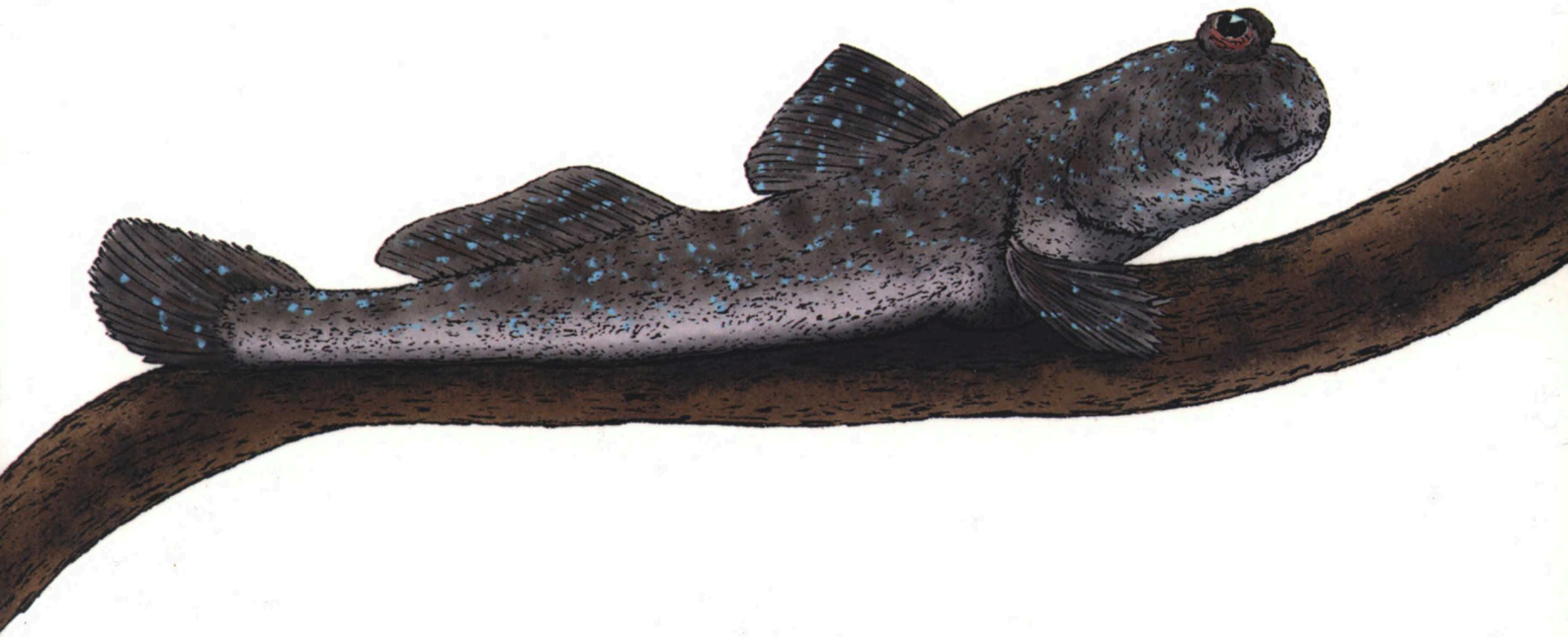
O'REILLY®

Расширенная аналитика с PySpark

Практические примеры анализа больших наборов данных с использованием Python и Spark



Акаш Тандон,
Сэнди Райза, Ури Ласерсон,
Шон Оуэн, Джош Уиллс

bhv®

Advanced Analytics with PySpark

Patterns for Learning from Data at Scale Using Python and Spark

Akash Tandon, Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY

Расширенная аналитика с PySpark

Практические примеры анализа больших наборов данных с использованием Python и Spark

Акаш Тандон,
Сэнди Райза, Ури Ласерсон,
Шон Оуэн, Джош Уиллс

Санкт-Петербург
«БХВ-Петербург»
2023

УДК 004.43
ББК 32.973.26-018.1
Р24

Р24 Расширенная аналитика с PySpark: Пер. с англ. — СПб.: БХВ-Петербург, 2023. — 224 с.: ил.

ISBN 978-5-9775-1770-6

Книга посвящена практическим методам анализа больших объемов данных с использованием языка Python и фреймворка Spark, она знакомит с моделью программирования Spark и основами системы с открытым исходным кодом PySpark. Каждая глава описывает отдельный аспект анализа данных, показаны основы обработки данных в PySpark и Python на примере очистки данных, подробно освещается машинное обучение с помощью Spark. Книга поможет читателю понять, как устроен и работает весь конвейер PySpark для комплексной аналитики больших наборов данных: от создания и оценки моделей до очистки, предварительной обработки и исследования данных с особым акцентом на производственные приложения. Отдельные главы посвящены обработке изображений и библиотеке Spark NLP.

Для специалистов по обработке данных и машинному обучению

УДК 004.43
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Павел Шалин*
Зав. редакцией	*Людмила Гауль*
Редактор	*Наталья Смирнова*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Анна Брезман*
Оформление обложки	*Зои Канторович*

Подписано в печать 02.02.23.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 18,06.
Тираж 1000 экз. Заказ № 6024.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-098-10365-1 (англ.)
ISBN 978-5-9775-1770-6 (рус.)

Оглавление

Предисловие

Многочисленные предшественники Apache Spark, от MPI (message passing interface, интерфейс передачи сообщений) до MapReduce, позволяли разрабатывать приложения, которые используют огромные вычислительные ресурсы, абстрагируясь от деталей распределенных систем. Сначала потребность в обработке больших данных обусловила создание этих платформ, а затем индустрия больших данных стала целиком зависеть от того, с какими объемами эти платформы могут справиться. Первоначальный замысел создателей Spark заключался в том, чтобы пойти немного дальше — сделать так, чтобы разработка распределенных приложений мало чем отличалась от разработки обычных программ.

Рост популярности Spark совпал с ростом популярности экосистемы данных Python (PyData). Поэтому вполне логично, что популярность PySpark — API Spark для Python — значительно выросла за последние несколько лет. Хотя в экосистеме PyData недавно появились несколько инструментов распределенного программирования, Apache Spark остается одним из самых популярных вариантов для работы с большими наборами данных в разных отраслях и областях. Благодаря недавней интеграции PySpark с другими инструментами PyData, внедрение этого фреймворка наверняка поможет вам значительно повысить производительность труда специалистов по обработке и анализу данных.

Мы считаем, что лучший способ преподавания науки о данных — это личный пример. С этой целью мы наполнили книгу примерами приложений, пытаясь продемонстрировать взаимосвязь между наиболее распространенными алгоритмами, наборами данных и приемами проектирования в аналитике больших данных. Эта книга не предназначена для чтения от корки до корки: бегло пролистайте страницы, найдите главу, которая наиболее близка к вашей задаче или просто заинтересовала вас, и начните с нее.

Почему мы написали эту книгу именно сейчас?

В 2020 г. Apache Spark претерпел глубокое обновление до версии 3.0. Одним из самых больших улучшений стало введение механизма адаптивного выполнения Spark Adaptive Execution. Он устраняет большую часть затруднений с настройкой и оп-

тимизацией. Мы не упоминаем этот механизм в книге, потому что он включен по умолчанию в Spark 3.2 и более поздних версиях, и вы автоматически используете его преимущества.

Изменения экосистемы в сочетании с последним крупным выпуском Spark — на наш взгляд, наиболее подходящее время для издания этой книги. В отличие от предыдущих изданий книги про анализ больших данных с помощью Spark и языка программирования Scala, в текущей редакции мы будем использовать Python. Мы рассмотрим лучшие приемы программирования и интегрируем их с более широкой косистемой Python для обработки и анализа данных, насколько это возможно. Все главы были обновлены с учетом использования последней версии API PySpark. Кроме того, в книгу были добавлены две новые главы, а несколько глав подверглись серьезной переработке. При этом мы не будем рассматривать потоковую и графическую библиотеки Spark. Мы надеемся, что Spark вступил в новую эру зрелости и стабильности, и обновленная книга сохранит свою актуальность как полезный ресурс по аналитике данных на долгие годы.

Как устроена эта книга

В *главе 1* мы рассмотрим Spark и PySpark в более широком контексте науки о данных и анализа больших данных. Далее в каждой главе будет описан отдельный аспект анализа данных с использованием PySpark. *Глава 2* познакомит с основами обработки данных в PySpark и Python на примере очистки данных. Следующие несколько глав посвящены машинному обучению с помощью Spark и демонстрируют применение некоторых наиболее распространенных алгоритмов в канонических приложениях. Остальные главы представляют собой скорее набор разрозненных тем и рассказывают о применении Spark для решения несколько более экзотических задач — например, для запросов к Википедии через скрытые семантические связи в тексте, анализа данных геномики и выявления похожих изображений.

Эта книга не рассказывает о достоинствах и недостатках PySpark. Книга знакомит с моделью программирования Spark и основами PySpark — API Python для Spark. Тем не менее она не претендует на то, чтобы служить справочником по Spark или быть исчерпывающим путеводителем по всем закоулкам Spark. Она также не претендует на роль справочника по машинному обучению, статистике или линейной алгебре, хотя во многих главах содержится небольшой вводный материал перед их использованием.

Эта книга поможет читателю понять, как устроен и работает весь конвейер PySpark для комплексной аналитики больших наборов данных, а это не только создание и оценка моделей, но также очистка, предварительная обработка и исследование данных с особым акцентом на производственные приложения. Мы верим, что лучший способ научить этому — личный пример и опыт.

Вот примеры некоторых задач, решение которых будут рассмотрено в этой книге:

Прогнозирование лесного покрова

Мы прогнозируем тип лесного покрова, используя соответствующие характеристики, такие, как местоположение и тип почвы, используя деревья решений *(см. главу 4)*.

Поиск похожих записей в Википедии

Мы определяем отношения между статьями и запрашиваем текстовый корпус Википедии, используя методы NLP (natural language processing, обработка естественного языка) *(см. главу 6)*.

Оценка загруженности нью-йоркских такси

Мы рассчитываем среднее время ожидания такси в зависимости от местоположения, выполняя временной и геопространственный анализ *(см. главу 7)*.

Снижение риска для инвестиционного портфеля

Мы оцениваем финансовый риск для инвестиционного портфеля, используя моделирование Монте-Карло *(см. главу 8)*.

Когда это возможно, мы пытаемся не просто предоставить решение, но и продемонстрировать полный рабочий процесс обработки данных со всеми его итерациями, тупиками и перезапусками. Эта книга предназначена для знакомства с Python, Spark, машинным обучением и анализом данных. Однако она преследует и более широкие цели — мы надеемся, что прежде всего эта книга научит вас правильному подходу к прикладным задачам анализа данных. В каждой главе, состоящей примерно из 20 страниц, мы постарались максимально подробно продемонстрировать создание одного из приложений для обработки данных.

Условные обозначения, используемые в этой книге

В этой книге используются следующие шрифтовые выделения:

Курсив

Обозначает новые термины.

Полужирный шрифт

Адреса электронной почты.

`Моноширинный шрифт`

Используется для листингов программ, а также внутри абзацев для выделения элементов программы, таких, как имена переменных или функций, базы данных, типы данных, переменные среды, операторы и ключевые слова.

`Полужирный моноширинный шрифт`

Обозначает команды или другой текст, который пользователь должен ввести как показано в книге.

Курсивный моноширинный шрифт

Обозначает текст, который следует заменить значениями, заданными пользователем, или значениями, определяемыми контекстом.

Этот элемент означает подсказку или предложение.

Этот элемент означает общее примечание.

Этот элемент указывает на предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) доступен для загрузки по адресу **https://github.com/sryza/aas**.

Если у вас есть технический вопрос или проблема с использованием примеров кода, отправьте электронное письмо по адресу **bookquestions@oreilly.com**.

Эта книга написана для того, чтобы помочь вам успешно осуществить задуманное. В большинстве случаев вы можете свободно использовать код из этой книги в своих программах и документации. Вам не нужно обращаться к нам за разрешением, если вы не копируете значительную часть кода. Например, для написания программы, использующей несколько фрагментов кода из этой книги, разрешения не требуется. Однако для продажи или распространения компакт-дисков с примерами из этой книги вам потребуется разрешение правообладателя. Если отвечая на вопрос, вам понадобится процитировать информацию из этой книги и код примера, разрешения не требуется. Но использование значительного количества примеров кода из этой книги в документации по вашему продукту уже потребует разрешения.

Мы приветствуем указание авторства при цитировании, но не настаиваем на нем. Атрибуция обычно включает название, автора, издателя и ISBN. Например: «*Advanced Analytics with PySpark* by Akash Tandon, Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills (O'Reilly). Copyright 2022 Akash Tandon, 978-1-098-10365-1».

Если вы считаете, что использование вами примеров кода выходит за рамки добросовестного использования или упомянутого здесь разрешения, не стесняйтесь обращаться к нам по адресу **permissions@oreilly.com**.

Благодарности

Совершенно очевидно, что вы бы не читали эту книгу, если бы не существовали платформы Apache Spark и MLlib. Мы все должны поблагодарить команду, которая

создала и открыла исходный код, а также сотни сотрудников, которые внесли свой вклад в его создание.

Мы хотим поблагодарить всех, кто потратил время на изучение предыдущих изданий книги и предоставил свою экспертную оценку: Майкла Бернико, Адама Брейндела, Яна Басса, Парвиза Дейхима, Джереми Фримена, Криса Фрегли, Дебашиша Гоша, Джульет Хоугланд, Джонатана Киблера, Нишу Муктевар, Фрэнка Нотафта, Ника Пентрита, Костаса Сакеллиас, Тома Уайта, Марсело Ванзина и еще раз Джульет Хоугланд. Спасибо всем! Мы в долгу перед вами. Ваша помощь сделала эту книгу намного лучше.

Сэнди выражает благодарность Джордану Пинкусу и Ричарду Вангу за помощь в изложении теоретических основ главы о рисках.

Спасибо Джеффу Блайэлу и О'Рейли за советы и огромную поддержку на всем пути этой книги от написания до публикации.

ГЛАВА 1

Анализ больших данных

Когда мы говорим, что живем в эпоху больших данных, то обычно имеем в виду, что у нас есть инструменты для сбора, хранения и обработки информации в невиданных ранее масштабах. Следующие задачи просто не могли быть решены 10 или 15 лет назад:

- создание модели для обнаружения мошенничества с кредитными картами, использующей тысячи признаков и опыт миллиардов транзакций;
- предложения миллионов товаров для миллионов покупателей;
- оценка финансовых рисков с помощью моделирования портфелей, включающих миллионы инвестиционных инструментов;
- использование данных о геномах тысяч людей для выявления генетической предрасположенности к заболеваниям;
- оценка эффективности использования сельскохозяйственных земель и урожайности сельскохозяйственных культур для совершенствования стратегии земледелия путем регулярной обработки миллионов спутниковых изображений.

За этими возможностями стоит экосистема программного обеспечения с открытым исходным кодом, которая может использовать кластеры серверов для обработки огромных объемов данных. После выпуска Apache Hadoop в 2006 г. большую популярность приобрели распределенные вычисления, а экосистема больших данных и соответствующие инструменты начали стремительно развиваться. За последние пять лет на рынок вышли различные популярные библиотеки *машинного обучения* (machine learning, ML) и *глубокого обучения* (deep learning, DL) с открытым исходным кодом. Эти инструменты нацелены на использование огромных объемов данных, которые мы научились собирать и хранить.

Но, точно так же, как наличие долота и мраморной глыбы еще не гарантирует создание прекрасной статуи, также существует и ощутимый разрыв между наличием инструментов и больших данных и возможностью получить от них какую-то пользу. Часто «получение пользы» означает построение схемы табличных данных и использование SQL-запросов для ответа на такие вопросы, как «Сколько из миллионов пользователей, дошедших до третьей страницы нашего процесса регистрации, старше 25 лет?». Проектирование хранилищ данных и организация информации (базы данных, озера данных и т. д.), представляют собой отдельную и весьма обширную область знаний, в этой книге мы в эту тему особо углубляться не станем.

Иногда «получение пользы» от данных требует дополнительной работы. SQL по-прежнему может служить основой, но для устранения искажений данных или вы-

полнения сложного анализа нам нужна более гибкая парадигма программирования с более богатыми функциональными возможностями в таких областях, как машинное обучение и статистика. Именно здесь на помощь приходит наука о данных, и именно об этом мы поговорим в данной книге.

В этой главе мы начнем с рассмотрения концепции больших данных и обсудим некоторые проблемы, возникающие при работе с большими наборами данных. Затем мы рассмотрим Apache Spark, платформу с открытым исходным кодом для распределенных вычислений, и ее ключевые компоненты. Мы сосредоточимся на фреймворке PySpark — API Spark для Python — и на том, как он вписывается в более широкую экосистему. Далее следует обсуждение изменений, внесенных Spark 3.0, первым значительным обновлением фреймворка за четыре года. Мы закончим главу краткой заметкой о том, как PySpark решает проблемы науки о данных и почему это отличное дополнение к вашему набору навыков.

В предыдущих изданиях этой книги в качестве примеров кода мы использовали API Spark на языке Scala. В этом издании мы решили использовать PySpark из-за феноменальной популярности Python в сообществе специалистов по данным и повышенного внимания основной команды Spark к поддержке языка. К концу этой главы вы наверняка оцените это решение по достоинству.

Работа с большими данными

Многие из наших любимых инструментов для работы с ограниченными объемами данных терпят неудачу при работе с большими данными. Такие библиотеки, как pandas, не приспособлены для работы с данными, которые не помещаются в оперативной памяти нашего компьютера. Как должен выглядеть аналогичный процесс, способный использовать кластеры компьютеров для достижения тех же результатов на больших наборах данных? Проблемы распределенных вычислений требуют от нас переосмысления многих основных предпосылок, на которые мы полагаемся в системах с одним вычислительным узлом. Например, поскольку данные должны быть разделены по многим узлам в кластере, многие алгоритмы будут страдать от того, что скорость передачи данных по сети на несколько порядков ниже, чем при доступе к оперативной памяти. По мере увеличения количества машин, работающих над задачей, увеличивается вероятность сбоя. Эти соображения заставляют нас искать парадигмы программирования, учитывающие характеристики вычислительной системы. Современные инструменты должны препятствовать неправильному выбору подходов и упрощать написание кода с высокой степенью параллельного выполнения.

Насколько велики большие данные?

Без точки отсчета термин «большие данные» неоднозначен. Более того, устаревшее двухуровневое определение малых и больших данных может сбивать с толку. Когда дело доходит до размера данных, более полезным является трехуровневое определение (табл. 1.1).

Таблица 1.1. Многоуровневое определение размеров данных

Тип набора данных	Помещается в ОЗУ?	Помещается на локальный диск?
Небольшой набор данных	Да	Да
Средний набор данных	Нет	Да
Большой набор данных	Нет	Нет

Согласно таблице, если набор данных может поместиться в памяти или на диске в одной машине, его нельзя назвать большими данными. Это определение не идеально, но оно действует как хорошее эмпирическое правило в контексте средней машины.

Цель этой книги — научить вас эффективно работать с большими данными. Если ваш набор данных небольшой и может поместиться в памяти, избегайте использования распределенных систем. Для анализа наборов данных среднего размера иногда бывает достаточно обычной базы данных или параллельных вычислений. В иных случаях вам, возможно, придется настроить кластер и использовать инструменты для работы с большими данными. Надеемся, что опыт, который вы приобретете в следующих главах, поможет вам сделать правильный выбор.

Одномашинные инструменты, получившие в последнее время распространение в сообществе разработчиков программного обеспечения, — не единственные инструменты, применяемые для анализа данных. Некоторые научные области, такие, как геномика, по определению имеющие дело с большими наборами данных, десятилетиями используют фреймворки для параллельных вычислений. Большинство исследователей, занятых обработкой больших данных в подобных областях, знакомы со средой кластерных вычислений, называемой HPC (high-performance computing, высокопроизводительные вычисления). В то время как ограничения прямого использования языков Python и R обусловлены их неспособностью к масштабированию, проблемы организации высокопроизводительных вычислений заключаются в относительно низком уровне абстракции и сложности использования. Например, чтобы параллельно обрабатывать большой файл, содержащий результаты секвенирования ДНК, мы должны вручную разделить его на более мелкие файлы и отправить задания для каждого из этих файлов в планировщик кластера. Если некоторые из заданий завершатся сбоем, пользователь должен обнаружить сбой и снова вручную отправить задачу на выполнение. Если же для анализа требуются операции «отношение всех ко всем», такие, как сортировка полного набора данных, ученым приходится пропускать большие данные через один узел или прибегать к распределенным средам более низкого уровня, таким, как MPI, которые трудно программировать без обширных знаний о программировании на C и распределенных сетевых системах.

Инструменты, написанные для сред высокопроизводительных вычислений, часто не могут отделить модели данных в памяти от моделей хранения более низкого уровня. Например, многие инструменты умеют только считывать данные из файловой системы POSIX в одном потоке, что затрудняет естественное распараллеливание инструментов или использование других механизмов хранения, таких, как базы данных. Современные платформы распределенных вычислений предоставляют абстракции, которые позволяют пользователям обращаться с кластером компьютеров как с одним компьютером — автоматически разделять файлы и распределять хранилище по множеству компьютеров, делить работу на более мелкие задачи и выполнять их распределенным образом, а также восстанавливаться после сбоев. Они могут автоматизировать рутинную часть работы с большими наборами данных и делают это намного дешевле, чем HPC.

Суть *распределенных систем* легче понять, если представить группу независимых компьютеров, которые выглядят для конечного пользователя как один компьютер. Они позволяют выполнять *горизонтальное масштабирование*. Это означает добавление большего количества компьютеров, а не обновление характеристик одной системы (*вертикальное масштабирование*). Второй способ относительно дорог и часто недостаточен для больших рабочих нагрузок. Распределенные системы отлично подходят для обеспечения масштабирования и надежности вычислений, но при этом усложняют проектирование, создание и отладку производственных решений. Прежде чем выбрать такой инструмент, следует оценить для себя критерии компромисса.

Знакомство с Apache Spark и PySpark

Мы начнем со знакомства с Apache Spark — платформой с открытым исходным кодом, которая сочетает в себе механизм для распространения программ по кластерам машин с элегантной моделью для написания программ поверх кластеров. Движок Spark был создан группой AMPLab в Калифорнийском университете в Беркли, а затем передан Apache Software Foundation. На момент выпуска это было, пожалуй, первое программное обеспечение с открытым исходным кодом, которое сделало распределенное программирование действительно доступным для специалистов по данным.

Компоненты Spark

Помимо основного вычислительного движка (Spark Core), Spark состоит из четырех основных компонентов. Код Spark, написанный пользователем с использованием любого из его API, выполняется в так называемых *воркерах* JVM (Java Virtual Machine, виртуальная машина Java) по всему кластеру (подробнее об этом в *главе 2*). Эти компоненты доступны в виде отдельных библиотек, как показано на рис. 1.1. Кратко поясним назначение компонентов:

Spark SQL и DataFrames + наборы данных

Модуль для работы со структурированными данными.

MLlib

Масштабируемая библиотека машинного обучения.

Структурированная потоковая передача

Упрощает создание масштабируемых отказоустойчивых потоковых приложений.

GraphX (устаревший компонент)

GraphX — это библиотека Apache Spark для графов и граф-параллельных вычислений. Однако для графовой аналитики рекомендуется использовать GraphFrames вместо GraphX, который не так активно разрабатывается и недостаточно адаптирован к Python. GraphFrame — это базовая библиотека обработки графов с открытым исходным кодом, похожая на GraphX Apache Spark, но использующая API на основе DataFrame.

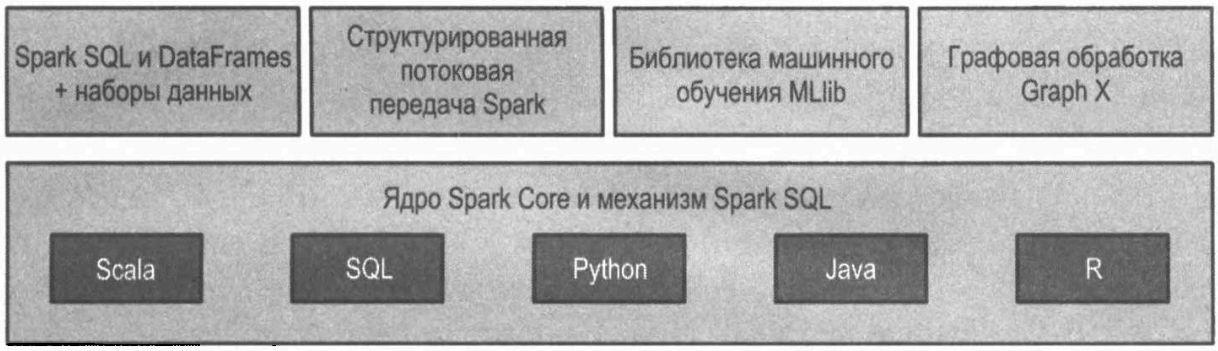


Рис. 1.1. Компоненты Apache Spark

Сравнение с MapReduce

Один из лучших способов понять возможности Spark — оценить его преимущества по сравнению с предшественником MapReduce от Apache Hadoop. Фреймворк MapReduce произвел революцию в вычислениях с огромными наборами данных, предложив простую и устойчивую модель для написания программ, которые могли выполняться параллельно на сотнях и тысячах машин. Он разбивал работу на небольшие задачи и мог изящно справляться с ошибками их выполнения, не ставя под угрозу задание, к которому они принадлежали.

Spark поддерживает линейную масштабируемость и отказоустойчивость MapReduce, при этом расширяя их возможности тремя важными способами:

- Во-первых, вместо того чтобы полагаться на жесткий формат «сопоставить, затем уменьшить», его движок может выполнять более обобщенный ориентированный ациклический граф операторов. Это означает, что в ситуациях, когда MapReduce вынужден записать промежуточные результаты в распределенную файловую систему, Spark может передать их непосредственно на следующий шаг конвейера.

- Во-вторых, вычислительные возможности Spark дополнены богатым набором преобразований, которые позволяют пользователям более естественно выражать вычислительные алгоритмы в коде. Предусмотрены готовые функции для различных задач, включая числовые вычисления, обработку даты и времени и работу со строками.

В-третьих, Spark расширяет возможности своих предшественников за счет обработки в памяти. Если для следующих шагов понадобится прежний набор данных, их не придется пересчитывать или перезагружать с диска. Spark хорошо подходит для итерационных алгоритмов, а также для специальных запросов.

PySpark

PySpark — это API Spark для Python. Проще говоря, PySpark — это основанная на Python оболочка над базовой структурой Spark, которая написана в основном на Scala. PySpark предоставляет интуитивно понятную среду программирования для специалистов по обработке данных и сочетает гибкость Python с возможностями распределенной обработки Spark.

PySpark позволяет нам работать с последовательностью моделей программирования. Например, распространенным подходом является выполнение крупномасштабных рабочих нагрузок *извлечения, преобразования и загрузки* (extract, transform and load, ETL) с помощью Spark, а затем сбор результатов на локальный компьютер с последующей обработкой с помощью библиотеки pandas. Мы рассмотрим такие модели программирования при написании кода PySpark в следующих главах. Вот пример кода из официальной документации, чтобы вы получили представление о том, что будет дальше:

```
from pyspark.ml.classification import LogisticRegression

# Загрузка обучающих данных
training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Подгонка модели
lrModel = lr.fit(training)

# Печать коэффициентов и точки пересечения логистической регрессии
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))

# Мы также можем использовать для бинарной классификации
# полиномиальное семейство
mlr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8,
                         family="multinomial")
```

```
# Подгонка модели
mlrModel = mlr.fit(training)

# Печать коэффициентов и точки пересечения логистической регрессии
# для полиномиального семейства
print("Multinomial coefficients: " + str(mlrModel.coefficientMatrix))
print("Multinomial intercepts: " + str(mlrModel.interceptVector))
```

Spark, PySpark или SparkSQL?

Различие между Spark, PySpark и SparkSQL может сбить с толку новичков. Мы используем эти три названия по отдельности, поэтому давайте кратко перечислим различия, чтобы избежать путаницы в дальнейшем.

Spark

Фреймворк распределенной обработки, написанный в основном на языке программирования Scala. Фреймворк предлагает различные языковые API поверх основного фреймворка на основе Scala.

PySpark

API Spark для Python. Фактически, это оболочка на языке Python поверх ядра Spark.

SparkSQL

Модуль Spark предназначен для обработки структурированных данных. Он является частью базовой платформы Spark и доступен через все его языковые API, включая PySpark.

Экосистема

В экосистеме больших данных Spark больше всего похож на швейцарский армейский нож, славящийся, как известно, своим огромным функционалом. К тому же Spark расширяем и хорошо интегрируется с остальной частью экосистемы. В отличие от упомянутых ранее систем Apache Hadoop и HPC, Spark разделяет хранилище и вычислительные ресурсы. Это означает, что мы можем использовать Spark для чтения данных, хранящихся в самых разных источниках — Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, реляционных СУБД и т. д., и обрабатывать их все в памяти. API Spark DataFrameReader и DataFrameWriter также можно расширить для чтения данных из других источников, таких, как Apache Kafka, Amazon Kinesis, Azure Storage и Amazon S3. Он также поддерживает несколько режимов развертывания, от локальных сред до кластеров Apache YARN и Kubernetes.

Вокруг Spark существует широкое сообщество, участники которого разработали много сторонних пакетов. Созданный сообществом список таких пакетов можно найти по адресу **https://github.com/awesome-spark/awesome-spark.** Основные по-

ставщики облачных услуг (AWS EMR, Azure Databricks, GCP Dataproc) также предоставляют приложения сторонних разработчиков для запуска управляемых рабочих нагрузок Spark. Кроме того, существуют специальные конференции и региональные группы, которые могут пригодиться для изучения интересных приложений и обмена передовым опытом.

Spark 3.0

В 2020 году Apache Spark выпустил Spark 3.0 — первую значительно обновленную версию с 2016 г., когда был выпущен Spark 2.0. В предыдущем издании этой книги, выпущенной в 2017 г., говорилось об изменениях, которые принес с собой выпуск Spark 2.0. В версии Spark 3.0 не так много серьезных изменений API, как в последнем крупном обновлении. В этом выпуске основное внимание уделяется улучшению производительности и удобства использования без существенной обратной несовместимости.

В модуле Spark SQL производительность значительно улучшена за счет адаптивного выполнения запросов и динамического сокращения секций. Проще говоря, новые технологии позволяют Spark на ходу адаптировать план физического выполнения и соответственно пропускать данные, которые не требуются в результатах запроса. Эти обновления избавляют пользователей от значительных усилий, которые пользователям ранее приходилось прилагать для ручной настройки и оптимизации. Spark 3.0 работает почти в два раза быстрее, чем Spark 2.4, на TPC-DS, эталоне аналитической обработки, являющемся отраслевым стандартом. Поскольку большинство приложений Spark основаны на механизме SQL, преимущества обновления коснулись всех библиотек более высокого уровня, включая MLlib и структурированную потоковую передачу, а также API более высокого уровня, включая SQL и DataFrames. Соответствие стандарту ANSI SQL делает API SQL более удобным в использовании.

