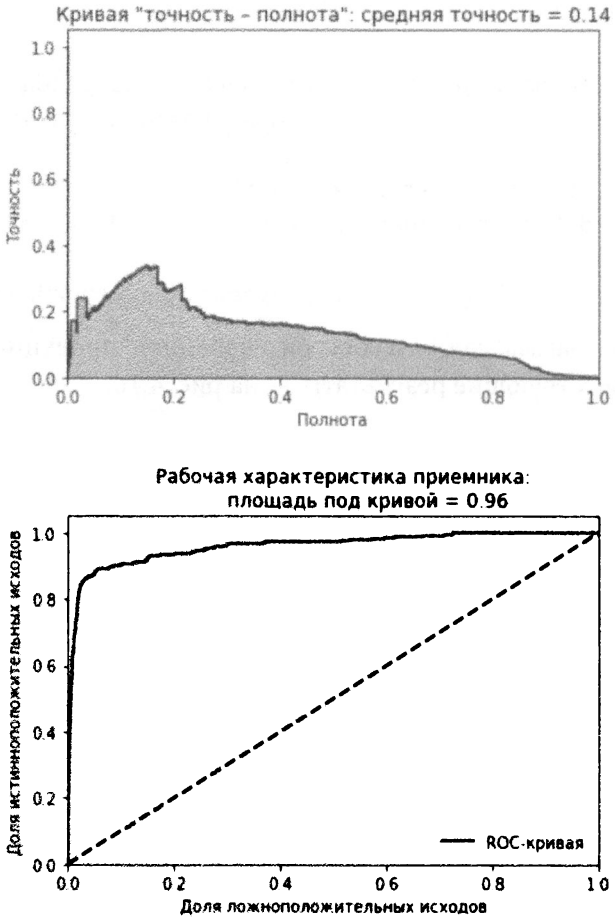


Рис. 4.10. *Результаты, полученные с использованием гауссовской случайной проекции и 27 компонент*

В данном случае вместо значения гиперпараметра `eps` мы укажем требуемое количество компонент. Кроме того, подобно решению с гауссовской случайной проекцией, мы используем собственную функцию `inverse_transform` для создания оригинальных измерений из компонент, полученных методом разреженной случайной проекции.

```
# Разреженная случайная проекция
from sklearn.random_projection import SparseRandomProjection

n_components = 27
density = 'auto'
eps = .01
```

```
dense_output = True
random_state = 2018

SRP = SparseRandomProjection(n_components=n_components, \
    density=density, eps=eps, dense_output=dense_output, \
    random_state=random_state)

X_train_SRP = SRP.fit_transform(X_train)
X_train_SRP = pd.DataFrame(data=X_train_SRP, index=X_train.index)

scatterPlot(X_train_SRP, y_train, "разреженная случайная проекция")
```

Точечная диаграмма для разреженной случайной проекции приведена на рис. 4.11, а соответствующие результаты — на рис. 4.12.

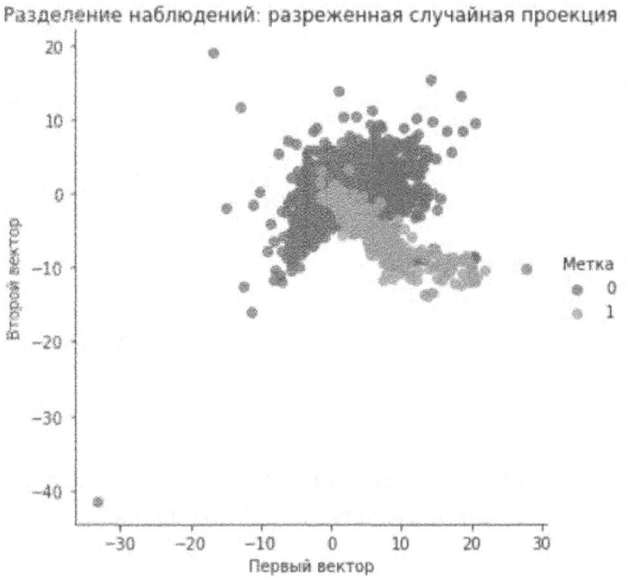


***Рис. 4.11.** Разделение наблюдений с использованием разреженной случайной проекции и 27 компонент*

Как видим, этот вариант также не привел к удовлетворительным результатам, поэтому мы продолжим наше исследование, используя другие методы снижения размерности.

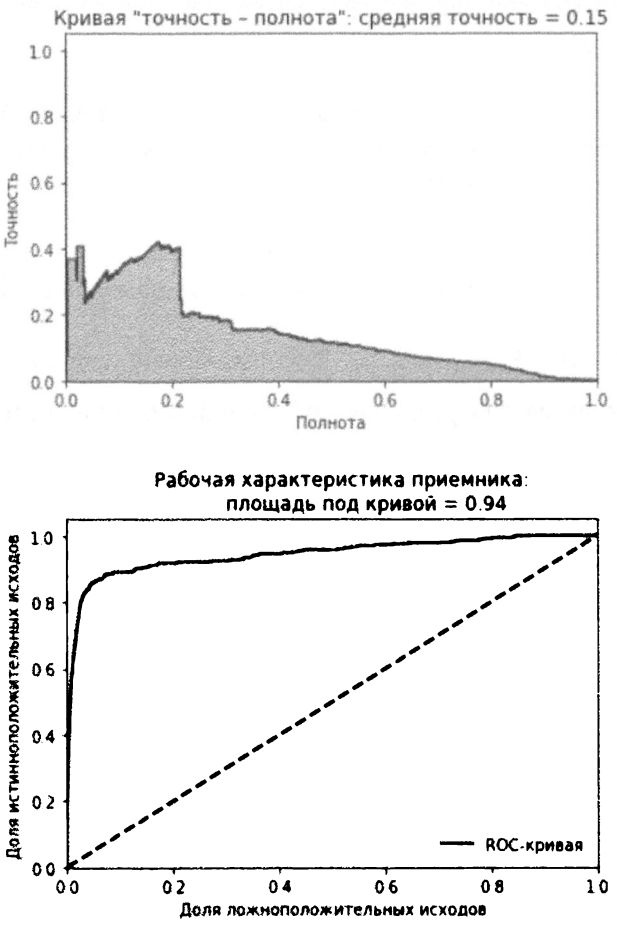


Рис. 4.12. *Результаты, полученные с использованием разреженной случайной проекции и 27 компонент*

Нелинейные методы обнаружения аномалий

До сих пор мы разрабатывали решения, в которых использовались линейные методы снижения размерности, такие как стандартный PCA, разреженный PCA, гауссовская случайная проекция и разреженная случайная проекция. Мы также разработали решение на основе нелинейной версии PCA — ядерного метода PCA.

К настоящему моменту наилучшим из исследованных нами решений является стандартный метод PCA.

Мы могли бы обратиться к нелинейным алгоритмам снижения размерности, но версии этих алгоритмов с открытым исходным кодом работают очень медленно и не годятся для быстрого обнаружения попыток мошенничества. Поэтому мы пропустим этот класс методов и сразу же перейдем к рассмотрению методов снижения размерности, не основанных на пространственных метриках: словарное обучение и анализ независимых компонент.

Обнаружение аномалий с помощью словарного обучения

На этот раз мы разработаем решение, предполагающее словарное обучение. Вспомните, что соответствующий алгоритм обучается разреженному представлению исходных данных. Использование векторов в обучаемом словаре позволяет реконструировать каждый экземпляр в виде взвешенной суммы этих обученных векторов.

Для обнаружения аномалий мы хотим использовать неполный словарь, чтобы количество векторов было меньше количества оригинальных измерений. Это ограничение упрощает реконструкцию обычных транзакций, которые встречаются наиболее часто, и намного затрудняет конструирование редко встречающихся поддельных транзакций.

В нашем случае мы будем генерировать 28 векторов (или компонент). В процессе обучения словарю будет передано 10 пакетов, каждый из которых содержит 200 образцов.

Нам также понадобится собственная функция `inverse_transform`.

```
# Мини-пакетное словарное обучение
from sklearn.decomposition import MiniBatchDictionaryLearning

n_components = 28
alpha = 1
batch_size = 200
n_iter = 10
random_state = 2018

miniBatchDictLearning = \
    MiniBatchDictionaryLearning(n_components=n_components, \
        alpha=alpha, batch_size=batch_size, n_iter=n_iter, \
        random_state=random_state)
```

```
miniBatchDictLearning.fit(X_train)
X_train_miniBatchDictLearning = \
    miniBatchDictLearning.fit_transform(X_train)
X_train_miniBatchDictLearning = \
    pd.DataFrame(data=X_train_miniBatchDictLearning, \
    index=X_train.index)

scatterPlot(X_train_miniBatchDictLearning, y_train, \
            "мини-пакетное словарное обучение")
```

Точечная диаграмма для словарного обучения приведена на рис. 4.13, а соответствующие результаты — на рис. 4.14.

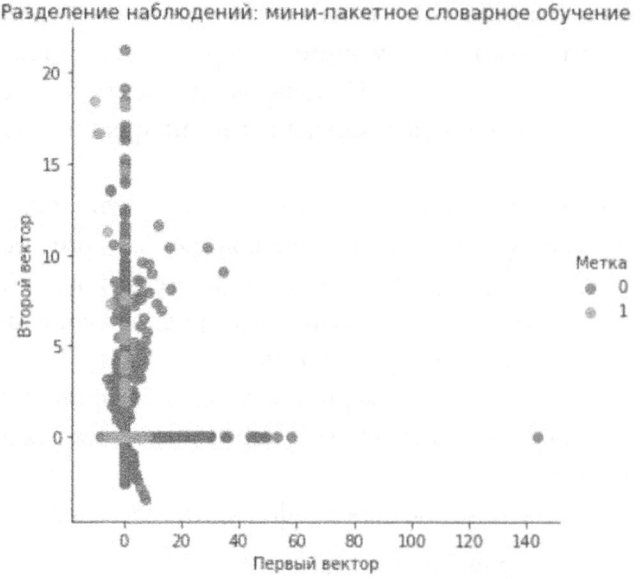


Рис. 4.13. *Разделение наблюдений с использованием словарного обучения и 28 компонент*

Эти результаты гораздо лучше, чем те, которые были получены с использованием ядерного PCA, а также гауссовской и разреженной случайной проекции, но несравнимы с результатами стандартного метода PCA.

Используя код, доступный на сайте GitHub, вы сможете провести собственные эксперименты, чтобы выяснить, удастся ли улучшить данное решение, но пока что лидерство остается за PCA.

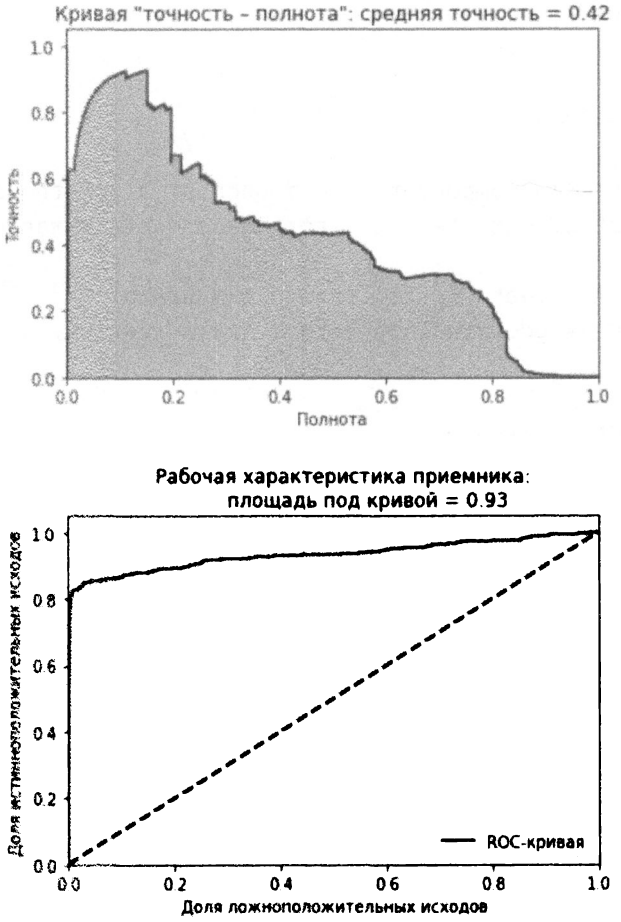


Рис. 4.14. *Результаты, полученные с использованием словарного обучения и 28 компонент*

Обнаружение аномалий с помощью метода ICA

Теперь мы создадим решение на основе метода ICA.

Нам нужно указать количество компонент, которое мы установим равным 27. Библиотека Scikit-learn предоставляет функцию `inverse_transform`, поэтому в создании собственной функции нет никакой необходимости.

```
# Анализ независимых компонент
from sklearn.decomposition import FastICA

n_components = 27
```

```
algorithm = 'parallel'
whiten = True
max_iter = 200
random_state = 2018

fastICA = FastICA(n_components=n_components, algorithm=algorithm, \
    whiten=whiten, max_iter=max_iter, random_state=random_state)

X_train_fastICA = fastICA.fit_transform(X_train)
X_train_fastICA = pd.DataFrame(data=X_train_fastICA, \
                               index=X_train.index)

X_train_fastICA_inverse =
    fastICA.inverse_transform(X_train_fastICA)
X_train_fastICA_inverse =
    pd.DataFrame(data=X_train_fastICA_inverse, index=X_train.index)

scatterPlot(X_train_fastICA, y_train, "анализ независимых компонент")
```

Точечная диаграмма для метода ICA приведена на рис. 4.15, а соответствующие результаты — на рис. 4.16.

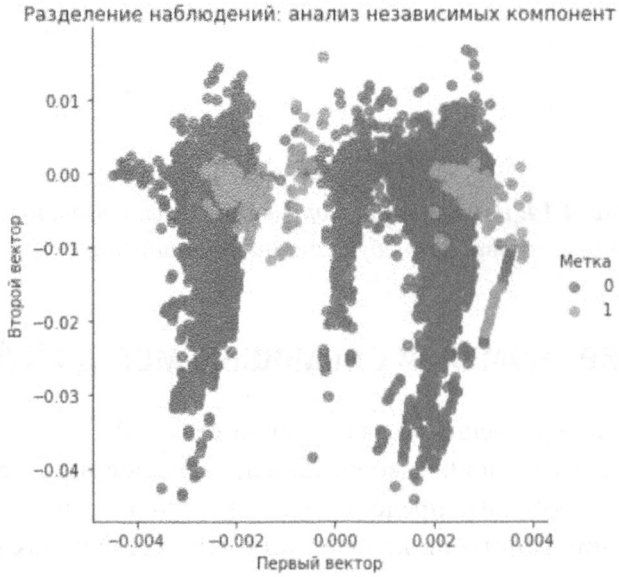


Рис. 4.15. *Разделение наблюдений с использованием метода ICA и 27 компонент*

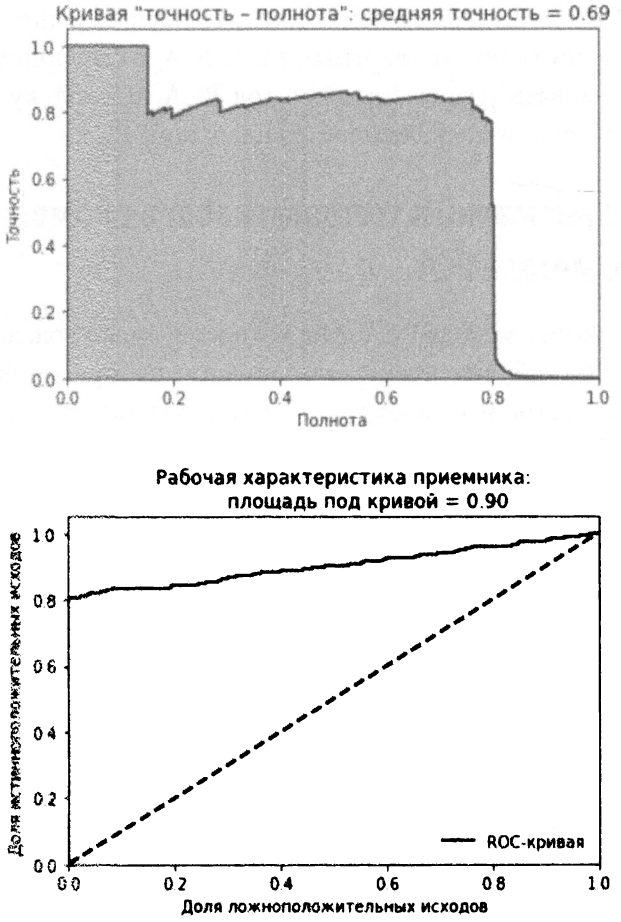


Рис. 4.16. *Результаты, полученные с использованием метода ICA и 27 компонент*

Эти результаты совпадают с теми, которые были нолучены с использованием стандартного метода PCA. Таким образом, данное решение сравнимо с наилучшим решением из тех, которые мы исследовали до сих пор.

Обнаружение попыток мошенничества на тестовом наборе

Оценим наши решения по обнаружению мошеннических транзакций, применив их к тестовому набору, содержащему примеры, которые до сих пор не

встречались. Мы сделаем это в отношении трех лучших решений из числа тех, которые мы разработали: стандартный PCA, ICA и словарное обучение. Мы не будем использовать разреженный метод PCA, поскольку его результаты очень близки к результатам на основе стандартного PCA.

Обнаружение аномалий в тестовом наборе с помощью стандартного метода PCA

Начнем с обычного метода PCA. Мы воспользуемся вложением PCA, которому алгоритм PCA обучился на тренировочном наборе, и применим его для преобразования тестового набора. Затем мы вызовем функцию `inverse_transform` из библиотеки Scikit-learn для воссоздания оригинальных измерений из матрицы главных компонент тестового набора.

Сравнив между собой матрицы оригинального и воссозданного наборов, мы сможем рассчитать оценки аномальности (как неоднократно делали ранее).

```
# Применение метода PCA к тестовому набору
X_test_PCA = pca.transform(X_test)
X_test_PCA = pd.DataFrame(data=X_test_PCA, index=X_test.index)

X_test_PCA_inverse = pca.inverse_transform(X_test_PCA)
X_test_PCA_inverse = pd.DataFrame(data=X_test_PCA_inverse, \
                                  index=X_test.index)

scatterPlot(X_test_PCA, y_test, "PCA")
```

Точечная диаграмма для метода PCA, примененного к тестовому набору, приведена на рис. 4.17, а соответствующие результаты — на рис. 4.18.

Полученные результаты впечатляют. Мы смогли захватить 80% известных поддельных транзакций в тестовом наборе с 80%-ной точностью — и все это без использования меток!

Обнаружение аномалий в тестовом наборе с помощью метода ICA

Перейдем к методу ICA и применим его для обнаружения мошеннических транзакций в тестовом наборе.

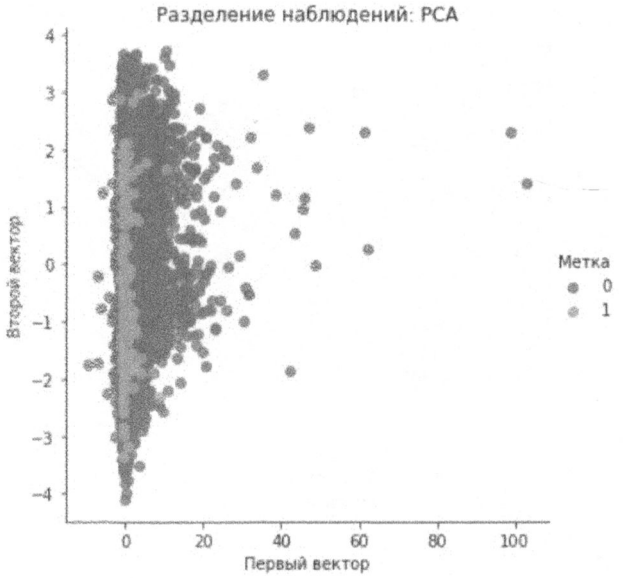


Рис. 4.17. Разделение наблюдений с использованием метода PCA и 27 компонент для тестового набора

```
# Применение анализа независимых компонент к тестовому набору
X_test_fastICA = fastICA.transform(X_test)
X_test_fastICA = pd.DataFrame(data=X_test_fastICA, \
                              index=X_test.index)

X_test_fastICA_inverse = fastICA.inverse_transform(X_test_fastICA)
X_test_fastICA_inverse = pd.DataFrame(data=X_test_fastICA_inverse, \
                                      index=X_test.index)

scatterPlot(X_test_fastICA, y_test, "анализ независимых компонент")
```

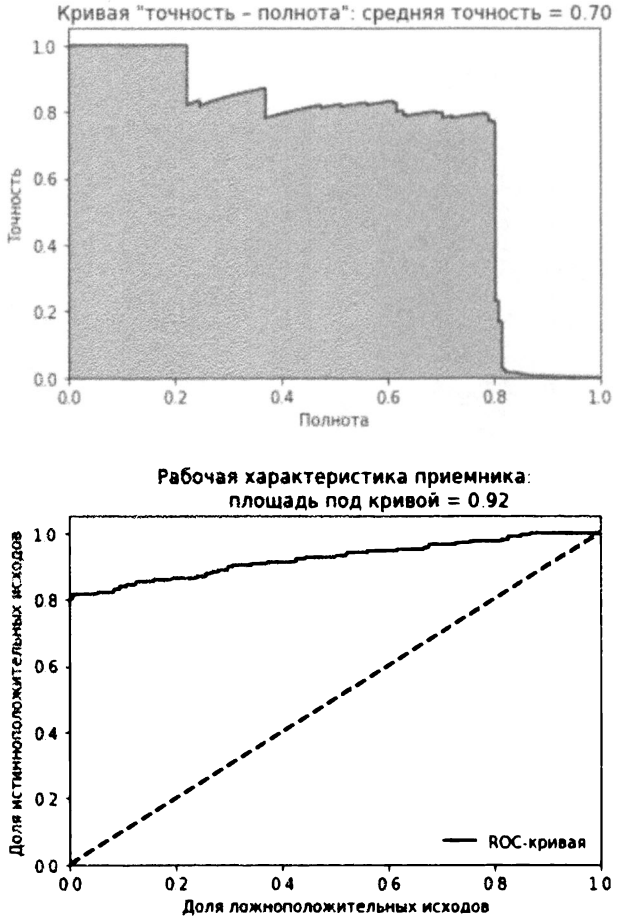


Рис. 4.18. *Результаты, полученные с использованием метода PCA и 27 компонент на тестовом наборе*

Точечная диаграмма для метода ICA, примененного к тестовому набору, приведена на рис. 4.19, а соответствующие результаты — на рис. 4.20.

Эти результаты идентичны тем, которые были получены с использованием стандартного метода PCA, а значит, тоже впечатляющие.

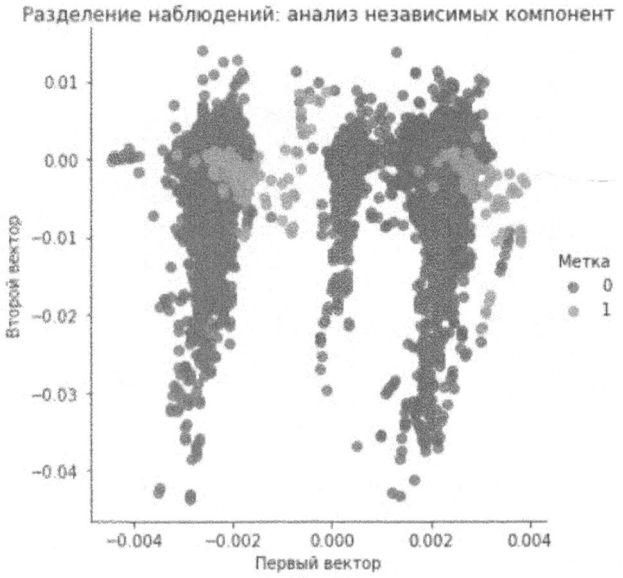


Рис. 4.19. *Разделение наблюдений с использованием метода ICA и 27 компонент для тестового набора*

Обнаружение аномалий в тестовом наборе с помощью словарного обучения

Наконец, обратимся к словарному обучению, которое, хотя и не смогло продемонстрировать столь же хороших результатов, что и методы PCA/ICA, все же заслуживает финального рассмотрения.

```
X_test_miniBatchDictLearning = \
    miniBatchDictLearning.transform(X_test)
X_test_miniBatchDictLearning = \
    pd.DataFrame(data=X_test_miniBatchDictLearning, \
                 index=X_test.index)

scatterPlot(X_test_miniBatchDictLearning, y_test, \
            "мини-пакетное словарное обучение")
```

Точечная диаграмма для словарного обучения, примененного к тестовому набору, приведена на рис. 4.21, а соответствующие результаты — на рис. 4.22.

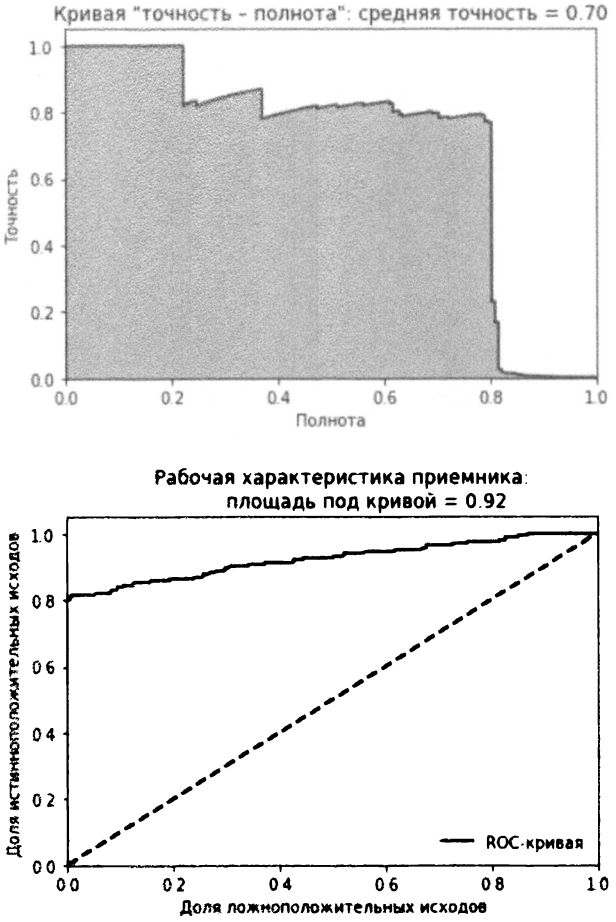


Рис. 4.20. *Результаты, полученные с использованием метода ICA и 27 компонент на тестовом наборе*

Несмотря на то что эти результаты не являются катастрофически плохими (мы смогли захватить 80% подделок с 20%-ной точностью), им все же далеко до результатов, полученных с использованием методов PCA и ICA.

Резюме

В этой главе мы применили основные алгоритмы снижения размерности для разработки системы по обнаружению мошеннических операций с банковскими картами в наборе данных, который был описан в главе 2.

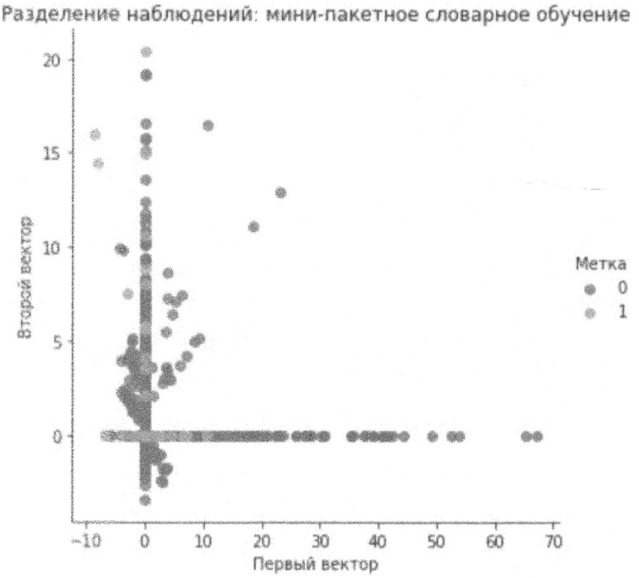


***Рис. 4.21.** Разделение наблюдений с использованием словарного обучения и 28 компонент для тестового набора*

В главе 2 мы создавали систему на основе меток, здесь же они нам не понадобились. Другими словами, мы реализовали систему обучения без учителя.

Несмотря на то что не все алгоритмы снижения размерности должным образом проявили себя при работе с транзакционным набором данных, два из них (стандартный метод PCA и метод ICA) замечательно справились со своей задачей. Каждому из методов — PCA и ICA — удалось захватить свыше 80% известных мошеннических транзакций. Для сравнения отметим, что наилучшая из исследованных в главе 2 систем обучения с учителем захватила свыше 90% мошеннических транзакций с 80%-ной точностью. Система на основе обучения без учителя продемонстрировала лишь незначительное ухудшение производительности при захвате известных примеров мошенничества.

Вспомните, что тренировка систем обнаружения фальсификаций на основе обучения без учителя не требует использования меток. Такие системы хорошо адаптируются к изменению мошеннических схем и способны захватывать новые типы поддельных образцов. С учетом этих дополнительных преимуществ системы на основе обучения без учителя обычно работают лучше систем на основе обучения с учителем в отношении захвата как известных, так и неизвестных (которые могут появиться в будущем) схем мошенничества, хотя оптимальным вариантом является совместное использование обоих типов систем.

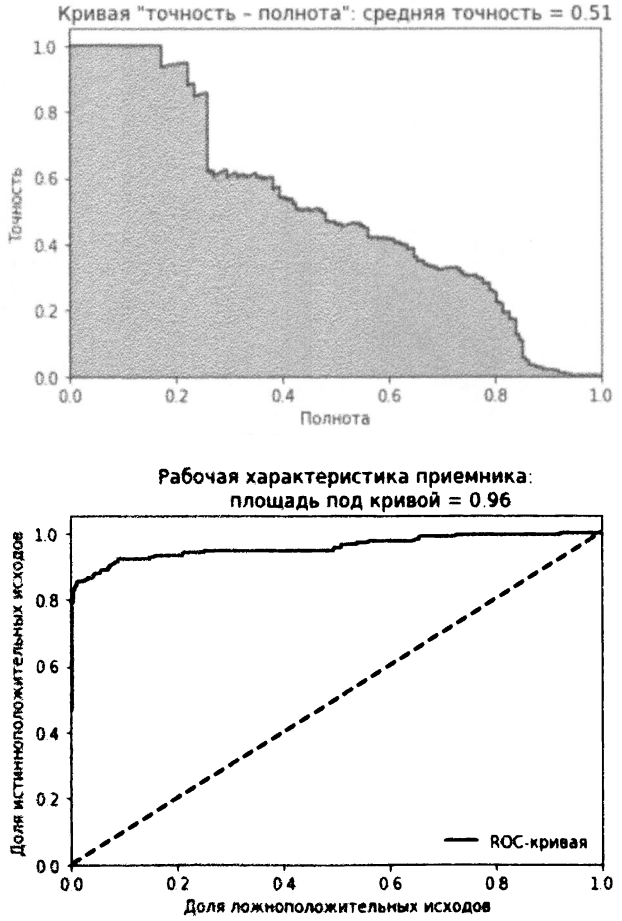


Рис. 4.22. *Результаты, полученные с использованием словарного обучения и 28 компонент на тестовом наборе*

Теперь, когда мы познакомились с алгоритмами снижения размерности и обнаружения аномалий, можно перейти к исследованию *кластеризации* — еще одного фундаментального понятия в области обучения без учителя.

ГЛАВА 5

Кластеризация

В главе 3 мы познакомились с наиболее важными алгоритмами снижения размерности, основанными на обучении без учителя, и увидели их возможности в отношении захвата важной информации. В главе 4 мы использовали алгоритмы снижения размерности для построения системы обнаружения аномалий. В частности, мы применили эти алгоритмы для выявления мошеннических операций с банковскими картами без использования меток. Алгоритмы обучались внутренней структуре транзакций, после чего мы отделили обычные транзакции от редко встречающихся (потенциально мошеннических) транзакций на основании ошибки реконструкции данных.

В этой главе мы продолжим знакомство с методами обучения без учителя и рассмотрим концепцию *кластеризации*, предполагающей группирование объектов на основе их взаимного сходства. Суть подхода заключается в сравнении данных одного наблюдения с данными других наблюдений и определении степени их сходства без использования меток.

Кластеризация находит множество применений. Например, в случае мошенничества с банковскими картами кластеризация позволяет сгруппировать поддельные транзакции, отделив их от нормальных транзакций. Или же, если мы располагаем метками для небольшого количества наблюдений в наборе данных, кластеризацию можно использовать для предварительного группирования наблюдений (без использования меток). Затем мы можем перенести метки имеющихся помеченных наблюдений на остальные наблюдения, входящие в ту же группу. Этот подход представляет собой одну из форм *переносимого обучения* (transfer learning), быстро развивающейся области машинного обучения.

В таких сферах деятельности, как электронная коммерция и розничная торговля, маркетинг, социальные сети, рекомендательные системы фильмов, службы знакомств и т.п., кластеризация позволяет группировать людей на основании их персональных предпочтений. Установление таких групп помогает предпринимателям лучше понимать своих клиентов и создает основу для формирования целевых бизнес-стратегий, ориентированных на специфику каждой конкретной группы.

Аналогично тому, как мы поступали при рассмотрении алгоритмов снижения размерности, сначала мы введем основные понятия и лишь после этого приступим в следующей главе к созданию приложения на основе обучения без учителя.

База данных рукописных цифр MNIST

Чтобы не усложнять примеры, мы продолжим работать с набором изображений MNIST, о котором шла речь в главе 3.

Подготовка данных

Сначала необходимо загрузить требуемые библиотеки.

```
# Импорт библиотек
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time
import pickle, gzip

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score
```

Далее следует загрузить набор данных и создать объекты `DataFrames` библиотеки Pandas.

```
# Загрузка наборов данных
current_path = os.getcwd()
```

```
file = os.path.sep.join(['', 'datasets', 'mnist_data', \
                         'mnist.pkl.gz'])

f = gzip.open(current_path+file, 'rb')
train_set, validation_set, test_set = \
    pickle.load(f, encoding='latin1')
f.close()

X_train, y_train = train_set[0], train_set[1]
X_validation, y_validation = validation_set[0], validation_set[1]
X_test, y_test = test_set[0], test_set[1]

# Создание объектов DataFrame библиотеки Pandas из наборов данных
train_index = range(0, len(X_train))
validation_index = range(len(X_train), \
                         len(X_train)+len(X_validation))
test_index = range(len(X_train)+len(X_validation), \
                   len(X_train)+len(X_validation)+len(X_test))

X_train = pd.DataFrame(data=X_train, index=train_index)
y_train = pd.Series(data=y_train, index=train_index)

X_validation = pd.DataFrame(data=X_validation, \
                            index=validation_index)
y_validation = pd.Series(data=y_validation, index=validation_index)

X_test = pd.DataFrame(data=X_test, index=test_index)
y_test = pd.Series(data=y_test, index=test_index)
```

Алгоритмы кластеризации

Прежде чем выполнять кластеризацию, необходимо снизить размерность данных, используя метод PCA. Как было показано в главе 3, алгоритмы снижения размерности захватывают наиболее существенную информацию, содержащуюся в наборе данных, одновременно уменьшая его размер.

По мере того как мы переходим от большего числа измерений к меньшему, содержание шума в наборе данных минимизируется, поскольку алгоритм снижения размерности (PCA в данном случае) должен захватывать наиболее важные элементы оригинальных данных, игнорируя редко встречающиеся элементы (такие, как содержащийся в данных шум).

Вспомните: алгоритмы снижения размерности обладают чрезвычайно мощными возможностями в плане обучения базовой структуре данных. В главе 3 было показано, насколько эффективно можно разделять изображения MNIST на основании представляемых ими цифр, используя всего лишь два измерения, полученные в результате снижения размерности.

Вновь применим метод PCA к набору данных MNIST.

```
# Анализ главных компонент
from sklearn.decomposition import PCA

n_components = 784
whiten = False
random_state = 2018

pca = PCA(n_components=n_components, whiten=whiten, \
          random_state=random_state)

X_train_PCA = pca.fit_transform(X_train)
X_train_PCA = pd.DataFrame(data=X_train_PCA, index=train_index)
```

И хотя мы не снизили размерность набора явным образом, мы обозначили количество главных компонент, которые будут использоваться на этапе кластеризации, что равносильно снижению размерности.

Теперь приступим к кластеризации. Три основных алгоритма кластеризации — *метод* k-*средних*, *иерархическая кластеризация* и *DBSCAN*. Рассмотрим каждый из них по отдельности.

Метод *k*-средних

Цель кластеризации — выявить в наборе данных такие группы, для которых наблюдения в пределах одной группы сходны между собой, но отличаются от наблюдений в других группах. В случае кластеризации методом *k*-средних мы указываем требуемое количество кластеров, *k*, и алгоритм будет приписывать каждое наблюдение только к одному из этих *k* кластеров. Алгоритм оптимизирует группы путем минимизации *внутрикластерной вариации* (также известной как *инерция*), чтобы сумма внутрикластерных вариаций по всем *k* кластерам была наименьшей.

Разные запуски алгоритма *k*-средних будут приводить к немного разным распределениям наблюдений по кластерам. Это обусловлено тем, что алгоритм *k*-средних случайным образом приписывает каждое наблюдение к одному из

k кластеров для ускоренной инициализации процесса кластеризации. После этапа случайной инициализации алгоритм заново относит наблюдения к различным кластерам, пытаясь минимизировать евклидово расстояние между каждым наблюдением и центральной точкой его кластера, или *центроидом*. Такая инициализация служит источником случайности, что и приводит к незначительным изменениям в распределении наблюдений по кластерам от одного запуска алгоритма к другому.

Обычно алгоритм *k*-средних запускается несколько раз и выбирает запуск с наилучшим разделением, которое определяется наименьшей общей суммой внутрикластерных вариаций по всем *k* кластерам.

Инерция метода *k*-средних

Познакомимся поближе с самим алгоритмом. Мы должны указать требуемое количество кластеров (`n_clusters`), число инициализаций, которые мы хотели бы выполнить (`n_init`), максимальное количество итераций, выполняемых алгоритмом для отнесения наблюдений к кластерам в процессе минимизации инерции (`max_iter`), и допустимое отклонение (`tol`) в качестве критерия сходимости.

Мы оставим заданные по умолчанию значения для количества инициализаций (`10`), максимального числа итераций (`300`) и допустимого отклонения (`0.0001`). Кроме того, пока что нам будет достаточно использовать первые 100 главных компонент PCA (`cutoff`). Чтобы проверить, насколько сильна зависимость инерции от количества кластеров, запустим алгоритм *k*-средних для кластеров с размерами от 2 до 20 и запишем значения инерции для каждого из них.

Вот соответствующий код.

```
# Метод k-средних - изменение инерции
# с изменением количества кластеров
from sklearn.cluster import KMeans

n_clusters = 10
n_init = 10
max_iter = 300
tol = 0.0001
random_state = 2018

kMeans_inertia = pd.DataFrame(data=[], index=range(2, 21), \
                              columns=['Инерция'])
```

```
for n_clusters in range(2, 21):
    kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, \
                    max_iter=max_iter, tol=tol, \
                    random_state=random_state)

    cutoff = 99
    kmeans.fit(X_train_PCA.loc[:, 0:cutoff])
    kMeans_inertia.loc[n_clusters] = kmeans.inertia_
```

Как следует из рис. 5.1, с увеличением количества кластеров инерция уменьшается, что вполне логично. Чем больше кластеров, тем выше степень однородности наблюдений в пределах каждого кластера. Однако с меньшим количеством кластеров легче работать, поэтому нахождение подходящего количества кластеров является одним из важных факторов, которые необходимо учитывать, работая по методу *k*-средних.

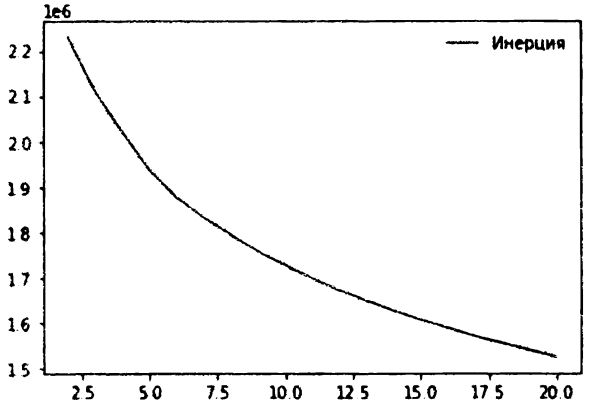


Рис. 5.1. *Инерция метода k-средних для кластеров с размерами от 2 до 20*

Оценка результатов кластеризации

Чтобы продемонстрировать, как работает метод *k*-средних и как увеличение количества кластеров приводит к повышению их однородности, определим функцию, анализирующую результаты каждого выполненного нами эксперимента. Распределения по кластерам, генерируемые алгоритмом кластеризации, будут сохраняться в объекте `DataFrame` библиотеки Pandas с именем `clusterDF`.

Подсчитаем число наблюдений в каждом кластере и сохраним эти значения в объекте `DataFrame` библиотеки Pandas с именем `countByCluster`.

```
def analyzeCluster(clusterDF, labelsDF):
    countByCluster = \
        pd.DataFrame(data=clusterDF['cluster'].value_counts())
    countByCluster.reset_index(inplace=True, drop=False)
    countByCluster.columns = ['cluster', 'clusterCount']
```

Далее присоединим объект `clusterDF` к массиву истинных меток, присвоив ему имя `labelsDF`.

```
    preds = pd.concat([labelsDF,clusterDF], axis=1)
    preds.columns = ['trueLabel', 'cluster']
```

Подсчитаем также количество наблюдений для каждой истинной метки в тренировочном наборе (оно не будет меняться, но знать его полезно).

```
    countByLabel =
        pd.DataFrame(data=preds.groupby('trueLabel').count())
```

Дополнительно подсчитаем количество наблюдений для каждой отдельной метки внутри каждого кластера. Если в заданном кластере имеется три тысячи наблюдений, то из них две тысячи могут представлять цифру 2, пятьсот — цифру 1, триста — цифру 0, а оставшиеся двести — цифру 9.

Выполнив эти расчеты, мы сохраним значения счетчиков наиболее часто встречающихся цифр для каждого кластера. В приведенном примере мы сохранили бы значение счетчика, соответствующего двум тысячам наблюдений.

```
    countMostFreq = \
        pd.DataFrame(data=preds.groupby('cluster').agg(lambda \
            x:x.value_counts().iloc[0]))
    countMostFreq.reset_index(inplace=True, drop=False)
    countMostFreq.columns = ['cluster', 'countMostFrequent']
```

Наконец, мы будем судить об успешности каждого запуска кластеризации на основании того, насколько тесно сгруппированы наблюдения в пределах каждого кластера. В приведенном выше примере в кластере, содержащем в общей сложности три тысячи наблюдений, имеются две тысячи наблюдений с одной и той же меткой. Такой кластер не представляет собой ничего особенного, поскольку в идеальном случае мы хотели бы группировать в одном и том же кластере все сходные наблюдения и исключать наблюдения, отличающиеся от них.

Определим общую точность кластеризации как сумму значений счетчиков наиболее часто встречающихся наблюдений по всем кластерам, деленную на общее количество наблюдений в тренировочном наборе (т.е. 50 000).

```
    accuracyDF = countMostFreq.merge(countByCluster, \
        left_on="cluster", right_on="cluster")
    overallAccuracy = accuracyDF.countMostFrequent.sum() / \
                      accuracyDF.clusterCount.sum()
```

Мы также можем оценить точность для одиночного кластера.

```
    accuracyByLabel = accuracyDF.countMostFrequent / \
                      accuracyDF.clusterCount
```

Полный код этого примера доступен в виде единой функции на сайте GitHub (`http://bit.ly/2Gd4v7e`).

Точность метода *k*-средних

Выполним эксперименты, как мы это делали раньше, только теперь будем вычислять не инерцию, а суммарную однородность кластеров, основываясь на мере точности, которую мы определили для нашего набора MNIST, содержащего изображения рукописных цифр.

```
# Метод k-средних - изменение точности
# с изменением количества кластеров

n_clusters = 5
n_init = 10
max_iter = 300
tol = 0.0001
random_state = 2018

kMeans_inertia = pd.DataFrame(data=[], index=range(2, 21), \
    columns=['Инерция'])
overallAccuracy_kMeansDF = pd.DataFrame(data=[], \
    index=range(2, 21), columns=['Общая точность'])

for n_clusters in range(2, 21):
    kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, \
                    max_iter=max_iter, tol=tol, \
                    random_state=random_state)

    cutoff = 99
    kmeans.fit(X_train_PCA.loc[:, 0:cutoff])
    kMeans_inertia.loc[n_clusters] = kmeans.inertia_
    X_train_kmeansClustered = kmeans.predict(X_train_PCA.loc[:, \
        0:cutoff])
```

```
X_train_kmeansClustered = \
    pd.DataFrame(data=X_train_kmeansClustered, \
    index=X_train.index, columns=['cluster'])

countByCluster_kMeans, countByLabel_kMeans, \
    countMostFreq_kMeans, accuracyDF_kMeans, \
    overallAccuracy_kMeans, accuracyByLabel_kMeans = \
        analyzeCluster(X_train_kmeansClustered, y_train)

overallAccuracy_kMeansDF.loc[n_clusters] = overallAccuracy_kMeans
```

График общей точности для кластеров различного размера приведен на рис. 5.2.

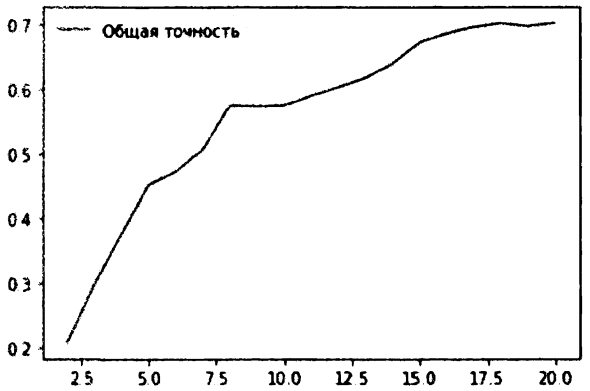


Рис. 5.2. *Точность метода k-средних для кластеров с размерами от 2 до 20*

Как следует из рис. 5.2, по мере увеличения количества кластеров точность увеличивается. Другими словами, с увеличением числа кластеров они становятся более однородными в силу того, что их размеры при этом уменьшаются и они становятся более компактными.

Точность может довольно заметно меняться при переходе от кластера к кластеру, поскольку одни кластеры проявляют более высокую степень однородности, другие — меньшую. Например, в некоторых кластерах свыше 90% изображений соответствуют одной и той же цифре, тогда как в других — менее 50%.

```
0    0.636506
1    0.928505
2    0.848714
3    0.521805
```

```
4    0.714337
5    0.950980
6    0.893103
7    0.919040
8    0.404707
9    0.500522
10   0.381526
11   0.587680
12   0.463382
13   0.958046
14   0.870888
15   0.942325
16   0.791192
17   0.843972
18   0.455679
19   0.926480
dtype: float64
```

Метод *k*-средних и количество главных компонент

Проведем еще один эксперимент, на этот раз чтобы оценить, как изменение количества главных компонент, используемых в алгоритме кластеризации, влияет на однородность кластеров (определенную в качестве меры точности).

В предыдущих экспериментах мы использовали 100 главных компонент, полученных в качестве выхода стандартного метода PCA. Вспомните, что количество измерений в первоначальном наборе данных MNIST равно 784. Если алгоритм PCA хорошо справляется с максимально компактным захватом базовой структуры данных, то алгоритму кластеризации будет несложно сгруппировать схожие изображения, независимо от того, используем мы лишь незначительную долю главных компонент или намного большее их количество. Другими словами, кластеризация должна выполняться одинаково хорошо при использовании как нескольких десятков, так и нескольких сотен главных компонент.

Давайте проверим эту гипотезу. Мы будем последовательно передавать алгоритму 10, 50, 100, 200, 300, 400, 500, 600, 700 и 784 компоненты и замерять точность кластеризации в каждом эксперименте. Затем мы отложим эти результаты на графике, чтобы увидеть, как изменение количества главных компонент влияет на точность кластеризации.

```
# Метод k-средних - зависимость точности
# от количества главных компонент

n_clusters = 20
n_init = 10
max_iter = 300
tol = 0.0001
random_state = 2018

kMeans_inertia = pd.DataFrame(data=[], index=[9, 49, 99, 199, 299, \
    399, 499, 599, 699, 784], columns=['Инерция'])

overallAccuracy_kMeansDF = pd.DataFrame(data=[],index=[9, 49, 99, \
    199, 299, 399, 499, 599, 699, 784], columns=['Общая точность'])

for cutoffNumber in [9, 49, 99, 199, 299, 399, 499, 599, 699, 784]:
    kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, \
                    max_iter=max_iter, tol=tol, \
                    random_state=random_state)

    cutoff = cutoffNumber
    kmeans.fit(X_train_PCA.loc[:, 0:cutoff])
    kMeans_inertia.loc[cutoff] = kmeans.inertia_
    X_train_kmeansClustered = kmeans.predict(X_train_PCA.loc[:, \
        0:cutoff])
    X_train_kmeansClustered = \
        pd.DataFrame(data=X_train_kmeansClustered, \
        index=X_train.index, columns=['cluster'])

    countByCluster_kMeans, countByLabel_kMeans, \
        countMostFreq_kMeans, accuracyDF_kMeans, \
        overallAccuracy_kMeans, accuracyByLabel_kMeans = \
        analyzeCluster(X_train_kmeansClustered, y_train)

    overallAccuracy_kMeansDF.loc[cutoff] = overallAccuracy_kMeans
```

График зависимости точности кластеризации от количества главных компонент приведен на рис. 5.3.

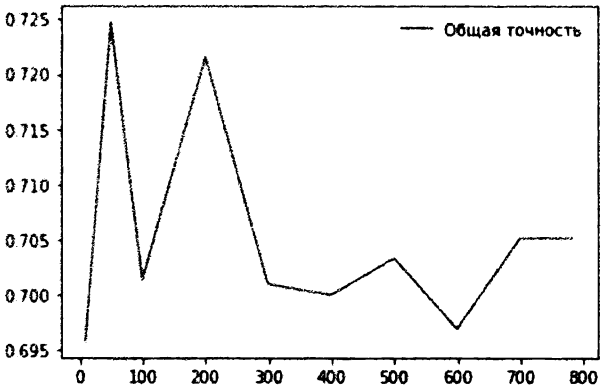


Рис. 5.3. *Зависимость точности кластеризации по методу k-средних от количества главных компонент*

Этот график подтверждает нашу гипотезу. При изменении количества главных компонент от 10 до 784 точность кластеризации ведет себя стабильно и согласованно, оставаясь на уровне около 70%. Это одна из причин того, почему кластеризацию следует применять к редуцированным наборам данных: обычно алгоритмы кластеризации работают на таких наборах лучше как в смысле точности кластеризации, так и в смысле быстродействия.

В случае набора данных MNIST с его исходными 784 измерениями алгоритм кластеризации способен справиться со своей задачей, но представьте ситуацию, когда количество измерений исчисляется тысячами или миллионами. В подобных сценариях снижение размерности набора данных становится еще более обоснованным.

Применение метода *k*-средних к оригинальному набору данных

Чтобы сделать приведенную выше аргументацию еще более убедительной, выполним кластеризацию на оригинальном наборе данных и выясним, как изменение количества измерений, передаваемого алгоритму кластеризации, влияет на точность кластеризации.

В случае рассмотренного в предыдущем разделе набора данных, редуцированного с помощью метода PCA, изменение количества главных компонент, передаваемых алгоритму кластеризации, не оказывало влияния на точность, которая стабильно держалась примерно на уровне 70%. Проверим, будет ли ситуация аналогичной в случае оригинального набора данных.

```
# Метод k-средних - зависимость точности от количества
# компонент для оригинального набора данных MNIST
# (а не редуцированного с помощью метода PCA)

n_clusters = 20
n_init = 10
max_iter = 300
tol = 0.0001
random_state = 2018

kMeans_inertia = pd.DataFrame(data=[], index=[9, 49, 99, 199, 299, \
    399, 499, 599, 699, 784], columns=['Инерция'])
overallAccuracy_kMeansDF = pd.DataFrame(data=[], index=[9, 49, 99, \
    199, 299, 399, 499, 599, 699, 784], columns=['Общая точность'])

for cutoffNumber in [9, 49, 99, 199, 299, 399, 499, 599, 699, 784]:
    kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, \
                    max_iter=max_iter, tol=tol, \
                    random_state=random_state)

    cutoff = cutoffNumber
    kmeans.fit(X_train.loc[:, 0:cutoff])
    kMeans_inertia.loc[cutoff] = kmeans.inertia_
    X_train_kmeansClustered = \
        kmeans.predict(X_train.loc[:, 0:cutoff])
    X_train_kmeansClustered =
        pd.DataFrame(data=X_train_kmeansClustered, \
        index=X_train.index, columns=['cluster'])

    countByCluster_kMeans, countByLabel_kMeans, \
        countMostFreq_kMeans, accuracyDF_kMeans, \
        overallAccuracy_kMeans, accuracyByLabel_kMeans = \
        analyzeCluster(X_train_kmeansClustered, y_train)

    overallAccuracy_kMeansDF.loc[cutoff] = overallAccuracy_kMeans
```

График зависимости точности кластеризации от количества измерений приведен на рис. 5.4.

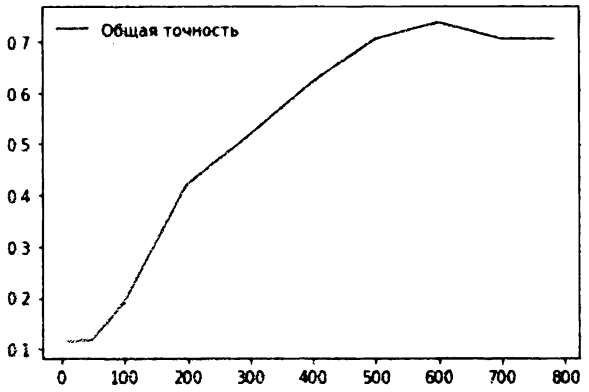


***Рис. 5.4.** Зависимость точности кластеризации по методу k-средних от количества исходных измерений*

Как следует из графика, точность кластеризации очень низкая при небольшом количестве измерений, но улучшается примерно до уровня 70%, как только количество измерений достигает 600.

В случае метода PCA точность кластеризации составляла примерно 70% даже при 10 измерениях, что свидетельствует об огромных перспективах снижения размерности в отношении захвата наиболее существенной информации из оригинального набора данных.

Иерархическая кластеризация

Перейдем к рассмотрению второго подхода к кластеризации, получившего название *иерархическая кластеризация*. В этом подходе не требуется предварительно задавать определенное количество кластеров. Вместо этого мы можем выбрать, сколько кластеров мы хотели бы иметь по завершении кластеризации.

Используя наблюдения, содержащиеся в наборе данных, алгоритм иерархической кластеризации построит *дендрограмму*, которая будет отображаться в виде перевернутого дерева с листьями внизу и стволом вверху.

Самые нижние листья — это индивидуальные примеры в наборе данных. По мере перемещения вверх по перевернутому дереву алгоритм иерархической кластеризации объединяет листья на основании их взаимного сходства. В первую очередь объединяются примеры (или группы примеров) с наибольшим взаимным сходством, затем — менее схожие экземпляры. В результате такого итеративного процесса все примеры в конечном счете объединяются, формируя единый ствол дерева.

Эта вертикальная картинка очень полезна. Как только алгоритм иерархической кластеризации закончит свою работу, мы сможем просмотреть дендрограмму и определить, где хотим усечь дерево, — чем ниже мы его обрежем, тем больше ветвей (а значит, и больше кластеров) в нем останется. Если мы хотим иметь меньше кластеров, дерево следует обрезать в более высоких позициях на дендрограмме, ближе к стволу, располагающемуся в самом верху перевернутого дерева.

Выбор позиции вертикальной обрезки аналогичен выбору количества кластеров, k, в алгоритме кластеризации методом k-средних.

Агломеративная иерархическая кластеризация

Версия иерархической кластеризации, которую мы собираемся исследовать, называется *агломеративная кластеризация* (agglomerative clustering). Несмотря на то что библиотека Scikit-learn содержит соответствующий модуль, он работает крайне медленно. Вместо него мы используем другой пакет иерархической кластеризации: *fastcluster*. Это библиотека на языке C++ с интерфейсом Python/SciPy[1].

Основная функция библиотеки, которая нам понадобится, — `fastcluster.linkage_vector`. Ей необходимо передать несколько аргументов, включая тренировочную матрицу *X*, *метод* и *метрику*. Аргумент `method` может иметь значения `single`, `centroid`, `median` или `ward`. Он задает схему кластеризации, которую следует использовать для определения расстояний между новым узлом в дендрограмме и остальными узлами. Аргумент `metric` в большинстве случаев должен быть равен `euclidean`. Более того, это единственный вариант для методов `centroid`, `median` и `ward`. Для получения более подробной информации об этих аргументах обратитесь к документации библиотеки *fastcluster*.

Приступим к настройке алгоритма иерархической кластеризации для наших данных. Как и прежде, мы будем тренировать алгоритм на первых ста главных компонентах из набора изображений MNIST, редуцированного с помощью метода PCA. Установим для аргумента `method` значение `ward` (оно проявило себя наилучшим образом в проведенных ранее экспериментах), а для аргумента `metric` — значение `euclidean`.

[1] Более подробную информацию об этом методе можно найти по адресу `https://ru.wikipedia.org/wiki/Иерархическая_кластеризация`.

Значение `ward` задает использование *метода Уорда*[2]. Это неплохой вариант по умолчанию для иерархической кластеризации, но на практике лучше поэкспериментировать на собственных наборах данных.

```
import fastcluster
from scipy.cluster.hierarchy import dendrogram, cophenet
from scipy.spatial.distance import pdist

cutoff = 100
Z = fastcluster.linkage_vector(X_train_PCA.loc[:, 0:cutoff], \
                               method='ward', metric='euclidean')
Z_dataFrame = pd.DataFrame(data=Z, columns=['clusterOne', \
                           'clusterTwo', 'distance', \
                           'newClusterSize'])
```

Алгоритм иерархической кластеризации возвращает матрицу `Z`. Он обрабатывает каждое из 50 000 наблюдений, входящих в наш набор рукописных цифр MNIST, как кластер, включающий только одну точку данных, и на каждой итерации объединяет два кластера, характеризующихся наименьшим расстоянием между ними.

Первоначально алгоритм объединяет лишь одноточечные кластеры, но впоследствии он начнет объединять многоточечные кластеры либо с одноточечными, либо с другими многоточечными. В ходе итеративного процесса все кластеры в конечном итоге объединятся, сформировав ствол перевернутого дерева (дендрограммы).

Дендрограмма

В табл. 5.1 представлена матрица `Z`, сгенерированная алгоритмом кластеризации. Она демонстрирует, на что способен данный алгоритм.

В первых двух столбцах таблицы, `clusterOne` и `clusterTwo`, указано, какие два кластера (ими могут быть как одноточечные кластеры, т.е. оригинальные наблюдения, так и многоточечные) объединяются с учетом взаимного расстояния между ними. В третьем столбце отображается само расстояние, `distance`, которое было определено по методу Уорда с использованием евклидовой метрики (`euclidean`), переданной алгоритму кластеризации.

Как нетрудно заметить, расстояние монотонно увеличивается. Другими словами, сначала объединяются кластеры, расстояния между которыми

[2] Более подробную информацию об этом методе можно найти по адресу `https://ru.wikipedia.org/wiki/Иерархическая_кластеризация`.

являются наименьшими, после чего алгоритм итеративно присоединяет к ним кластеры со следующими кратчайшими расстояниями между ними, и так до тех пор, пока все точки не будут объединены в единый кластер, расположенный в самом верху дендрограммы.

Таблица 5.1. Начальные строки матрицы иерархической кластеризации

	clusterOne	clusterTwo	distance	newClusterSize
0	42194.0	43025.0	0.562682	2.0
1	28350.0	37674.0	0.590866	2.0
2	26696.0	44705.0	0.621506	2.0
3	12634.0	32823.0	0.627762	2.0
4	24707.0	43151.0	0.637668	2.0
5	20465.0	24483.0	0.662557	2.0
6	466.0	42098.0	0.664189	2.0
7	46542.0	49961.0	0.665520	2.0
8	2301.0	5732.0	0.671215	2.0
9	37564.0	47668.0	0.675121	2.0
10	3375.0	26243.0	0.685797	2.0
11	15722.0	30368.0	0.686356	2.0
12	21247.0	21575.0	0.694412	2.0
13	14900.0	42486.0	0.696769	2.0
14	30100.0	41908.0	0.699261	2.0
15	12040.0	13254.0	0.701134	2.0
16	10508.0	25434.0	0.708872	2.0
17	30695.0	30757.0	0.710023	2.0
18	31019.0	31033.0	0.712052	2.0
19	36264.0	37285.0	0.713130	2.0

Сначала алгоритм объединяет одноточечные кластеры, формируя новые кластеры удвоенного размера, как это указано в четвертом столбце, `newClusterSize`. Однако по мере дальнейшего продвижения по дереву алгоритм начинает объединять большие многоточечные кластеры с другими большими многоточечными кластерами, как показано в табл. 5.2. На самой последней итерации (`49998`), два больших кластера объединяются, сливаясь в один кластер — расположенный в самом верху дерева ствол, который содержит все 50 000 оригинальных наблюдений.

Таблица 5.2. Последние строки матрицы иерархической кластеризации

	clusterOne	clusterTwo	distance	newClusterSize
49980	99965.0	99972.0	161.106998	5197.0
49981	99932.0	99980.0	172.070003	6505.0
49982	99945.0	99960.0	182.840860	3245.0
49983	99964.0	99976.0	184.475761	3683.0
49984	99974.0	99979.0	185.027847	7744.0
49985	99940.0	99975.0	185.345207	5596.0
49986	99957.0	99967.0	211.854714	5957.0
49987	99938.0	99983.0	215.494857	4846.0
49988	99978.0	99984.0	216.760365	11072.0
49989	99970.0	99973.0	217.355871	4899.0
49990	99969.0	99986.0	225.468298	8270.0
49991	99981.0	99982.0	238.845135	9750.0
49992	99968.0	99977.0	266.146782	5567.0
49993	99985.0	99989.0	270.929453	10495.0
49994	99990.0	99991.0	346.840948	18020.0
49995	99988.0	99993.0	394.365194	21567.0
49996	99987.0	99995.0	425.142387	26413.0
49997	99992.0	99994.0	440.148301	23587.0
49998	99996.0	99997.0	494.383855	50000.0

Возможно, вас несколько смущают значения, указанные в столбцах `clusterOne` и `clusterTwo` этой таблицы. Например, в последней строке — 49998 — кластер 99996 соединяется с кластером 99997. Но, как мы знаем, набор данных MNIST содержит всего 50 000 наблюдений.

Номера кластеров в столбцах `clusterOne` и `clusterTwo` относятся к оригинальным наблюдениям только для номеров от 0 до 49999. Номера свыше 49999 относятся к ранее кластеризованным точкам. Например, номер 50000 относится к вновь сформированному кластеру в строке 0, номер 50001 — к вновь сформированному кластеру в строке 1 и т.д.

В строке 49998 в столбце `clusterOne` номер 99996 относится к кластеру, сформированному в строке 49996, а номер 99997 в столбце `clusterTwo` — к кластеру, сформированному в строке 49997. Пользуясь этой формулой, вы сможете просмотреть всю таблицу, чтобы увидеть, как объединяются кластеры.

Оценка результатов кластеризации

Теперь, когда мы располагаем дендрограммой, нам предстоит определить, в каком месте ее следует обрезать, чтобы получить требуемое количество кластеров. Мы можем упростить сравнение результатов иерархической кластеризации и метода *k*-средних, сформировав ровно 20 кластеров. Далее мы используем метрику точности кластеризации (определенную в разделе "Метод *k*-средних"), с помощью которой можно будет судить о степени однородности кластеров.

Для создания требуемого количества кластеров необходимо импортировать модуль *fcluster* из библиотеки SciPy. Мы должны задать для дендрограммы *пороговое расстояние*, чтобы определить, какое количество кластеров мы собираемся оставить. Чем больше пороговое расстояние, тем меньше кластеров мы получим. Точки данных в пределах указанного расстояния будут принадлежать к одному и тому же кластеру. Задание большого порогового расстояния равносильно обрезке перевернутого дерева в расположенной высоко по вертикали точке. По мере продвижения вверх по дереву группирование охватывает все больше точек, что будет приводить к уменьшению количества остающихся кластеров.

Чтобы получить ровно 20 кластеров, мы должны поэкспериментировать с различными значениями порогового расстояния, в соответствии с которыми библиотека *fcluster* будет выполнять обрезку дерева. Каждое из 50 000 наблюдений, включенных в набор данных MNIST, получит метку кластера, и мы сохраним их в объекте `DataFrame` библиотеки Pandas.

```
from scipy.cluster.hierarchy import fcluster

distance_threshold = 160
clusters = fcluster(Z, distance_threshold, criterion='distance')
X_train_hierClustered = pd.DataFrame(data=clusters, \
    index=X_train_PCA.index, columns=['cluster'])
```

Проверим, получаем ли мы ровно 20 различных кластеров при заданном выборе порогового расстояния.

```
print("Количество различных кластеров: ", \
      len(X_train_hierClustered['cluster'].unique()))
```

Как и следовало ожидать, мы действительно получили соответствующее подтверждение.

```
Количество различных кластеров: 20
```

А теперь вычислим результаты.

```
countByCluster_hierClust, countByLabel_hierClust, \
    countMostFreq_hierClust, accuracyDF_hierClust, \
    overallAccuracy_hierClust, accuracyByLabel_hierClust = \
        analyzeCluster(X_train_hierClustered, y_train)

print("Общая точность иерархической кластеризации: ", \
      overallAccuracy_hierClust)
```

Мы видим, что общая точность составляет примерно 77%, что превышает значение 70%, достигнутое с помощью метода *k*-средних:

```
Общая точность иерархической кластеризации: 0.76882
```

Заодно оценим точность для каждого кластера. Как следует из приведенных ниже результатов, точность существенно варьируется. Для одних кластеров она очень высокая, ближе к 100%, в то время как для других она снижается до скромных 50%.

```
0       0.987962
1       0.983727
2       0.988998
3       0.597356
4       0.678642
5       0.442478
6       0.950033
7       0.829060
8       0.976062
9       0.986141
10      0.990183
11      0.992183
12      0.971033
13      0.554273
14      0.553617
15      0.720183
16      0.538891
17      0.484590
18      0.957732
19      0.977310
dtype:  float64
```

В целом иерархическая кластеризация хорошо справилась с набором рукописных цифр MNIST. Не забывайте о том, что мы не использовали никаких меток!

На практике это должно работать следующим образом: сначала мы применяем алгоритм снижения размерности (например, PCA), затем выполняем кластеризацию (например, иерархическую) и, наконец, вручную размечаем несколько точек для каждого кластера. Например, если бы в случае набора MNIST отсутствовали метки, то мы просмотрели бы несколько изображений и пометили их в соответствии с представляемыми ими цифрами. При условии, что кластеры достаточно однородны, несколько вручную сгенерированных меток могли бы автоматически назначаться всем остальным изображениям в данном кластере.

Вот так, совсем неожиданно, не прилагая особых усилий, нам удалось пометить все 50 000 изображений в нашем наборе с почти 77%-ной точностью. Этот впечатляющий результат демонстрирует эффективность обучения без учителя.

DBSCAN

Перейдем к рассмотрению третьего (и последнего) из основных алгоритмов кластеризации: DBSCAN (density-based spatial clustering of applications with noise — основанная на плотности пространственная кластеризация для приложений с шумами).

Алгоритм DBSCAN группирует близко расположенные точки, где близость определяется как минимальное количество точек, которые должны существовать в пределах определенного расстояния. Если точка находится в пределах указанного расстояния от нескольких кластеров, то она будет группироваться с ближайшим к ней кластером. Любой экземпляр, не находящийся в пределах данного расстояния от другого кластера, помечается как выброс.

В случае использования метода k-средних и иерархической кластеризации выбросы плохо обрабатывались, и все точки приходилось кластеризовать. В случае алгоритма DBSCAN мы можем явно помечать точки как выбросы, избегая их кластеризации, что очень удобно. По сравнению с другими алгоритмами кластеризации алгоритм DBSCAN намного меньше подвержен искажениям, которые обычно вызываются наличием выбросов. Кроме того, как и в случае иерархической кластеризации, но в отличие от метода k-средних, нам не нужно предварительно задавать количество используемых кластеров.

Алгоритм DBSCAN

Сначала мы используем модуль DBSCAN, входящий в состав библиотеки Scikit-learn. Мы должны задать максимальное расстояние (гиперпараметр eps) между двумя точками, при котором они еще могут считаться соседними, и *минимальное количество образцов* (гиперпараметр min_samples), позволяющее назвать группу кластером. По умолчанию значение гиперпараметра eps равно 0.5, а гиперпараметра min_samples — 5. Если для гиперпараметра eps установлено слишком малое значение, то никакие точки не будут считаться расположенными достаточно близко к другим точкам, чтобы их можно было считать соседями. Следовательно, все точки останутся некластеризированными. В случае же слишком больших значений гиперпараметра eps многие точки могут быть включены в кластеры, и лишь немногие точки останутся некластеризованными и в конечном счете будут помечены как выбросы.

Мы должны выполнить поиск оптимального значения eps для нашего набора изображений MNIST. Гиперпараметр min_samples задает минимальное количество точек, которые должны находиться в пределах расстояния eps, чтобы они могли считаться кластером. Как только набирается min_samples тесно расположенных точек, все остальные точки, находящиеся в пределах расстояния eps от любой из этих *центровых* точек, считаются частью кластера, даже если вокруг них нет min_samples точек в пределах указанного расстояния eps. Точки последнего типа называются *граничными точками кластера*.

В общем случае количество кластеров уменьшается по мере увеличения гиперпараметра min_samples. Как и в случае гиперпараметра eps, мы должны выполнить поиск оптимального значения min_samples для нашего набора рукописных цифр MNIST. В кластерах имеются центровые и граничные точки, но во всех остальных отношениях они принадлежат к одной и той же группе. Все точки, которые остались не сгруппированными, будь то центровые или граничные точки кластера, помечаются как выбросы.

Применение алгоритма DBSCAN к нашему набору данных

Перейдем к рассмотрению нашей задачи. Как и прежде, применим алгоритм DBSCAN к первым ста главным компонентам набора данных MNIST, редуцированного по методу PCA.

```
from sklearn.cluster import DBSCAN

eps = 3
```

```
min_samples = 5
leaf_size = 30
n_jobs = 4

db = DBSCAN(eps=eps, min_samples=min_samples, leaf_size=leaf_size, \
            n_jobs=n_jobs)

cutoff = 99
X_train_PCA_dbscanClustered = \
    db.fit_predict(X_train_PCA.loc[:, 0:cutoff])
X_train_PCA_dbscanClustered = \
    pd.DataFrame(data=X_train_PCA_dbscanClustered, \
                 index=X_train.index, columns=['cluster'])

countByCluster_dbscan, countByLabel_dbscan, \
    countMostFreq_dbscan, accuracyDF_dbscan, \
    overallAccuracy_dbscan, accuracyByLabel_dbscan = \
    analyzeCluster(X_train_PCA_dbscanClustered, y_train)

overallAccuracy_dbscan
```

Мы оставим для гиперпараметра `min_samples` заданное по умолчанию значение 5, но используем для гиперпараметра `eps` значение 3, чтобы избежать кластеризации слишком малого количества точек.

Для общей точности мы получаем следующее значение:

```
Общая точность алгоритма DBSCAN: 0.242
```

Это очень низкое значение по сравнению с теми, которые были получены с помощью метода *k*-средних и иерархической кластеризации. Мы могли бы поэкспериментировать с подбором гиперпараметров `eps` и `min_samples`, чтобы попытаться улучшить этот результат, но, по-видимому, алгоритм DBSCAN плохо приспособлен для кластеризации наблюдений, входящих в данный конкретный набор.

Чтобы выяснить, почему так происходит, рассмотрим данные кластеров (табл. 5.3).

Большинство точек остаются некластеризованными. Из 50 000 наблюдений тренировочного набора 39 575 отнесены к кластеру `-1`, а это означает, что они не принадлежат ни к одному из кластеров. Они помечены как выбросы — другими словами, как шум. 8885 точек принадлежат к кластеру `0`. Кроме того, имеется длинный "хвост", образованный кластерами небольшого размера. По-видимому, алгоритму DBSCAN нелегко находить отчетливо плотные

группы точек, поэтому он плохо справляется с кластеризацией изображений MNIST на основании цифр, которые они представляют.

Таблица 5.3. Результаты кластеризации с использованием алгоритма DBSCAN

	cluster	clusterCount
0	–1	39575
1	0	8885
2	8	720
3	5	92
4	18	51
5	38	38
6	41	22
7	39	22
8	4	16
9	20	16

Алгоритм HDBSCAN

Испытаем другую версию алгоритма DBSCAN — *HDBSCAN*, или *иерархический DBSCAN*, — и проверим, позволит ли это улучшить результаты. Отправной точкой для данного алгоритма служит уже знакомый нам алгоритм DBSCAN, который преобразуется в алгоритм иерархической кластеризации. Другими словами, алгоритм осуществляет группирование в соответствии с плотностью точек, а затем итеративно связывает кластеры на основании расстояний между ними, как это делает алгоритм иерархической кластеризации, с которым мы познакомились в одном из предыдущих разделов.

Два основных гиперпараметра этого алгоритма — `min_cluster_size` и `min_samples`, причем последний, если задать для него значение `None`, автоматически получает значение `min_cluster_size`. Используем изначально установленные значения параметров и откалибруем их, если HDBSCAN сработает в отношении набора рукописных цифр MNIST лучше, чем DBSCAN.

```
import hdbscan

min_cluster_size = 30
min_samples = None
alpha = 1.0
```

```
cluster_selection_method = 'eom'

hdb = hdbscan.HDBSCAN(min_cluster_size=min_cluster_size, \
    min_samples=min_samples, alpha=alpha, \
    cluster_selection_method=cluster_selection_method)

cutoff = 10
X_train_PCA_hdbscanClustered = \
    hdb.fit_predict(X_train_PCA.loc[:, 0:cutoff])

X_train_PCA_hdbscanClustered = \
    pd.DataFrame(data=X_train_PCA_hdbscanClustered, \
                 index=X_train.index, columns=['cluster'])

countByCluster_hdbscan, countByLabel_hdbscan, \
    countMostFreq_hdbscan, accuracyDF_hdbscan, \
    overallAccuracy_hdbscan, accuracyByLabel_hdbscan = \
    analyzeCluster(X_train_PCA_hdbscanClustered, y_train)
```

Для общей точности мы получаем следующее значение:

```
Общая точность алгоритма HDBSCAN: 0.24696
```

В нашем случае 25% — это лишь незначительное улучшение точности по сравнению с алгоритмом DBSCAN и слишком далеко от более чем 70%-ной точности, достигнутой с помощью метода *k*-средних и иерархической кластеризации. Значения точности для различных кластеров приведены в табл. 5.4.

***Таблица 5.4.** Результаты кластеризации с использованием алгоритма HDBSCAN*

	cluster	clusterCount
0	−1	42570
1	4	5140
2	7	942
3	0	605
4	6	295
5	3	252
6	1	119
7	5	45
8	2	32

Здесь наблюдается та же картина, что и в случае алгоритма DBSCAN. Большинство точек остаются некластеризованными, а кроме того, имеется длинный "хвост", образованный кластерами небольшого размера. Таким образом, говорить о существенном улучшении результатов не приходится.

Резюме

В этой главе мы рассмотрели три основных типа алгоритмов кластеризации — метод *k*-средних, иерархическая кластеризация и DBSCAN — и применили их к набору рукописных цифр MNIST пониженной размерности. Первые два алгоритма продемонстрировали очень высокую эффективность на наборе данных, сгруппировав изображения достаточно хорошо для того, чтобы получить более чем 70%-ную согласованность меток в кластерах. Алгоритм DBSCAN проявил себя значительно хуже для этого набора, но все равно представляет собой работоспособный алгоритм кластеризации.

Теперь, когда мы познакомились с алгоритмами кластеризации, применим их в главе 6 для построения приложения на основе обучения без учителя.

ГЛАВА 6

Сегментирование групп

В главе 5 мы рассмотрели кластеризацию — подход, базирующийся на обучении без учителя, который позволяет определять базовую структуру данных и группировать точки на основании их сходства. Формируемые группы (называемые кластерами) однородны и четко различимы. Иначе говоря, элементы, принадлежащие к одной группе, в чем-то очень схожи между собой и резко отличаются от элементов других групп.

С практической точки зрения возможность сегментирования элементов по группам на основании их сходства, причем без привлечения каких-либо меток, открывает перед нами широкие перспективы. Такую методику можно, к примеру, использовать в электронной коммерции для выявления различных групп потребителей с последующим формированием маркетинговых стратегий (например, бюджетные покупатели, метросексуалы, технари, аудиофилы и т.п.). Сегментирование групп способно повысить эффективность целевой рекламы и улучшить качество рекомендаций в рекомендательных системах фильмов, музыки, новостей, социальных сетей, сайтов знакомств и пр.

В этой главе мы построим приложение на основе обучения без учителя, используя алгоритмы кластеризации, рассмотренные в предыдущей главе, и, в частности, выполним сегментирование групп.

Данные кредитной компании LendingClub

В этой главе мы используем данные о займах, предоставляемых американской кредитной компанией LendingClub. Заемщики могут брать кредиты в виде беззалоговых персональных займов на сумму от 1000 до 40 000 долларов и на срок от трех до пяти лет.

Инвесторы могут просматривать кредитные заявки и принимать решения на основании анализа кредитной истории заемщика, а также суммы, категории и цели получения ссуды. Доходом инвесторов становятся проценты по займу, тогда как сама компания LendingClub зарабатывает на комиссиях за оформление займа и предоставление услуг.

База данных, с которой мы будем работать, охватывает транзакции за 2007–2011 гг. и доступна на сайте LendingClub (`https://www.lendingclub.com/info/statistics.action`) для зарегистрированных пользователей. Там же доступен и словарь данных.

Подготовка данных

Аналогично тому, как мы поступали в предыдущих главах, подготовим среду для работы с набором данных LendingClub.

Загрузка библиотек

Прежде всего, загрузим необходимые библиотеки.

```
# Импорт библиотек
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn import impute as imp
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score

'''Алгоритмы'''
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
import fastcluster
from scipy.cluster.hierarchy import dendrogram, cophenet, fcluster
from scipy.spatial.distance import pdist
```

Исследование данных

Следующий шаг — загрузка данных о займах и определение столбцов, с которыми мы будем работать.

Оригинальный файл данных содержит 144 столбца, но большинство из них пустые или не представляют никакого интереса. Поэтому мы выделим подмножество столбцов, которые по большей части заполнены данными и заслуживают того, чтобы использовать их в нашем приложении. Сюда входят атрибуты займа, такие как запрошенный объем, предоставленный объем, срок займа, процентная ставка, категория займа и т.п., а также атрибуты заемщика, например его трудовой стаж, статус домовладельца, годовой доход, адрес и цель получения займа.

```
# Загрузка данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'lending_club_data', \
                         'LoanStats3a.csv'])
data = pd.read_csv(current_path + file)

# Выбор оставляемых столбцов
columnsToKeep = \
    ['loan_amnt', 'funded_amnt', 'funded_amnt_inv', 'term', \
     'int_rate', 'installment', 'grade', 'sub_grade', 'emp_length', \
     'home_ownership', 'annual_inc', 'verification_status', \
     'pymnt_plan', 'purpose', 'addr_state', 'dti', 'delinq_2yrs', \
     'earliest_cr_line', 'mths_since_last_delinq', \
     'mths_since_last_record', 'open_acc', 'pub_rec', 'revol_bal', \
     'revol_util', 'total_acc', 'initial_list_status', 'out_prncp', \
     'out_prncp_inv', 'total_pymnt', 'total_pymnt_inv', \
     'total_rec_prncp', 'total_rec_int', 'total_rec_late_fee', \
     'recoveries', 'collection_recovery_fee', 'last_pymnt_d', \
     'last_pymnt_amnt']

data = data.loc[:, columnsToKeep]

data.shape

data.head()
```

Данные охватывают 42 542 займа и 37 признаков. Фрагмент данных приведен в табл. 6.1.

Таблица 6.1. Начальный фрагмент базы данных о займах

	loan_amnt	funded_amnt	funded_amnt_Inv	term	Int_rate	Instsallment	grade
0	5000.0	5000.0	4975.0	36 months	10.65%	162.87	B
1	2500.0	2500.0	2500.0	60 months	15.27%	59.83	C
2	2400.0	2400.0	2400.0	35 months	15.96%	84.33	C
3	10000.0	10000.0	10000.0	36 months	13.49%	339.31	C
4	3000.0	3000.0	3000.0	60 months	12.69%	67.79	B

Преобразование строкового формата в числовой

Некоторые признаки, такие как срок займа, процентная ставка и трудовой стаж заемщика, требуют преобразования из текстового формата в числовой.

```
# Преобразование признаков из строкового формата в числовой
for i in ["term", "int_rate", "emp_length", "revol_util"]:
    data.loc[:,i] = \
        data.loc[:, i].apply(lambda x: re.sub("[^0-9]", "", str(x)))
    data.loc[:,i] = pd.to_numeric(data.loc[:,i])
```

В нашем приложении мы будем учитывать только числовые признаки, игнорируя все категориальные, поскольку нечисловые признаки не могут обрабатываться нашими алгоритмами кластеризации в их нынешнем виде.

Замена отсутствующих значений

Необходимо найти все числовые признаки и подсчитать количество значений NaN для каждого признака. Вместо пустых значений мы подставим либо среднее значение признака, либо (в некоторых случаях) нулевое значение, в зависимости от того, что оправданнее с точки зрения бизнеса.

```
# Определение того, какие признаки являются числовыми
numericalFeats = [x for x in data.columns \
                  if data[x].dtype != 'object']

# Отображение количества значений NaN для каждого признака
nanCounter = np.isnan(data.loc[:, numericalFeats]).sum()nanCounter
```

Ниже приведена сводка по признакам.

```
loan_amnt                 7
funded_amnt               7
funded_amnt_inv           7
```

```
term                        7
int_rate                    7
installment                 7
emp_length                  1119
annual_inc                  11
dti                         7
delinq_2yrs                 36
mths_since_last_delinq      26933
mths_since_last_record      38891
open_acc                    36
pub_rec                     36
revol_bal                   7
revol_util                  97
total_acc                   36
out_prncp                   7
out_prncp_inv               7
total_pymnt                 7
total_pymnt_inv             7
total_rec_prncp             7
total_rec_int               7
total_rec_late_fee          7
recoveries                  7
collection_recovery_fee     7
last_pymnt_amnt             7
dtype: int64
```

По большинству признаков количество значений NaN невелико, но есть и исключения, например количество месяцев, прошедших с даты начала просрочки или последнего изменения статуса.

Мы заменим все отсутствующие значения, чтобы не иметь с ними дела в процессе кластеризации.

```
# Замена значений NaN средним значением признака
fillWithMean = ['loan_amnt', 'funded_amnt', 'funded_amnt_inv', \
     'term', 'int_rate', 'installment', 'emp_length', \
     'annual_inc', 'dti', 'open_acc', 'revol_bal', 'revol_util', \
     'total_acc', 'out_prncp', 'out_prncp_inv', 'total_pymnt', \
     'total_pymnt_inv', \'total_rec_prncp', 'total_rec_int', \
     'last_pymnt_amnt']

# Замена значений NaN нулями
fillWithZero = ['delinq_2yrs', 'mths_since_last_delinq', \
     'mths_since_last_record', 'pub_rec', 'total_rec_late_fee', \
```

```
    'recoveries', 'collection_recovery_fee']

# Выполнение подстановки
im = imp.SimpleImputer(strategy='mean')

data.loc[:, fillWithMean] = im.fit_transform(data[fillWithMean])
data.loc[:, fillWithZero] = data.loc[:, \
    fillWithZero].fillna(value=0, axis=1)
```

Пересчитаем количество значений NaN и убедимся в том, что мы полностью избавились от них.

```
nanCounter = np.isnan(data.loc[:,numericalFeats]).sum()
nanCounter
```

```
loan_amnt                 0
funded_amnt               0
funded_amnt_inv           0
term                      0
int_rate                  0
installment               0
emp_length                0
annual_inc                0
dti                       0
delinq_2yrs               0
mths_since_last_delinq    0
mths_since_last_record    0
open_acc                  0
pub_rec                   0
revol_bal                 0
revol_util                0
total_acc                 0
out_prncp                 0
out_prncp_inv             0
total_pymnt               0
total_pymnt_inv           0
total_rec_prncp           0
total_rec_int             0
total_rec_late_fee        0
recoveries                0
collection_recovery_fee   0
last_pymnt_amnt           0
dtype: int64
```

Конструирование признаков

Сконструируем ряд признаков в дополнение к уже существующим. Эти новые признаки в основном представляют собой соотношения между размером займа, доступным балансом, размерами платежей и годовым доходом заемщика.

```
# Конструирование признаков
data['installmentOverLoanAmnt'] = data.installment/data.loan_amnt
data['loanAmntOverIncome'] = data.loan_amnt/data.annual_inc
data['revol_balOverIncome'] = data.revol_bal/data.annual_inc
data['totalPymntOverIncome'] = data.total_pymnt/data.annual_inc
data['totalPymntInvOverIncome'] = data.total_pymnt_inv/data.annual_inc
data['totalRecPrncpOverIncome'] = data.total_rec_prncp/data.annual_inc
data['totalRecIncOverIncome'] = data.total_rec_int/data.annual_inc

newFeats = ['installmentOverLoanAmnt', 'loanAmntOverIncome', \
            'revol_balOverIncome', 'totalPymntOverIncome', \
            'totalPymntInvOverIncome', 'totalRecPrncpOverIncome', \
            'totalRecIncOverIncome']
```

Выбор окончательного набора признаков и масштабирование

Далее мы генерируем кадр данных для обучения и масштабируем признаки для алгоритма кластеризации.

```
# Выбор признаков для обучения
numericalPlusNewFeats = numericalFeats+newFeats
X_train = data.loc[:, numericalPlusNewFeats]

# Масштабирование данных
sX = pp.StandardScaler()
X_train.loc[:, :] = sX.fit_transform(X_train)
```

Назначение меток для оценки

Кластеризация основана на обучении без учителя, поэтому метки не задействуются. В то же время, для того чтобы можно было оценить эффективность нашего алгоритма кластеризации в плане формирования отчетливо

различающихся однородных групп заемщиков, мы будем использовать категорию займа в качестве заменителя метки.

Категории займов имеют буквенную градацию. Займы категории "A" считаются наиболее выгодными, займы категории "G" — наименее выгодными.

```
labels = data.grade
labels.unique()

array(['B', 'C', 'A', 'E', 'F', 'D', 'G', nan], dtype=object)
```

Среди градаций имеются значения NaN. Мы заполним эти поля значением `"Z"` и используем фупкцию `LabelEncoder` из библиотеки Scikit-learn, чтобы преобразовать буквенные градации в числовые. Для согласованности мы загрузим метки в массив `y_train`.

```
# Заполнение отсутствующих меток
labels = labels.fillna(value="Z")

# Преобразование меток в числовые значения
lbl = pp.LabelEncoder()
lbl.fit(list(labels.values))
labels = pd.Series(data=lbl.transform(labels.values), name="grade")

# Сохранение числовых меток в массиве y_train
y_train = labels

labelsOriginalVSNew = pd.concat([labels, data.grade], axis=1)
labelsOriginalVSNew
```

Как следует из табл. 6.2, все градации "A" были преобразованы в 0, градации "B" — в 1 и т.д.

Проверим, имеют ли займы категории "A" наименьшую процентную ставку, поскольку с ними связаны наименьшие риски, тогда как за остальные займы берется более высокий процент.

```
# Сопоставление градаций займов с процентными ставками
interestAndGrade = pd.DataFrame(data=[data.int_rate, labels])
interestAndGrade = interestAndGrade.T

interestAndGrade.groupby("grade").mean()
```

Таблица 6.2. Соответствие между числовыми и буквенными обозначениями градаций займов

	grade	grade
0	1	B
1	2	C
2	2	C
3	2	C
4	1	B
5	0	A
6	2	C
7	4	E
8	5	F
9	1	B
10	2	C
11	1	B
12	2	C
13	1	B
14	1	B
15	3	D
16	2	C

Данные, приведенные в табл. 6.3, подтверждают наше предположение. Чем выше градация займа, тем выше процентная ставка[1].

Таблица 6.3. Сравнение градаций и процентных ставок

grade	int_rate
0.0	734.270844
1.0	1101.420857
2.0	1349.988902
3.0	1557.714927
4.0	1737.676783
5.0	1926.530361
6.0	2045.125000
7.0	1216.501563

[1] Категорию "7" можно игнорировать, так как ей соответствует категория займа "Z". Она была искусственно назначена займам, для которых отсутствует категория.

Пригодность кластеров

Итак, данные подготовлены к работе. Мы имеем массив `X_train`, содержащий все 34 числовых признака, и массив `y_train`, содержащий числовые категории займов, которые мы используем лишь для валидации результатов, а не для обучения с помощью алгоритма, как это положено делать в случае задач, решаемых по технологии обучения без учителя. Прежде чем приступить к созданию нашего первого кластерного приложения, напишем функцию, с помощью которой будем анализировать пригодность кластеров, генерируемых алгоритмами кластеризации. Для оценки пригодности каждого кластера мы используем понятие однородности.

Если алгоритм кластеризации хорошо справляется с разделением заемщиков в наборе данных LendingClub, то каждый кластер должен включать заемщиков, весьма сходных между собой и непохожих на заемщиков из других групп. Предположительно, схожие заемщики одной группы должны иметь аналогичные кредитные профили — другими словами, их кредитоспособность должна быть близкой.

Если это действительно так (а на практике многие из подобных предположений справедливы лишь частично), то, как правило, заемщикам, относящимся к одному кластеру, будет присвоена одна и та же числовая категория займа, которую мы будем использовать для валидации с помощью числовых категорий займов, хранящихся в массиве `y_train`. Чем выше в каждом кластере процентная доля заемщиков с наиболее часто встречающимися числовыми категориями, тем лучше работает приложение кластеризации.

В качестве примера рассмотрим кластер с сотней заемщиков. Если 30 из них имеют числовую категорию 0, 25 — категорию 1, а 20 — категорию 2, в то время как категории других заемщиков изменяются в пределах от 3 до 7, то мы сказали бы, что кластер характеризуется 30%-ной точностью, при условии, что наиболее часто встречающаяся в данном кластере категория относится лишь к 30% заемщиков, входящих в этот кластер.

Если бы мы не располагали массивом `y_train`, содержащим числовые категории займов для валидации пригодности кластеров, то можно было бы предложить альтернативный подход: выбрать нескольких заемщиков в каждом кластере, определить для них категории вручную и выяснить, удалось ли нам присвоить в грубом приближении одну и ту же числовую категорию этим заемщикам. Если нам это действительно удалось, то кластер следует считать вполне пригодным — он достаточно однороден для того, чтобы мы могли присвоить одну и ту же категорию всем заемщикам в данной выборке.

В противном случае кластер не является достаточно пригодным — заемщики слишком неоднородны, и мы должны попытаться улучшить решение, используя больше данных, другой алгоритм кластеризации и т.п.

С учетом того, что мы уже располагаем числовыми категориями заемщиков, нам не требуется создавать выборки и вручную присваивать категории, но важно не забывать о такой возможности на случай отсутствия меток.

Код функции, анализирующей кластеры, приведен ниже.

```
def analyzeCluster(clusterDF, labelsDF):
    countByCluster = \
        pd.DataFrame(data=clusterDF['cluster'].value_counts())
    countByCluster.reset_index(inplace=True, drop=False)
    countByCluster.columns = ['cluster', 'clusterCount']

    preds = pd.concat([labelsDF, clusterDF], axis=1)
    preds.columns = ['trueLabel', 'cluster']

    countByLabel = \
        pd.DataFrame(data=preds.groupby('trueLabel').count())

    countMostFreq = \
        pd.DataFrame(data=preds.groupby('cluster').agg(lambda \
            x:x.value_counts().iloc[0]))
        countMostFreq.reset_index(inplace=True, drop=False)
        countMostFreq.columns = ['cluster', 'countMostFrequent']

    accuracyDF = countMostFreq.merge(countByCluster, \
        left_on="cluster", right_on="cluster")

    overallAccuracy = accuracyDF.countMostFrequent.sum() / \
        accuracyDF.clusterCount.sum()
    accuracyByLabel = accuracyDF.countMostFrequent / \
        accuracyDF.clusterCount

    return countByCluster, countByLabel, countMostFreq, accuracyDF, \
        overallAccuracy, accuracyByLabel
```

Применение метода *k*-средних

В нашем первом приложении кластеризации, применяемом к набору данных LendingClub, мы используем метод *k*-средних, введенный в главе 5. Вспомните, что в методе *k*-средних требуется указать нужное количества кластеров, *k*, и алгоритм будет относить каждого заемщика исключительно к одному из этих *k* кластеров.

Алгоритм будет достигать этого, минимизируя внутрикластерную вариацию, называемую *инерцией*, так чтобы сумма внутрикластерных вариаций по всем *k* кластерам была наименьшей.

Вместо того чтобы указать только одно значение *k*, мы выполним эксперимент, в котором для *k* устанавливаются значения в диапазоне 10–30, и отложим в виде графика значения точности в соответствии с тем, как мы ее определили в предыдущем разделе.

Базируясь на значении *k*, обеспечивающем наилучшую точность, мы можем построить конвейер для кластеризации с использованием этого наилучшего значения *k*.

```
from sklearn.cluster import KMeans

n_clusters = 10
n_init = 10
max_iter = 300
tol = 0.0001
random_state = 2018

kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, \
                max_iter=max_iter, tol=tol, \
                random_state=random_state)

kMeans_inertia = pd.DataFrame(data=[], index=range(10, 31), \
                              columns=['Инерция'])

overallAccuracy_kMeansDF = pd.DataFrame(data=[], \
                                        index=range(10, 31), \
                                        columns=['Общая точность'])

for n_clusters in range(10,31):
    kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, \
                    max_iter=max_iter, tol=tol, \
                    random_state=random_state, n_jobs=n_jobs)
```

```
    kmeans.fit(X_train)
    kMeans_inertia.loc[n_clusters] = kmeans.inertia_
    X_train_kmeansClustered = kmeans.predict(X_train)
    X_train_kmeansClustered = \
        pd.DataFrame(data=X_train_kmeansClustered, \
            index=X_train.index, columns=['cluster'])

    countByCluster_kMeans, countByLabel_kMeans, \
        countMostFreq_kMeans, accuracyDF_kMeans, \
        overallAccuracy_kMeans, accuracyByLabel_kMeans = \
        analyzeCluster(X_train_kmeansClustered, y_train)

    overallAccuracy_kMeansDF.loc[n_clusters] = overallAccuracy_kMeans

overallAccuracy_kMeansDF.plot()
```

Результаты представлены в графическом виде на рис. 6.1.

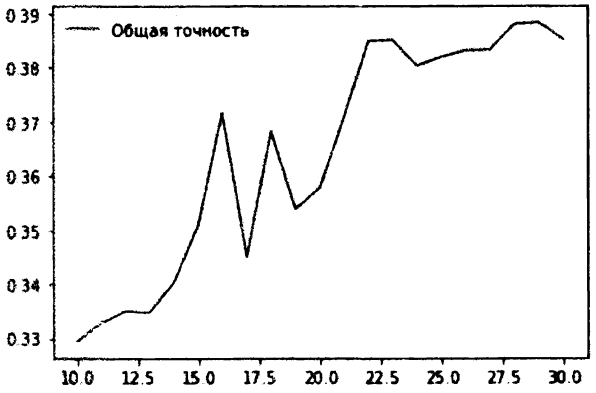


Рис. 6.1. *Общая точность для различных значений k при использовании метода k-средних*

Как видим, наилучшая точность достигается при использовании примерно 30 кластеров, выходя на уровень 39%. Другими словами, для любого заданного кластера наиболее часто встречающаяся метка применяется приблизительно к 39% заемщиков. Оставшиеся 61% заемщиков имеют метки, которые не относятся к числу встречающихся наиболее часто.

Ниже показана точность для каждого кластера при $k = 30$.

```
0       0.326633
1       0.258993
2       0.292240
```

```
3       0.234242
4       0.388794
5       0.325654
6       0.303797
7       0.762116
8       0.222222
9       0.391381
10      0.292910
11      0.317533
12      0.206897
13      0.312709
14      0.345233
15      0.682208
16      0.327250
17      0.366605
18      0.234783
19      0.288757
20      0.500000
21      0.375466
22      0.332203
23      0.252252
24      0.338509
25      0.232000
26      0.464418
27      0.261583
28      0.376327
29      0.269129
dtype: float64
```

От кластера к кластеру точность испытывает довольно заметные колебания. Одни кластеры намного однороднее других. Например, точность для кластера 7 составляет 76%, а для кластера 12 — всего лишь 21%. Это стартовая точка для создания приложения кластеризации, автоматически распределяющего новых заемщиков в уже существующую группу на основании их сходства с другими заемщиками. Базируясь на данном варианте кластеризации, можно автоматически присваивать пробные числовые категории займов новым заемщикам, и эти категории будут корректными приблизительно в 39% случаев.

Описанное решение не является наилучшим, и мы должны проанализировать, можно ли улучшить результаты за счет получения дополнительных данных, конструирования большего количества признаков, настройки различных гиперпараметров алгоритма *k*-средних или использования другого

алгоритма кластеризации. Вполне возможно, что мы не располагаем достаточным количеством данных, обеспечивающим более эффективное разделение заемщиков на отчетливо различимые однородные группы по сравнению с имеющимся. Если это действительно так, то нам потребуется большее данных, а также сконструированных и отобранных признаков. Или же может случиться так, что для имеющегося ограниченного набора данных алгоритм k-средних не является оптимальным решением для такого разделения.

Попробуем применить иерархическую кластеризацию и проверим, поможет ли это улучшить результаты.

Применение иерархической кластеризации

Вспомните, что в случае использования иерархической кластеризации мы не должны предварительно привязываться к определенному количеству кластеров. Вместо этого мы указываем, какое количество кластеров желательно иметь по завершении процесса кластеризации. Алгоритм иерархической кластеризации построит дендрограмму, которую можно представить в виде перевернутого дерева. Листья в самом низу — это индивидуальные заемщики LendingClub, подавшие заявки на получение кредитов.

Иерархическая кластеризация объединяет заемщиков по мере продвижения вверх по перевернутому дереву, руководствуясь степенью их взаимного сходства. В первую очередь объединяются заемщики с наибольшим сходством, тогда как менее схожие заемщики объединяются значительно позже. В конечном счете все заемщики объединяются в виде ствола в самом верху перевернутого дерева.

С маркетинговой точки зрения такой процесс кластеризации открывает широкие возможности. Если нам удастся найти схожих заемщиков и сгруппировать их, то мы сможем более эффективно присвоить им рейтинги кредитоспособности. Мы также сможем сформировать разные стратегии для разных групп заемщиков и тем самым повысить качество предоставления услуг.

Когда кластеризация завершается, мы должны определить, где следует обрезать дерево. Чем ниже линия обреза, тем больше групп заемщиков останется.

Для начала обучим алгоритм кластеризации, как мы это делали в главе 5.

```
import fastcluster
from scipy.cluster.hierarchy import dendrogram
from scipy.cluster.hierarchy import cophenet
from scipy.spatial.distance import pdist
```

```
Z = fastcluster.linkage_vector(X_train, method='ward', \
                               metric='euclidean')

Z_dataFrame = pd.DataFrame(data=Z, columns=['clusterOne', \
                           'clusterTwo', 'distance', \
                           'newClusterSize'])
```

В табл. 6.4 показано, как выглядит кадр данных. Первые 20 строк представляют заемщиков, которым соответствуют листья дерева в самом низу.

Таблица 6.4. *Самые нижние листья дерева иерархической кластеризации*

	clusterOne	clusterTwo	distance	newClusterSize
0	39786.0	39787.0	0.000000e+00	2.0
1	39788.0	42542.0	0.000000e+00	3.0
2	42538.0	42539.0	0.000000e+00	2.0
3	42540.0	42544.0	0.000000e+00	3.0
4	42541.0	42545.0	3.399350e-17	4.0
5	42543.0	42546.0	5.139334e-17	7.0
6	33251.0	33261.0	1.561313e-01	2.0
7	42512.0	42535.0	3.342654e-01	2.0
8	42219.0	42316.0	3.368231e-01	2.0
9	6112.0	21928.0	3.384368e-01	2.0
10	33248.0	33275.0	3.583819e-01	2.0
11	33253.0	33265.0	3.595331e-01	2.0
12	33258.0	42552.0	3.719377e-01	3.0
13	20430.0	23299.0	3.757307e-01	2.0
14	5455.0	32845.0	3.828709e-01	2.0
15	28615.0	30306.0	3.900294e-01	2.0
16	9056.0	9769.0	3.967378e-01	2.0
17	11162.0	13857.0	3.991124e-01	2.0
18	33270.0	42548.0	3.995620e-01	3.0
19	17422.0	17986.0	4.061704e-01	2.0

Вспомните, что последние строки представляют верхушку перевернутого дерева, и в конечном счете должны быть объединены все 42 541 заемщиков (табл. 6.5).

Таблица 6.5. Самые верхние листья дерева иерархической кластеризации

	clusterOne	clusterTwo	distance	newClusterSize
42521	85038.0	85043.0	132.715723	3969.0
42522	85051.0	85052.0	141.386569	2899.0
42523	85026.0	85027.0	146.976703	2351.0
42524	85048.0	85049.0	152.660192	5691.0
42525	85036.0	85059.0	153.512281	5956.0
42526	85033.0	85044.0	160.825959	2203.0
42527	85055.0	85061.0	163.701428	668.0
42528	85062.0	85066.0	168.199295	6897.0
42529	85054.0	85060.0	168.924039	9414.0
42530	85028.0	85064.0	185.215769	3118.0
42531	85067.0	85071.0	187.832588	15370.0
42532	85056.0	85073.0	203.212147	17995.0
42533	85057.0	85063.0	205.285993	9221.0
42534	85068.0	85072.0	207.902660	5321.0
42535	85069.0	85075.0	236.754581	9889.0
42536	85070.0	85077.0	298.587755	16786.0
42537	85058.0	85078.0	309.946867	16875.0
42538	85074.0	85079.0	375.698458	34870.0
42539	85065.0	85080.0	400.711547	37221.0
42540	85076.0	85081.0	644.047472	42542.0

Обрежем дендрограмму так, чтобы осталось разумное количество кластеров, поддающееся обработке. Оно задается на основании значения переменной `distance_threshold`. Методом проб и ошибок установлено, что значение `distance_threshold`, равное `100`, приводит к созданию 32 кластеров; его мы и используем в данном примере.

```
from scipy.cluster.hierarchy import fcluster

distance_threshold = 100
clusters = fcluster(Z, distance_threshold, criterion='distance')
X_train_hierClustered = pd.DataFrame(data=clusters, \
    index=X_train_PCA.index, columns=['cluster'])

print("Количество различных кластеров:", \
      len(X_train_hierClustered['cluster'].unique()))
```

Количество различных кластеров при выбранном нами значении `distance_threshold` равно 32.

```
countByCluster_hierClust, countByLabel_hierClust, \
    countMostFreq_hierClust, accuracyDF_hierClust, \
    overallAccuracy_hierClust, accuracyByLabel_hierClust = \
    analyzeCluster(X_train_hierClustered, y_train)
print("Общая точность иерархической кластеризации:", \
      overallAccuracy_hierClust)
```

Ниже показана общая точность иерархической кластеризации:

```
Общая точность иерархической кластеризации: 0.3651685393258427
```

Она составляет приблизительно 37%, что немного хуже, чем в методе *k*-средних. Но следует оговориться, что иерархическая кластеризация работает по собственному алгоритму, поэтому может точнее группировать некоторых заемщиков, тогда как в каких-то группах лучше проявляет себя алгоритм *k*-средних.

Другими словами, эти два алгоритма кластеризации могут дополнять друг друга, и данную возможность имеет смысл исследовать, объединив алгоритмы в один ансамбль и сравнив результаты работы ансамбля с результатами автономных решений (возможности ансамблевого подхода исследовались в главе 2). Как и в случае метода *k*-средних, точность заметно варьируется при переходе от одного кластера к другому. Некоторые кластеры гораздо более однородны по сравнению с остальными.

```
Точность кластеров при иерархической кластеризации

0       0.304124
1       0.219001
2       0.228311
3       0.379722
4       0.240064
5       0.272011
6       0.314560
7       0.263930
8       0.246138
9       0.318942
10      0.302752
11      0.269772
12      0.335717
13      0.330403
```

```
14      0.346320
15      0.440141
16      0.744155
17      0.502227
18      0.294118
19      0.236111
20      0.254727
21      0.241042
22      0.317979
23      0.308771
24      0.284314
25      0.243243
26      0.500000
27      0.289157
28      0.365283
29      0.479693
30      0.393559
31      0.340875
```

Применение кластеризации по методу HDBSCAN

Перейдем к рассмотрению алгоритма HDBSCAN и применим его для группирования схожих заемщиков из набора данных LendingClub.

Вспомните, что алгоритм HDBSCAN будет выполнять группирование заемщиков исходя из того, насколько плотно упакованы их атрибуты в пространстве высокой размерности. В отличие от метода *k*-средних или иерархической кластеризации будут сгруппированы не все заемщики. Некоторая часть заемщиков, значительно отличающаяся от других групп, может остаться несгруппированной. Они представляют собой выбросы и должны быть исследованы на предмет того, существуют ли какие-то серьезные бизнес-факторы, которые обусловливают такие отличия. Некоторым группам заемщиков можно автоматически назначать числовые категории займов, но для других, несхожих, заемщиков может потребоваться более придирчивый подход к оценке их кредитоспособности.

Проверим, насколько хорошо сработает алгоритм HDBSCAN.

```
import hdbscan

min_cluster_size = 20
min_samples = 20
```

```
alpha = 1.0
cluster_selection_method = 'leaf'

hdb = hdbscan.HDBSCAN(min_cluster_size=min_cluster_size, \
    min_samples=min_samples, alpha=alpha, \
    cluster_selection_method=cluster_selection_method)

X_train_hdbscanClustered = hdb.fit_predict(X_train)
X_train_hdbscanClustered = \
    pd.DataFrame(data=X_train_hdbscanClustered, index=X_train.index, \
                 columns=['cluster'])

countByCluster_hdbscan, countByLabel_hdbscan, \
    countMostFreq_hdbscan, accuracyDF_hdbscan, \
    overallAccuracy_hdbscan, accuracyByLabel_hdbscan = \
    analyzeCluster(X_train_hdbscanClustered, y_train)
```

Ниже показана общая точность алгоритма HDBSCAN:

```
Общая точность алгоритма HDBSCAN: 0.3246203751586667
```

Как видите, общая точность составляет приблизительно 32%, что хуже, чем давали методы *k*-средних и иерархической кластеризации.

Результаты кластеризации представлены в табл. 6.6.

***Таблица 6.6.** Результаты кластеризации с использованием алгоритма HDBSCAN*

	cluster	clusterCount
0	–1	32708
1	7	4070
2	2	3668
3	1	1096
4	4	773
5	0	120
6	6	49
7	3	38
8	5	20

К кластеру `-1` отнесены 32 708 заемщиков. Это означает, что они остаются несгруппированными.

Ниже представлены значения точности для каждого кластера по отдельности. Точность колеблется в пределах от 28 до 59%.

```
0       0.284487
1       0.341667
2       0.414234
3       0.332061
4       0.552632
5       0.438551
6       0.400000
7       0.408163
8       0.590663
```

Резюме

В этой главе мы разработали несколько приложений кластеризации на основе обучения без учителя, используя данные компании LendingClub о заемщиках, подававших заявки на получение беззалоговых персональных ссуд на протяжении 2007–2011 гг. Эти приложения базировались на методе *k*-средних, иерархической кластеризации и иерархическом алгоритме DBSCAN. Лучшим оказался метод *k*-средних, обеспечивший общую точность на уровне приблизительно 39%.

Несмотря на то что приложения работали удовлетворительно, их можно существенно улучшить. Используйте их в качестве отправной точки и попытайтесь усовершенствовать их, экспериментируя с применяемыми алгоритмами.

На этом мы завершаем часть, посвященную обучению без учителя с применением библиотеки Scikit-learn. В следующих главах мы исследуем технологии обучения без учителя на основе нейронных сетей с использованием библиотек TensorFlow и Keras. В главе 7 мы рассмотрим технологию обучения признакам и познакомимся с автокодировщиками.

Обучение без учителя с использованием библиотек TensorFlow и Keras

Итак, мы завершили часть, посвященную обучению без учителя с использованием библиотеки Scikit-learn, и готовы перейти к рассмотрению подходов на основе нейронных сетей. В следующих главах мы познакомимся с нейронными сетями и популярными фреймворками TensorFlow и Keras.

В главе 7 мы применим автокодировщик — мелкую нейронную сеть — для автоматического конструирования и отбора признаков. В главе 8 мы узнаем, как с помощью автокодировщиков решать практические задачи. В главе 9 мы обсудим, как превратить задачу обучения без учителя в задачу с частичным привлечением учителя, воспользовавшись несколькими имеющимися метками для улучшения точности и полноты модели, основанной исключительно на обучении без учителя.

Закончив с обзором мелких нейронных сетей, в последней части книги мы перейдем к рассмотрению глубоких нейронных сетей.

ГЛАВА 7

Автокодировщики

В первых шести главах демонстрировалось, как применять обучение без учителя для снижения размерности, а также кластеризации данных. С помощью рассмотренных алгоритмов мы создали ряд приложений, предназначенных для обнаружения аномалий и сегментирования групп на основе сходства.

Однако обучение без учителя способно на гораздо большее. Одно из направлений, в которых обучению без учителя нет равных, — *выделение признаков* (feature extraction), метод, применяемый для генерировапия нового представления признаков из их оригинального набора. Это новое представление называется *обученным*; оно используется для улучшения производительности в задачах обучения с учителем. Другими словами, извлечение признаков — это предварительное обучение без учителя, завершающееся стадией обучения с учителем.

Одну из технологий извлечения признаков реализуют автокодировщики. В них используется *нерекуррентная нейронная сеть прямого распространения для обучения признакам* (feature learning). Концепция обучения признакам составляет суть всего направления машинного обучения, основанного на нейронных сетях.

В автокодировщиках каждый слой нейронной сети обучается представлению оригинальных признаков, и каждый последующий слой достраивает представление, которому обучился предыдущий слой. Слой за слоем автокодировщик обучается все более сложным представлениям, последовательно выстраивая структуру, известную как *иерархия понятий*, которая становится все более и более абстрактной.

Выходным слоем будет конечное обученное представление оригинальных признаков. Это обученное представление можно затем подать на вход любой модели обучения с учителем с целью уменьшения ошибки обобщения.

Но не будем забегать далеко вперед, а начнем с ознакомления с нейронными сетями и предназначенными для работы с ними фреймворками Python: TensorFlow и Keras.

Нейронные сети

По своей сути нейронные сети реализуют обучение признакам, при котором каждый слой нейронной сети обучается представлению, созданному предыдущим слоем. Формируя слой за слоем все более детализированные представления, отражающие дополнительные нюансы, нейронные сети способны справляться с такими нетривиальными задачами, как компьютерное зрение, распознавание речи и машинный перевод.