Python стал лидером науки о данных с точки зрения распространенности в экосистеме. Не удивительно, что Python сейчас является наиболее широко используемым языком в Spark. Пакет PySpark имеет более пяти миллионов скачиваний в месяц. Spark 3.0 улучшает его функциональность и удобство использования. *Пользовательские функции* (user-defined function, UDF) pandas были переработаны для поддержки подсказок типов Python и итераторов в качестве аргументов. Были включены новые типы UDF pandas, а обработка ошибок теперь стала ближе к стандартному механизму Python. Версии Python ниже 3.6 считаются устаревшими и не поддерживаются после Spark 3.2.

За последние четыре года экосистема науки о данных также менялась быстрыми темпами. Все больше внимания уделяется внедрению в производство моделей машинного обучения. Глубокое обучение дало замечательные результаты, и в настоящее время команда Spark экспериментирует, чтобы ввести в планировщик проекта поддержку аппаратных ускорителей, таких, как графические процессоры.

PySpark решает проблемы науки о данных

Чтобы система, предназначенная для анализа огромных объемов данных, была успешной, она должна учитывать или, по крайней мере, не усугублять некоторые фундаментальные проблемы, с которыми сталкиваются специалисты по обработке и анализу данных.

- Во-первых, основную часть работы, необходимой для проведения успешного анализа, составляет *предварительная обработка* данных. Данные беспорядочны, и, прежде чем сделать с ними что-либо полезное, обычно необходимо выполнить очистку, обработку, слияние, смешивание и многие другие операции.
- Во-вторых, неотъемлемой частью науки о данных является *итерация*. Моделирование и анализ обычно требуют нескольких проходов по одним и тем же данным. Популярные процедуры оптимизации, такие, как стохастический градиентный спуск, подразумевают многократное сканирование входных данных для достижения сходимости. Итерация также постоянно встречается в рабочем процессе специалиста по данным. Выбор правильных функций и алгоритмов, запуск правильных тестов значимости и поиск правильных гиперпараметров — все это требует повторных экспериментов.
- В-третьих, работа над задачей не заканчивается, когда построена хорошо работающая модель. Цель науки о данных состоит в том, чтобы сделать данные полезными для тех, кто не занимается данными. Кульминацией разработки рекомендательных механизмов и систем обнаружения мошенничества в реальном времени являются приложения для работы с данными. В таких системах модели становятся частью производственного процесса и могут нуждаться в периодической перестройке — иногда в режиме реального времени.

PySpark хорошо справляется с вышеупомянутыми проблемами науки о данных, признавая, что самым узким местом в создании приложений для работы с данными является не процессор, диск или сеть, а производительность аналитика. Если поместить все этапы разработки, от предварительной обработки данных до оценки модели, в единую среду программирования, это может существенно ускорить разработку приложений. Упаковывая выразительную модель программирования с набором аналитических библиотек в среду REPL (read-eval-print loop, цикл чтение-оценка-печать), PySpark избегает лишней возни с IDE. Чем раньше аналитики смогут экспериментировать со своими данными, тем выше вероятность того, что они сделают с ними что-то полезное.

REPL, представляет собой компьютерную среду, в которой вводимые пользователем данные считываются и обрабатываются, а затем результаты возвращаются пользователю.

Основные API-интерфейсы PySpark образуют фундамент для преобразования данных в статистике, машинном обучении или матричной алгебре. Изучая набор данных и получая представление о нем, специалисты по обработке и анализу данных могут хранить данные в памяти во время выполнения запросов, а также легко кэшировать преобразованные версии данных, не тратя время на обращение к диску.

Появление платформы, которая упрощает построение моделей и при этом хорошо подходит для производственных систем, стало огромным достижением для экосистемы науки о данных.

Что дальше?

Spark заполняет промежуток между системами, предназначенными для *исследовательской аналитики*, и системами, предназначенными для *оперативной аналитики*. Часто говорят, что хороший специалист по данным — это тот, кто лучше разбирается в инженерии, чем большинство статистиков, и лучше понимает в статистике, чем большинство инженеров. По крайней мере, Spark лучше работает как прикладная система, чем большинство исследовательских систем, и лучше подходит для исследовательских целей, чем прикладные системы. Мы надеемся, что эта глава была вам полезна, и теперь вы с нетерпением ждете возможности приступить к работе с PySpark. Этим мы и займемся, начиная со следующей главы!

Введение в анализ данных с помощью PySpark

Python — это самый популярный язык программирования в прикладной науке о данных. Возможность заниматься статистическими вычислениями и веб-программированием с использованием одного и того же языка способствовала росту его популярности в начале 2010-х годов и впоследствии привела к появлению процветающей экосистемы инструментов и развитого сообщества аналитиков данных, часто называемого экосистемой PyData. Во многом именно это стало причиной популярности PySpark. Современные специалисты по обработке и анализу данных обычно хорошо знают Python, поэтому возможность использовать распределенные вычисления Spark в виде фреймворка на знакомом языке значительно увеличивает продуктивность их работы. По той же причине мы решили писать наши примеры именно в PySpark.

Трудно выразить словами, насколько революционным является переход к выполнению всех операций по обработке и анализу данных в одной среде, независимо от того, где хранятся и обрабатываются сами данные. Чтобы до конца понять глубину перемен, вы должны через них пройти, и мы надеемся, что наши примеры отражают то волшебное чувство, которое мы испытали, когда впервые начали использовать PySpark. Например, PySpark обеспечивает взаимодействие с pandas, одним из самых популярных инструментов PyData. Мы подробнее рассмотрим эту возможность далее в этой главе.

На примере упражнения по очистке данных мы в данной главе рассмотрим мощный API-интерфейс PySpark под названием DataFrame. В PySpark DataFrame или *фрейм данных* — это абстракция для наборов данных, имеющих обычную структуру, в которой каждая запись представляет собой строку, состоящую из набора столбцов, и каждый столбец имеет четко определенный тип данных. Фрейм данных в Spark можно рассматривать, как аналог таблицы в реляционной базе данных. Несмотря на то что соглашение об именах может натолкнуть вас на мысль об объекте `pandas.DataFrame`, на самом деле DataFrame в Spark — это совсем другое дело. Дело в том, что DataFrame в среде Spark представляют распределенные наборы данных в кластере, а не локальные данные, где каждая строка данных хранится на одном компьютере. Хотя есть определенное сходство в том, как вы используете DataFrame и роль, которую они играют в экосистеме Spark, некоторые вещи, доступные вам при работе с фреймами данных в pandas или R, не имеют отношения к Spark, поэтому лучше с самого начала воспринимайте их как разные сущности и старайтесь не смешивать.

Что касается очистки данных, то это первый шаг в любом проекте по анализу данных и часто самый важный. Многие аналитические исследования были забракованы, потому что проанализированные данные имели фундаментальные проблемы с качеством или скрытые артефакты, которые искажали анализ или заставляли специалиста по данным видеть то, чего не было в действительности. Согласитесь, нет лучшего способа познакомить вас с работой с данными, используя PySpark и DataFrames, чем выполнить упражнение по очистке данных.

Мы начнем со знакомства с основами PySpark и закрепим знания на примере набора данных из репозитория машинного обучения Калифорнийского университета в Ирвине. Мы покажем, почему PySpark — хороший выбор для науки о данных, и представим его модель программирования. Затем мы настроим PySpark в нашей системе или кластере и проанализируем набор данных с помощью API PySpark DataFrame. В большинстве задач анализа данных с помощью PySpark вы будете использовать API DataFrame, так что постарайтесь познакомиться с ним поближе. Это подготовит вас к следующим главам, в которых мы займемся освоением различных алгоритмов машинного обучения.

Для решения прикладных задач в области науки о данных от вас не требуется глубокого знания принципов работы Spark. Однако понимание основных концепций архитектуры Spark облегчит работу с PySpark и поможет принимать более правильные решения при написании кода. Этот вопрос мы рассмотрим в следующем разделе.

DataFrame и RDD

RDD (resilient distributed dataset, устойчивый распределенный набор данных) — это самая основная абстракция в Spark. Это неизменяемая распределенная коллекция элементов ваших данных, разделенных по машинам в кластере Spark. Разделы могут работать параллельно с низкоуровневым API, который предлагает преобразования и действия. На момент создания Spark это была основная абстракция, обращенная к пользователю. Однако у нее есть некоторые проблемы. Следует учесть, что любые вычисления, которые вы выполняете поверх RDD, непрозрачны для Spark Core. Следовательно, нет встроенного механизма оптимизации, которым можно было бы воспользоваться. В случае с PySpark проблема становится еще острее. Мы не будем вдаваться в детали на уровне архитектуры, поскольку это выходит за рамки данной книги.

Концепция DataFrame была представлена в Spark 1.3. Фреймы данных подобны распределенным таблицам в памяти с именованными столбцами и схемами, где каждый столбец хранит определенный тип данных: целое число, строка, массив, вещественное число, дата, отметка времени и т. д. Также существует базовый набор операций, которые позволяют выполнять типичные вычисления (объединения, агрегации). Благодаря этому Spark может создать эффективный план выполнения, обеспечивающий более высокую производительность по сравнению с RDD.

Мы будем изучать и использовать DataFrame на протяжении всей книги. Если вы хотите глубже изучить различия между DataFrame и RDD, отличным источником послужит книга Learning Spark (издательство O'Reilly).

При использовании API DataFrame ваш код PySpark должен обеспечивать сравнимую производительность со Scala. Если вы используете UDF или RDD, это повлияет на производительность.

Архитектура Spark

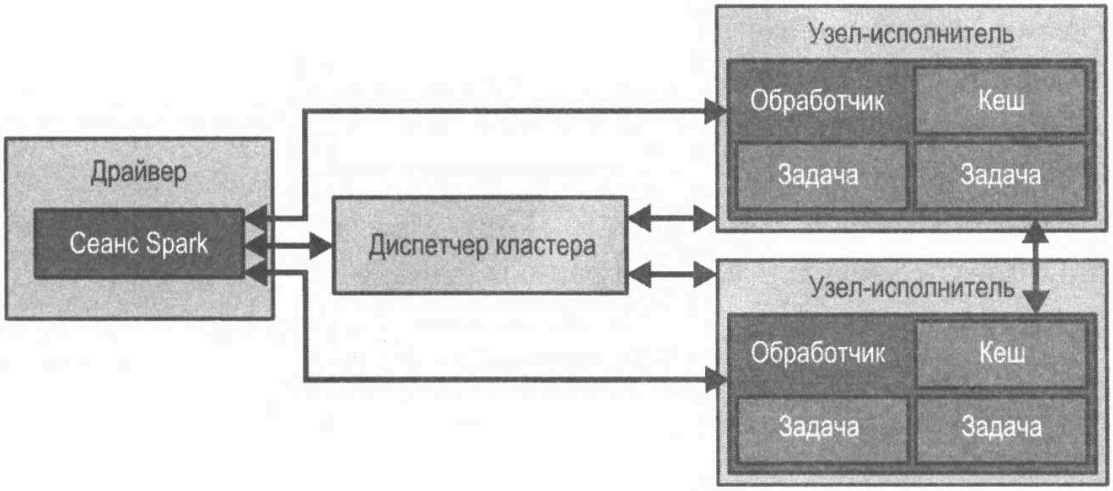


Рис. 2.1. Схема архитектуры Spark

На рис. 2.1 изображены высокоуровневые компоненты архитектуры Spark. Приложения Spark запускаются как независимые наборы процессов в кластере или на локальной машине. На высоком уровне приложение Spark состоит из процесса-драйвера, диспетчера кластера и набора процессов-исполнителей. *Драйвер* является центральным компонентом и отвечает за распределение задач между *исполнителями*[1]. Всегда существует только один процесс-драйвер. Когда мы говорим о масштабировании, мы имеем в виду увеличение количества исполнителей. *Диспетчер кластера* просто управляет ресурсами.

Spark — это распределенная вычислительная машина с параллельными данными. В модели с параллельными данными чем больше разделов данных, тем больше параллелизм. Разделение обеспечивает эффективное распараллеливание. Распределенная схема разбиения данных на разделы позволяет исполнителям Spark обрабатывать только близко расположенные данные, минимизируя нагрузку на сеть. То есть каждому ядру исполнителя назначается собственный раздел данных для работы. Имейте это в виду всякий раз, когда принимаете решение о необходимости разделения данных.

[1] Иногда их называют воркерами (worker). — *Примеч. пер.*

Программирование Spark начинается с набора данных, обычно представленного в виде распределенного постоянного хранилища, такого, как распределенная файловая система Hadoop (Hadoop distributed file system, HDFS) или облачное решение, такое, как AWS S3, и в таком формате, как Parquet. Разработка программы Spark обычно выполняется в несколько шагов:

1. Определение набора преобразований для входного набора данных.
2. Вызов действий, которые выводят преобразованные наборы данных в постоянное хранилище или возвращают результаты в локальную память драйвера. В идеале эти действия должны выполняться узлами-исполнителями, как показано справа на рис. 2.1.
3. Запуск локальных вычислений, которые работают с результатами, полученными распределенным способом. На этом этапе можно решить, какие преобразования и действия предпринять дальше.

Важно помнить, что все высокоуровневые абстракции PySpark по-прежнему опираются на философию, которая присутствовала в Spark с самого начала: *взаимодействие между хранением и выполнением*. Понимание этого принципа поможет вам лучше использовать Spark для анализа данных.

Далее мы установим и настроим PySpark на нашем компьютере и приступим к анализу данных. Обязательно выполните эти действия, потому что иначе вы не сможете запустить примеры кода из этой и следующих глав.

Установка PySpark

Примеры кода в этой книге предполагают, что у вас есть Spark 3.1.1. Чтобы гарантировать наличие правильной версии, установите PySpark из репозитория PyPI.

```
$ pip3 install pyspark
```

Обратите внимание, что для PySpark требуется установить Java 8 или более позднюю версию. Если вам нужны SQL, ML и/или MLlib в качестве дополнительных зависимостей, можете установить их прямо сейчас. Все равно они понадобятся нам позже.

```
$ pip3 install pyspark[sql,ml,mllib]
```

При установке из PyPI пропускаются библиотеки, необходимые для запуска Scala, Java или R. Полные релизы можно получить на сайте проекта Spark. Обратитесь к документации Spark за инструкциями по настройке среды Spark в кластере или просто на вашем локальном компьютере.

Теперь мы готовы запустить `pyspark-shell`, который представляет собой REPL для языка Python с некоторыми расширениями, специфичными для Spark. Это похоже на оболочку Python или IPython, которую вам, возможно, доводилось использовать раньше. Если вы просто выполняете примеры на своем персональном компьютере, вы можете запустить локальный кластер Spark, указав опцию `local[N]`, где *N* — ко-

личество выполняемых потоков или *, что соответствует количеству ядер, доступных на вашем компьютере. Например, для запуска локального кластера, использующего восемь потоков на восьмиядерной машине, введите такую команду:

```
$ pyspark --master local[*]
```

Само приложение Spark тоже часто называют *кластером* Spark. Это логическая абстракция, и она отличается от физического кластера (несколько машин).

Если у вас есть кластер Hadoop, на котором работает версия Hadoop, поддерживающая YARN, вы можете запускать задания Spark в кластере, используя опцию `yarn` с аргументом `--master`:

```
$ pyspark --master yarn --deploy-mode client
```

В остальных примерах этой книги мы не будем указывать аргумент `--master` для `spark-shell`, но обычно вам потребуется указать этот аргумент в соответствии с вашей средой выполнения.

Возможно, вам потребуется указать дополнительные аргументы, чтобы оболочка Spark полностью использовала доступные ресурсы. Список аргументов можно получить, выполнив команду `pyspark --help`. Например, при запуске Spark с локальным диспетчером вы можете использовать `--driver-memory 2g`, чтобы позволить одному локальному процессу использовать 2 ГБ памяти. Конфигурация памяти YARN более сложна, и назначение соответствующих параметров, таких, как `--executor-memory`, объясняется в документации Spark по YARN.

Платформа Spark официально поддерживает четыре режима развертывания кластера: автономный, YARN, Kubernetes и Mesos. Дополнительные сведения можно найти в документации по развертыванию Spark.

После запуска одной из этих команд вы увидите множество сообщений журнала Spark, поскольку он выполняет процесс инициализации, а затем появится ASCII-лого, за которым следуют дополнительные сообщения журнала и приглашение командной строки:

```
Python 3.6.12 |Anaconda, Inc.| (default, Sep 8 2020, 23:10:56)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.0.1
      /_/

Using Python version 3.6.12 (default, Sep 8 2020 23:10:56)
SparkSession available as 'spark'.
```

Вы можете запустить в оболочке команду `:help`. В ответ она предложит вам либо запустить режим интерактивной справки, либо запросить помощь по конкретным

объектам Python. В дополнение к примечанию о `:help` в сообщениях журнала Spark сказано `SparkSession available as spark` (`SparkSession` доступен как `spark`). Это ссылка на сеанс `SparkSession`, которая действует как точка входа для всех операций и данных Spark. Теперь введите в командной строке `spark`:

```
spark
...
<pyspark.sql.session.SparkSession object at DEADBEEF>
```

REPL напечатает строковое представление объекта. Для объекта `SparkSession` это просто его имя плюс шестнадцатеричный адрес объекта в памяти. (`DEADBEEF` — это заполнитель; точное значение, которое вы видите у себя, будет меняться от запуска к запуску.) В интерактивной оболочке драйвер Spark создает для вас экземпляр `SparkSession`, а в приложении Spark вы сами создаете объект `SparkSession`.

В Spark 2.0 SparkSession стал единой точкой входа для всех операций и данных Spark. Через него также можно получить доступ к ранее использовавшимся точкам входа, таким, как SparkContext, SQLContext, HiveContext, SparkConf и StreamingContext.

Что именно мы можем сделать с переменной `spark`? `SparkSession` — это объект, поэтому с ним связаны методы. Мы можем увидеть, что это за методы в оболочке PySpark, введя имя переменной, затем точку и символ табуляции:

```
spark.[\t]
...
spark.Builder(          spark.conf
spark.newSession(       spark.readStream
spark.stop(             spark.udf
spark.builder           spark.createDataFrame(
spark.range(            spark.sparkContext
spark.streams           spark.version
spark.catalog           spark.getActiveSession(
spark.read              spark.sql(
spark.table(
```

Из всех методов, предоставляемых SparkSession, те, которые мы собираемся использовать чаще всего, позволяют нам создавать фреймы данных. Теперь, завершив настройку PySpark, мы можем настроить интересующий нас набор данных и начать использовать API DataFrame PySpark, взаимодействуя с ним. Мы займемся этим в следующем разделе.

Подготовка учебных данных

Репозиторий машинного обучения Калифорнийского университета в Ирвине — отличный источник интересных (и бесплатных) наборов данных для исследований и образования. Набор данных, который мы будем анализировать, был создан на основе исследования взаимосвязей между записями, проведенного в немецкой боль-

нице в 2010 г., и содержит несколько миллионов пар записей пациентов, которые были сопоставлены по нескольким различным критериям, таким, как имя и фамилия пациента, адрес и день рождения. Каждому совпадающему полю был присвоен числовой балл от 0,0 до 1,0 в зависимости от того, насколько похожи строки, а затем данные были помечены вручную, чтобы определить, какие пары представляют одного и того же человека, а какие нет. Базовые значения полей, которые использовались для создания набора данных, были удалены для защиты конфиденциальности пациентов. Числовые идентификаторы, представляющие собой оценки совпадений для полей и метки для каждой пары (совпадение или несовпадение), были опубликованы для использования в исследованиях связей между записями.

Связывание записей

Задача связывания записей в общем виде выглядит примерно так: у нас есть большая коллекция записей из одной или нескольких исходных систем, и вполне вероятно, что несколько записей относятся к одному и тому же базовому объекту, такому, как клиент, пациент, место проведения мероприятия или само мероприятие. Каждая сущность имеет ряд атрибутов, таких, как имя, адрес или день рождения, и нам нужно будет использовать эти атрибуты, чтобы найти записи, относящиеся к одной и той же сущности. К сожалению, значения этих атрибутов не идеальны: они могут иметь различное форматирование, опечатки или отсутствующую информацию, а это означает, что простая проверка на совпадение значений атрибутов приведет к тому, что мы пропустим значительное количество повторяющихся записей. Например, давайте сравним записи с атрибутами компаний, показанные в табл. 2.1.

Таблица 2.1. *Задача связи записей (для наглядности данные приведены без перевода)*

Название	Адрес	Город	Штат	Телефон
Josh's Coffee Shop	1234 Sunset Boulevard	West Hollywood	CA	(213)-555-1212
Josh Coffee	1234 Sunset Blvd West	Hollywood	CA	555-1212
Coffee Chain #1234	1400 Sunset Blvd #2	Hollywood	CA	206-555-1212
Coffee Chain Regional Office	1400 Sunset Blvd Suite 2	Hollywood	California	206-555-1212

Первые две записи в этой таблице относятся к *одной и той же* небольшой кофейне, хотя из-за ошибки ввода данных кажется, что они находятся в двух разных городах (Западном Голливуде и Голливуде). Вторые две записи, напротив, на самом деле относятся к *разным* офисам одной и той же сети кофеен, которые имеют общий адрес: одна из записей относится к кофейне, а другая относится к местному корпоративному офису, который находится в том же здании. В обеих записях указан официальный номер телефона штаб-квартиры корпорации в Сиэтле.

Этот пример иллюстрирует все, что делает связывание записей таким сложным: хотя записи в обеих парах выглядят похожими друг на друга, критерии, которые мы используем для принятия решения о наличии или отсутствии дублирования, уникальные для каждой пары. Это различие из числа тех, которые легко обнаруживаются и распознаются человеком, но очень плохо даются компьютеру.

Связывание записей имеет много разных названий в литературе и на практике: разрешение сущностей, дедупликация записей, слияние и очистка, промывка списков. В этой книге мы будем называть такую задачу связыванием записей.

Используя оболочку командной строки, извлечем данные из репозитория:

```
$ mkdir linkage
$ cd linkage/
$ curl -L -o donation.zip https://bit.ly/1Aoywaq
$ unzip donation.zip
$ unzip 'block_*.zip'
```

Если у вас есть под рукой кластер Hadoop, вы можете создать каталог для блоковых данных в HDFS и скопировать туда файлы из набора данных:

```
$ hadoop dfs -mkdir linkage
$ hadoop dfs -put block_*.csv linkage
```

Чтобы создать фрейм данных для нашего набора данных связывания записей, мы будем использовать объект `Spark Session`. В частности, мы будем использовать метод `csv` в API Reader:

```
prev = spark.read.csv("linkage/block*.csv")
...
prev
...
DataFrame[_c0: string, _c1: string, _c2: string, _c3: string,...
```

По умолчанию каждый столбец в файле CSV обрабатывается как строковый тип `string`, а имена столбцов по умолчанию имеют вид `_c0`, `_c1`, `_c2` и т. д. Мы можем посмотреть заголовок фрейма данных в оболочке, вызвав его метод `show`:

```
prev.show(2)
...
+-----+-----+------------+------------+------------+------------+-------+------+
|  _c0|  _c1|         _c2|         _c3|         _c4|         _c5|    _c6|   _c7|
+-----+-----+------------+------------+------------+------------+-------+------+
| id_1| id_2|cmp_fname_c1|cmp_fname_c2|cmp_lname_c1|cmp_lname_c2|cmp_sex|cmp_bd|
| 3148| 8326|           1|           ?|           1|           ?|      1|     1|
|14055|94934|           1|           ?|           1|           ?|      1|     1|
|33948|34740|           1|           ?|           1|           ?|      1|     1|
|  946|71870|           1|           ?|           1|           ?|      1|     1|
```

Мы видим, что первая строка DataFrame, как мы и ожидали, — это имя столбцов заголовка и что CSV-файл четко разделен на отдельные столбцы. Мы также видим символы `?` в некоторых столбцах, которые нам нужно будет обрабатывать как отсутствующие значения. В дополнение к правильному именованию каждого столбца

было бы идеально, если бы Spark мог правильно определить для нас тип данных каждого из столбцов.

К счастью, считыватель CSV от Spark предоставляет нам такую возможность с помощью параметров, которые мы можем задать в API. Вы можете увидеть полный список параметров, которые принимает API, в документации `pyspark`. На данный момент мы будем читать и анализировать данные связывания записей следующим образом:

```
parsed = spark.read.option("header", "true").option("nullValue", "?").\
         option("inferSchema", "true").csv("linkage/block*.csv")
```

Когда мы вызываем метод `show` для проанализированных данных, мы видим, что имена столбцов установлены правильно, а символы `?` были заменены нулевыми значениями. Чтобы увидеть предполагаемый тип данных для каждого столбца, мы можем распечатать схему нашего проанализированного фрейма данных `parsed` следующим образом:

```
parsed.printSchema()
...
root
 |-- id_1: integer (nullable = true)
 |-- id_2: integer (nullable = true)
 |-- cmp_fname_c1: double (nullable = true)
 |-- cmp_fname_c2: double (nullable = true)
...
```

Каждый экземпляр `Column` содержит имя столбца, наиболее подходящий тип данных, который может поддерживать данные, содержащиеся в каждой записи, и логическое поле, указывающее, может ли столбец содержать нулевые значения (по умолчанию `true` — истина). Чтобы вывести на экран схему, Spark должен выполнить два прохода по набору данных: один проход для определения типа каждого столбца и второй проход для фактического синтаксического анализа. Первый проход, если это необходимо, может работать на ограниченной выборке.

Если вы заранее знаете схему, которую хотите использовать для файла, вы можете создать экземпляр класса `pyspark.sql.types.StructType` и передать его API считывателя через функцию `schema`. Когда набор данных очень большой, это может дать значительный выигрыш в быстродействии, поскольку Spark не нужно будет выполнять дополнительный проход по данным, чтобы определить тип данных каждого столбца.

Вот пример определения схемы с использованием `StructType` и `StructField`:

```
from pyspark.sql.types import *
schema = StructType([StructField("id_1", IntegerType(), False),
  StructField("id_2", StringType(), False),
  StructField("cmp_fname_c1", DoubleType(), False)])

spark.read.schema(schema).csv("...")
```

Другой способ определить схему — использовать операторы DDL (data definition language, язык определения данных):

```
schema = "id_1 INT, id_2 INT, cmp_fname_c1 DOUBLE"
```

Форматы и источники данных

Spark поставляется со встроенной поддержкой чтения и записи фреймов данных в различных форматах через API `DataFrameReader` и `DataFrameWriter`. В дополнение к обсуждаемому здесь формату CSV вы также можете читать и записывать структурированные данные из следующих источников:

`parquet`

Основной формат хранения данных, ориентированный на столбцы (опция по умолчанию в Spark).

`orc`

Еще один формат хранения данных, ориентированный на столбцы.

`json`

Поддерживает многие из тех же функций вывода схемы, что и формат CSV.

`jdbc`

Подключается к реляционной базе данных, используя стандарт подключения к данным JDBC.

`avro`

Обеспечивает эффективную сериализацию и десериализацию сообщений при использовании источника потоковых данных, такого, как Apache Kafka.

`text`

Сопоставляет каждую строку файла с фреймом данных с одним столбцом типа `string`.

`image`

Загружает файлы изображений из каталога в виде фрейма с одним столбцом, содержащим данные изображения, хранящиеся в виде схемы изображения.

`libsvm`

Популярный формат текстового файла для представления помеченных наблюдений с разреженными признаками.

`binary`

Читает двоичные файлы и преобразует каждый файл в одну строку фрейма данных (добавлено в Spark 3.0).

`xml`

Простой текстовый формат для представления структурированной информации, такой, как документы, данные, конфигурация или книги (доступен в пакете `spark-xml`).

Вы получаете доступ к методам API `DataFrameReader`, вызывая метод `read` в экземпляре `SparkSession`, и можете загружать данные из файла, используя методы `format` и `load` или один из методов быстрого доступа для встроенных форматов:

```
d1 = spark.read.format("json").load("file.json")
d2 = spark.read.json("file.json")
```

В этом примере `d1` и `d2` ссылаются на одни и те же базовые данные JSON и будут иметь одинаковое содержимое. Каждый из различных форматов файлов имеет свой собственный набор параметров, которые можно установить с помощью того же метода `option`, который мы использовали для файлов CSV.

Чтобы снова записать данные, вы получаете доступ к API `DataFrameWriter` через метод `write` в любом экземпляре DataFrame. API `DataFrameWriter` поддерживает те же встроенные форматы, что и API `DataFrameReader`, поэтому следующие два метода являются эквивалентными способами записи содержимого `d1` DataFrame в виде файла Parquet:

```
d1.write.format("parquet").save("file.parquet")
d1.write.parquet("file.parquet")
```

По умолчанию Spark выдаст ошибку, если вы попытаетесь сохранить фрейм данных в уже существующий файл. Вы можете управлять поведением Spark в этой ситуации с помощью метода `mode` в API `DataFrameWriter`, чтобы перезаписать (`Overwrite`) существующий файл, добавить (`Append`) данные в DataFrame к файлу (если он существует), либо игнорировать (`Ignore`) операцию записи, если файл уже существует и оставить его без изменений:

```
d2.write.format("parquet").mode("overwrite").save("file.parquet")
```

Вы можете указать режим как строковый литерал (`"overwrite"`, `"append"`, `"ignore"`).

DataFrames имеет ряд методов, которые позволяют нам считывать данные из кластера в PySpark REPL на своем клиентском компьютере. Возможно, самым простым из них является `first`, который возвращает клиенту первый элемент фрейма данных:

```
parsed.first()
...
Row(id_1=3148, id_2=8326, cmp_fname_c1=1.0, cmp_fname_c2=None,...
```

Первый метод может быть полезен для проверки работоспособности набора данных, но обычно мы заинтересованы в том, чтобы извлекать большие выборки фрейма данных в клиентский компьютер для анализа. Когда мы знаем, что фрейм данных содержит лишь небольшое количество записей, мы можем использовать метод `toPandas` или `collect` для возврата клиенту всего содержимого DataFrame в виде массива. В случае очень больших фреймов использование этих методов грозит вызвать исключение из-за нехватки памяти. Поскольку мы пока не знаем, насколько велик набор данных связывания, временно отложим эту операцию.

В следующих нескольких разделах мы будем использовать сочетание локальной разработки и тестирования, а также кластерных вычислений, чтобы выполнить дополнительную обработку и анализ данных о связывании записей. Но, возможно, вам потребуется некоторое время, чтобы прийти в себя, после того как вы окунулись в новый мир удивительных вещей.

Преобразования и действия

Создание фрейма данных не приводит к распределенным вычислениям в кластере. Вместо этого он определяет логические наборы данных, которые являются промежуточными этапами вычислений. Операции Spark с распределенными данными можно разделить на два типа: *преобразования* и *действия*.

Результаты преобразований не вычисляются сразу, а записываются в виде последовательности операций. Это позволяет Spark оптимизировать план запроса. Распределенные вычисления происходят при вызове действия над фреймом данных. Например, действие `count` возвращает количество объектов в DataFrame:

```
df.count()
...
15
```

Действие `collect` возвращает массив `Array` со всеми объектами `Row` из DataFrame. Этот массив находится в локальной памяти, а не в кластере:

```
df.collect()
[Row(id='12', department='sales'), ...
```

Действия должны не только возвращать результаты локальному процессу. Действие `save` сохраняет содержимое DataFrame в постоянном хранилище:

```
df.write.format("parquet").("user/ds/mynumbers")
```

Помните, что `DataFrameReader` может принимать каталог текстовых файлов в качестве входных данных, а это означает, что будущее задание Spark может ссылаться на `mynumbers` как на входной каталог.

Анализ данных с помощью API DataFrame

API DataFrame поставляется с мощным набором инструментов, которые, вероятно, будут знакомы специалистам по данным, ранее работавшим с Python и SQL. В данном разделе мы приступим к изучению этих инструментов и покажем, как их можно применить к данным о связывании записей.