Нейронные сети бывают мелкими и глубокими. Мелкие сети состоят из небольшого количества слоев, а глубокие сети — из множества слоев. Глубокое обучение унаследовало свое название от глубоких (многослойных) нейронных сетей. Мелкие сети не слишком эффективны, поскольку степень обучения признакам ограничена малым количеством слоев. С другой стороны, глубокие сети обладают огромными возможностями и в настоящее время олицетворяют собой передовой край машинного обучения.

Для большей ясности следует подчеркнуть, что как мелкое, так и глубокое обучение — лишь часть экосистемы машинного обучения в целом. Машинное обучение с использованием нейронных сетей отличается от классического машинного обучения в основном тем, что в первом случае конструирование признаков в значительной мере осуществляется автоматически, а во втором — вручную.

Нейронные сети содержат *входной слой*, один или несколько *скрытых слоев* и *выходной слой*. Количество скрытых слоев определяет *глубину нейронной сети*. Скрытые слои можно рассматривать как слои промежуточных вычислений; их совокупное действие позволяет нейронной сети аппроксимировать сложные функции.

Каждый слой состоит из определенного количества образующих его узлов, называемых *нейронами*. Узлы каждого слоя соединяются с узлами следующего слоя. В процессе обучения нейронная сеть определяет, какой вес следует назначить каждому из узлов.

Помимо добавления в нейронную сеть дополнительных слоев, мы можем добавлять в нее дополнительные узлы для расширения возможностей сети с точки зрения моделирования сложных отношений. Выходы этих узлов передаются *функции активации*, которая определяет, какое значение текущего слоя будет передано следующему слою нейронной сети. К числу распространенных функций активации относятся *линейная функция*, *сигмоида*, *гиперболический тангенс* и *линейный выпрямитель* (*ReLU*). В качестве конечной функции активации обычно используют функцию *Softmax*, выводящую вероятность того,

что входное наблюдение относится к определенному классу. Такая ситуация довольно типична для задач классификации.

Кроме того, нейронные сети могут иметь *узлы смещения* (bias nodes). В отличие от обычных узлов они всегда имеют постоянные значения и не связываются с предыдущим слоем. Их назначение заключается в том, чтобы смещать выход функции активации в сторону больших или меньших значений. Посредством скрытых слоев — включая нейроны, узлы смещения и функции активации — нейронная сеть пытается обучиться правильно аппроксимировать функцию, которая применяется для трансляции входного слоя в выходной.

В задачах обучения с учителем это делается достаточно просто. Входной слой представляет признаки, поступающие в нейронную сеть, а выходной — метки, назначенные каждому наблюдению. В процессе обучения нейронная сеть определяет, какие значения *весов* узлов обеспечивают минимизацию расхождения между предсказанной меткой каждого наблюдения и истинной меткой. В случае обучения без учителя нейронная сеть обучается представлениям входного слоя через различные скрытые слои, не руководствуясь никакими метками.

Нейронные сети обладают огромными возможностями и способны моделировать сложные нелинейные отношения на уровне, которого классическим алгоритмам машинного обучения трудно достичь. В целом это замечательное свойство нейронных сетей, однако существуют и потенциальные риски. Поскольку нейронные сети могут моделировать столь сложные нелинейные отношения, они в гораздо большей степени подвержены переобучению, что необходимо учитывать при создании приложений машинного обучения на основе нейронных сетей[1].

Существует множество типов нейронных сетей, таких как *рекуррентные нейронные сети*, данные которых могут перетекать в любом направлении (используются для распознавания речи и машинного перевода), и *сверточные нейронные сети* (применяются в машинном зрении). Но мы сосредоточимся на рассмотрении более простых нейронных сетей прямого распространения, в которых данные перемещаются лишь в одном направлении: прямом.

Кроме того, чтобы создаваемые нами нейронные сети обладали хорошей производительностью, мы должны уделить внимание оптимизации гиперпараметров, включая выбор функции потерь, алгоритма минимизации потерь, типа инициализации начальных значений весов, количества итераций тренировки нейронной сети (т.е. количества эпох), количества наблюдений,

[1] Процесс контроля переобучения называется *регуляризацией*.

передаваемых перед каждым обновлением весов (т.е. размер пакета), и величины шага изменения весов (т.е. скорости обучения) в процессе тренировки сети.

TensorFlow

Прежде чем приступить к знакомству с автокодировщиками, рассмотрим возможности *TensorFlow* — основной библиотеки, которую мы будем использовать для построения нейронных сетей. TensorFlow — это библиотека с открытым исходным кодом, предназначенная для выполнения высокопроизводительных вычислений. Первоначально она была разработана командой проекта GoogleBrain для применения внутри компании Google. В ноябре 2015 года она была выпущена в виде ПО с открытым исходным кодом[2].

Библиотека TensorFlow доступна для большинства операционных систем (включая Linux, macOS, Windows, Android и iOS) и может выполняться на множестве CPU и GPU, предлагая высокую степень масштабируемости программ и возможность их развертывания на рабочих станциях и мобильных устройствах, а также в веб-среде или в облаке.

Вся прелесть библиотеки TensorFlow заключается в том, что пользователь может определить нейронную сеть — или, в более общей формулировке, вычислительный граф — на языке Python, но при этом выполнять вычисления, используя код на языке C++, который работает намного быстрее кода Python.

Кроме того, TensorFlow позволяет *распараллеливать* вычисления, разбивая последовательность операций на блоки и выполняя их параллельно на нескольких CPU и GPU. Вопросы производительности играют важную роль при проектировании крупномасштабных приложений машинного обучения, таких как поисковые системы Google.

Несмотря на то что существуют другие нейросетевые библиотеки с открытым исходным кодом, TensorFlow пользуется наибольшей популярностью, во многом благодаря авторитету Google.

Пример использования TensorFlow

Для начала создадим граф TensorFlow и выполним вычисления. Мы импортируем библиотеку TensorFlow, определим несколько переменных (программный интерфейс TensorFlow напоминает интерфейс библиотеки Scikit-learn, с

[2] Дополнительная информация о библиотеке TensorFlow доступна на сайте `www.tensorflow.org`.

которой мы работали в предыдущих главах), а затем вычислим значения этих переменных.

```
import tensorflow as tf

b = tf.constant(50)
x = b * 10
y = x + b

with tf.Session() as sess:
    result = y.eval()
    print(result)
```

Важно понимать, что в данном случае имеют место две фазы. Сначала мы конструируем вычислительный граф, определяя переменные `b`, `x` и `y`. Затем мы запускаем расчет графа посредством вызова `tf.Session()`. До этого момента ни CPU, ни GPU не выполняют никаких вычислений. Мы лишь сохраняем инструкции для будущих вычислений. Результатом работы программы будет вывод числа `550`.

В следующих главах мы будем создавать нейронные сети с помощью TensorFlow.

Keras

Keras — высокоуровневая библиотека с открытым исходным кодом, работающая поверх TensorFlow. Она предоставляет удобный интерфейс для доступа к TensorFlow, с которым проще выполнять эксперименты, чем пытаться напрямую вводить команды TensorFlow. Библиотека Keras тоже была разработана одним из сотрудников компании Google, инженером Франсуа Шолле.

Когда мы начнем строить нейросетевые модели с помощью TensorFlow, мы познакомимся с библиотекой Keras и исследуем ее преимущества.

Автокодировщик: кодировщик и декодировщик

Теперь, когда вы уже имеете определенное представление о нейронных сетях и популярных библиотеках для работы с ними — TensorFlow и Keras, — необходимо познакомиться с автокодировщиком, который реализует одну из простейших нейронных сетей для обучения без учителя.

Автокодировщик состоит из двух компонентов: *кодировщика* и *декодировщика*. Кодировщик преобразует входной набор признаков в другое представ-

ление (посредством обучения признакам), а декодировщик преобразует это новое представление в оригинальный формат.

Базовая концепция автокодировщика аналогична концепции снижения размерности, которую мы изучили в главе 3. Как и в случае снижения размерности данных, автокодировщик не запоминает оригинальные наблюдения и признаки, что было бы эквивалентно *тождественному отображению*. От такого автокодировщика не было бы никакой пользы. Вместо этого он должен аппроксимировать оригинальные наблюдения с как можно более высокой, но не абсолютной, точностью, используя обученное представление. Иными словами, автокодировщик учится аппроксимировать тождественное отображение.

Поскольку автокодировщик ограничен в своих действиях, он вынужден обучаться наиболее существенным свойствам оригинальных данных, захватывая их базовую структуру. Это аналогично тому, что происходит при снижении размерности данных. Наличие ограничения служит важной характеристикой автокодировщиков — оно заставляет их тщательно взвешивать решения о том, какую информацию следует захватывать ввиду ее важности, а какую — отбрасывать, поскольку она менее важна или не является полезной.

Концепция автокодировщиков была предложена несколько десятилетий назад, и с тех пор они широко применяются для снижения размерности данных и автоматического конструирования признаков. В наши дни на их основе создают *генеративные модели*, в частности, *генеративно-состязательные сети*.

Неполные автокодировщики

В автокодировщике нас в первую очередь интересует кодировщик, поскольку именно он обучается новому представлению оригинальных данных. Этим новым представлением служит новый набор признаков, полученных из оригинального набора признаков и наблюдений.

Обозначим через $h = f(x)$ функцию-кодировщик, которая получает оригинальные наблюдения x и применяет обученное представление, захваченное функцией f, для формирования вывода h. Функция-декодировщик $r = g(h)$ реконструирует оригинальные наблюдения.

Как видите, декодировщик получает выход h кодировщика и реконструирует наблюдения r, используя функцию g. Если все сделано корректно, то функция $g(f(x))$ не будет совпадать с x во всех точках, но будет близка к ним.

Как ограничить возможности функции-кодировщика в отношении аппроксимации x таким образом, чтобы вынудить ее обучаться лишь наиболее существенным свойствам x, избегая их точного копирования?

Мы можем ограничить выход h функции-кодировщика так, чтобы он имел меньшую размерность, чем x. Такой кодировщик называется *неполным*, поскольку его размерность меньше оригинальной размерности входа. Опять-таки, это напоминает то, что происходит в процессе снижения размерности данных, когда оригинальные входные измерения редуцируются до набора гораздо меньшей размерности.

Ограниченный подобным способом автокодировщик пытается минимизировать *функцию потерь*, определенную нами так, чтобы сделать ошибку реконструкции (измеренную после приближенного реконструирования наблюдений на основе выхода кодировщика) как можно меньшей. Важно понимать, что скрытые слои находятся там, где количество измерений ограничено. Другими словами, выход кодировщика имеет меньшую размерность, чем оригинальный вход. Однако выходом декодировщика становится реконструкция оригинальных данных, имеющая ту же размерность, что и оригинальный вход.

Если декодировщик линейный, а функцией потерь является среднеквадратическая ошибка, то неполный автокодировщик обучится тому же представлению, что и в случае PCA (алгоритм снижения размерности, рассмотренный в главе 3). В то же время, если функции кодировщика и декодировщика нелинейны, то автокодировщик может обучиться намного более сложным представлениям. Это именно то, что интересует нас больше всего. Впрочем, если предоставить автокодировщику чересчур большую свободу действий при моделировании сложных нелинейных представлений, то он просто запомнит/сохранит оригинальные наблюдения вместо того, чтобы извлечь из них наиболее существенную информацию. Поэтому мы должны налагать на автокодировщик разумные ограничения во избежание подобного развития событий.

Сверхполные автокодировщики

Если автокодировщик обучается представлению, размерность которого превышает размерность входа, то он рассматривается как *сверхполный*. Такие автокодировщики просто копируют оригинальные представления и не имеют цели эффективно и компактно захватывать информацию об оригинальном распределении, как это делают неполные автокодировщики. Но если применить ту или иную форму *регуляризации*, при которой нейронная сеть штрафуется за обучение ненужным сложным функциям, то сверхполные

автокодировщики могут послужить для снижения размерности данных и автоматического конструирования признаков.

В сравнении с неполными автокодировщиками *регуляризованные сверхполные автокодировщики* сложнее проектировать, но их возможности гораздо шире, поскольку они могут обучаться более сложным представлениям, которые лучше аппроксимируют оригинальные наблюдения.

По сути, хорошие автокодировщики способны обучаться новым представлениям, которые достаточно точно аппроксимируют оригинальные наблюдения, но не являются их точной копией. Это достигается за счет обучения новому распределению вероятности.

Плотные и разреженные автокодировщики

В главе 3 мы исследовали как плотные (обычные), так и разреженные версии алгоритмов снижения размерности. Аналогичным образом работают и автокодировщики. До сих пор мы обсуждали лишь обычный автокодировщик, формирующий на выходе плотную матрицу, в которой захваченная наиболее существенная информация об оригинальных данных распределяется по ключевым признакам. Альтернативный вариант — выводить разреженную матрицу, в которой захваченная наиболее существенная информация более равномерно распределяется по обучаемым признакам.

Для этого мы должны включить в автокодировщик не только *ошибку реконструкции*, но и *штрафование разреженности*, чтобы автокодировщик учитывал разреженность окончательной матрицы. Разреженные автокодировщики обычно являются переполненными: количество элементов в скрытых слоях превышает количество входных признаков, с той оговоркой, что лишь небольшой доле скрытых элементов разрешено находиться в активном состоянии в одно и то же время. Соответствующий этому определению *разреженный автокодировщик* будет выводить окончательную матрицу со многими нулями и лучшим распределением захваченной информации по признакам, которым обучилась модель.

В некоторых приложениях машинного обучения разреженные автокодировщики демонстрируют лучшую производительность и обучаются несколько иным представлениям, чем обычные (плотные) автокодировщики. В следующих главах мы рассмотрим практические примеры, чтобы увидеть разницу между этими двумя типами автокодировщиков.

Шумоподавляющий автокодировщик

Как вы уже знаете, автокодировщики способны обучаться новым (и улучшенным) представлениям на оригинальных входных данных, захватывая наиболее существенные признаки и отбрасывая содержащийся в данных шум.

В некоторых случаях необходимо, чтобы проектируемый автокодировщик более агрессивно подавлял шум, особенно если мы подозреваем, что данные были повреждены. Представьте, что вам нужно записать разговор двух людей в шумном кафетерии в полдень. Мы хотели бы изолировать разговор (сигнал) от фоновых звуков (шума). Другая похожая задача — обработка набора зернистых или размытых изображений. В этом случае мы хотели бы отделить основное изображение (сигнал) от искажений (шума).

Для решения подобных задач создается *шумоподавляющий* (*обесшумливающий*) *автокодировщик* (denoising autoencoder), который получает поврежденные данные на вход и обучается воссоздавать на выходе оригинальные данные с как можно более низким процентом повреждений. Добиться этого не так-то просто, но совершенно очевидно, что автокодировщики такого типа будут чрезвычайно полезными при решении прикладных задач.

Вариационный автокодировщик

До сих пор мы обсуждали применение автокодировщиков для обучения новым представлениям оригинальных входных данных (получаемых кодировщиком) с целью минимизации ошибки реконструкции выходных данных (генерируемых декодировщиком).

В таких конфигурациях кодировщик имеет фиксированный размер n, где n обычно меньше количества оригинальных измерений. Другими словами, мы обучаем неполный автокодировщик. Иногда n превышает количество оригинальных измерений (сверхполный автокодировщик), однако ограничивается штрафами регуляризации, разреженности и т.п. Так или иначе, во всех этих случаях кодировщик выводит одиночный вектор фиксированного размера n.

Альтернативное решение — *вариационный автокодировщик* (variational autoencoder), в котором кодировщик формирует два вектора вместо одного: вектор средних значений (*мю*) и вектор стандартных отклонений (*сигма*). Эти два вектора образуют такие случайные переменные, что i-е элементы векторов *мю* и *сигма* соответствуют *среднему значению* и *стандартному отклонению* i-й

случайной переменной. Создавая такой стохастический выход посредством своего кодировщика, вариационный автокодировщик может формировать выборки в пределах всего непрерывного пространства, исходя из того, чему он обучился на входных данных.

Действие вариационного автокодировщика не ограничено лишь образцами, на которых он обучался. Он способен обобщаться и создавать на выходе новые образцы, даже если ничего подобного им он раньше не получал. Это невероятно мощная возможность, поскольку теперь вариационные автокодировщики могут генерировать новые синтетические данные, близкие к распределению, которому автокодировщик обучился на оригинальных входных данных. Разработки подобного рода привели к совершенно новому и набирающему все более широкий размах направлению в области обучения без учителя, известному как *генеративное (порождающее) моделирование*, которое охватывает *генеративно-состязательные сети*. Применение таких моделей делает возможным построение синтетических изображений, речи, музыки и т.п., что открывает широкие перспективы для генерирования данных с помощью ИИ.

Резюме

В этой главе мы познакомились с нейронными сетями и популярными библиотеками для работы с ними: TensorFlow и Keras. Мы также исследовали автокодировщики и выяснили их способность к обучению новым представлениям на оригинальных входных данных. Существует множество разновидностей автокодировщиков, включая разреженные, шумоподавляющие и вариационные.

В главе 8 мы разработаем ряд приложений на основе изученных здесь концепций.

Прежде чем продолжать, следует еще раз обсудить вопрос о том, почему автоматическое выделение признаков играет столь важную роль. Не имея такой возможности, исследователи и разработчики вынуждены были бы вручную конструировать признаки, требуемые для решения прикладных задач. Это трудоемкое занятие, которое существенно замедлило бы прогресс в области ИИ.

По правде говоря, до тех пор пока Джеффри Хинтон и другие исследователи не разработали методы автоматического обучения новым признакам с помощью нейронных сетей, тем самым положив начало революции глубокого обучения в 2006 году, такие задачи, как компьютерное зрение, распознавание

речи, машинный перевод и многое другое, в значительной степени оставались нерешаемыми.

С появлением автокодировщиков и других разновидностей нейронных сетей, обеспечивших возможность автоматического извлечения признаков из входных данных, многие из задач перешли в категорию решаемых, что привело к важным прорывам, произошедшим в области машинного обучения за последнее десятилетие.

Вы сами убедитесь в возможностях автоматического извлечения признаков на примере приложений, рассмотренных в главе 8.

Реализация автокодировщиков

В этой главе мы будем создавать приложения, используя различные варианты автокодировщиков, включая неполный, сверхполный, разреженный и шумоподавляющий.

Мы вновь обратимся к задаче выявления попыток мошенничества с банковскими картами, начатой в главе 2. Всего имеются данные о 284 807 транзакциях, из которых лишь 492 мошеннические. Используя модель обучения с учителем, мы достигли средней точности 0,83, что несомненно является впечатляющим результатом. Мы смогли выявить 80% подделок с точностью, превышающей 80%. Применив модель обучения без учителя, мы достигли средней точности на уровне 0,69, что тоже неплохо, если учесть, что мы обошлись без каких-либо меток. Нам удалось обнаружить свыше 75% подделок с точностью свыше 75%.

Посмотрим, каких результатов удастся добиться с помощью автокодировщика, который тоже представляет собой алгоритм обучения без учителя, только на основе нейронной сети.

Подготовка данных

Прежде всего, загрузим необходимые библиотеки.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline
```

```
'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score

'''Алгоритмы'''
import lightgbm as lgb

'''TensorFlow и Keras'''
import tensorflow as tf
import keras
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout
from keras.layers import BatchNormalization, Input, Lambda
from keras import regularizers
from keras.losses import mse, binary_crossentropy
```

Теперь загрузим набор данных и подготовим его к работе. Мы создадим матрицу `dataX` со всеми PCA-компонентами, но отбросим признаки `Class` и `Time`. Метки класса будут храниться в матрице `dataY`. Мы также масштабируем признаки, сохраненные в матрице `dataX`, так, чтобы их среднее значение было равно нулю, а стандартное отклонение — единице.

```
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'credit_card_data', \
                         'credit_card.csv'])
data = pd.read_csv(current_path + file)
dataX = data.copy().drop(['Class', 'Time'], axis=1)
dataY = data['Class'].copy()
featuresToScale = dataX.columns
sX = pp.StandardScaler(copy=True, with_mean=True, with_std=True)
dataX.loc[:, featuresToScale] = \
    sX.fit_transform(dataX[featuresToScale])
```

Как и в главе 3, создадим тренировочный набор, содержащий две трети данных и меток, и тестовый набор, содержащий оставшуюся треть.

Сохраним тренировочный и тестовый наборы в переменных `X_train_AE` и `X_test_AE` соответственно. Вскоре мы используем их в автокодировщике.

```
X_train, X_test, y_train, y_test = \
    train_test_split(dataX, dataY, test_size=0.33, \
                     random_state=2018, stratify=dataY)

X_train_AE = X_train.copy()
X_test_AE = X_test.copy()
```

Кроме того, мы повторно используем введенную ранее функцию `anomalyScores` для вычисления ошибки реконструкции, характеризующей расхождение между оригинальной и реконструированной матрицами признаков. Фуикция получает сумму квадратичных ошибок и нормализует их путем приведения к диапазону значений от нуля до единицы.

Эта функция играет решающую роль. Наиболее аномальные транзакции — те, ошибка реконструкции которых близка к единице (т.е. максимальна). Они, вероятнее всего, являются мошенническими. Транзакции с ошибками, близкими к нулю, имеют наименьшую ошибку реконструкции и, скорее всего, являются нормальными.

```
def anomalyScores(originalDF, reducedDF):
    loss = np.sum((np.array(originalDF) - np.array(reducedDF))**2, \
                  axis=1)
    loss = pd.Series(data=loss, index=originalDF.index)
    loss = (loss - np.min(loss)) / (np.max(loss) - np.min(loss))
    return loss
```

Мы также повторно используем функцию `plotResults`, которая строит график "точность — полнота", вычисляет среднюю точность и отображает ROC-кривую.

```
def plotResults(trueLabels, anomalyScores, returnPreds = False):
    preds = pd.concat([trueLabels, anomalyScores], axis=1)
    preds.columns = ['trueLabel', 'anomalyScore']
    precision, recall, thresholds = \
        precision_recall_curve(preds['trueLabel'], \
                               preds['anomalyScore'])
    average_precision = average_precision_score(preds['trueLabel'], \
        preds['anomalyScore'])

    plt.step(recall, precision, color='k', alpha=0.7, where='post')
    plt.fill_between(recall, precision, step='post', alpha=0.3, \
```

```
                color='k')

    plt.xlabel('Полнота)
    plt.ylabel('Точность')
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])

    plt.title('Кривая "точность - полнота": средняя точность = \
              {0:0.2f}'.format(average_precision))

    fpr, tpr, thresholds = roc_curve(preds['trueLabel'], \
                                     preds['anomalyScore'])
    areaUnderROC = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='r', lw=2, label='ROC-кривая')
    plt.plot([0, 1], [0, 1], color='k', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Доля ложноположительных исходов')
    plt.ylabel('Доля истинноположительных исходов')
    plt.title('    Рабочая характеристика приемника: \n \
          площадь под кривой = {0:0.2f}'.format(areaUnderROC))
    plt.legend(loc="lower right")
    plt.show()

    if returnPreds==True:
        return preds
```

Компоненты автокодировщика

Сначала мы создадим очень простой автокодировщик с входным слоем, единственным скрытым слоем и выходным слоем. Мы будем передавать автокодировщику оригинальную матрицу признаков x — она представляется входным слоем. Затем ко входному слою применяется функция активации f, генерирующая скрытый слой. Это наш *кодировщик*. Скрытый слой h, являющийся эквивалентом $f(x)$, представляет собой обученное представление.

Для реконструкции оригинальных наблюдений к скрытому слою (т.е. к обученному представлению) применяется функция активации g. Это наш *декодировщик*. Выходной слой r, являющийся эквивалентом $g(h)$, представляет

собой реконструированные наблюдения. Мы будем вычислять ошибку реконструкции, сравнивая реконструированные наблюдения r с оригинальными наблюдениями x.

Функции активации

Прежде чем принимать решение относительно количества узлов, используемых в автокодировщике с одним скрытым слоем, обсудим функции активации.

Нейронная сеть обучается весам, которые назначаются узлам в каждом из слоев, но будут ли активироваться узлы (для использования в следующем слое), определяется функцией активации. Другими словами, функция активации применяется к взвешенному входу (плюс смещение, если оно введено) каждого слоя. Мы назовем этот взвешенный вход Y.

Функция активации получает значение Y и либо активируется (если Y превышает определенный порог), либо не активируется. В первом случае информация передается от данного узла в следующий слой, во втором — не передается. Однако нас не устраивают простые бинарные функции активации. Мы хотим работать с диапазоном активационных значений. В этом плане у нас есть выбор между линейной и нелинейной функциями активации. Линейная функция не имеет ограничений и способна генерировать значения от минус бесконечности до плюс бесконечности. К числу распространенных нелинейных фуикций относятся *сигмоида*, гиперболический тангенс (*tanh*), линейный выпрямитель (*ReLU*) и *Softmax*.

Сигмоида

Сигмоида ограничена и генерирует значения в диапазоне от нуля до единицы.

Гиперболический тангенс

Функция *tanh* также ограничена и может генерировать значения в диапазоне от минус единицы до плюс единицы. Она имеет более крутой градиент, чем сигмоида.

ReLU

Эта фуикция обладает одним интересным свойством. Если Y имеет положительное значение, то ReLU возвращает Y, в противном случае возвращается нуль. Поэтому линейный выпрямитель не ограничен для положительных значений Y.

Softmax

Softmax используют в качестве конечной функции активации нейронной сети для задач классификации, поскольку она нормализует вероятности классифицируемых категорий, сумма которых в итоге дает вероятность, равную единице.

Из всех этих функций линейная функция самая простая и требует наименьшего объема вычислений. Вторая по вычислительной стоимости — функция ReLU, за которой следуют все остальные.

Наш первый автокодировщик

Начнем с двухслойного автокодировщика с линейной функцией активации как в кодировщике, так и в декодировщике. Обратите внимание на то, что при подсчете числа слоев нейронной сети учитывается только вклад скрытых слоев и выходного слоя. Поскольку у нас имеется всего один скрытый слой, мы говорим, что сеть двухслойная.

Для создания нейронной сети с помощью библиотек TensorFlow и Keras мы прежде всего должны вызвать *API-функцию модели Sequential*. Эта модель представляет собой линейный стек слоев. Прежде чем компилировать ее и обучать на данных, необходимо указать ей, какие типы слоев нам нужны[1].

```
# Модель №1:
# двухслойный полный автокодировщик с линейной функцией активации

# Вызов API-функции нейронной сети
model = Sequential()
```

После вызова модели Sequential мы должны задать форму входа в виде числа измерений, которое должно совпадать с количеством измерений оригинальной матрицы признаков `dataX`, равным 29.

Следует также задать функцию активации (*кодировщик*), которая будет применяться ко входному слою, и количество узлов в скрытом слое. Мы будем использовать линейную функцию активации (`linear`).

Наш первый автокодировщик будет полным. В нем количество узлов скрытого слоя равно количеству узлов входного слоя, т.е. 29. Все это делается с помощью одной строки кода:

```
model.add(Dense(units=29, activation='linear', input_dim=29))
```

[1] Дополнительную информацию о модели Sequential можно найти в официальной документации (`http://bit.ly/2FZbUrq`).

Аналогичным образом задается функция активации (*декодировщик*), применяемая к скрытому слою, и количество узлов в выходном слое. Поскольку мы хотим, чтобы окончательная реконструированная матрица имела то же количество измерений, что и оригинальная, оно должно быть равно 29. Для декодировщика мы также будем использовать линейную функцию активации:

```
model.add(Dense(units=29, activation='linear'))
```

Далее необходимо скомпилировать слои нейронной сети. В качестве аргументов компилятора указывается *функция потерь* (loss function), управляющая обучением весов, *оптимизатор*, задающий процесс, в соответствии с которым обучаются веса, и список выводимых *метрик*, что поможет нам оценить пригодность нейронной сети.

Функция потерь

Начнем с функции потерь. Вспомните, что мы оцениваем модель на основании ошибки реконструкции, отражающей расхождение между матрицей признаков, реконструированной с помощью автокодировщика, и оригинальной матрицей признаков, которую мы передаем автокодировщику.

Поэтому в качестве функции потерь мы выбираем *среднеквадратическую ошибку*[2]. (В нашем примере мы используем ее эквивалент — *сумму квадратов ошибок*.)

Оптимизатор

Тренировка нейронных сетей осуществляется в несколько этапов, называемых *эпохами*. На каждой эпохе нейронная сеть подстраивает обученные веса с целью снижения уровня потерь, достигнутого на предыдущем этапе. Процесс обучения весов задается оптимизатором. Нам нужен процесс, который поможет нейронной сети эффективно обучаться оптимальным весам для различных узлов по всем слоям и минимизировать выбранную функцию потерь.

Чтобы обучиться оптимальным весам, нейронная сеть должна определенным образом корректировать свои "догадки". Один из подходов заключается в итеративном смещении весов в направлении, ведущем к инкрементному уменьшению функции потерь. Более оптимальный подход — смещение весов в этом же направлении, но с определенной степенью случайности, иными словами, стохастическое смещение.

[2] Дополнительную информацию о функциях потерь можно найти в официальной документации Keras (`https://keras.io/losses/`).

Этот процесс известен как *стохастический градиентный спуск* (stochastic gradient descent — SGD). Именно он чаще всего применяется при обучении нейронных сетей[3]. В SGD у всех весов одна скорость обучения, задаваемая гиперпараметром *альфа*, и эта скорость не изменяется в процессе тренировки. И все же в большинстве случаев следует корректировать скорость обучения по ходу тренировки. Например, на ранних эпохах имеет смысл более заметно изменять веса — другими словами, использовать большие значения скорости обучения *альфа*.

В более поздние эпохи, когда веса уже достигли значений, близких к оптимальным, разумнее использовать меньшую скорость обучения и перейти к тонкой настройке весов, вместо того чтобы совершать крупные шаги в том или ином направлении. Эти соображения привели к созданию еще более эффективного оптимизатора, чем SGD, — *алгоритма оптимизации Adam* (adaptive moment estimation — адаптивная оценка моментов). В отличие от SGD, оптимизатор Adam динамически подстраивает скорость обучения в процессе тренировки, и именно его мы будем использовать[4].

В оптимизаторе Adam мы можем изменять значение гиперпараметра *альфа*, управляющего скоростью обновления весов. Большие значения *альфа* приводят к ускоренному начальному обучению.

Тренировка модели

Наконец, необходимо задать оценочную метрику. Чтобы не усложнять анализ, мы выберем метрику `accuracy`[5].

```
model.compile(optimizer='adam', loss='mean_squared_error', \
              metrics=['accuracy'])
```

Далее следует указать количество эпох и размер пакета, после чего начать процесс обучения, вызвав метод `fit`. Количество эпох определяет, сколько раз будет выполнена тренировка по всему набору данных, переданному нейронной сети. Установим это количество равным `10`.

Размер пакета определяет количество выборок, на которых обучается нейронная сеть, прежде чем выполнить очередное обновление градиента. Если

[3] Дополнительную информацию о стохастическом градиентном спуске можно найти в Википедии (`http://bit.ly/2G3Ak30`).

[4] Для получения более подробной информации об оптимизаторах обратитесь к официальной документации Keras (`https://keras.io/optimizers/`).

[5] Дополнительную информацию об оценочных метриках можно найти в официальной документации Keras (`https://keras.io/metrics/`).

размер пакета равен общему количеству наблюдений, то нейронная сеть будет обновлять градиент только один раз за эпоху. В противном случае градиент будет обновляться несколько раз на протяжении каждой эпохи. Мы установим для этого параметра типичное значение 32.

Методу `fit` передается начальная входная матрица `x` и целевая матрица `y`. В нашем случае как `x`, так и `y` будут оригинальной матрицей признаков `X_train_AE`, поскольку мы хотим вычислить ошибку реконструкции, сравнив выход автокодировщика (реконструированную матрицу признаков) с исходной матрицей.

Вспомните, что рассматриваемое решение основано исключительно на обучении без учителя, поэтому мы вообще не будем использовать матрицу `y`. По ходу дела мы также будем проверять нашу модель, вычисляя ошибку реконструкции на всей тренировочной матрице.

```
num_epochs = 10
batch_size = 32

history = model.fit(x=X_train_AE, y=X_train_AE, epochs=num_epochs, \
    batch_size=batch_size, shuffle=True, \
    validation_data=(X_train_AE, X_train_AE), verbose=1)
```

Поскольку мы работаем с полным автокодировщиком, в котором скрытый слой имеет то же количество измерений, что и входной слой, потери будут очень низкими как для тренировочного, так и для валидационного наборов.

```
Epoch 1/10
190820/190820 [==============================] - 29s 154us/step -
 loss: 0.1056 - acc: 0.8728 - val_loss: 0.0013 - val_acc: 0.9903
Epoch 2/10
190820/190820 [==============================] - 27s 140us/step -
 loss: 0.0012 - acc: 0.9914 - val_loss: 1.0425e-06 - val_acc: 0.9995
Epoch 3/10
190820/190820 [==============================] - 23s 122us/step -
 loss: 6.6244 e-04 - acc: 0.9949 - val_loss: 5.2491e-04 - val_acc:
 0.9913
Epoch 4/10
190820/190820 [==============================] - 23s 119us/step -
 loss: 0.0016 - acc: 0.9929 - val_loss: 2.2246e-06 - val_acc: 0.9995
Epoch 5/10
190820/190820 [==============================] - 23s 119us/step -
 loss: 5.7424 e-04 - acc: 0.9943 - val_loss: 9.0811e-05 - val_acc:
 0.9970
```

```
Epoch 6/10
190820/190820 [==============================] - 22s 118us/step -
 loss: 5.4950 e-04 - acc: 0.9941 - val_loss: 6.0598e-05 - val_acc:
 0.9959
Epoch 7/10
190820/190820 [==============================] - 22s 117us/step -
 loss: 5.2291 e-04 - acc: 0.9946 - val_loss: 0.0023 - val_acc: 0.9675
Epoch 8/10
190820/190820 [==============================] - 22s 117us/step -
 loss: 6.5130 e-04 - acc: 0.9932 - val_loss: 4.5059e-04 - val_acc:
 0.9945
Epoch 9/10
190820/190820 [==============================] - 23s 122us/step -
 loss: 4.9077 e-04 - acc: 0.9952 - val_loss: 7.2591e-04 - val_acc:
 0.9908
Epoch 10/10
190820/190820 [==============================] - 23s 118us/step -
 loss: 6.1469 e-04 - acc: 0.9945 - val_loss: 4.4131e-06 - val_acc:
 0.9991
```

Это не оптимальное решение — автокодировщик слишком точно реконструировал исходную матрицу признаков, просто запомнив входы.

Не забывайте о том, что автокодировщик должен обучаться новому представлению, которое захватывает лишь наиболее существенную информацию из оригинальной входной матрицы, отбрасывая менее релевантную информацию. Простое запоминание входов, называемое *тождественным отображением*, не приводит к обучению улучшенному представлению.

Оценка модели на тестовом наборе

Используем тестовый набор для оценки того, насколько успешно данный автокодировщик способен идентифицировать мошеннические транзакции с банковскими картами в наборе данных. Мы сделаем это с помощью метода `predict`.

```
predictions = model.predict(X_test, verbose=1)
anomalyScoresAE = anomalyScores(X_test, predictions)
preds = plotResults(y_test, anomalyScoresAE, True)
```

Как показано на рис. 8.1, средняя точность равна `0.64`, что пока не может нас удовлетворить. Наилучшее значение средней точности, полученное с помощью обучения без учителя в главе 4, составило `0.69`, тогда как для системы

на основе обучения с учителем оно было равно `0.82`. Однако стоит учесть, что каждый тренировочный процесс будет приводить к разным результатам, поэтому в вашем случае результаты могут оказаться иными.

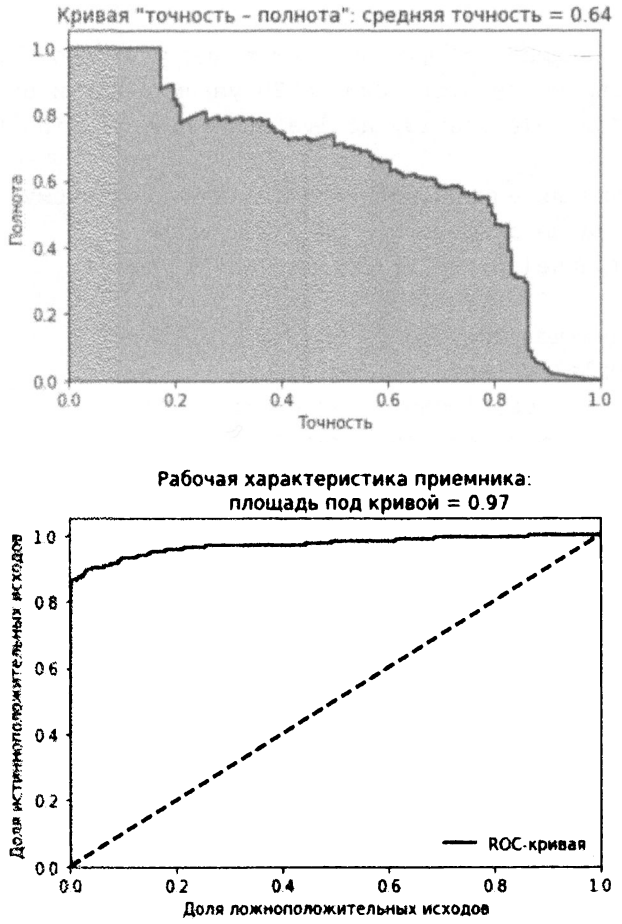


Рис. 8.1. *Оценочные метрики полного автокодировщика*

Чтобы получить более целостное представление о том, как нолный двухслойный автокодировщик проявляет себя на тестовом наборе, запустим тренировочный процесс 10 раз и сохраним среднее значение точности для каждого прогона. Мы поймем, насколько хорошо полный автокодировщик справляется с выявлением мошеннических транзакций, усреднив показатели средней точности на этих десяти прогонах.

Ниже приведен код программы, выполняющей 10 прогонов.

```
# 10 прогонов - мы будем определять усредненное
# значение средней точности
```

```
test_scores = []
for i in range(0, 10):
    # Вызов API-функции нейронной сети
    model = Sequential()

    # Применение линейной функции активации к входному слою;
    # генерирование скрытого слоя с 29 узлами, как и во входном слое
    model.add(Dense(units=29, activation='linear', input_dim=29))

    # Применение линейной функции активации к скрытому слою;
    # генерирование выходного слоя с 29 узлами
    model.add(Dense(units=29, activation='linear'))

    # Компиляция модели
    model.compile(optimizer='adam', \
                  loss='mean_squared_error', \
                  metrics=['accuracy'])

    # Тренировка модели
    num_epochs = 10
    batch_size = 32

    history = model.fit(x=X_train_AE, y=X_train_AE, \
        epochs=num_epochs, batch_size=batch_size, shuffle=True, \
        validation_data=(X_train_AE, X_train_AE), verbose=1)

    # Оценка на тестовом наборе
    predictions = model.predict(X_test, verbose=1)
    anomalyScoresAE = anomalyScores(X_test, predictions)
    preds, avgPrecision = plotResults(y_test, anomalyScoresAE, True)
    test_scores.append(avgPrecision)

print("Средняя точность, усредненная по 10 прогонам:", \
      np.mean(test_scores))
test_scores
```

Ниже подытожены результаты 10 прогонов. Усредненное значение средней точности равно 0.30, но сама средняя точность варьируется в пределах от 0.02 до 0.72. *Коэффициент вариации* (определенный как стандартное отклонение, деленное на среднее за 10 прогонов) равен 0.88.

```
Средняя точность, усредненная по 10 прогонам: 0.30108318944579776
Коэффициент вариации по 10 прогонам: 0.8755095071789248
```

```
[0.25468022666666157,
0.092705950994909,
0.716481644928299,
0.01946589342639965,
0.25623865457838263,
0.33597083510378234,
0.018757053070824415,
0.6188569405068724,
0.6720552647581304,
0.025619070873716072]
```

Попытаемся улучшить эти результаты, создавая различные варианты этого автокодировщика.

Двухслойный неполный автокодировщик с линейной функцией активации

Попробуем использовать вместо полного автокодировщика неполный.

Единственное, что изменится по сравнению с предыдущим примером, — это количество узлов в скрытом слое. Вместо того чтобы задавать его равным количеству оригинальных измерений (29), мы установим количество узлов равным 20. Другими словами, это ограниченный автокодировщик. Функция-кодировщик вынуждена извлекать информацию из входного слоя, ограничиваясь меньшим количеством узлов, а функция-декодировщик должна реконструировать оригинальную матрицу на основе этого нового представления.

В данном случае нам следует ожидать роста потерь по сравнению с полным автокодировщиком. Чтобы протестировать, насколько хорошо неполный автокодировщик справляется с выявлением мошеннических транзакций, мы выполним 10 независимых прогонов.

```
# Модель №2:
# двухслойный неполный автокодировщик с линейной функцией активации
# и 20 слоями в скрытом слое

# 10 прогонов - мы будем определять усредненное
# значение средней точности
test_scores = []
for i in range(0, 10):
```

```
    # Вызов API-функции нейронной сети
    model = Sequential()

    # Применение линейной функции активации к входному слою;
    # генерирование скрытого слоя с 20 узлами
    model.add(Dense(units=20, activation='linear', input_dim=29))

    # Применение линейной функции активации к скрытому слою;
    # генерирование выходного слоя с 29 узлами
    model.add(Dense(units=29, activation='linear'))

    # Компиляция модели
    model.compile(optimizer='adam', \
                  loss='mean_squared_error', \
                  metrics=['accuracy'])

    # Тренировка модели
    num_epochs = 10
    batch_size = 32

    history = model.fit(x=X_train_AE, y=X_train_AE, \
        epochs=num_epochs, batch_size=batch_size, shuffle=True, \
        validation_data=(X_train_AE, X_train_AE), verbose=1)

    # Оценка на тестовом наборе
    predictions = model.predict(X_test, verbose=1)
    anomalyScoresAE = anomalyScores(X_test, predictions)
    preds, avgPrecision = plotResults(y_test, anomalyScoresAE, True)
    test_scores.append(avgPrecision)

print("Средняя точность, усредненная по 10 прогонам:", \
      np.mean(test_scores))
test_scores
```

Как следует из приведенных ниже результатов, потери неполного автокодировщика оказались намного выше, чем в предыдущем случае. Это и неудивительно, ведь автокодировщик обучается новому ограниченному представлению, которое компактнее, чем оригинальная входная матрица. В такой ситуации он просто не способен запоминать входы.

```
Epoch 1/10
190820/190820 [==============================] - 28s 145us/step -
 loss: 0.3588 - acc: 0.5672 - val_loss: 0.2789 - val_acc: 0.6078
```

```
Epoch 2/10
190820/190820 [==============================] - 29s 153us/step -
 loss: 0.2817 - acc: 0.6032 - val_loss: 0.2757 - val_acc: 0.6115
Epoch 3/10
190820/190820 [==============================] - 28s 147us/step -
 loss: 0.2793 - acc: 0.6147 - val_loss: 0.2755 - val_acc: 0.6176
Epoch 4/10
190820/190820 [==============================] - 30s 155us/step -
 loss: 0.2784 - acc: 0.6164 - val_loss: 0.2750 - val_acc: 0.6167
Epoch 5/10
190820/190820 [==============================] - 29s 152us/step -
 loss: 0.2786 - acc: 0.6188 - val_loss: 0.2746 - val_acc: 0.6126
Epoch 6/10
190820/190820 [==============================] - 29s 151us/step -
 loss: 0.2776 - acc: 0.6140 - val_loss: 0.2752 - val_acc: 0.6043
Epoch 7/10
190820/190820 [==============================] - 30s 156us/step -
 loss: 0.2775 - acc: 0.5947 - val_loss: 0.2745 - val_acc: 0.5946
Epoch 8/10
190820/190820 [==============================] - 29s 149us/step -
 loss: 0.2770 - acc: 0.5903 - val_loss: 0.2740 - val_acc: 0.5882
Epoch 9/10
190820/190820 [==============================] - 29s 153us/step -
 loss: 0.2768 - acc: 0.5921 - val_loss: 0.2770 - val_acc: 0.5801
Epoch 10/10
190820/190820 [==============================] - 29s 150us/step -
 loss: 0.2767 - acc: 0.5803 - val_loss: 0.2744 - val_acc: 0.5743
93987/93987[==============================] - 3s 36us/step
```

Именно так и должен работать автокодировщик — он должен обучаться новому представлению. На рис. 8.2 показано, насколько эффективно это новое представление в отношении выявления фальсификаций.

Средняя точность равна `0.29`, что оказалось меньше, чем в случае полного автокодировщика.

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.31`, но дисперсия отличается высокой плотностью (на что указывает коэффициент вариации, равный `0.03`). Эта система значительно стабильнее, чем та, которая была построена на основе полного автокодировщика.

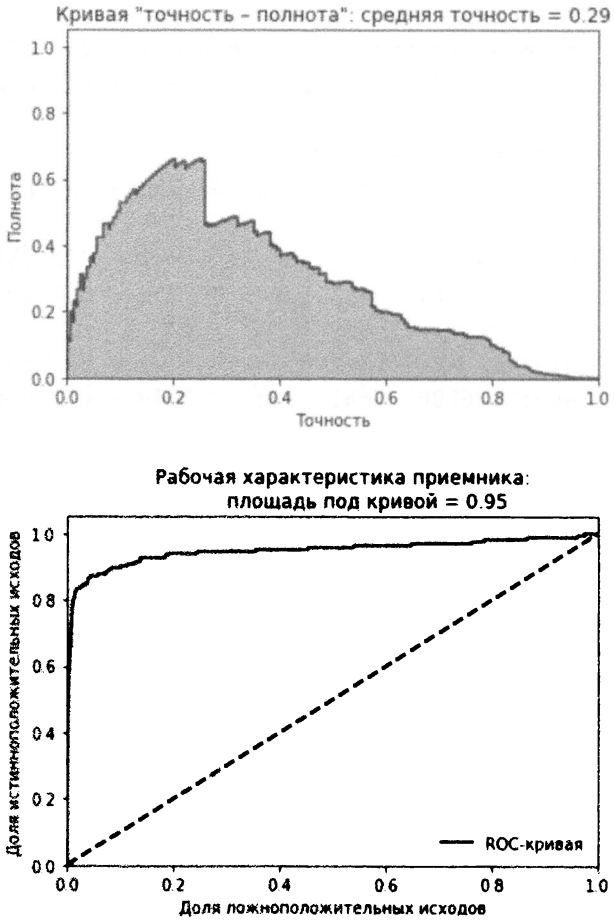


Рис. 8.2. *Оценочные метрики неполного автокодировщика с 20 узлами*

```
Средняя точность, усредненная по 10 прогонам: 0.30913783987972737
Коэффициент вариации по 10 прогонам: 0.032251659812254876

[0.2886910204920736,
0.3056142045082387,
0.31658073591381186,
0.30590858583039254,
0.31824197682595556,
0.3136952374067599,
0.30888135217515555,
0.31234000424933206,
0.29695149753706923,
0.3244746838584846]
```

Тем не менее мы остаемся на уровне довольно умеренной средней точности. В чем причина того, что неполный автокодировщик не смог отработать лучше? Возможно, ему просто оказалось недостаточно заданного количества узлов. А может быть, нам следовало тренировать сеть, используя большее количество скрытых слоев? Давайте поочередно испытаем оба варианта.

Увеличение количества узлов

Приведенные ниже результаты отражают тренировочные потери в случае использования двухслойного неполного автокодировщика с 27 узлами вместо 20.

```
Epoch 1/10
190820/190820 [==============================] - 29s 150us/step -
 loss: 0.1169 - acc: 0.8224 - val_loss: 0.0368 - val_acc: 0.8798
Epoch 2/10
190820/190820 [==============================] - 29s 154us/step -
 loss: 0.0388 - acc: 0.8610 - val_loss: 0.0360 - val_acc: 0.8530
Epoch 3/10
190820/190820 [==============================] - 30s 156us/step -
 loss: 0.0382 - acc: 0.8680 - val_loss: 0.0359 - val_acc: 0.8745
Epoch 4/10
190820/190820 [==============================] - 30s 156us/step -
 loss: 0.0371 - acc: 0.8811 - val_loss: 0.0353 - val_acc: 0.9021
Epoch 5/10
190820/190820 [==============================] - 30s 155us/step -
 loss: 0.0373 - acc: 0.9114 - val_loss: 0.0352 - val_acc: 0.9226
Epoch 6/10
190820/190820 [==============================] - 30s 155us/step -
 loss: 0.0377 - acc: 0.9361 - val_loss: 0.0370 - val_acc: 0.9416
Epoch 7/10
190820/190820 [==============================] - 30s 156us/step -
 loss: 0.0361 - acc: 0.9448 - val_loss: 0.0358 - val_acc: 0.9378
Epoch 8/10
190820/190820 [==============================] - 30s 156us/step -
 loss: 0.0354 - acc: 0.9521 - val_loss: 0.0350 - val_acc: 0.9503
Epoch 9/10
190820/190820 [==============================] - 29s 153us/step -
 loss: 0.0352 - acc: 0.9613 - val_loss: 0.0349 - val_acc: 0.9263
Epoch 10/10
190820/190820 [==============================] - 29s 153us/step -
 loss: 0.0353 - acc: 0.9566 - val_loss: 0.0343 - val_acc: 0.9477
93987/93987[==============================] - 4s 39us/step
```

Кривая "точность — полнота", значение средней точности и кривая auROC приведены на рис. 8.3.

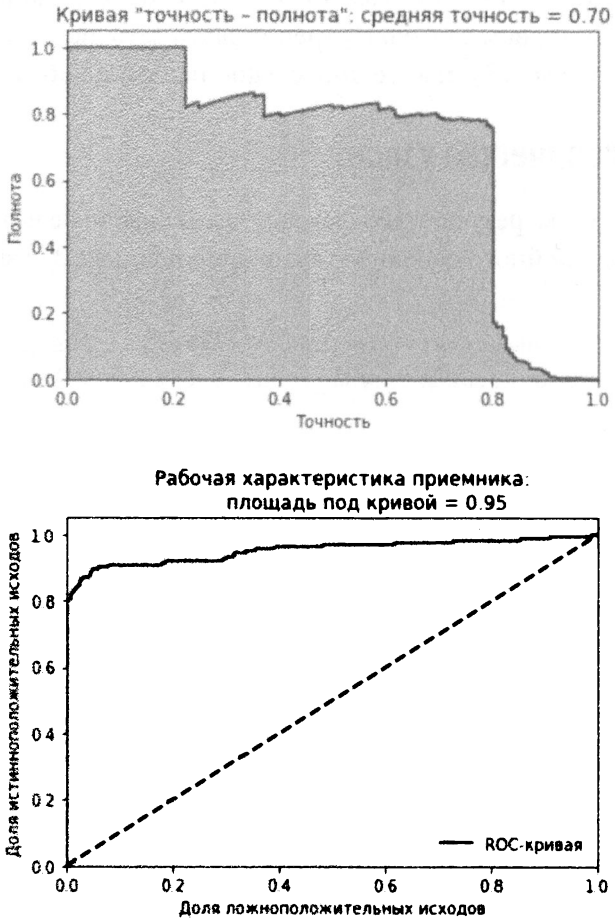


Рис. 8.3. *Оценочные метрики неполного автокодировщика с 27 узлами*

Средняя точность значительно улучшилась, достигнув уровня `0.70`. Это лучше, чем средняя точность полного автокодировщика, и превосходит результат наилучшего решения на основе обучения без учителя из главы 4.

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.53`, что значительно лучше достигнутой ранее средней точности, равной примерно `0.30`. Дисперсия средней точности более-менее разумна, так как коэффициент вариации равен `0.50`.

```
Средняя точность, усредненная по 10 прогонам: 0.5273341559141779
Коэффициент вариации по 10 прогонам: 0.5006880691999009
```

```
[0.689799495450694,
0.7092146840717755,
0.7336692377321005,
0.6154173765950426,
0.7068800243349335,
0.35250757724667586,
0.6904117414832501,
0.02335388808244066,
0.690798140588336,
0.061289393556529626]
```

Налицо явное улучшение по сравнению с предыдущей системой обнаружения аномалий.

Добавление дополнительных скрытых слоев

Проверим, сможем ли мы улучшить результаты, добавив в автокодировщик дополнительный скрытый слой. При этом мы по-прежнему используем линейные функции активации.

Выполнение экспериментов — ключевая часть процесса нахождения оптимальной архитектуры нейронной сети для решаемой задачи. Некоторые изменения будут приводить к улучшению результатов, некоторые — к ухудшению. Важно знать эффективные способы изменения нейронной сети и настройки ее гиперпараметров.

Вместо одиночного скрытого слоя с 27 узлами мы используем один скрытый слой с 28 узлами и еще один — с 27 узлами. Это лишь незначительное изменение программы по сравнению с предыдущим вариантом. Теперь у нас имеется трехслойная сеть, поскольку она содержит два скрытых слоя и один выходной. Напомним, что при подсчете количества слоев в нейронной сети входной слой не учитывается.

Учет дополнительного скрытого слоя требует добавления всего одной строки кода.

```
# Модель №3:
# трехслойный неполный автокодировщик с линейной функцией активации,
# содержащий 28 и 27 узлов в двух скрытых слоях

model = Sequential()
```

```
model.add(Dense(units=28, activation='linear', input_dim=29))
model.add(Dense(units=27, activation='linear'))
model.add(Dense(units=29, activation='linear'))
```

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.36`, что хуже полученного ранее значения `0.53`. Дисперсия средней точности также ухудшилась, поскольку коэффициент вариации равен `0.94` (чем выше, тем хуже).

```
Средняя точность, усредненная по 10 прогонам: 0.36075271075596366
Коэффициент вариации по 10 прогонам: 0.9361649046827353

[0.02259626054852924,
0.6984699403560997,
0.011035001202665167,
0.06621450000830197,
0.008916986608776182,
0.705399684020873,
0.6995233144849828,
0.008263068338243631,
0.6904537524978872,
0.6966545994932775]
```

Нелинейный автокодировщик

Теперь мы создадим неполный автокодировщик, применив нелинейную функцию активации. Мы используем функцию ReLU, но вы вправе поэкспериментировать с гиперболическим тангенсом, сигмоидой или любой другой нелинейной функцией активации.

Наша сеть будет включать три скрытых слоя, содержащих 27, 22 и 27 узлов. С концептуальной точки зрения первые две функции активации (применяемые ко входному и первому скрытому слою) служат кодировщиками, создавая второй скрытый слой с 22 узлами. Следующие две функции активации осуществляют декодирование, восстанавливая представление, насчитывающее 22 узла, до оригинального количества измерений, равного 29.