Если мы взглянем на схему фрейма `parsed` и первые несколько строк данных в нем, то увидим следующее:

```
parsed.printSchema()
...
root
```

```
 |-- id_1: integer (nullable = true)
 |-- id_2: integer (nullable = true)
 |-- cmp_fname_c1: double (nullable = true)
 |-- cmp_fname_c2: double (nullable = true)
 |-- cmp_lname_c1: double (nullable = true)
 |-- cmp_lname_c2: double (nullable = true)
 |-- cmp_sex: integer (nullable = true)
 |-- cmp_bd: integer (nullable = true)
 |-- cmp_bm: integer (nullable = true)
 |-- cmp_by: integer (nullable = true)
 |-- cmp_plz: integer (nullable = true)
 |-- is_match: boolean (nullable = true)

...
```

```
parsed.show(5)
...
+-----+-----+------------+------------+------------+------------+.....
| id_1| id_2|cmp_fname_c1|cmp_fname_c2|cmp_lname_c1|cmp_lname_c2|.....
+-----+-----+------------+------------+------------+------------+.....
| 3148| 8326|         1.0|        null|         1.0|        null|.....
|14055|94934|         1.0|        null|         1.0|        null|.....
|33948|34740|         1.0|        null|         1.0|        null|.....
|  946|71870|         1.0|        null|         1.0|        null|.....
|64880|71676|         1.0|        null|         1.0|        null|.....
```

- Первые два поля — это целочисленные идентификаторы, представляющие пациентов, которые сопоставлены в записи.
- Следующие девять полей содержат числовые значения (возможно, отсутствующие), представляющие оценки совпадений в различных полях записей пациентов, таких, как их имена, дни рождения и местонахождение. Поля сохраняются как целые числа, когда единственными возможными значениями являются совпадение (1) или несоответствие (0), и числа двойной точности, когда возможны частичные совпадения.
- Последнее поле представляет собой логическое значение (`true` или `false`), указывающее, совпадала ли пара записей пациентов, представленных линией.

Наша цель — придумать простой классификатор, который позволит нам предсказывать, на основе значений оценок совпадений для записей пациентов, будет ли запись совпадать,. Давайте начнем с того, что получим представление о количестве записей, с которыми мы имеем дело, с помощью метода `count`:

```
parsed.count()
...
5749132
```

Это относительно небольшой набор данных — настолько маленький, чтобы поместиться в памяти на одном из узлов в кластере или даже на вашем локальном компьютере, если у вас нет своего кластера. До сих пор каждый раз, когда мы обрабатывали данные, Spark повторно открывал файл, повторно анализировал строки, а затем выполнял запрошенное действие, например, отображал первые несколько строк данных или подсчитывал количество записей. Когда мы зададим другой вопрос, Spark будет выполнять одни и те же операции снова и снова, даже если мы отфильтровали данные до небольшого количества записей или работаем с агрегированной версией исходного набора данных.

Это неоптимальное использование наших вычислительных ресурсов. После того как данные были проанализированы один раз, было бы разумно сохранить их в проанализированном виде в кластере, чтобы нам не приходилось повторять анализ каждый раз, когда мы хотим задать новый вопрос. Spark допускает такой вариант использования, позволяя нам указать, что данный фрейм должен быть кэширован в памяти после его создания, вызывая метод `cache` для экземпляра. Давайте сделаем это сейчас для нашего фрейма данных `parsed`:

```
parsed.cache()
```

После того как наши данные были кэшированы, следующее, что нам нужно узнать, — это относительная доля записей, которые совпадали, по сравнению с несовпадающими:

```
from pyspark.sql.functions import col

parsed.groupBy("is_match").count().orderBy(col("count").desc()).show()
...
+--------+-------+
|is_match|  count|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

Вместо того чтобы писать функцию для извлечения столбца `is_match`, мы просто передаем его имя методу `groupBy` в DataFrame, вызываем метод `count`, чтобы подсчитать количество записей внутри каждой группы, сортируем полученные данные в порядке убывания на основе столбца `count`, а затем визуализируем результат вычислений в REPL с помощью `show`. За кулисами механизм Spark определяет наиболее эффективный способ выполнения агрегирования и возвращает результат. Этот пример иллюстрирует ясный, быстрый и наглядный способ анализа данных, который предоставляет Spark.

Обратите внимание, что есть два способа, которыми мы можем ссылаться на имена столбцов во фрейме данных: либо как литеральные строки, например, в `groupBy("is_match")`, либо как объекты `Columns`, используя функцию `col`, которую мы применяли к столбцу `count`. В большинстве случаев допустим любой подход, но мы использовали функцию `col`, чтобы вызвать метод `desc` для результирующего объекта столбца `count`.

Функции агрегации DataFrame

Помимо подсчета, мы также можем вычислять более сложные агрегации, такие, как суммы, минимальные и максимальные значения, средние значения и стандартное отклонение, используя метод `agg` API DataFrame в сочетании с функциями агрегирования, определенными в коллекции `pyspark.sql.functions`. Например, чтобы найти среднее значение и стандартное отклонение поля `cmp_sex` во всем фрейме `parsed`, мы могли бы ввести:

```
from pyspark.sql.functions import avg, stddev

parsed.agg(avg("cmp_sex"), stddev("cmp_sex")).show()
+-----------------+--------------------+
|     avg(cmp_sex)|stddev_samp(cmp_sex)|
+-----------------+--------------------+
|0.955001381078048|  0.2073011111689795|
+-----------------+--------------------+
```

Обратите внимание, что по умолчанию Spark вычисляет стандартное отклонение выборки; существует также функция `stddev_pop` для вычисления стандартного отклонения генеральной совокупности.

Возможно, вы заметили, что функции API DataFrame аналогичны компонентам SQL-запроса. Это не совпадение, и на самом деле у нас есть возможность рассматривать любой фрейм данных, который мы создаем, как некую таблицу базы данных и формулировать наши вопросы, используя знакомый и мощный синтаксис SQL. Для начала нам нужно сообщить механизму выполнения Spark SQL имя, которое он должен ассоциировать с фреймом `parsed`, поскольку имя самой переменной (`parsed`) недоступно для Spark:

```
parsed.createOrReplaceTempView("linkage")
```

Поскольку DataFrame `parsed` доступен только в течение текущего сеанса PySpark REPL, это *временная* таблица. Spark SQL также можно использовать для запросов к постоянным таблицам в HDFS, если мы настроим Spark для подключения к хранилищу метаданных Apache Hive, которое отслеживает схемы и расположение структурированных наборов данных.

Как только наша временная таблица зарегистрирована в движке Spark SQL, мы можем обратиться к ней с запросом следующим образом:

```
spark.sql("""
  SELECT is_match, COUNT(*) cnt
  FROM linkage
  GROUP BY is_match
  ORDER BY cnt DESC
""").show()
...
```

```
+--------+-------+
|is_match|    cnt|
+--------+-------+
|   false|5728201|
|    true|  20931|
+--------+-------+
```

У вас есть возможность запустить Spark либо с помощью версии Spark SQL, совместимой с ANSI 2003 (по умолчанию), либо в режиме HiveQL, вызвав метод `enableHiveSupport` при создании экземпляра `SparkSession` через его API Builder.

Подключение Spark SQL к Hive

Вы можете подключиться к хранилищу метаданных Hive через файл hive-site.xml, а также использовать HiveQL в запросах, вызвав метод `enableHiveSupport` в API SparkSession Builder:

```
spark_session = SparkSession.builder.master("local[4]").\
                enableHiveSupport().getOrCreate()
```

Вы можете рассматривать любую таблицу в хранилище метаданных Hive как фрейм данных, выполнять SQL-запросы Spark к таблицам, определенным в этом хранилище, а также сохранять выходные данные этих запросов в хранилище метаданных, чтобы их можно было запрашивать с помощью других инструментов, включая сам Hive, Apache Impala или Presto.

Что лучше использовать для анализа в PySpark — Spark SQL или API DataFrame? У каждого есть свои плюсы и минусы. Преимущество SQL заключается в том, что он широко распространен и удобен для простых запросов. Он также позволяет запрашивать данные с помощью коннекторов JDBC/ODBC из таких баз данных, как PostgreSQL, или таких инструментов, как Tableau. Недостаток SQL заключается в том, что сложный, многоэтапный анализ бывает трудно выполнить динамическим, удобочитаемым и тестируемым способом — именно здесь себя блестяще проявляет API DataFrame. В оставшейся части книги мы используем как Spark SQL, так и API DataFrame и оставляем читателю в качестве упражнения возможность изучить сделанный нами выбор и перевести наши вычисления из одного интерфейса в другой.

Мы можем последовательно применять функции к нашему фрейму данных для получения таких статистических данных, как количество и среднее значение. Однако PySpark предлагает лучший способ получения сводной статистики для DataFrames. Мы рассмотрим его следующем разделе.

Быстрая сводная статистика для фреймов данных

Несмотря на то что существует много видов анализа, которые могут быть одинаково хорошо выполнены при помощи SQL или API DataFrame, есть некоторые вещи,

которые утомительно делать в SQL. Одним из таких часто востребованных анализов является вычисление минимального, максимального, среднего значения и стандартного отклонения всех ненулевых значений в числовых столбцах фрейма данных. В PySpark эта функция имеет то же имя `describe`, что и в pandas:

```
summary = parsed.describe()
...
summary.show()
```

Вывод `summary` DataFrame содержит один столбец для каждой переменной в фрейме `parsed` и другой столбец (также называемый `summary`), который указывает, какая метрика — `count`, `mean`, `stddev`, `min` или `max` — присутствует в остальных столбцах строки. Мы можем использовать метод `select`, чтобы выбрать подмножество столбцов и тем самым упростить чтение и сравнение сводной статистики:

```
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()
+-------+------------------+------------------+
|summary|      cmp_fname_c1|      cmp_fname_c2|
+-------+------------------+------------------+
|  count|           5748125|            103698|
|   mean|0.7129024704436274|0.9000176718903216|
| stddev|0.3887583596162788|0.2713176105782331|
|    min|               0.0|               0.0|
|    max|               1.0|               1.0|
+-------+------------------+------------------+
```

Обратите внимание на разницу в значении переменной `count` между `cmp_fname_c1` и `cmp_fname_c2`. Хотя почти каждая запись имеет ненулевое значение для `cmp_fname_c1`, менее 2% записей имеют ненулевое значение для `cmp_fname_c2`. Чтобы создать полезный классификатор, нам нужно полагаться на переменные, которые почти всегда присутствуют в данных, если только факт их отсутствия не указывает на что-то значимое в отношении совпадения записей.

Получив общее представление о распределении переменных в наших данных, мы попытаемся понять, как значения этих переменных коррелируют со значением столбца `is_match`. Поэтому наш следующий шаг — вычислить ту же сводную статистику только для подмножеств фрейма `parsed`, которые соответствуют совпадениям и несовпадениям. Мы можем фильтровать фрейм, используя либо синтаксис в стиле SQL, либо с помощью объектов `Column` и API DataFrame, а затем применить `describe` к полученным фреймам:

```
matches = parsed.where("is_match = true")
match_summary = matches.describe()

misses = parsed.filter(col("is_match") == False)
miss_summary = misses.describe()
```

Логика внутри строки, которую мы передаем функции `where`, может включать операторы, которые будут допустимы внутри предложения `WHERE` в Spark SQL. В условии фильтрации, использующем API DataFrame, мы применяем оператор `==` к объекту столбца `is_match`, чтобы проверить равенство с логическим объектом False,

потому что это всего лишь Python, а не SQL. Обратите внимание, что функция `where` является псевдонимом функции `filter`; в приведенном выше фрагменте мы можем поменять местами вызовы `where` и `filter`, и ничего не изменится.

Теперь мы можем начать сравнивать фреймы данных `match_summary` и `miss_summary`, чтобы увидеть, как меняется распределение переменных в зависимости от того, является ли запись совпадением или нет. Несмотря на то что это относительно небольшой набор данных, выполнение этого сравнения несколько утомительно. Что нам действительно нужно, так это транспонировать фреймы данных `match_summary` и `miss_summary`, чтобы строки и столбцы поменялись местами, это позволит нам объединить транспонированные фреймы данных с помощью переменной и анализировать сводную статистику — практика, известная большинству специалистов по данным как *поворот* (pivoting) или *изменение формы* (reshaping) набора данных. В следующем разделе мы покажем вам, как выполнять эти преобразования.

Поворот и изменение формы фреймов данных

Мы можем транспонировать фрейм данных, используя только функции, предоставляемые PySpark. Однако есть и другой способ выполнить эту задачу. PySpark позволяет конвертировать данные между фреймами Spark и pandas. Мы преобразуем рассматриваемые фреймы данных Spark в фреймы данных pandas, изменим их форму и преобразуем обратно в фреймы Spark. Благодаря небольшому размеру фреймов данных `summary`, `match_summary` и `miss_summary` мы можем делать это без опасений, хотя фреймы данных pandas целиком размещаются в памяти. Для аналогичных преобразований больших наборов данных в следующих главах мы будем использовать операции Spark.

Преобразование в формат pandas и обратно стало возможно благодаря проекту Apache Arrow, который обеспечивает эффективную передачу данных между процессами JVM и Python. Библиотека PyArrow была установлена как зависимость модуля Spark SQL, когда мы установили `pyspark[sql]` с помощью pip.

Преобразуем `summary` в фрейм данных pandas:

```
summary_p = summary.toPandas()
```

Теперь мы можем применить функции pandas к фрейму данных `summary_p`:

```
summary_p.head()
...
summary_p.shape
...
(5,12)
```

Далее мы можем выполнить операцию транспонирования, чтобы поменять местами строки и столбцы, используя знакомые методы pandas:

```
summary_p = summary_p.set_index('summary').transpose().reset_index()
...
```

```
summary_p = summary_p.rename(columns={'index':'field'})
...
summary_p = summary_p.rename_axis(None, axis=1)
...
summary_p.shape
...
(11,6)
```

Мы успешно транспонировали фрейм данных `summary_p` в pandas. Преобразуем его в Spark, используя метод `createDataFrame` в SparkSession:

```
summaryT = spark.createDataFrame(summary_p)
...
summaryT.show()
...
+------------+-------+-------------------+-------------------+---+------+
|       field|  count|               mean|             stddev|min|   max|
+------------+-------+-------------------+-------------------+---+------+
|        id_1|5749132|  33324.48559643438| 23659.859374488064|  1| 99980|
|        id_2|5749132|  66587.43558331935| 23620.487613269695|  6|100000|
|cmp_fname_c1|5748125| 0.7129024704437266|0.38875835961628014|0.0|   1.0|
|cmp_fname_c2| 103698| 0.9000176718903189| 0.2713176105782334|0.0|   1.0|
|cmp_lname_c1|5749132| 0.3156278193080383| 0.3342336339615828|0.0|   1.0|
|cmp_lname_c2|   2464| 0.3184128315317443|0.36856706620066537|0.0|   1.0|
|     cmp_sex|5749132|  0.955001381078048|0.20730111116897781|  0|     1|
|      cmp_bd|5748337|0.22446526708507172|0.41722972238462636|  0|     1|
|      cmp_bm|5748337|0.48885529849763504| 0.4998758236779031|  0|     1|
|      cmp_by|5748337| 0.2227485966810923| 0.4160909629831756|  0|     1|
|     cmp_plz|5736289|0.00552866147434343|0.07414914925420046|  0|     1|
+------------+-------+-------------------+-------------------+---+------+
```

Мы еще не закончили. Выведем на печать схему фрейма данных `summaryT`:

```
summaryT.printSchema()
...
root
 |-- field: string (nullable = true)
 |-- count: string (nullable = true)
 |-- mean: string (nullable = true)
 |-- stddev: string (nullable = true)
 |-- min: string (nullable = true)
 |-- max: string (nullable = true)
```

В сводной схеме, полученной методом `describe`, каждое поле обрабатывается как строка. Поскольку мы хотим анализировать сводную статистику в виде чисел, нам нужно преобразовать значения из строк в числа двойной точности:

```
from pyspark.sql.types import DoubleType
for c in summaryT.columns:
```

```
if c == 'field':
continue
summaryT = summaryT.withColumn(c, summaryT[c].cast(DoubleType()))
...
summaryT.printSchema()
...
root
 |-- field: string (nullable = true)
 |-- count: double (nullable = true)
 |-- mean: double (nullable = true)
 |-- stddev: double (nullable = true)
 |-- min: double (nullable = true)
 |-- max: double (nullable = true)
```

Теперь, когда мы выяснили, как транспонировать фрейм `summary`, давайте реализуем эту логику в виде функции, которую мы сможем повторно применить к фреймам данных `match_summary` и `miss_summary`:

```
from pyspark.sql import DataFrame
from pyspark.sql.types import DoubleType

def pivot_summary(desc):
  # конвертируем в фрейм данных pandas
  desc_p = desc.toPandas()
  # транспонируем
  desc_p = desc_p.set_index('summary').transpose().reset_index()
  desc_p = desc_p.rename(columns={'index':'field'})
  desc_p = desc_p.rename_axis(None, axis=1)
  # конвертируем в фрейм данных Spark
  descT = spark.createDataFrame(desc_p)
  # конвертируем строковые значения в числа двойной точности
  for c in descT.columns:
    if c == 'field':
      continue
    else:
      descT = descT.withColumn(c, descT[c].cast(DoubleType()))
  return descT
```

Далее в вашей оболочке Spark примените функцию `pivot_summary` к фреймам `match_summary` и `miss_summary`:

```
match_summaryT = pivot_summary(match_summary)
miss_summaryT = pivot_summary(miss_summary)
```

Теперь, когда у нас есть успешно транспонированные сводные фреймы данных, мы можем их соединить и сравнить, чем мы и займемся в следующем разделе. Далее мы также выберем желаемые признаки для построения нашей модели.

Соединение фреймов данных и выбор признаков

До сих пор мы использовали Spark SQL и API DataFrame только для фильтрации и агрегирования записей из набора данных, но мы также можем использовать эти инструменты для выполнения различных соединений (внутренних, левых внешних, правых внешних или полных внешних) в DataFrames. Хотя API DataFrame включает в себя функцию соединения `join`, часто бывает проще реализовать эти соединения с помощью Spark SQL, особенно когда таблицы, которые мы соединяем, имеют большое количество общих имен столбцов, и мы хотим иметь возможность четко указать, какой столбец мы имеем в виду в наших операторах `SELECT`. Давайте создадим временные представления для фреймов данных `match_summaryT` и `miss_summaryT`, соединим их в столбце `field` и вычислим простую сводную статистику по результирующим строкам:

```
match_summaryT.createOrReplaceTempView("match_desc")
miss_summaryT.createOrReplaceTempView("miss_desc")
spark.sql("""
  SELECT a.field, a.count + b.count total, a.mean - b.mean delta
  FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
  WHERE a.field NOT IN ("id_1", "id_2")
  ORDER BY delta DESC, total DESC
""").show()
...
+------------+---------+--------------------+
|       field|    total|               delta|
+------------+---------+--------------------+
|     cmp_plz|5736289.0|  0.9563812499852176|
|cmp_lname_c2|   2464.0|  0.8064147192926264|
|      cmp_by|5748337.0|  0.7762059675300512|
|      cmp_bd|5748337.0|   0.775442311783404|
|cmp_lname_c1|5749132.0|  0.6838772482590526|
|      cmp_bm|5748337.0|  0.5109496938298685|
|cmp_fname_c1|5748125.0|  0.2854529057460786|
|cmp_fname_c2| 103698.0| 0.09104268062280008|
|     cmp_sex|5749132.0|0.03240818525033284 4|
+------------+---------+--------------------+
```

Хороший признак имеет два свойства: у него существенно различаются значения для совпадений и несоответствий (поэтому разница между средними значениями будет большой), и он встречается в данных достаточно часто, так что мы можем надеяться, что он регулярно доступен для любой пары записей. По этим критериям `cmp_fname_c2` нам не подходит, потому что состоит по большей части из пропусков, а разница в среднем значении для совпадений и несовпадений относительно мала — 0,09 для оценки, которая колеблется от 0 до 1. Признак `cmp_sex` также не особенно полезен, потому что, хотя он доступен для любой пары записей, разница в средних значениях составляет всего 0,03.

Зато признаки cmp_plz и cmp_by просто превосходны. Они почти всегда встречаются для любой пары записей, и разница в средних значениях весьма велика (более 0,77 для обоих признаков). Признаки cmp_bd, cmp_lname_c1 и cmp_bm также выглядят полезными: они обычно доступны в наборе данных, а разница в средних значениях совпадений и несоответствий значительна.

Признаки cmp_fname_c1 и cmp_lname_c2 представляют собой нечто промежуточное: cmp_fname_c1 не очень хорошо отделяет совпадения от несовпадений (разница в средних значениях составляет всего 0,28), хотя обычно он доступен для пары записей, тогда как cmp_lname_c2 имеет большую разницу в средних значениях, но почти всегда отсутствует в данных. Поэтому не совсем понятно, при каких обстоятельствах мы должны включать эти признаки в нашу модель на основе текущих данных.

Сейчас мы будем использовать простую модель оценки, которая ранжирует сходство пар записей на основе сумм значений явно хороших признаков cmp_plz, cmp_by, cmp_bd, cmp_lname_c1 и cmp_bm. Для нескольких записей, в которых отсутствуют значения этих признаков, при суммировании мы будем использовать 0 вместо значения null. Мы можем получить приблизительное представление о точности нашей простой модели, создав фрейм данных вычисленных оценок и значений столбца is_match и определив, насколько хорошо оценка модели различает совпадения и несовпадения при различных пороговых значениях.

Функция оценивания и проверка модели

Для нашей функции оценивания мы суммируем значения пяти полей (cmp_lname_c1, cmp_plz, cmp_by, cmp_bd и cmp_bm). Для этого мы будем использовать expr из pyspark.sql.functions. Функция expr преобразует строку входного выражения в столбец, который она представляет. Эта строка может даже включать несколько столбцов.

Создадим необходимую строку выражения:

```
good_features = ["cmp_lname_c1", "cmp_plz", "cmp_by", "cmp_bd", "cmp_bm"]
...
sum_expression = " + ".join(good_features)
...
sum_expression
...
'cmp_lname_c1 + cmp_plz + cmp_by + cmp_bd + cmp_bm'
```

Теперь мы можем использовать строку sum_expression для вычисления оценки. При суммировании значений мы будем учитывать и заменять значения null на 0, используя метод fillna DataFrame:

```
from pyspark.sql.functions import expr
scored = parsed.fillna(0, subset=good_features).\
               withColumn('score', expr(sum_expression)).\
               select('score', 'is_match')
...
```

```
scored.show()
...
+-----+--------+
|score|is_match|
+-----+--------+
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  5.0|    true|
|  4.0|    true|
...
```

Последним шагом в создании нашей функции оценивания является решение, какой порог должен превысить показатель, чтобы мы могли предсказать, что две записи совпадают. Если мы установим слишком высокий порог, то мы неправильно пометим совпадающую запись как несовпадающую (так называемая *ложноотрицательная* оценка); но если мы установим слишком низкий порог, мы неправильно пометим несовпадения как совпадения (*ложноположительная* оценка). Для любой нетривиальной задачи нам всегда приходится искать компромисс между ложноположительными и ложноотрицательными оценками, и вопрос о том, каким должно быть пороговое значение, обычно сводится к относительной стоимости двух видов ошибок в ситуации, к которой применяется модель.

Чтобы от чего-то отталкиваться в выборе порогового значения, полезно создать *таблицу сопряженности признаков* (contingency table), которую иногда называют *факторной, сводной* или *перекрестной* таблицей. В ней подсчитывается количество записей, оценки которых оказываются выше/ниже порогового значения, в сопоставлении с количеством записей в каждой из тех категорий, которые были/не были совпадениями. Поскольку мы еще не знаем, какое пороговое значение мы собираемся использовать, давайте создадим функцию, которая принимает фрейм `scored` и выбранный порог в качестве параметров и вычисляет таблицы сопряженности с помощью API DataFrame:

```
def crossTabs(scored: DataFrame, t: DoubleType) -> DataFrame:
return scored.selectExpr(f"score >= {t} as above", "is_match").\
       groupBy("above").pivot("is_match", ("true", "false")).\
       count()
```

Обратите внимание, что мы применяем метод `selectExpr` API DataFrame для динамического определения значения поля `above` на основе значения аргумента `t` с использованием синтаксиса форматирования f-строки Python, что позволяет нам заменять переменные по имени, если мы предваряем строковый литерал буквой `f` (еще одно удобное заклинание тайной магии Scala). Как только указанное выше поле определено, мы создаем таблицу сопряженности при помощи стандартной комбинации методов `groupBy`, `pivot` и `count`, которые мы использовали ранее.

Применяя высокое пороговое значение 4.0, означающее, что среднее значение пяти признаков равно 0,8, мы можем отфильтровать почти все несовпадения, сохраняя при этом более 90% совпадений:

```
crossTabs(scored, 4.0).show()
...
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20871|    637|
|false|   60|5727564|
+-----+-----+-------+
```

Применив более низкий порог 2.0, мы можем гарантировать, что зафиксируем *все известные* совпадающие записи, но заплатим за это большим количеством ложноположительных оценок (верхняя правая ячейка):

```
crossTabs(scored, 2.0).show()
...
+-----+-----+-------+
|above| true|  false|
+-----+-----+-------+
| true|20931| 596414|
|false| null|5131787|
+-----+-----+-------+
```

Несмотря на то что количество ложноположительных оценок выше, чем хотелось бы, этот более щедрый фильтр по-прежнему удаляет из нашего рассмотрения 90% несовпадающих записей, оставляя каждое положительное совпадение. Хотя это весьма неплохо, можно сделать еще лучше; подумайте, сможете ли вы найти способ использовать некоторые другие значения из `MatchData` (как отсутствующие, так и нет), чтобы придумать функцию оценки, которая успешно идентифицирует каждое совпадение `true` ценой менее 100 ложноположительных срабатываний.

Что дальше?

Если в этой главе вы впервые выполняли подготовку и анализ данных с помощью PySpark, мы надеемся, что вы почувствовали, какую мощную основу для работы обеспечивают эти инструменты. Если вы уже некоторое время используете Python и Spark, мы надеемся, что вы предложите прочитать эту главу своим друзьям и коллегам, чтобы познакомить их с этой мощью.

Нашей целью в этой главе было дать вам достаточно знаний, чтобы вы смогли понять и завершить остальные примеры в этой книге. Если вы относитесь к тому типу людей, которые лучше всего учатся на практических примерах, ваш следующий шаг — перейти к следующим главам, где мы познакомим вас с MLlib, библиотекой машинного обучения, разработанной для Spark.

ГЛАВА 3

Рекомендация музыки и набор данных Audioscrobbler

Механизм рекомендаций — один из самых популярных примеров крупномасштабного машинного обучения; например, большинство людей знакомы с рекомендациями товаров на сайте Amazon. Это общий знаменатель, потому что рекомендательные механизмы есть везде, от социальных сетей до видеосайтов и интернет-магазинов. Мы также можем непосредственно наблюдать за ними в действии. К примеру, мы знаем, что компьютер выбирает для нас музыкальные треки для воспроизведения на Spotify, и не всегда задумываемся, что Gmail решает за нас, является ли входящая почта спамом.

Механизм рекомендаций, реализованный на практике, часто называют *рекомендательной системой*. Вывод рекомендательной системы более интуитивно понятен, чем другие алгоритмы машинного обучения. Он выглядит захватывающе. Несмотря на то что мы считаем свой музыкальный вкус индивидуальным и необъяснимым, рекомендательные системы на удивление хорошо справляются с подбором треков, которые нам понравятся. В таких областях, как музыка или фильмы, где часто используются рекомендательные системы, сравнительно легко объяснить, почему рекомендуемое музыкальное произведение соответствует чьей-то истории прослушивания. Увы, не все алгоритмы кластеризации или классификации поддаются такому объяснению. Например, машинный классификатор опорных векторов представляет собой набор коэффициентов, и даже опытным специалистам трудно сформулировать, что означают эти числа, когда делаются прогнозы.

Мы сочли наиболее уместным начать следующие три главы, в которых будут рассмотрены ключевые алгоритмы машинного обучения в PySpark, с главы, посвященной механизмам рекомендаций и, в частности, рекомендациям в области музыки. Это доступный способ продемонстрировать реальное использование PySpark и MLlib, а также некоторые основные идеи машинного обучения, которые будут применяться в последующих главах.

В этой главе мы реализуем рекомендательную систему в PySpark. В частности, мы будем использовать *алгоритм чередующихся наименьших квадратов* (alternating least squares, ALS) в открытом наборе данных, предоставленном службой потоковой передачи музыки. Мы начнем с изучения набора данных и его импорта в PySpark. Затем мы обсудим причины выбора алгоритма ALS и его реализацию в PySpark. За этим последует подготовка данных и построение нашей модели с ис-

пользованием PySpark. В заключение мы дадим некоторые рекомендации пользователям и обсудим способы улучшения нашей модели с помощью настройки гиперпараметров.

Подготовка данных

Мы будем использовать набор данных, опубликованный сервисом Audioscrobler. Это первая система музыкальных рекомендаций для Last.fm, одного из первых интернет-сайтов потокового радио, основанного в 2002 г. Audioscrobbler предоставил открытый API для «скробблинга», т. е. сбора информации о воспроизведении песен слушателями. Last.fm использовал эту информацию для создания мощного механизма музыкальных рекомендаций. Система охватила миллионы пользователей, потому что сторонние приложения и сайты могли передавать данные прослушивания обратно в механизм рекомендаций.

В то время разработки рекомендательных систем в основном сводились к обучению на основе рейтинга сопоставимых данных. Иными словами, рекомендательные системы обычно рассматривались как инструменты, работающие на входных данных, таких, как «Боб оценивает песни Эминема на 3,5 звезды». Набор данных Audioscrobbler интересен тем, что он просто фиксирует факт воспроизведения: «Боб включил песню Эминема». Воспроизведение несет меньше информации, чем рейтинг. Тот факт, что Боб включил определенную песню, не означает, что она действительно ему понравилась. Мы с вами тоже можем иногда включить песню группы, которая нас не интересует, или даже целый альбом, и через минуту выйти из комнаты.

Однако слушатели оценивают музыку гораздо реже, чем воспроизводят. Следовательно, набор данных о воспроизведении музыки намного больше, охватывает больше пользователей и певцов[1] и содержит больше общей информации, чем набор рейтинговых данных, даже если каждая отдельная точка данных несет меньше информации. Этот тип данных часто называют *неявными данными обратной связи*, потому что связи между пользователем и певцом прослеживаются как побочный эффект других действий, а не в виде явных рейтингов или лайков.

Снимок набора данных, распространяемого Last.fm в 2005 г., можно найти в Интернете в виде сжатого архива. Скачайте архив и найдите в нем несколько файлов. Убедитесь, что файлы набора данных доступны. Если вы используете удаленный кластер, скопируйте все три файла данных в хранилище. В этой главе предполагается, что файлы доступны в каталоге data/.

Запустите `pyspark-shell`. Обратите внимание, что вычисления в этой главе займут больше памяти, чем простые приложения. Например, если вы работаете локально, а

[1] Мы будем называть певцов и музыкальные группы общим словом «певец», чтобы избежать путаницы, потому что «исполнитель» является общепринятым переводом термина *executor* в Spark. — *Примеч. пер.*

не в кластере, вам, вероятно, потребуется добавить опцию наподобие `--driver-memory 4g`, чтобы иметь достаточно памяти для этих вычислений. Основной набор данных находится в файле user_artist_data.txt. Он содержит около 141 000 уникальных пользователей и 1,6 млн уникальных певцов. В наборе записано около 24,2 млн пользовательских воспроизведений. Давайте прочитаем этот набор данных в DataFrame и рассмотрим его:

```
raw_user_artist_path = "data/audioscrobbler_data/user_artist_data.txt"
raw_user_artist_data = spark.read.text(raw_user_artist_path)
raw_user_artist_data.show(5)
...