```
# Модель №4:
# четырехслойный неполный автокодировщик с функцией активации ReLU;
# 29 -> 27 -> 22 -> 27 -> 29

model = Sequential()
```

```
model.add(Dense(units=27, activation='relu', input_dim=29))
model.add(Dense(units=22, activation='relu'))
model.add(Dense(units=27, activation='relu'))
model.add(Dense(units=29, activation='relu'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.4 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 32s 169us/step -
 loss: 0.7010- acc: 0.5626 - val_loss: 0.6339 - val_acc: 0.6983
Epoch 2/10
190820/190820 [==============================] - 33s 174us/step -
 loss: 0.6302 - acc: 0.7132 - val_loss: 0.6219 - val_acc: 0.7465
Epoch 3/10
190820/190820 [==============================] - 34s 177us/step -
 loss: 0.6224 - acc: 0.7367 - val_loss: 0.6198 - val_acc: 0.7528
Epoch 4/10
190820/190820 [==============================] - 34s 179us/step -
 loss: 0.6227 - acc: 0.7380 - val_loss: 0.6205 - val_acc: 0.7471
Epoch 5/10
190820/190820 [==============================] - 33s 174us/step -
 loss: 0.6206 - acc: 0.7452 - val_loss: 0.6202 - val_acc: 0.7353
Epoch 6/10
190820/190820 [==============================] - 33s 175us/step -
 loss: 0.6206 - acc: 0.7458 - val_loss: 0.6192 - val_acc: 0.7485
Epoch 7/10
190820/190820 [==============================] - 33s 174us/step -
 loss: 0.6199 - acc: 0.7481 - val_loss: 0.6239 - val_acc: 0.7308
Epoch 8/10
190820/190820 [==============================] - 33s 175us/step -
 loss: 0.6203 - acc: 0.7497 - val_loss: 0.6183 - val_acc: 0.7626
Epoch 9/10
190820/190820 [==============================] - 34s 177us/step -
 loss: 0.6197 - acc: 0.7491 - val_loss: 0.6188 - val_acc: 0.7531
Epoch 10/10
190820/190820 [==============================] - 34s 177us/step -
 loss: 0.6201 - acc: 0.7486 - val_loss: 0.6188 - val_acc: 0.7540
93987/93987 [==============================] - 5s 48 us/step
```

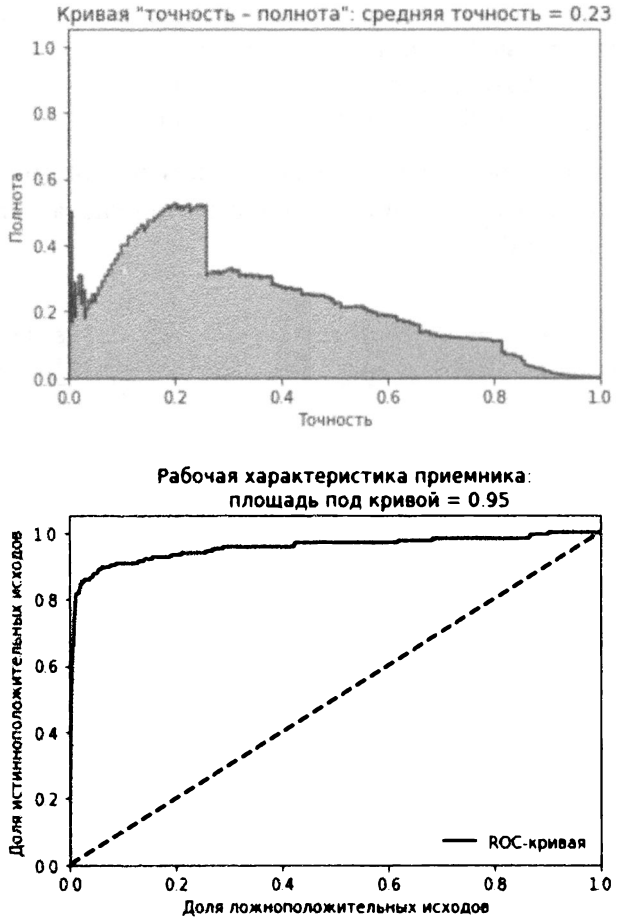


Рис. 8.4. *Оценочные метрики неполного автокодировщика с тремя скрытыми слоями и функцией активации ReLU*

Эти результаты значительно хуже предыдущих.

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.22`, что хуже полученного ранее значения `0.53`. Дисперсия средней точности отличается высокой плотностью, так как коэффициент вариации равен `0.06`.

```
Средняя точность, усредненная по 10 прогонам: 0.2232934196381843
Коэффициент вариации по 10 прогонам: 0.060779960264380296

[0.22598829389665595,
0.22616147166925166,
0.22119489753135715,
```

```
0.2478548473814437,
0.2251289336369011,
0.2119454446242229,
0.2126914064768752,
0.24581338950742185,
0.20665608837737512,
0.20949942328033827]
```

Эти результаты значительно хуже тех, которые были достигнуты с помощью простого автокодировщика с использованием линейной функции активации. Не исключено, что для нашего набора данных неполный линейный автокодировщик — наилучшее решение.

В случае других наборов данных ситуация может оказаться иной. Как всегда, поиск оптимального решения требует проведения экспериментов. Попробуйте поменять число узлов или количество скрытых слоев, используйте ансамбль функций активации и посмотрите, как это влияет на результаты: ухудшаются они или улучшаются.

Такого рода эксперименты называются *оптимизацией гиперпараметров*. В процессе поиска оптимального решения вы настраиваете гиперпараметры нейронной сети: количество узлов, количество слоев и ансамбль функций активации.

Сверхполный автокодировщик с линейной функцией активации

Вспомним о том, какая проблема присуща сверхполным автокодировщикам. В скрытом слое такого автокодировщика содержится больше узлов, чем во входном или выходном слое. Ввиду столь большой *емкости* нейронной сети автокодировщик просто запоминает наблюдения, на которых обучается.

Другими словами, автокодировщик обучается *тождественному отображению*, а это именно то, чего мы стремимся избежать. Автокодировщик будет переобучаться на тренировочных данных и плохо справляться с отделением мошеннических транзакций от нормальных.

Нам нужен автокодировщик, способный обучаться наиболее существенным аспектам операций с банковскими картами, чтобы он мог распознавать нормальные транзакции и не запоминать информацию, связанную с более редкими поддельными транзакциями.

Отличать мошеннические транзакции от нормальных может лишь автокодировщик, который способен терять часть информации, содержащейся в тренировочном наборе.

```
# Модель №5:
# двухслойный сверхполный автокодировщик
# с линейной функцией активации;
# 29 -> 40 -> 29

model = Sequential()
model.add(Dense(units=40, activation='linear', input_dim=29))
model.add(Dense(units=29, activation='linear'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.5 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 31s 161us/step -
 loss: 0.0498 - acc: 0.9438 - val_loss: 9.2301e-06 - val_acc: 0.9982
Epoch 2/10
190820/190820 [==============================] - 33s 171us/step -
 loss: 0.0014 - acc: 0.9925 - val_loss: 0.0019 - val_acc: 0.9909
Epoch 3/10
190820/190820 [==============================] - 33s 172us/step -
 loss: 7.6469 e-04 - acc: 0.9947 - val_loss: 4.5314e-05 - val_acc:
 0.9970
Epoch 4/10
190820/190820 [==============================] - 35s 182us/step -
 loss: 0.0010 - acc: 0.9930 - val_loss: 0.0039 - val_acc: 0.9859
Epoch 5/10
190820/190820 [==============================] - 32s 166us/step -
 loss: 0.0012 - acc: 0.9924 - val_loss: 8.5141e-04 - val_acc: 0.9886
Epoch 6/10
190820/190820 [==============================] - 31s 163us/step -
 loss: 5.0655 e-04 - acc: 0.9955 - val_loss: 8.2359e-04 - val_acc:
 0.9910
Epoch 7/10
190820/190820 [==============================] - 30s 156us/step -
 loss: 7.6046 e-04 - acc: 0.9930 - val_loss: 0.0045 - val_acc: 0.9933
Epoch 8/10
190820/190820 [==============================] - 30s 157us/step -
 loss: 9.1609 e-04 - acc: 0.9930 - val_loss: 7.3662e-04 - val_acc:
```

```
 0.9872
Epoch 9/10
190820/190820 [==============================] - 30s 158us/step -
 loss: 7.6287 e-04 - acc: 0.9929 - val_loss: 2.5671e-04 - val_acc:
 0.9940
Epoch 10/10
190820/190820 [==============================] - 30s 157us/step -
 loss: 7.0697 e-04 - acc: 0.9928 - val_loss: 4.5272e-06 - val_acc:
 0.9994
93987/93987[==============================] - 4s 48us/step
```

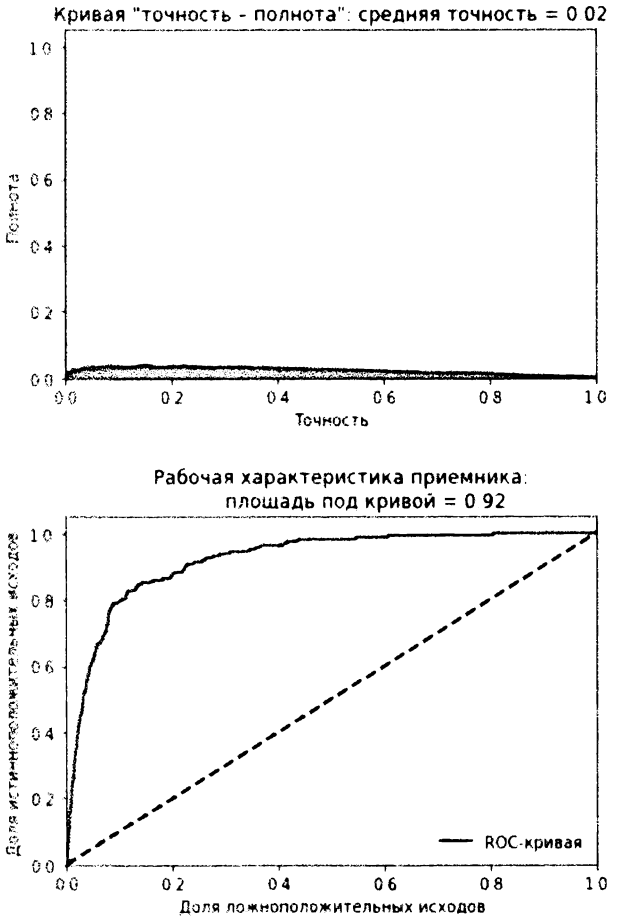


Рис. 8.5. *Оценочные метрики сверхполного автокодировщика с одним скрытым слоем и линейной функцией активации*

Как и ожидалось, потери оказались очень низкими, и переобученный сверхполный автокодировщик плохо справился с обнаружением мошеннических транзакций.

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.31`, что хуже полученного ранее значения `0.53`. Дисперсия средней точности не слишком уплотнена, и коэффициент вариации равен `0.89`.

```
Средняя точность, усредненная по 10 прогонам: 0.3061984081568074
Коэффициент вариации по 10 прогонам: 0.8896921668864564

[0.03394897465567298,
0.14322827274920255,
0.03610123178524601,
0.019735235731640446,
0.012571999125881402,
0.6788921569665146,
0.5411349583727725,
0.388474572258503,
0.7089617645810736,
0.4989349153415674]
```

Сверхполный автокодировщик с линейной функцией активации и дропаутом

Один из способов улучшения решения на основе сверхполного автокодировщика заключается в применении регуляризации для уменьшения эффектов переобучения. Эффективная методика регуляризации — *дропаут*, или *исключение*. С помощью дропаута мы заставляем автокодировщик исключить заданный процент элементов из слоев нейронной сети.

При наличии такого ограничения сверхполный автокодировщик не сможет запоминать данные транзакций, хранящиеся в нашем наборе. Вместо этого ему придется создавать обобщения. Он будет вынужден обучаться большему количеству существенных признаков и отбрасывать менее существенную информацию.

Мы будем исключать посредством дропаута 10% узлов скрытого слоя. Другими словами, будут исключаться 10% нейронов. Чем выше процент, тем сильнее регуляризация. Это делается с помощью одной-единственной строки кода.

Проверим, улучшает ли это результаты.

```
# Модель №6:
# двухслойный сверхполный автокодировщик
# с линейной функцией активации и дропаутом;
# 29 -> 40 -> 29
# дропаут 10%

model = Sequential()
model.add(Dense(units=40, activation='linear', input_dim=29))
model.add(Dropout(0.10))
model.add(Dense(units=29, activation='linear'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.6 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 27s 141us/step -
 loss: 0.1358 - acc: 0.7430 - val_loss: 0.0082 - val_acc: 0.9742
Epoch 2/10
190820/190820 [==============================] - 28s 146us/step -
 loss: 0.0782 - acc: 0.7849 - val_loss: 0.0094 - val_acc: 0.9689
Epoch 3/10
190820/190820 [==============================] - 28s 149us/step -
 loss: 0.0753 - acc: 0.7858 - val_loss: 0.0102 - val_acc: 0.9672
Epoch 4/10
190820/190820 [==============================] - 28s 148us/step -
 loss: 0.0772 - acc: 0.7864 - val_loss: 0.0093 - val_acc: 0.9677
Epoch 5/10
190820/190820 [==============================] - 28s 147us/step -
 loss: 0.0813 - acc: 0.7843 - val_loss: 0.0108 - val_acc: 0.9631
Epoch 6/10
190820/190820 [==============================] - 28s 149us/step -
 loss: 0.0756 - acc: 0.7844 - val_loss: 0.0095 - val_acc: 0.9654
Epoch 7/10
190820/190820 [==============================] - 29s 150us/step -
 loss: 0.0743 - acc: 0.7850 - val_loss: 0.0077 - val_acc: 0.9768
Epoch 8/10
190820/190820 [==============================] - 29s 150us/step -
 loss: 0.0767 - acc: 0.7840 - val_loss: 0.0070 - val_acc: 0.9759
Epoch 9/10
190820/190820 [==============================] - 29s 150us/step -
```

```
 loss: 0.0762 - acc: 0.7851 - val_loss: 0.0072 - val_acc: 0.9733
Epoch 10/10
190820/190820 [==============================] - 29s 151us/step -
 loss: 0.0756 - acc: 0.7849 - val_loss: 0.0067 - val_acc: 0.9749
93987/93987 [==============================] - 3s 32us/step
```

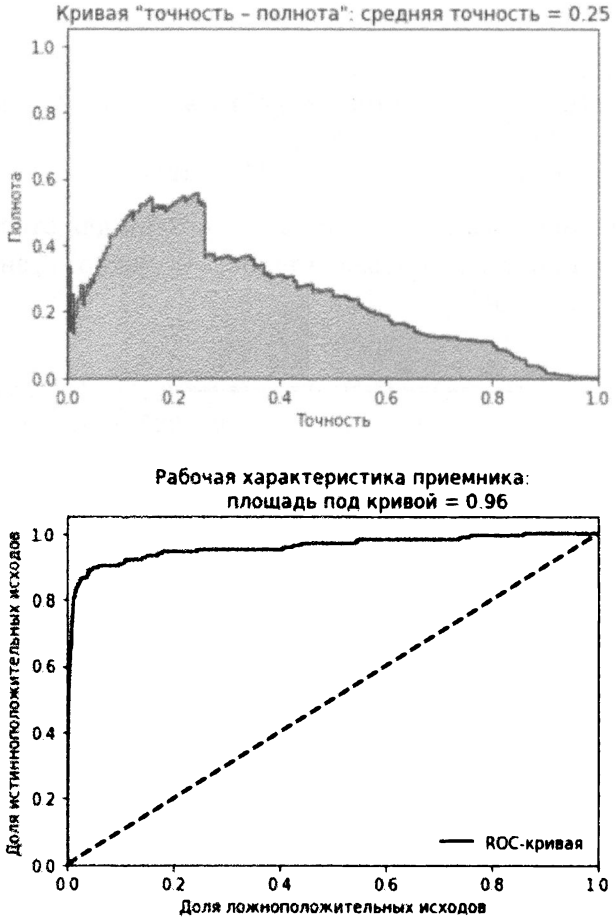


Рис. 8.6. *Оценочные метрики сверхполного автокодировщика с одним скрытым слоем, дропаутом и линейной функцией активации*

Как и ожидалось, потери оказались очень низкими, и переобученный сверхполный автокодировщик плохо справился с обнаружением мошеннических транзакций с банковскими картами.

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.21`, что хуже полученного ранее значения `0.53`. Коэффициент вариации равен `0.40`.

```
Средняя точность, усредненная по 10 прогонам: 0.21150415381770646
Коэффициент вариации по 10 прогонам: 0.40295807771579256

[0.22549974304927337,
0.22451178120391296,
0.17243952488912334,
0.2533716906936315,
0.13251890273915556,
0.1775116247503748,
0.4343283958332979,
0.10469065867732033,
0.19480068075466764,
0.19537213558630712]
```

Разреженный сверхполный автокодировщик с линейной функцией активации

Еще одна эффективная методика регуляризации — *разреженность*. Мы можем заставить автокодировщик учитывать разреженность матрицы, чтобы большая часть нейронов оставалась неактивной большую часть времени. Это снижает вероятность тождественного отображения даже в случае сверхполного автокодировщика, поскольку большинство узлов отключено, а значит, автокодировщику сложнее переобучиться.

Мы используем в сверхполном автокодировщике всего лишь один скрытый слой, включив в него, как и ранее, 40 узлов, но вместо дропаута применим штрафы за разреженность.

Проверим, позволит ли это улучшить достигнутое ранее усредненное значение средней точности, равное `0.21`.

```
# Модель №7:
# двухслойный разреженный сверхполный автокодировщик
# с линейной функцией активации;
# 29 -> 40 -> 29

model = Sequential()
    model.add(Dense(units=40, activation='linear', \
        activity_regularizer=regularizers.l1(10e-5), input_dim=29))
model.add(Dense(units=29, activation='linear'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.7 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

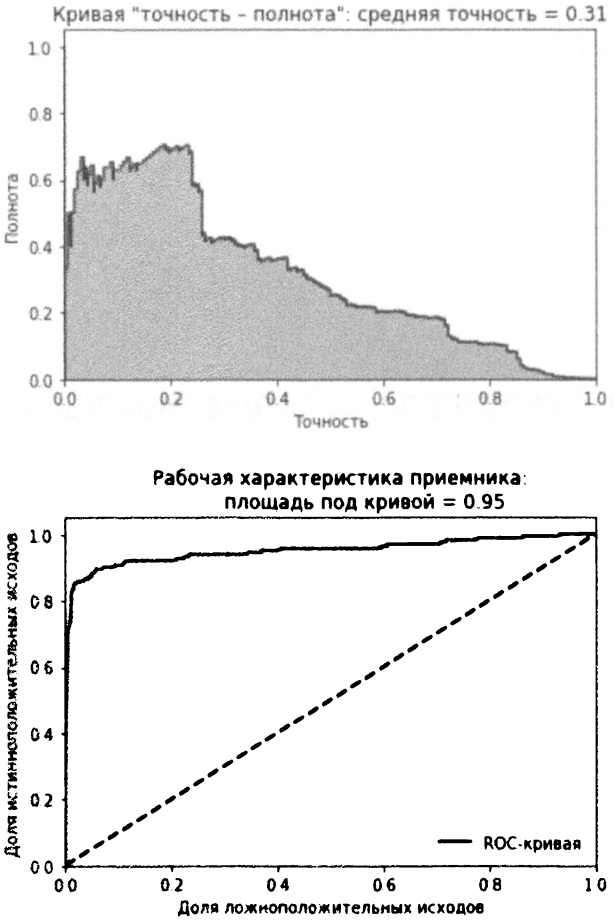


Рис. 8.7. *Оценочные метрики разреженного сверхполного автокодировщика с одним скрытым слоем и линейной функцией активации*

```
Epoch 1/10
190820/190820 [==============================] - 27s 142us/step -
 loss: 0.0985 - acc: 0.9380 - val_loss: 0.0369 - val_acc: 0.9871
Epoch 2/10
190820/190820 [==============================] - 26s 136us/step -
 loss: 0.0284 - acc: 0.9829 - val_loss: 0.0261 - val_acc: 0.9698
Epoch 3/10
190820/190820 [==============================] - 26s 136us/step -
```

```
 loss: 0.0229 - acc: 0.9816 - val_loss: 0.0169 - val_acc: 0.9952
Epoch 4/10
190820/190820 [==============================] - 26s 137us/step -
 loss: 0.0201 - acc: 0.9821 - val_loss: 0.0147 - val_acc: 0.9943
Epoch 5/10
190820/190820 [==============================] - 26s 137us/step -
 loss: 0.0183 - acc: 0.9810 - val_loss: 0.0142 - val_acc: 0.9842
Epoch 6/10
190820/190820 [==============================] - 26s 137us/step -
 loss: 0.0206 - acc: 0.9774 - val_loss: 0.0158 - val_acc: 0.9906
Epoch 7/10
190820/190820 [==============================] - 26s 136us/step -
 loss: 0.0169 - acc: 0.9816 - val_loss: 0.0124 - val_acc: 0.9866
Epoch 8/10
190820/190820 [==============================] - 26s 137us/step -
 loss: 0.0165 - acc: 0.9795 - val_loss: 0.0208 - val_acc: 0.9537
Epoch 9/10
190820/190820 [==============================] - 26s 136us/step -
 loss: 0.0164 - acc: 0.9801 - val_loss: 0.0105 - val_acc: 0.9965
Epoch 10/10
190820/190820 [==============================] - 27s 140us/step -
 loss: 0.0167 - acc: 0.9779 - val_loss: 0.0102 - val_acc: 0.9955
93987/93987 [==============================] - 3s 32us/step
```

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.21`, что хуже полученного ранее значения `0.53`. Коэффициент вариации равен `0.99`.

```
Средняя точность, усредненная по 10 прогонам: 0.21373659011504448
Коэффициент вариации по 10 прогонам: 0.9913040763536749

[0.1370972172100049,
0.28328895710699215,
0.6362677613798704,
0.3467265637372019,
0.5197889253491589,
0.01871495737323161,
0.0812609121251577,
0.034749761900336684,
0.04846036143317335,
0.031010483535317393]
```

Разреженный сверхполный автокодировщик с линейной функцией активации и дропаутом

Разумеется, ничто не мешает нам комбинировать методики регуляризации для улучшения решения. В данном случае мы используем разреженный сверхполный автокодировщик с линейной функцией активации, 40 узлами в единственном скрытом слое и дропаутом 5%.

```
# Модель №8:
# двухслойный разреженный сверхполный автокодировщик
# с линейной функцией активации и дропаутом;
# 29 -> 40 -> 29
# дропаут 5%

model = Sequential()
    model.add(Dense(units=40, activation='linear', \
        activity_regularizer=regularizers.l1(10e-5), input_dim=29))
    model.add(Dropout(0.05))
model.add(Dense(units=29, activation='linear'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.8 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 31s 162us/step -
 loss: 0.1477 - acc: 0.8150 - val_loss: 0.0506 - val_acc: 0.9727
Epoch 2/10
190820/190820 [==============================] - 29s 154us/step -
 loss: 0.0756 - acc: 0.8625 - val_loss: 0.0344 - val_acc: 0.9788
Epoch 3/10
190820/190820 [==============================] - 29s 152us/step -
 loss: 0.0687 - acc: 0.8612 - val_loss: 0.0291 - val_acc: 0.9790
Epoch 4/10
190820/190820 [==============================] - 29s 154us/step -
 loss: 0.0644 - acc: 0.8606 - val_loss: 0.0274 - val_acc: 0.9734
Epoch 5/10
190820/190820 [==============================] - 31s 163us/step -
 loss: 0.0630 - acc: 0.8597 - val_loss: 0.0242 - val_acc: 0.9746
Epoch 6/10
190820/190820 [==============================] - 31s 162us/step -
 loss: 0.0609 - acc: 0.8600 - val_loss: 0.0220 - val_acc: 0.9800
Epoch 7/10
```

```
190820/190820 [==============================] - 30s 156us/step -
 loss: 0.0624 - acc: 0.8581 - val_loss: 0.0289 - val_acc: 0.9633
Epoch 8/10
190820/190820 [==============================] - 29s 154us/step -
loss: 0.0589 - acc: 0.8588 - val_loss: 0.0574 - val_acc: 0.9366
Epoch 9/10
190820/190820 [==============================] - 29s 154us/step -
 loss: 0.0596 - acc: 0.8571 - val_loss: 0.0206 - val_acc: 0.9752
Epoch 10/10
190820/190820 [==============================] - 31s 165us/step -
 loss: 0.0593 - acc: 0.8590 - val_loss: 0.0204 - val_acc: 0.9808
93987/93987 [==============================] - 4s 38us/step
```

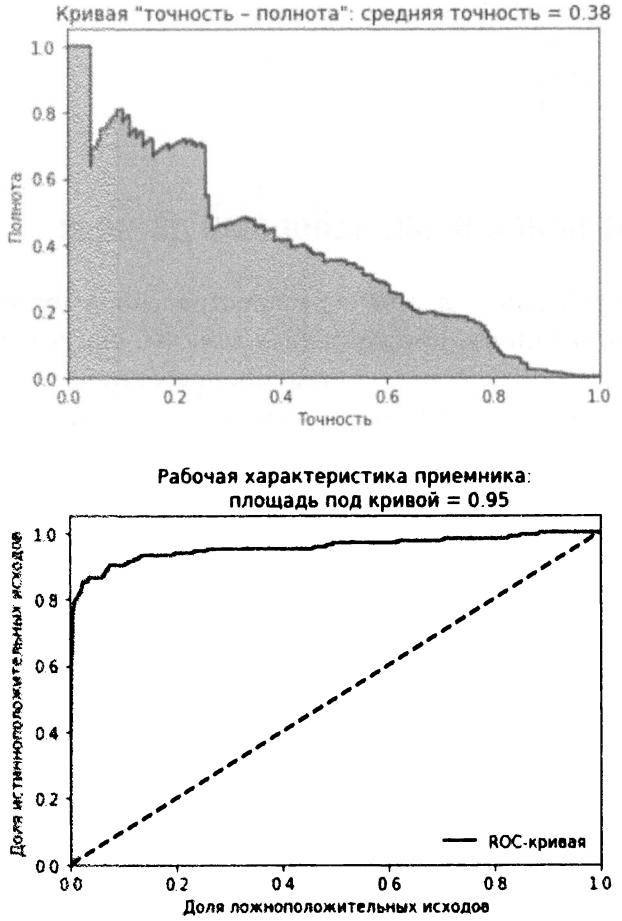


Рис. 8.8. *Оценочные метрики разреженного сверхполного автокодировщика с одним скрытым слоем, дропаутом и линейной функцией активации*

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно 0.24, что хуже полученного ранее значения 0.53. Коэффициент вариации равен 0.62.

```
Средняя точность, усредненная по 10 прогонам: 0.2426994231628755
Коэффициент вариации по 10 прогонам: 0.6153219870606188

[0.6078198313533932,
0.20862366991302814,
0.25854513247057875,
0.08496595007072019,
0.26313491674585093,
0.17001322998258625,
0.15338215561753896,
0.1439107390306835,
0.4073422280287587,
0.1292563784156162]
```

Работа с зашумленными наборами данных

При работе с реальными данными распространенной проблемой становится зашумленность данных, обусловленная тем, что они были так или иначе искажены в процессе извлечения, миграции, преобразования и т.п. Нам нужны автокодировщики, достаточно устойчивые к воздействию шумов, чтобы ничто не могло сбить их с толку и они могли обучаться действительно важной базовой структуре данных.

Сымитируем шум, добавив в наш набор данных гауссовскую случайную матрицу шума, и обучим автокодировщик на этом зашумленном тренировочном наборе. После этого мы проанализируем, насколько хорошо автокодировщику удается предсказывать поддельные транзакции в зашумленном тестовом наборе.

```
noise_factor = 0.50
X_train_AE_noisy = X_train_AE.copy() + noise_factor * \
    np.random.normal(loc=0.0, scale=1.0, size=X_train_AE.shape)
X_test_AE_noisy = X_test_AE.copy() + noise_factor * \
    np.random.normal(loc=0.0, scale=1.0, size=X_test_AE.shape)
```

Шумоподавляющий автокодировщик

Размер штрафа за переобучение на зашумленном наборе транзакционных данных значительно больше, чем в случае оригинальных, неискаженных данных. Набор данных содержит достаточное количество шума, поэтому автокодировщик, чересчур хорошо приспособившийся к зашумленным данным, не сможет отличать мошеннические транзакции от нормальных.

Нам нужен автокодировщик, обученный таким образом, чтобы он был способен достаточно хорошо реконструировать большую часть наблюдений и при этом случайно не реконструировать также шум. Другими словами, мы хотим, чтобы автокодировщик обучился базовой структуре данных, но забыл о содержащемся в них шуме.

Мы опробуем некоторые из вариантов, которые ранее продемонстрировали неплохую производительность. Сначала мы протестируем неполный автокодировщик с одним скрытым слоем с 27 узлами и линейной функцией активации. Затем мы проверим, как работает разреженный сверхполный автокодировщик с одним скрытым слоем с 40 узлами и дропаутом. В завершение мы используем автокодировщик с нелинейной функцией активации.

Двухслойный шумоподавляющий неполный автокодировщик с линейной функцией активации

На исходном наборе данных автокодировщик с одним скрытым слоем с 27 узлами и линейной функцией активации дал среднюю точность, равную `0.7`. Проверим, насколько хорошо он работает с зашумленным набором. Автокодировщик, который пытается избавиться от шума, называется *шумоподавляющим*, или *обесшумливающим* (denoising autoencoder).

Приведенный ниже код аналогичен тому, с которым мы работали ранее, но теперь мы применим его к зашумленным тренировочному и тестовому наборам данных, `X_train_AE_noisy` и `X_test_AE_noisy` соответственно.

```
# Модель №9:
# двухслойный шумоподавляющий неполный автокодировщик
# с линейной функцией активации
# 29 -> 27 -> 29

for i in range(0, 10):
    # Вызов API-функции нейронной сети
    model = Sequential()
```

```
# Генерирование скрытого слоя с 27 узлами
# и линейной функцией активации
model.add(Dense(units=27, activation='linear', input_dim=29))

# Генерирование выходного слоя с 29 узлами
model.add(Dense(units=29, activation='linear'))

# Компиляция модели
model.compile(optimizer='adam', \
              loss='mean_squared_error', \
              metrics=['accuracy'])

# Тренировка модели
num_epochs = 10
batch_size = 32

history = model.fit(x=X_train_AE_noisy, y=X_train_AE_noisy, \
    epochs=num_epochs, batch_size=batch_size, shuffle=True, \
    validation_data=(X_train_AE, X_train_AE), verbose=1)

# Оценка на тестовом наборе
predictions = model.predict(X_test_AE_noisy, verbose=1)
anomalyScoresAE = anomalyScores(X_test, predictions)
preds, avgPrecision = plotResults(y_test, anomalyScoresAE, True)
test_scores.append(avgPrecision)
model.reset_states()

print("Средняя точность, усредненная по 10 прогонам:", \
      np.mean(test_scores))
test_scores
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.9 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 25s 133us/step -
 loss: 0.1733 - acc: 0.7756 - val_loss: 0.0356 - val_acc: 0.9123
Epoch 2/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0546 - acc: 0.8793 - val_loss: 0.0354 - val_acc: 0.8973
Epoch 3/10
```

```
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0531 - acc: 0.8764 - val_loss: 0.0350 - val_acc: 0.9399
Epoch 4/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0525 - acc: 0.8879 - val_loss: 0.0342 - val_acc: 0.9573
Epoch 5/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0530 - acc: 0.8910 - val_loss: 0.0347 - val_acc: 0.9503
Epoch 6/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0524 - acc: 0.8889 - val_loss: 0.0350 - val_acc: 0.9138
Epoch 7/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0531 - acc: 0.8845 - val_loss: 0.0343 - val_acc: 0.9280
Epoch 8/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0530 - acc: 0.8798 - val_loss: 0.0339 - val_acc: 0.9507
Epoch 9/10
190820/190820 [==============================] - 24s 126us/step -
 loss: 0.0526 - acc: 0.8877 - val_loss: 0.0337 - val_acc: 0.9611
Epoch 10/10
190820/190820 [==============================] - 24s 127us/step -
 loss: 0.0528 - acc: 0.8885 - val_loss: 0.0352 - val_acc: 0.9474
93987/93987 [==============================] - 3s 34us/step
```

Теперь усредненное значение средней точности равно `0.28`. Как видите, линейному автокодировщику непросто подавить шум в наборе данных.

```
Средняя точность, усредненная по 10 прогонам: 0.2825997155005206
Коэффициент вариации по 10 прогонам: 1.1765416185187383

[0.6929639885685303,
0.008450118408150287,
0.6970753417267612,
0.011820311633718597,
0.008924124892696377,
0.010639537507746342,
0.6884911855668772,
0.006549332886020607,
0.6805304226634528,
0.02055279115125298]
```

Этому автокодировщику трудно справиться с отделением истинной базовой структуры данных от добавленного нами гауссовского шума.

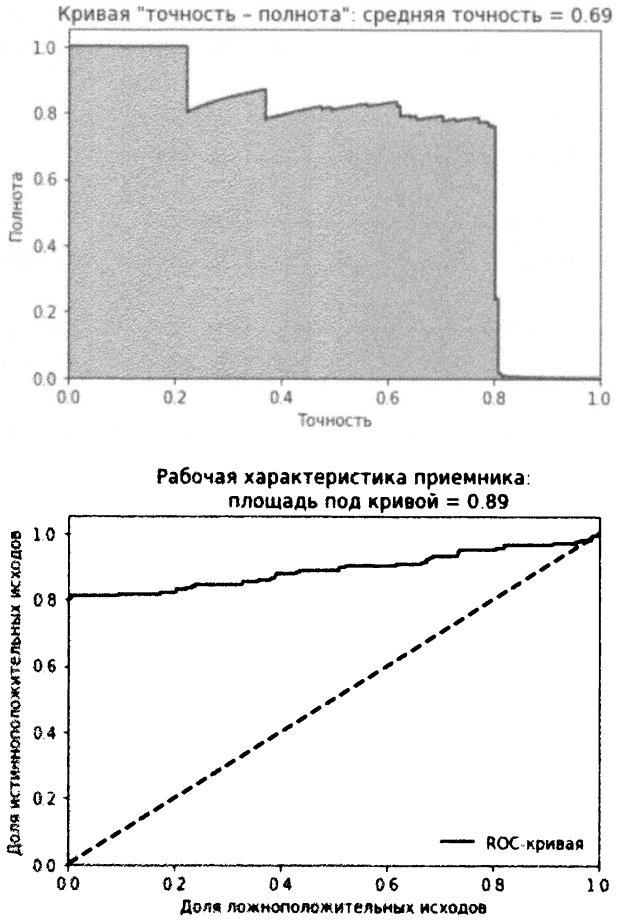


Рис. 8.9. *Оценочные метрики шумоподавляющего сверхполного автокодировщика с одним скрытым слоем и линейной функцией активации*

Двухслойный шумоподавляющий сверхполный автокодировщик с линейной функцией активации, разреженностью и дропаутом

Используем теперь сверхполный автокодировщик с одним скрытым слоем с 40 узлами, регуляризатором на основе разреженности и дропаутом 5%.

На оригинальном наборе данных этот автокодировщик дал среднюю точность `0.38`.

```
# Модель №10:
# двухслойный шумоподавляющий сверхполный автокодировщик
# с линейной функцией активации, регуляризатором на основе
# разреженности и дропаутом;
# 29 -> 40 -> 29
# дропаут 5%

model = Sequential()
model.add(Dense(units=40, activation='linear',
    activity_regularizer=regularizers.l1(10e-5), input_dim=29))
model.add(Dropout(0.05))
model.add(Dense(units=29, activation='linear'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.10 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 28s 145us/step -
 loss: 0.1726 - acc: 0.8035 - val_loss: 0.0432 - val_acc: 0.9781
Epoch 2/10
190820/190820 [==============================] - 26s 138us/step -
 loss: 0.0868 - acc: 0.8490 - val_loss: 0.0307 - val_acc: 0.9775
Epoch 3/10
190820/190820 [==============================] - 26s 138us/step -
 loss: 0.0809 - acc: 0.8455 - val_loss: 0.0445 - val_acc: 0.9535
Epoch 4/10
190820/190820 [==============================] - 26s 138us/step -
 loss: 0.0777 - acc: 0.8438 - val_loss: 0.0257 - val_acc: 0.9709
Epoch 5/10
190820/190820 [==============================] - 27s 139us/step -
 loss: 0.0748 - acc: 0.8434 - val_loss: 0.0219 - val_acc: 0.9787
Epoch 6/10
190820/190820 [==============================] - 26s 138us/step -
 loss: 0.0746 - acc: 0.8425 - val_loss: 0.0210 - val_acc: 0.9794
Epoch 7/10
190820/190820 [==============================] - 26s 138us/step -
 loss: 0.0713 - acc: 0.8437 - val_loss: 0.0294 - val_acc: 0.9503
Epoch 8/10
190820/190820 [==============================] - 26s 138us/step -
 loss: 0.0708 - acc: 0.8426 - val_loss: 0.0276 - val_acc: 0.9606
Epoch 9/10
190820/190820 [==============================] - 26s 139us/step -
```

```
 loss: 0.0704 - acc: 0.8428 - val_loss: 0.0180 - val_acc: 0.9811
Epoch 10/10
190820/190820 [==============================] - 27s 139us/step -
 loss: 0.0702 - acc: 0.8424 - val_loss: 0.0185 - val_acc: 0.9710
93987/93987 [==============================] - 4s 38us/step
```

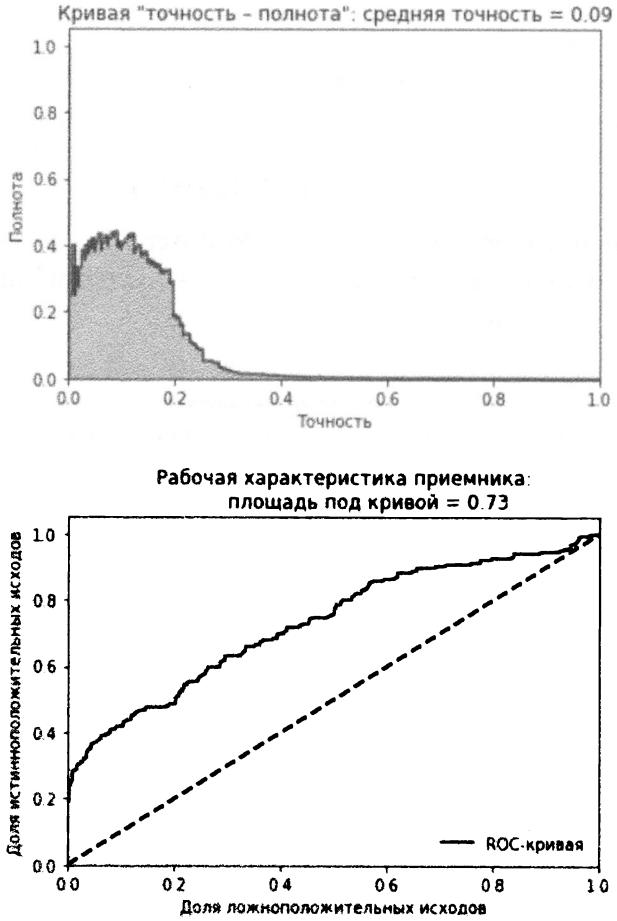


Рис. 8.10. *Оценочные метрики шумоподавляющего сверхполного автокодировщика с дропаутом и линейной функцией активации*

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.10`, что хуже полученного ранее значения `0.53`. Коэффициент вариации равен `0.83`.

```
Средняя точность, усредненная по 10 прогонам: 0.10112931070692295
Коэффициент вариации по 10 прогонам: 0.8343774832756188
```

```
[0.08283546387140524,
0.043070120657586454,
0.018901753737287603,
0.02381040174486509,
0.16038446580196433,
0.03461061251209459,
0.17847771715513427,
0.2483282420447288,
0.012981344347664117,
0.20789298519649893]
```

Двухслойный шумоподавляющий сверхполный автокодировщик с функцией активации ReLU

Проверим, как сработает этот же автокодировщик, но с использованием функции ReLU вместо линейной функции активации. Вспомните, что на оригинальном наборе данных автокодировщик с нелинейной функцией активации продемонстрировал худшую производительность, чем автокодировщик с линейной функцией.

```
# Модель №11:
# двухслойный шумоподавляющий сверхполный автокодировщик
# с функцией активации ReLU, регуляризатором на основе
# разреженности и дропаутом;
# 29 -> 40 -> 29
# дропаут 5%

model = Sequential()
    model.add(Dense(units=40, activation='relu', \
        activity_regularizer=regularizers.l1(10e-5), input_dim=29))
    model.add(Dropout(0.05))
model.add(Dense(units=29, activation='relu'))
```

Приведенные ниже результаты отражают потери для этого автокодировщика. На рис. 8.11 приведены кривая "точность — полнота", значение средней точности и кривая auROC.

```
Epoch 1/10
190820/190820 [==============================] - 29s 153us/step -
 loss: 0.3049 - acc: 0.6454 - val_loss: 0.0841 - val_acc: 0.8873
Epoch 2/10
190820/190820 [==============================] - 27s 143us/step -
```

```
 loss: 0.1806 - acc: 0.7193 - val_loss: 0.0606 - val_acc: 0.9012
Epoch 3/10
190820/190820 [==============================] - 27s 143us/step -
 loss: 0.1626 - acc: 0.7255 - val_loss: 0.0500 - val_acc: 0.9045
Epoch 4/10
190820/190820 [==============================] - 27s 143us/step -
 loss: 0.1567 - acc: 0.7294 - val_loss: 0.0445 - val_acc: 0.9116
Epoch 5/10
190820/190820 [==============================] - 27s 143us/step -
 loss: 0.1484 - acc: 0.7309 - val_loss: 0.0433 - val_acc: 0.9136
Epoch 6/10
190820/190820 [==============================] - 27s 144us/step -
 loss: 0.1467 - acc: 0.7311 - val_loss: 0.0375 - val_acc: 0.9101
Epoch 7/10
190820/190820 [==============================] - 27s 143us/step -
 loss: 0.1427 - acc: 0.7335 - val_loss: 0.0384  val_acc: 0.9013
Epoch 8/10
190820/190820 [==============================] - 27s 143us/step -
 loss: 0.1397 - acc: 0.7307 - val_loss: 0.0337 - val_acc: 0.9145
Epoch 9/10
190820/190820 [==============================] - 27s 143us/step -
 loss: 0.1361 - acc: 0.7322 - val_loss: 0.0343 - val_acc: 0.9066
Epoch 10/10
190820/190820 [==============================] - 27s 144us/step -
 loss: 0.1349 - acc: 0.7331 - val_loss: 0.0325 - val_acc: 0.9107
93987/93987 [==============================] - 4s 41us/step
```

Приведенная ниже сводка отражает распределение средней точности по 10 прогонам. Усредненное значение средней точности равно `0.20`, что хуже полученного ранее значения `0.53`. Коэффициент вариации равен `0.55`.

```
Средняя точность, усредненная по 10 прогонам: 0.1969608394689088
Коэффициент вариации по 10 прогонам: 0.5566706365802669

[0.22960316854089222,
0.37609633487223315,
0.11429775486529765,
0.10208135698072755,
0.4002384343852861,
0.13317480663248088,
0.15764518571284625,
0.2406315655171392,
0.05080529996343734,
0.1650344872187474]
```

Вы сможете самостоятельно поэкспериментировать, изменяя количество узлов и слоев, а также степень разреженности, процент дропаута и функции активации, чтобы проверить, удастся ли улучшить эти результаты.

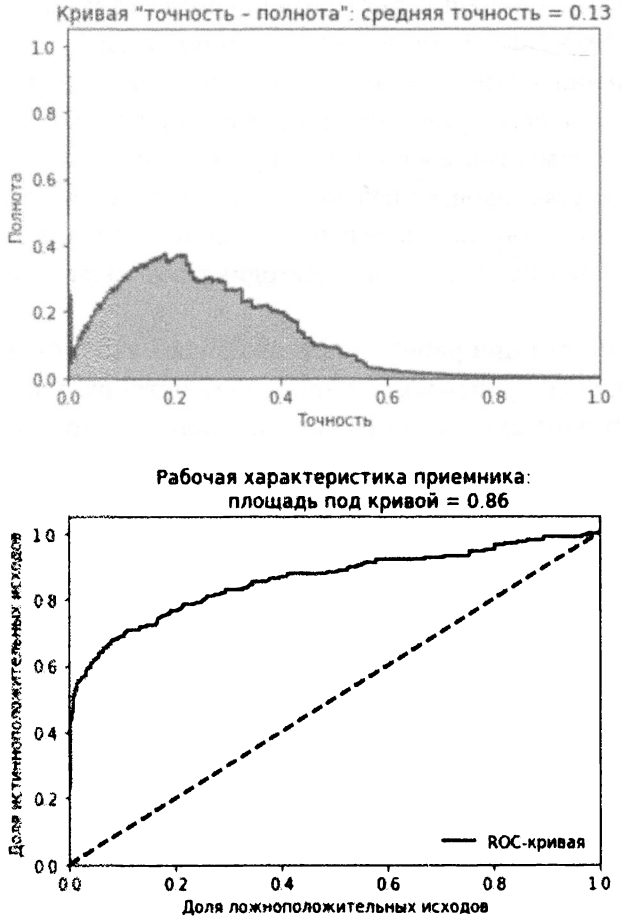


Рис. 8.11. *Оценочные метрики шумоподавляющего сверхполного автокодировщика с дропаутом и функцией активации ReLU*

Резюме

В этой главе мы вернулись к задаче обнаружения мошеннических операций с банковскими картами и разработали соответствующее решение на основе обучения без учителя с помощью нейронной сети.

Для нахождения оптимальной архитектуры нейронной сети мы провели ряд экспериментов с привлечением автокодировщиков различного типа. В ходе этих экспериментов мы проверили работу полного, неполного и сверхполного автокодировщиков, включающих один или несколько скрытых слоев. При этом мы использовали как линейную, так и нелинейную функции активации с применением двух основных методик регуляризации: разреженности и дропаута.

Мы выяснили, что при работе с оригинальным набором транзакционных данных наилучшую производительность демонстрирует довольно простая двухслойная нейронная сеть на основе неполного автокодировщика с линейной функцией активации, но для того чтобы справиться с шумом в зашумленном наборе данных, нам пришлось использовать разреженный двухслойный сверхполный автокодировщик с дропаутом.

Многие наши эксперименты были проведены методом проб и ошибок: в каждом эксперименте мы подстраивали гиперпараметры и сравнивали результаты с предыдущими итерациями. Вполне возможно, что для обнаружения мошеннических операций с банковскими картами существует еще более эффективное решение на основе автокодировщика, и я призываю вас выполнить собственные эксперименты, чтобы это проверить.

До сих пор мы рассматривали обучение с учителем и обучение без учителя как два независимых направления машинного обучения, но в главе 9 мы исследуем совместное использование двух моделей для разработки приложения на основе обучения с частичным привлечением учителя, которое позволяет добиться лучших результатов, чем любой из подходов по отдельности.

Обучение с частичным привлечением учителя

До сих пор мы рассматривали обучение с учителем и обучение без учителя как два разных направления машинного обучения. Обучение с учителем уместно использовать в случае помеченных наборов данных, а обучение без учителя — когда набор данных не размечен.

Однако на практике это различие не настолько явное. Обычно наборы данных частично размечены, и мы хотим эффективно пометить неразмеченные наблюдения, используя информацию, содержащуюся в размеченной части набора. В случае обучения с учителем нам пришлось бы проигнорировать большую часть набора данных, поскольку она неразмечена. В случае обучения без учителя мы работали бы с этими данными, но не знали бы, как воспользоваться преимуществами доступных меток.

Обучение с частичным привлечением учителя (*полуавтоматическое обучение*) берет лучшее из двух миров, что позволяет применять имеющиеся метки для раскрытия внутренней структуры набора и разметки остальной его части.

В этой главе мы продолжим использовать набор данных об операциях с банковскими картами и продемонстрируем на нем, как работает обучение с частичным привлечением учителя.

Подготовка данных

Для начала загрузим необходимые библиотеки и подготовим данные.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
```

```
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score

'''Алгоритмы'''
import lightgbm as lgb

'''TensorFlow и Keras'''
import tensorflow as tf
import keras
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout
from keras.layers import BatchNormalization, Input, Lambda
from keras import regularizers
from keras.losses import mse, binary_crossentropy
```

Как и раньше, сгенерируем тренировочный и тестовый наборы, только на этот раз исключим из тренировочного набора 90% меток поддельных транзакций, чтобы сымитировать работу с частично размеченным набором.

Подобный шаг может показаться чересчур радикальным, однако на практике поддельные транзакции встречаются примерно так же редко (примерно одна попытка фальсификации на 10 000 случаев). Исключив 90% меток из тренировочного набора, мы сымитируем реальную ситуацию.

```
# Загрузка данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'credit_card_data', \
                         'credit_card.csv'])
data = pd.read_csv(current_path + file)

dataX = data.copy().drop(['Class','Time'],axis=1)
dataY = data['Class'].copy()
```

```
# Масштабирование данных
featuresToScale = dataX.columns
sX = pp.StandardScaler(copy=True, with_mean=True, with_std=True)
dataX.loc[:, featuresToScale] = \
    sX.fit_transform(dataX[featuresToScale])

# Разбиение на тренировочный и тестовый наборы
X_train, X_test, y_train, y_test = train_test_split(dataX, dataY, \
    test_size=0.33, random_state=2018, stratify=dataY)

# Исключение 90% меток из тренировочного набора
toDrop = y_train[y_train==1].sample(frac=0.90, random_state=2018)
X_train.drop(labels=toDrop.index, inplace=True)
y_train.drop(labels=toDrop.index, inplace=True)
```

Мы также повторно используем функции anomalyScores и plotResults.

```
def anomalyScores(originalDF, reducedDF):
    loss = np.sum((np.array(originalDF) - np.array(reducedDF))**2, \
                  axis=1)
    loss = pd.Series(data=loss, index=originalDF.index)
    loss = (loss - np.min(loss)) / (np.max(loss) - np.min(loss))
    return loss

def plotResults(trueLabels, anomalyScores, returnPreds = False):
    preds = pd.concat([trueLabels, anomalyScores], axis=1)
    preds.columns = ['trueLabel', 'anomalyScore']
    precision, recall, thresholds = \
        precision_recall_curve(preds['trueLabel'], \
                               preds['anomalyScore'])
    average_precision = average_precision_score(preds['trueLabel'], \
        preds['anomalyScore'])

    plt.step(recall, precision, color='k', alpha=0.7, where='post')
    plt.fill_between(recall, precision, step='post', alpha=0.3, \
                     color='k')

    plt.xlabel('Полнота')
    plt.ylabel('Точность')
    plt.ylim([0.0, 1.05])
    plt.xlim([0.0, 1.0])
```

```
    plt.title('Кривая "точность - полнота": средняя точность = \
              {0:0.2f}'.format(average_precision))

    fpr, tpr, thresholds = roc_curve(preds['trueLabel'], \
                                     preds['anomalyScore'])
    areaUnderROC = auc(fpr, tpr)

    plt.figure()
    plt.plot(fpr, tpr, color='r', lw=2, label='ROC-кривая')
    plt.plot([0, 1], [0, 1], color='k', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Доля ложноположительных исходов')
    plt.ylabel('Доля истинноположительных исходов')
    plt.title('    Рабочая характеристика приемника: \n \
          площадь под кривой = {0:0.2f}'.format(areaUnderROC))
    plt.legend(loc="lower right")
    plt.show()
    if returnPreds==True:
        return preds, average_precision
```

Нам понадобится новая функция, `precisionAnalysis`, которая поможет оценить точность наших моделей при определенном уровне полноты классификации. В частности, мы определим, какая точность соответствует 75%-ному захвату поддельных транзакций в тестовом наборе. Чем выше точность, тем лучше модель.

Это разумная точка отсчета. Мы хотим обнаруживать 75% подделок с как можно более высокой точностью. Если нам не удастся достичь нужной точности, то мы будем непреднамеренно отвергать вполне законные транзакции, что вызовет неизбежные жалобы со стороны пользователей.

```
def precisionAnalysis(df, column, threshold):
    df.sort_values(by=column, ascending=False, inplace=True)
    threshold_value = threshold * df.trueLabel.sum()
    i = 0
    j = 0
    while i < threshold_value+1:
        if df.iloc[j]["trueLabel"]==1:
            i += 1
        j += 1
    return df, i/j
```

Модель на основе обучения с учителем

Чтобы оценить качество итогового решения, мы сначала должны понять, насколько эффективна каждая из моделей на основе обучения с учителем и без учителя по отдельности.

Начнем с модели на основе обучения с учителем и воспользуемся решением, которое продемонстрировало наилучшие результаты в главе 2: градиентный бустинг LightGBM (LightGBM). Мы применим пятикратную кросс-проверку:

```
k_fold = StratifiedKFold(n_splits=5, shuffle=True, random_state=2018)
```

Далее настроим параметры градиентного бустинга.

```
params_lightGB = {
    'task': 'train',
    'application': 'binary',
    'num_class': 1,
    'boosting': 'gbdt',
    'objective': 'binary',
    'metric': 'binary_logloss',
    'metric_freq': 50,
    'is_training_metric': False,
    'max_depth': 4,
    'num_leaves': 31,
    'learning_rate': 0.01,
    'feature_fraction': 1.0,
    'bagging_fraction': 1.0,
    'bagging_freq': 0,
    'bagging_seed': 2018,
    'verbose': 0,
    'num_threads':16
}
```

Теперь обучим алгоритм.

```
trainingScores = []
cvScores = []
predictionsBasedOnKFolds = pd.DataFrame(data=[], \
                                        index= y_train.index, \
                                        columns=['prediction'])

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
```

```
    X_train_fold, X_cv_fold = X_train.iloc[train_index, :], \
        X_train.iloc[cv_index, :]
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    lgb_train = lgb.Dataset(X_train_fold, y_train_fold)
    lgb_eval = lgb.Dataset(X_cv_fold, y_cv_fold, reference=lgb_train)
    gbm = lgb.train(params_lightGB, lgb_train, num_boost_round=2000, \
                    valid_sets=lgb_eval, early_stopping_rounds=200)

    loglossTraining = log_loss(y_train_fold, \
        gbm.predict(X_train_fold, num_iteration=gbm.best_iteration))
    trainingScores.append(loglossTraining)

    predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'] = \
        gbm.predict(X_cv_fold, num_iteration=gbm.best_iteration)
    loglossCV = log_loss(y_cv_fold, \
        predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'])
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения:', loglossTraining)
    print('Логарифмические потери валидации:', loglossCV)

loglossLightGBMGradientBoosting = log_loss(y_train, \
    predictionsBasedOnKFolds.loc[:, 'prediction'])
print('Логарифмические потери градиентного бустинга LightGBM:', \
      loglossLightGBMGradientBoosting)
```

Мы используем эту модель для предсказания подделок в тестовом наборе транзакций с банковскими картами. Результаты представлены на рис. 9.1.

В соответствии с кривой "точность — полнота" средняя точность составила 0.31. Выявлению 75% подделок соответствует точность, равная всего лишь 0,01%.

Модель на основе обучения без учителя

Сейчас мы построим решение на основе обучения без учителя. В частности, мы создадим разреженный двухслойный сверхполный автокодировщик с линейной функцией активации, 40 узлами в скрытом слое и дропаутом 2%.

В то же время мы скорректируем тренировочный набор, применив избыточное семплирование для увеличения количества имеющихся примеров

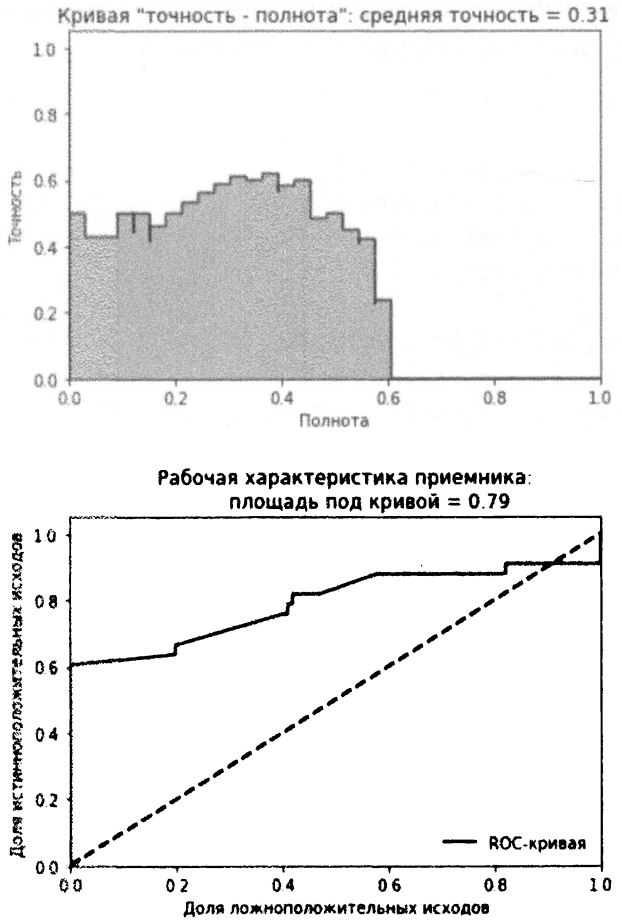


Рис. 9.1. *Результаты, полученные с помощью модели на основе обучения с учителем*

фальсификации транзакций. *Избыточное семплирование* (oversampling) — это методика, используемая для корректировки распределения классов в наборе данных. Мы хотим добавить больше примеров подделок в тренировочный набор, чтобы автокодировщику, который мы обучаем, было легче отделять нормальные транзакции от поддельных.

Вспомните, что после исключения 90% поддельных примеров из тренировочного набора у нас осталось всего-навсего 33 примера мошеннических транзакций. Мы возьмем эти 33 примера, создадим 100 дубликатов и присоединим к тренировочному набору. Кроме того, мы сохраним копии наборов, не подвергнутых избыточному семплированию, для использования на остальных этапах конвейера.

Тестовый набор мы не трогаем: избыточное семплирование применяется только к тренировочному набору, но не к тестовому.