+-------------------+
|              value|
+-------------------+
|       1000002 1 55|
| 1000002 1000006 33|
|  1000002 1000007 8|
|1000002 1000009 144|
|1000002 1000010 314|
+-------------------+
```

Задачи машинного обучения, такие, как ALS, наверняка будут более ресурсоемкими, чем простая обработка текста. Иногда полезно разбить данные для обработки на более мелкие части. Вы можете добавить вызов `.repartition(n)` после чтения текстового файла, чтобы указать другое (обычно большее) количество разделов. Вы можете, например, установить это значение таким, чтобы оно соответствовало количеству ядер в вашем кластере.

В наборе данных также указаны имена каждого певца по идентификатору в файле artist_data.txt. Обратите внимание, что при скробблировании произведений клиентское приложение отправляет имя певца, песня которого воспроизводится. Имя или название группы может быть написано с ошибкой или нестандартно, но выяснится это позже. Например, «The Smiths», «Smiths, The» и «the smiths» могут отображаться в наборе данных как разные идентификаторы певцов, даже если они явно принадлежат одному и тому же объекту данных. Поэтому набор данных также включает в себя файл artist_alias.txt, который сопоставляет идентификаторы певцов, написанных с известными орфографическими ошибками или вариациями, с каноническим идентификатором этого певца. Давайте также прочитаем эти два набора данных в PySpark:

```
raw_artist_data = spark.read.text("data/audioscrobbler_data/artist_data.txt")

raw_artist_data.show(5)

...
+--------------------+
|               value|
```

```
+--------------------+
|1134999\t06Crazy ...|
|6821360\tPang Nak...|
|10113088\tTerfel,...|
|10151459\tThe Fla...|
|6826647\tBodensta...|
+--------------------+
only showing top 5 rows
...

raw_artist_alias = spark.read.text("data/audioscrobbler_data/artist_alias.txt")

raw_artist_alias.show(5)

...
+-----------------+
|            value|
+-----------------+
| 1092764\t1000311|
| 1095122\t1000557|
| 6708070\t1007267|
|10088054\t1042317|
| 1195917\t1042317|
+-----------------+
only showing top 5 rows
```

Теперь, когда у нас есть общее представление о наборах данных, мы можем обсудить наши требования к рекомендательному алгоритму и, следовательно, понять, почему алгоритм чередующихся наименьших квадратов является хорошим выбором.

Наши требования к рекомендательной системе

Нам нужно выбрать рекомендательный алгоритм, который подходит для наших данных. Вот наши соображения:

Неявная обратная связь

Данные полностью состоят из взаимодействий между пользователями и песнями определенных певцов. Они не содержит никакой информации о пользователях или об певцах, кроме их имен. Нам нужен алгоритм, который обучается без доступа к атрибутам пользователя или певца. Обычно их называют *алгоритмами совместной фильтрации*. Например, решение о том, что два пользователя могут иметь схожие вкусы, потому что они одного возраста, не является примером совместной фильтрации. В качестве примера можно рассматривать решение о том, что двум пользователям может нравиться одна и та же песня, потому что они слушают множество других одинаковых песен.

Разреженность

Наш набор данных выглядит большим, потому что он содержит записи о десятках миллионов воспроизведений. Но в другом смысле он маленький и скудный, потому что разреженный. В среднем каждый пользователь воспроизводил песни примерно 171 певца — а ведь певцов 1,6 миллиона! При этом некоторые пользователи слушали только одного певца. Нам же нужен алгоритм, который может дать достойные рекомендации даже этим пользователям. В конце концов, каждый слушатель когда-то начинал свой список воспроизведения с одной песни!

Масштабируемость и прогнозы в реальном времени

Наконец, нам нужен масштабируемый алгоритм как для построения больших моделей, так и для быстрого создания рекомендаций. Рекомендации обычно требуются в режиме реального времени — то есть в течение секунды, а не завтра.

Широкий класс алгоритмов, которые могут быть подходящими, — это *модели скрытых факторов* (latent factor models). Они пытаются объяснить наблюдаемые взаимодействия между большим количеством пользователей и элементов относительно небольшим количеством неочевидных скрытых причин. Например, возьмем клиента, который купил альбомы рок-групп Megadeth и Pantera, а также классического композитора Моцарта. На первый взгляд, трудно объяснить, почему клиент выбрал такое странное сочетание альбомов. Вероятнее всего, это просто небольшой фрагмент в гораздо более широком наборе вкусов. Возможно, клиенту нравится гармоничный спектр музыки от металла до прогрессивного рока и классики. Это объяснение проще и, в качестве бонуса, позволяет предложить множество других альбомов, которые могли бы представлять интерес. В этом примере «любовь к металлу, прогрессивному року и классике» — это три скрытых фактора, которые могут объяснить десятки тысяч индивидуальных предпочтений в отношении альбомов.

В нашем случае мы специально будем использовать модель разложения матрицы на множители (матричной факторизации). Математически эти алгоритмы обрабатывают данные пользователя и певца, как если бы это была большая матрица A, где запись в строке i и столбце j существует, если пользователь i воспроизвел певца j. Матрица A является разреженной: большинство записей A равны 0, потому что лишь несколько из всех возможных комбинаций пользователя и певца фактически появляются в данных. Алгоритм разлагает A как матричное произведение двух меньших матриц, X и Y. Они очень узкие — у обеих много строк, потому что у A много строк и столбцов, но у обеих всего несколько столбцов (k). Количество столбцов k соответствует скрытым факторам, которые используются для объяснения известных взаимодействий.

Разложение может быть только приблизительным, поскольку k мало, как показано на рис. 3.1.

Эти алгоритмы иногда называют *алгоритмами дополнения матриц*, потому что исходная матрица A может быть очень разреженной, но произведение XY^T плотное. Очень немногие записи равны 0, если такие вообще есть, и, следовательно, модель является лишь приближением к A. Этот алгоритм является моделью в том смысле,

что он моделирует («дополняет») значение даже для многих отсутствующих записей (то есть 0) в оригинале A.

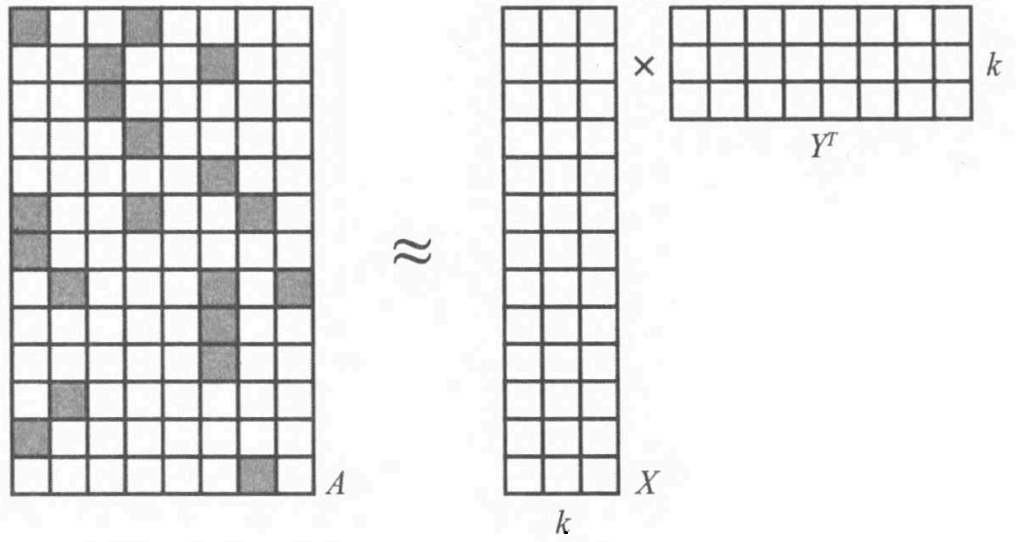


Рис. 3.1. Матричная факторизация

Это тот случай, когда, к счастью, линейная алгебра напрямую и элегантно превращается в интуитивно понятный результат. Две матрицы содержат строку для каждого пользователя и каждого певца. В строках мало значений — всего k. Каждое значение соответствует скрытому признаку в модели. Таким образом, строки показывают, насколько пользователи и певцы ассоциируются с этими k скрытыми признаками, которые могут соответствовать вкусам или жанрам. И это просто произведение матрицы «пользователь-признак» и «певец-певец», которая дает полную оценку всей плотной матрицы взаимодействия пользователя и певца. Этот продукт можно рассматривать как сопоставление элементов с их атрибутами, а затем их взвешивание по пользовательским атрибутам.

Плохая новость заключается в том, что $A = XY^T$ вообще не имеет точного решения, потому что X и Y недостаточно велики (иначе говоря, у них слишком низкий ранг), чтобы идеально представить A. Но, если вдуматься, на самом деле это хорошо. Матрица A — это всего лишь крошечная выборка всех взаимодействий, которые в принципе могут произойти. В каком-то смысле мы считаем, что A — это ужасно неоднородное и, следовательно, труднообъяснимое представление о более простой фундаментальной реальности, которая хорошо объясняется лишь неким небольшим числом факторов, k из них. Представьте большой пазл с изображением кошки. Собранный пазл можно легко описать одним словом: *кошка*. Однако, когда перед вами лежат всего несколько деталей пазла, картину, которую вы видите, довольно сложно описать.

Результат XY^T все равно должен быть как можно ближе к A. В конце концов, это все, что нам нужно. Он не будет и не должен точно воспроизводить матрицу A. Плохая новость заключается в том, что это решение нельзя найти сразу для лучшего X и лучшего Y одновременно. Хорошая новость же состоит в том, что найти наи-

лучшее значение X, если известно Y, и наоборот, несложно. Но ни то, ни другое значение не известно заранее!

К счастью, есть алгоритмы, которые могут избежать этой ловушки и найти достойное решение. Одним из таких алгоритмов, доступных в PySpark, является алгоритм ALS.

Альтернативный алгоритм наименьших квадратов

Мы будем использовать алгоритм чередующихся наименьших квадратов для вычисления скрытых факторов из нашего набора данных. Этот подход был обнародован во время конкурса Netflix Prize в таких статьях, как «Совместная фильтрация наборов данных с неявной обратной связью» и «Крупномасштабная параллельная совместная фильтрация для Netflix Prize». Реализация ALS в PySpark MLlib опирается на идеи из обеих этих статей и является единственным рекомендательным алгоритмом, реализованным в настоящее время в Spark MLlib.

Вот фрагмент кода (нефункциональный), чтобы дать вам представление о том, что будет дальше описано в этой главе:

```
from pyspark.ml.recommendation import ALS

als = ALS(maxIter=5, regParam=0.01, userCol="user",
          itemCol="artist", ratingCol="count")
model = als.fit(train)

predictions = model.transform(test)
```

С помощью ALS мы будем обрабатывать наши входные данные как большую разреженную матрицу A и найдем X и Y, как обсуждалось в предыдущем разделе. Поначалу множитель Y неизвестен, но его можно инициализировать матрицей, полной случайно выбранных векторов-строк. Тогда простые алгебраические вычисления дают наилучшее решение для X при заданных A и Y. На самом деле, несложно вычислить каждую строку i матрицы X отдельно как функцию Y и одной строки матрицы A. Поскольку эти операции выполняются независимо, можно организовать параллельные вычисления, и это отличное свойство алгоритма для крупномасштабных вычислений:

$$A_i Y(Y^T Y)^{-1} = X_i$$

Точное равенство в данном случае недостижимо, поэтому цель состоит в том, чтобы минимизировать $|A_i Y(Y^T Y)^{-1} - X_i|$ или сумму квадратов разностей между элементами двух матриц. Отсюда и «наименьшие квадраты» в названии алгоритма. На практике это уравнение никогда не решается путем фактического вычисления инверсий. Решение получают быстрее и проще с помощью таких методов, как QR-разложение. Это уравнение просто развивает теорию вычисления вектора-строки.

То же самое можно сделать для вычисления каждого Y_j из X. И снова для вычисления X из Y и так далее. Вот откуда берется «чередование» в названии алгоритма. Есть только одна небольшая проблема: матрица Y была выдумана — причем слу-

чайным образом! Да, матрица X вычисляется оптимально, но дает фиктивное решение для Y. К счастью, если этот процесс повторяется, X и Y в конечном итоге сходятся к достойным решениям.

При использовании для разложения матрицы, представляющей неявные данные, разложение ALS несколько усложняется. Это факторизация не входной матрицы A напрямую, а матрицы P, состоящей из 0 и 1. Она содержит 1 там, где A содержит положительное значение, и 0 в остальных местах. Соответствующие значения A включают позже в качестве весов. Этот прием выходит за рамки данной книги и не является необходимым для понимания того, как использовать данный алгоритм.

Наконец, алгоритм ALS может использовать разреженность входных данных. Это, а также его опора на простую оптимизированную линейную алгебру и параллельный характер вычислений делают его очень быстрым при работе с большими данными.

Далее мы выполним предварительную обработку нашего набора данных и сделаем его пригодным для использования с алгоритмом ALS.

Подготовка данных

Первым шагом в построении модели является изучение доступных данных и их анализ или преобразование в формы, полезные для анализа в Spark.

Реализация ALS в Spark MLlib не требует строго числовых идентификаторов для пользователей и объектов, но более эффективна, когда идентификаторы представлены в виде 32-разрядных целых чисел, потому что фактически данные представляются с использованием типа данных JVM. Давайте проверим, соответствует ли наш набор данных этому требованию?

```
raw_user_artist_data.show(10)
...
+-------------------+
|              value|
+-------------------+
|       1000002 1 55|
| 1000002 1000006 33|
|  1000002 1000007 8|
|1000002 1000009 144|
|1000002 1000010 314|
|  1000002 1000013 8|
| 1000002 1000014 42|
| 1000002 1000017 69|
|1000002 1000024 329|
|  1000002 1000025 1|
+-------------------+
```

Каждая строка файла содержит идентификатор пользователя, идентификатор певца и количество воспроизведений, разделенные пробелами. Чтобы вычислить стати-

стику по идентификатору пользователя, мы разбиваем строку по пробелам и анализируем значения как целые числа. Результат представляет собой три «столбца»: идентификатор пользователя, идентификатор певца и количество в виде чисел типа `int`. Имеет смысл преобразовать его в фрейм данных со столбцами `user` (пользователь), `artist` (певец в нашем случае) и `count` (количество), потому что тогда становится легче вычислять простые статистические данные, такие, как максимум и минимум:

```
from pyspark.sql.functions import split, min, max
from pyspark.sql.types import IntegerType, StringType

user_artist_df = raw_user_artist_data.withColumn('user',
                 split(raw_user_artist_data['value'], ' ').\
                 getItem(0).\
                 cast(IntegerType()))
user_artist_df = user_artist_df.withColumn('artist',
                 split(raw_user_artist_data['value'], ' ').\
                 getItem(1).\
                 cast(IntegerType()))
user_artist_df = user_artist_df.withColumn('count',
                 split(raw_user_artist_data['value'], ' ').\
                 getItem(2).\
                 cast(IntegerType())).drop('value')

user_artist_df.select([min("user"), max("user"), min("artist"),\
                                    max("artist")]).show()
...
+---------+---------+-----------+-----------+
|min(user)|max(user)|min(artist)|max(artist)|
+---------+---------+-----------+-----------+
|       90|  2443548|          1|   10794401|
+---------+---------+-----------+-----------+
```

Максимальные идентификаторы пользователя и певца — 2443548 и 10794401 соответственно (минимальные — 90 и 1; отрицательных значений нет). Очевидно, что они меньше и соответственно удобнее, чем, например, число 2147483647. И следовательно, для использования этих идентификаторов не требуется никаких дополнительных преобразований.

Позже в этом примере нам пригодится знание настоящих имен певцов, соответствующих неочевидным числовым идентификаторам. Фрейм `raw_artist_data` содержит идентификатор и имя певца, разделенные табуляцией. Функция `split` PySpark принимает значения регулярного выражения в качестве параметра `pattern`. Мы можем разделить значения, используя символ пробела `\s`:

```
from pyspark.sql.functions import col

artist_by_id = raw_artist_data.withColumn('id', split(col('value'),
                                                '\s+', 2).\
```

```
                                        getItem(0).\
                                        cast(IntegerType()))
artist_by_id = artist_by_id.withColumn('name', split(col('value'),
                                       '\s+', 2).\
                                       getItem(1).\
                                       cast(StringType())).drop('value')

artist_by_id.show(5)
...
+--------+--------------------+
|      id|                name|
+--------+--------------------+
| 1134999|        06Crazy Life|
| 6821360|        Pang Nakarin|
|10113088|Terfel, Bartoli- ...|
|10151459| The Flaming Sidebur|
| 6826647|   Bodenstandig 3000|
+--------+--------------------+
```

У нас получается фрейм данных с идентификаторами и именами певцов в виде столбцов `id` и `name`.

Фрейм `raw_artist_alias` сопоставляет идентификаторы певцов, которые могут быть написаны с ошибками или нестандартно, с идентификатором канонического имени певца. Этот набор данных относительно небольшой и содержит около 200 000 записей. Он содержит по два идентификатора в строке, разделенные табуляцией. Мы проанализируем их так же, как и `raw_artist_data`:

```
artist_alias = raw_artist_alias.withColumn('artist',
                                split(col('value'), '\s+').\
                                      getItem(0).\
                                      cast(IntegerType())).\
                       withColumn('alias',
                                  split(col('value'), '\s+').\
                                  getItem(1).\
                                  cast(StringType())).\
                                  drop('value')

artist_alias.show(5)
...
+--------+-------+
|  artist|  alias|
+--------+-------+
| 1092764|1000311|
| 1095122|1000557|
| 6708070|1007267|
|10088054|1042317|
| 1195917|1042317|
+--------+-------+
```

Первая запись сопоставляет ID 1092764 с 1000311. Мы можем найти их в фрейме artist_by_id:

```
artist_by_id.filter(artist_by_id.id.isin(1092764, 1000311)).show()
...
+-------+--------------+
|     id|          name|
+-------+--------------+
|1000311| Steve Winwood|
|1092764|Winwood, Steve|
+-------+--------------+
```

Эта запись, очевидно, сопоставляет «Уинвуд, Стив» со «Стив Уинвуд», где второй вариант является правильным именем певца.

Создание первой модели

Хотя набор данных в принципе подходит для использования с реализацией ALS Spark MLlib, все же он нуждается в небольшом преобразовании. Если существует другой канонический идентификатор, то набор данных псевдонимов следует применять для преобразования всех идентификаторов певцов в канонический идентификатор:

```
from pyspark.sql.functions import broadcast, when

train_data = train_data = user_artist_df.join(broadcast(artist_alias),
                                              'artist', how='left').\
train_data = train_data.withColumn('artist',
                           when(col('alias').isNull(), col('artist')).\
                           otherwise(col('alias'))) ❶
train_data = train_data.withColumn('artist', col('artist').\
                                             cast(IntegerType())).\
                                             drop('alias')
train_data.cache()

train_data.count()
...
24296858
```

❶ Получить псевдоним певца, если он существует; в противном случае получить оригинальное имя певца.

Мы применяем функцию `broadcast` к созданному ранее фрейму данных `artist_alias`. Это заставляет Spark отправлять и хранить в памяти только одну копию для каждого *исполнителя* (executor) в кластере. Когда есть тысячи задач и многие из них выполняются параллельно, это может значительно сэкономить сетевой

трафик и память. Как показывает опыт, при выполнении слияния с очень большим набором данных разумнее передавать по сети меньший из наборов данных.

Широковещательные переменные

Когда Spark запускает *этап* (stage), он создает двоичное представление всей информации, необходимой для выполнения задач на этом этапе; это представление называется *замыканием* (closure) функции, которую необходимо выполнить. Это замыкание включает в себя все структуры данных в драйвере, на который ссылается функция. Spark распределяет замыкание с каждой задачей, отправляемой исполнителю в кластере.

Широковещательные переменные (broadcast variable) полезны, когда многим задачам требуется доступ к одной и той же (неизменяемой) структуре данных. Они расширяют обычную обработку замыкания задач, позволяя:

- кеширование данных в виде необработанных объектов Java на каждом исполнителе, поэтому их не нужно десериализовать для каждой задачи;
- кеширование данных по нескольким заданиям, этапам и задачам.

Например, рассмотрим приложение для обработки естественного языка, которому требуется большой словарь английских слов и функция `score`, которая принимает строку ввода и словарь. Широковещание словаря означает, что он передается каждому исполнителю только один раз.

Операции DataFrame также могут иногда автоматически использовать широковещательные рассылки при выполнении слияний между большой и маленькой таблицей. Просто передавать маленькую таблицу иногда выгоднее. Этот механизм называется *широковещательным слиянием хешей* (broadcast hash join).

Вызов `cache` предлагает Spark временно сохранить этот фрейм после вычисления и, кроме того, сохранить именно в памяти кластера. Это полезно, потому что алгоритм ALS является итеративным и обычно требуется доступ к этим данным 10 или более раз. Без этого фрейм пришлось бы многократно пересчитывать из исходных данных каждый раз, когда к нему обращаются! Вкладка **Storage** (Хранилище) в пользовательском интерфейсе Spark покажет, какая часть объекта DataFrame кешируется и сколько памяти она использует (рис. 3.2). В данном случае фрейм занимает около 120 МБ в кластере.

Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
Memory Deserialized 1x Replicated	8	100%	120.3 MB	0.0 B

Рис. 3.2. Вкладка **Storage** в пользовательском интерфейсе Spark, показывающая использование памяти кешированным DataFrame

Когда вы используете `cash` или `persist`, DataFrame не кешируется полностью, пока вы не активируете действие, которое проходит через каждую запись (например, `count`). Если вы используете такое действие, как `show(1)`, кешируется только один раздел. Так происходит, потому что оптимизатор PySpark поймет, что вам не нужно вычислять все разделы для получения только одной записи.

Обратите внимание, что метка **Deserialized** (Десериализованные) в пользовательском интерфейсе на рис. 3.2 фактически относится только к RDD, где **Serialized** (Сериализованные) означают, что данные хранятся в памяти не как объекты, а как сериализованные байты. Однако экземпляры DataFrame, подобные этому, выполняют собственное «кодирование» общих типов данных в памяти отдельно.

На самом деле, 120 МБ — это на удивление мало. Если учесть, что здесь хранится около 24 млн воспроизведений, быстрый предварительный подсчет показывает, что каждая запись «пользователь-певец» занимает в среднем всего 5 байтов. Однако только три 32-битных целых числа должны занимать 12 байтов! Здесь как раз и проявляется одно из преимуществ DataFrame. Поскольку типы хранимых данных представляют собой банальные 32-разрядные целые числа, их представление можно оптимизировать внутри памяти.

Интерфейс Spark

Пользовательский интерфейс Spark — это веб-интерфейс работающего приложения Spark, который отслеживает состояние и потребление ресурсов вашего кластера Spark (рис. 3.3). По умолчанию он доступен по адресу http://*<драйвер>*:4040.

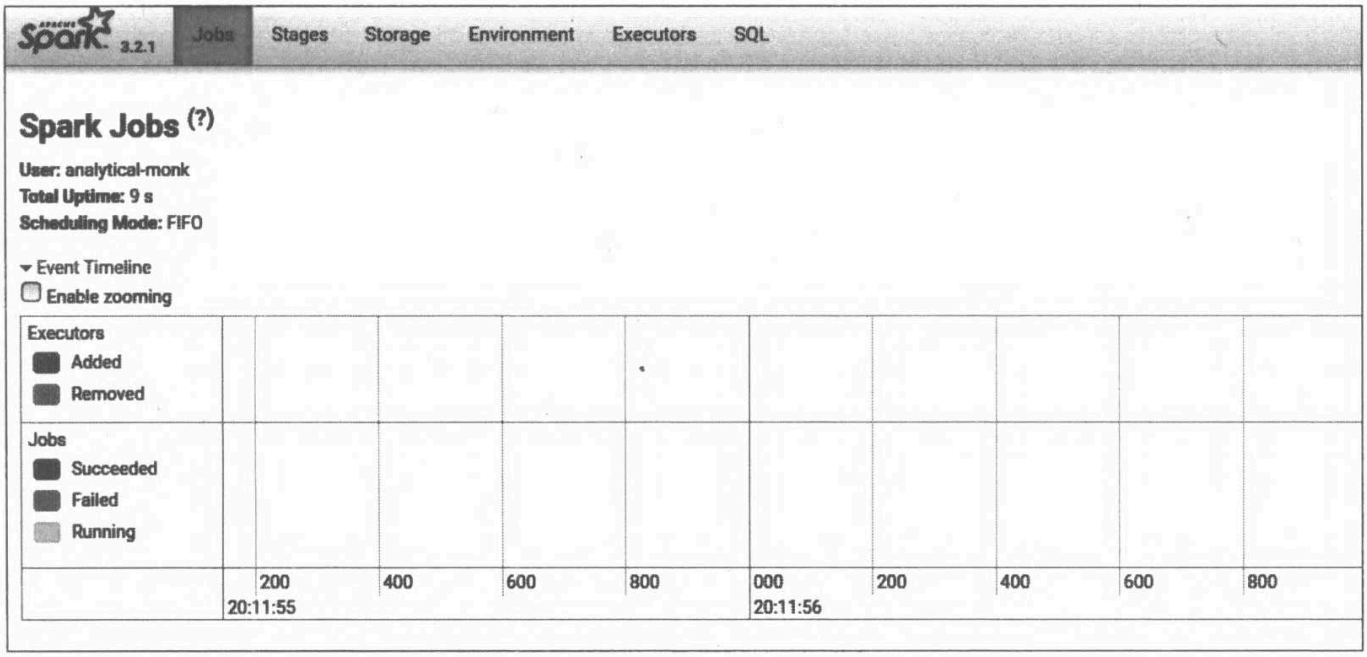


Рис. 3.3. Интерфейс Spark

Пользовательский интерфейс Spark содержит следующие вкладки:

- **Jobs** (Задания);

- **Stages** (Этапы);
- **Storage** (Хранилище) с размером DataFrame и занятой памяти;
- **Environment** (Окружение);
- **Executors** (Исполнители);
- **SQL.**

Некоторые вкладки, например, связанные с потоковой передачей, создаются по мере необходимости.

Наконец, мы можем построить модель:

```
from pyspark.ml.recommendation import ALS

model = ALS(rank=10, seed=0, maxIter=5, regParam=0.1,
            implicitPrefs=True, alpha=1.0, userCol='user',
            itemCol='artist', ratingCol='count'). \
        fit(train_data)
```

Этот код создает `model` как `ALSModel` с конфигурацией по умолчанию. Операция, скорее всего, займет несколько минут или больше в зависимости от вашего кластера. По сравнению с некоторыми моделями машинного обучения, которые в окончательном виде могут состоять всего из нескольких параметров или коэффициентов, модель этого типа огромна. Она содержит вектор признаков из 10 значений для каждого пользователя и продукта в модели, а в данном случае их более 1,7 млн. Модель содержит эти большие матрицы характеристик пользователя и характеристик продукта как собственные фреймы данных.

Ваши конкретные значения могут несколько отличаться от показанных в книге. Окончательная модель зависит от случайно выбранного начального набора векторов признаков. Однако нужно иметь в виду, что по умолчанию MLlib использует один и тот же набор случайных выборок, опираясь на фиксированное начальное число для генерации псевдослучайной последовательности. Этим MLlib отличается от других библиотек, где поведение случайных элементов обычно не фиксируется по умолчанию. Поэтому здесь и в других примерах кода случайное начальное число устанавливается с помощью `(... seed=0,...)`.

Чтобы увидеть некоторые векторы признаков, попробуйте следующий код, который выводит только одну строку и не обрезает широкое отображение вектора признаков:

```
model.userFactors.show(1, truncate = False)

...
+---+------------------------------------------------ ...
|id |features ...
+---+------------------------------------------------ ...
|90 |[0.16020626, 0.20717518, -0.1719469, 0.06038466 ...
+---+------------------------------------------------ ...
```

Другие методы, вызываемые в `ALS`, такие, как `setAlpha`, устанавливают *гиперпараметры*, значения которых могут влиять на качество рекомендаций модели. О них мы поговорим позже. Более важный первый вопрос заключается в следующем: насколько хороша модель, т. е. дает ли она хорошие рекомендации? Именно на этот вопрос мы постараемся ответить в следующем разделе.

Выборочная проверка рекомендаций

Сначала посмотрим, выглядят ли разумными рекомендации какого-либо певца, изучив пользователя, произведения и рекомендации для этого пользователя. Возьмем, к примеру, пользователя 2093760. Сначала рассмотрим его список воспроизведения, чтобы понять вкусы этого человека. Извлеките идентификаторы певцов, которых слушал этот пользователь, и напечатайте их имена. Это означает поиск входных данных для идентификаторов певцов, чьи песни слушал этот пользователь, а затем фильтрацию полученного списка певцов по этим идентификаторам для печати имен по порядку:

```
user_id = 2093760

existing_artist_ids = train_data.filter(train_data.user == user_id) \ ❶
  .select("artist").collect() ❷

existing_artist_ids = [i[0] for i in existing_artist_ids]

artist_by_id.filter(col('id').isin(existing_artist_ids)).show() ❸
...
+-------+---------------+
|     id|           name|
+-------+---------------+
|   1180|     David Gray|
|    378|  Blackalicious|
|    813|     Jurassic 5|
|1255340|The Saw Doctors|
|    942|         Xzibit|
+-------+---------------+
```

❶ Находим строки, в которых указан пользователь 2093760.

❷ Собираем набор данных по идентификатору певца.

❸ Отфильтровываем этих певцов.

Музыкальные стили выглядят как смесь поп-музыки и хип-хопа. Фанат Jurassic 5, серьезно? Это же группа 2005 года. Кстати, если вам интересно, Saw Doctors — очень популярная в Ирландии рок-группа.

Теперь для нас не составит труда выдать пользователю рекомендации, хотя вычисление их таким образом займет несколько минут. Этот метод подходит для поверхностной оценки, но не для использования в реальном времени:

```
user_subset = train_data.select('user').where(col('user') ==
                                              user_id).distinct()
top_predictions = model.recommendForUserSubset(user_subset, 5)

top_predictions.show()
...
+-------+--------------------+
|   user|     recommendations|
+-------+--------------------+
|2093760|[{2814, 0.0294106...|
+-------+--------------------+
```

Полученные рекомендации содержат списки, состоящие из идентификатора певца и, конечно же, «прогнозов». Для этого типа алгоритма ALS прогноз представляет собой непрозрачное значение, обычно от 0 до 1, где более высокие значения означают лучшую рекомендацию. Это не вероятность, но ее можно рассматривать как оценку того, насколько охотно, по мнению модели, пользователь будет взаимодействовать с определенным певцом.

После извлечения идентификаторов рекомендованных певцов мы можем найти их имена:

```
top_predictions_pandas = top_predictions.toPandas()
print(top_prediction_pandas)
...

     user                                     recommendations
0 2093760 [(2814, 0.029410675168037415), (1300642, 0.028...
...