```
oversample_multiplier = 100

X_train_original = X_train.copy()
y_train_original = y_train.copy()
X_test_original = X_test.copy()
y_test_original = y_test.copy()

X_train_oversampled = X_train.copy()
y_train_oversampled = y_train.copy()
X_train_oversampled = X_train_oversampled.append( \
    [X_train_oversampled[y_train==1]] * oversample_multiplier, \
    ignore_index=False)
y_train_oversampled = y_train_oversampled.append( \
    [y_train_oversampled[y_train==1]] * oversample_multiplier, \
    ignore_index=False)

X_train = X_train_oversampled.copy()
y_train = y_train_oversampled.copy()
```

Теперь обучим наш автокодировщик.

```
model = Sequential()
model.add(Dense(units=40, activation='linear', \
                activity_regularizer=regularizers.l1(10e-5), \
                input_dim=29, name='hidden_layer'))
model.add(Dropout(0.02))
model.add(Dense(units=29, activation='linear'))

model.compile(optimizer='adam', \
              loss='mean_squared_error', \
              metrics=['accuracy'])

num_epochs = 5
batch_size = 32

history = model.fit(x=X_train, y=X_train, \
                    epochs=num_epochs, \
                    batch_size=batch_size, \
                    shuffle=True, \
                    validation_split=0.20, \
                    verbose=1)
```

```
predictions = model.predict(X_test, verbose=1)
anomalyScoresAE = anomalyScores(X_test, predictions)
preds, average_precision = plotResults(y_test, anomalyScoresAE, True)
```

Результаты представлены на рис. 9.2.

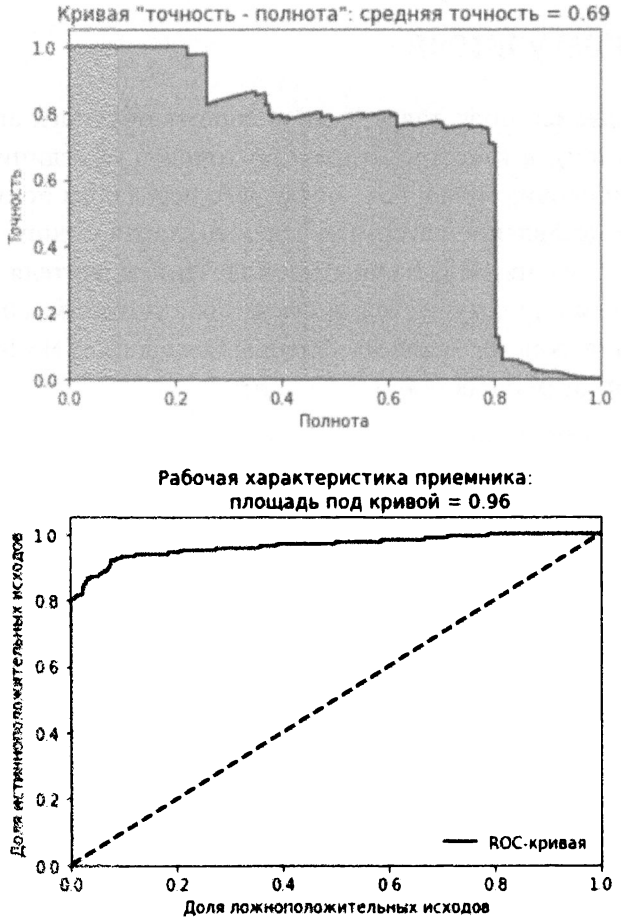


Рис. 9.2. *Результаты, полученные с помощью модели на основе обучения без учителя*

В соответствии с кривой "точность — полнота" средняя точность составила `0.69`. Выявлению 75% подделок соответствует точность 75%. Средняя точность решения на основе обучения без учителя повысилась более чем в два раза по сравнению с решением на основе обучения с учителем, но самое существенное изменение — точность 75%, достигаемая при 75%-ной полноте

классификации, что представляет собой огромное улучшение по сравнению с 0,01%.

И все же решение на основе обучения без учителя не самое эффективное.

Модель на основе обучения с частичным привлечением учителя

Мы воспользуемся представлением, которому обучился автокодировщик (его скрытый слой), в качестве отправной точки и объединим его с оригинальным тренировочным набором, после чего передадим полученное комбинированное представление алгоритму градиентного бустинга. Такой подход, основанный на обучении с частичным привлечением учителя, позволяет воспользоваться всеми преимуществами обучения с учителем и без учителя.

Для получения скрытого слоя мы обращаемся к классу `Model` из библиотеки Keras и используем функцию `get_layer`.

```
layer_name = 'hidden_layer'

intermediate_layer_model = Model(inputs=model.input, \
    outputs=model.get_layer(layer_name).output)
intermediate_output_train = \
    intermediate_layer_model.predict(X_train_original)
intermediate_output_test = \
    intermediate_layer_model.predict(X_test_original)
```

Сохраним эти представления автокодировщика в объектах `DataFrame` и объединим их с исходным тренировочным набором.

```
intermediate_output_trainDF = \
    pd.DataFrame(data=intermediate_output_train, \
                 index=X_train_original.index)
intermediate_output_testDF = \
    pd.DataFrame(data=intermediate_output_test, \
                 index=X_test_original.index)

X_train = X_train_original.merge(intermediate_output_trainDF, \
                                 left_index=True, right_index=True)
X_test = X_test_original.merge(intermediate_output_testDF, \
                               left_index=True, right_index=True)
y_train = y_train_original.copy()
```

После этого мы обучим модель градиентного бустинга на новом тренировочном наборе, включающем 69 признаков (29 из оригинального набора плюс 40 из представления автокодировщика).

```
trainingScores = []
cvScores = []
predictionsBasedOnKFolds = \
    pd.DataFrame(data=[], index=y_train.index, \
                 columns=['prediction'])

for train_index, cv_index in k_fold.split(np.zeros(len(X_train)), \
                                          y_train.ravel()):
    X_train_fold, X_cv_fold = X_train.iloc[train_index, :], \
        X_train.iloc[cv_index, :]
    y_train_fold, y_cv_fold = y_train.iloc[train_index], \
        y_train.iloc[cv_index]

    lgb_train = lgb.Dataset(X_train_fold, y_train_fold)
    lgb_eval = lgb.Dataset(X_cv_fold, y_cv_fold, reference=lgb_train)
    gbm = lgb.train(params_lightGB, lgb_train, num_boost_round=5000, \
                    valid_sets=lgb_eval, early_stopping_rounds=200)

    loglossTraining = log_loss(y_train_fold, \
                               gbm.predict(X_train_fold, \
                               num_iteration=gbm.best_iteration))
    trainingScores.append(loglossTraining)

    predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'] = \
        gbm.predict(X_cv_fold, num_iteration=gbm.best_iteration)
    loglossCV = log_loss(y_cv_fold, \
        predictionsBasedOnKFolds.loc[X_cv_fold.index, 'prediction'])
    cvScores.append(loglossCV)

    print('Логарифмические потери обучения:', loglossTraining)
    print('Логарифмические потери валидации:', loglossCV)

loglossLightGBMGradientBoosting = log_loss(y_train, \
    predictionsBasedOnKFolds.loc[:, 'prediction'])
print('Логарифмические потери градиентного бустинга LightGBM:', \
      loglossLightGBMGradientBoosting)
```

Результаты представлены на рис. 9.3.

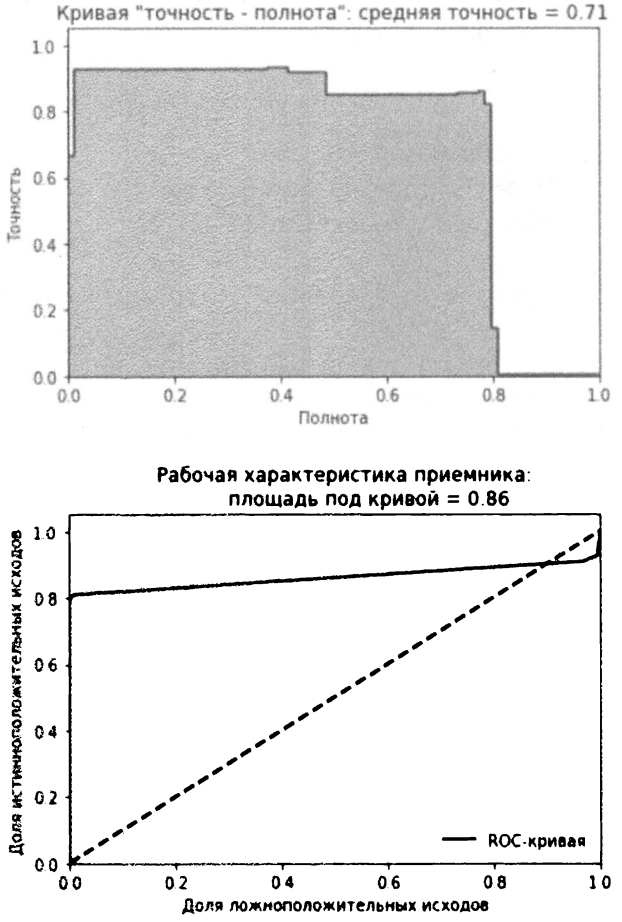


Рис. 9.3. *Результаты, полученные с помощью модели на основе обучения с частичным привлечением учителя*

В соответствии с кривой "точность — полнота" средняя точность составила `0.71`. Это лучше, чем в моделях на основе обучения с учителем и без учителя.

Обнаружению 75% подделок соответствует точность 85%, что представляет собой значительное улучшение. При таком уровне точности платежная система может с достаточной долей уверенности отвергать транзакции, помеченные моделью как потенциально мошеннические. Ошибки могут совершаться примерно в одном из десяти случаев, и при этом нам удастся выявить примерно 75% подделок.

Важность обучения без учителя и обучения с учителем

В данном приложении, основанном на обучении с частичным привлечением учителя, как обучение с учителем, так и обучение без учителя играют очень важную роль. Чтобы понять это, можно проанализировать, какие признаки определяются моделью градиентного бустинга как наиболее значимые.

Извлечем значения, характеризующие важность признаков, из обученной модели.

```
featuresImportance = \
    pd.DataFrame(data=list(gbm.feature_importance()), \
                 index=X_train.columns, columns=['featImportance'])
featuresImportance = featuresImportance / featuresImportance.sum()
featuresImportance.sort_values(by='featImportance', ascending=False, \
                               inplace=True)
featuresImportance
```

Часть наиболее важных признаков, отсортированных по убыванию, приведена в табл. 9.1.

Таблица 9.1. Важность признаков в модели на основе обучения с частичным привлечением учителя

	featImportance
V14	0.15
V1	0.10
Amount	0.10
27	0.10
V7	0.10

Как видите, некоторые из ключевых признаков в таблице — это признаки скрытого слоя, которым обучился автокодировщик (не помечены префиксом "V"), тогда как остальные признаки — это главные компоненты, полученные на основе оригинального набора данных (помечены префиксом "V"), и величина транзакции (`Amount`).

Резюме

Модель с частичным привлечением учителя по своей производительности превосходит взятые по отдельности модели обучения с учителем и без учителя.

Это было лишь поверхностное знакомство с технологией полуавтоматического обучения, но оно должно побудить вас поменять стратегию и перестать раздумывать, какой из вариантов обучения — без учителя или с учителем — лучше. Теперь у вас есть возможность сочетать обе технологии в поисках оптимального решения.

Глубокое обучение без учителя с использованием библиотек TensorFlow и Keras

До сих пор мы работали лишь с мелкими нейронными сетями, насчитывавшими всего несколько скрытых слоев. Такие сети находят применение при создании систем машинного обучения, но основные достижения в этой области, достигнутые за прошедшее десятилетие, связаны с *глубокими нейронными сетями*, включающими множество скрытых слоев. Соответствующий раздел машинного обучения называется *глубокое обучение* (deep learning). Применительно к большим размеченным наборам данных глубокое обучение обеспечило коммерческий успех проектов в таких областях, как компьютерное зрение, распознавание объектов, распознавание речи и машинный перевод.

В этой части мы будем работать с большими неразмеченными наборами, задействуя *глубокое обучение без учителя* (deep unsupervised learning) — относительно новое многообещающее направление, которое пока еще не может похвастаться серьезными достижениями, в отличие от аналогичного подхода на основе обучения с учителем. В следующих главах мы займемся созданием систем глубокого обучения без учителя, начав с простейших строительных блоков.

В главе 10 обсуждаются ограниченные машины Больцмана, которые мы применим для построения рекомендательной системы фильмов. В главе 11 мы создадим так называемые *сети глубокого доверия* на основе каскадов, формируемых из нескольких ограниченных машин Больцмана. В главе 12 мы научимся генерировать синтетические данные с помощью *генеративно-состязательных сетей* — это одно из самых передовых направлений глубокого обучения без учителя на сегодняшний день. Наконец, в главе 13 мы вновь вернемся к кластеризации, но на этот раз будем работать с данными временных рядов.

Рекомендательные системы на основе ограниченных машин Больцмана

В предыдущих главах мы применяли обучение без учителя для выявления базовой (скрытой) структуры в неразмеченных данных. В частности, мы снижали размерность многомерных наборов данных и создавали системы обнаружения аномалий. Мы также выполняли кластеризацию, группируя объекты на основе их сходства.

Теперь мы перейдем к рассмотрению *порождающих* (*генеративных*) *моделей*, которые обучаются вероятностному распределению классов в исходном наборе данных и приобретают способность делать выводы относительно данных, которые еще не встречались. В последующих главах мы применим такие модели для генерирования синтетических данных, которые порой совершенно неотличимы от подлинных данных.

До сих пор мы имели дело преимущественно с *дискриминативными моделями*, которые обучаются разделять наблюдения на основе знаний, извлекаемых из данных. Такие модели не могут обучаться вероятностным распределениям классов в исходных данных. В эту категорию входят такие модели, как логистическая регрессия и деревья принятия решений, рассмотренные в главе 2, а также различные методы кластеризации, в том числе метод k-средних и иерархическая кластеризация, которые обсуждались в главе 5.

Мы начнем с рассмотрения простейшей порождающей модели обучения без учителя, известной как *ограниченная машина Больцмана*.

Машины Больцмана

Машины Больцмана были предложены в 1985 году Джеффри Хинтоном (на тот момент — профессор университета Карнеги — Меллона, а в настоящее время — профессор университета Торонто и один из ведущих инженеров компании Google) и Терри Сейновски (на тот момент — профессор университета Джона Хопкинса).

Машины Больцмана (неограниченного типа) состоят из нейронной сети с входным слоем и одним или несколькими скрытыми слоями. Нейроны или элементы нейронной сети принимают стохастические решения относительно того, включаться или не включаться, на основании данных, получаемых в процессе тренировки, и поведения функции потерь, которую машина Больцмана пытается минимизировать. В ходе тренировки машина Больцмана выявляет признаки, представляющие наибольший интерес, что помогает моделировать сложные отношения и закономерности, свойственные данным.

В неограниченных машинах Больцмана используются нейронные сети, в которых нейроны связаны не только с другими слоями, но и с нейронами, принадлежащими к тому же слою. Этот фактор в сочетании с наличием большого количества скрытых слоев делает тренировку машины Больцмана очень неэффективной. Как следствие, на протяжении 1980–1990-х годов неограниченные машины Больцмана не могли похвастаться сколь-нибудь значительными успехами.

Ограниченные машины Больцмана

В 2000-е годы Джеффри Хинтон с коллегами совершили серьезный прорыв, начав использовать модифицированную версию исходной машины Больцмана. *Ограниченная машина Больцмана* (restricted Boltzmann machine — RBM) содержит один входной слой (также называемый *видимым слоем*) и единственный скрытый слой, а соединения между нейронами ограничены таким образом, что нейроны связываются лишь с нейронами других слоев, но не собственного слоя. Иными словами, отсутствуют связи типа "видимый — видимый" и "скрытый — скрытый"[1].

Джеффри Хинтон также продемонстрировал возможность каскадирования простых машин таким образом, чтобы выход скрытого слоя одной RBM служил входным слоем для другой RBM. Такого типа каскадирование можно повторять многократно для обучения скрытым представлениям, захватывающим все более тонкие детали структуры исходных данных. Подобную сеть, состоящую из многих RBM, можно рассматривать как одну глубокую многослойную модель, что и ознаменовало старт глубокого обучения в 2006 году.

Следует отметить, что для обучения базовой структуре данных в RBM применяется *стохастический* подход, тогда как, например, в автокодировщиках используется *детерминированный* подход.

[1] Для обучения машин такого класса чаще всего применяют *градиентный алгоритм контрастивной дивергенции.*

Рекомендательные системы

В данной главе мы будем использовать RBM для построения *рекомендательной системы*. На сегодняшний день это одно из наиболее успешных применений машинного обучения. Рекомендательные системы широко применяются для предсказания предпочтений пользователей в отношении фильмов, музыки, книг, новостей, поисковых запросов, покупок, цифровой рекламы и онлайн-знакомств.

Существуют две основные категории рекомендательных систем: на основе *коллаборативной фильтрации* и на основе *фильтрации по содержимому*. Коллаборативная фильтрация подразумевает создание рекомендательной системы, обучение которой базируется на уже накопленных сведениях о данном пользователе и других пользователях с похожими предпочтениями. Такая система способна предсказывать объекты, которые могут заинтересовать пользователя, даже если он никогда ранее не проявлял явного интереса к ним. Именно коллаборативная фильтрация положена в основу рекомендательной системы фильмов, используемой компанией Netflix.

Фильтрация на основе содержимого подразумевает обучение различным свойствам объекта, что позволяет рекомендовать другие объекты с похожими свойствами. Именно так генерируются рекомендации на сайте Pandora.com, облегчающие пользователям поиск музыки, которая им нравится.

Коллаборативная фильтрация

Фильтрация по содержимому не получила широкого распространения, поскольку обучение различным свойствам объектов — довольно трудоемкая задача. Достичь понимания на таком уровне — серьезный вызов, на который системы искусственного интеллекта пока не могут достойно ответить. Гораздо легче собрать и проанализировать большой объем информации, касающейся поведения и предпочтений пользователей, и на основании этого строить прогнозы. Поэтому область применения коллаборативной фильтрации намного шире, и наше внимание будет сосредоточено на ней.

Коллаборативная фильтрация не требует наличия знаний о самих объектах. Вместо этого предполагается, что поведение пользователей с похожими предпочтениями в прошлом будет столь же сходным и в будущем, а предпочтения пользователей останутся примерно теми же. Моделируя степень сходства пользователей, коллаборативная фильтрация позволяет получать довольно точные предсказания. Более того, коллаборативная фильтрация не

полагается на *явные данные* (т.е. рейтинги, предоставляемые пользователями). Вместо этого она работает с *неявными данными*, например данными о том, как долго пользователь просматривает конкретный объект или как часто щелкает на нем. Например, если раньше сайт Netflix предлагал пользователям ставить оценки фильмам, то теперь он самостоятельно делает заключения о том, нравится фильм пользователям или нет, анализируя неявные данные об их поведении.

В то же время применение коллаборативной фильтрации сопряжено с определенными трудностями. Во-первых, для создания надежных рекомендаций требуется доступ к большим объемам данных о поведении пользователей. Во-вторых, эта задача требует выполнения весьма трудоемких вычислений. В-третьих, соответствующие наборы данных, как правило, очень разрежены, поскольку предпочтения, проявляемые пользователями, охватывают лишь малую часть всех возможных объектов. Но если предположить, что имеется достаточно большой объем исходных данных, то существуют методики, позволяющие справиться с проблемой разреженности, рассмотрению которых и посвящена данная глава.

Соревнование Netflix Prize

В 2006 году компания Netflix спонсировала трехлетний конкурс на улучшение своей рекомендательной системы фильмов. Главный приз в один миллион долларов был обещан команде, которая сможет улучшить точность существующей системы по крайней мере на 10%. Участникам был предоставлен набор данных, содержащий свыше 100 млн рейтингов фильмов. В сентябре 2009 года приз был вручен команде *BellKor's Pragmatic Chaos*, которая использовала ансамблевую модель, объединяющую многие алгоритмические подходы.

Это получившее широкую огласку соревнование с богатым набором исходных данных и внушительным денежным призом вдохновило сообщество машинного обучения, что привело к существенному прогрессу в исследовании рекомендательных систем за последние годы.

Мы используем аналогичный набор данных о рейтингах фильмов для построения собственной рекомендательной системы с помощью ограниченных машин Больцмана.

Набор данных MovieLens

Вместо того чтобы работать с набором данных Netflix с его 100 млн рейтингов фильмов, мы задействуем меньший набор *MovieLens 20M Dataset*, предоставленный исследовательской лабораторией факультета информатики и вычислительной техники университета Миннесоты. Этот набор сформирован на базе отзывов 138 493 пользователей за период с 9 января 1995 года по 31 марта 2015 года и содержит 20 000 263 рейтингов, которые охватывают 27 278 фильмов. Среди всех пользователей, которые присвоили рейтинги по крайней мере 20 фильмам, мы отберем случайное подмножество.

Работать с этим набором гораздо проще, чем с набором Netflix, содержащим 100 млн рейтингов. Тем не менее размер файла превышает 100 Мбайт, поэтому он недоступен на GitHub. Вы сможете загрузить этот файл непосредственно с сайта MovieLens (`http://bit.ly/2G0ZHCn`).

Подготовка данных

Как всегда, сначала загрузим необходимые библиотеки.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip, datetime

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
```

```
from sklearn.metrics import roc_curve, auc, roc_auc_score, \
    mean_squared_error

'''Алгоритмы'''
import lightgbm as lgb

'''TensorFlow (1.x) и Keras'''
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import keras
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout
from keras.layers import BatchNormalization, Input, Lambda
from keras import regularizers
from keras.losses import mse, binary_crossentropy
```

Далее загрузим набор рейтингов и преобразуем поля в подходящие типы данных. В нашем распоряжении имеется всего несколько полей: идентификатор пользователя (`userID`), идентификатор фильма (`movieID`), рейтинг, присвоенный пользователем данному фильму (`rating`), и метка времени.

```
# Загрузка данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'movielens_data', \
                         'ratings.csv'])
ratingDF = pd.read_csv(current_path + file)

# Преобразование полей в подходящие типы данных
ratingDF.userId = ratingDF.userId.astype(str).astype(int)
ratingDF.movieId = ratingDF.movieId.astype(str).astype(int)
ratingDF.rating = ratingDF.rating.astype(str).astype(float)
ratingDF.timestamp = ratingDF.timestamp.apply(lambda x: \
    datetime.utcfromtimestamp(x).strftime('%Y-%m-%d %H:%M:%S'))
```

Фрагмент данных приведен в табл. 10.1.

Подтвердим количество имеющихся уникальных пользователей, уникальных фильмов и общее количество оценок, а заодно подсчитаем среднее количество оценок, приходящееся на одного пользователя.

```
n_users = ratingDF.userId.unique().shape[0]
n_movies = ratingDF.movieId.unique().shape[0]
n_ratings = len(ratingDF)
avg_ratings_per_user = n_ratings/n_users
```

```
print('Количество уникальных пользователей:', n_users)
print('Количество уникальных фильмов:', n_movies)
print('Общее количество рейтингов:', n_ratings)
print('Среднее количество рейтингов на одного пользователя:', \
      avg_ratings_per_user)
```

Таблица 10.1. Данные рейтингов MovieLens

	userId	movieId	rating	timestamp
0	1	2	3.5	2005-04-02 23:53:47
1	1	29	3.5	2005-04-02 23:31:16
2	1	32	3.5	2005-04-02 23:33:39
3	1	47	3.5	2005-04-02 23:32:07
4	1	50	3.5	2005-04-02 23:29:40
5	1	112	3.5	2004-09-10 03:09:00
6	1	151	4.0	2004-09-10 03:08:54
7	1	223	4.0	2005-04-02 23:46:13
8	1	253	4.0	2005-04-02 23:35:40
9	1	260	4.0	2005-04-02 23:33:46
10	1	293	4.0	2005-04-02 23:31:43
11	1	296	4.0	2005-04-02 23:32:47
12	1	318	4.0	2005-04-02 23:33:18
13	1	337	3.5	2004-09-10 03:08:29

Полученные результаты соответствуют нашим ожиданиям.

```
Количество уникальных пользователей: 138493
Количество уникальных фильмов: 26744
Общее количество рейтингов: 20000263
Среднее количество рейтингов на одного пользователя:
144.4135299257002
```

Снизим сложность этого набора и уменьшим его размер, сосредоточив внимание на первой тысяче самых рейтинговых фильмов. В результате количество оценок сократится с ~20 до ~12,8 млн.

```
movieIndex = ratingDF.groupby("movieId").count().sort_values(by= \
    "rating", ascending=False)[0:1000].index
ratingDFX2 = ratingDF[ratingDF.movieId.isin(movieIndex)]
ratingDFX2.count()
```

Кроме того, сформируем случайную выборку из 1000 пользователей и отберем из набора данные, относящиеся только к этим пользователям, что позволит дополнительно снизить количество оценок с ~12,8 млн до всего лишь 90 213. Такого количества нам хватит для того, чтобы продемонстрировать, как работает коллаборативная фильтрация.

```
userIndex = ratingDFX2.groupby("userId").count().sort_values(by= \
    "rating", ascending=False).sample(n=1000, \
    random_state=2018).index
ratingDFX3 = ratingDFX2[ratingDFX2.userId.isin(userIndex)]
ratingDFX3.count()
```

После сокращения набора выполним переиндексацию идентификаторов `movieID` и `userID`, чтобы привести их к диапазону значений 1–1000.

```
movies = ratingDFX3.movieId.unique()
moviesDF = pd.DataFrame(data=movies, columns=['originalMovieId'])
moviesDF['newMovieId'] = moviesDF.index+1

users = ratingDFX3.userId.unique()
usersDF = pd.DataFrame(data=users, columns=['originalUserId'])
usersDF['newUserId'] = usersDF.index+1

ratingDFX3 = ratingDFX3.merge(moviesDF, left_on='movieId', \
                              right_on='originalMovieId')
ratingDFX3.drop(labels='originalMovieId', axis=1, inplace=True)
ratingDFX3 = ratingDFX3.merge(usersDF, left_on='userId', \
                              right_on='originalUserId')
ratingDFX3.drop(labels='originalUserId', axis=1, inplace=True)
```

Заново подсчитаем количество имеющихся уникальных пользователей, уникальных фильмов и общее количество оценок, а также среднее количество оценок, приходящееся на одного пользователя, для уменьшенного набора данных.

```
n_users = ratingDFX3.userId.unique().shape[0]
n_movies = ratingDFX3.movieId.unique().shape[0]
n_ratings = len(ratingDFX3)
avg_ratings_per_user = n_ratings/n_users

print('Количество уникальных пользователей:', n_users)
print('Количество уникальных фильмов:', n_movies)
print('Общее количество рейтингов:', n_ratings)
print('Среднее количество рейтингов на одного пользователя:', \
      avg_ratings_per_user)
```

Полученные результаты подтверждают наши ожидания.

```
Количество уникальных пользователей: 1000
Количество уникальных фильмов: 1000
Общее количество рейтингов: 90213
Среднее количество рейтингов на одного пользователя: 90.213
```

Наконец, сгенерируем из редуцированного набора данных тестовый и валидационный наборы, каждый из которых будет содержать 5% данных редуцированного набора.

```
X_train, X_test = train_test_split(ratingDFX3, \
   test_size=0.10, shuffle=True, random_state=2018)
X_validation, X_test = train_test_split(X_test, \
   test_size=0.50, shuffle=True, random_state=2018)
```

Проверим размеры тренировочного, тестового и валидационного наборов.

```
Размер тренировочного набора: 81191
Размер валидационного набора: 4511
Размер тестового набора: 4511
```

Определение функции потерь: среднеквадратическая ошибка

Теперь можно приступать к работе с имеющимися данными.

Прежде всего создадим матрицу размером $m \times n$, где m — количество пользователей, а n — количество фильмов. Эта матрица будет разреженной, поскольку пользователи оценивают лишь небольшую часть фильмов. Например, матрица, соответствующая одной тысяче пользователей и одной тысяче фильмов, содержит лишь 81 191 рейтинг. Если бы каждый из тысячи пользователей оценил каждый из тысячи фильмов, то матрица содержала бы один миллион оценок, но пользователи оценивают в среднем лишь небольшое подмножество фильмов, поэтому мы будем располагать намного меньшим числом рейтингов. Остальные элементы матрицы (примерно 92%) будут иметь нулевые значения.

```
# Генерирование тренировочной матрицы рейтингов
ratings_train = np.zeros((n_users, n_movies))
for row in X_train.itertuples():
    ratings_train[row[6]-1, row[5]-1] = row[3]

# Вычисление степени разреженности тренировочной матрицы
```

```
sparsity = float(len(ratings_train.nonzero()[0]))
sparsity /= (ratings_train.shape[0] * ratings_train.shape[1])
sparsity *= 100
print('Разреженность: {:4.2f}%'.format(sparsity))
```

Сгенерируем аналогичные матрицы для валидационного и тестового наборов, которые, конечно же, будут еще более разреженными.

```
# Генерирование валидационной матрицы рейтингов
ratings_validation = np.zeros((n_users, n_movies))
for row in X_validation.itertuples():
    ratings_validation[row[6]-1, row[5]-1] = row[3]

# Генерирование тестовой матрицы рейтингов
ratings_test = np.zeros((n_users, n_movies))
for row in X_test.itertuples():
    ratings_test[row[6]-1, row[5]-1] = row[3]
```

Прежде чем приступить к созданию рекомендательной системы, мы должны определить функцию потерь, которую будем использовать для оценки качества модели. В качестве таковой выберем *среднеквадратическую ошибку* (mean squared error — MSE) между предсказанными и фактическими значениями. Это одна из простейших функций потерь в машинном обучении. Для вычисления MSE нам потребуются два вектора размером [n, 1], где n — количество предсказываемых рейтингов, равное 4511 для валидационного набора. Один вектор будет содержать фактические оценки, а второй — предсказания.

Уплощим разреженную матрицу рейтингов для валидационного набора. В результате получим вектор фактических оценок.

```
actual_validation = \
    ratings_validation[ratings_validation.nonzero()].flatten()
```

Опорные эксперименты

Проведем опорный эксперимент, предсказав средний рейтинг `3.5` для валидационного набора, и вычислим MSE.

```
pred_validation = np.zeros((len(X_validation), 1))
pred_validation[pred_validation==0] = 3.5

naive_prediction = mean_squared_error(pred_validation, \
                                      actual_validation)
```

Показатель MSE для этого наивного предсказания составил 1.05. Данное значение послужит точкой отсчета.

```
Среднеквадратическая ошибка с использованием наивного предсказания:
1.055420084238528
```

Проверим, сможем ли мы улучшить результаты, если предскажем пользовательскую оценку заданного фильма на основании средней оценки, присвоенной данным пользователем другим фильмам.

```
ratings_validation_prediction = np.zeros((n_users, n_movies))
i = 0
for row in ratings_train:
    ratings_validation_prediction[i]
        [ratings_validation_prediction[i]==0] = np.mean(row[row>0])
    i += 1

pred_validation = ratings_validation_prediction \
    [ratings_validation.nonzero()].flatten()
user_average = mean_squared_error(pred_validation, actual_validation)
print('Среднеквадратическая ошибка с использованием усредненной \
       пользовательской оценки:', user_average)
```

Показатель MSE улучшился до 0.909.

```
Среднеквадратическая ошибка с использованием усредненной
пользовательской оценки: 0.9090717929472647
```

Теперь давайте предскажем пользовательскую оценку заданного фильма на основании среднего значения рейтингов, присвоенных данному фильму остальными пользователями.

```
ratings_validation_prediction = np.zeros((n_users, n_movies)).T
i = 0
for row in ratings_train.T:
    ratings_validation_prediction[i] \
        [ratings_validation_prediction[i]==0] = np.mean(row[row>0])
    i += 1

ratings_validation_prediction = ratings_validation_prediction.T
pred_validation = ratings_validation_prediction \
    [ratings_validation.nonzero()].flatten()
movie_average = mean_squared_error(pred_validation, actual_
validation)
print('Среднеквадратическая ошибка с использованием усредненной \
       оценки фильма другими пользователями:', movie_average)
```

При таком подходе показатель MSE составил `0.914`, что близко к результату предыдущего эксперимента.

```
Среднеквадратическая ошибка с использованием усредненной оценки
фильма другими пользователями: 0.9136057106858655
```

Матричная факторизация

Прежде чем приступить к построению рекомендательной системы на основе RBM, создадим подобную систему, используя *матричную факторизацию* — один из наиболее успешных и популярных алгоритмов коллаборативной фильтрации на сегодняшний день. При таком подходе матрица "пользователь — объект" разлагается на два множителя. Для представления пользователей и объектов применяются латентные пространства меньшей размерности.

Пусть матрица R представляет m пользователей и n объектов. В результате факторизации получим две матрицы меньшей размерности, H и W, где H — матрица размером "m пользователей" × "k факторов", а W — матрица размером "k факторов" × "n объектов".

Рейтинги вычисляются посредством операции матричного умножения: $R = HW$.

Число факторов k определяет емкость модели, которая возрастает с увеличением k. Повышая емкость, мы можем улучшить персонализацию рейтинговых предсказаний для пользователей, но при слишком высоких значениях k модель будет переобучаться.

Все это должно быть вам знакомо. Модель матричной факторизации обучается представлениям пользователей и объектов в пространстве более низкой размерности и строит прогнозы, базируясь на этих новых представлениях.

Один фактор

Начнем с простейшей формы матричной факторизации, когда имеется всего один фактор. Для факторизации матриц мы воспользуемся средствами библиотеки Keras.

Прежде всего мы должны определить граф, входом которого служат два вектора — одномерный вектор пользователей и одномерный вектор фильмов, — используемые для создания соответствующих вложений. Затем эти вложения уплощаются. Далее мы генерируем выходное произведение векторов, вычисляя скалярное произведение вектора пользователей и вектора фильмов. Для

минимизации функции потерь, определяемой как mean_squared_error (среднеквадратическая ошибка), применим *оптимизатор Adam*.

```
n_latent_factors = 1

user_input = Input(shape=[1], name='user')
user_embedding = Embedding(input_dim=n_users + 1, \
    output_dim=n_latent_factors, name='user_embedding')(user_input)
user_vec = Flatten(name='flatten_users')(user_embedding)
movie_input = Input(shape=[1], name='movie')
movie_embedding = Embedding(input_dim=n_movies + 1, \
    output_dim=n_latent_factors, name='movie_embedding')(movie_input)
movie_vec = Flatten(name='flatten_movies')(movie_embedding)

product = dot([movie_vec, user_vec], axes=1)
model = Model(inputs=[user_input, movie_input], outputs=product)
model.compile('adam', 'mean_squared_error')
```

Обучим модель, подав на вход векторы пользователей и фильмов из тренировочного набора данных. В процессе обучения модели мы будем оценивать ее на валидационном наборе. Показатель MSE будет вычисляться относительно имеющихся рейтингов.

Мы обучим модель на протяжении ста эпох и сохраним историю результатов тренировки и валидации, после чего построим график.

```
history = model.fit(x=[X_train.newUserId, X_train.newMovieId], \
                    y=X_train.rating, epochs=100, \
                    validation_data=([X_validation.newUserId, \
                    X_validation.newMovieId], X_validation.rating), \
                    verbose=1)

pd.Series(history.history['val_loss'][10:]).plot(logy=False)
plt.xlabel("Эпоха")
plt.ylabel("Ошибка валидации")
print('Минимальная MSE:', min(history.history['val_loss']))
```

Результаты представлены на рис. 10.1.

В случае матричной факторизации с одним фактором минимальное значение MSE составило `0.796`. Этот результат лучше по сравнению с подходами, основанными на усреднении оценок по пользователям и фильмам.

Проверим, удастся ли нам улучшить результат, если мы увеличим количество факторов (т.е. емкость модели).

```
Минимальная MSE: 0.7962130332067097
```

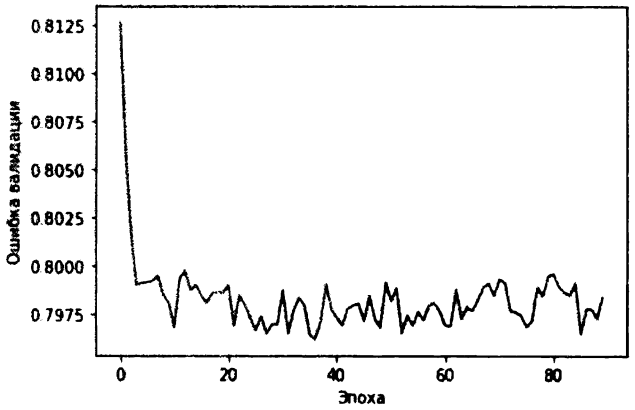


Рис. 10.1. *График MSE валидационного набора с использованием матричной факторизации и одного фактора*

Три фактора

На рис. 10.2 представлены результаты, соответствующие использованию трех факторов.

```
Минимальная MSE: 0.7754676667318934
```

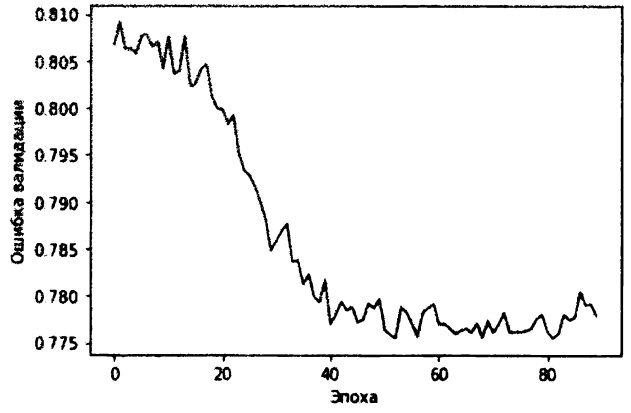


Рис. 10.2. *График MSE валидационного набора с использованием матричной факторизации и трех факторов*

Минимальное значение MSE составило `0.775`, что лучше, чем в случае одного фактора.

Пять факторов

Наконец, построим модель матричной факторизации, используя пять факторов (рис. 10.3).

```
Минимальная MSE: 0.7725457164195286
```

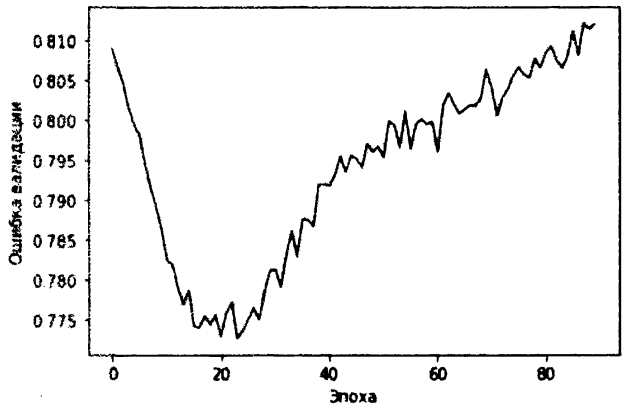


***Рис. 10.3.** График MSE валидационного набора с использованием матричной факторизации и пяти факторов*

Минимальное значение MSE не улучшилось, а после прохождения примерно 20 эпох появились явные признаки переобучения: ошибка валидации достигает минимума, а затем начинает расти. Дальнейшее увеличение емкости модели не даст никакого существенного улучшения.

Коллаборативная фильтрация с использованием RBM

Вернемся к модели на основе RBM. Вспомните, что в ограниченной машине Больцмана имеется два слоя: входной/видимый и скрытый. Нейроны каждого из этих слоев связаны с нейронами других слоев, но не с нейронами собственного слоя. Иными словами, внутрислойные связи отсутствуют — это и есть *ограничительный* элемент RBM.

Другое важное свойство RBM заключается в том, что связь между слоями осуществляется в обоих направлениях, а не только в одном. Например, в автокодировщиках нейроны сообщаются со следующим слоем, передавая информацию только в направлении прямого распространения. В RBM нейроны видимого слоя сообщаются со скрытым слоем, после чего скрытый слой передает информацию в обратном направлении видимому слою, и этот процесс повторяется многократно. RBM осуществляет обмен данными между видимым и

скрытым слоями в прямом и обратном направлении, стремясь сформировать такую порождающую модель, в которой выходные реконструкции скрытого слоя совпадают с оригинальными входами.

Таким образом, ограниченная машина Больцмана пытается создать порождающую модель, которая позволяет предсказывать, понравится ли пользователю фильм, который он еще не видел, на основании сходства этого фильма с другими фильмами, уже оцененными пользователем, а также на основании того, насколько предпочтения пользователя близки к предпочтениям других пользователей, уже оценивших данный фильм.

Видимый слой будет иметь x нейронов, где x — количество фильмов в наборе данных. Каждый нейрон будет содержать нормализованную оценку, значения которой находятся в диапазоне от 0 до 1, где 0 означает, что пользователь не видел данный фильм. Чем ближе нормализованное значение оценки к 1, тем вероятнее, что пользователю понравится фильм, представляемый данным нейроном.

Нейроны видимого слоя будут передавать информацию в скрытый слой, который будет пытаться обучиться базовым латентным признакам, характеризующим предпочтения пользователей.

Ограниченную машину Больцмана еще называют *симметричным двудольным двунаправленным графом*, поскольку такой граф содержит два слоя узлов, где каждый видимый узел связан с каждым скрытым узлом и обмен данными происходит в двух направлениях.

Архитектура нейронной сети на основе RBM

В нашей рекомендательной системе фильмов имеется матрица размером $m \times n$ с m пользователями и n фильмами. Для обучения RBM мы передаем нейронной сети пакет с данными о k пользователях и предоставленных ими рейтингах n фильмов и тренируем ее в течение определенного количества *эпох*.

Каждый вход x, передаваемый нейронной сети, представляет рейтинговые предпочтения одного пользователя для всех n фильмов, где n равно 1000 в нашем примере. Поэтому видимый слой содержит n узлов, по одному на каждый фильм.

Мы можем задать количество узлов в скрытом слое, которое обычно будет меньше, чем в видимом слое, чтобы заставить скрытый слой обучаться наиболее существенным аспектам входных данных настолько эффективно, насколько это возможно.

Каждый входной вектор v_0 умножается на соответствующий ему вес W. Этим весам обучаются соединения, связывающие видимый слой со скрытым слоем.

Затем полученное произведение суммируется с вектором смещения скрытого слоя, h_b. Добавление смещения гарантирует запуск по крайней мере части нейронов. Результат выражения $W * v_0 + h_b$ передается функции активации.

Далее мы формируем выборку выходов, генерируемую путем *семплирования по Гиббсу*. Другими словами, результатом активации скрытого слоя становятся окончательные выходы, генерируемые стохастически. Случайность способствует построению более производительной и надежной порождающей модели.

После этого выход гиббсовского семплирования, h_0, пропускается через нейронную сеть в противоположном направлении в рамках процесса, получившего название *обратное распространение ошибки*. Активации, полученные путем распространения результатов гиббсовского семплирования в *прямом направлении*, передаются скрытому слою и умножаются на ту же матрицу весов W, что и раньше. Затем мы прибавляем к произведению новый вектор смещения видимого слоя, v_b.

Результат выражения $W * h_0 + v_b$ пропускается через функцию активации, после чего выполняется семплирование по Гиббсу. Выходом становится вектор v_1, который затем передается видимому слою в качестве нового входа и вновь пропускается через нейронную сеть в прямом направлении.

RBM выполняет серию проходов в прямом и обратном направлении, обучаясь оптимальным весам, которые могут привести к построению надежной порождающей модели. Ограниченные машины Больцмана — первый тип порождающих моделей, который мы исследуем. Выполняя семплирование по Гиббсу и многократную тренировку весов с помощью процессов прямого и обратного распространения, RBM пытается обучиться *распределению вероятностей* оценок во входных данных. В частности, RBM минимизирует *расстояние Кульбака — Лейблера*, которое служит мерой расхождения между различными распределениями вероятностей. В нашем случае RBM минимизирует расхождение между распределением вероятностей реконструированных и входных данных.

Итеративно перенастраивая веса в нейронной сети, RBM обучается апроксимировать оригинальные данные настолько точно, насколько это возможно.

Ограниченные машины Больцмана, обученные этому новому распределению вероятностей, способны генерировать прогнозы в отношении ранее не встречавшихся данных. Проектируемая нами сеть будет пытаться предсказывать рейтинги фильмов, которые пользователь еще не видел, на основании сходства данного пользователя с остальными пользователями и оценок фильмов, полученных от других пользователей.

Создание класса RBM

Конструктору класса RBM передается несколько параметров, а именно: размер входа RBM (input_size), размер выхода (output_size), скорость обучения (learning_rate), количество эпох обучения (epochs) и размер пакетов, используемых в процессе обучения (batch_size).

Мы также создадим нулевые матрицы для весов и векторов смещений скрытого и видимого слоев.

```
# Определение класса RBM
class RBM(object):

    def __init__(self, input_size, output_size, learning_rate, \
                 epochs, batchsize):
        # Определение гиперпараметров
        self._input_size = input_size
        self._output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batchsize = batchsize

        # Инициализация весов и смещений нулевыми матрицами
        self.w = np.zeros([input_size, output_size], "float")
        self.hb = np.zeros([output_size], "float")
        self.vb = np.zeros([input_size], "float")
```

Далее определим функции для передачи данных в прямом и обратном направлениях и семплирования данных в процессе прохождения этих этапов.

Вот как выглядит функция для передачи данных в прямом направлении (h обозначает скрытый слой, а v — видимый).

```
    def prob_h_given_v(self, visible, w, hb):
        return tf.nn.sigmoid(tf.matmul(visible, w) + hb)
```

Аналогичная функция для передачи данных в обратном направлении выглядит так.

```
    def prob_v_given_h(self, hidden, w, vb):
        return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)
```

Функция семплирования имеет следующий вид.

```
    def sample_prob(self, probs):
        return tf.nn.relu(tf.sign(probs - \
                          tf.random_uniform(tf.shape(probs))))
```

Теперь нам нужна функция, реализующая процесс обучения. Поскольку мы используем библиотеку TensorFlow, мы прежде всего должны создать заместители для графа TensorFlow, которые впоследствии будут применяться при передаче данных сеансу TensorFlow[2].

Нам потребуются заместители для матрицы весов, а также векторов смещений скрытого и видимого слоев. Кроме того, мы должны инициализировать все эти заместители нулевыми значениями, а также создать один набор для хранения текущих значений и еще один — для хранения предыдущих значений.

```
def train(self, X):
    _w = tf.placeholder("float", [self._input_size, \
                        self._output_size])
    _hb = tf.placeholder("float", [self._output_size])
    _vb = tf.placeholder("float", [self._input_size])

    prv_w = np.zeros([self._input_size, self._output_size], \
                     "float")
    prv_hb = np.zeros([self._output_size], "float")
    prv_vb = np.zeros([self._input_size], "float")

    cur_w = np.zeros([self._input_size, self._output_size], \
                     "float")
    cur_hb = np.zeros([self._output_size], "float")
    cur_vb = np.zeros([self._input_size], "float")
```

Еще нам нужен заместитель для видимого слоя. Скрытый слой формируется путем умножения матрицы видимого слоя на матрицу весов и прибавления к полученному результату вектора смещений скрытого слоя.

```
    v0 = tf.placeholder("float", [None, self._input_size])
    h0 = self.sample_prob(self.prob_h_given_v(v0, _w, _hb))
```

В процессе обратного распространения ошибки мы берем выход скрытого слоя, умножаем его на транспонированную матрицу весов, которая использовалась в процессе прямого распространения, и прибавляем к полученному результату вектор смещений видимого слоя. Подчеркнем, что в процессах прямого и обратного распространения используется одна и та же матрица весов. Затем мы вновь повторяем процесс прямого распространения.

```
    v1 = self.sample_prob(self.prob_v_given_h(h0, _w, _vb))
    h1 = self.prob_h_given_v(v1, _w, _hb)
```

[2] Этот пример ориентирован на TensorFlow 1.*x*. — *Примеч. ред.*

Веса обновляются с применением контрастивной дивергенции[3]. Кроме того, мы определяем MSE в качестве меры ошибки.

```
positive_grad = tf.matmul(tf.transpose(v0), h0)
negative_grad = tf.matmul(tf.transpose(v1), h1)

update_w = _w + self.learning_rate * \
           (positive_grad - negative_grad) / \
           tf.to_float(tf.shape(v0)[0])
update_vb = _vb + self.learning_rate * \
            tf.reduce_mean(v0 - v1, 0)
update_hb = _hb + self.learning_rate * \
            tf.reduce_mean(h0 - h1, 0)

err = tf.reduce_mean(tf.square(v0 - v1))
```

Выполнив указанные действия, мы подготовились к тому, чтобы инициализировать сеанс TensorFlow с помощью только что созданных переменных.

Как только будет выполнен вызов `sess.run`, мы сможем начать процесс обучения нейронной сети, передавая ей пакеты данных. На этапе обучения мы делаем проходы в прямом и обратном направлениях, а ограниченная машина Больцмана обновляет веса на основании сравнения сгенерированных данных с исходными входными данными. Мы будем выводить ошибку реконструкции для каждой эпохи.

```
error_list = []
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch in range(self.epochs):
        for start, end in zip(range(0, len(X), \
                                    self.batchsize), \
                              range(self.batchsize, len(X), \
                              self.batchsize)):
            batch = X[start:end]
            cur_w = sess.run(update_w, feed_dict={v0: batch, \
                _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            cur_hb = sess.run(update_hb, feed_dict={v0: \
                batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
```

[3] Для получения более подробной информации по этой теме обратитесь к статье *On Contrastive Divergence Learning* (`http://bit.ly/2RukFuX`).

```
                cur_vb = sess.run(update_vb, feed_dict={v0: \
                    batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
                prv_w = cur_w
                prv_hb = cur_hb
                prv_vb = cur_vb
            error = sess.run(err, feed_dict={v0: X, _w: cur_w, \
                _vb: cur_vb, _hb: cur_hb})
            print('Эпоха: %d' % epoch, \
                  'ошибка реконструкции: %f' % error)
            error_list.append(error)
        self.w = prv_w
        self.hb = prv_hb
        self.vb = prv_vb
        return error_list
```

Тренировка рекомендательной системы с использованием RBM-модели

Для обучения RBM мы создадим массив NumPy `inputX` на основе матрицы оценок `ratings_train` и преобразуем его значения в вещественные числа типа `float32`. Мы определим ограниченную машину Больцмана с размерностями входа и выхода, равными 1000, зададим скорость обучения `0.3` и проведем тренировку на протяжении 500 эпох с применением пакетов размером 200. Это всего лишь предварительный вариант выбора параметров. Желательно, чтобы вы провели собственные эксперименты для поиска более оптимальных значений параметров.

```
# Начало тренировочного цикла

# Приведение массива inputX к типу float32
inputX = ratings_train
inputX = inputX.astype(np.float32)

# Определение параметров RBM
rbm = RBM(1000, 1000, 0.3, 500, 200)
```

Начнем тренировку.

```
rbm.train(inputX)
outputX, reconstructedX, hiddenX = rbm.rbm_output(inputX)
```

График ошибок реконструкции приведен на рис. 10.4.

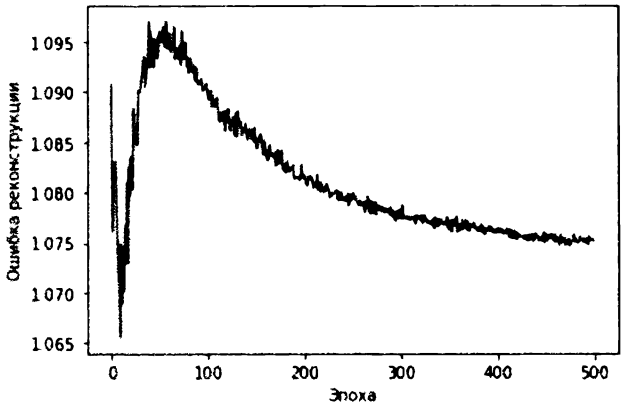


***Рис. 10.4.** График ошибок реконструкции RBM*

С увеличением длительности тренировки эта ошибка реконструкции обычно уменьшается.

А теперь используем обученную модель для предсказания рейтингов фильмов на основе валидационного набора (который включает тех же пользователей, что и тренировочный набор).

```
# Предсказание рейтингов для валидационного набора
inputValidation = ratings_validation
inputValidation = inputValidation.astype(np.float32)

finalOutput_validation, reconstructedOutput_validation, _ = \
    rbm.rbm_output(inputValidation)
```

Преобразуем предсказания в массив и вычислим MSE относительно истинных оценок валидационного набора.

```
predictionsArray = reconstructedOutput_validation
pred_validation = \
    predictionsArray[ratings_validation.nonzero()].flatten()
actual_validation = \
    ratings_validation[ratings_validation.nonzero()].flatten()

rbm_prediction = mean_squared_error(pred_validation, \
    actual_validation)
print('Среднеквадратическая ошибка с использованием \
      предсказания RBM': rbm_prediction)
```

Выведем показатель MSE для валидационного набора.

```
Среднеквадратическая ошибка с использованием предсказания RBM:
9.331135003325205
```

Полученное значение MSE может служить в качестве отправной точки, и вполне вероятно, что его можно улучшить, проведя дополнительные эксперименты.

Резюме

В этой главе мы исследовали, как работают ограниченные машины Больцмана, и применили их для создания рекомендательной системы фильмов. Построенная нами система обучалась распределению вероятностей рейтингов на основании имеющихся оценок, присвоенных данным пользователем и другими пользователями со схожими предпочтениями. Обученная нейронная сеть приобрела способность предсказывать рейтинги фильмов, которые еще не были просмотрены пользователями.

В главе 11 мы будем создавать глубокие сети доверия путем формирования каскада ограниченных машин Больцмана и применять их для решения еще более сложных задач на основе обучения без учителя.

ГЛАВА 11

Обнаружение признаков с помощью глубоких сетей доверия

В главе 10 мы исследовали ограниченные машины Больцмана (RBM) и использовали их для построения рекомендательной системы фильмов. В этой главе мы объединим несколько RBM в каскад для создания *глубокой сети доверия* (deep belief network — DBN). Впервые такой тип сетей был предложен Джеффри Хинтоном из университета Торонто в 2006 году.

В RBM имеется всего два слоя: видимый и скрытый. Другими словами, это мелкая нейронная сеть. Глубокие сети доверия образуются из множества RBM: скрытый слой одной RBM служит видимым слоем следующей. В результате DBN превращается в глубокую нейронную сеть. Это первый тип глубоких сетей обучения без учителя, с которым мы познакомимся.

Мелкие нейронные сети наподобие RBM не способны обучаться внутренней структуре таких сложных данных, как изображения, звук и текст, в отличие от DBN. Глубокие сети доверия применяются для распознавания и кластеризации изображений, захвата видео, звука и текста, хотя за последнее десятилетие были разработаны другие методы глубокого обучения, превосходящие DBN по своей производительности.

Что собой представляют глубокие сети доверия

Подобно RBM, глубокие сети доверия могут обучаться базовой структуре входных данных и реконструировать их на вероятностной основе. Другими словами, DBN, как и RBM, — порождающие модели. Слои в DBN связаны только между собой и не содержат внутренних связей между узлами одного слоя.

В DBN слои обучаются поочередно, начиная с самого первого скрытого слоя, который вместе с входным слоем образует первую RBM. Как только первая RBM обучена, ее скрытый слой превращается в видимый слой следующей RBM и используется для обучения второго скрытого слоя DBN.

Описанный процесс продолжается до тех пор, пока не будут обучены все слои DBN. Каждый слой DBN, за исключением первого и последнего, выступает в качестве как скрытого, так и видимого слоя RBM.