recommended_artist_ids = [i[0] for i in top_predictions_pandas.\
                                        recommendations[0]]

artist_by_id.filter(col('id').isin(recommended_artist_ids)).show()
...
+-------+----------+
|     id|      name|
+-------+----------+
|   2814|   50 Cent|
|   4605|Snoop Dogg|
|1007614|     Jay-Z|
|1001819|      2Pac|
|1300642|  The Game|
+-------+----------+
```

В рекомендациях сплошной хип-хоп. На первый взгляд, это не очень хороший набор рекомендаций. Хотя это популярные исполнители, похоже, они не очень-то соответствуют вкусам нашего пользователя.

Оценка качества рекомендаций

Конечно, это всего лишь одно субъективное суждение о рекомендациях для одного пользователя. Кому-то, кроме этого пользователя, трудно количественно оценить, насколько хороши рекомендации. Более того, невозможно, чтобы какой-либо человек вручную оценивал даже небольшую выборку выходных данных для оценки результатов.

Разумно предположить, что пользователи, как правило, слушают песни привлекательных для них певцов и игнорируют непривлекательных. Итак, информация о списке воспроизведения пользователя дает частичное представление о «хороших» и «плохих» рекомендациях. Это проблематичное предположение, но это лучшее, что можно сделать без каких-либо других данных. Например, предположительно пользователю 2093760 нравится гораздо больше певцов, чем 5 перечисленных ранее, а из 1,7 миллиона других непрослушанных певцов некоторые представляют интерес, и не все они являются «плохими» рекомендациями.

Что если бы рекомендательную систему оценивали по ее способности ставить хороших певцов на первое место в списке рекомендаций? Это одна из нескольких общих метрик, которые можно применять к системе, ранжирующей вещи, например, к рекомендациям. Проблема в том, что понятие «хорошо» определяется как «певцы, которых слушал пользователь», а рекомендательная система уже получила всю эту информацию в качестве входных данных. Она может тривиально возвращать исполнителей, которых ранее слушал пользователь, в качестве лучших рекомендаций и получать отличные оценки. Но это бесполезный результат, тем более что задача рекомендательной системы — предлагать певцов, которых пользователь никогда не слушал.

Чтобы сделать результаты более значимыми, некоторые данные о прослушивании певца можно отложить и скрыть от процесса построения модели ALS. Затем эти отложенные данные можно интерпретировать как набор хороших рекомендаций для каждого пользователя, но тех, которые рекомендательная система еще не дала. Ее просят ранжировать все элементы в модели и проверяют ранги оставшихся певцов. В идеале рекомендательная система помещает их всех в начало списка или рядом с ним.

Затем мы можем вычислить оценку рекомендательной системы, сравнив рейтинги всех прошедших отбор певцов с остальными. (На практике мы вычисляем ее, изучая только ограниченную выборку всех таких пар, потому что потенциально их может существовать огромное количество.) Оценкой является доля пар, в которых прошедший отбор певец имеет более высокий рейтинг. Оценка 1,0 — идеально, 0,0

— это наихудшая возможная оценка, а 0,5 — это ожидаемое значение, полученное от случайного ранжирования исполнителей.

Этот показатель напрямую связан с концепцией поиска информации, называемой *характеристической кривой обнаружения*[2] (receiver operating characteristic, ROC). Показатель, упомянутый в предыдущем абзаце, равен площади под этой кривой ROC и широко известен под аббревиатурой AUC (area under curve, площадь под кривой). AUC можно рассматривать как вероятность того, что случайно выбранная хорошая рекомендация окажется выше случайно выбранной плохой рекомендации.

Показатель AUC также используется при оценке классификаторов. Он реализован вместе со связанными методами в классе MLlib `BinaryClassificationMetrics`. Для нашей рекомендательной системы мы рассчитаем AUC *на пользователя* и усредним результат. Полученное значение немного отличается и может быть условно названо «средняя AUC». Мы реализуем вычисление этого показателя самостоятельно, потому что он совсем не реализован в PySpark.

Другие оценочные показатели, относящиеся к системам ранжирования, реализованы в `RankingMetrics`. К ним относятся такие показатели, как точность, полнота и *именованная средняя точность* (mean average precision, MAP). MAP также часто используется на практике и более узко фокусируется на качестве лучших рекомендаций. Однако здесь мы будем использовать AUC как общую и широкую меру качества всего вывода модели.

На самом деле процесс вычленения некоторых данных для выбора модели и оценки ее точности является обычной практикой в машинном обучении. Как правило, данные делятся на три подмножества: наборы данных для обучения, для *перекрестной проверки* (cross-validation, CV) и тестовые наборы. Для простоты в нашем примере мы будем использовать только два набора: обучение и CV. Этого будет достаточно для выбора модели. В *главе 4* этот подход будет расширен за счет добавления тестового набора.

Вычисление AUC

Реализация среднего AUC представлена в исходном коде, прилагаемом к этой книге. Здесь он не приводится, но подробно объяснен в комментариях к коду. Он принимает набор CV как «подходящих» певцов для каждого пользователя и функцию прогнозирования. Эта функция переводит фрейм данных, содержащий каждую пару «пользователь-певец», в другой фрейм данных, который также содержит предполагаемую силу взаимосвязи в качестве «прогноза» — это число, где более высокие значения означают более высокий рейтинг в рекомендациях.

Чтобы воспользоваться входными данными, мы должны разделить их на обучающий набор и набор CV. Модель ALS будет обучаться только на обучающем наборе

[2] Ее также называют *рабочей характеристикой приемника*, так как термин позаимствован из области радарного обнаружения целей. — *Примеч. пер.*

данных, а набор CV мы применим для оценки модели. В нашем случае 90% данных используются для обучения, а остальные 10% — для перекрестной проверки:

```
def area_under_curve(
    positive_data,
    b_all_artist_IDs,
predict_function):
...

all_data = user_artist_df.join(broadcast(artist_alias), 'artist', how='left') \
    .withColumn('artist', when(col('alias').isNull(), col('artist'))\
    .otherwise(col('alias'))) \
    .withColumn('artist',
                col('artist').cast(IntegerType())).drop('alias')

train_data, cv_data = all_data.randomSplit([0.9, 0.1], seed=54321)
train_data.cache()
cv_data.cache()

all_artist_ids = all_data.select("artist").distinct().count()
b_all_artist_ids = broadcast(all_artist_ids)

model = ALS(rank=10, seed=0, maxIter=5, regParam=0.1,
            implicitPrefs=True, alpha=1.0, userCol='user',
            itemCol='artist', ratingCol='count') \
        .fit(train_data)
area_under_curve(cv_data, b_all_artist_ids, model.transform)
```

Обратите внимание, что `areaUnderCurve` принимает функцию в качестве третьего аргумента. Здесь передается метод `transform` из `ALSModel`, но вскоре он будет заменен на альтернативный.

Результат около 0,879. Насколько он хорош? Он, безусловно, выше, чем 0,5 (чисто случайные рекомендации), и близок к 1,0, что является максимально возможным показателем. Как правило, значение AUC более 0,9 считается высоким.

Но тогда возникает другой вопрос: а насколько точна оценка? Процедуру оценки можно повторить с другими 90% в качестве обучающей выборки. Полученное среднее значение AUC может быть более точной оценкой точности алгоритма в наборе данных. Один из распространенных приемов состоит в том, чтобы разделить данные на k подмножеств одинакового размера, использовать k–1 подмножеств для обучения и оценить модель на оставшемся подмножестве. Мы можем повторить эти действия k раз, используя каждый раз другой набор подмножеств. Это называется k-кратной перекрестной проверкой. Для простоты мы не будем выполнять ее в наших примерах, но определенная поддержка этой методики существует в MLlib в ее API `CrossValidator`. API проверки будет повторно рассмотрен в *разд. «Случайные леса» главы 4*.

Полезно сравнить текущий алгоритм с более простым подходом. Например, вы можете порекомендовать каждому пользователю самых популярных исполнителей

в мире. Такие рекомендации не персонализированы, но просты и иногда бывают на удивление эффективны. Определим эту простую функцию прогнозирования и оценим ее показатель AUC:

```
from pyspark.sql.functions import sum as _sum

def predict_most_listened(train):
    listen_counts = train.groupBy("artist")\
                    .agg(_sum("count").alias("prediction"))\
                    .select("artist", "prediction")

return all_data.join(listen_counts, "artist", "left_outer").\
                select("user", "artist", "prediction")

area_under_curve(cv_data, b_all_artist_ids,
                 predict_most_listened(train_data))
```

Результат также составляет около 0,880. Это говорит о том, что даже неперсонализированные рекомендации достаточно эффективны. Однако мы ожидали, что персонализированные рекомендации будут иметь заметно более высокие оценки! Очевидно, наша модель персонализованных рекомендаций нуждается в доработке. Можно ли сделать ее лучше?

Выбор гиперпараметра

До сих пор значения гиперпараметров, используемых для построения `ALSModel`, просто приводились без комментариев. Они не изучаются алгоритмом и должны быть заданы вызывающей функцией. Настроенные гиперпараметры таковы:

`setRank(10)`

Количество скрытых факторов в модели или, что эквивалентно, количество столбцов k в матрицах. В нетривиальных случаях это также их ранг.

`setMaxIter(5)`

Количество итераций, выполняемых процедурой факторизации. Дополнительные итерации занимают больше времени, но могут привести к лучшей факторизации.

`setRegParam(0.01)`

Стандартный параметр переобучения, также обычно называемый *лямбдой*. Более высокие значения препятствуют переобучению, но слишком высокие значения снижают точность факторизации.

`setAlpha(1.0)`

Определяет относительный вес наблюдаемых и ненаблюдаемых взаимодействий пользователь–продукт при факторизации.

Переменные `rank`, `regParam` и `alpha` можно рассматривать как гиперпараметры модели. (Здесь `maxIter` — это скорее ограничение на ресурсы, используемые при факторизации.) Это не значения, которые появляются в матрицах внутри ALSModel и

представляют собой параметры, выбранные алгоритмом. В отличие от них, гиперпараметры являются параметрами самого процесса построения модели.

Значения, использованные в предыдущем листинге, не обязательно оптимальны. Выбор хороших значений гиперпараметров — распространенная проблема в машинном обучении. Самый простой способ выбрать значения — просто попробовать комбинации значений и оценить модель по определенному критерию для каждого из них, а затем выбрать комбинацию, которая дает наилучшее значение оценки.

В следующем примере мы проверяем восемь возможных комбинаций: `rank = 5` или `30`, `regParam = 4,0` или `0,0001` и `alpha = 1,0` или `40,0`. Эти значения фактически являются предположениями, но они выбраны так, чтобы охватить широкий диапазон значений параметров. Результаты печатаются в порядке убывания значения AUC:

```
from pprint import pprint
from itertools import product

ranks = [5, 30]
reg_params = [4.0, 0.0001]
alphas = [1.0, 40.0]
hyperparam_combinations = list(product(*[ranks, reg_params, alphas]))

evaluations = []

for c in hyperparam_combinations:
    rank = c[0]
    reg_param = c[1]
    alpha = c[2]
    model = ALS().setSeed(0).setImplicitPrefs(true).setRank(rank).\
                setRegParam(reg_param).setAlpha(alpha).setMaxIter(20).\
                setUserCol("user").setItemCol("artist").\
                setRatingCol("count").setPredictionCol("prediction").\
        fit(trainData)

    auc = area_under_curve(cv_aata, b_all_artist_ids, model.transform)

    model.userFactors.unpersist()❶
    model.itemFactors.unpersist()

    evaluations.append((auc, (rank, regParam, alpha)))

evaluations.sort(key=lambda x:x[0], reverse=True)❷
pprint(evaluations)
...
(0.8928367485129145,(30,4.0,40.0))
(0.891835487024326,(30,1.0E-4,40.0))
(0.8912376926662007,(30,4.0,1.0))
(0.889240668173946,(5,4.0,40.0))
(0.8886268430389741,(5,4.0,1.0))
```

```
(0.8883278461068959,(5,1.0E-4,40.0))
(0.8825350012228627,(5,1.0E-4,1.0))
(0.8770527940660278,(30,1.0E-4,1.0))
```

❶ Немедленное освобождение ресурсов модели.

❷ Сортировка значений AUC по убыванию и печать.

Для значений AUC различия оценок невелики в абсолютном выражении, но все же достаточно значительны. Интересно, что параметр `alpha = 40` всегда дает результат лучше, чем `1`. (Для любопытных читателей заметим, что 40 было значением, предложенным по умолчанию в одном из оригинальных ALS в упомянутых ранее публикациях.) Это значение гиперпараметра можно интерпретировать как указание на то, что модели лучше сосредоточиться на том, что пользователь действительно слушал, чем на произведениях, которые он не слушал.

Более высокий `regParam` тоже выглядит лучше. Это говорит о том, что модель в некоторой степени подвержена переобучению, и поэтому ей требуется более высокое значение `regParam`, чтобы противостоять попыткам слишком точной подгонки под разреженный ввод, полученный от каждого пользователя. Переобучение будет рассмотрено более подробно в *разд. «Случайные леса» главы 4*.

Как и ожидалось, 5 признаков довольно мало для модели такого размера, и она уступает модели, которая использует 30 признаков для объяснения вкусов пользователей. Вполне возможно, что наилучшее количество признаков на самом деле превышает 30 и что эти значения схожи в том, что они слишком малы.

Конечно, этот процесс можно повторить для разных диапазонов значений или большего количества значений. Это грубое средство выбора гиперпараметров. Однако в мире, где кластеры с терабайтами памяти и сотнями процессорных ядер не редкость и с такими фреймворками, как Spark, которые могут использовать параллелизм и память для повышения скорости, это становится вполне приемлемым способом.

Вовсе не обязательно точно понимать, что означает каждый гиперпараметр, хотя полезно знать, каковы нормальные диапазоны значений, чтобы начать поиск в пространстве параметров, которые не являются ни слишком большими, ни слишком маленькими.

Сейчас мы рассмотрели ручной способ перебора гиперпараметров, построения моделей и их оценки. В *главе 4*, узнав больше об API Spark ML, вы обнаружите, что существует более автоматизированный способ вычислений с использованием `Pipelines` и `TrainValidationSplit`.

Выдача рекомендаций

Что новая модель рекомендует пользователю 2093760 с учетом найденного нами наилучшего набора гиперпараметров?

```
+-----------+
|       name|
+-----------+
```

```
|  [unknown]|
|The Beatles|
|     Eminem|
|         U2|
|  Green Day|
+-----------+
```

Как ни странно, для данного пользователя получилась более осмысленная рекомендация, поскольку в ней преобладает поп-рок, а не сплошной хип-хоп. Объект `[unknown]` явно не певец. Запрос к исходному набору данных показывает, что он встречается 429 447 раз, что ставит его почти в первую сотню! На самом деле, это значение по умолчанию для воспроизведения песни без указания исполнителя, которое может быть предоставлено определенным клиентом скробблинга. Это бесполезная информация, и мы должны отбросить ее из данных, прежде чем начинать очередное обучение модели. Вот вам пример итеративности науки о данных, когда открытия в отношении данных происходят на каждом этапе.

Эта модель может быть использована для предоставления рекомендаций всем пользователям. Как вариант, можно настроить периодический запуск процесса, который пересчитывает модель и рекомендации для пользователей каждый час или даже реже, в зависимости от размера данных и скорости кластера.

Однако на данный момент реализация ALS в Spark MLlib не поддерживает метод, позволяющий выдавать рекомендации сразу всем пользователям. Можно выдать рекомендации одному пользователю за раз, как показано выше, хотя для каждого из них это будет всего лишь запуск распределенного задания продолжительностью в несколько секунд. Мы можем с легкостью организовать пересчет рекомендаций для небольших групп пользователей. В следующем коде вычисляются и выводятся на печать рекомендации 100 пользователям:

```
some_users = all_data.select("user").distinct().limit(100) ❶
val someRecommendations =
  someUsers.map(userID => (userID, makeRecommendations(model, userID, 5)))
someRecommendations.foreach { case (userID, recsDF) =>
  val recommendedArtists = recsDF.select("artist").as[Int].collect()
  println(s"$userID -> ${recommendedArtists.mkString(", ")}")
}

...
1000190 -> 6694932, 435, 1005820, 58, 1244362
1001043 -> 1854, 4267, 1006016, 4468, 1274
1001129 -> 234, 1411, 1307, 189, 121
...
```

❶ Подмножество из 100 отдельных пользователей.

В данном случае рекомендации просто выводятся в терминал. С таким же успехом их можно записать во внешнее хранилище, такое, как HBase, которое обеспечивает быстрый поиск во время выполнения.

Что дальше?

Естественно, можно потратить больше времени на настройку параметров модели и поиск и исправление аномалий во входных данных, наподобие певца `[unknown]`. Например, быстрый анализ количества воспроизведений показывает, что пользователь 2064012 воспроизвел песни певца 4468 целых 439 771 раз! Певец 4468 — невероятно успешная альтернативная метал-группа System of a Down, которая ранее упоминалась в рекомендациях. Если предположить, что средняя продолжительность песни составляет 4 минуты, это более 33 лет исполнения таких хитов, как «Chop Suey!» и «B.Y.O.B.» Поскольку группа начала записывать альбомы в 1998 г., для такого результата пришлось бы воспроизводить четыре или пять треков одновременно в течение семи лет. Вероятно, это злонамеренная «накрутка» рейтинга или ошибка данных, а также еще один пример проблем с данными, которые должна решать реальная рекомендательная система.

ALS — не единственный возможный рекомендательный алгоритм, но в настоящее время Spark MLlib поддерживает только его. Однако MLlib также поддерживает вариант ALS для неявных данных. Его использование идентично, за исключением того что ALS сконфигурирован с параметром `setImplicitPrefs(false)`. Неявный вариант применим, когда данные больше похожи на рейтинг, чем на подсчет. Например, он подходит, когда набор данных представляет собой пользовательские оценки певцов по шкале от 1 до 5. Результирующий столбец `prediction`, возвращенный `ALSModel.transform`, является прогнозируемым рейтингом. В этом случае для оценки рекомендательной системы подходит простая метрика RMSE (среднеквадратичная ошибка).

В дальнейшем в Spark MLlib и прочих библиотеках могут появиться и другие рекомендательные алгоритмы.

В производственной среде рекомендательные механизмы часто должны давать результат в режиме реального времени, потому что они используются в таких приложениях, как сайты электронной коммерции, где рекомендации часто запрашиваются в момент выдачи клиенту страниц продуктов. Чтобы иметь возможность быстро получать рекомендации для произвольного количества пользователей, можно вычислить их заранее и сохранить. Одним из недостатков этого подхода является необходимость предварительного расчета рекомендаций для всех пользователей, которым вскоре могут понадобиться рекомендации, то есть потенциально любому из них. Например, если за день только 10 000 из 1 миллиона пользователей посещают сайт, то ежедневный предварительный расчет рекомендаций для миллиона пользователей на 99% будет напрасным трудом.

Было бы лучше вычислять рекомендации на лету, по мере необходимости. Хотя мы можем вычислить рекомендации для одного пользователя с помощью `ALSModel`, но это распределенная операция, которая неизбежно занимает несколько секунд, потому что `ALSModel` чрезвычайно велика и фактически является распределенным набором данных. Это не относится к другим моделям, которые обеспечивают гораздо более быстрый вывод.

ГЛАВА 4

Прогнозирование с помощью деревьев и лесов решений

Классификация и *регрессия* — старейшие и наиболее хорошо изученные виды предиктивной аналитики. Большинство алгоритмов, с которыми вы, вероятно, столкнетесь в аналитических пакетах и библиотеках, представляют собой методы классификации или регрессии, такие, как *машины опорных векторов*, *логистическая регрессия*, *нейронные сети* и *глубокое обучение*. Общая нить, связывающая регрессию и классификацию, заключается в том, что обе они предполагают прогнозирование одного (или нескольких) значений на основании одного (или нескольких) других значений. Для этого им требуется набор входных и выходных данных, на которых можно учиться. Их нужно обеспечить как вопросами, так и эталонными ответами. По этой причине они относятся к разновидностям *обучения с учителем*.

PySpark MLlib предлагает реализации ряда алгоритмов классификации и регрессии. К ним относятся деревья решений, наивный байесовский метод, логистическая регрессия и линейная регрессия. Самое примечательное в этих алгоритмах то, что они помогают предсказать будущее или, по крайней мере, предсказать то, чего мы еще не знаем наверняка, например, вероятность того, что вы купите автомобиль, основываясь на вашем поведении в Интернете; является ли электронное письмо спамом, учитывая слова, которые оно содержит; или на каких участках земли будет собран наилучший урожай сельскохозяйственных культур, учитывая их местоположение и химический состав почвы.

В этой главе мы сосредоточимся на популярном и гибком типе алгоритма как для классификации, так и для регрессии (деревья решений), а также на расширении этого алгоритма (случайные леса решений). Мы начнем со знакомства с основами деревьев и лесов решений и представим первую реализацию в PySpark. Реализация деревьев решений в PySpark поддерживает бинарную и многоклассовую классификацию, а также регрессию. Эта реализация разделяет данные по строкам, позволяя проводить распределенное обучение с миллионами или даже миллиардами экземпляров. За этим последует подготовка набора данных и создание нашего первого дерева решений. Затем мы настроим нашу модель дерева решений. Мы закончим главу обучением модели случайного леса на обработанном наборе данных и сделаем прогнозы.

Хотя с реализацией дерева решений PySpark можно быстро приступить к работе, будет полезно понять основы алгоритмов дерева решений и случайного леса. Мы рассмотрим эти темы в следующем разделе.

Деревья и леса решений

Деревья решений — это семейство алгоритмов, которые естественным образом могут обрабатывать как категориальные, так и числовые признаки. С помощью параллельных вычислений можно построить одно дерево, а можно — много деревьев. Они устойчивы к выбросам в данных, а это означает, что несколько экстремальных и, возможно, ошибочных точек данных скорее всего вообще не повлияют на прогнозы. Они могут потреблять данные разных типов и в разных масштабах без предварительной обработки или нормализации.

Преимущество алгоритмов на основе дерева решений заключается в том, что они интуитивно понятны для понимания и рассуждений. Фактически мы все используем неявные рассуждения в форме деревьев решений в повседневной жизни. Например, я собираюсь пить утренний кофе с молоком. Прежде чем добавить молоко в свой напиток, я пытаюсь сделать прогноз: молоко испорчено? Я не знаю наверняка. Я могу проверить, не истек ли срок годности. Если нет, то я предсказываю, что молоко не испорчено. Если срок годности истек, но это случилось не более 3 дней назад, я рискну предположить, что нет, молоко не испорчено. В противном случае я просто понюхаю молоко. Если оно странно пахнет, я даю прогноз «да, испорчено», если нормально — «нет, не испорчено».

Эта серия решений «да/нет», пошагово приводящая к прогнозу, представляет собой суть деревьев решений. Каждое решение приводит к одному из двух результатов, который является либо прогнозом, либо другим решением, как показано на рис. 4.1. В этом смысле естественно рассматривать процесс принятия решения как дерево решений, где каждый внутренний узел дерева — это решение, а каждый конечный (иногда говорят листовой) узел — окончательный ответ.

Рис. 4.1. Дерево решений: испорчено ли молоко?

Это упрощенное дерево решений, построенное без какой-либо строгости. Для уточнения рассмотрим еще один пример. Робот устроился на работу в зоомагазин экзотических животных. Прежде чем магазин откроется, робот хочет узнать, какие животные в магазине станут хорошим домашним питомцем для ребенка. Владелец магазина перечисляет девять домашних животных, которые подходят и не подходят детям, и уходит домой. Робот собирает в табл. 4.1 информацию, полученную при осмотре животных.

Таблица 4.1. *Таблица признаков для магазина экзотических животных*

Кличка	Вес, кг	Кол-во ног	Цвет	Подходит детям?
Фидо	20,5	4	Коричневый	Да
Мистер Слизер	3,1	0	Зеленый	Нет
Немо	0,2	0	Оранжевый	Да
Дамбо	1390,8	4	Серый	Нет
Китти	12,1	4	Серый	Да
Джим	150,9	2	Оранжевый	Нет
Милли	0,1	100	Коричневый	Нет
Мистер Голубь	1,0	2	Серый	Нет
Пятнышко	10,0	4	Коричневый	Да

Роботу не составит труда принять решение относительно девяти перечисленных питомцев. Но в магазине есть еще много других животных. Поэтому роботу нужна методика принятия решения о том, какие животные среди остальных подходят в качестве домашних питомцев для детей. Будем считать, что характеристики всех животных имеются. Используя готовые решения, предоставленные владельцем магазина, и дерево решений, мы можем научить робота узнавать, как выглядит подходящий питомец для ребенка.

Хотя имя животного известно, оно не будет включено в качестве признака в нашу модель дерева решений. Вряд ли имя позволяет сделать какой-то осмысленный вывод: Феликсом с равной вероятностью могли назвать кота или ядовитого тарантула. Итак, у нас есть два числовых признака (вес, количество ног) и один категориальный признак (цвет), предсказывающие категориальную цель (является/не является хорошим питомцем для ребенка).

Работа дерева решений заключается в последовательном принятии одного или нескольких решений на основе предоставленных признаков. Для начала робот может попытаться подогнать к этим обучающим данным простое дерево решений, состоящее из одного решения, основанного на весе, как показано на рис. 4.2.

Логика дерева решений очевидна и имеет смысл: животные весом 500 кг определенно не подходят в качестве домашнего питомца для ребенка. Это правило предсказывает правильное значение в пяти из девяти случаев. Беглый взгляд на данные пока-

зывает, что мы могли бы улучшить правило, снизив порог веса до 100 кг. Этот критерий соответствует шести из девяти примеров. Тяжелые животные теперь предсказываются правильно; но для более легких животных прогноз верен лишь отчасти.

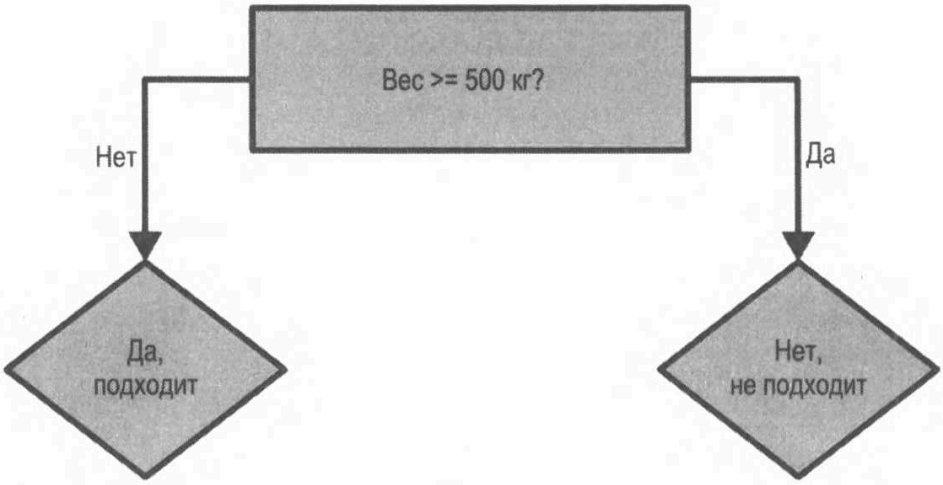


Рис. 4.2. Первое дерево решений робота

Далее можно построить второе решение, уточняющее прогноз для примеров животных с весом менее 100 кг. Было бы хорошо выбрать признак, который заменяет некоторые из неправильных прогнозов «Да» на «Нет». Например, есть одно маленькое зеленое животное, подозрительно похожее на змею, которое будет классифицировано нашей текущей моделью как подходящий кандидат на домашнее животное. Робот может получить правильный прогноз, добавив решение на основе цвета, как показано на рис. 4.3.

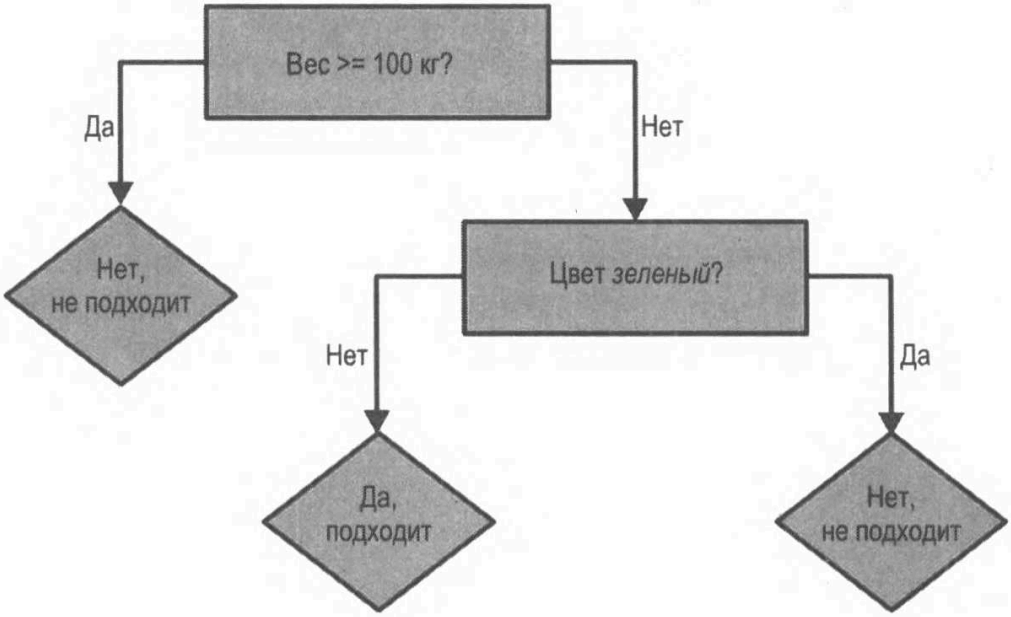


Рис. 4.3. Второе дерево решений робота

Итак, семь из девяти прогнозов верны. Конечно, правила принятия решений можно добавлять до тех пор, пока все девять прогнозов не станут правильными. Логика,

воплощенная в результирующем дереве решений, прозвучала бы несколько неправдоподобно при переводе на обычный язык: «Если вес животного меньше 100 кг, его цвет коричневый, а не зеленый, и у него меньше 10 ног, то да, это подходящий питомец». И хотя такое дерево решений идеально подходит для приведенных примеров, оно не может предсказать, что маленькая коричневая четвероногая росомаха (дикий хищник) не является подходящим домашним животным. Необходим некоторый баланс, чтобы избежать этого явления, известного как *переобучение*.

Деревья решений обобщаются в более мощный алгоритм, называемый случайным лесом. *Случайные леса* объединяют множество деревьев решений, чтобы снизить риск переобучения, и обучают деревья решений по отдельности. Алгоритм вводит случайность в процесс обучения, так что каждое дерево решений немного отличается от остальных. Объединение прогнозов снижает дисперсию прогнозов, делает результирующую модель более обобщаемой и повышает оценку при проверке модели на тестовых данных.

Этого знакомства с деревьями решений и случайными лесами достаточно, чтобы начать использовать их с PySpark. В следующем разделе мы проанализируем набор данных, с которым будем работать, и подготовим его для использования в PySpark.

Подготовка данных

Набор данных, используемый в этой главе, представляет собой хорошо известный набор Covtype, доступный онлайн в виде сжатого файла данных covtype.data.gz в формате CSV и сопутствующего информационного файла covtype.info.

Набор данных содержит записи о покрытых лесом участках земли в Колорадо, США. Это просто совпадение, что набор данных касается реальных лесов! Каждая запись содержит несколько характеристик, описывающих отдельный участок земли, таких, как его высота, уклон, расстояние до воды, тень и тип почвы, а также известный тип леса, покрывающего землю. Мы будем прогнозировать тип лесного покрова по нескольким признакам, которых в общей сложности 54.

Этот набор данных использовался в исследованиях и даже в соревнованиях Kaggle. Для нас этот набор сейчас особенно интересен, поскольку он содержит как категориальные, так и числовые признаки. В наборе содержится 581 012 записей, которые не дотягивают до категории больших данных, но их достаточно много, чтобы можно было использовать в качестве примера и при этом обнаружить некоторые проблемы масштаба.

К счастью, данные уже представлены в простом формате CSV и не требуют особой очистки или другой подготовки для использования с PySpark MLlib. Файл covtype.data следует извлечь из архива и скопировать в локальное или облачное хранилище (например, AWS S3).

Запустите `pyspark-shell`. Возможно, вам будет полезно предоставить оболочке достаточный объем памяти для работы, поскольку построение леса принятия решений

может потребовать больших ресурсов. Если у вашего компьютера достаточно много памяти, укажите `--driver-memory 8g` или больше.

Файлы CSV содержат в основном табличные данные, организованные по типу «строка-столбцы». Иногда этим столбцам дают имена в строке заголовка, но здесь это не так. Имена столбцов указаны в сопроводительном файле covtype.info. Вообще-то, у каждого столбца CSV-файла также есть тип — число или строка, но в CSV-файле это не указано.

Вполне логично анализировать эти данные как фрейм данных, потому что это абстракция PySpark для табличных данных с определенной схемой столбцов, включая имена и типы столбцов. PySpark имеет встроенную поддержку чтения данных CSV. Давайте прочитаем наш набор данных как фрейм данных DataFrame, используя встроенное средство чтения CSV:

```
data_without_header = spark.read.option("inferSchema", True)\
                      .option("header", False).csv("data/covtype.data")
data_without_header.printSchema()
...
root
 |-- _c0: integer (nullable = true)
 |-- _c1: integer (nullable = true)
 |-- _c2: integer (nullable = true)
 |-- _c3: integer (nullable = true)
 |-- _c4: integer (nullable = true)
 |-- _c5: integer (nullable = true)
...
```

Этот код считывает ввод как CSV и не пытается анализировать первую строку как заголовок имен столбцов. Он также требует, чтобы тип каждого столбца определялся путем изучения данных. Он правильно делает вывод, что все столбцы являются числами, а точнее, целыми числами. К сожалению, он может именовать столбцы только как `_c0` и так далее.

Мы можем посмотреть имена столбцов в файле covtype.info.