DBN — это иерархия представлений, которая, подобно всем нейронным сетям, применяется для обучения признакам. Следует отметить, что в глубоких сетях доверия нет никаких меток. Вместо этого они обучаются базовой структуре входных данных слой за слоем.

Метки могут использоваться для тонкой настройки нескольких последних слоев DBN, но только после того, как будет завершено начальное обучение без учителя. Например, если мы хотим применить DBN в качестве классификатора, то сначала должны выполнить обучение без учителя (*предварительное обучение*) и только затем перейти к тонкой настройке DBN.

Классификация изображений MNIST

Приступим к построению классификатора изображений на основе глубокой сети доверия. Для этого мы вновь обратимся к набору данных MNIST.

Сначала загрузим необходимые библиотеки.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip, datetime

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl
from mpl_toolkits.axes_grid1 import Grid

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss, accuracy_score
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
```

```
from sklearn.metrics import roc_curve, auc, roc_auc_score, \
    mean_squared_error

'''Алгоритмы'''
import lightgbm as lgb

'''TensorFlow (1.x) и Keras'''
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import keras
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout
from keras.layers import BatchNormalization, Input, Lambda
from keras.layers import Embedding, Flatten, dot
from keras import regularizers
from keras.losses import mse, binary_crossentropy
```

Теперь загрузим данные и сохраним их в объектах `DataFrame` библиотеки Pandas. Мы также преобразуем метки в векторы с помощью прямого кодирования. Это аналогично тому, что мы делали, когда впервые начали работать с набором данных MNIST в предыдущих главах.

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'mnist_data', \
                         'mnist.pkl.gz'])
f = gzip.open(current_path+file, 'rb')
train_set, validation_set, test_set = pickle.load(f, \
    encoding='latin1')
f.close()

X_train, y_train = train_set[0], train_set[1]
X_validation, y_validation = validation_set[0], validation_set[1]
X_test, y_test = test_set[0], test_set[1]

# Создание объектов DataFrame библиотеки Pandas из наборов данных
train_index = range(0, len(X_train))
validation_index = range(len(X_train), len(X_train) + \
                         len(X_validation))
test_index = range(len(X_train) + len(X_validation), \
                   len(X_train) + len(X_validation) + len(X_test))
```

```
X_train = pd.DataFrame(data=X_train, index=train_index)
y_train = pd.Series(data=y_train, index=train_index)

X_validation = pd.DataFrame(data=X_validation, \
                            index=validation_index)
y_validation = pd.Series(data=y_validation, index=validation_index)

X_test = pd.DataFrame(data=X_test, index=test_index)
y_test = pd.Series(data=y_test, index=test_index)

def view_digit(X, y, example):
    label = y.loc[example]
    image = X.loc[example, :].values.reshape([28, 28])
    plt.title('Пример: %d Метка: %d' % (example, label))
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.show()

def one_hot(series):
    label_binarizer = pp.LabelBinarizer()
    label_binarizer.fit(range(max(series) + 1))
    return label_binarizer.transform(series)

# Создание векторов прямого кодирования для меток
y_train_oneHot = one_hot(y_train)
y_validation_oneHot = one_hot(y_validation)
y_test_oneHot = one_hot(y_test)
```

Ограниченные машины Больцмана

Далее мы создадим класс RBM, который позволит организовать быстрое последовательное обучение нескольких RBM (являющихся строительными блоками DBN).

Вспомните, что в ограниченной машине Больцмана есть входной (видимый) слой и единственный скрытый слой, а соединения между нейронами ограничены так, что нейроны одного слоя могут связываться только с нейронами других слоев, но не между собой. Кроме того, вспомните, что обмен данными между слоями происходит в обоих направлениях, а не только в направлении прямого распространения, как в случае автокодировщиков.

В RBM нейроны видимого слоя взаимодействуют со скрытым слоем, скрытый слой генерирует данные из вероятностного представления модели,

которому обучена RBM, после чего скрытый слой передает сгенерированную информацию обратно видимому слою. Видимый слой получает сгенерированные данные от скрытого слоя, семплирует их и сравнивает с исходными данными, после чего, основываясь на ошибке реконструкции между выборкой из сгенерированных данных и оригинальными данными, передает новую информацию скрытому слою, и весь процесс повторяется снова.

Описанный двунаправленный обмен данными позволяет создать на основе RBM порождающую модель, в которой реконструированные представления на выходе скрытого слоя воспроизводят оригинальные входные данные.

Создание класса RBM

Пройдемся по различным компонентам класса RBM подобно тому, как мы это делали в главе 10.

Прежде всего мы инициализируем ряд параметров данного класса, а именно: входной размер RBM (`_input_size`), выходной размер (`_output_size`), скорость обучения (`learning_rate`), количество эпох обучения (`epochs`) и размер пакетов, используемых в процессе обучения (`batchsize`). Мы также создадим нулевые матрицы для весов и векторов смещений скрытого и видимого слоев.

```
# Определение класса RBM
class RBM(object):

    def __init__(self, input_size, output_size, learning_rate, \
                 epochs, batchsize):
        # Определение гиперпараметров
        self._input_size = input_size
        self._output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batchsize = batchsize

        # Инициализация весов и смещений нулевыми матрицами
        self.w = np.zeros([input_size, output_size], "float")
        self.hb = np.zeros([output_size], "float")
        self.vb = np.zeros([input_size], "float")
```

Далее определим функции для передачи данных в прямом и обратном направлениях и семплирования данных в процессе прохождения этих этапов.

Вот как выглядит функция для передачи данных в прямом направлении (h обозначает скрытый слой, а v — видимый).

```
def prob_h_given_v(self, visible, w, hb):
    return tf.nn.sigmoid(tf.matmul(visible, w) + hb)
```

Аналогичная функция для передачи данных в обратном направлении выглядит так.

```
def prob_v_given_h(self, hidden, w, vb):
    return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)
```

Функция семплирования имеет следующий вид.

```
def sample_prob(self, probs):
    return tf.nn.relu(tf.sign(probs - \
                      tf.random_uniform(tf.shape(probs))))
```

Нам также нужна функция, реализующая процесс обучения. Поскольку мы используем библиотеку TensorFlow, мы прежде всего должны создать заместители для графа TensorFlow, посредством которых будем передавать данные сеансу TensorFlow[1].

Нам потребуются заместители для матрицы весов, а также векторов смещений скрытого и видимого слоев. Все три заместителя должны быть инициализированы нулями. Кроме того, нам нужно создать один набор для хранения текущих значений и еще один — для хранения предыдущих значений.

```
def train(self, X):
    _w = tf.placeholder("float", [self._input_size, \
                        self._output_size])
    _hb = tf.placeholder("float", [self._output_size])
    _vb = tf.placeholder("float", [self._input_size])

    prv_w = np.zeros([self._input_size, self._output_size], \
                     "float")
    prv_hb = np.zeros([self._output_size], "float")
    prv_vb = np.zeros([self._input_size], "float")

    cur_w = np.zeros([self._input_size, self._output_size], \
                     "float")
    cur_hb = np.zeros([self._output_size], "float")
    cur_vb = np.zeros([self._input_size], "float")
```

[1] Этот пример ориентирован на TensorFlow 1.*x*. — *Примеч. ред.*

Еще нам нужен заместитель для видимого слоя. Скрытый слой формируется путем умножения матрицы видимого слоя на матрицу весов и прибавления к полученному результату вектора смещений скрытого слоя.

```
v0 = tf.placeholder("float", [None, self._input_size])
h0 = self.sample_prob(self.prob_h_given_v(v0, _w, _hb))
```

В процессе обратного распространения ошибки мы берем выход скрытого слоя, умножаем его на транспонированную матрицу весов, которая использовалась в процессе прямого распространения, и прибавляем к полученному результату вектор смещений видимого слоя. Подчеркнем, что в процессах прямого и обратного распространения используется одна и та же матрица весов.

Затем мы вновь повторяем процесс прямого распространения.

```
v1 = self.sample_prob(self.prob_v_given_h(h0, _w, _vb))
h1 = self.prob_h_given_v(v1, _w, _hb)
```

Веса обновляются с применением контрастивной дивергенции, которая обсуждалась в главе 10. Кроме того, мы определяем MSE в качестве функции потерь.

```
positive_grad = tf.matmul(tf.transpose(v0), h0)
negative_grad = tf.matmul(tf.transpose(v1), h1)

update_w = _w + self.learning_rate * \
           (positive_grad - negative_grad) /
           tf.to_float(tf.shape(v0)[0])
update_vb = _vb + self.learning_rate * \
            tf.reduce_mean(v0 - v1, 0)
update_hb = _hb + self.learning_rate * \
            tf.reduce_mean(h0 - h1, 0)

err = tf.reduce_mean(tf.square(v0 - v1))
```

Выполнив указанные действия, мы подготовились к тому, чтобы инициализировать сеанс TensorFlow с помощью только что созданных переменных.

Как только будет выполнен вызов `sess.run`, мы сможем начать процесс обучения нейронной сети, передавая ей пакеты данных. На этапе обучения мы делаем проходы в прямом и обратном направлениях, а ограниченная машина Больцмана обновляет веса на основании сравнения сгенерированных данных с исходными входными данными. Мы будем выводить ошибку реконструкции для каждой эпохи.

```
        error_list = []
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())

            for epoch in range(self.epochs):
                for start, end in zip(range(0, len(X), \
                                            self.batchsize), \
                                      range(self.batchsize, len(X), \
                                            self.batchsize)):
                    batch = X[start:end]
                    cur_w = sess.run(update_w, \
                        feed_dict={v0: batch, _w: prv_w, \
                                   _hb: prv_hb, _vb: prv_vb})
                    cur_hb = sess.run(update_hb, \
                        feed_dict={v0: batch, _w: prv_w, \
                                   _hb: prv_hb, _vb: prv_vb})
                    cur_vb = sess.run(update_vb, \
                        feed_dict={v0: batch, _w: prv_w, \
                                   _hb: prv_hb, _vb: prv_vb})
                    prv_w = cur_w
                    prv_hb = cur_hb
                    prv_vb = cur_vb
                error = sess.run(err, feed_dict={v0: X, _w: cur_w, \
                    _vb: cur_vb, _hb: cur_hb})
                print('Эпоха: %d' % epoch, \
                      'ошибка реконструкции: %f' % error)
                error_list.append(error)
            self.w = prv_w
            self.hb = prv_hb
            self.vb = prv_vb
            return error_list
```

Генерирование изображений с использованием RBM-модели

Определим функцию, генерирующую новые изображения на основе порождающей модели, которой обучилась RBM.

```
    def rbm_output(self, X):
        input_X = tf.constant(X)
        _w = tf.constant(self.w)
        _hb = tf.constant(self.hb)
        _vb = tf.constant(self.vb)
        out = tf.nn.sigmoid(tf.matmul(input_X, _w) + _hb)
```

```
    hiddenGen = self.sample_prob(self.prob_h_given_v(input_X, \
                                                     _w, _hb))
    visibleGen = self.sample_prob(self.prob_v_given_h(hiddenGen, \
                                                      _w, _vb))
    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        return sess.run(out), sess.run(visibleGen), \
               sess.run(hiddenGen)
```

Этой функции передается оригинальная матрица изображений, X. Мы создаем заместители TensorFlow для нее и матрицы весов, а также для векторов смещений скрытого и видимого слоев. Затем мы используем входную матрицу для получения выходных данных на этапе прямого распространения (`out`), для создания выборки скрытого слоя (`hiddenGen`) и выборки реконструированных изображений, сгенерированных моделью (`visibleGen`).

Просмотр содержимого промежуточных детекторов признаков

Наконец, определим функцию для вывода содержимого детекторов признаков скрытого слоя.

```
def show_features(self, shape, suptitle, count=-1):
    maxw = np.amax(self.w.T)
    minw = np.amin(self.w.T)
    count = self._output_size if count == -1 or count > \
            self._output_size else count
    ncols = count if count < 14 else 14
    nrows = count // ncols
    nrows = nrows if nrows > 2 else 3
    fig = plt.figure(figsize=(ncols, nrows), dpi=100)
    grid = Grid(fig, rect=111, nrows_ncols=(nrows, ncols), \
                axes_pad=0.01)

    for i, ax in enumerate(grid):
        x = self.w.T[i] if i < self._input_size else \
            np.zeros(shape)
        x = (x.reshape(1, -1) - minw) / maxw
        ax.imshow(x.reshape(*shape), cmap=mpl.cm.Greys)
        ax.set_axis_off()

    fig.text(0.5, 1, suptitle, font size=20, \
```

```
                horizontalalignment='center')
    fig.tight_layout()
    plt.show()
    return
```

Эта и остальные функции будут использованы для работы с набором данных MNIST.

Обучение трех RBM, образующих глубокую сеть доверия

Далее мы будем последовательно обучать три RBM на наборе данных MNIST таким образом, чтобы скрытый слой одной RBM становился видимым слоем следующей RBM. Эти три RBM образуют глубокую сеть доверия, которую мы строим для классификации изображений.

Прежде всего сохраним тренировочные данные в массиве NumPy. Затем мы создадим список `rbm_list`, предназначенный для хранения обучаемых RBM, и определим гиперпараметры для всех трех RBM, а именно: входной размер (`_input_size`), выходной размер (`_output_size`), скорость обучения (`learning_rate`), количество эпох обучения (`epochs`) и размер пакетов, используемых в процессе обучения (`batchsize`).

Все эти операции выполняются с помощью созданного нами класса `RBM`.

Первая RBM будет получать оригинальный 784-мерный входной набор и создавать выходную матрицу размерностью 700. Следующая RBM будет использовать 700-мерную выходную матрицу первой RBM и создавать на выходе 600-мерную матрицу. Наконец, последняя из RBM будет получать 600-мерную матрицу и создавать на выходе 500-мерную матрицу.

Все три RBM будут тренироваться со скоростью обучения `1.0` в течение 100 эпох при размере пакета, равном 200.

```
# Задание входных тренировочных данных
inputX = np.array(X_train)

# Создание списка для хранения ограниченных машин Больцмана
rbm_list = []

# Определение параметров обучаемых RBM
rbm_list.append(RBM(784, 700, 1.0, 100, 200))
rbm_list.append(RBM(700, 600, 1.0, 100, 200))
rbm_list.append(RBM(600, 500, 1.0, 100, 200))
```

Теперь можем приступить к обучению RBM. Мы будем хранить обученные RBM в списке `outputList`.

Обратите внимание на то, что для получения выходной матрицы скрытого слоя, которая будет использоваться в качестве входного/видимого слоя следующей RBM, мы используем созданную ранее функцию `rbm_output`.

```
outputList = []
error_list = []
# Для каждой RBM из нашего списка
for i in range(0, len(rbm_list)):
    print('RBM', i+1)
    # Обучение новой RBM
    rbm = rbm_list[i]
    err = rbm.train(inputX)
    error_list.append(err)
    # Возврат выходного слоя
    outputX, reconstructedX, hiddenX = rbm.rbm_output(inputX)
    outputList.append(outputX)
    inputX = hiddenX
```

Чем дольше тренируются RBM, тем меньше ошибки реконструкции (рис. 11.1–11.3). Отметим, что ошибка реконструкции отражает степень сходства между реконструированными данными конкретной RBM и данными, поступающими на вход этой же RBM.

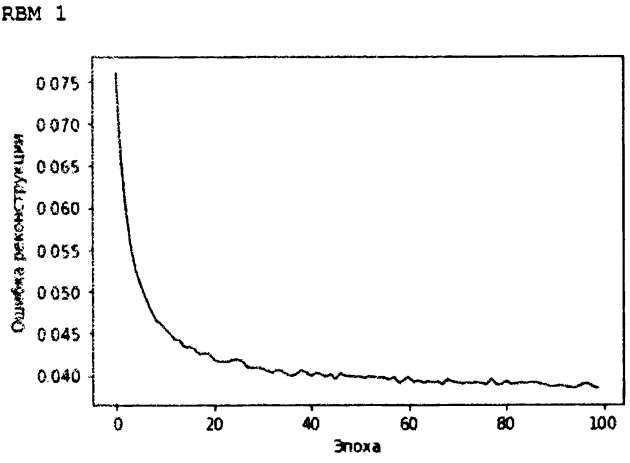


Рис. 11.1. *Ошибки реконструкции первой RBM*

```
RBM 2
```

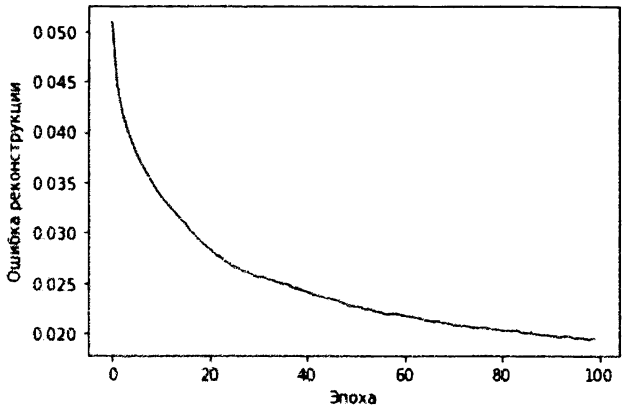


***Рис. 11.2.** Ошибки реконструкции второй RBM*

```
RBM 3
```

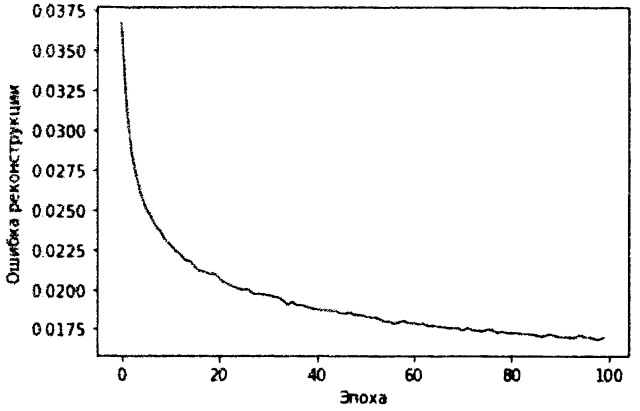


***Рис. 11.3.** Ошибки реконструкции третьей RBM*

Проверка детекторов признаков

Посмотрим, как выглядят обученные признаки каждой RBM, используя созданную ранее функцию `show_features`.

```
rbm_shapes = [(28, 28), (25, 24), (25, 20)]
for i in range(0, len(rbm_list)):
    rbm = rbm_list[i]
    print(rbm.show_features(rbm_shapes[i], \
          "Признаки из MNIST, которым обучилась RBM", 56))
```

Соответствующие результаты для каждой RBM представлены на рис. 11.4.

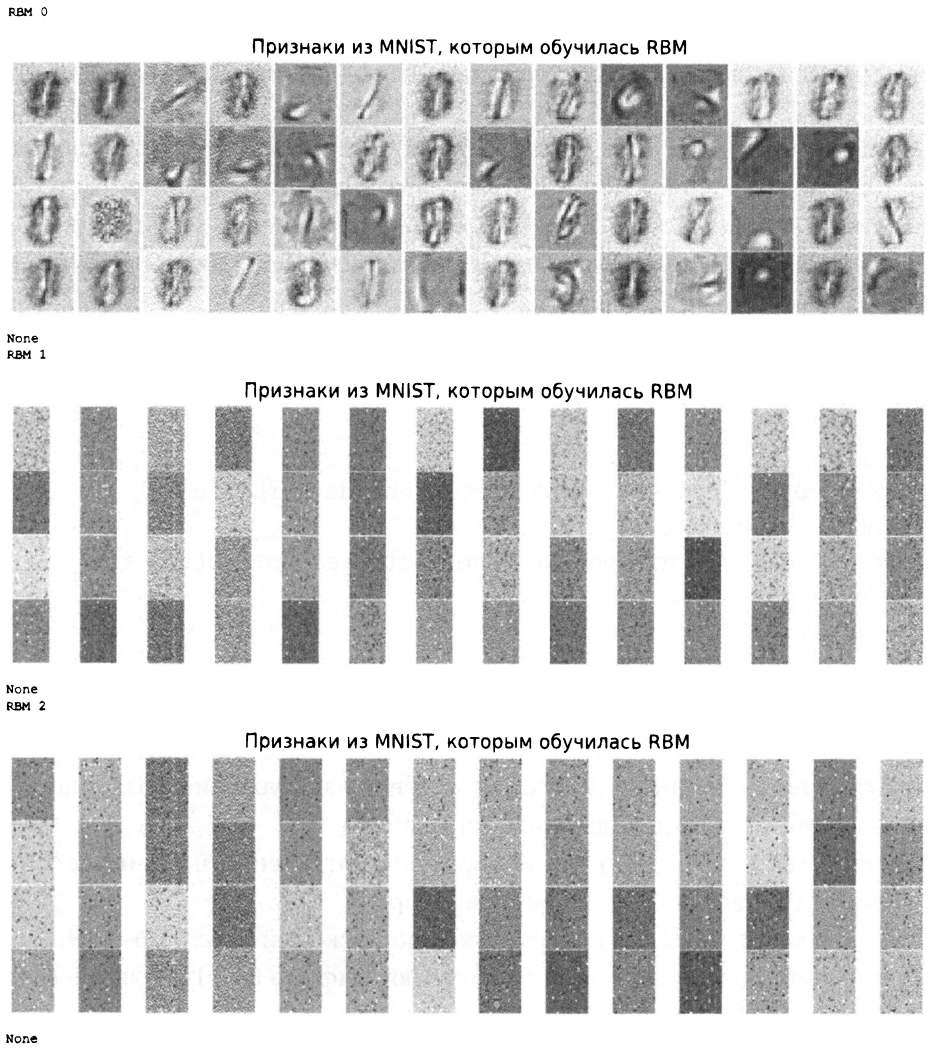


Рис. 11.4. *Признаки, которым обучилась каждая RBM*

Как видите, каждая RBM обучается все более абстрактным признакам данных MNIST. Если в признаках первой RBM еще можно заметить какое-то слабое сходство с цифрами, то признаки второго и третьего слоев отличаются все большей абстракцией и меньшей различимостью. Такая ситуация довольно типична в сфере обработки изображений: чем глубже расположен слой нейронной сети, тем более абстрактные признаки оригинальных изображений он распознает.

Просмотр сгенерированных изображений

Прежде чем приступить к построению глубокой сети доверия, посмотрим, как выглядят изображения, сгенерированные одной из только что обученных RBM.

Чтобы не усложнять пример, мы передадим оригинальную тренировочную матрицу MNIST первой RBM, которая выполнит проходы в прямом и обратном направлениях, а затем создаст нужные нам сгенерированные изображения. Мы сравним первые десять изображений из набора данных MNIST с новыми изображениями.

```
inputX = np.array(X_train)
rbmOne = rbm_list[0]

print('RBM 1')
outputX_rbmOne, reconstructedX_rbmOne, hiddenX_rbmOne = \
    rbmOne.rbm_output(inputX)
reconstructedX_rbmOne = pd.DataFrame(data=reconstructedX_rbmOne, \
    index=X_train.index)
for j in range(0, 10):
    example = j
    view_digit(reconstructedX, y_train, example)
    view_digit(X_train, y_train, example)
```

Для сравнения на рис. 11.5 показано первое изображение, созданное RBM, и первое изображение из оригинального набора.

Как видите, сгенерированное изображение отдаленно напоминает оригинал: в обоих случаях можно различить цифру 5.

Сравним между собой еще несколько изображений (рис. 11.6–11.9).

Приведенные изображения соответствуют цифрам 0, 4, 1 и 9, причем сгенерированные изображения вполне узнаваемы.

Полноценная DBN

Создадим теперь класс `DBN`, который будет получать три обученные RBM и добавлять четвертую RBM, выполняющую проходы в прямом и обратном направлениях с целью уточнения общей порождающей модели на основе DBN.

Прежде всего определим гиперпараметры класса. В их число входят размер оригинального входного набора (`_original_input_size`), размер входа третьей из обученных RBM (`_input_size`), конечный размер выхода DBN, который мы хотим получить (`_output_size`), скорость обучения

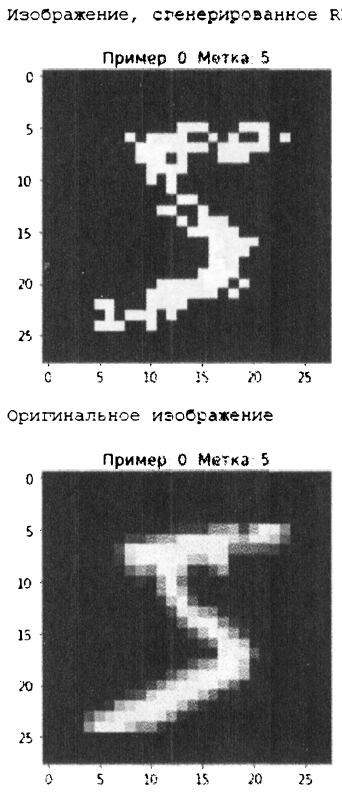


Рис. 11.5. Первое изображение, сгенерированное первой RBM

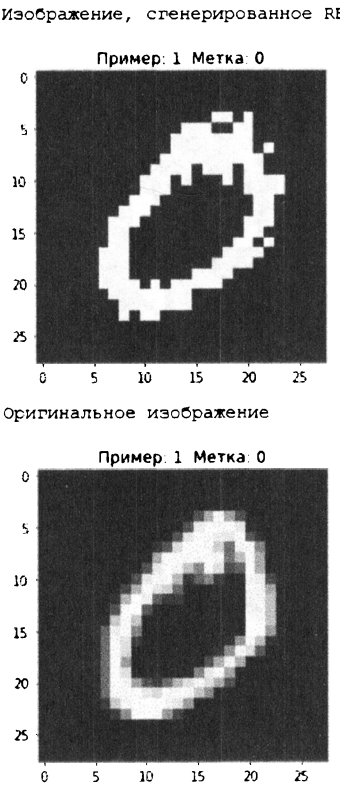


Рис. 11.6. Второе изображение, сгенерированное первой RBM

(learning_rate), количество эпох обучения (epochs), размер пакетов, используемых в процессе обучения (batch_size), и три RBM, которые мы обучили. Как и прежде, нам потребуются нулевые матрицы для инициализации весов, а также векторов смещений скрытого и видимого слоев.

```
class DBN(object):
    def __init__(self, original_input_size, input_size,
                 output_size, learning_rate, epochs, batchsize,
                 rbmOne, rbmTwo, rbmThree):
        # Определение гиперпараметров
        self._original_input_size = original_input_size
        self._input_size = input_size
        self._output_size = output_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batchsize = batchsize
```

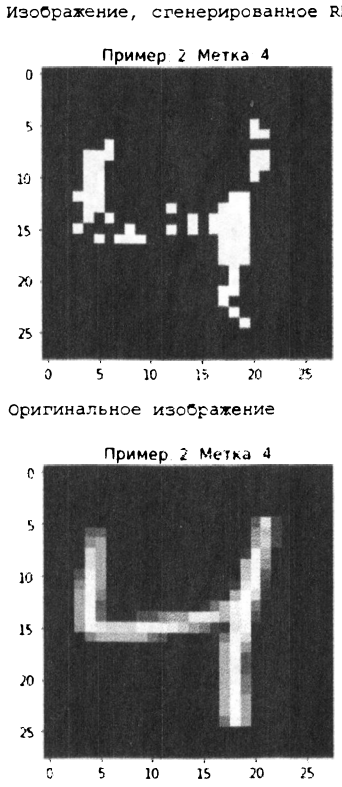


Рис. 11.7. Третье изображение, сгенерированное первой RBM

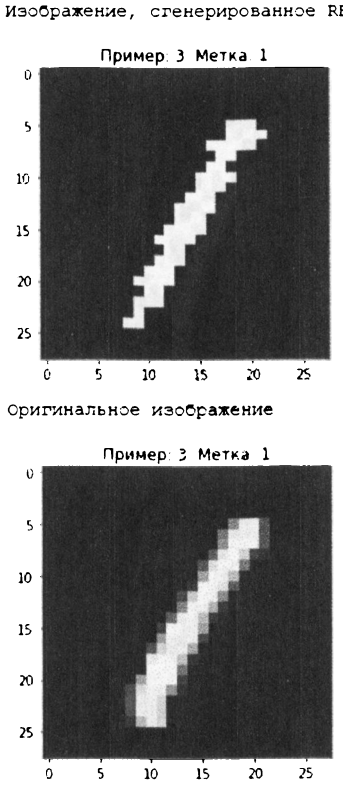


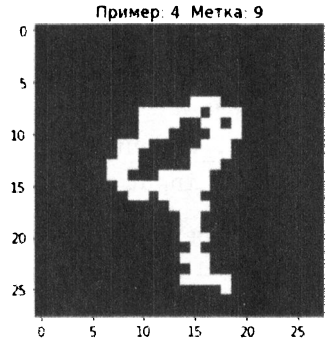
Рис. 11.8. Четвертое изображение, сгенерированное первой RBM

```
        self.rbmOne = rbmOne
        self.rbmTwo = rbmTwo
        self.rbmThree = rbmThree

        self.w = np.zeros([input_size, output_size], "float")
        self.hb = np.zeros([output_size], "float")
        self.vb = np.zeros([input_size], "float")
```

Далее определим функции, предназначенные для выполнения проходов в прямом и обратном направлениях, а также формирования соответствующих выборок.

```
    def prob_h_given_v(self, visible, w, hb):
        return tf.nn.sigmoid(tf.matmul(visible, w) + hb)

    def prob_v_given_h(self, hidden, w, vb):
        return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)
```

Изображение, сгенерированное RBM



Оригинальное изображение

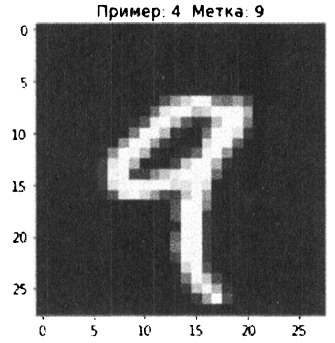


Рис. 11.9. *Пятое изображение, сгенерированное первой RBM*

```
def sample_prob(self, probs):
    return tf.nn.relu(tf.sign(probs - \
        tf.random_uniform(tf.shape(probs))))
```

Чтобы начать процесс обучения, нам потребуются заместители для весов, а также векторов смещений скрытого и видимого слоев. Кроме того, нам нужно создать один набор для хранения текущих значений и еще один — для хранения предыдущих значений.

```
def train(self, X):
    _w = tf.placeholder("float", [self._input_size, \
                        self._output_size])
    _hb = tf.placeholder("float", [self._output_size])
    _vb = tf.placeholder("float", [self._input_size])

    prv_w = np.zeros([self._input_size, self._output_size], \
                     "float")
```

```
prv_hb = np.zeros([self._output_size], "float")
prv_vb = np.zeros([self._input_size], "float")

cur_w = np.zeros([self._input_size, self._output_size], \
                 "float")
cur_hb = np.zeros([self._output_size], "float")
cur_vb = np.zeros([self._input_size], "float")
```

Далее определим заместитель для видимого слоя. Мы берем начальные входные данные — видимый слой — и пропускаем их через три RBM, которые перед этим были обучены. Полученная в результате выходная матрица `forward` передается четвертой RBM, которую мы обучаем в рамках класса DBN.

```
v0 = tf.placeholder("float", [None, \
                    self._original_input_size])
forwardOne = tf.nn.relu(tf.sign(tf.nn.sigmoid(tf.matmul(v0, \
    self.rbmOne.w) + self.rbmOne.hb) - tf.random_uniform( \
    tf.shape(tf.nn.sigmoid(tf.matmul(v0, self.rbmOne.w) + \
    self.rbmOne.hb)))))
forwardTwo = tf.nn.relu(tf.sign(tf.nn.sigmoid(tf.matmul( \
    forwardOne, self.rbmTwo.w) + self.rbmTwo.hb) - \
    tf.random_uniform(tf.shape(tf.nn.sigmoid(tf.matmul( \
    forwardOne, self.rbmTwo.w) + self.rbmTwo.hb)))))
forward = tf.nn.relu(tf.sign(tf.nn.sigmoid(tf.matmul( \
    forwardTwo, self.rbmThree.w) + self.rbmThree.hb) - \
    tf.random_uniform(tf.shape(tf.nn.sigmoid(tf.matmul( \
    forwardTwo, self.rbmThree.w) + self.rbmThree.hb)))))
h0 = self.sample_prob(self.prob_h_given_v(forward, _w, _hb))
v1 = self.sample_prob(self.prob_v_given_h(h0, _w, _vb))
h1 = self.prob_h_given_v(v1, _w, _hb)
```

Вновь, как и раньше, применим контрастивную дивергенцию.

```
positive_grad = tf.matmul(tf.transpose(forward), h0)
negative_grad = tf.matmul(tf.transpose(v1), h1)

update_w = _w + self.learning_rate * (positive_grad - \
           negative_grad) / tf.to_float(tf.shape(forward)[0])
update_vb = _vb + self.learning_rate * \
            tf.reduce_mean(forward - v1, 0)
update_hb = _hb + self.learning_rate * \
            tf.reduce_mean(h0 - h1, 0)
```

Совершив полный проход в прямом направлении через нашу DBN (которая включает три ранее обученные RBM, плюс самую последнюю, четвертую RBM), мы должны передать выход скрытого слоя четвертой RBM в обратном направлении по всей сети. Для этого нам потребуется выполнить процесс обратного распространения ошибки, причем как через четвертую RBM, так и через первые три. В качестве функции потерь опять используется MSE. Это делается следующим образом.

```
backwardOne = tf.nn.relu(tf.sign(tf.nn.sigmoid(tf.matmul(v1, \
    self.rbmThree.w.T) + self.rbmThree.vb) - \
    tf.random_uniform(tf.shape(tf.nn.sigmoid(tf.matmul(v1, \
    self.rbmThree.w.T) + self.rbmThree.vb)))))
backwardTwo = tf.nn.relu(tf.sign(tf.nn.sigmoid(tf.matmul( \
    backwardOne, self.rbmTwo.w.T) + self.rbmTwo.vb) - \
    tf.random_uniform(tf.shape(tf.nn.sigmoid(tf.matmul( \
    backwardOne, self.rbmTwo.w.T) + self.rbmTwo.vb)))))
backward = tf.nn.relu(tf.sign(tf.nn.sigmoid(tf.matmul( \
    backwardTwo, self.rbmOne.w.T) + self.rbmOne.vb) - \
    tf.random_uniform(tf.shape(tf.nn.sigmoid(tf.matmul( \
    backwardTwo, self.rbmOne.w.T) + self.rbmOne.vb)))))

err = tf.reduce_mean(tf.square(v0 - backward))
```

Ниже приведен код класса DBN, который отвечает за фактическое обучение модели. Он напоминает аналогичный код класса RBM.

```
error_list = []
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    for epoch in range(self.epochs):
        for start, end in zip(range(0, len(X), \
                                    self.batchsize), \
                              range(self.batchsize, len(X), \
                                    self.batchsize)):
            batch = X[start:end]
            cur_w = sess.run(update_w, feed_dict={v0: \
                batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            cur_hb = sess.run(update_hb, feed_dict={v0: \
                batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            cur_vb = sess.run(update_vb, feed_dict={v0: \
                batch, _w: prv_w, _hb: prv_hb, _vb: prv_vb})
            prv_w = cur_w
```

```
                prv_hb = cur_hb
                prv_vb = cur_vb
            error = sess.run(err, feed_dict={v0: X, _w: cur_w, \
                                _vb: cur_vb, _hb: cur_hb})
            print ('Эпоха: %d' % epoch, \
                   'ошибка реконструкции: %f' % error)
            error_list.append(error)
        self.w = prv_w
        self.hb = prv_hb
        self.vb = prv_vb
        return error_list
```

Определим функции, предназначенные для создания сгенерированных изображений на выходе DBN и отображения признаков. Эти операции аналогичны тем, которые выполнялись применительно к RBM, только мы пропускаем данные через все четыре ограниченные машины Больцмана, используемые в классе DBN, а не через одну.

```
def dbn_output(self, X):
    input_X = tf.constant(X)
    forwardOne = tf.nn.sigmoid(tf.matmul(input_X, \
                               self.rbmOne.w) + self.rbmOne.hb)
    forwardTwo = tf.nn.sigmoid(tf.matmul(forwardOne, \
                               self.rbmTwo.w) + self.rbmTwo.hb)
    forward = tf.nn.sigmoid(tf.matmul(forwardTwo, \
                            self.rbmThree.w) + self.rbmThree.hb)

    _w = tf.constant(self.w)
    _hb = tf.constant(self.hb)
    _vb = tf.constant(self.vb)

    out = tf.nn.sigmoid(tf.matmul(forward, _w) + _hb)
    hiddenGen = self.sample_prob(self.prob_h_given_v(forward, \
                                                     _w, _hb))
    visibleGen = self.sample_prob(self.prob_v_given_h(hiddenGen, \
                                                      _w, _vb))

    backwardTwo = tf.nn.sigmoid(tf.matmul(visibleGen, \
        self.rbmThree.w.T) + self.rbmThree.vb)
    backwardOne = tf.nn.sigmoid(tf.matmul(backwardTwo, \
        self.rbmTwo.w.T) + self.rbmTwo.vb)
    backward = tf.nn.sigmoid(tf.matmul(backwardOne, \
        self.rbmOne.w.T) + self.rbmOne.vb)
```

```
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            return sess.run(out), sess.run(backward)

    def show_features(self, shape, suptitle, count=-1):
        maxw = np.amax(self.w.T)
        minw = np.amin(self.w.T)
        count = self._output_size if count == -1 or count > \
            self._output_size else count
        ncols = count if count < 14 else 14
        nrows = count//ncols
        nrows = nrows if nrows > 2 else 3
        fig = plt.figure(figsize=(ncols, nrows), dpi=100)
        grid = Grid(fig, rect=111, nrows_ncols=(nrows, ncols), \
                    axes_pad=0.01)

        for i, ax in enumerate(grid):
            x = self.w.T[i] if i < \
                self._input_size else np.zeros(shape)
            x = (x.reshape(1, -1) - minw) / maxw
            ax.imshow(x.reshape(*shape), cmap=mpl.cm.Greys)
            ax.set_axis_off()

        fig.text(0.5, 1, suptitle, fontsize=20, \
                 horizontalalignment='center')
        fig.tight_layout()
        plt.show()
        return
```

Как происходит обучение DBN

Каждая из трех RBM, которые мы обучили, имеет собственную матрицу весов, а также векторы смещения скрытого и видимого слоя. В процессе обучения четвертой RBM мы не выполняем подстройку первых трех RBM. Вместо этого мы используем их как фиксированные компоненты DBN, предназначенные лишь для проходов в прямом и обратном направлениях (и формирования выборок из генерируемых ими данных).

В процессе тренировки четвертой RBM мы будем подстраивать лишь ее веса и смещения. Другими словами, четвертая RBM получает выходные данные от первых трех RBM и выполняет проходы в прямом и обратном направлениях

для обучения порождающей модели, минимизирующей ошибку реконструкции между сгенерированными и оригинальными изображениями.

В качестве альтернативы можно было бы обучать и подстраивать веса всех четырех RBM в процессе прямого и обратного распространения ошибки через всю сеть. Но такой способ обучения DBN оказывается слишком трудоемким с вычислительной точки зрения. Возможно, для современных компьютеров это не столь критично, однако по меркам 2006 года, когда глубокие сети доверия только появились, подобные вычисления обходились чересчур дорого.

Если бы мы все же захотели выполнить более тщательное предварительное обучение, то могли бы разрешить подстройку весов отдельных RBM (по одной RBM за раз) в процессе передачи пакетов в прямом и обратном направлениях по сети. Мы не будем заходить так далеко, но я рекомендую вам провести самостоятельные эксперименты.

Обучение DBN

Теперь можем приступить к обучению DBN. Мы установим для оригинальных изображений и выхода третьей RBM размерности 784 и 500 соответственно, а в качестве требуемой размерности DBN — 500. Мы используем скорость обучения `1.0`, будем тренировать сеть в течение 50 эпох и использовать пакеты размером 200. Наконец, мы будем вызывать первые три обученные RBM.

```
# Создание экземпляра класса DBN
dbn = DBN(784, 500, 500, 1.0, 50, 200, rbm_list[0], rbm_list[1], \
          rbm_list[2])
```

Обучим модель.

```
inputX = np.array(X_train)
error_list = []
error_list = dbn.train(inputX)
```

График ошибок реконструкции в процессе обучения DBN приведен на рис. 11.10.

На рис. 11.11 показаны обученные признаки, взятые из последнего слоя DBN, т.е. скрытого слоя четвертой RBM.

Все это напоминает результаты, полученные для отдельных RBM.

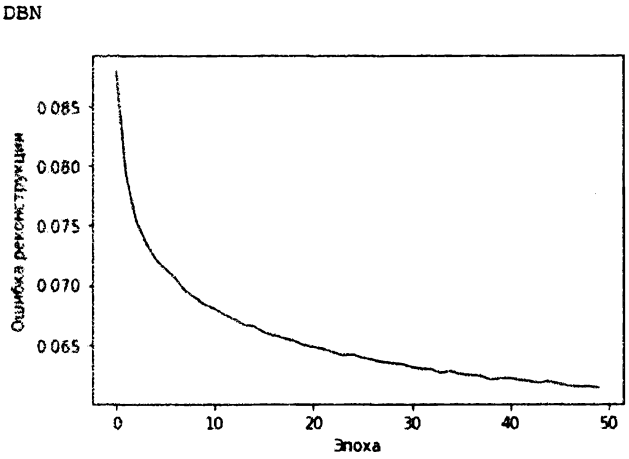


Рис. 11.10. *Ошибки реконструкции DBN*

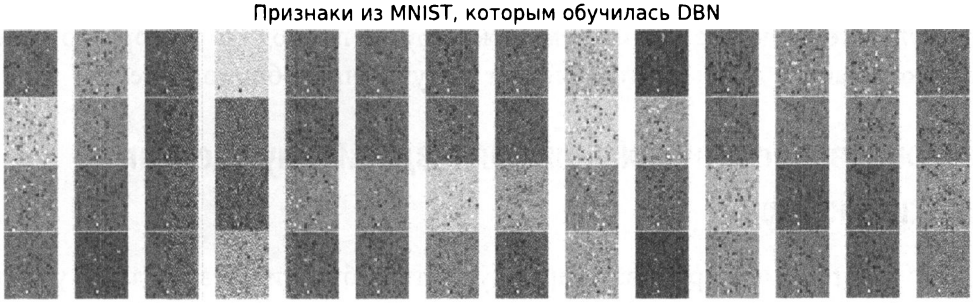


Рис. 11.11. *Обученные признаки четвертой RBM, входящей в состав DBN*

Как обучение без учителя может содействовать обучению с учителем

До сих пор все, что мы делали в процессе тренировки RBM и DBN, включало обучение без учителя. Никакие метки изображений вообще не использовались. Вместо этого мы строили порождающие модели, обучая их релевантным латентным признакам, извлекаемым из оригинальных изображений MNIST в составе тренировочного набора, который включает 50 000 примеров. Эти модели генерируют изображения, которые достаточно близки к оригинальным (минимизируют ошибку реконструкции).

Чтобы понять, в чем польза от порождающей модели, вернемся на шаг назад. Вспомните, что большая часть существующих в мире данных не марки-

рована никакими метками. Следовательно, несмотря на всю эффективность обучения с учителем, мы нуждаемся в обучении без учителя для работы с неразмеченными данными. Одного обучения с учителем оказывается недостаточно.

Представьте, что было бы, если бы вместо 50 000 маркированных изображений MNIST из тренировочного набора мы располагали лишь некоторой их частью, включающей, скажем, 5000 изображений. Классификатор изображений на основе обучения с учителем, имеющий доступ лишь к 5000 помеченных изображений, был бы далеко не так эффективен по сравнению с классификатором, которому доступно 50 000 изображений. Чем больше помеченных данных есть в нашем распоряжении, тем лучше будет работать система машинного обучения.

Каким же образом обучение без учителя может принести пользу в подобной ситуации? Одним из способов может стать генерирование новых размеченных примеров, дополняющих исходный размеченный набор изображений. В результате обучение с учителем можно будет проводить на гораздо более широком наборе данных, что приведет к улучшению общего решения.

Генерирование изображений для создания улучшенного классификатора

Чтобы продемонстрировать преимущества обучения без учителя, уменьшим наш тренировочный набор MNIST до всего лишь 5000 размеченных примеров и сохраним эти примеры во фрейме `inputXReduced`.

Затем, отталкиваясь от этих 5000 размеченных изображений, мы сгенерируем новые изображения с помощью нашей порождающей модели на основе DBN. Причем мы повторим процедуру 20 раз. Другими словами, мы сгенерируем 5000 новых изображений 20 раз для создания набора данных, содержащего 100 000 изображений, каждое из которых будет помеченным. С технической точки зрения мы сохраняем не сами реконструированные изображения, а лишь выходы последнего скрытого слоя. Впрочем, реконструированные изображения мы тоже сохраним, чтобы позже их можно было оценить.

Сохраним 100 000 результатов в массиве NumPy `generatedImages`.

```
# Генерирование и сохранение изображений
inputXReduced = X_train.loc[:4999]
for i in range(0, 20):
    print("Прогон", i)
```

```
    finalOutput_DBN, reconstructedOutput_DBN = \
        dbn.dbn_output(inputXReduced)
    if i==0:
        generatedImages = finalOutput_DBN
    else:
        generatedImages = np.append(generatedImages, \
                                    finalOutput_DBN, axis=0)
```

Затем запустим цикл из 20 итераций, обрабатывающий первые 5000 меток из тренировочного набора (`y_train`) для получения массива меток `labels`.

```
# Получение вектора меток для сгенерированных изображений
for i in range(0, 20):
    if i==0:
        labels = y_train.loc[:4999]
    else:
        labels = np.append(labels, y_train.loc[:4999])
```

Наконец, сгенерируем результаты по валидационному набору, чтобы впоследствии можно было оценить классификатор изображений, который мы вскоре создадим.

```
# Генерирование изображений на основе валидационного набора
inputValidation = np.array(X_validation)
finalOutput_DBN_validation, reconstructedOutput_DBN_validation = \
    dbn.dbn_output(inputValidation)
```

Прежде чем использовать только что сгенерированные данные, посмотрим, как выглядят некоторые из реконструированных изображений.

```
# Просмотр сгенерированных изображений
for i in range(0, 10):
    example = i
    reconstructedX = pd.DataFrame(data=reconstructedOutput_DBN, \
                                  index=X_train[0:5000].index)
    view_digit(reconstructedX, y_train, example)
    view_digit(X_train, y_train, example)
```

Как можно увидеть на рис. 11.12, сгенерированное изображение очень напоминает оригинал — в обоих случаях можно узнать цифру 5. В отличие от изображений, генерируемых ограниченной машиной Больцмана, которые мы просматривали ранее, в данном случае сходство с оригинальными изображениями MNIST более сильное. Переданы даже полутона.

Для сравнения просмотрим еще несколько изображений (рис. 11.13–11.16).

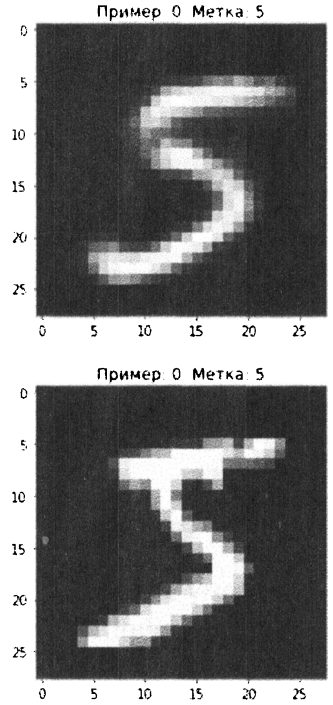


Рис. 11.12. *Первое изображение, сгенерированное DBN*

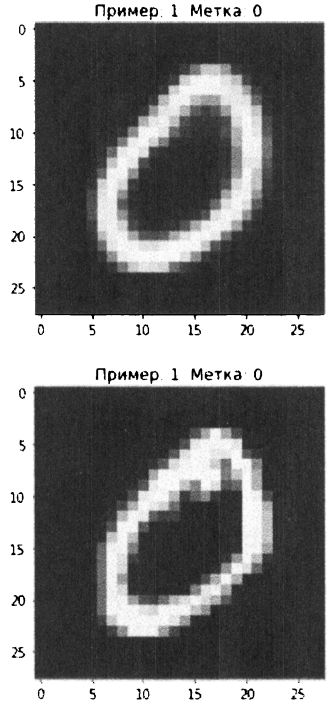
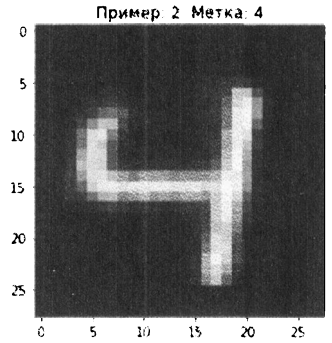


Рис. 11.13. *Второе изображение, сгенерированное DBN*

Стоит отметить, что DBN (как и RBM) относится к категории порождающих моделей, а потому изображения формируются стохастически. Поскольку процесс не детерминирован, изображения одного и того же примера могут меняться от прогона к прогону.

Чтобы сымитировать это, возьмем первое изображение MNIST и используем DBN для генерирования новых изображений 10 раз.

```
# Генерировать первый пример 10 раз
inputXReduced = X_train.loc[:0]
for i in range(0, 10):
    example = 0
    print("Прогон", i)
    finalOutput_DBN_fives, reconstructedOutput_DBN_fives = \
        dbn.dbn_output(inputXReduced)
    reconstructedX_fives = \
        pd.DataFrame(data=reconstructedOutput_DBN_fives, index=[0])
    print("Сгенерировано")
    view_digit(reconstructedX_fives, y_train.loc[:0], example)
```



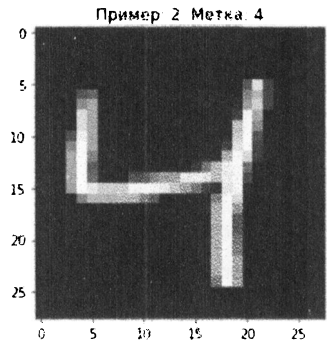
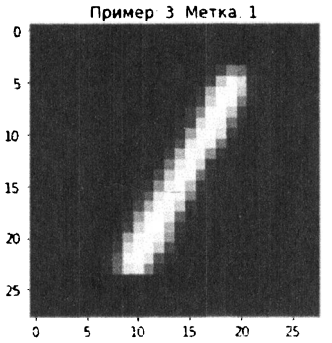


Рис. 11.14. *Третье изображение, сгенерированное DBN*



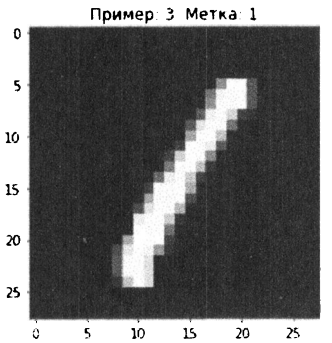


Рис. 11.15. *Четвертое изображение, сгенерированное DBN*

Как можно увидеть на рис. 11.17–11.21, все сгенерированные изображения напоминают цифру 5 и в то же время незначительно отличаются друг от друга, несмотря на то что все они были получены на основе одного и того же исходного изображения MNIST.

Создание классификатора изображений с использованием алгоритма LightGBM

Построим классификатор, используя введенный ранее алгоритм градиентного бустинга LightGBM.

Только обучение с учителем

Наш первый классификатор изображений будет опираться лишь на первые 5000 помеченных изображений MNIST. Это подмножество оригинального тренировочного набора MNIST, содержащего 50 000 помеченных изображений.

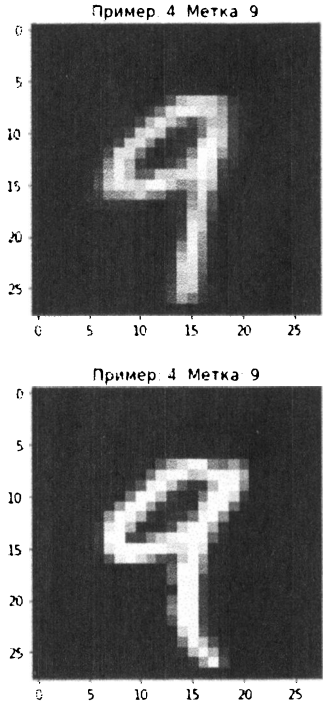


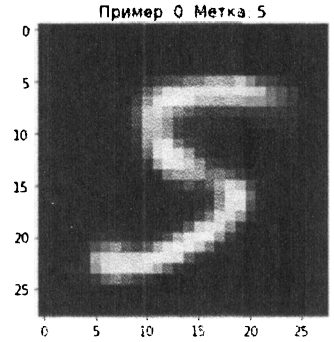
Рис. 11.16. *Пятое изображение, сгенерированное DBN*

Мы поступаем так, чтобы сымитировать ситуацию, близкую к реальной, ведь на практике обычно доступно лишь сравнительно небольшое количество размеченных примеров. Поскольку градиентный бустинг и алгоритм LightGBM нами уже обсуждались, мы опустим подробности.

Зададим параметры алгоритма.

```
predictionColumns = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

params_lightGB = {
    'task': 'train',
    'application': 'binary',
    'num_class': 10,
    'boosting': 'gbdt',
    'objective': 'multiclass',
    'metric': 'multi_logloss',
    'metric_freq': 50,
    'is_training_metric': False,
    'max_depth': 4,
```

Прогон 0
Сгенерировано



Прогон 1
Сгенерировано

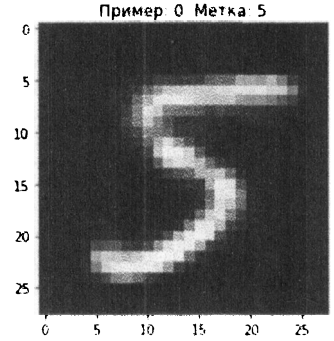
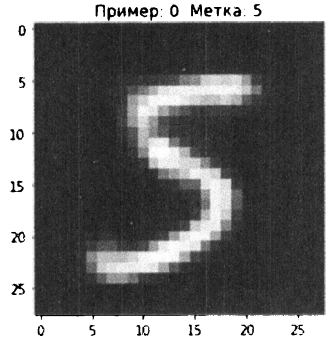


Рис. 11.17. *Первое и второе сгенерированные изображения цифры 5*

Прогон 2
Сгенерировано



Прогон 3
Сгенерировано

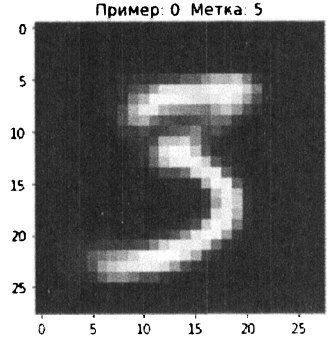
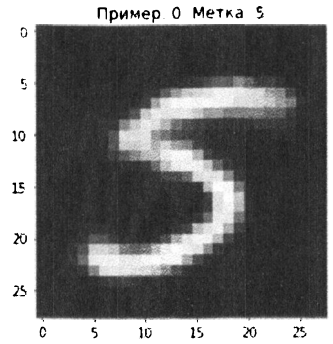


Рис. 11.18. *Третье и четвертое сгенерированные изображения цифры 5*

```
    'num_leaves': 31,
    'learning_rate': 0.1,
    'feature_fraction': 1.0,
    'bagging_fraction': 1.0,
    'bagging_freq': 0,
    'bagging_seed': 2018,
    'verbose': 0,
    'num_threads': 16
}
```

Далее используем сокращенный тренировочный набор, содержащий 5000 помеченных изображений MNIST, и валидационный набор, содержащий 10 000 таких изображений.

Прогон 4
Сгенерировано



Прогон 5
Сгенерировано

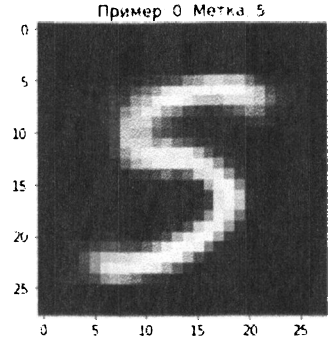
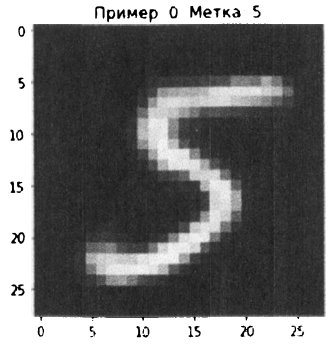


Рис. 11.19. *Пятое и шестое сгенерированные изображения цифры 5*

Прогон 6
Сгенерировано



Прогон 7
Сгенерировано

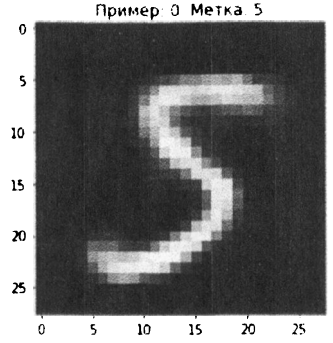


Рис. 11.20. *Седьмое и восьмое сгенерированные изображения цифры 5*

```
trainingScore = []
validationScore = []
predictionsLightGBM = pd.DataFrame(data=[], \
    index=y_validation.index, columns=predictionColumns)

lgb_train = lgb.Dataset(X_train.loc[:4999], y_train.loc[:4999])
lgb_eval = lgb.Dataset(X_validation, y_validation, \
                       reference=lgb_train)
gbm = lgb.train(params_lightGB, lgb_train, num_boost_round=2000, \
                valid_sets=lgb_eval, early_stopping_rounds=200)

loglossTraining = log_loss(y_train.loc[:4999], \
    gbm.predict(X_train.loc[:4999], num_iteration=gbm.best_
iteration))
```

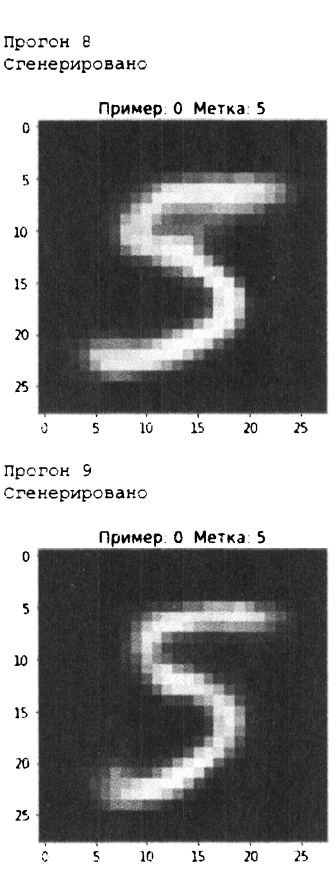


Рис. 11.21. *Девятое и десятое сгенерированные изображения цифры 5*

```
trainingScore.append(loglossTraining)

predictionsLightGBM.loc[X_validation.index, predictionColumns] = \
    gbm.predict(X_validation, num_iteration=gbm.best_iteration)
loglossValidation = log_loss(y_validation,
    predictionsLightGBM.loc[X_validation.index, predictionColumns])
validationScore.append(loglossValidation)

print('Логарифмические потери обучения:', loglossTraining)
print('Логарифмические потери валидации:', loglossValidation)

loglossLightGBM = log_loss(y_validation, predictionsLightGBM)
print('Логарифмические потери градиентного бустинга LightGBM:',
      loglossLightGBM)
```

Ниже показаны логарифмические потери в процессе тренировки и валидации для решения, в котором используется лишь обучение с учителем.

```
Логарифмические потери обучения: 0.0018646953029132292
Логарифмические потери валидации: 0.19124276982588717
```

Следующий фрагмент кода выводит общую точность данного классификатора.

```
predictionsLightGBM_firm = np.argmax(np.array(predictionsLightGBM), \
                                     axis=1)
accuracyValidation_lightGBM = accuracy_score(np.array(y_validation), \
                                             predictionsLightGBM_firm)
print("Точность обучения с учителем:", accuracyValidation_lightGBM)

Точность обучения с учителем: 0.9439
```

Совместное обучение с учителем и без учителя

На этот раз вместо обучения на наборе, состоящем из 5000 помеченных изображений MNIST, мы используем 100 000 изображений, сгенерированных глубокой сетью доверия.

```
# Подготовка объектов DataFrame для градиентного бустинга LightGBM
generatedImagesDF = pd.DataFrame(data=generatedImages, \
                                 index=range(0, 100000))
labelsDF = pd.DataFrame(data=labels, index=range(0, 100000))

X_train_lgb = pd.DataFrame(data=generatedImagesDF, \
                           index=generatedImagesDF.index)
X_validation_lgb = pd.DataFrame(data=finalOutput_DBN_validation, \
                                index=X_validation.index)

# Тренировка LightGBM
trainingScore = []
validationScore = []
predictionsDBN = pd.DataFrame(data=[], index=y_validation.index, \
                              columns=predictionColumns)

lgb_train = lgb.Dataset(X_train_lgb, labels)
lgb_eval = lgb.Dataset(X_validation_lgb, y_validation, \
                       reference=lgb_train)
gbm = lgb.train(params_lightGB, lgb_train, num_boost_round=2000, \
                valid_sets=lgb_eval, early_stopping_rounds=200)
```

```
loglossTraining = log_loss(labelsDF, gbm.predict(X_train_lgb, \
                           num_iteration=gbm.best_iteration))
trainingScore.append(loglossTraining)

predictionsDBN.loc[X_validation.index, predictionColumns] = \
    gbm.predict(X_validation_lgb, num_iteration=gbm.best_iteration)
loglossValidation = log_loss(y_validation, \
    predictionsDBN.loc[X_validation.index, predictionColumns])
validationScore.append(loglossValidation)

print('Логарифмические потери обучения:', loglossTraining)
print('Логарифмические потери валидации:', loglossValidation)

loglossDBN = log_loss(y_validation, predictionsDBN)
print('Логарифмические потери градиентного бустинга LightGBM:', \
      loglossDBN)
```

Ниже показаны логарифмические потери в процессе тренировки и валидации для решения, улучшенного за счет подключения обучения без учителя.

```
Логарифмические потери обучения: 0.004145635328203315
Логарифмические потери валидации: 0.16377638170016542
```

А вот какой получилась общая точность данного классификатора:

```
Точность решения на основе DBN: 0.9525
```

Как видите, точность повысилась почти на один процент, что является довольно существенным улучшением.

Резюме

В главе 10 мы познакомились с простейшим типом порождающих моделей: ограниченными машинами Больцмана (RBM). В этой главе мы реализовали более сложные порождающие модели, известные как глубокие сети доверия (DBN), которые состоят из нескольких RBM, образующих каскад.

Мы продемонстрировали, как работают глубокие сети доверия: используя исключительно обучение без учителя, DBN обучается внутренней структуре данных и использует полученные знания, чтобы генерировать новые синтетические данные. В зависимости от того, насколько хорошо синтетические данные согласуются с оригинальными, DBN постепенно улучшает свои порождающие способности, добиваясь как можно большего правдоподобия

генерируемых данных. Также было показано, каким образом синтетические данные, сгенерированные глубокой сетью доверия, могут дополнять существующие наборы помеченных данных, улучшая характеристики моделей обучения с учителем за счет увеличения общего размера тренировочного набора.

Разработанное нами приложение с частичным привлечением учителя, в котором мы использовали DBN (обучение без учителя) и градиентный бустинг (обучение с учителем), продемонстрировало лучшую производительность по сравнению с решением, основанным исключительно на обучении с учителем, в задаче классификации изображений MNIST.

В главе 12 вы узнаете об одной из новых технологий в области обучения без учителя: *генеративно-состязательных сетях*.

Генеративно-состязательные сети

Мы уже изучили два типа порождающих моделей: RBM (ограниченная машина Больцмана) и DBN (глубокая сеть доверия). В данной главе мы исследуем генеративно-состязательные сети — одно из наиболее перспективных направлений в области обучения без учителя.

Базовая концепция

Концепция *генеративно-состязательной сети* (generative adversarial network — GAN) была предложена Яном Гудфеллоу с коллегами по Монреальскому университету в 2014 году. В случае GAN мы имеем две нейронные сети. Одна из них, *генератор*, выдает данные на основании модели, которая была создана с использованием выборок реальных данных, поступающих на вход. Другая сеть, *дискриминатор*, пытается отличить поддельные данные, созданные с помощью генератора, от оригинальных данных.

Генератор можно уподобить фальшивомонетчику, а дискриминатор — криминалисту, пытающемуся выявить подделку. Между этими двумя сетями возникает так называемая *антагонистическая игра*, или *игра с нулевой суммой* (zero-sum game). Генератор пытается обмануть дискриминатор, заставляя его считать, будто синтетические данные взяты из исходного набора, а дискриминатор пытается распознать эти данные как поддельные.

Генеративно-состязательные сети относятся к категории алгоритмов обучения без учителя, поскольку генератор способен обучаться базовой структуре истинного распределения даже в отсутствие меток, используя ряд параметров, количество которых значительно меньше количества тренировочных данных. Это ключевой аспект обучения без учителя, о чем мы не раз говорили в предыдущих главах. Наличие такого ограничения вынуждает генератор захватывать лишь наиболее существенные аспекты истинного распределения данных. Это напоминает обучение признакам при глубоком обучении. Каждый скрытый слой нейронной сети генератора захватывает представление входных данных, начиная с простейшего, и каждый последующий слой обнаруживает все более сложные признаки, достраивая их на основе более простых предыдущих слоев.

Благодаря наличию множества слоев генератор обучается базовой структуре данных и пытается создавать синтетические данные, почти идентичные истинным. Если генератору удастся уловить суть, то синтетические данные будут казаться настоящими.

Возможности генеративно-состязательных сетей

В главе 11 мы исследовали возможность использования синтетических данных, сгенерированных моделью обучения без учителя (например, глубокой сетью доверия), для улучшения характеристик модели обучения с учителем. Подобно DBN, генеративно-состязательные сети хорошо справляются с генерированием синтетических данных.

Если требуется сгенерировать большое количество обучающих примеров для пополнения существующих тренировочных данных (например, чтобы повысить точность распознавания изображений), то можно использовать генератор для создания синтетических данных, добавить эти новые данные в существующий набор, а затем запустить модель обучения с учителем уже на расширенном наборе.

Генеративно-состязательные сети также отлично справляются с обнаружением аномалий. Если мы заинтересованы именно в этом (например, для выявления попыток мошенничества, взлома или других подозрительных действий), то можем использовать дискриминатор для оценки каждого образца поступающих данных. Примеры, которые дискриминатор распознает как "вероятно синтетические", самые аномальные и вероятнее всего представляют собой образцы мошеннического поведения.

Глубокие сверточные генеративно-состязательные сети (DCGAN)

В этой главе мы вновь обратимся к набору MNIST, с которым работали в предыдущих главах, и применим GAN для генерирования синтетических данных, чтобы дополнить существующий набор MNIST. Затем мы применим модель, основанную на обучении с учителем, для классификации изображений. Это еще один вариант обучения с частичным привлечением учителя.

Полагаю, вы уже хорошо представляете, насколько эффективно обучение с частичным привлечением учителя. Поскольку большинство доступных нам данных не размечено, обучение с учителем

оказывает серьезную помощь в разметке данных, благодаря чему применяется во всех успешных коммерческих приложениях, основанных на обучении с учителем.

В то же время обучение без учителя ценно и само по себе, так как позволяет обучаться на неразмеченных данных. Это одна из тех областей машинного обучения, которые имеют наибольший потенциал для перехода от слабого к сильному ИИ.

Реализуемая нами разновидность GAN называется *глубокая сверточная генеративно-состязательная сеть* (deep convolutional generative adversarial network — DCGAN). Такого рода сети были впервые описаны Алеком Рэдфордом, Люком Метцом и Сумитом Чинталой в конце 2015 года[1].

DCGAN — это разновидность *сверточных нейронных сетей* (convolutional neural network — CNN), которые широко применяются — причем весьма успешно — в системах компьютерного зрения и классификации изображений. Прежде чем переходить к DCGAN, необходимо сначала исследовать сверточные сети и понять, как их использовать для классификации изображений в рамках модели обучения с учителем.

Сверточные нейронные сети

Обработка изображений и видео — намного более трудоемкая задача по сравнению с числовыми и текстовыми данными. Например, изображение стандарта 4K Ultra HD имеет размерность 4096 × 2160 × 3 (26 542 080 пикселей). Для тренировки нейронной сети на изображениях с таким разрешением потребовалось бы использовать десятки миллионов нейронов, что сильно замедлило бы скорость обучения.

Вместо того чтобы обучать нейронную сеть непосредственно на исходных изображениях, мы воспользуемся тем фактом, что тесно расположенные пиксели демонстрируют сильную корреляцию, чего нельзя сказать о далеко отстоящих пикселях.

Свертка (термин, от которого сверточные сети и получили свое название) — это процесс фильтрации изображения с целью уменьшения его размера без потери связей между близлежащими пикселями[2].

[1] Оригинальная статья с описанием DCGAN доступна по адресу `https://arxiv.org/abs/1511.06434`.

[2] Дополнительная информация о сверточных слоях содержится в статье *An Introduction to Different Types of Convolutions in Deep Learning* (`http://bit.ly/2GeMQfu`).

Получив оригинальное изображение, мы применяем к нему несколько фильтров определенного размера (*размер ядра*), перемещая их с небольшим *шагом фильтра* (stride) для получения нового, редуцированного пиксельного представления. После выполнения свертки мы дополнительно уменьшаем размер представления, поочередно выбирая пиксели максимальной интенсивности в небольших областях редуцированного пиксельного слоя. Этот процесс получил название *пулинг по максимальному значению*, или *пулинг с функцией максимума* (max pooling).

Свертка и пулинг выполняются несколько раз для уменьшения сложности изображений. После этого мы уплощаем изображения и применяем обычный полносвязный слой для классификации изображений.

Итак, давайте создадим CNN и применим ее для классификации изображений, входящих в набор MNIST. Прежде всего загрузим необходимые библиотеки.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip, datetime

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl
from mpl_toolkits.axes_grid1 import Grid

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss, accuracy_score
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score, \
    mean_squared_error
from keras.utils import to_categorical
```

```
'''Алгоритмы'''
import lightgbm as lgb

'''TensorFlow и Keras'''
import tensorflow as tf
import keras
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout, Flatten, \
    Conv2D, MaxPool2D
from keras.layers import LeakyReLU, Reshape, UpSampling2D, \
    Conv2DTranspose
from keras.layers import BatchNormalization, Input, Lambda
from keras.layers import Embedding, Flatten, dot
from keras import regularizers
from keras.losses import mse, binary_crossentropy
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.optimizers import Adam, RMSprop
```

Далее загрузим наборы MNIST и сохраним данные изображений в виде 4D-тензоров, поскольку библиотека Keras требует использования именно этого формата. Мы также создадим из входных признаков векторы прямого кодирования с помощью функции `to_categorical` библиотеки Keras.

Кроме того, мы создадим на основе данных объекты `DataFrame` библиотеки Pandas, которые нам понадобятся далее. Для просмотра изображений мы повторно воспользуемся готовой функцией `view_digit`.

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'mnist_data', \
                         'mnist.pkl.gz'])
f = gzip.open(current_path+file, 'rb')
train_set, validation_set, test_set = \
    pickle.load(f, encoding='latin1')
f.close()

X_train, y_train = train_set[0], train_set[1]
X_validation, y_validation = validation_set[0], validation_set[1]
X_test, y_test = test_set[0], test_set[1]

X_train_keras = X_train.reshape(50000, 28, 28, 1)
```

```
X_validation_keras = X_validation.reshape(10000, 28, 28, 1)
X_test_keras = X_test.reshape(10000, 28, 28, 1)

y_train_keras = to_categorical(y_train)
y_validation_keras = to_categorical(y_validation)
y_test_keras = to_categorical(y_test)

# Создание объектов DataFrame из наборов данных
train_index = range(0, len(X_train))
validation_index = range(len(X_train), \
                         len(X_train) + len(X_validation))
test_index = range(len(X_train) + len(X_validation), len(X_train) + \
                   len(X_validation) + len(X_test))

X_train = pd.DataFrame(data=X_train, index=train_index)
y_train = pd.Series(data=y_train, index=train_index)

X_validation = pd.DataFrame(data=X_validation, \
                            index=validation_index)
y_validation = pd.Series(data=y_validation, index=validation_index)

X_test = pd.DataFrame(data=X_test, index=test_index)
y_test = pd.Series(data=y_test, index=test_index)

def view_digit(X, y, example):
    label = y.loc[example]
    image = X.loc[example, :].values.reshape([28, 28])
    plt.title('Пример: %d Метка: %d' % (example, label))
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.show()
```

Теперь перейдем к созданию самой CNN.

Чтобы начать построение модели, мы вызовем функцию `Sequential` из библиотеки Keras. Затем мы добавим два сверточных слоя с функцией активацией ReLU, каждый из которых содержит 32 фильтра с размером ядра 5 × 5 и шагом 1, заданным по умолчанию. После этого выполним пулинг по максимальному значению с размером окна 2 × 2 и шагом 1. Кроме того, применим регуляризацию в форме дропаута, чтобы снизить вероятность переобучения нейронной сети. В частности, мы исключим 25% входных элементов.

На следующем этапе мы добавим еще два сверточных слоя, на этот раз с 64 фильтрами и размером ядра 3 × 3, после чего выполним пулинг по максималь-

ному значению с размером окна 2 × 2 и шагом 2. Этап завершается добавлением слоя пулинга, в котором исключаются 25% входных элементов.

Наконец, уплощим изображения, добавим обычную нейронную сеть с 256 скрытыми элементами, применим 50%-ный дропаут и выполним классификацию изображений по 10 классам, используя функцию активации Softmax.

```
model = Sequential()

model.add(Conv2D(filters = 32, kernel_size = (5, 5), \
                 padding = 'Same', activation ='relu', \
                 input_shape = (28, 28, 1)))
model.add(Conv2D(filters = 32, kernel_size = (5, 5), \
                 padding = 'Same', activation ='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(filters = 64, kernel_size = (3, 3), \
                 padding = 'Same', activation ='relu'))
model.add(Conv2D(filters = 64, kernel_size = (3, 3), \
                 padding = 'Same', activation ='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation = "relu"))
model.add(Dropout(0.5))
model.add(Dense(10, activation = "softmax"))
```

Для тренировки этой сверточной сети мы будем использовать оптимизатор Adam и минимизировать кросс-энтропию. Кроме того, мы сохраним точность классификации изображений в качестве оценочной метрики.

Теперь обучим модель на протяжении 100 эпох и оценим результаты, получаемые на валидационном наборе.

```
# Обучение CNN
model.compile(optimizer='adam', \
              loss='categorical_crossentropy', \
              metrics=['accuracy'])

model.fit(X_train_keras, y_train_keras, \
          validation_data=(X_validation_keras, y_validation_keras), \
          epochs=100)
```

График изменения точности на протяжении 100 эпох приведен на рис. 12.1.

```
Окончательная точность CNN: 0.9952999949455261
```

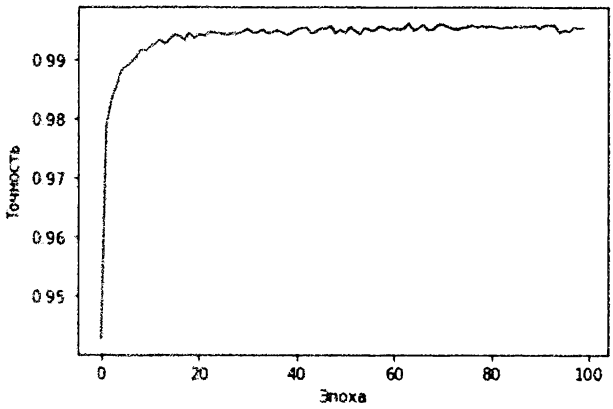


***Рис. 12.1.** Результаты обучения сверточной сети*

Как видите, окончательная точность для обученной сверточной сети составила 99,53%, что превышает точность любого из рассмотренных ранее решений, предназначенных для классификации изображений MNIST.

Возвращаемся к DCGAN

Итак, вернемся к DCGAN и построим порождающую модель для создания синтетических изображений, напоминающих оригинальные изображения MNIST.

Чтобы начать получать реалистичные синтетические изображения, мы должны обучить генератор, создающий новые изображения на основе оригинальных изображений из набора MNIST, и дискриминатор, который пытается отличить синтетические изображения от оригинальных.

Задачу можно рассмотреть под другим углом. Оригинальный набор данных MNIST представляет исходное распределение данных. Генератор обучается на этом распределении и начинает создавать новые изображения на основе приобретенных знаний, тогда как дискриминатор пытается определить, можно ли отличить сгенерированные изображения от оригинального распределения.

Что касается генератора, то мы возьмем за образец архитектуру, описанную в вышеупомянутой статье Рэдфорда, Метца и Чинталы (рис. 12.2).

Генератор получает начальный *зашумленный вектор* (z) размером 100×1, после чего переформатирует его в тензор размером $1024 \times 4 \times 4$. Эта операция

проецирования и изменения формы вектора противоположна свертке и называется *обратная свертка* (*деконволюция*).

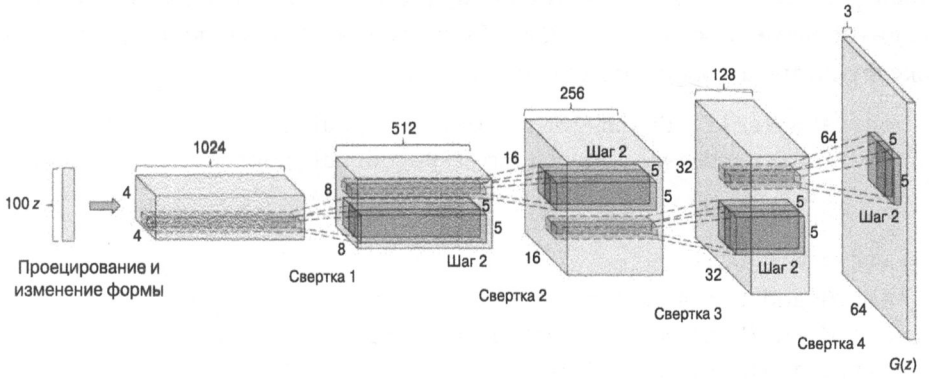


***Рис. 12.2.** Генератор DCGAN*

Выполнив начальную деконволюцию, генератор применяет четыре дополнительных сверточных слоя, транслируя полученный тензор в конечный тензор размерностью 64 × 3 × 3.

Вот как выглядят различные стадии этого процесса:

100 × 1 → 1024 × 4 × 4 → 512 × 8 × 8 → 256 × 16 × 16 → 128 × 32 × 32 → 64 × 64 × 3

Мы применим похожую архитектуру, спроектировав генеративно-состязательную сеть для классификации изображений MNIST.

Генератор DCGAN

Отправной точкой для нашей модели DCGAN послужит работа Роуэла Атьенса[3]. Мы создадим класс `DCGAN`, который будем использовать при построении генератора, дискриминатора, а также дискриминативной и состязательной моделей.

Начнем с генератора. Прежде всего зададим гиперпараметры, в том числе процент дропаута (по умолчанию 30%), глубину тензора (по умолчанию 256) и количество других измерений (по умолчанию 7 × 7). Мы также применим пакетную нормализацию со значением импульса, по умолчанию равным `0.8`. Размерность входа равна 100, а размерность окончательного выхода — 28 × 28 × 1.

Вспомните, что дропаут и пакетная нормализация — это регуляризаторы, позволяющие избежать переобучения нейронной сети.

[3] Оригинальный проект представлен на сайте GitHub (`http://bit.ly/2DLp4G1`).

Для создания генератора мы вызовем функцию `Sequential()` библиотеки Keras, после чего добавим слой плотной полносвязной нейронной сети, вызвав функцию `Dense()`. Этот слой будет иметь входную размерность 100 и выходную размерность 7 × 7 × 256. Мы применим пакетную нормализацию, а также функцию активации ReLU и дропаут.

```
def generator(self, depth=256, dim=7, dropout=0.3, momentum=0.8, \
              window=5, input_dim=100, output_depth=1):
    if self.G:
        return self.G
    self.G = Sequential()
    self.G.add(Dense(dim*dim*depth, input_dim=input_dim))
    self.G.add(BatchNormalization(momentum=momentum))
    self.G.add(Activation('relu'))
    self.G.add(Reshape((dim, dim, depth)))
    self.G.add(Dropout(dropout))
```

Затем мы дважды выполним *повышающую дискретизацию* (upsampling) и трижды — обратную свертку. При этом будем каждый раз вдвое уменьшать глубину выходного пространства (256 → 128 → 64 → 32), одновременно увеличивая число других измерений. Мы будем поддерживать окно свертки размером 5 × 5 и шаг фильтра, по умолчанию равный 1. В процессе выполнения каждой свертки мы будем применять пакетную нормализацию и функцию активации ReLU.

Этот процесс можно описать следующим образом:

100 → 7 × 7 × 256 → 14 × 14 × 128 → 28 × 28 × 64 → 28 × 28 × 32 → 28 × 28 × 1

```
    self.G.add(UpSampling2D())
    self.G.add(Conv2DTranspose(int(depth/2), window, padding='same'))
    self.G.add(BatchNormalization(momentum=momentum))
    self.G.add(Activation('relu'))

    self.G.add(UpSampling2D())
    self.G.add(Conv2DTranspose(int(depth/4), window, padding='same'))
    self.G.add(BatchNormalization(momentum=momentum))
    self.G.add(Activation('relu'))

    self.G.add(Conv2DTranspose(int(depth/8), window, padding='same'))
    self.G.add(BatchNormalization(momentum=momentum))
    self.G.add(Activation('relu'))
```

Наконец, на выходе генератора мы получим изображение с теми же размерами 28 × 28, что и оригинальное изображение MNIST.

```
    self.G.add(Conv2DTranspose(output_depth, window, padding='same'))
    self.G.add(Activation('sigmoid'))
    self.G.summary()
    return self.G
```

Дискриминатор DCGAN

Приступая к созданию дискриминатора, установим значения по умолчанию: дропаут — 30%, глубина — 64, гиперпараметр `alpha` для функции активации `LeakyReLU` — `0.3`[4].

Прежде всего загрузим изображение размерностью 28 × 28 × 1 и выполним свертку, используя 64 канала, фильтр 5 × 5 и шаг 2, а также фупкцию активации `LeakyReLU` и дропаут. Процесс повторяется трижды, и каждый раз глубина выходного пространства удваивается с одновременным уменьшением количества других измерений. На каждом шаге используется функция активации `LeakyReLU` и дропаут.

Наконец, уплощим изображения и применим сигмоиду для вывода вероятности. Эта вероятность отражает степень уверенности дискриминатора в том, что входное изображение является поддельным (значению `0.0` соответствует подделка, а значению `1.0` — истинное изображение).

Этот процесс можно описать следующим образом:

28 × 28 × 1 → 14 × 14 × 64 → 7 × 7 × 128 → 4 × 4 × 256 → 4 × 4 × 512 → 1

```
def discriminator(self, depth=64, dropout=0.3, alpha=0.3):
    if self.D:
        return self.D
    self.D = Sequential()
    input_shape = (self.img_rows, self.img_cols, self.channel)
    self.D.add(Conv2D(depth*1, 5, strides=2, \
               input_shape=input_shape, padding='same'))
    self.D.add(LeakyReLU(alpha=alpha))
    self.D.add(Dropout(dropout))
```

[4] `LeakyReLU` — это усовершенствованная функция активации, которая аналогична обычной активации ReLU, но допускает наличие небольшого градиента в тех случаях, когда элемент не активен (`https://keras.io/layers/advanced-activations/`). Данная функция все чаще применяется для решения задач машинного обучения, связанных с обработкой изображений.

```
    self.D.add(Conv2D(depth*2, 5, strides=2, padding='same'))
    self.D.add(LeakyReLU(alpha=alpha))
    self.D.add(Dropout(dropout))

    self.D.add(Conv2D(depth*4, 5, strides=2, padding='same'))
    self.D.add(LeakyReLU(alpha=alpha))
    self.D.add(Dropout(dropout))

    self.D.add(Conv2D(depth*8, 5, strides=1, padding='same'))
    self.D.add(LeakyReLU(alpha=alpha))
    self.D.add(Dropout(dropout))

    self.D.add(Flatten())
    self.D.add(Dense(1))
    self.D.add(Activation('sigmoid'))
    self.D.summary()
    return self.D
```

Дискриминативная и состязательная модели

Далее мы определим модель дискриминатора (аналог криминалиста, обнаруживающего подделки) и состязательную модель (аналог фальшивомонетчика, обучающегося на решениях криминалиста). Для обеих моделей мы используем оптимизатор RMSprop, определим функцию потерь в форме бинарной кросс-энтропии и зададим точность в качестве оценочной метрики.

Для построения состязательной модели мы используем созданные ранее сети генератора и дискриминатора. В случае дискриминативной модели используется только сеть дискриминатора.

```
def discriminator_model(self):
    if self.DM:
        return self.DM
    optimizer = RMSprop(lr=0.0002, decay=6e-8)
    self.DM = Sequential()
    self.DM.add(self.discriminator())
    self.DM.compile(loss='binary_crossentropy', \
                    optimizer=optimizer, metrics=['accuracy'])
    return self.DM

def adversarial_model(self):
    if self.AM:
        return self.AM
```

```
    optimizer = RMSprop(lr=0.0001, decay=3e-8)
    self.AM = Sequential()
    self.AM.add(self.generator())
    self.AM.add(self.discriminator())
    self.AM.compile(loss='binary_crossentropy', \
                    optimizer=optimizer, metrics=['accuracy'])
    return self.AM
```

DCGAN для набора данных MNIST

Теперь создадим генеративно-состязательную сеть для набора данных MNIST. Прежде всего инициализируем класс `MNIST_DCGAN` для изображений MNIST с размерностью 28 × 28 × 1 и подключим генератор, а также определенные ранее дискриминативную и состязательную модели.

```
class MNIST_DCGAN(object):
    def __init__(self, x_train):
        self.img_rows = 28
        self.img_cols = 28
        self.channel = 1

    self.x_train = x_train

    self.DCGAN = DCGAN()
    self.discriminator = self.DCGAN.discriminator_model()
    self.adversarial = self.DCGAN.adversarial_model()
    self.generator = self.DCGAN.generator()
```

Функция `train` по умолчанию тренирует сеть в течение 2000 эпох, используя размер пакета 256. В этой функции мы передаем пакеты изображений сети DCGAN, которую только что определили. Генератор будет выдавать изображения, а дискриминатор — распознавать изображения как подлинные или поддельные. По мере того как генератор и дискриминатор соревнуются между собой в рамках состязательной модели, синтетические изображения становятся все более похожими на оригинальные изображения из набора MNIST.

```
    def train(self, train_steps=2000, batch_size=256, \
              save_interval=0):
        noise_input = None
        if save_interval > 0:
            noise_input = np.random.uniform(-1.0, 1.0, \
                                            size=[16, 100])
```

```
    for i in range(train_steps):
        images_train = self.x_train[np.random.randint(0, \
            self.x_train.shape[0], size=batch_size), :, :, :]
        noise = np.random.uniform(-1.0, 1.0, \
                                  size=[batch_size, 100])
        images_fake = self.generator.predict(noise)
        x = np.concatenate((images_train, images_fake))
        y = np.ones([2*batch_size, 1])
        y[batch_size:, :] = 0

        d_loss = self.discriminator.train_on_batch(x, y)

        y = np.ones([batch_size, 1])
        noise = np.random.uniform(-1.0, 1.0, \
                                  size=[batch_size, 100])
        a_loss = self.adversarial.train_on_batch(noise, y)
        log_mesg = "%d: [Потери дискриминатора: %f, \
            точность: %f]" % (i, d_loss[0], d_loss[1])
        log_mesg = "%s [Потери состязательной модели: %f, \
            точность: %f]" % (log_mesg, a_loss[0], a_loss[1])

        print(log_mesg)
        if save_interval > 0:
            if (i + 1) % save_interval == 0:
                self.plot_images(save2file=True, \
                                 samples=noise_input.shape[0], \
                                 noise=noise_input, step=(i+1))
```

Также определим функцию, которая в графическом виде выводит изображения, сгенерированные сетью.

```
def plot_images(self, save2file=False, fake=True, samples=16, \
                noise=None, step=0):
    filename = 'mnist.png'
    if fake:
        if noise is None:
            noise = np.random.uniform(-1.0, 1.0, \
                                      size=[samples, 100])
        else:
            filename = "mnist_%d.png" % step
        images = self.generator.predict(noise)
    else:
        i = np.random.randint(0, self.x_train.shape[0], samples)
```

```
            images = self.x_train[i, :, :, :]

        plt.figure(figsize=(10, 10))
        for i in range(images.shape[0]):
            plt.subplot(4, 4, i+1)
            image = images[i, :, :, :]
            image = np.reshape(image, [self.img_rows, self.img_cols])
            plt.imshow(image, cmap='gray')
            plt.axis('off')
        plt.tight_layout()
        if save2file:
            plt.savefig(filename)
            plt.close('all')
        else:
            plt.show()
```

Применение генеративно-состязательной сети к набору данных MNIST

Итак, класс `MNIST_DCGAN` готов. Вызовем его и начнем процесс обучения. Мы будем тренировать модель на протяжении 10 000 эпох, используя размер пакетов 256.

```
# Инициализация и обучение сети MNIST_DCGAN
mnist_dcgan = MNIST_DCGAN(X_train_keras)
timer = ElapsedTimer()
mnist_dcgan.train(train_steps=10000, batch_size=256, \
                  save_interval=500)
```

Приведенная ниже сводка отражает потери и точность для дискриминатора и состязательной модели.

```
0: [Потери дискриминатора: 0.692640, acc: 0.527344]
   [Потери состязательной модели: 1.297974, точность: 0.000000]
1: [Потери дискриминатора: 0.651119, точность: 0.500000]
   [Потери состязательной модели: 0.920461, точность: 0.000000]
2: [Потери дискриминатора: 0.735192, точность: 0.500000]
   [Потери состязательной модели: 1.289153, точность: 0.000000]
3: [Потери дискриминатора: 0.556142, точность: 0.947266]
   [Потери состязательной модели: 1.218020, точность: 0.000000]
4: [Потери дискриминатора: 0.492492, точность: 0.994141]
   [Потери состязательной модели: 1.306247, точность: 0.000000]
```

```
5: [Потери дискриминатора: 0.491894, точность: 0.916016]
   [Потери состязательной модели: 1.722399, точность: 0.000000]
6: [Потери дискриминатора: 0.607124, точность: 0.527344]
   [Потери состязательной модели: 1.698651, точность: 0.000000]
7: [Потери дискриминатора: 0.578594, точность: 0.921875]
   [Потери состязательной модели: 1.042844, точность: 0.000000]
8: [Потери дискриминатора: 0.509973, точность: 0.587891]
   [Потери состязательной модели: 1.957741, точность: 0.000000]
9: [Потери дискриминатора: 0.538314, точность: 0.896484]
   [Потери состязательной модели: 1.133667, точность: 0.000000]
10: [Потери дискриминатора: 0.510218, точность: 0.572266]
    [Потери состязательной модели: 1.855000, точность: 0.000000]
11: [Потери дискриминатора: 0.501239, точность: 0.923828]
    [Потери состязательной модели: 1.098140, точность: 0.000000]
12: [Потери дискриминатора: 0.509211, точность: 0.519531]
    [Потери состязательной модели: 1.911793, точность: 0.000000]
13: [Потери дискриминатора: 0.482305, точность: 0.923828]
    [Потери состязательной модели: 1.187290, точность: 0.000000]
14: [Потери дискриминатора: 0.395886, точность: 0.900391]
    [Потери состязательной модели: 1.465053, точность: 0.000000]
15: [Потери дискриминатора: 0.346876, точность: 0.992188]
    [Потери состязательной модели: 1.443823, точность: 0.000000]
```

Начальная точность дискриминатора колеблется в широких пределах, но в целом остается существенно выше отметки `0.50`. Другими словами, дискриминатор поначалу очень хорошо справляется с выявлением подделок, неудачно сконструированных генератором. Но затем, по мере того как генератор совершенствует свое умение подделывать изображения, дискриминатор начинает испытывать трудности, и его точность падает до уровня, близкого к `0.50`.

```
9985: [Потери дискриминатора: 0.696480, точность: 0.521484]
      [Потери состязательной модели: 0.955954, точность: 0.125000]
9986: [Потери дискриминатора: 0.716583, точность: 0.472656]
      [Потери состязательной модели: 0.761385, точность: 0.363281]
9987: [Потери дискриминатора: 0.710941, точность: 0.533203]
      [Потери состязательной модели: 0.981265, точность: 0.074219]
9988: [Потери дискриминатора: 0.703731, точность: 0.515625]
      [Потери состязательной модели: 0.679451, точность: 0.558594]
9989: [Потери дискриминатора: 0.722460, точность: 0.492188]
      [Потери состязательной модели: 0.899768, точность: 0.125000]
9990: [Потери дискриминатора: 0.691914, точность: 0.539062]
```

```
      [Потери состязательной модели: 0.726867, точность: 0.464844]
9991: [Потери дискриминатора: 0.716197, точность: 0.500000]
      [Потери состязательной модели: 0.932500, точность: 0.144531]
9992: [Потери дискриминатора: 0.689704, точность: 0.548828]
      [Потери состязательной модели: 0.734389, точность: 0.414062]
9993: [Потери дискриминатора: 0.714405, точность: 0.517578]
      [Потери состязательной модели: 0.850408, точность: 0.218750]
9994: [Потери дискриминатора: 0.690414, точность: 0.550781]
      [Потери состязательной модели: 0.766320, точность: 0.355469]
9995: [Потери дискриминатора: 0.709792, точность: 0.511719]
      [Потери состязательной модели: 0.960070, точность: 0.105469]
9996: [Потери дискриминатора: 0.695851, точность: 0.500000]
      [Потери состязательной модели: 0.774395, точность: 0.324219]
9997: [Потери дискриминатора: 0.712254, точность: 0.521484]
      [Потери состязательной модели: 0.853828, точность: 0.183594]
9998: [Потери дискриминатора: 0.702689, точность: 0.529297]
      [Потери состязательной модели: 0.802785, точность: 0.308594]
9999: [Потери дискриминатора: 0.698032, точность: 0.517578]
      [Потери состязательной модели: 0.810278, точность: 0.304688]
```

Генерирование синтетических изображений

Теперь, когда генеративно-состязательная сеть прошла обучение, используем ее для генерирования нескольких синтетических изображений (рис. 12.3).

Нельзя сказать, что эти синтетические изображения неотличимы от истинных изображений из набора MNIST, но тем не менее они достаточно похожи на настоящие цифры. Чем дольше будет тренироваться генеративно-состязательная сеть, тем выше будет схожесть синтетических изображений с истинными. Это позволит использовать DCGAN для расширения набора данных MNIST.

Несмотря на то что наше решение оказалось довольно неплохим, существует множество дополнительных способов улучшить его. В статье "Improved Techniques for Training GANs" (`https://arxiv.org/pdf/1606.03498.pdf`) детально рассмотрены продвинутые методы улучшения производительности генеративно-состязательных сетей. Соответствующие программные решения доступны на сайте GitHub (`https://github.com/openai/improved-gan`).

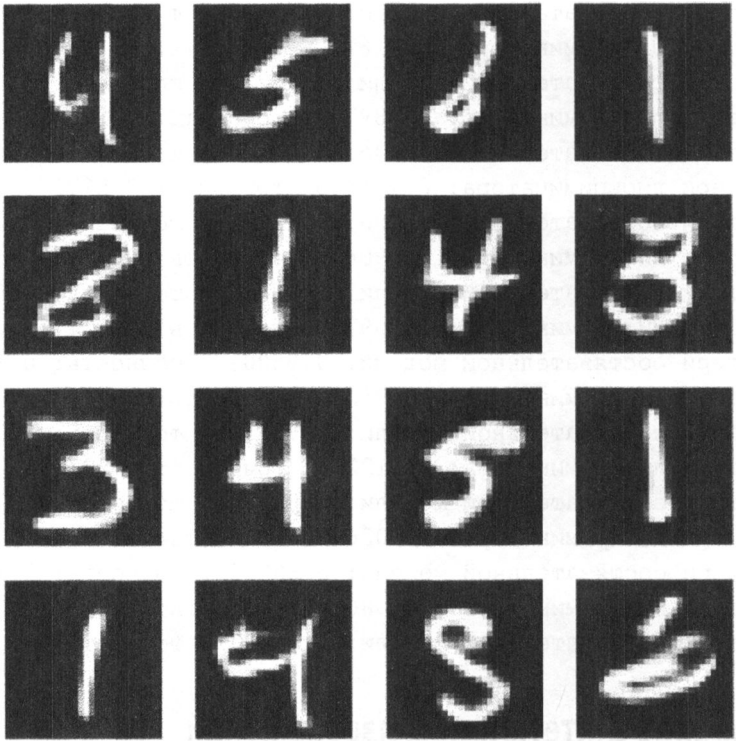


Рис. 12.3. *Синтетические изображения, сгенерированные генеративно-состязательной сетью*

Резюме

В этой главе мы исследовали глубокие сверточные генеративно-состязательные сети — разновидность GAN, хорошо приспособленная для работы с изображениями и системами компьютерного зрения.

GAN — это порождающая модель с двумя нейронными сетями, взаимодействующими в рамках игры с нулевой суммой. Одна из сетей — генератор (аналог фальшивомонетчика) — генерирует синтетические данные на основе реальных, тогда как другая сеть — дискриминатор (аналог криминалиста) — пытается отличить подделки от подлинных данных[5]. Эта антагонистическая игра, в которой генератор учится на действиях дискриминатора, позволяет получить модель, способную генерировать довольно реалистичные

[5] Рекомендуем прочитать статью в блоге OpenAI, посвященную порождающим моделям (`https://openai.com/blog/generative-models/`).

синтетические данные и к тому же улучшаться со временем (т.е. по мере увеличения числа тренировочных эпох).

Генеративно-состязательные сети — относительно новая технология. Они были впервые предложены Яном Гудфеллоу с коллегами в 2014 году[6]. В настоящее время основные сферы применения GAN — обнаружение аномалий и генерирование синтетических данных, но в ближайшем будущем у генеративно-состязательных сетей может появиться множество других применений. Исследователи только начинают открывать для себя их возможности, и если вы решите применять их в своих проектах машинного обучения, будьте готовы активно экспериментировать[7].

В главе 13 мы исследуем кластеризацию временны*х* рядов, которая тоже представляет собой разновидность обучения без учителя.

[6] Оригинальная статья доступна по адресу `https://arxiv.org/abs/1406.2661`.

[7] Советы и рекомендации по обучению генеративно-состязательных сетей доступны по адресу `https://github.com/soumith/ganhacks` и `http://bit.ly/2G2FJHq`.

ГЛАВА 13

Кластеризация временных рядов

До сих пор мы работали главным образом с *перекрестными данными*, собранными путем наблюдения за многими объектами на протяжении одного и того же периода времени. К этой категории относится набор данных о двухдневных транзакциях с банковскими картами и набор изображений рукописных цифр MNIST. Мы применяли к этим наборам обучение без учителя для изучения базовой структуры данных и группирования сходных транзакций и изображений без использования каких-либо меток.

Однако обучение без учителя также хорошо подходит и для работы с *временными рядами* (time series), когда данные накапливаются путем наблюдения за одним и тем же объектом в различные периоды времени. Нам предстоит разработать приложение, способное обучаться базовой структуре данных на основании таких наблюдений. Если нам это удастся, мы сможем выявлять шаблоны временных закономерностей и группировать схожие ряды.

Описанный подход находит применение в таких областях, как финансы, медицина, робототехника, астрономия, биология, метеорология и т.п., поскольку специалисты, работающие в этих сферах, тратят много времени на анализ данных с целью классификации текущих событий на основании их сходства с событиями, имевшими место в прошлом. Путем группирования текущих событий со схожими событиями прошлого профессионалы могут принимать обоснованные решения о том, какие действия следует предпринять.

В данной главе мы будем выполнять кластеризацию временных рядов на основе сходства шаблонов поведения. Это подход, основанный исключительно на обучении без учителя, который не требует аннотирования данных для тренировки модели, хотя аннотированные данные нужны для валидации результатов, как и в случае любых других экспериментов в области обучения без учителя.

Существует третья категория данных, в которой перекрестные данные сочетаются с временными рядами. Такие данные называются *панельными* или *продольными*.

Данные ЭКГ

Чтобы задача классификации временны*х* рядов стала более понятной, рассмотрим конкретный пример. Представьте, что вы работаете в медицинском учреждении и занимаетесь интерпретацией данных электрокардиограммы (ЭКГ). Электрокардиографы регистрируют данные об электрической активности сердца через закрепленные на теле электроды. ЭКГ записывается в течение примерно 10 секунд, и полученные показания помогают диагностировать проблемы с сердцем.

Большинство показаний ЭКГ соответствует нормальной сердечной деятельности, но специалисты должны замечать любые отклонения, чтобы своевременно выявлять нарушения в работе сердца. ЭКГ представляет собой график с множеством зубцов, поэтому задача классификации показаний сводится к задаче распознавания образов, хорошо решаемой средствами машинного обучения.

На практике показания ЭКГ часто оказываются недостаточно четкими, что затрудняет классификацию изображений и может приводить к ошибкам диагностики. Например, колебания *амплитуды* (расстояния от центральной линии до вершины зубца), *периода* (расстояния между зубцами), *фазового сдвига* (смещения по горизонтали) и *сдвига по вертикали* создают проблемы для любой компьютерной диагностической системы.

Особенности кластеризации временны*х* рядов

Любой подход к кластеризации временны*х* рядов потребует от нас обработки вышеупомянутых искажений. Как известно, кластеризация основывается на метриках расстояния, определяющих пространственную близость между различными точками данных, что обеспечивает возможность их группирования в отчетливо различимые однородные кластеры.

Кластеризация временны*х* рядов работает аналогичным образом, но нам необходимо иметь такую меру расстояния, которая была бы масштабируемой и инвариантной к фазовым сдвигам, чтобы сходные данные временны*х* рядов группировались независимо от тривиальных различий в амплитуде, периоде и сдвиге.

Алгоритм k-Shape

Один из новейших подходов к кластеризации временных рядов, отвечающих указанному выше критерию, основан на алгоритме *k-Shape*, который впервые был представлен на конференции SIGMOD (Special Interest Group on Management of Data) в 2015 году Джоном Папарризосом и Луисом Гравано[1].

Мера расстояния, используемая в алгоритме k-Shape, инвариантна к масштабированию и сдвигу и позволяет сохранять формы временных последовательностей, что облегчает их сравнение. В частности, в алгоритме k-Shape для вычисления центроидов кластеров применяется нормализованная версия взаимной корреляции, а распределение временных рядов по кластерам обновляется на каждой итерации.

Кроме того, алгоритм k-Shape не зависит от предметной области, что требует минимальной настройки гиперпараметров. Его процедура итеративного уточнения линейно масштабируется с изменением количества последовательностей. Эти характеристики делают его одним из самых эффективных алгоритмов кластеризации временных рядов, доступных на сегодняшний день.

Читателям уже должно быть очевидно, что алгоритм k-Shape напоминает алгоритм *k*-средних: в обоих случаях применяется итеративный подход к распределению данных по группам на основании расстояний между данными и центроидом ближайшей группы. Ключевое отличие заключается в том, как в алгоритме k-Shape вычисляются расстояния. Для этого задействуется мера расстояния, основанная на взаимной корреляции.

Кластеризация временных рядов по методу k-Shape применительно к набору ECGFiveDays

Приступим к созданию модели кластеризации временных рядов с использованием алгоритма k-Shape.

В этой главе мы будем работать с коллекцией временных рядов UCR Time Series 2015. В связи с тем что размер архива превышает 100 Мбайт, его нельзя распространять через GitHub. Посетите сайт UCR Time Series по адресу `http://bit.ly/2CXPcfq` и самостоятельно загрузите архив версии 2015.

Это самая большая свободно доступная коллекция размеченных временных рядов, насчитывающая 85 наборов из множества предметных областей,

[1] Соответствующая статья доступна по адресу `http://www.cs.columbia.edu/~jopa/kshape.html`.

что позволяет тестировать, насколько хорошо наше решение подходит для решения самых разных задач. Каждый временной ряд принадлежит к одному классу, поэтому у нас заодно есть метки классов для проверки результатов кластеризации.

Подготовка данных

Начнем с загрузки необходимых библиотек.

```
'''Основные библиотеки'''
import numpy as np
import pandas as pd
import os, time, re
import pickle, gzip, datetime
from os import listdir, walk
from os.path import isfile, join

'''Визуализация данных'''
import matplotlib.pyplot as plt
import seaborn as sns
color = sns.color_palette()
import matplotlib as mpl
from mpl_toolkits.axes_grid1 import Grid

%matplotlib inline

'''Подготовка данных и оценка модели'''
from sklearn import preprocessing as pp
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import log_loss, accuracy_score
from sklearn.metrics import precision_recall_curve, \
    average_precision_score
from sklearn.metrics import roc_curve, auc, roc_auc_score, \
    mean_squared_error
from keras.utils import to_categorical
from sklearn.metrics import adjusted_rand_score
import random

'''Алгоритмы'''
from kshape.core import kshape, zscore
import tslearn
from tslearn.utils import to_time_series_dataset
```

```
from tslearn.clustering import KShape, \
    TimeSeriesScalerMeanVariance
from tslearn.clustering import TimeSeriesKMeans
import hdbscan

'''TensorFlow и Keras'''
import tensorflow as tf
import keras
from keras import backend as K
from keras.models import Sequential, Model
from keras.layers import Activation, Dense, Dropout, Flatten, \
    Conv2D, MaxPool2D
from keras.layers import LeakyReLU, Reshape, UpSampling2D, \
    Conv2DTranspose
from keras.layers import BatchNormalization, Input, Lambda
from keras.layers import Embedding, Flatten, dot
from keras import regularizers
from keras.losses import mse, binary_crossentropy
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot
from keras.optimizers import Adam, RMSprop
```

Пакет *tslearn* содержит реализацию алгоритма k-Shape на Python. Он реализован на основе библиотеки Scikit-learn, но ориентирован на работу с временными рядами.

Далее загрузим тренировочные и тестовые данные из набора *ECGFiveDays*, включенного в архив UCR Time Series. Первый столбец матрицы содержит метки классов, тогда как остальные столбцы — это значения временного ряда. Мы сохраним данные в переменных `X_train`, `y_train`, `X_test` и `y_test`.

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'ucr_time_series_data', ''])
data_train = np.loadtxt(current_path + file + \
                        "ECGFiveDays/ECGFiveDays_TRAIN", \
                        delimiter=",")
X_train = to_time_series_dataset(data_train[:, 1:])
y_train = data_train[:, 0].astype(np.int)
data_test = np.loadtxt(current_path + file + \
                       "ECGFiveDays/ECGFiveDays_TEST", \
                       delimiter=",")
X_test = to_time_series_dataset(data_test[:, 1:])
y_test = data_test[:, 0].astype(np.int)
```

Следующий фрагмент кода выводит количество временных рядов, количество уникальных классов и длину каждого временного ряда.

```
# Суммарная статистика
print("Количество временных рядов:", len(data_train))
print("Количество уникальных классов:", \
      len(np.unique(data_train[:,0])))
print("Длина временного ряда:", len(data_train[0,1:]))

Количество временных рядов: 23
Количество уникальных классов: 2
Длина временного ряда: 136
```

Всего имеется 23 временных ряда и 2 уникальных класса, причем длина каждого ряда равна 136. Несколько примеров каждого класса представлено в графическом виде на рис. 13.1–13.4. Именно так выглядят данные ЭКГ.

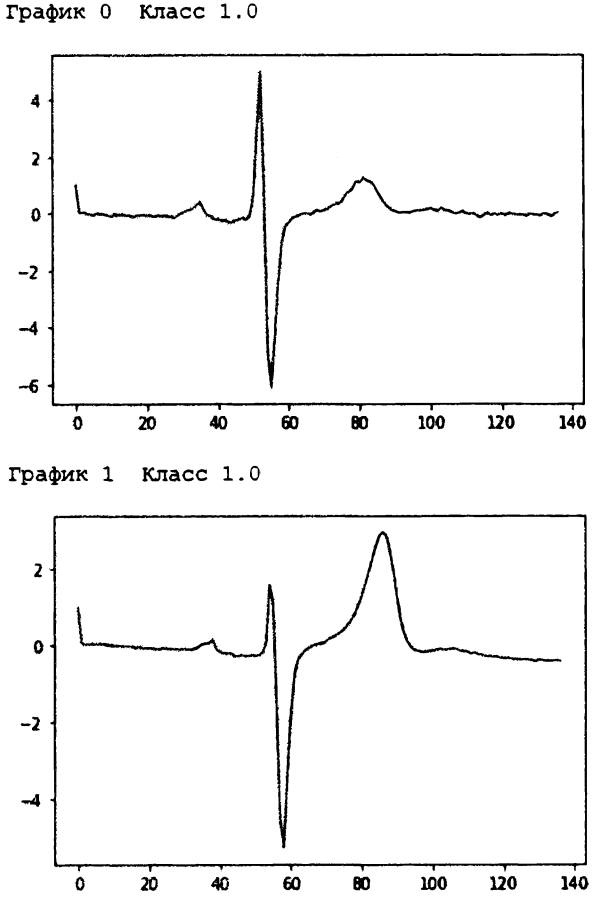


Рис. 13.1. *Набор ECGFiveDays, класс 1 — первые два примера*

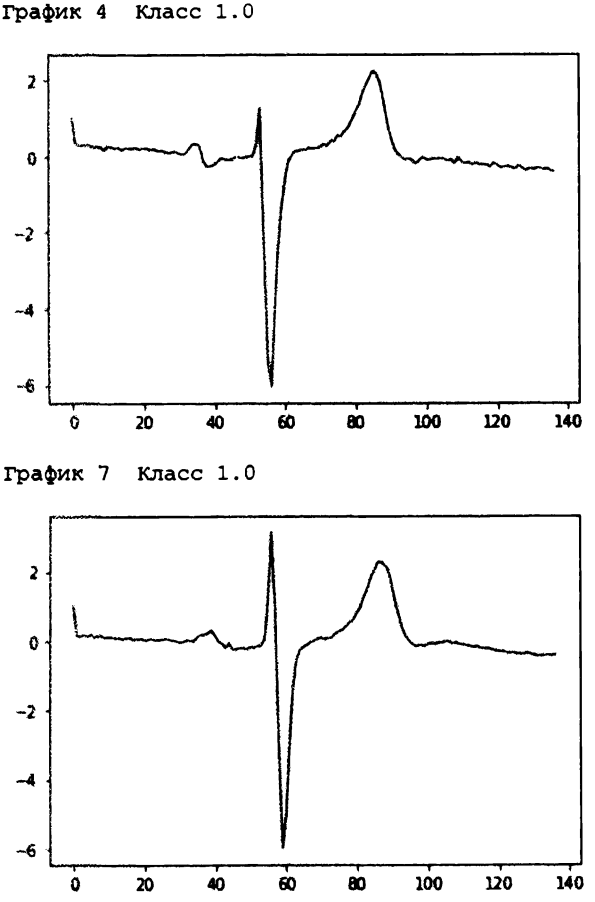


Рис. 13.2. Набор ECGFiveDays, класс 1 — вторые два примера

```
# Примеры класса 1
for i in range(0, 10):
    if data_train[i, 0] == 1.0:
        print("График", i, " Класс", data_train[i, 0])
        plt.plot(data_train[i])
        plt.show()
```

Следующий код выводит графики для класса 2.