```
$ cat data/covtype.info


...
[...]
7.      Attribute information:

Given is the attribute name, attribute type, the measurement unit
and
a brief description. The forest cover type is the classification
problem. The order of this listing corresponds to the order of
numerals along the rows of the database.
(Даны имя атрибута, тип атрибута, единица измерения и краткое описание.
Тип лесного покрова является задачей классификации.
Порядок этого списка соответствует порядку номеров в строках базы данных.)
```

Name	Data Type
Elevation	quantitative
Aspect	quantitative
Slope	quantitative
Horizontal_Distance_To_Hydrology	quantitative
Vertical_Distance_To_Hydrology	quantitative
Horizontal_Distance_To_Roadways	quantitative
Hillshade_9am	quantitative
Hillshade_Noon	quantitative
Hillshade_3pm	quantitative
Horizontal_Distance_To_Fire_Points	quantitative
Wilderness_Area (4 binary columns)	qualitative
Soil_Type (40 binary columns)	qualitative
Cover_Type (7 types)	integer

Measurement Description

meters	Elevation in meters (Высота в метрах)
azimuth	Aspect in degrees azimuth (Азимут направления наклона в градусах)
degrees	Slope in degrees (Наклон в градусах)
meters	Horz Dist to nearest surface water features (Расст. до ближайшего водоема по горизонтали)
meters	Vert Dist to nearest surface water features (Расст. до ближайшего водоема по вертикали)
meters	Horz Dist to nearest roadway (Расст. до ближайшей дороги по горизонтали)
0 to 255 index	Hillshade index at 9am, summer solstice (Индекс тени в 9:00, летнее солнцестояние)
0 to 255 index	Hillshade index at noon, summer solstice (Индекс тени в полдень, летнее солнцестояние)
0 to 255 index	Hillshade index at 3pm, summer solstice (Индекс тени в 15:00, летнее солнцестояние)
meters	Horz Dist to nearest wildfire ignition point (Расст. до ближ. точки возг. лесного пожара)
0 (absence) or 1 (presence)	Wilderness area designation (Обозначение заповедной зоны)
0 (absence) or 1 (presence)	Soil Type designation (Обозначение типа грунта)
1 to 7	Forest Cover Type designation (Обозначение типа лесного покрова)

...

Из информации о столбцах становится ясно, что некоторые признаки действительно являются числовыми. Высота измеряется в метрах, а уклон — в градусах. Одна-

ко признак `Wilderness_Area` — это нечто иное, потому что считается, что он охватывает четыре столбца, каждый из которых равен `0` или `1`. На самом деле `Wilderness_Area` — это категориальное значение, а не числовое.

Эти четыре столбца представляют собой *унитарный код* или *код 1-из-N*. Когда эта форма кодирования применяется к категориальному признаку, один категориальный признак, который принимает N различных значений, превращается в N числовых признаков, каждый из которых принимает значение 0 или 1. В унитарном коде только одно из N значений равно 1, а остальные равны 0. Например, категориальный признак погоды, которая может быть облачной, дождливой или ясной, превращается в три числовых характеристики, где облачность представлена кодом 1,0,0; дождь — 0,1,0; и так далее. Эти три числовых признака можно рассматривать как признаки `is_cloudy`, `is_rainy` и `is_clear`. Аналогично, 40 столбцов наших данных на самом деле представляют собой один категориальный признак `Soil_Type`.

Это не единственный возможный способ кодирования категориального признака в виде числа. Другой вариант кодировки просто присваивает отдельное числовое значение каждому возможному значению категориального признака. Например, облачность может получить обозначение 1.0, дождь — 2.0 и так далее. Цель нашего прогноза `Cover_Type` представляет собой категориальное значение, закодированное как число от 1 до 7.

Будьте осторожны при кодировании категориального признака в виде одиночного числа. Исходные категориальные значения не упорядочены, но когда они закодированы как число, возникает порядок. Однако обработка закодированного признака как числа приводит к бессмысленным результатам, потому что алгоритм фактически делает вид, что дождливая погода каким-то образом в два раза больше, чем облачность. Будет правильно не использовать числовое значение кодировки в качестве числа.

В нашем наборе данных присутствуют оба типа кодирования категориальных признаков. Возможно, было бы проще и понятнее не кодировать такие объекты (к тому же, двумя способами), а вместо этого просто включать их значения напрямую, например, «Заповедник Рава». Такая форма данных сложилась исторически; набор данных был выпущен в 1998 году. Из соображений быстродействия или для соответствия формату, принятому в библиотеках того времени, созданных в основном для задач регрессии, наборы данных часто закодированы таким образом.

В любом случае, прежде чем продолжить, полезно добавить имена столбцов в этот фрейм данных, чтобы упростить работу с ним:

```
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import col

colnames = ["Elevation", "Aspect", "Slope", \
            "Horizontal_Distance_To_Hydrology", \
            "Vertical_Distance_To_Hydrology", \
            "Horizontal_Distance_To_Roadways", \
            "Hillshade_9am", "Hillshade_Noon", "Hillshade_3pm", \
            "Horizontal_Distance_To_Fire_Points"] + \
```

```
[f"Wilderness_Area_{i}" for i in range(4)] + \ ❶
[f"Soil_Type_{i}" for i in range(40)] + \
["Cover_Type"]

data = data_without_header.toDF(*colnames).\
                          withColumn("Cover_Type",
                                col("Cover_Type").cast(DoubleType()))

data.head()
...
Row(Elevation=2596,Aspect=51,Slope=3,Horizontal_Distance_To_Hydrology=258,...)
```

❶ + выполняет конкатенацию коллекций.

Столбцы, связанные с дикой природой и почвой, называются `Wilderness_Area_0`, `Soil_Type_0` и т. д., и небольшой код на Python может сгенерировать эти 44 названия без необходимости вводить их вручную. Наконец, целевой столбец `Cover_Type` заранее преобразуется в значение типа `double`, потому что на самом деле необходимо будет использовать его именно как `double`, а не `int` во всех API-интерфейсах PySpark MLlib. Позже вы увидите почему.

Вы можете вызвать метод `data.show`, чтобы увидеть несколько строк набора данных, но ширины дисплея недостаточно, и прочитать их полностью будет сложно. Метод `data.head` отображает строки как необработанный объект `Row`, который в этом случае будет более удобочитаемым.

Теперь, когда мы познакомились с набором данных и обработали его, можно приступать к обучению модели дерева решений.

Наше первое дерево решений

В *главе 3* мы сразу построили рекомендательную модель на всех доступных данных. Модель выдавала рекомендации, которые мог проверить любой, кто хоть немного разбирается в музыке: глядя на привычки и рекомендации пользователя, мы увидели, что она дает хорошие результаты. Здесь это невозможно. Мы понятия не имеем, как составить описание нового участка земли в Колорадо, состоящее из 54 признаков, и уж тем более, какой лесной покров может быть на этом участке.

Вместо этого мы должны сразу перейти к сохранению некоторых данных для оценки полученной модели. Ранее метрика AUC использовалась для оценки соответствия между данными прослушивания песен и прогнозами, полученными на основе рекомендаций. AUC можно рассматривать как вероятность того, что случайно выбранная хорошая рекомендация окажется выше случайно выбранной плохой рекомендации. Принцип здесь тот же, хотя метрика оценки будет другой — точность. Большая часть данных (90%) снова будет использоваться для обучения, а позже вы увидите, что часть этой обучающей выборки будет отложена для перекрестной

проверки (выборка валидации). Остальные 10%, представленные здесь, на самом деле являются *третьим* подмножеством — эталонной тестовой выборкой.

```
(train_data, test_data) = data.randomSplit([0.9, 0.1])
train_data.cache()
test_data.cache()
```

Данные нуждаются в дополнительной подготовке для использования с классификатором в MLlib. Входной фрейм содержит множество столбцов, каждый из которых содержит один признак, пригодный для прогнозирования целевого столбца. MLlib требует, чтобы все входные данные были собраны в *один* столбец, значением которого является вектор. Класс `VectorAssembler` в PySpark представляет собой абстракцию векторов в смысле линейной алгебры и содержит только числа. В большинстве случаев они работают как простой массив значений типа `double` (числа с плавающей запятой). Конечно, некоторые из входных признаков по сути категориальны, даже если все они представлены числами во входных данных.

К счастью, класс `VectorAssembler` может сделать эту работу:

```
from pyspark.ml.feature import VectorAssembler

input_cols = colnames[:-1] ❶
vector_assembler = VectorAssembler(inputCols=input_cols,
                                   outputCol="featureVector")
Our First Decision Tree | 67
assembled_train_data = vector_assembler.transform(train_data)

assembled_train_data.select("featureVector").show(truncate = False)
...
+------------------------------------------------------------------ ...
|featureVector                                                      ...
+------------------------------------------------------------------ ...
|(54,[0,1,2,5,6,7,8,9,13,18],[1874.0,18.0,14.0,90.0,208.0,209.0, ...
|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1879.0,28.0,19.0,30.0,12.0,95.0, ...
...
```

❶ Исключая метку `Cover_Type`.

Ключевыми параметрами `VectorAssembler` являются столбцы, подлежащие объединению в вектор признаков, и имя нового столбца, содержащего вектор признаков. Здесь все столбцы, кроме последнего, включены в качестве входных признаков. Полученный фрейм данных имеет новый столбец `featureVector`, как показано в листинге выше.

Вывод не выглядит как ожидаемая последовательность чисел, но это потому, что он показывает необработанное представление вектора `SparseVector`, применяемое для экономии памяти. Поскольку большинство из 54 значений равны 0, в памяти хранятся только ненулевые значения и их индексы. Эта техническая деталь не имеет значения для классификации.

Вектор признаков

Представьте, что вам нужно предсказать завтрашнюю максимальную дневную температуру, исходя из сегодняшней погоды. В этой идее нет ничего плохого, но «сегодняшняя погода» — это случайное понятие, которое требует структурирования, прежде чем его можно будет ввести в алгоритм обучения. Действительно, определенные характеристики сегодняшней погоды позволяют предсказать завтрашнюю температуру, например, дневная и ночная температура, средняя влажность, облачно, дождливо или ясно. Также имеет значение количество синоптиков, предсказывающих завтрашнее похолодание.

Эти признаки также иногда называют *измерениями*, *предикторами* или просто *переменными*. Каждый из этих признаков может быть определен количественно. Например, высокие и низкие температуры измеряются в градусах Цельсия, влажность может измеряться дробным числом от 0 до 1, а тип погоды может обозначаться словами «облачно», «дождливо» или «ясно». Количество синоптиков — это, конечно, целое число. Таким образом, сегодняшняя погода может быть сведена к списку значений, таких, как `13.1,19.0,0.73,облачно,1`.

Эти пять признаков вместе, расположенных по порядку, называются *вектором признаков* и могут описывать погоду в любой день. Такое определение вектора имеет некоторое сходство с использованием термина «вектор» в линейной алгебре, за исключением того, что вектор признаков в принципе может содержать нечисловые значения и даже не иметь некоторых значений. Эти признаки не все одного типа. Первые два признака измеряются в градусах Цельсия, а третий является безразмерной дробью. Четвертый — вообще не число, а пятый — всегда только целое положительное число.

Алгоритм обучения должен воспользоваться имеющимися данными, чтобы научить модель делать прогнозы. Векторы признаков обеспечивают организованный способ описания входных данных для алгоритма обучения (например, `12.5,15.5,0.10,ясно,0`). Результат или *цель* прогноза также можно рассматривать как признак. Здесь это числовой признак `17.2`.

Нередко цель просто включается в качестве еще одного признака в вектор признаков. Весь обучающий пример можно представить как `12.5,15.5,0.10,ясно,0,17.2`. Совокупность всех этих примеров называется *обучающей выборкой*.

`VectorAssembler` — это пример метода `Transformer` в текущем API MLlib Pipelines. Он преобразует входной фрейм данных в другой фрейм в соответствии с определенной логикой и может быть объединен с другими преобразованиями в конвейер. Позже в этой главе наши преобразования тоже будут объединены в конвейер. А пока мы вызываем преобразование напрямую, этого достаточно для построения первой модели-классификатора на основе дерева решений:

```
from pyspark.ml.classification import DecisionTreeClassifier

classifier = DecisionTreeClassifier(seed = 1234, labelCol="Cover_Type",
                                    featuresCol="featureVector",
                                    predictionCol="prediction")
```

```
model = classifier.fit(assembled_train_data)
print(model.toDebugString)
...
DecisionTreeClassificationModel: uid=DecisionTreeClassifier_da03f8ab5e28, ...
  If (feature 0 <= 3036.5)
    If (feature 0 <= 2546.5)
      If (feature 10 <= 0.5)
        If (feature 0 <= 2412.5)
          If (feature 3 <= 15.0)
            Predict: 4.0
          Else (feature 3 > 15.0)
            Predict: 3.0
        Else (feature 0 > 2412.5)
...
```

Опять же, основная конфигурация классификатора состоит из имен столбцов: столбец, содержащий входные векторы признаков, и столбец, содержащий целевое значение для прогнозирования. Поскольку позже модель будет использоваться для прогнозирования новых целевых значений, ей присваивается имя столбца для хранения прогнозов.

В приведенном выше листинге модели видна часть ее древовидной структуры. Она состоит из серии вложенных решений о признаках, сравнивающих значения признаков с пороговыми значениями. (К сожалению, здесь по историческим причинам признаки упоминаются только по номеру, а не по имени.)

Деревья решений способны оценивать важность входных признаков как часть процесса построения дерева. То есть они могут оценить, насколько каждый входной признак способствует правильному прогнозированию. Эту информацию легко извлечь из модели:

```
import pandas as pd

pd.DataFrame(model.featureImportances.toArray(),
             index=input_cols, columns=['importance']).\
             sort_values(by="importance", ascending=False)
...

                                  importance
Elevation                         0.826854
Hillshade_Noon                    0.029087
Soil_Type_1                       0.028647
Soil_Type_3                       0.026447
Wilderness_Area_0                 0.024917
Horizontal_Distance_To_Hydrology  0.024862
Soil_Type_31                      0.018573
Wilderness_Area_2                 0.012458
Horizontal_Distance_To_Roadways   0.003608
Hillshade_9am                     0.002840
...
```

В этом листинге значения важности (чем выше, тем лучше) сопоставлены с именами столбцов и выведены на печать в порядке от наиболее важного к наименее важному. Высота, по-видимому, доминирует как самый важный признак; и, судя по оценке, большинство признаков практически не играют особой роли при прогнозировании типа растительного покрова!

Результирующая модель `DecisionTreeClassificationModel` сама по себе является трансформером (преобразователем), поскольку она может преобразовывать фрейм данных, содержащий векторы признаков, в фрейм данных, содержащий прогнозы.

Например, было бы интересно посмотреть, что модель предсказывает на обучающих данных, и сравнить ее предсказание с известным правильным типом покрова:

```
predictions = model.transform(assembled_train_data)
predictions.select("Cover_Type", "prediction", "probability").\
            show(10, truncate = False)
```

```
...
+----------+----------+------------------------------------------------ ...
|Cover_Type|prediction|probability                                      ...
+----------+----------+------------------------------------------------ ...
|6.0       |4.0       |[0.0,0.0,0.028372324539571926,0.2936784469885515, ...
|6.0       |3.0       |[0.0,0.0,0.024558587479935796,0.6454654895666132, ...
|6.0       |3.0       |[0.0,0.0,0.024558587479935796,0.6454654895666132, ...
|6.0       |3.0       |[0.0,0.0,0.024558587479935796,0.6454654895666132, ...
...
```

Интересно, что выходные данные также содержат столбец `probability`, который содержит оценку модели на предмет того, насколько вероятно, что каждый возможный результат является правильным. Очевидно, в некоторых случаях можно с уверенностью сказать, что ответ равен 3, и совершенно точно, что ответ не равен 1.

Внимательные читатели могут заметить, что векторы вероятности на самом деле имеют восемь значений, хотя возможных исходов всего семь. Значения вектора с индексами от 1 до 7 действительно содержат вероятность исходов от 1 до 7. Однако есть также значение с индексом 0, которое всегда отображается как вероятность 0,0. Это значение следует игнорировать, так как индекс 0 отсутствует в данных, но в программировании индексы вектора начинаются с нуля. Это особенность представления информации в виде вектора, о которой стоит знать.

Глядя на приведенный выше фрагмент вывода, мы можем сделать вывод, что модель нуждается в некоторой доработке, поскольку ее прогнозы слишком часто выглядят ошибочными. Как и в случае алгоритма ALS, описанного в *главе 3*, реализация `DecisionTreeClassifier` имеет несколько гиперпараметров, для которых необходимо выбрать оптимальные значения, но сейчас они назначены по умолчанию. Чтобы получить объективную оценку точности модели с гиперпараметрами по умолчанию, нужно воспользоваться набором тестов.

Далее мы применим `MulticlassClassificationEvaluator` для вычисления точности и других показателей, которые отражают качество прогнозов модели.

Вот пример оценщика в MLlib, который отвечает за оценку качества выходного фрейма данных:

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type",
                                              predictionCol="prediction")

evaluator.setMetricName("accuracy").evaluate(predictions)
evaluator.setMetricName("f1").evaluate(predictions)

...
0.6989423087953562
0.6821216079701136
```

После извлечения столбца, содержащего так называемую «метку» (целевое или известное правильное выходное значение), и столбца, содержащего прогноз, обнаруживается, что они совпадают примерно в 70% случаев. Такова *точность* нашего классификатора. Он может вычислять другие связанные показатели, такие, как оценка F1. В этой книге для оценки классификаторов мы будем использовать точность.

Это единственное число дает хорошее представление о качестве выходных данных классификатора. Однако иногда полезно взглянуть на *матрицу несоответствий* (confusion matrix). Это таблица со строкой и столбцом для каждого возможного значения цели. Поскольку существует семь значений целевых категорий, это матрица 7×7, где каждая строка соответствует фактическому правильному значению, а каждый столбец — прогнозируемому значению по порядку. Запись в строке *i* и столбце *j* содержит подсчет, сколько раз пример с истинной категорией *i* был предсказан как категория *j*. Таким образом, подсчеты *правильных* прогнозов модели размещены по диагонали, а неправильных — в остальных ячейках.

Матрицу несоответствий можно рассчитать непосредственно с помощью API DataFrame, используя его более общие операторы.

```
confusion_matrix = predictions.groupBy("Cover_Type").\
  pivot("prediction", range(1,8)).count().\
  na.fill(0.0).\ ❶
  orderBy("Cover_Type")

confusion_matrix.show()

...
+----------+------+------+-----+---+---+---+-----+
|Cover_Type|     1|     2|    3|  4|  5|  6|    7|
+----------+------+------+-----+---+---+---+-----+
|       1.0|133792| 51547|  109|  0|  0|  0| 5223|
|       2.0| 57026|192260| 4888| 57|  0|  0|  750|
|       3.0|     0|  3368|28238|590|  0|  0|    0|
```

```
|        4.0|      0|      0| 1493|956|  0|  0|     0|
|        5.0|      0|   8282|  283|  0|  0|  0|     0|
|        6.0|      0|   3371|11872|406|  0|  0|     0|
|        7.0|   8122|     74|    0|  0|  0|  0|10319|
+----------+------+------+-----+---+---+---+-----+
```

❶ Заменяем `null` на 0.

Пользователи электронных таблиц, возможно, увидят здесь явную аналогию с вычислением *сводной таблицы*. Сводная таблица группирует значения по двум измерениям, значения которых становятся строками и столбцами вывода, и вычисляет некоторую агрегацию внутри этих группировок. В данном случае выполняется подсчет прогнозов. Этот способ также доступен как функция `PIVOT` в нескольких базах данных и поддерживается Spark SQL. Но, возможно, наш способ вычисления оценки окажется более элегантным и мощным.

Хотя точность 70% звучит солидно, не сразу понятно, замечательная она или плохая. Насколько хорошо упрощенный подход поможет установить базовый уровень? Точно так же как сломанные часы правильно показывают время дважды в сутки, случайное угадывание классификации для каждого примера также может иногда давать правильный ответ.

Мы можем построить такой случайный «классификатор», выбирая класс случайным образом пропорционально его присутствию в обучающей выборке. Например, если 30% обучающей выборки занимает лесной покров типа 1, то случайный классификатор будет угадывать «1» в 30% случаев. Каждая классификация будет правильной пропорционально ее присутствию в тестовой выборке. Если 40% тестовой выборки занимает покров 1, то версия «1» будет правильной в 40% случаев. Таким образом, тип покрова 1 будет угадан правильно в 30% × 40% = 12% случаев и внесет 12% в общую оценку точности модели. Следовательно, мы можем оценить точность полностью случайного угадывания, просуммировав эти произведения вероятностей:

```
from pyspark.sql import DataFrame

def class_probabilities(data):
    total = data.count()
    return data.groupBy("Cover_Type").count().\❶
    orderBy("Cover_Type").\❷
    select(col("count").cast(DoubleType())).\
    withColumn("count_proportion", col("count")/total).\
    select("count_proportion").collect()

train_prior_probabilities = class_probabilities(train_data)
test_prior_probabilities = class_probabilities(test_data)

train_prior_probabilities
...
```

```
[Row(count_proportion=0.36455357859838705),
 Row(count_proportion=0.4875111371136425),
 Row(count_proportion=0.06155716924206445),
 Row(count_proportion=0.00468236760696409),
 Row(count_proportion=0.016375858943914835),
 Row(count_proportion=0.029920118693908142),
 Row(count_proportion=0.03539976980111887)]

...

train_prior_probabilities = [p[0] for p in train_prior_probabilities]
test_prior_probabilities = [p[0] for p in test_prior_probabilities]

sum([train_p * cv_p for train_p, cv_p in zip(train_prior_probabilities,
                                             test_prior_probabilities)])❸
...

0.37735294664034547
```

❶ Подсчет по категориям.

❷ Сортировка подсчетов по категориям.

❸ Сумма произведений пар в обучающей и тестовой выборках.

Случайное угадывание достигает точности 37%, что делает 70% хорошим результатом. Но этот результат был достигнут с гиперпараметрами по умолчанию. Мы можем добиться еще большего успеха, изучив, что означают гиперпараметры для процесса построения дерева. Мы займемся этим в следующем разделе.

Гиперпараметры дерева решений

В *главе 3* алгоритм ALS использовал несколько гиперпараметров, значения которых мы подбирали, создавая модели с различными комбинациями значений, а затем оценивая качество каждого результата с помощью определенной метрики. Здесь процесс не изменился, только метрика теперь представляет собой точность многоклассовой классификации, а не AUC. Гиперпараметры, управляющие выбором решений дерева, также будут совершенно другими: максимальная глубина, максимальное количество ячеек, мера примеси и минимальный прирост информации.

Максимальная глубина просто ограничивает количество уровней в дереве решений. Это максимальное количество связанных решений, которые классификатор выполнит для классификации примера. Полезно ограничить значение этого параметра, чтобы избежать переобучения на обучающих данных, как было показано ранее в примере с зоомагазином.

Алгоритм дерева решений отвечает за создание потенциальных правил принятия решений, которые можно попробовать на каждом уровне; например, это были ре-

шения `weight >= 100` или `weight >= 500` в примере с зоомагазином. Решения всегда оформлены одинаково: для числовых признаков они представлены условием `характеристика >= значение`; а для категориальных признаков они имеют вид `признак в (значение1, значение2, ...)`. Таким образом, набор правил принятия решений, которые нужно попробовать, на самом деле является набором значений, которые необходимо подставить в правило принятия решений. В реализации PySpark MLlib они называются *бинами* (bin). Большее количество бинов требует больше времени на обработку, но может привести к нахождению более оптимального правила принятия решения.

Что делает правило хорошим? Логично предположить, что хорошее правило будет осмысленно различать образцы по значению целевой категории. Например, правило, которое делит набор данных Covtype на группы только с категориями 1–3, с одной стороны, и 4–7, с другой, было бы превосходным, поскольку оно четко отделяет одни категории от других. Правило, которое приводит к примерно такому же сочетанию категорий, как и во всем наборе данных, является бесполезным. Следование любой из ветвей такого решения приводит примерно к одинаковому распределению возможных целевых значений и, таким образом, не решает задачу классификации.

Иными словами, хорошие правила делят целевые значения обучающих данных на относительно однородные или «чистые» подмножества. Выбор наилучшего правила означает минимизацию примесей в двух подмножествах, которые оно порождает. Есть две обычно используемые меры примеси: примесь Джини и энтропия.

Примесь Джини напрямую связана с точностью случайного классификатора. Для подмножества она означает вероятность того, что случайно выбранная классификация случайно выбранного образца неверна (оба выбора делаются в соответствии с распределением классов в подмножестве). Чтобы вычислить эту вероятность, мы сначала умножаем каждый класс на его соответствующую долю среди всех классов. Затем мы вычитаем сумму всех значений из 1. Если подмножество имеет N классов и p_i представляет собой долю образцов класса i, то его примесь Джини определяется следующим уравнением:

$$I_G(p) = 1 - \sum_{i=1}^{N} p_i^2.$$

Если подмножество содержит только один класс, то значение примеси Джини равно 0, поскольку оно полностью «чистое». Когда в подмножестве есть N классов, это значение больше 0 и является наибольшим, когда классы встречаются одинаковое количество раз — максимально загрязнено.

Энтропия — еще одна мера количества примесей, заимствованная из теории информации. Природу энтропии объяснить сложнее, но она показывает, насколько неопределенна совокупность целевых значений в подмножестве прогнозов для данных, попадающих в это подмножество. Подмножество, содержащее только один класс, предполагает, что результат для подмножества полностью определен и имеет нулевую энтропию, т. е. неопределенность отсутствует. С другой стороны, под-

множество, содержащее по одному образцу из каждого возможного класса, предполагает большую неопределенность прогноза для этого подмножества. Такое подмножество имеет высокую энтропию. Следовательно, низкая энтропия, как и низкая примесь Джини, — это хорошо. Энтропия определяется уравнением энтропии:

$$I_E(p) = \sum_{i=1}^{N} p_i \log\left(\frac{1}{p_i}\right) = -\sum_{i=1}^{N} p_i \log(p_i).$$

Интересно, что неопределенность имеет свои единицы измерения. Поскольку в уравнении присутствует натуральный логарифм (по основанию *e*), единицы измерения — *наты*, а основание *e* соответствует более знакомым *битам* (которые мы можем получить, используя логарифм по основанию 2). Это и в самом деле мера информации, поэтому также принято говорить о *приросте информации* (information gain) правила принятия решений при использовании энтропии с деревьями решений.

Та или иная мера может быть лучшей метрикой для выбора правил принятия решений в заданном наборе данных. В чем-то они похожи. Оба включают средневзвешенное значение: сумму значений, взвешенных по числу p_i. По умолчанию в реализации PySpark используется примесь Джини.

Наконец, *минимальный прирост информации* — это гиперпараметр, который определяет минимальное улучшение чистоты подмножества, вносимое правилом-кандидатом. Правила, которые недостаточно улучшают чистоту подмножеств, отбрасываются. Как и более низкое значение максимальной глубины, этот гиперпараметр помогает модели противостоять переобучению, потому что правила, которые слабо разделяют обучающие данные, могут вообще не справиться с разделением реальных данных.

Теперь, когда мы понимаем назначение гиперпараметров алгоритма дерева решений, в следующем разделе мы займемся настройкой модели, чтобы повысить ее точность.

Настройка деревьев решений

При беглом взгляде на данные не очевидно, какая мера примесей приводит к большей точности или какая максимальная глубина или количество бинов достаточны, но не являются чрезмерными. К счастью, как и в *главе 3*, можно просто позволить PySpark попробовать несколько комбинаций этих значений и сообщить о результатах.

Прежде всего, необходимо настроить конвейер, инкапсулирующий два шага, которые мы выполнили в предыдущих разделах, — создание вектора признаков и его использование для построения модели дерева решений. Мы создаем два объекта `Transformer` — `VectorAssembler` и `DecisionTreeClassifier` и объединяем их в один объект `Pipeline`:

```
from pyspark.ml import Pipeline

assembler = VectorAssembler(inputCols=input_cols, outputCol="featureVector")
```

```
classifier = DecisionTreeClassifier(seed=1234, labelCol="Cover_Type",
                                    featuresCol="featureVector",
                                    predictionCol="prediction")

pipeline = Pipeline(stages=[assembler, classifier])
```

Разумеется, конвейеры могут быть намного длиннее и сложнее, но лучше делать их по возможности проще. Теперь мы можем определить комбинации гиперпараметров, которые следует протестировать, используя встроенную поддержку API PySpark ML, `ParamGridBuilder`. Также пришло время определить метрику оценки, которая будет использоваться для выбора лучшей комбинации, и это снова будет `MulticlassClassificationEvaluator`:

```
from pyspark.ml.tuning import ParamGridBuilder

paramGrid = ParamGridBuilder(). \
  addGrid(classifier.impurity, ["gini", "entropy"]). \
  addGrid(classifier.maxDepth, [1, 20]). \
  addGrid(classifier.maxBins, [40, 300]). \
  addGrid(classifier.minInfoGain, [0.0, 0.05]). \
  build()

multiclassEval = MulticlassClassificationEvaluator(). \
setLabelCol("Cover_Type"). \
setPredictionCol("prediction"). \
setMetricName("accuracy")
```

Из кода следует, что модель будет построена и оценена для двух значений четырех гиперпараметров. Это 16 моделей. Они будут оцениваться по критерию многоклассовой точности. Наконец, `TrainValidationSplit` объединяет эти компоненты — конвейер, который создает модели, метрики оценки моделей и гиперпараметры для проверки — и может выполнять оценку на обучающих данных. Стоит отметить, что здесь также можно использовать `CrossValidator` для выполнения полной k-кратной перекрестной проверки, но он в k раз дороже в вычислительном отношении и не дает заметного выигрыша при наличии больших данных. Итак, запускаем `TrainValidationSplit`:

```
from pyspark.ml.tuning import TrainValidationSplit

validator = TrainValidationSplit(seed=1234,
  estimator=pipeline,
  evaluator=multiclassEval,
  estimatorParamMaps=paramGrid,
  trainRatio=0.9)

validator_model = validator.fit(train_data)
```

Выполнение займет несколько минут или больше, в зависимости от быстродействия вашего оборудования, потому что нужно построить и оценить несколько моделей. Обратите внимание, что параметру коэффициента обучения присвоено значе-

ние 0,9. Это означает, что `TrainValidationSplit` разделяет обучающие данные на выборки в соотношении 90% / 10%. Первая выборка применяется для обучения каждой модели. Оставшиеся 10% входных данных используются в качестве данных перекрестной проверки для оценки модели. Если мы сейчас отделяем часть данных для оценки, то зачем мы оставили 10% исходных данных в качестве тестовой выборки?

Если назначением проверочной выборки была оценка *параметров*, соответствующих обучающей выборке, то тестовая выборка предназначена для оценки *гиперпараметров*, которые «подходят» этой выборке. То есть тестовая выборка обеспечивает объективную оценку точности конечной выбранной модели и ее гиперпараметров.

Допустим, наша лучшая модель на выборке валидации демонстрирует точность 90%. Хотелось бы верить, что она обеспечит такую же точность на будущих (незнакомых) данных. Однако в построении моделей есть элемент случайности. По воле случая эта конкретная модель и ее оценка могли оказаться необычайно удачными. Лучшей модели могло просто повезти, поэтому оценка ее точности, скорее всего, излишне оптимистична. Иными словами, гиперпараметры тоже могут переобучиться.

Чтобы представить, насколько хорошо эта лучшая модель сможет работать на будущих незнакомых данных, нам нужно оценить ее на выборке, которая не использовалась для обучения. И также следует избегать выборки валидации, которая применялась для оценки при подборе параметров модели. Вот почему была заранее приготовлена третья выборка — тестовая.

Результат работы валидатора содержит наилучшую найденную им модель. Фактически, это представление лучшего общего конвейера, который он нашел, потому что мы передавали ему экземпляр конвейера для запуска. Чтобы получить параметры, выбранные `DecisionTreeClassifier`, необходимо вручную извлечь `DecisionTreeClassificationModel` из полученной `PipelineModel`, которая является последним этапом конвейера:

```
from pprint import pprint

best_model = validator_model.bestModel
pprint(best_model.stages[1].extractParamMap())
...
{Param(...name='predictionCol', doc='prediction column name.'):
  'prediction',
  Param(...name='probabilityCol', doc='...'): 'probability',
  [...]
  Param(...name='impurity', doc='...'): 'entropy',
  Param(...name='maxDepth', doc='...'): 20,
  Param(...name='minInfoGain', doc='...'): 0.0,
  [...]
  Param(...name='featuresCol', doc='features column name.'):
  'featureVector',
```

```
  Param(...name='maxBins', doc='...'): 40,
  [...]
  Param(...name='labelCol', doc='label column name.'): 'Cover_Type'}
  ...
}
```

Этот вывод содержит много информации о подобранной модели, но он также показывает, что энтропия, по-видимому, лучше всего работает в качестве меры примеси и что максимальная глубина 20 вполне ожидаемо лучше, чем 1. Может показаться удивительным, что наилучшая модель соответствует всего 40 бинам, но это скорее говорит о том, что 40 «достаточно много», а не «лучше», чем 300. Наконец, модель с отсутствием минимального прироста информации была лучше, чем с небольшим приростом. Этот факт может говорить о том, что модель более склонна к недообучению, чем к переобучению.

Возникает вопрос, можно ли увидеть точность, достигнутую каждой из моделей для каждой комбинации гиперпараметров? Гиперпараметры и оценки предоставляются методами `getEstimatorParamMaps` и `validationMetrics` соответственно. Их вывод можно объединить для отображения всех комбинаций параметров, отсортированных по значению метрики:

```
validator_model = validator.fit(train_data)

metrics = validator_model.validationMetrics
params = validator_model.getEstimatorParamMaps()
metrics_and_params = list(zip(metrics, params))

metrics_and_params.sort(key=lambda x: x[0], reverse=True)
metrics_and_params

...
[(0.9130409881445563,
  {Param(...name='minInfoGain' ...): 0.0,
   Param(...name='maxDepth'...): 20,
   Param(...name='maxBins' ...): 40,
   Param(...name='impurity'...): 'entropy'}),
 (0.9112655352131498,
  {Param(...name='minInfoGain',...): 0.0,
   Param(...name='maxDepth' ...): 20,
   Param(...name='maxBins'...): 300,
   Param(...name='impurity'...: 'entropy'}),
...
```

Какой точности достигла эта модель на выборке валидации? И, наконец, какой точности модель достигает на тестовой выборке?

```
metrics.sort(reverse=True)
print(metrics[0])
...
```

```
0.9130409881445563
...

multiclassEval.evaluate(best_model.transform(test_data))❶
...
0.9138921373048084
```

❶ `best_Model` — это полный конвейер.

Оба результата близки к 91%. Случается, что оценка на выборке валидации изначально оказывается довольно хорошей. Стоит отметить, что оценка на тестовой выборке обычно не сильно отличается.

Это подходящий момент, чтобы вернуться к проблеме переобучения. Как мы говорили ранее, можно построить настолько глубокое и сложное дерево решений, что оно будет очень хорошо или идеально соответствовать обучающей выборке, но не сможет обобщаться на другие выборки, потому что оно слишком точно соответствует особенностям и шуму обучающих данных. Это проблема, присущая большинству алгоритмов машинного обучения, а не только деревьям решений.

Когда дерево решений переобучено, оно будет демонстрировать высокую точность при работе с данными из той же выборки, на которой обучалась модель, но низкую точность на других выборках. Здесь точность окончательной модели составила около 91% на других, новых данных. Для сравнения можно так же легко оценить точность модели на обучающих данных `trainData`. Это дает точность около 95%. Разница невелика, но предполагает, что дерево решений в некоторой степени переобучено. Возможно, имеет смысл слегка уменьшить максимальную глубину дерева.

До сих пор мы неявно обрабатывали все входные признаки, включая категориальные, как если бы они были числовыми. Можем ли мы еще больше улучшить точность нашей модели, рассматривая категориальные признаки именно как категории? Мы рассмотрим этот вопрос далее.

Еще раз о категориальных признаках

Категориальные признаки в нашем наборе данных представлены унитарным кодом как несколько двоичных значений 0/1. Мы можем обрабатывать эти отдельные признаки как числа, потому что любое правило принятия решения для «числовых» признаков будет использовать пороги 0 и 1, и все они эквивалентны, поскольку все значения равны 0 или 1.

Подобное кодирование заставляет алгоритм дерева решений рассматривать значения базовых категориальных признаков по отдельности. Поскольку такие признаки, как тип почвы, разбиты на множество составляющих признаков, а деревья решений рассматривают признаки независимо, нам сложно учитывать информацию о связях между типами почвы.

Например, девять различных типов почвы на самом деле являются частью одного семейства Лейтона, и модель будет работать лучше, если дерево решений сможет использовать эту информацию. Если бы тип почвы был закодирован как один категориальный признак с 40 значениями почвы, то дерево могло бы напрямую выразить правило «*если тип почвы входит в семейство Лейтона*». С другой стороны, при кодировании 40 независимых признаков дерево могло бы выучить последовательность из девяти решений о типе почвы, и сделать то же самое. Такое дерево потенциально может вырабатывать более точные решения.

Однако наличие 40 отдельных числовых признаков, представляющих один 40-значный категориальный признак, увеличивает использование памяти и замедляет работу.

Что если мы откажемся от унитарного кодирования? Давайте, например, заменим четыре столбца, кодирующие тип дикой природы, одним столбцом `Cover_Type`, который кодирует тип дикой природы как число от 0 до 3:

```
def unencode_one_hot(data):
    wilderness_cols = ['Wilderness_Area_' + str(i) for i in range(4)]
    wilderness_assembler = VectorAssembler().\
                           setInputCols(wilderness_cols).\
                           setOutputCol("wilderness")

    unhot_udf = udf(lambda v: v.toArray().tolist().index(1))❶

    with_wilderness = wilderness_assembler.transform(data).\
      drop(*wilderness_cols).\ ❷
      withColumn("wilderness", unhot_udf(col("wilderness")))

    soil_cols = ['Soil_Type_' + str(i) for i in range(40)]
    soil_assembler = VectorAssembler().\
                     setInputCols(soil_cols).\
                     setOutputCol("soil")
    with_soil = soil_assembler.\
                transform(with_wilderness).\
                drop(*soil_cols).\
                withColumn("soil", unhot_udf(col("soil")))

    return with_soil
```

❶ Обратите внимание на определение UDF.

❷ Отбрасываем столбцы унитарного кода, они больше не нужны.

В этом коде `VectorAssembler` применяется для объединения 4 и 40 столбцов типа дикой природы и почвы в два столбца `Vector`. Все значения в этих векторах равны 0, за исключением одного, которое равно 1. В API DataFrame для этого нет простой

функции, поэтому мы должны определить нашу собственную UDF, которую можно использовать для работы со столбцами. Она превращает эти два новых столбца в числа именно того типа, который нам нужен.

Теперь мы можем преобразовать набор данных, удалив унитарное кодирование с помощью нашей функции, определенной выше:

```
unenc_train_data = unencode_one_hot(train_data)
unenc_train_data.printSchema()
...
root
 |-- Elevation: integer (nullable = true)
 |-- Aspect: integer (nullable = true)
 |-- Slope: integer (nullable = true)
 |-- Horizontal_Distance_To_Hydrology: integer (nullable = true)
 |-- Vertical_Distance_To_Hydrology: integer (nullable = true)
 |-- Horizontal_Distance_To_Roadways: integer (nullable = true)
 |-- Hillshade_9am: integer (nullable = true)
 |-- Hillshade_Noon: integer (nullable = true)
 |-- Hillshade_3pm: integer (nullable = true)
 |-- Horizontal_Distance_To_Fire_Points: integer (nullable = true)
 |-- Cover_Type: double (nullable = true)
 |-- wilderness: string (nullable = true)
 |-- soil: string (nullable = true)
...

unenc_train_data.groupBy('wilderness').count().show()
...

+----------+------+
|wilderness| count|
+----------+------+
|         3| 33271|
|         0|234532|
|         1| 26917|
|         2|228144|
+----------+------+
```

Далее можно использовать почти тот же процесс, что и раньше, для настройки гиперпараметров модели дерева решений, построенной на этих данных, а также для выбора и оценки лучшей модели. Однако есть одно важное отличие. В двух новых числовых столбцах нет ничего, что указывало бы на то, что они на самом деле представляют собой категориальные значения. Относиться к ним как к числам некорректно, так как их порядок не имеет смысла. Модель по-прежнему будет построена, но из-за недоступности некоторой информации в этих признаках точность может пострадать.

К счастью, MLlib может хранить дополнительные метаданные о каждом столбце. Метаданные обычно скрыты от вызывающей функции, но содержат такую информацию, как кодирует ли столбец категориальное значение и сколько различных значений он принимает. Чтобы добавить эти метаданные, необходимо пропустить данные через `VectorIndexer`. Его задача состоит в том, чтобы превратить входные данные в правильно помеченные столбцы категориальных признаков. Мы уже проделали большую часть работы по превращению категориальных признаков в числа, а `VectorIndexer` позаботится о метаданных.

Нам нужно добавить этот этап в `Pipeline`:

```
from pyspark.ml.feature import VectorIndexer

cols = unenc_train_data.columns
inputCols = [c for c in cols if c!='Cover_Type']

assembler = VectorAssembler().setInputCols(inputCols).setOutputCol("featureVector")

indexer = VectorIndexer().\
  setMaxCategories(40).\❶
  setInputCol("featureVector").setOutputCol("indexedVector")

classifier = DecisionTreeClassifier().setLabelCol("Cover_Type").\
                                      setFeaturesCol("indexedVector").\
                                      setPredictionCol("prediction")

pipeline = Pipeline().setStages([assembler, indexer, classifier])
```

❶ ≥40, потому что у типов почвы 40 значений.

Этот подход предполагает, что обучающая выборка содержит все возможные значения каждого из категориальных признаков хотя бы один раз. То есть он работает правильно только в том случае, если в обучающей выборке присутствуют все 4 значения почвы и все 40 значений дикой природы, так что все возможные значения получают сопоставление. В нашем случае это так, но условие может не выполняться для небольших обучающих выборок, в которых некоторые метки появляются очень редко. В таких случаях может потребоваться вручную создать и добавить `VectorIndexerModel` с полным сопоставлением значений, предоставленных в явном виде.

В остальном процесс обучения и выбора модели такой же, как и раньше. Вы обнаружите, что валидатор снова выбрал лучшую модель, но точность на тестовой выборке составляет около 93%. Обрабатывая категориальные признаки как фактические категории, а не как числа, классификатор повысил свою точность почти на 2%.

Итак, мы обучили и настроили дерево решений. Теперь мы перейдем к более мощному алгоритму случайных лесов. Как вы увидите в следующем разделе, реализовать их с помощью PySpark будет на удивление легко.

Случайные леса

Если вы выполняли примеры кода, то могли заметить, что ваши результаты немного отличаются от результатов, представленных в листингах кода в книге. Это связано с тем, что в построении деревьев решений есть элемент случайности, и случайность вступает в игру, когда вы решаете, какие данные использовать и какие правила принятия решений исследовать.

Алгоритм не рассматривает все возможные решающие правила на каждом уровне. Для этого потребуется невероятно большое количество времени. Для категориального признака, имеющего N значений, существует $2^N - 2$ возможных правил принятия решений (каждое подмножество, кроме пустого множества и полного множества). И даже для умеренно большого N это порождает миллиарды возможных правил принятия решений.

Вместо этого деревья решений используют несколько эвристик (взаимосвязанных систематизированных приемов решения задач), чтобы определить, какие правила следует учитывать. Процесс выбора правил также содержит элемент случайности; каждый раз просматриваются только несколько признаков, выбранных случайным образом, и только значения из случайного подмножества обучающих данных. Такой подход жертвует точностью в пользу скорости, но это также означает, что алгоритм дерева решений не будет каждый раз строить одно и то же дерево. На самом деле, это хорошо.

Почему? По той же причине, по которой *мудрость толпы* обычно превосходит индивидуальные прогнозы. Чтобы понять суть этого явления, пройдите небольшой тест: сколько такси черного цвета работает в Лондоне?

Не смотрите на ответ, сначала угадайте.

Я предположил 10 000, что далеко от правильного ответа около 19 000. Поскольку мое предположение меньше правильного ответа, а ваше предположение может оказаться больше, среднее значение наших ответов будет более точным. Фактически, это регрессия к среднему. Среднее значение неофициального опроса 13 человек в моем офисе действительно было ближе к правильному ответу: 11 170.

Ключом к эффекту мудрости толпы является то, что догадки были независимыми и не влияли друг на друга. (Вы не подглядывали, правда?) Упражнение было бы бесполезным, если бы мы все сговорились и использовали одну и ту же методологию, чтобы сделать предположение, потому что предположения были бы одним и тем же ответом — одним и тем же потенциально совершенно неправильным ответом. Было бы только хуже, если бы я повлиял на вас, изложив свою догадку заранее.

Было бы здорово иметь не одно дерево, а много деревьев, каждое из которых дает разумные, но разные и независимые оценки правильного целевого значения. Их коллективный средний прогноз должен быть ближе к истинному ответу, чем прогноз любого отдельного дерева. Именно случайность в процессе построения деревьев помогает создать эту независимость. В этом суть *случайных лесов*.

Случайность вводится путем построения множества деревьев, каждое из которых видит разные случайные подмножества данных и даже признаков. Это делает лес в целом менее склонным к переобучению. Если конкретный признак содержит зашумленные данные или является обманчиво прогнозирующим только в обучающей выборке, то большинство деревьев не будут учитывать этот проблемный признак. Таким образом, большинство деревьев не будут подвержены шуму и смогут «перевесить» деревья, которые зашумлены.

Предсказание случайного леса — это просто средневзвешенное значение предсказаний деревьев. Для категориальной цели это может быть большинство голосов или наиболее вероятное значение, основанное на среднем значении вероятностей, полученных деревьями. Случайные леса, как и деревья решений, также поддерживают регрессию, и прогноз леса в этом случае представляет собой среднее значение предсказаний каждого дерева.

Хотя случайные леса — более мощный и сложный метод классификации, хорошая новость заключается в том, что их использование практически ничем не отличается от конвейера, разработанного ранее в этой главе. Просто добавьте `RandomForestClassifier` вместо `DecisionTreeClassifier` и действуйте, как раньше. Вам не потребуется какой-то новый код или API:

```
from pyspark.ml.classification import RandomForestClassifier

classifier = RandomForestClassifier(seed=1234, labelCol="Cover_Type",
                                    featuresCol="indexedVector",
                                    predictionCol="prediction")
```

Обратите внимание, что у этого классификатора есть еще один гиперпараметр: количество деревьев, которые нужно построить. Как и в случае с гиперпараметром максимального количества бинов, более высокие значения будут давать лучшие результаты лишь до определенного момента. Однако за точность приходится платить тем, что построение множества деревьев, конечно же, занимает во много раз больше времени, чем строительство одного дерева.

Точность лучшей модели случайного леса, созданной с помощью аналогичного процесса подбора гиперпараметров, составляет 95% от эталона — примерно на 2% лучше предыдущего результата. С другой стороны, это снижение частоты ошибок на 28% по сравнению с лучшим деревом решений, построенным ранее (с 7% до 5%). Продолжая настройку гиперпараметров, вы можете улучшить результат.

Между прочим, на данный момент у нас есть более достоверная картина важности признаков:

```
forest_model = best_model.stages[1]

feature_importance_list = list(zip(input_cols,
                           forest_model.featureImportances.toArray()))
feature_importance_list.sort(key=lambda x: x[1], reverse=True)

pprint(feature_importance_list)
...
```

```
(0.28877055118903183,Elevation)
(0.17288279582959612,soil)
(0.12105056811661499,Horizontal_Distance_To_Roadways)
(0.1121550648692802,Horizontal_Distance_To_Fire_Points)
(0.08805270405239551,wilderness)
(0.04467393191338021,Vertical_Distance_To_Hydrology)
(0.04293099150373547,Horizontal_Distance_To_Hydrology)
(0.03149644050848614,Hillshade_Noon)
(0.02840848357813760 5,Hillshade_9am)
(0.027185325937200706,Aspect)
(0.027075578474331806,Hillshade_3pm)
(0.015317564027809389,Slope)
```

Случайные леса привлекательны в контексте больших данных, поскольку деревья должны строиться независимо, а такие инструменты, как Spark и MapReduce, по своей сути нуждаются в параллельных задачах, где части общего решения могут быть вычислены независимо от частей данных. Тот факт, что деревья могут и должны обучаться только на подмножестве признаков или входных данных, упрощает распараллеливание построения деревьев.

Получение прогнозов

Построение классификатора является увлекательным процессом, но это не конечная цель. Мы стремимся получать прогнозы, и это относительно легко сделать.

Полученная лучшая модель на самом деле представляет собой целый конвейер операций. Он включает в себя преобразование данных для использования с моделью и саму модель, которая может делать прогнозы. Модель может работать с фреймом новых данных. Единственное отличие от объекта DataFrame, с которого мы начали, заключается в том, что в нем отсутствует столбец `Cover_Type`. Как говорил знаменитый физик Нильс Бор, когда мы делаем прогнозы — особенно о будущем, — результат, конечно же, неочевиден.

Чтобы доказать это, попробуйте удалить `Cover_Type` из тестовых данных и получить прогноз:

```
unenc_test_data = unencode_one_hot(test_data)
bestModel.transform(unenc_test_data.drop("Cover_Type")).\
                    select("prediction").show()
...
+----------+
|prediction|
+----------+
|       6.0|
+----------+
```

Мы получаем результат 6.0, что соответствует классу 7 в исходном наборе данных Covtype (исходные признаки проиндексированы с 1, а в программе применяется индексация с 0). Предсказанный тип покрова для земли, описанной в этом примере, — Krummholz.

Что дальше?

В этой главе представлены два взаимосвязанных и важных типа машинного обучения — классификация и регрессия, а также некоторые основополагающие концепции построения и настройки моделей: признаки, векторы, обучение и перекрестная проверка. Было продемонстрировано предсказание типа лесного покрова на основе таких данных, как местоположение и тип почвы, с использованием набора данных Covtype, при помощи деревьев и лесов решений, реализованных в PySpark.

Как и в случае с рекомендательными системами в *главе 3*, было бы полезно продолжить изучение влияния гиперпараметров на точность. Подбор гиперпараметров деревьев решений подразумевает поиск компромисса между скоростью и точностью. Большее количество бинов и деревьев, как правило, обеспечивает лучшую точность, но существует точка убывающей отдачи.

Наш классификатор оказался очень точным. Редко удается достичь точности более 95%. В целом, вы добьетесь дальнейшего повышения точности, включив больше признаков или преобразовав существующие признаки в более предсказательную форму. Это обычный повторяющийся шаг в итеративном улучшении модели классификатора. Например, в нашем наборе данных два признака, кодирующие горизонтальное и вертикальное расстояние до поверхностных вод, могут дать третий признак: расстояние до поверхностных вод по прямой. Он может оказаться более полезным, чем исходные признаки. Или, если бы удалось собрать больше данных, мы могли бы попытаться добавить новую информацию, такую, как влажность почвы.

Конечно, не все проблемы прогнозирования в реальном мире точно такие же, как набор данных Covtype. Например, некоторые задачи требуют предсказания непрерывного числового значения, а не дискретной категории. Большая часть рассмотренного нами конвейера применима и к этому типу задач регрессии; в таком случае будет полезен класс `RandomForestRegressor`.

Кроме того, деревья и леса решений — не единственные алгоритмы классификации или регрессии и не единственные, реализованные в PySpark. Каждый алгоритм работает совершенно иначе, чем деревья и леса решений. Однако многие моменты совпадают: они тоже включаются в конвейер и работают со столбцами в фрейме данных, а также имеют гиперпараметры, которые необходимо выбирать с помощью обучающих, проверочных и тестовых выборок входных данных. Те же самые общие принципы также могут быть применены для моделирования задач классификации и регрессии.

Мы рассмотрели примеры *обучения с учителем* — метода, который получил такое название, потому что модели показывают правильное значение на выходе, полученное от эксперта-«учителя», для каждого примера на входе. Что происходит, когда некоторые или все целевые значения неизвестны? В следующей главе мы расскажем, что можно сделать в этой ситуации.

Обнаружение аномалий при помощи кластеризации K-средних

Классификация и регрессия — мощные, хорошо изученные методы машинного обучения. В *главе 4* мы продемонстрировали использование классификатора в качестве предсказателя неизвестных значений. Но есть одна загвоздка: чтобы предсказать неизвестные значения для новых данных, нам нужно было знать правильные ответы для большого количества заранее приготовленных обучающих примеров. Классификаторы могут помочь только в том случае, если мы, специалисты по данным, знаем, что мы ищем, и можем предоставить обучаемой модели множество примеров сопоставления вход-выход, используя метод обучения с учителем.

Однако иногда правильный вывод неизвестен для некоторых или всех примеров. Возьмем, к примеру, задачу разделения клиентов сайта интернет-магазина по их покупательским привычкам и вкусам. Входными характеристиками являются их покупки, переходы по ссылкам, демографическая информация и многое другое. На выходе должны получиться группы покупателей: возможно, одна группа будет представлять покупателей, следящих за модой, другая будет соответствовать чувствительным к цене охотникам за скидками и так далее.

Если бы вас попросили определить эту целевую метку для каждого нового клиента, вы бы сразу столкнулись с проблемой при попытке применить к классификатору технику обучения с учителем: например, вы не знаете априори, кого следует считать следящим за модой. Начнем с того, что вовсе не обязательно «знатоки моды» — это значимая группа клиентов сайта!

К счастью, существуют методы *обучения без учителя*. Эти методы не позволяют предсказать целевое значение, потому что оно недоступно. Однако они могут изучать структуру данных и находить группы похожих входных данных или узнавать, какие типы входных данных могут встречаться, а какие нет. В этой главе будет представлено обучение без учителя с использованием кластеризации, реализованной в MLlib. В частности, мы будем использовать *алгоритм кластеризации K-средних* для обнаружения аномалий в данных сетевого трафика. Обнаружение аномалий часто используется для выявления попыток мошенничества, сетевых атак или проблем с серверами или другим оборудованием, оснащенным датчиками. В этих случаях важно иметь возможность находить новые типы аномалий, которые никогда раньше не наблюдались, — новые формы мошенничества, вторжений и режимы отказа серверов. Здесь как раз и пригодятся методы обучения без учителя, потому что они способны самостоятельно узнать, как выглядят обычные входные

данные, и, следовательно, определить, когда новые данные отличаются от обычных. Такие данные не обязательно являются атаками или мошенничеством; они просто необычные и поэтому заслуживают дальнейшего изучения.

Мы начнем с изучения основ алгоритма кластеризации K-средних. За этим последует знакомство с набором данных KDD Cup 1999. Затем мы создадим нашу первую модель K-средних, используя PySpark, и рассмотрим методы определения подходящего значения k — количества кластеров — при реализации алгоритма K-средних. Далее мы улучшим нашу модель, нормализовав входные признаки и используя ранее отброшенные категориальные признаки, при помощи метода унитарного кодирования. Завершая главу, мы рассмотрим метрику энтропии и оценим результаты, предоставленные нашей модели.

Кластеризация K-средних

Неотъемлемой проблемой обнаружения аномалий является, как следует из названия, обнаружение необычных вещей. Если бы мы знали, что означает «аномальный» для конкретного набора данных, мы могли бы легко обнаружить аномалии в данных с помощью обучения с учителем. Алгоритм машинного обучения будет получать входные данные, помеченные как «норма» и «аномалия», и учиться различать их. Однако природа аномалий такова, что они представляют собой *неизвестные неизвестные*[1]. Иными словами, наблюдаемая и понятая аномалия больше не является аномалией.

Кластеризация — самый известный тип обучения без учителя. Алгоритмы кластеризации пытаются найти естественные сходства в данных. Точки данных, которые похожи друг на друга, но отличаются от других, скорее всего, представляют собой осмысленную группу, поэтому алгоритмы кластеризации пытаются поместить такие данные в один и тот же кластер.

В свою очередь, кластеризация K-средних является, пожалуй, наиболее широко используемым алгоритмом кластеризации. Он пытается обнаружить k кластеров в наборе данных, где k задается специалистом по данным. k — это гиперпараметр модели, и правильное значение будет зависеть от набора данных. На самом деле, выбор подходящего значения для k будет главной сюжетной линией в этой главе.

Что означает «схожесть»? Когда набор данных содержит такую информацию, как активность клиентов или банковские транзакции? Алгоритм K-средних опирается на понятия *расстояния между точками данных*. Обычно для измерения расстояния между точками данных в алгоритме K-средних используется простое евклидово расстояние, но это одна из двух функций расстояния, поддерживаемых Spark MLlib на момент написания книги, а другая — косинус. Евклидово расстояние вычисляют

[1] В отличие от *известных неизвестных*, например, x в уравнении. Мы точно знаем, что известная неизвестная x существует, знаем, что нужно найти ее значение, и можем проверить ответ. Относительно неизвестных неизвестных мы не знаем даже, существуют ли они. — *Примеч. пер.*

для точек данных, все характеристики которых являются числовыми. «Похожие» точки данных — это те, расстояние между которыми невелико.

С точки зрения алгоритма K-средних *кластер* — это просто точка: общий центр всех точек, составляющих кластер. На самом деле это не точки, а просто *векторы признаков*, содержащие все числовые признаки, и их можно назвать векторами. Однако сейчас нам удобнее считать их точками, потому что они рассматриваются как точки в евклидовом пространстве.

Этот центр называется *центроидом* кластера и представляет собой среднее арифметическое точек — отсюда и название *K-средних*. Для начала алгоритм выбирает некоторые точки данных в качестве начальных центроидов кластера. Затем каждую точку данных ассоциируют с ближайшим центроидом, образуя кластер. На следующем шаге для каждого кластера вычисляется новый центроид как среднее значение точек данных, только что присвоенных этому кластеру. Процесс итеративно повторяется.

Теперь мы рассмотрим практический пример, который показывает, как кластеризация K-средних помогает обнаружить потенциально аномальную активность в сети.

Выявление аномального сетевого трафика

Кибератаки не перестают мелькать в новостях. Некоторые атаки пытаются залить компьютер шквалом сетевого трафика, чтобы вытеснить законный трафик. В других случаях хакеры пытаются использовать недостатки в сетевом программном обеспечении для получения несанкционированного доступа к компьютеру. Когда компьютер подвергается бомбардировке трафиком, это сразу видно, а вот обнаружение эксплойта больше похоже на поиск иголки в невероятно большом стоге сена сетевых запросов.

Некоторые эксплойты следуют известным шаблонам. Например, быстрый перебор всех сетевых портов на машине не похож на поведение обычной программы. Тем не менее это типичный первый шаг для злоумышленника, который ищет службы, работающие на компьютере, в качестве потенциальных целей для атаки.

Если вы возьметесь регулярно подсчитывать количество сетевых портов, к которым обращается удаленный хост за короткое время, то напишете функцию, которая достаточно хорошо предсказывает атаку сканирования портов. Несколько обращений к портам — это норма; сотни указывают на атаку. То же самое касается обнаружения других типов атак по другим характеристикам сетевых подключений — количеству отправленных и полученных байтов, ошибкам TCP и так далее.

Но как быть с неизвестными неизвестными? Самой большой угрозой может быть та, которая еще никогда не была обнаружена и классифицирована. Частью выявления потенциальных сетевых вторжений является обнаружение аномалий. Это подключения, которые не являются очевидными атаками, но не похожи на подключения, которые наблюдались в прошлом.

В этом случае для обнаружения аномальных сетевых подключений как раз уместно использовать алгоритм обучения без учителя, такой, как K-средние. Этот алгоритм может кластеризовать подключения на основе статистических данных о каждом из них. Полученные кластеры сами по себе не интересны, но в совокупности они определяют типы подключений, похожих на прошлые подключения. Все точки данных, расположенные в отдалении от кластеров, потенциально указывают на аномалию. Кластеры интересны тем, что они определяют области *нормальных* подключений; все остальное необычно и потенциально аномально.

Набор данных Кубка KDD 1999 г.