```
# Примеры класса 2
for i in range(0, 10):
    if data_train[i, 0] == 2.0:
        print("График", i, " Класс", data_train[i, 0])
        plt.plot(data_train[i])
        plt.show()
```

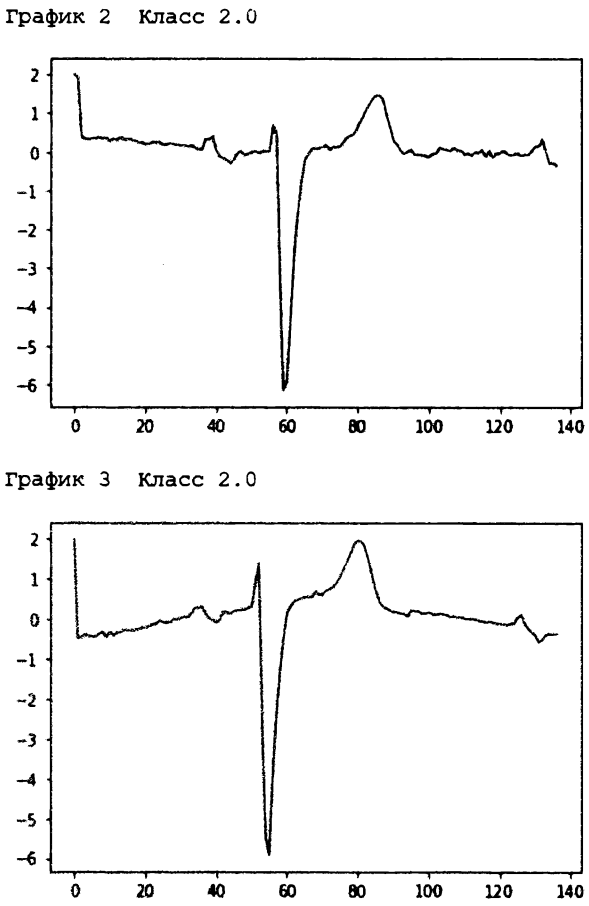


***Рис. 13.3.** Набор ECGFiveDays, класс 2 — первые два примера*

Неподготовленному пользователю трудно понять, чем отличаются примеры для классов 1 и 2, поэтому все наблюдения аннотированы специалистами в данной предметной области. Приведенные графики зашумлены искажениями. Имеются также различия в амплитуде, периоде, фазовом и вертикальном сдвиге, которые затрудняют классификацию.

Подготовим данные для обработки с помощью алгоритма k-Shape. Мы нормализуем данные таким образом, чтобы среднее значение было равно нулю, а стандартное отклонение — единице.

```
# Подготовка данных - масштабирование
X_train = TimeSeriesScalerMeanVariance(mu=0., \
    std=1.).fit_transform(X_train)
X_test = TimeSeriesScalerMeanVariance(mu=0., \
    std=1.).fit_transform(X_test)
```

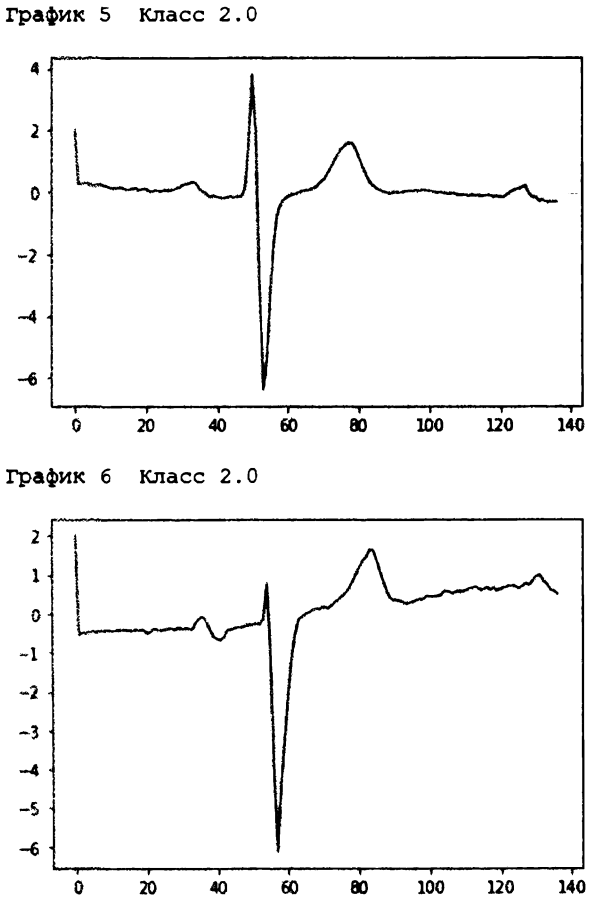


Рис. 13.4. Набор ECGFiveDays, класс 2 — вторые два примера

Тренировка и оценка модели

Далее мы запустим алгоритм k-Shape, задав количество кластеров равным 2, максимальное количество итераций — 100 и количество циклов тренировки — 100[2].

```
# Тренировка с использованием алгоритма k-Shape
ks = KShape(n_clusters=2, max_iter=100, n_init=100, verbose=0)
ks.fit(X_train)
```

Для оценки качества кластеризации временных рядов мы используем *скорректированный индекс Rand* (adjusted Rand score — ARS) — меру сходства двух

[2] Более подробную информацию относительно настройки гиперпараметров можно найти в официальной документации к классу `KShape` (`http://bit.ly/2Gfg0L9`).

вариантов кластеризации данных, скорректированную с учетом вероятности группирования элементов. С математической точки зрения это аналог точности классификации[3].

Индекс Rand представляет собой количественную оценку соответствий между предсказанной и истинной кластеризацией. Если значение скорректированного индекса Rand близко к нулю, то разбиение данных на кластеры является чисто случайным. Если же его значение близко к единице, то предсказанная кластеризация в точности совпадает с истинной.

Для вычисления скорректированного индекса Rand мы используем функцию `adjusted_rand_score` из библиотеки Scikit-learn.

Сгенерируем предсказания для кластеризации и вычислим скорректированный индекс Rand.

```
# Создание предсказаний для тренировочного набора
# и вычисление скорректированного индекса Rand
preds = ks.predict(X_train)
ars = adjusted_rand_score(data_train[:, 0], preds)
print("Скорректированный индекс Rand:", ars)
```

По результатам этого запуска скорректированный индекс Rand оказался равен `0.668`. Выполнив тренировку несколько раз, вы увидите, что индекс Rand колеблется в определенных пределах, но при этом значительно больше `0.0`.

```
Скорректированный индекс Rand: 0.668041237113402
```

Сгенерируем предсказания и вычислим скорректированный индекс Rand для тестового набора.

```
# Создание предсказаний для тестового набора
# и вычисление скорректированного индекса Rand
preds_test = ks.predict(X_test)
ars = adjusted_rand_score(data_test[:, 0], preds_test)
print("Скорректированный индекс Rand на тестовом наборе:", ars)
```

Скорректированный индекс Rand для тестового набора оказался значительно более низким, едва превышая нулевое значение. Предсказания кластеризации соответствуют почти случайным распределениям — попытка сгруппировать временны*е* ряды на основании критериев сходства не удалась.

```
Скорректированный индекс Rand на тестовом наборе:
0.0006332050676187496
```

[3] Дополнительную информацию об индексе Rand можно найти в Википедии (`https://en.wikipedia.org/wiki/Rand_index`).

Если бы у нас был намного больший тренировочный набор для обучения модели кластеризации временны*х* рядов по методу k-Shape, то можно было бы ожидать лучших результатов на тестовом наборе.

Кластеризация временны*х* рядов по методу k-Shape применительно к набору ECG5000

Вместо набора ECGFiveDays, насчитывающего всего 23 наблюдения в тренировочном наборе и 861 наблюдение — в тестовом, используем гораздо больший набор ECG5000, также включенный в архив UCR Time Series. Суммарно он содержит пять тысяч записей ЭКГ (т.е. временны*х* рядов) в тренировочном и тестовом наборах.

Подготовка данных

После загрузки данных мы самостоятельно разобьем их на тренировочный и тестовый наборы, включив в первый из них 80% оригинальных наблюдений, а во второй — оставшиеся 20%. Благодаря использованию гораздо большего тренировочного набора мы рассчитываем на то, что разрабатываемая модель кластеризации временны*х* рядов продемонстрирует гораздо лучшую производительность как на тренировочном, так и, что наиболее важно, на тестовом наборе.

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'ucr_time_series_data', ''])
data_train = np.loadtxt(current_path + file + \
                        "ECG5000/ECG5000_TRAIN", \
                        delimiter=",")

data_test = np.loadtxt(current_path + file + \
                       "ECG5000/ECG5000_TEST", \
                       delimiter=",")

data_joined = np.concatenate((data_train, data_test), axis=0)
data_train, data_test = train_test_split(data_joined, \
                                         test_size=0.20, \
                                         random_state=2019)

X_train = to_time_series_dataset(data_train[:, 1:])
y_train = data_train[:, 0].astype(np.int)
```

```
X_test = to_time_series_dataset(data_test[:, 1:])
y_test = data_test[:, 0].astype(np.int)
```

Исследуем имеющийся набор данных.

```
# Суммарная статистика
print("Количество временных рядов:", len(data_train))
print("Количество уникальных классов:", \
      len(np.unique(data_train[:, 0])))
print("Длина временного ряда:", len(data_train[0, 1:]))
```

Ниже приведена базовая сводка по тренировочному набору. Он содержит 4000 записей, которые сгруппированы в пять различных классов, а длина каждого временного ряда равна 140.

```
Количество временных рядов: 4000
Количество уникальных классов: 5
Длина временного ряда: 140
```

Проверим, какое количество записей относится к каждому классу.

```
# Вычисление количества записей на класс
print("Количество временных рядов в классе 1.0:", \
      len(data_train[data_train[:, 0]==1.0]))
print("Количество временных рядов в классе 2.0:", \
      len(data_train[data_train[:, 0]==2.0]))
print("Количество временных рядов в классе 3.0:", \
      len(data_train[data_train[:, 0]==3.0]))
print("Количество временных рядов в классе 4.0:", \
      len(data_train[data_train[:, 0]==4.0]))
print("Количество временных рядов в классе 5.0:", \
      len(data_train[data_train[:, 0]==5.0]))
```

```
Количество временных рядов в классе 1.0: 2327
Количество временных рядов в классе 2.0: 1423
Количество временных рядов в классе 3.0: 75
Количество временных рядов в классе 4.0: 156
Количество временных рядов в классе 5.0: 19
```

Большая часть записей попадает в класс 1, за которым следует класс 2. Значительно меньшее количество записей относится к классам 3, 4 и 5.

Для получения более полного представления о свойствах различных классов рассчитаем средние значения временных рядов для каждого из них.

```
# Отображение характеристик каждого класса
for j in np.unique(data_train[:, 0]):
```

```
dataPlot = data_train[data_train[:, 0]==j]
cnt = len(dataPlot)
dataPlot = dataPlot[:, 1:].mean(axis=0)
print("Класс", j, " Счетчик", cnt)
plt.plot(dataPlot)
plt.show()
```

Класс 1 (рис. 13.5) характеризуется резким спадом, за которым следуют резкий подъем и стабилизация. Это наиболее типичный случай.

```
Класс 1.0  Счетчик 2327
```

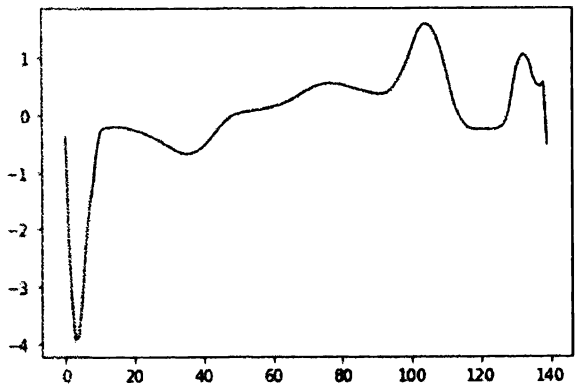


***Рис. 13.5.** Набор ECG5000, класс 1*

Класс 2 (рис. 13.6) характеризуется резким спадом, за которым следует восстановление уровня, а затем еще более глубокий спад с частичным восстановлением. Это второй по распространенности случай.

```
Класс 2.0  Счетчик 1423
```

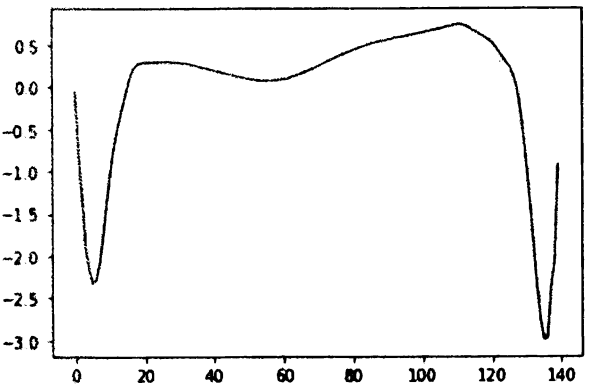


***Рис. 13.6.** Набор ECG5000, класс 2*

Класс 3 (рис. 13.7) характеризуется резким спадом, за которым следует восстановление, а затем еще более глубокий спад без восстановления. В наборе данных таких примеров немного.

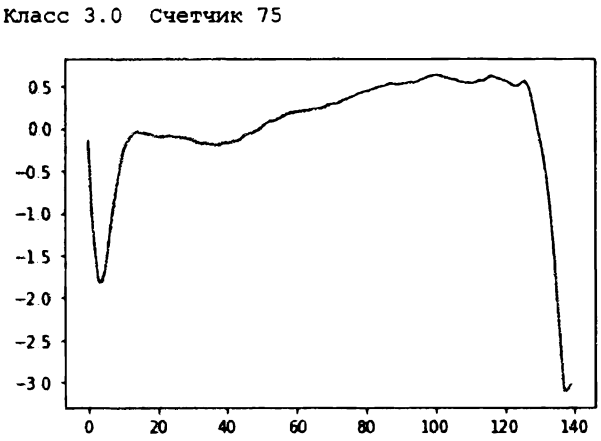


Рис. 13.7. *Набор ECG5000, класс 3*

Класс 4 (рис. 13.8) характеризуется резким спадом, за которым следует восстановление, а затем незначительный спад и стабилизация. В наборе данных таких примеров тоже не очень много.

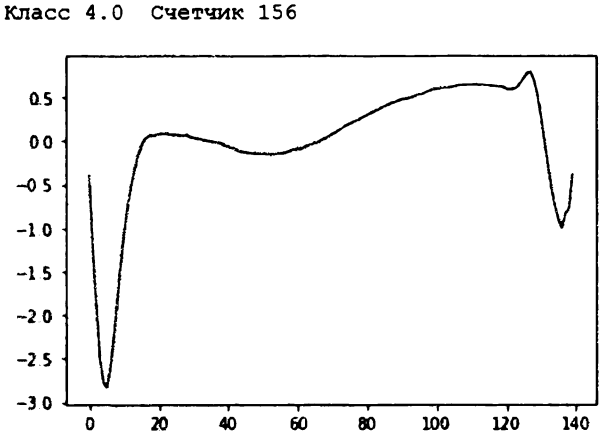


Рис. 13.8. *Набор ECG5000, класс 4*

Класс 5 (рис. 13.9) характеризуется резким спадом, за которым следует неравномерное восстановление и подъем, а затем неустойчивый спад в более мелкую впадину. В наборе данных таких примеров меньше всего.

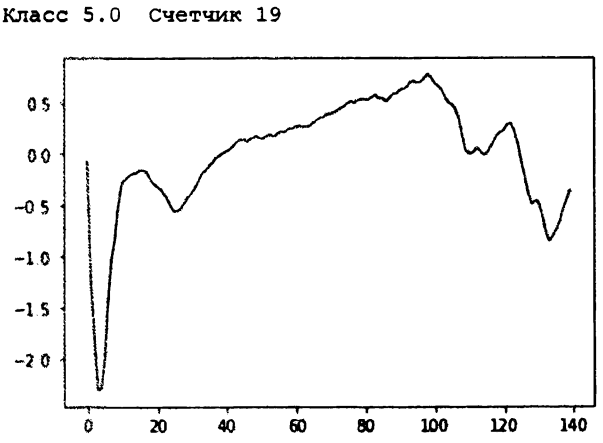


Рис. 13.9. Набор ECG5000, класс 5

Тренировка и оценка модели

Как и прежде, нормализуем данные таким образом, чтобы среднее значение было равно нулю, а стандартное отклонение — единице. Затем настроим алгоритм k-Shape, задав на этот раз количество кластеров равным 5. Все остальные параметры сохраняют прежние значения.

```
# Подготовка данных - масштабирование
X_train = TimeSeriesScalerMeanVariance(mu=0., \
    std=1.).fit_transform(X_train)
X_test = TimeSeriesScalerMeanVariance(mu=0., \
    std=1.).fit_transform(X_test)

# Тренировка с использованием алгоритма k-Shape
ks = KShape(n_clusters=5, max_iter=100, n_init=10, verbose=1, \
            random_state=2019)
ks.fit(X_train)
```

Оценим результаты для тренировочного набора.

```
# Создание предсказаний для тренировочного набора
# и вычисление скорректированного индекса Rand
preds = ks.predict(X_train)
ars = adjusted_rand_score(data_train[:, 0], preds)
print("Скорректированный индекс Rand на тренировочном наборе:", ars)
```

Скорректированный индекс Rand для тренировочного набора существенно вырос и стал равен `0.75`.

```
Скорректированный индекс Rand на тренировочном наборе:
0.7499312374127193
```

Оценим теперь результаты для тестового набора.

```
# Создание предсказаний для тестового набора
# и вычисление скорректированного индекса Rand
preds_test = ks.predict(X_test)
ars = adjusted_rand_score(data_test[:, 0], preds_test)
print("Скорректированный индекс Rand на тестовом наборе:", ars)
```

Скорректированный индекс Rand для тестового набора также значительно улучшился и достиг отметки `0.72`.

```
Скорректированный индекс Rand на тестовом наборе:
0.7172302400677499
```

Увеличив размер тренировочного набора до 4000 временных рядов (вместо 23), мы получили значительно более эффективную модель кластеризации временных рядов.

Проведем дополнительное исследование предсказанных кластеров, чтобы выяснить, насколько они однородны. Для каждого предсказанного кластера мы оценим распределение истинных меток. Если кластеры четко сформированы и однородны, то большинство записей в каждом кластере должно иметь одну и ту же истинную метку.

```
# Оценка качества кластеров
preds_test = preds_test.reshape(1000, 1)
preds_test = np.hstack((preds_test, \
                        data_test[:, 0].reshape(1000, 1)))
preds_test = pd.DataFrame(data=preds_test)
preds_test = preds_test.rename(columns={0: 'prediction', \
                                        1: 'actual'})

counter = 0
for i in np.sort(preds_test.prediction.unique()):
    print("Предсказанный кластер", i)
    print(preds_test.actual[preds_test.prediction == \
        i].value_counts())
    print()
    cnt = preds_test.actual[preds_test.prediction == \
        i].value_counts().iloc[1:].sum()
    counter = counter + cnt
print("Счетчик непервичных точек:", counter)
```

Приведенная ниже сводка отражает степень однородности кластеров.

```
Предсказанный кластер 0.0
2.0     29
4.0      2
1.0      2
3.0      2
5.0      1
Name: actual, dtype: int64

Предсказанный кластер 1.0
2.0    270
4.0     14
3.0      8
1.0      2
5.0      1
Name: actual, dtype: int64

Предсказанный кластер 2.0
1.0    553
4.0     16
2.0      9
3.0      7
Name: actual, dtype: int64

Предсказанный кластер 3.0
2.0     35
1.0      5
4.0      5
5.0      3
3.0      3
Name: actual, dtype: int64

Предсказанный кластер 4.0
1.0     30
4.0      1
3.0      1
2.0      1
Name: actual, dtype: int64

Счетчик непервичных точек: 83
```

Большинство записей в пределах каждого предсказанного кластера принадлежит только к одному классу, помеченному истинной меткой. Это свидетельствует о надежном определении кластеров и высокой степени их однородности.

Кластеризация временны*х* рядов по методу *k*-средних применительно к набору ECG5000

Для полноты картины сравним результаты, полученные с помощью алгоритма k-Shape и по методу *k*-средних. Как и прежде, мы используем библиотеку *tslearn* для тренировки и оценки модели на основе скорректированного индекса Rand.

Количество кластеров зададим равным 5, максимальное количество итераций для одного запуска — 100, количество независимых запусков — 100, значение переменной `random_state` — `2019`, метрика расстояния — `euclidean`.

```
# Тренировка по методу k-средних
km = TimeSeriesKMeans(n_clusters=5, max_iter=100, n_init=100, \
    metric="euclidean", verbose=1, random_state=2019)
km.fit(X_train)

# Создание предсказаний для тренировочного набора
# и вычисление скорректированного индекса Rand
preds = km.predict(X_train)
ars = adjusted_rand_score(data_train[:, 0], preds)
print("Скорректированный индекс Rand на тренировочном наборе:", ars)

# Создание предсказаний для тестового набора
# и вычисление скорректированного индекса Rand
preds_test = km.predict(X_test)
ars = adjusted_rand_score(data_test[:, 0], preds_test)
print("Скорректированный индекс Rand на тестовом наборе:", ars)
```

В случае использования евклидовой метрики алгоритм *TimeSeriesKMean* работает быстрее алгоритма k-Shape, но результаты получаются хуже.

```
Скорректированный индекс Rand на тренировочном наборе:
0.5063464656715959
```

Скорректированный индекс Rand для тренировочного набора оказался равен `0.506`.

```
Скорректированный индекс Rand на тестовом наборе:
0.4864981997585834
```

Скорректированный индекс Rand для тестового набора равен 0.486.

Кластеризация временны*х* рядов по методу HDBSCAN применительно к набору ECG5000

Наконец, применим иерархический алгоритм DBSCAN (*HDBSCAN*), который мы уже исследовали, и оценим его производительность.

Запустим алгоритм HDBSCAN с параметрами, заданными по умолчанию, и вычислим скорректированный индекс Rand.

```
# Тренировка модели и ее оценка на тренировочном наборе
min_cluster_size = 5
min_samples = None
alpha = 1.0
cluster_selection_method = 'eom'
prediction_data = True

hdb = hdbscan.HDBSCAN(min_cluster_size=min_cluster_size, \
    min_samples=min_samples, alpha=alpha, \
    cluster_selection_method=cluster_selection_method, \
    prediction_data=prediction_data)

preds = hdb.fit_predict(X_train.reshape(4000, 140))
ars = adjusted_rand_score(data_train[:, 0], preds)
print("Скорректированный индекс Rand на тренировочном наборе:", ars)
```

Скорректированный индекс Rand для тренировочного набора оказался довольно впечатляющим и составил 0.769.

```
Скорректированный индекс Rand на тренировочном наборе:
0.7689563655060421
```

Оценим производительность алгоритма на тестовом наборе.

```
# Создание предсказаний на тестовом наборе и их оценка
preds_test = hdbscan.prediction.approximate_predict(hdb, \
    X_test.reshape(1000, 140))
ars = adjusted_rand_score(data_test[:, 0], preds_test[0])
print("Скорректированный индекс Rand на тестовом наборе:", ars)
```

Скорректированный индекс Rand оказался в равной степени впечатляющим и составил 0.720.

```
Скорректированный индекс Rand на тестовом наборе:
0.7200816245545564
```

Сравнение трех алгоритмов кластеризации временны*х* рядов

Алгоритмы HDBSCAN и k-Shape продемонстрировали одинаково хорошую производительность на наборе ECG5000, тогда как результаты, полученные по методу *k*-средних, оказались хуже. В то же время нельзя делать какие-то серьезные выводы, оценивая производительность этих трех алгоритмов только на одном наборе данных.

Проведем расширенный эксперимент, который позволит выполнить комплексное сравнение алгоритмов. Прежде всего загрузим все файлы из папки *ucr_time_series_data*, чтобы протестировать все 85 доступных наборов данных.

```
# Загрузка наборов данных
current_path = os.getcwd()
file = os.path.sep.join(['', 'datasets', 'ucr_time_series_data', ''])

mypath = current_path + file
d = []
f = []
for (dirpath, dirnames, filenames) in walk(mypath):
    for i in dirnames:
        newpath = os.path.sep.join([mypath, i, ""])
        onlyfiles = [f for f in listdir(newpath) if \
                     isfile(join(newpath, f))]
        f.extend(onlyfiles)
    d.extend(dirnames)
    break
```

Далее мы применим каждый из трех алгоритмов кластеризации ко всем имеющимся наборам данных. Мы сохраним скорректированный индекс Rand для каждого из тренировочных и тестовых наборов и измерим время, которое потребуется алгоритмам на полный проход по всем 85 наборам.

Полный прогон с использованием алгоритма k-Shape

Сначала выполним эксперимент с алгоритмом k-Shape.

```
# Эксперимент с методом k-Shape - полный прогон

# Создание объекта DataFrame
kShapeDF = pd.DataFrame(data=[], index=[v for v in d], \
    columns=["ARS этапа тренировки", "ARS этапа тестирования"])

# Тренировка и оценка модели k-Shape
class ElapsedTimer(object):
    def __init__(self):
        self.start_time = time.time()
    def elapsed(self, sec):
        if sec < 60:
            return str(sec) + " с"
        elif sec < (60 * 60):
            return str(sec / 60) + " мин"
        else:
            return str(sec / (60 * 60)) + " ч"
    def elapsed_time(self):
        print("Прошло: %s " % self.elapsed(time.time() - \
              self.start_time))
        return (time.time() - self.start_time)

timer = ElapsedTimer()
cnt = 0
for i in d:
    cnt += 1
    print("Набор данных", cnt)
    newpath = os.path.sep.join([mypath, i, ""])
    onlyfiles = [f for f in listdir(newpath) if \
                 isfile(join(newpath, f))]
    j = onlyfiles[0]
    k = onlyfiles[1]
    data_train = np.loadtxt(newpath+j, delimiter=",")
    data_test = np.loadtxt(newpath+k, delimiter=",")

    data_joined = np.concatenate((data_train, data_test), axis=0)
    data_train, data_test = train_test_split(data_joined, \
        test_size=0.20, random_state=2019)
```

```
    X_train = to_time_series_dataset(data_train[:, 1:])
    y_train = data_train[:, 0].astype(np.int)
    X_test = to_time_series_dataset(data_test[:, 1:])
    y_test = data_test[:, 0].astype(np.int)

    X_train = TimeSeriesScalerMeanVariance(mu=0., \
        std=1.).fit_transform(X_train)
    X_test = TimeSeriesScalerMeanVariance(mu=0., \
        std=1.).fit_transform(X_test)

    classes = len(np.unique(data_train[:, 0]))
    ks = KShape(n_clusters=classes, max_iter=10, n_init=3, \
                verbose=0)
    ks.fit(X_train)

    print(i)
    preds = ks.predict(X_train)
    ars = adjusted_rand_score(data_train[:, 0], preds)
    print("Скорректированный индекс Rand на тренировочном наборе:", \
          ars)
    kShapeDF.loc[i, "ARS этапа тренировки"] = ars

    preds_test = ks.predict(X_test)
    ars = adjusted_rand_score(data_test[:, 0], preds_test)
    print("Скорректированный индекс Rand на тестовом наборе:", ars)
    kShapeDF.loc[i, "ARS этапа тестирования"] = ars

kShapeTime = timer.elapsed_time()
```

На выполнение этого алгоритма ушло больше часа. Мы сохраним значения скорректированного индекса Rand, чтобы иметь возможность сравнить производительность трех алгоритмов кластеризации.

Время выполнения алгоритма k-Shape зависит от установленных значений гиперпараметров, а также характеристик оборудования, на котором проводится эксперимент. Любые изменения данного рода могут оказывать значительное влияние на конечные результаты.

Полный прогон с использованием алгоритма *k*-средних

Используем теперь алгоритм *k*-средних.

```
# Эксперимент с методом k-средних - полный прогон

# Создание объекта DataFrame
kMeansDF = pd.DataFrame(data=[], index=[v for v in d], \
    columns=["ARS этапа тренировки", "ARS этапа тестирования"])

# Тренировка и оценка модели k-средних
timer = ElapsedTimer()
cnt = 0
for i in d:
    cnt += 1
    print("Набор данных", cnt)
    newpath = os.path.sep.join([mypath, i, ""])
    onlyfiles = [f for f in listdir(newpath) if \
                 isfile(join(newpath, f))]
    j = onlyfiles[0]
    k = onlyfiles[1]
    data_train = np.loadtxt(newpath+j, delimiter=",")
    data_test = np.loadtxt(newpath+k, delimiter=",")

    data_joined = np.concatenate((data_train, data_test),axis=0)
    data_train, data_test = train_test_split(data_joined, \
        test_size=0.20, random_state=2019)
    X_train = to_time_series_dataset(data_train[:, 1:])
    y_train = data_train[:, 0].astype(np.int)
    X_test = to_time_series_dataset(data_test[:, 1:])
    y_test = data_test[:, 0].astype(np.int)

    X_train = TimeSeriesScalerMeanVariance(mu=0., \
        std=1.).fit_transform(X_train)
    X_test = TimeSeriesScalerMeanVariance(mu=0., \
        std=1.).fit_transform(X_test)

    classes = len(np.unique(data_train[:, 0]))
    km = TimeSeriesKMeans(n_clusters=5, max_iter=10, n_init=10, \
        metric="euclidean", verbose=0, random_state=2019)
    km.fit(X_train)
```

```
    print(i)
    preds = km.predict(X_train)
    ars = adjusted_rand_score(data_train[:, 0], preds)
    print("Скорректированный индекс Rand на тренировочном наборе:", \
          ars)
    kMeansDF.loc[i, "ARS этапа тренировки"] = ars

    preds_test = km.predict(X_test)
    ars = adjusted_rand_score(data_test[:, 0], preds_test)
    print("Скорректированный индекс Rand на тестовом наборе:", ars)
    kMeansDF.loc[i, "ARS этапа тестирования"] = ars

kMeansTime = timer.elapsed_time()
```

Для обработки всех 85 наборов алгоритму *k*-средних потребовалось около пяти минут.

Полный прогон с использованием алгоритма HDBSCAN

Наконец, используем алгоритм HBDSCAN.

```
# Эксперимент с методом HDBSCAN - полный прогон

# Создание объекта DataFrame
hdbscanDF = pd.DataFrame(data=[], index=[v for v in d], \
    columns=["ARS этапа тренировки", "ARS этапа тестирования"])

# Тренировка и оценка модели HDBSCAN
timer = ElapsedTimer()
cnt = 0
for i in d:
    ccnt += 1
    print("Набор данных", cnt)
    newpath = os.path.sep.join([mypath, i, ""])
    onlyfiles = [f for f in listdir(newpath) if \
        isfile(join(newpath, f))]
    j = onlyfiles[0]
    k = onlyfiles[1]
    data_train = np.loadtxt(newpath+j, delimiter=",")
    data_test = np.loadtxt(newpath+k, delimiter=",")

    data_joined = np.concatenate((data_train, data_test), axis=0)
    data_train, data_test = train_test_split(data_joined, \
```

```
        test_size=0.20, random_state=2019)

    X_train = data_train[:, 1:]
    y_train = data_train[:, 0].astype(np.int)
    X_test = data_test[:, 1:]
    y_test = data_test[:, 0].astype(np.int)

    X_train = TimeSeriesScalerMeanVariance(mu=0., \
        std=1.).fit_transform(X_train)
    X_test = TimeSeriesScalerMeanVariance(mu=0., \
        std=1.).fit_transform(X_test)

    classes = len(np.unique(data_train[:, 0]))
    min_cluster_size = 5
    min_samples = None
    alpha = 1.0
    cluster_selection_method = 'eom'
    prediction_data = True

    hdb = hdbscan.HDBSCAN(min_cluster_size=min_cluster_size, \
        min_samples=min_samples, alpha=alpha, \
        cluster_selection_method=cluster_selection_method, \
        prediction_data=prediction_data)

    print(i)
    preds = hdb.fit_predict(X_train.reshape(X_train.shape[0], \
                                            X_train.shape[1]))
    ars = adjusted_rand_score(data_train[:, 0], preds)
    print("Скорректированный индекс Rand на тренировочном наборе:", \
          ars)
    hdbscanDF.loc[i, "ARS этапа тренировки"] = ars

    preds_test = hdbscan.prediction.approximate_predict(hdb,
        X_test.reshape(X_test.shape[0], X_test.shape[1]))
    ars = adjusted_rand_score(data_test[:, 0], preds_test[0])
    print("Скорректированный индекс Rand на тестовом наборе:", ars)
    hdbscanDF.loc[i, "ARS этапа тестирования"] = ars

hdbscanTime = timer.elapsed_time()
```

Для обработки всех 85 наборов алгоритму HBDSCAN тоже понадобилось около 10 минут.

Сравнение трех подходов к кластеризации временны*х* рядов

Теперь сравним между собой все три алгоритма кластеризации, чтобы выяснить, какой из них проявил себя лучше всего. Один из подходов заключается в вычислении средних значений скорректированного индекса Rand для тренировочных и тестовых наборов в каждом из алгоритмов.

Соответствующие оценки приведены ниже.

```
Результаты метода k-Shape

ARS этапа тренировки       0.165139
ARS этапа тестирования     0.151103

Результаты метода k-средних

ARS этапа тренировки       0.184789
ARS этапа тестирования     0.178960

Результаты метода HDBSCAN

ARS этапа тренировки       0.178754
ARS этапа тестирования     0.158238
```

Результаты оказались примерно сопоставимыми. Наибольшие значения индексов Rand получены по методу *k*-средних, за которым следуют алгоритмы k-Shape и HDBSCAN.

Для валидации некоторых из этих оценок подсчитаем, сколько раз каждый алгоритм занимал первое, второе и третье места по всем 85 наборам.

```
# Подсчет числа занятых мест
timeSeriesClusteringDF = pd.DataFrame(data=[], \
    index=kShapeDF.index, columns=["kShapeTest", \
    "kMeansTest", "hdbscanTest"])

timeSeriesClusteringDF.kShapeTest = \
    kShapeDF["ARS этапа тестирования"]
timeSeriesClusteringDF.kMeansTest = \
    kMeansDF["ARS этапа тестирования"]
timeSeriesClusteringDF.hdbscanTest = \
    hdbscanDF["ARS этапа тестирования"]

tscResults = timeSeriesClusteringDF.copy()
```

```
for i in range(0, len(tscResults)):
    maxValue = tscResults.iloc[i].max()
    tscResults.iloc[i][tscResults.iloc[i]==maxValue]=1
    minValue = tscResults.iloc[i].min()
    tscResults.iloc[i][tscResults.iloc[i]==minValue]=-1
    medianValue = tscResults.iloc[i].median()
    tscResults.iloc[i][tscResults.iloc[i]==medianValue]=0

# Вывод результатов
tscResultsDF = pd.DataFrame(data=np.zeros((3, 3)), \
    index=["firstPlace", "secondPlace", "thirdPlace"], \
    columns=["kShape", "kMeans", "hdbscan"])
tscResultsDF.loc["firstPlace", :] = \
    tscResults[tscResults==1].count().values
tscResultsDF.loc["secondPlace", :] = \
    tscResults[tscResults==0].count().values
tscResultsDF.loc["thirdPlace", :] = \
    tscResults[tscResults==-1].count().values
tscResultsDF
```

Лидером по количеству первых мест оказался алгоритм k-Shape, за которым следует алгоритм HDBSCAN. Метод *k*-средних чаще остальных занимал вторые места, демонстрируя не наилучшую, но и не наихудшую производительность для большинства наборов данных (табл. 13.1).

***Таблица 13.1.** Сводка результатов сравнения*

	kShape	kMeans	bdbscan
firstPlace	31.0	24.0	29.0
secondPlace	19.0	41.0	26.0
thirdPlace	35.0	20.0	30.0

Исходя из результатов сравнения трудно сказать, какой из алгоритмов однозначно превосходит другие в отношении производительности. Несмотря на то что алгоритм k-Shape чаще других занимал первые места, он работает значительно медленнее двух остальных алгоритмов.

В то же время алгоритм HDBSCAN и метод *k*-средних одинаково хорошо проявили себя, заняв первое место для значительного количества наборов.

Резюме

В этой главе мы исследовали временны*е* ряды и убедились в том, что обучение без учителя позволяет эффективно группировать шаблонные последовательности на основании их сходства, причем без каких-либо меток. Мы подробно рассмотрели работу с тремя алгоритмами кластеризации: k-Shape, метод *k*-средних и HDBSCAN. На сегодняшний день лучшим из них считается алгоритм k-Shape, но и остальные два алгоритма тоже дают неплохие результаты.

Самое главное то, что результаты, полученные для 85 наборов данных, продемонстрировали важность проведения экспериментов. Как это часто бывает в машинном обучении, ни одному из алгоритмов нельзя отдать явное предпочтение. Вы должны постоянно искать новые пути и экспериментировать, чтобы выяснить, какой из алгоритмов лучше всего подходит для решения конкретной задачи. Умение выбрать самый эффективный алгоритм является ключевым навыком специалиста в области интеллектуальной обработки данных.

Надеюсь, вы теперь лучше готовы к решению практических задач с использованием различных подходов на основе обучения без учителя, о которых вы узнали на протяжении книги.

ГЛАВА 14

Заключение

Искусственный интеллект переживает настоящий бум, который не наблюдался с момента появления Интернета в середине 1990-х годов. И у этого бума есть свои причины.

В предыдущие десятилетия разработки в области искусственного интеллекта и машинного обучения носили преимущественно теоретический и академический характер, а примеров успешных коммерческих решений было не так много. Но за последние годы ситуация поменялась к лучшему. Технологии приобрели более прикладную направленность, ориентированную на промышленное применение, причем флагманами процесса стали такие компании, как Google, Facebook, Amazon, Microsoft и Apple.

Смещение акцентов на разработку приложений машинного обучения для узкоспециализированных задач (слабый ИИ) вместо более амбициозных проектов (сильный ИИ) сделало эту область исследований более привлекательной для инвесторов, рассчитывающих с лихвой окупить вложенные средства в течение ближайших 7–10 лет. В свою очередь, приток инвестиций позволил достичь серьезных успехов не только в разработке слабого ИИ, но и в построении фундамента для будущих систем сильного ИИ.

Конечно же, все дело не только в инвестициях. Недавние успехи в равной степени связаны с началом эпохи больших данных, достижениями в аппаратных технологиях (в особенности имеется в виду появление мощных графических процессоров Nvidia, применяемых при обучении глубоких нейронных сетей), а также разработкой передовых алгоритмов.

Вся эта маркетинговая шумиха вполне может привести к разочарованиям в будущем, но пока что достигнутый прогресс ошеломляет, привлекая внимание широкой аудитории.

Обучение с учителем

На сегодняшний день большинство успехов в области машинного обучения связано с обучением с учителем. Успешные проекты можно разбить на следующие категории по типу обрабатываемых данных.

- **Изображения.** В эту категорию входят технологии оптического распознавания символов, классификации изображений и распознавания лиц. Например, Facebook автоматически тегирует лица на новых фотографиях, исходя из степени их сходства с ранее тегированными лицами в базе данных существующих фотографий.
- **Видео.** К этой категории относятся беспилотные автомобили, появление которых на дорогах не за горами. Инвестиции в данную область делают такие компании, как Google, Tesla и Uber.
- **Речь.** Это системы распознавания речи, представленные такими голосовыми помощниками, как Siri, Alexa, Google Assistant и Cortana.
- **Текст.** Классическим примером может служить фильтрация спама в электронной почте. Сюда же входят такие технологии, как машинный перевод (например, система Google Translate), сентимент-анализ, синтаксический анализ, распознавание языка и вопросно-ответные системы. Свидетельством успехов служит широкое распространение чат-ботов в последние годы.

Кроме того, обучение с учителем хорошо справляется с предсказанием временны*х* рядов, что находит множество применений в таких областях, как финансы, здравоохранение и рекламные технологии. Разумеется, подходы на основе обучения с учителем не ограничены работой только с одним типом данных. Например, в технологиях обработки видео распознавание изображений в сочетании с обработкой естественного языка применяется для машинного обучения систем генерирования субтитров.

Обучение без учителя

Обучение без учителя пока что не может похвастаться такими же успехами, как и обучение с учителем, но его потенциал все равно огромен. Большинство доступных в Интернете данных не размечено. Чтобы иметь возможность применять машинное обучение для решения более масштабных задач, чем те, которые уже решены с помощью обучения с учителем, нам придется работать как с размеченными, так и с неразмеченными данными.

Обучение без учителя очень хорошо справляется с обнаружением скрытых закономерностей путем обучения базовой структуре неразмеченных данных. Выявив эти закономерности, система обучения без учителя может сгруппировать найденные скрытые шаблоны на основании их схожести.

Как только такое группирование выполнено, оператор системы (т.е. человек) может выбрать несколько шаблонов из каждой группы и снабдить их понятными метками. Если группы определены достаточно четко (т.е. их элементы однородны и отчетливо отличаются от элементов других групп), то назначенные человеком метки можно применить к другим (оставшимся неразмеченными) элементам группы. Такой подход обеспечивает очень быстрое и эффективное маркирование ранее неразмеченных данных.

Другими словами, обучение без учителя позволяет успешно применять методы обучения с учителем. Подобная синергия обучения с учителем и без учителя, называемая *обучением с частичным привлечением учителя*, может породить новую волну успешных проектов машинного обучения.

Scikit-learn

Давайте кратко вспомним, о каких темах шла речь в предыдущих главах.

В главе 3 мы рассматривали, как применять методы снижения размерности путем обучения базовой структуре данных с сохранением только наиболее существенных признаков, которые транслировались в пространство меньшей размерности.

Перенос данных в пространство более низкой размерности значительно упрощает обнаружение скрытых закономерностей. В главе 4 мы продемонстрировали это, построив систему обнаружения аномалий, которая отделяла нормальные операции с банковскими картами от мошеннических.

В пространстве более низкой размерности легче группировать схожие точки данных, выполняя кластеризацию, которую мы исследовали в главе 5. В качестве примеров успешного применения кластеризации можно привести сегментирование групп, т.е. разделение объектов на основании степени их взаимного сходства. Мы применили сегментирование в главе 6 к базе данных заявок на получение займа. На этом мы завершили часть книги, посвященную использованию библиотеки Scikit-learn в обучении без учителя.

В главе 13 мы расширили кластеризацию на временные ряды и исследовали различные методы их кластеризации. Мы провели ряд экспериментов, выяснив, насколько важно располагать широким арсеналом методов машинного обучения, поскольку ни один из алгоритмов не может одинаково хорошо работать для всех наборов данных.

TensorFlow и Keras

В главах 7–12 мы переключились на библиотеки TensorFlow и Keras.

В первую очередь мы познакомились с нейронными сетями и концепцией обучения признакам. В главе 7 мы узнали, что такое автокодировщики и как они обучаются новым, более сжатым представлениям, создаваемым на основе оригинальных данных. Это еще один способ обучения базовой структуре данных с целью выявления скрытых закономерностей.

В главе 8 мы применили автокодировщики к набору данных об операциях с банковскими картами и построили приложение, способное обнаруживать мошеннические транзакции. И, что еще более важно, в главе 9 мы улучшили это решение, подключив обучение без учителя и тем самым продемонстрировав возможную синергию двух подходов.

В главе 10 мы ввели порождающие модели, начав с рассмотрения ограниченной машины Больцмана. Мы использовали данный тип нейронной сети для разработки рекомендательной системы фильмов, которая отдаленно напоминает аналогичные системы компаний Netflix и Amazon.

В главе 11 мы перешли от мелких нейронных сетей к глубоким и реализовали более продвинутую порождающую модель, объединив несколько ограниченных машин Больцмана в один каскад, называемый *глубокой сетью доверия*, что позволило нам сгенерировать синтетические изображения цифр, дополняющие существующий набор MNIST, и построить усовершенствованную систему классификации изображений. Это еще раз подчеркнуло, насколько перспективно использовать обучение без учителя для улучшения решений, основанных на обучении с учителем.

В главе 12 мы перешли к обсуждению другого класса порождающих моделей, который в настоящее время пользуется наибольшей популярностью: *генеративно-состязательные сети*. С помощью этих сетей мы сгенерировали дополнительные синтетические изображения цифр, аналогичные тем, которые содержатся в наборе данных MNIST.

Обучение с подкреплением

В этой книге мы не рассматривали обучение с подкреплением, но в последние годы оно привлекает к себе все больше внимания, особенно ввиду недавних успехов в таких областях, как настольные и видеоигры.

Несколько лет назад произошло знаменательное событие: компания Google DeepMind представила свою программу для игры в го, *AlphaGo*. Историчес-

кая победа этой программы над чемпионом мира по го Ли Седолем в марте 2016 года — результат, ожидавшийся специалистами не ранее чем через 10 лет, — позволила продемонстрировать всему миру прогресс, которого удалось добиться в области ИИ.

Чуть позже компания Google DeepMind применила комбинацию обучения с подкреплением и обучения без учителя для разработки улучшенной версии программы AlphaGo, получившей название *AlphaGo Zero*, в которой вообще не использовались данные игр, сыгранных между людьми.

Подобные успехи, являющиеся следствием объединения различных подходов к машинному обучению, лишь подтверждают лейтмотив данной книги: следующая волна успехов в машинном обучении будет связана с нахождением новых возможностей работы с неразмеченными данными для улучшения существующих решений, которые в наши дни нуждаются в интенсивном использовании размеченных данных.

Наиболее перспективные направления обучения без учителя на сегодняшний день

В завершение стоит поговорить о том, каковы ближайшие перспективы технологии обучения без учителя. На сегодняшний день она успешно применяется в нескольких областях, наиболее важные из которых — обнаружение аномалий, снижение размерности, кластеризация, эффективное маркирование неразмеченных наборов и аугментация данных.

Обучение без учителя лучше всего справляется с идентификацией ранее не известных шаблонов, особенно когда новые шаблоны резко отличаются от существующих. Это важно в тех областях, где метки прошлых шаблонов малопригодны с точки зрения захвата признаков неизвестных будущих шаблонов. Например, обнаружение аномалий применяется для выявления фальсификаций любого рода (например, мошеннических операций с кредитными картами, дебетовыми картами, банковскими переводами, онлайн-платежами, страховыми выплатами и т.п.), а также для маркирования соответствующими метками подозрительных транзакций, связанных с отмыванием денег, финансированием терроризма и торговлей людьми.

Обнаружение аномалий также задействуется в системах кибербезопасности для противодействия кибератакам. Системы, основанные на фиксированных правилах, сталкиваются с трудностями при появлении новых видов кибератак, поэтому обучение без учителя начинает играть все более важную

роль в этой сфере. Кроме того, обнаружение аномалий отлично подходит при решении проблем, связанных с качеством данных. Используя этот подход, можно более эффектно отсекать некачественные данные.

Обучение без учителя также помогает справиться с одной из главных проблем машинного обучения: проклятием размерности. Обычно специалисты в области интеллектуального анализа данных вынуждены ограничивать свой выбор некоторым подмножеством признаков, которые будут использоваться для построения моделей машинного обучения, поскольку размеры полного набора признаков затрудняют выполнение необходимых вычислений, а порой делают их практически неосуществимыми. Обучение без учителя позволяет аналитикам не только работать с оригинальным набором признаков, но и дополнять его сконструированными признаками, не опасаясь столкнуться с трудоемкими вычислениями в процессе построения модели.

Располагая подготовленным набором, состоящим из оригинальных и сконструированных признаков, аналитик может применить снижение размерности для того, чтобы исключить избыточные признаки, одновременно сохранив наиболее существенные, некоррелированные признаки для последующего анализа и построения модели. Такого рода сжатие данных можно рассматривать как полезный этап предварительной обработки данных в системах обучения с учителем (особенно при работе с изображениями и видео).

Обучение без учителя облегчает аналитикам и менеджерам выявление пользователей с нетипичным поведением, которое заметно отличается от поведения большинства других клиентов. Это становится возможным благодаря кластеризации сходных точек, что позволяет выполнять сегментирование групп. Идентифицировав отчетливые группы, специалист может проанализировать, в чем именно заключаются особенности каждой конкретной группы, отличающие ее от остальных групп. В свою очередь, это дает менеджерам возможность лучше разобраться в происходящих процессах и соответствующим образом скорректировать корпоративную стратегию.

Кластеризация значительно повышает эффективность маркирования неразмеченных данных. Ввиду того что сходные данные объединяются в группы, оператору (т.е. человеку) достаточно снабдить метками лишь небольшое количество точек в каждом кластере. После маркирования нескольких точек в каждом кластере другие точки, оставшиеся немаркированными, получают метки уже размеченных точек.

Наконец, порождающие модели позволяют генерировать синтетические данные для дополнения существующих наборов данных. Мы продемонстри-

ровали это на примере набора MNIST. Способность таких моделей генерировать новые синтетические данные различного типа, в том числе изображения и текст, открывает необычайно широкие перспективы, которые лишь недавно стали предметом серьезных исследований.

Будущее технологии обучения без учителя

Мы все еще на гребне волны интереса к искусственному интеллекту. Мы стали свидетелями больших успехов, достигнутых в этом направлении, но многое в мире ИИ пока что строится на энтузиазме и обещаниях. Потенциал технологий искусственного интеллекта огромен, но его только предстоит раскрыть.

Успехи в основном касаются узкоспециализированных задач, решаемых с помощью обучения без учителя. Но есть надежда, что по мере развития технологий мы перейдем от задач слабого ИИ, таких как классификация изображений, машинный перевод, распознавание речи и чат-боты, к более амбициозным проектам сильного ИИ: чат-боты, способные понимать человеческий язык и вести свободный диалог с человеком; роботы, которые ориентируются в пространстве и действуют, не полагаясь на размеченные данные; беспилотные автомобили, способные самообучаться вождению на уровне, превосходящем возможности человека; интеллектуальные агенты, умеющие заниматься творчеством на человеческом уровне.

Многие эксперты считают, что ключом к разработке сильного ИИ является обучение без учителя. При любом другом подходе искусственный интеллект будет скован ограничениями, зависящими от количества имеющихся размеченных данных.

В чем человек остается непревзойденным (причем с рождения), так это в способности обучаться выполнению разнообразных задач, не требуя множества примеров. К примеру, ребенок в раннем возрасте способен отличить кота от собаки, увидев их лишь считанное число раз. Современные системы искусственного интеллекта требуют множества примеров/меток. В идеале система должна уметь различать изображения, относящиеся к разным классам (например, коты и собаки), обходясь минимальным количеством меток, возможно одной или вообще без них. Реализация обучения такого типа возможна лишь за счет дальнейшего прогресса в области обучения без учителя.

Кроме того, современные системы искусственного интеллекта по большей части лишены творческих способностей. Они просто полагаются на оптимизацию распознавания шаблонов на основании меток, предоставляемых

им в процессе обучения. Чтобы создать систему, обладающую интуицией и творческими навыками, исследователи должны наделить ее способностью анализировать множество неразмеченных данных и выявлять шаблоны, оставшиеся незамеченными даже людьми.

К счастью, наблюдаются признаки того, что мы постепенно продвигаемся в направлении сильного ИИ.

Имеется в виду программа AlphaGo, разработанная компанией Google DeepMind. Первая версия программы одержала победу над профессиональным игроком в го (в октябре 2015 года), опираясь на данные предыдущих игр, сыгранных между людьми, и такие методы машинного обучения, как обучение с подкреплением, что позволило ей просчитывать игру на много ходов вперед и выявлять ходы, которые сильнее всего повышают шансы на победу.

Победа этой версии программы AlphaGo над одним из лучших профессиональных игроков в го Ли Седолем в матче из пяти партий, который был проведен в Сеуле в марте 2016 года, произвела огромное впечатление. Однако последняя версия AlphaGo продемонстрировала еще лучшую производительность.

Оригинальная программа AlphaGo полагалась на имеющиеся данные и опыт других игроков. Новейшая версия программы, *AlphaGo Zero*, обучалась игре и выигрышным стратегиям с чистого листа, играя сама с собой[1]. Другими словами, программа не опиралась на накопленные людьми знания, но это не помешало ей достичь сверхчеловеческого уровня, победив предыдущую версию AlphaGo со счетом 100:0.

Совершенно ничего не зная об игре го, программа AlphaGo Zero буквально за несколько дней приобрела опыт, для накопления которого людям понадобились тысячелетия. Но программа пошла еще дальше, продемонстрировав результативность, превосходящую возможности человека. Программе удалось обнаружить новые знания и разработать новые, ранее не известные стратегии выигрыша. Тем самым программа AlphaGo Zero проявила творческие способности.

Если технологии искусственного интеллекта продолжат развиваться в данном направлении, демонстрируя способность к обучению на небольших объемах имеющихся знаний или даже в условиях полного их отсутствия (т.е. с использованием небольшого количества размеченных данных или вообще

[1] Описание стратегии обучения программы AlphaGo Zero приведено в статье *AlphaGo Zero: Learning from Scratch*, доступной по адресу `https://deepmind.com/blog/alphago-zero-learning-scratch/`.

без них), то мы получим искусственный интеллект, способный творить, рассуждать и принимать сложные решения, т.е. интеллект, наделенный качествами, доселе присущими только людям[2].

Резюме

Мы ограничились лишь поверхностным рассмотрением технологии обучения без учителя и ее возможностей, но я надеюсь, что теперь вы будете лучше понимать, какие широкие перспективы открывает обучение без учителя и как применять его для построения систем машинного обучения.

Как минимум, вы получили базовое представление о том, что такое обучение без учителя и как создавать приложения, способные обнаруживать скрытые закономерности в данных, выявлять аномалии, выполнять кластеризацию и сегментирование групп, автоматически выделять признаки и генерировать синтетические данные на основе неразмеченных наборов данных.

У технологий искусственного интеллекта огромное будущее. Давайте приблизим его вместе!

[2] Компания OpenAI продемонстрировала заметные успехи в применении обучения без учителя для понимания естественного языка, что является важным шагом на пути к созданию сильного ИИ. О соответствующих достижениях можно прочитать в статьях *Unsupervised Sentiment Neuron* (`https://openai.com/blog/unsupervised-sentiment-neuron/`) и *Improving Language Understanding with Unsupervised Learning* (`https://openai.com/blog/language-unsupervised/`).

Предметный указатель

Н

О

П

Р

O'REILLY®

Прикладное машинное обучение без учителя
с использованием Python

По мнению многих отраслевых экспертов, обучение без учителя — передовой рубеж технологий искусственного интеллекта (ИИ) и, возможно, ключ к созданию сильного ИИ. Поскольку подавляющая часть накопленных в мире данных не размечена, к ним нельзя применять традиционное обучение с учителем. В то же время обучение без учителя позволяет успешно работать с неразмеченными наборами данных и выявлять заложенные в них закономерности, обнаружить которые человеку не под силу.

Автор книги показывает, как реализовать обучение без учителя на основе двух платформ Python: Scikit-learn и TensorFlow/Keras. Используя готовый код и практические примеры, специалисты по работе с данными смогут выявлять скрытые закономерности в информационных массивах, более глубоко анализировать деловые данные, обнаруживать аномалии, выполнять автоматическое конструирование признаков и генерировать синтетические наборы данных. Все, что потребуется от читателя, — знание программирования и предварительный опыт работы в области машинного обучения.

Основные темы книги:

- **Сравнение сильных и слабых сторон различных подходов к машинному обучению: с учителем, без учителя и с подкреплением**
- **Запуск готового проекта машинного обучения**
- **Создание системы обнаружения аномалий для выявления попыток мошенничества с банковскими картами**
- **Кластеризация пользователей путем разбиения их на отчетливо различимые однородные группы**
- **Обучение с частичным привлечением учителя**
- **Построение рекомендательной системы фильмов с использованием ограниченных машин Больцмана**
- **Генерирование синтетических изображений с помощью генеративно-состязательных сетей**

"Книга написана понятным языком и содержит практические примеры на Python, которые можно быстро и эффективно реализовать. Ее по достоинству оценят исследователи, инженеры и студенты".

Сара Надь,
главный специалист по работе с данными, Edison Software

Анкур Пател — вице-президент компании 7Park Data, входящей в портфель активов инвестиционной компании Vista Equity Partners. Вместе со своей командой разрабатывает программные продукты по обработке данных для хедж-фондов, а также систему MLaaS (машинное обучение как услуга), предназначенную для корпоративных клиентов.

ISBN: 978-5-907144-99-6

9 785907 144996 19145

Категория: компьютерные технологии/искусственный интеллект/Python
Уровень: для пользователей средней и высокой квалификации

ДИАЛЕКТИКА

http://www.williamspublishing.com

http://www.oreilly.com