Кубок KDD — это ежегодное соревнование по интеллектуальному анализу данных, организованное специальной группой Ассоциации вычислительной техники (ACM). Каждый год вместе с набором данных перед участниками ставили задачу машинного обучения и предлагали представить подробное описание своего решения. В 1999 г. речь шла о вторжении в сеть, и набор данных до сих пор доступен на веб-сайте KDD. Нам нужно будет загрузить файлы kddcupdata.data.gz и kddcup.info с веб-сайта. В оставшейся части этой главы мы рассмотрим создание системы для обнаружения аномального сетевого трафика с использованием Spark и набора данных, который мы скачали.

Не используйте этот набор данных для создания реальной системы защиты от проникновения в сеть! Данные не обязательно отражали реальный сетевой трафик, но даже если и отражали, то учтите — это было более 20 лет назад.

К счастью, организаторы конкурса уже преобразовали исходные данные сетевых пакетов в сводную информацию об отдельных сетевых подключениях. Набор данных имеет размер около 708 МБ и содержит около 4,9 млн подключений. Эти данные по современным меркам не огромные, но все же большие, и, безусловно, их вполне достаточно для решения наших задач. Для каждого подключения набор данных содержит такую информацию, как количество отправленных байтов, количество попыток входа в систему, ошибки TCP и т. д. Каждое подключение представляет собой одну строку данных в формате CSV, содержащую 38 признаков. Информацию о признаках и их порядок можно найти в файле kddcup.info.

Разархивируйте файл данных kddcup.data.gz и скопируйте его в свое хранилище. В этом примере, как и в других, предполагается, что файл доступен по адресу data/kddcup.data. Давайте посмотрим исходные на данные:

```
head -n 1 data/kddcup.data
...
0,tcp,http,SF,215,45076,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,1,...
```

В этой строке, например, представлено TCP-соединение со службой HTTP — было отправлено 215 байт, а получено 45 706 байт. Пользователь вошел в систему... и так далее.

Многие признаки являются результатами подсчета, как `num_file_creations` в 17-м столбце, как указано в файле kddcup.info. Многие признаки принимают значение 0

или 1, указывая на наличие или отсутствие поведения, как `su_attempted` в 15-м столбце. Они выглядят как категориальные признаки с унитарным кодированием из *главы 4*, но не сгруппированы и не связаны таким же образом. Каждый из них похож на признак «да/нет» и поэтому, возможно, является категориальной функцией. Не всегда допустимо рассматривать категориальные признаки как числа и обращаться с ними так, будто они имеют величину. Однако в особом случае бинарного категориального признака в большинстве алгоритмов машинного обучения будет хорошо работать его сопоставление с числовым признаком, принимающим значения 0 и 1.

Остальные признаки — это коэффициенты, такие, как `dst_host_srv_rerror_rate` в предпоследнем столбце, принимающие значения от 0,0 до 1,0 включительно.

Интересно, что в последнем поле указана метка. Большинство соединений помечены как обычные, но некоторые из них идентифицированы как примеры различных типов сетевых атак. Эта метка пригодилась бы при обучении с учителем, чтобы научиться отличать известную атаку от обычного соединения, но наша задача сейчас заключается в обнаружении аномалий и выявлении потенциально новых и неизвестных атак. Пока мы просто проигнорируем эту метку.

Первый опыт кластеризации

Откройте `pyspark-shell` и загрузите данные CSV в виде фрейма данных. Это снова файл CSV, но без информации заголовка. Необходимо указать имена столбцов, указанные в прилагаемом файле kddcup.info.

```
data_without_header = spark.read.option("inferSchema", True).\
                                 option("header", False).\
                                 csv("data/kddcup.data")

column_names = [ "duration", "protocol_type", "service", "flag",
  "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
  "hot", "num_failed_logins", "logged_in", "num_compromised",
  "root_shell", "su_attempted", "num_root", "num_file_creations",
  "num_shells", "num_access_files", "num_outbound_cmds",
  "is_host_login", "is_guest_login", "count", "srv_count",
  "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
  "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
  "dst_host_count", "dst_host_srv_count",
  "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
  "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
  "dst_host_serror_rate", "dst_host_srv_serror_rate",
  "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
  "label"]

data = data_without_header.toDF(*column_names)
```

Начнем с изучения набора данных. Какие метки присутствуют в данных и сколько их? Следующий код просто подсчитывает метки и выводит результаты в порядке убывания количества:

```
from pyspark.sql.functions import col
data.select("label").groupBy("label").count().\
      orderBy(col("count").desc()).show(25)
```

```
...
+----------------+-------+
|           label|  count|
+----------------+-------+
|          smurf.|2807886|
|        neptune.|1072017|
|         normal.| 972781|
|          satan.|  15892|
...
|            phf.|      4|
|           perl.|      3|
|            spy.|      2|
+----------------+-------+
```

Существует 23 различных метки, наиболее часто встречающиеся атаки — `smurf.` и `neptune`.

Обратите внимание, что данные содержат нечисловые признаки. Например, второй столбец может быть `tcp`, `udp` или `icmp`, но кластеризация K-средних требует числовых характеристик. Последний столбец метки также не является числовым. Для начала мы будем просто их игнорировать.

Помимо этого, создание кластеризации данных методом K-средних следует той же схеме, которая была рассмотрена в *главе 4*. `VectorAssembler` создает вектор признаков, реализация `KMeans` создает модель из векторов признаков, а `Pipeline` сшивает все это вместе. Из полученной модели можно извлечь и изучить центры кластеров.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans, KMeansModel
from pyspark.ml import Pipeline

numeric_only = data.drop("protocol_type", "service", "flag").cache()

assembler = VectorAssembler().setInputCols(numeric_only.columns[:-1]).\
                              setOutputCol("featureVector")

kmeans = KMeans().setPredictionCol("cluster").setFeaturesCol("featureVector")

pipeline = Pipeline().setStages([assembler, kmeans])
pipeline_model = pipeline.fit(numeric_only)
kmeans_model = pipeline_model.stages[1]
```

```
from pprint import pprint
pprint(kmeans_model.clusterCenters())

...
[array([4.83401949e+01, 1.83462155e+03, 8.26203190e+02, 5.71611720e-06,
       6.48779303e-04, 7.96173468e-06...]),
 array([1.0999000e+04, 0.0000000e+00, 1.3099374e+09, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00,...])]
```

Интерпретировать эти числа непросто, но каждое из них представляет собой центроид одного из кластеров, созданных моделью. Значения представляют собой координаты центроида с точки зрения каждой из числовых входных функций.

На печать выведены два вектора, значит, данные разбиты на $k = 2$ кластера. Для точного моделирования отдельных групп данных, то есть для сложного набора данных, который, как известно, демонстрирует, по крайней мере, 23 различных типа связей, этого, вероятнее всего, будет недостаточно.

У нас хорошая возможность использовать метки, чтобы понять, что вошло в эти два кластера. Для этого мы подсчитаем метки в каждом кластере.

```
with_cluster = pipeline_model.transform(numeric_only)
with_cluster.select("cluster", "label").groupBy("cluster", "label").count().\
orderBy(col("cluster"), col("count").desc()).show(25)
...
+-------+----------------+-------+
|cluster|           label| count|
+-------+----------------+-------+
|      0|          smurf.|2807886|
|      0|        neptune.|1072017|
|      0|         normal.| 972781|
|      0|          satan.|  15892|
|      0|        ipsweep.|  12481|
...
|      0|            phf.|      4|
|      0|           perl.|      3|
|      0|            spy.|      2|
|      1|      portsweep.|      1|
+-------+----------------+-------+
```

Результат показывает, что кластеризация совсем не помогла. Только одна точка данных оказалась в кластере 1!

Выбор параметра *k*

Двух кластеров явно недостаточно. Сколько кластеров подходит для этого набора данных? Ясно, что в данных есть 23 различных паттерна, поэтому кажется, что k может быть, как минимум, 23 или, вероятно, даже больше. Обычно пытаются найти наилучшее из множества значений k. Но что значит «наилучшее»?

Кластеризацию можно было бы считать хорошей, если бы каждая точка данных находилась вблизи своего ближайшего центроида, где «близость» определяется евклидовым расстоянием. Это простой и распространенный способ оценить качество кластеризации по среднему значению расстояний по всем точкам или иногда по среднему квадрату расстояний. На самом деле `KMeansModel` предлагает метод `ClusteringEvaluator`, который вычисляет сумму квадратов расстояний и может легко применяться для вычисления среднеквадратичного расстояния.

Достаточно просто вручную оценить стоимость кластеризации для нескольких значений *k*. Имейте в виду, что выполнение этого кода может занять 10 минут и более:

```
from pyspark.sql import DataFrame
from pyspark.ml.evaluation import ClusteringEvaluator

from random import randint

def clustering_score(input_data, k):
    input_numeric_only = input_data.drop("protocol_type", "service",
                                         "flag")
    assembler = VectorAssembler().setInputCols(input_numeric_only.columns[:-1]).\
                              setOutputCol("featureVector")
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).\
                              setPredictionCol("cluster").\
                              setFeaturesCol("featureVector")
    pipeline = Pipeline().setStages([assembler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)

    evaluator = ClusteringEvaluator(predictionCol='cluster',
                                    featuresCol="featureVector")
    predictions = pipeline_model.transform(numeric_only)
    score = evaluator.evaluate(predictions)
    return score

for k in list(range(20,100, 20)):
    print(clustering_score(numeric_only, k))❶

...
(20,6.649218115128446E7)
(40,2.5031424366033625E7)
(60,1.027261913057096E7)
(80,1.2514131711109027E7)
(100,7235531.565096531)
```

❶ Вывод на печать расстояния в экспоненциальном представлении.

Распечатка результатов показывает, что расстояние уменьшается по мере увеличения *k*. Обратите внимание, что оценки показаны в экспоненциальном представлении: первое значение больше 10^7, а не чуть больше 6.

В этом случае ваши значения тоже будут несколько отличаться. Кластеризация зависит от случайно выбранного начального набора центроидов.

Впрочем, результат очевиден. По мере добавления новых кластеров всегда должна быть возможность поместить точки данных ближе к ближайшему центроиду. На самом деле, если k выбрано равным количеству точек данных, среднее расстояние будет равно 0, потому что каждая точка будет отдельным кластером для самой себя!

Более того, в предыдущих результатах расстояние для $k = 80$ больше, чем для $k = 60$. Этого не должно происходить, потому что более высокое k всегда обеспечивает не менее хорошую кластеризацию, чем более низкое k. Проблема в том, что алгоритм K-средних не обязательно находит оптимальную кластеризацию для данного k. Его итерационный процесс может сойтись от случайной начальной точки к локальному минимуму, который хорош, но не оптимален.

Это по-прежнему верно, даже когда для выбора начальных центроидов применяются более интеллектуальные методы. K-средние++ и K-средние|| являются вариантами алгоритмов отбора, которые с большей вероятностью выбирают разнообразные, разделенные центроиды и более надежно приводят к хорошей кластеризации. Spark MLlib, по сути, реализует K-средние||. Однако все они по-прежнему имеют элемент случайности в выборе начальных центроидов и не могут гарантировать оптимальную кластеризацию.

Случайный начальный набор кластеров, выбранный для $k = 80$, возможно, привел к особенно субоптимальной кластеризации, или остановился раньше, чем достиг своего локального оптимума.

Мы можем улучшить его, запустив более долгую итерацию. Алгоритм получает через `setTol` пороговое значение, который задает минимальную величину смещения центроида кластера, считающуюся значительной; более низкие значения означают, что алгоритм K-средних позволит центроидам продолжать двигаться дольше. Увеличение максимального количества итераций с помощью `setMaxIter` также предотвращает преждевременную остановку кластеризации за счет возможных дополнительных вычислений.

```
def clustering_score_1(input_data, k):
    input_numeric_only = input_data.drop("protocol_type", "service",  "flag")
    assembler = VectorAssembler().\
                setInputCols(input_numeric_only.columns[:-1]).\
                setOutputCol("featureVector")
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).\❶
      setTol(1.0e-5).\❷
      setPredictionCol("cluster").setFeaturesCol("featureVector")
    pipeline = Pipeline().setStages([assembler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    #
    evaluator = ClusteringEvaluator(predictionCol='cluster',
    featuresCol="featureVector")
```

```
    predictions = pipeline_model.transform(numeric_only)
    score = evaluator.evaluate(predictions)
    #
    return score

for k in list(range(20,101, 20)):
    print(k, clustering_score_1(numeric_only, k))
```

❶ Увеличиваем относительно значения по умолчанию `20`.

❷ Уменьшаем относительно значения по умолчанию `1.0e-4`.

На этот раз расстояние убывает относительно равномерно:

```
(20,1.8041795813813403E8)
(40,6.33056876207124E7)
(60,9474961.544965891)
(80,9388117.93747141)
(100,8783628.926311461)
```

Мы хотим найти точку, после которой увеличение *k* перестает сильно снижать расстояние — так называемый «локоть» на графике зависимости расстояния от *k*, который обычно убывает, но в конечном итоге выравнивается. Здесь он, кажется, продолжает заметно уменьшаться после 100. Значит, правильное значение *k* может быть больше 100.

Визуализация с помощью SparkR

Перед повторной кластеризацией, возможно, будет полезнее сделать шаг назад и узнать побольше о данных. В частности, может быть полезно посмотреть на визуальное представление точек данных.

Сам Spark не оснащен инструментами визуализации, но популярная статистическая среда с открытым исходным кодом R имеет библиотеки как для исследования, так и для визуализации данных. Кроме того, Spark также обеспечивает базовую интеграцию с R через SparkR. В этом разделе мы кратко продемонстрируем использование R и SparkR для кластеризации данных и изучения полученных кластеров.

SparkR — это разновидность оболочки `spark-shell`, которая используется в этой книге и вызывается с помощью команды `sparkR`. Она запускает локальный интерпретатор R, аналогично тому, как `spark-shell` запускает вариант оболочки Scala. Машине, на которой выполняется `sparkR`, требуется локально установленный интерпретатор R, отсутствующий в составе пакета Spark. Его можно установить, например, с помощью команды `sudo apt-get install r-base` в дистрибутивах Linux, таких, как Ubuntu, или `brew install R` с помощью Homebrew в macOS.

SparkR — это среда оболочки командной строки, аналогичная R. Для просмотра визуализаций необходимо запускать эти команды в IDE-подобной среде, которая может отображать изображения. RStudio — это IDE для R (совместима со SparkR);

она работает в операционной системе обычного компьютера, поэтому здесь его можно будет использовать только в том случае, если вы экспериментируете со Spark локально, а не в кластере.

Если вы используете Spark локально, загрузите бесплатную версию RStudio и установите ее. Если нет, то остальную часть этого примера все еще можно запустить с помощью `sparkR` в командной строке, например, в кластере. Отметим, что визуализацию таким образом отображать будет невозможно.

Если вы работаете через RStudio, запустите IDE и настройте переменные `SPARK_HOME` и `JAVA_HOME`, если они еще не настроены в вашей локальной среде, чтобы они указывали на каталоги установки Spark и JDK соответственно:

```
Sys.setenv(SPARK_HOME = "/path/to/spark")❶
Sys.setenv(JAVA_HOME = "/path/to/java")
library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "local[*]",
sparkConfig = list(spark.driver.memory = "4g"))
```

❶ Разумеется, надо подставить реальные пути на вашем рабочем компьютере.

Эти шаги не нужны, если вы запускаете `sparkR` в командной строке. Вместо этого нужно просто передать параметры конфигурации командной строки, такие, как `--driver-memory`, как вы это делали в `spark-shell`.

SparkR — это, фактически, обертка на языке R для тех же API-интерфейсов DataFrame и MLlib, которые были продемонстрированы в этой главе. Следовательно, мы можем легко повторить простую кластеризацию данных методом K-средних:

```
clusters_data <- read.df("/path/to/kddcup.data", "csv",❶
                         inferSchema = "true", header = "false")
colnames(clusters_data) <- c(❷
  "duration", "protocol_type", "service", "flag",
  "src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent",
  "hot", "num_failed_logins", "logged_in", "num_compromised",
  "root_shell", "su_attempted", "num_root", "num_file_creations",
  "num_shells", "num_access_files", "num_outbound_cmds",
  "is_host_login", "is_guest_login", "count", "srv_count",
  "serror_rate", "srv_serror_rate", "rerror_rate", "srv_rerror_rate",
  "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate",
  "dst_host_count", "dst_host_srv_count",
  "dst_host_same_srv_rate", "dst_host_diff_srv_rate",
  "dst_host_same_src_port_rate", "dst_host_srv_diff_host_rate",
  "dst_host_serror_rate", "dst_host_srv_serror_rate",
  "dst_host_rerror_rate", "dst_host_srv_rerror_rate",
  "label")

numeric_only <- cache(drop(clusters_data,❸
                     c("protocol_type", "service", "flag", "label")))
```

```
kmeans_model <- spark.kmeans(numeric_only, ~ .,❹
                             k = 100, maxIter = 40, initMode = "k-means||")
```

❶ Замените на путь к `kddcup.data`.

❷ Присваиваем имена столбцам.

❸ Снова удаляем нечисловые столбцы.

❹ `~ .` означает все столбцы.

Теперь не составит труда назначить кластер каждой точке данных. Приведенные выше операции показывают использование API-интерфейсов SparkR, которые естественным образом соответствуют основным API-интерфейсам Spark, но выражаются в виде библиотек R с синтаксисом, подобным R. Фактическая кластеризация выполняется с использованием все той же реализации языка Scala на основе JVM в MLlib. Эти операции по сути являются дескриптором или псевдонимом для распределенных операций, которые не выполняются в R.

R имеет свой собственный богатый набор библиотек для анализа и собственную аналогичную концепцию фрейма данных. Поэтому иногда бывает полезно передать данные в интерпретатор R, чтобы иметь возможность использовать эти библиотеки R, не связанные со Spark.

Конечно, R и его библиотеки не поддерживают распределенную обработку, поэтому невозможно загрузить весь набор данных из 4 898 431 точки данных в R. Однако легко получить выборку:

```
clustering <- predict(kmeans_model, numeric_only)
clustering_sample <- collect(sample(clustering, FALSE, 0.01))❶

str(clustering_sample)

...
'data.frame': 48984 obs. of 39 variables:
  $ duration : int 0 0 0 0 0 0 0 0 0 0 ...
  $ src_bytes : int 181 185 162 254 282 310 212 214 181 ...
  $ dst_bytes : int 5450 9020 4528 849 424 1981 2917 3404 ...
  $ land : int 0 0 0 0 0 0 0 0 0 0 ...
...
  $ prediction : int 33 33 33 0 0 0 0 0 33 33 ...
```

❶ Выборка 1% без замены.

`clustering_sample` на самом деле является локальным фреймом данных R, а не фреймом данных Spark, поэтому им можно оперировать, как и любыми другими данными в R. В коде выше `str` показывает структуру фрейма данных.

Например, можно извлечь привязку кластера, а затем показать статистику распределения привязок:

```
clusters <- clustering_sample["prediction"]❶
data <- data.matrix(within(clustering_sample, rm("prediction")))❷
```

```
table(clusters)

...
clusters
    0 11 14 18 23  25  28 30 31   33 36 ...
47294  3  1  2  2 308 105  1 27 1219 15 ...
```

❶ Только столбец привязок кластеризации.

❷ Все, кроме привязок кластеризации.

Например, этот вывод показывает, что большинство точек попало в кластер 0. Хотя с этими данными в R можно сделать гораздо больше, дальнейшее рассмотрение этого вопроса выходит за рамки данной книги.

Для визуализации данных требуется библиотека rgl. Она будет работать только при запуске этого примера в RStudio. Сначала установите (один раз) и загрузите библиотеку:

```
install.packages("rgl")
library(rgl)
```

Имейте в виду, что R может предложить вам загрузить другие пакеты или инструменты компилятора для завершения установки, поскольку установка пакета означает компиляцию его исходного кода.

Наш набор данных является 38-мерным. Его нужно будет спроецировать не более чем в три измерения, чтобы визуализировать со *случайной проекцией*:

```
random_projection <- matrix(data = rnorm(3*ncol(data)), ncol = 3)❶
random_projection_norm <-
  random_projection / sqrt(rowSums(random_projection*random_projection))

projected_data <- data.frame(data %*% random_projection_norm)❷
```

❶ Делаем случайную трехмерную проекцию и нормализуем ее.

❷ Проецируем и создаем новый фрейм данных.

Этот код создает трехмерный набор данных из 38-мерного набора данных путем выбора трех случайных единичных векторов и проецирования данных на них. Это упрощенная, грубая форма уменьшения размерности. Конечно, существуют более сложные алгоритмы уменьшения размерности, такие, как анализ основных компонентов или разложение по сингулярным числам. Они доступны в R, но их выполнение занимает гораздо больше времени. Для наглядности в этом примере случайная проекция дает почти такой же результат, но делает это быстрее.

Наконец, кластеризованные точки можно отобразить в интерактивной трехмерной визуализации:

```
num_clusters <- max(clusters)
palette <- rainbow(num_clusters)
colors = sapply(clusters, function(c) palette[c])
plot3d(projected_data, col = colors, size = 10)
```

Обратите внимание, что для этого потребуется запустить RStudio в среде, поддерживающей библиотеку и графику rgl.

Полученная визуализация (рис. 5.1) отображает точки данных в трехмерном пространстве. Многие точки накладываются друг на друга, результат в целом разреженный, и его трудно интерпретировать. Однако доминирующей чертой визуализации является ее L-образная форма. Похоже, что точки сильно различаются только по двум измерениям и мало по остальным.

Это разумное предположение, потому что в наборе данных есть два признака, которые имеют гораздо больший диапазон значений, чем другие. В то время как большинство признаков имеют значения от 0 до 1, количество отправленных и полученных байтов варьируется от 0 до десятков тысяч. Таким образом, евклидово расстояние между точками почти полностью определяется этими двумя признаками, как будто других признаков не существует! Поэтому важно нормализовать эти различия в масштабе, чтобы поставить признаки в почти равные условия.

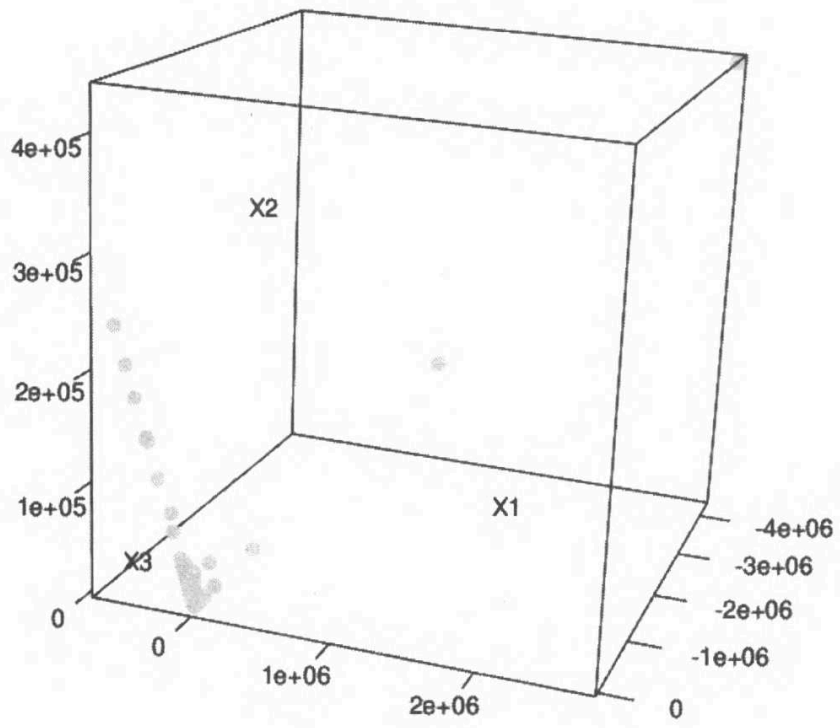


Рис. 5.1. Случайная трехмерная проекция

Нормализация признаков

Мы можем нормализовать каждый признак, преобразовав его в стандартную оценку. Для этого нужно вычесть среднее значение признака из каждого значения и разделить разность на стандартное отклонение, как показано в *стандартном уравнении оценки*:

$$normalized_i = \frac{feature_i - \mu_i}{\sigma_i}.$$

На самом деле вычитание среднего не влияет на кластеризацию, потому что оно эффективно сдвигает все точки данных на одинаковую величину в одном и том же направлении. Следовательно, вычитание среднего не влияет на межточечные евклидовы расстояния.

MLlib содержит `StandardScaler` — компонент, выполняющий упомянутую нормализацию. Его можно легко добавить в конвейер кластеризации.

Давайте запустим тот же тест с нормализованными данными в более высоком диапазоне *k*:

```
from pyspark.ml.feature import StandardScaler

def clustering_score_2(input_data, k):
    input_numeric_only = input_data.drop("protocol_type", "service",
                                         "flag")
    assembler = VectorAssembler().\
                setInputCols(input_numeric_only.columns[:-1]).\
                setOutputCol("featureVector")
    scaler = StandardScaler().setInputCol("featureVector").\
                              setOutputCol("scaledFeatureVector").\
                              setWithStd(True).setWithMean(False)
    kmeans = KMeans().setSeed(randint(100,100000)).\
                      setK(k).setMaxIter(40).\
                      setTol(1.0e-5).setPredictionCol("cluster").\
                      setFeaturesCol("scaledFeatureVector")
    pipeline = Pipeline().setStages([assembler, scaler, kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    #
    evaluator = ClusteringEvaluator(predictionCol='cluster',
    featuresCol="scaledFeatureVector")
    predictions = pipeline_model.transform(numeric_only)
    score = evaluator.evaluate(predictions)
    #
    return score

for k in list(range(60, 271, 30)):
    print(k, clustering_score_2(numeric_only, k))
...
(60,1.2454250178069293)
(90,0.7767730051608682)
(120,0.5070473497003614)
(150,0.4077081720067704)
(180,0.3344486714980788)
(210,0.276237617334138)
(240,0.24571877339169032)
(270,0.21818167354866858)
```

Нормализация помогла поставить измерения в более равное положение, и абсолютные расстояния между точками в абсолютном выражении стали намного меньше. Однако приведенный выше результат еще не дает очевидного значения k, за пределами которого его увеличение почти не делает результат лучше.

Как и ожидалось, новая трехмерная визуализация нормализованных точек данных показывает более богатую структуру. Некоторые точки расположены через регулярные дискретные интервалы в одном направлении; это, вероятнее всего, проекции дискретных измерений в таких данных, как подсчеты. Имея 100 кластеров, трудно понять, какие точки к каким кластерам относятся. Похоже, что один большой кластер доминирует, а остальные кластеры соответствуют небольшим компактным подобластям (некоторые из них опущены в увеличенном фрагменте всей трехмерной визуализации). Результат, показанный на рис. 5.2, не обязательно способствует анализу, но является интересной проверкой работоспособности.

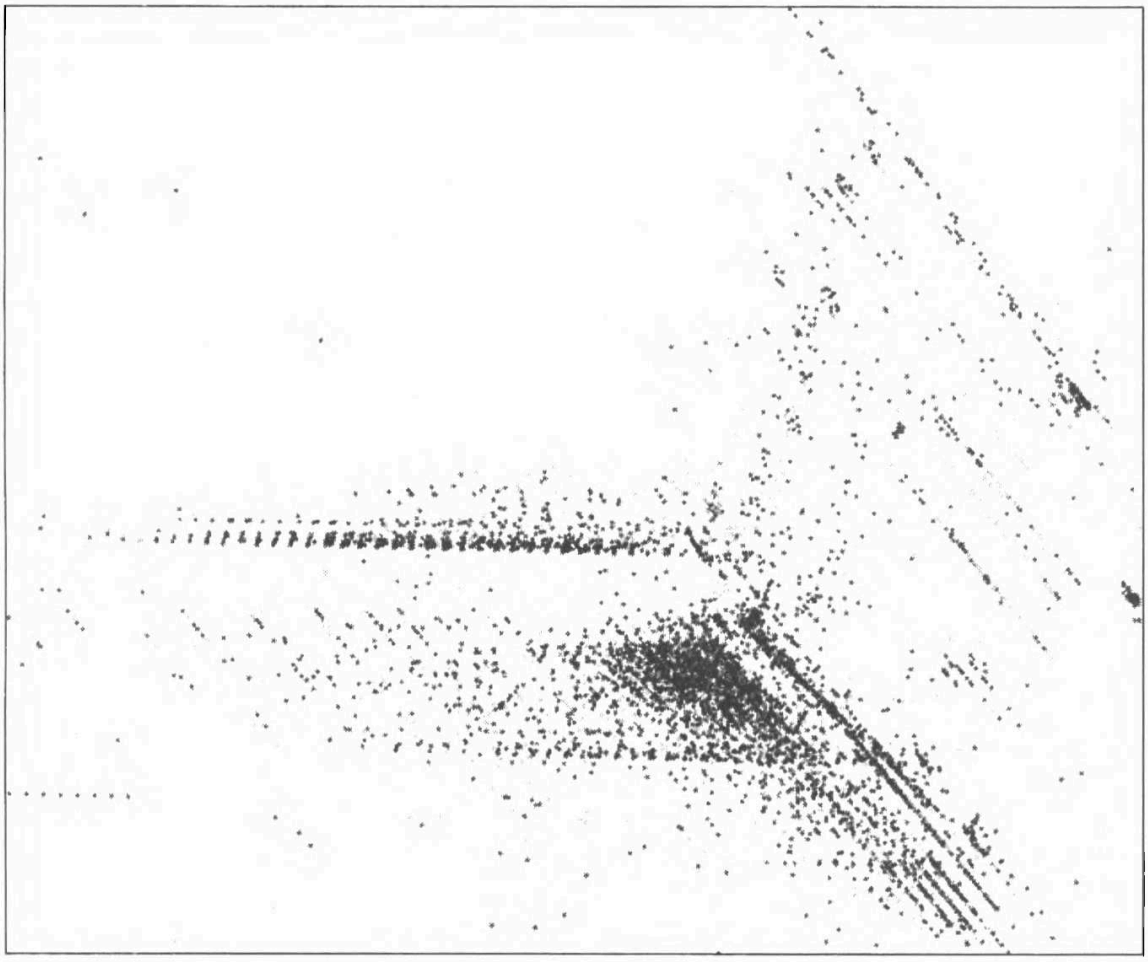


Рис. 5.2. Случайная трехмерная проекция после нормализации

Категориальные переменные

Нормализация стала ценным шагом вперед, но для улучшения кластеризации можно сделать еще больше. В частности, некоторые признаки были полностью исключены, поскольку они не являются числовыми. Это отбрасывание ценной информа-

ции. Добавление их обратно в той или иной форме должно привести к более информативной кластеризации.

Ранее были исключены три категориальных признака, поскольку нечисловые признаки нельзя использовать с функцией евклидова расстояния, которую алгоритм K-средних использует в MLlib. Это противоположно проблеме, отмеченной в *разд. «Случайные леса»*, где числовые признаки использовались для представления категориальных значений, но категориальный признак был желателен.

Категориальные признаки могут быть преобразованы в несколько признаков бинарных индикаторов с использованием прямого унитарного кодирования, которые можно рассматривать как числовые измерения. Например, второй столбец содержит тип протокола: `tcp`, `udp` или `icmp`. Этот признак можно рассматривать как совокупность трех признаков, как если бы в наборе данных были признаки «является TCP», «является UDP» и «является ICMP». Значение единственного признака `tcp` может стать равным `1,0,0`; `udp` может быть `0,1,0`; и так далее.

И для этого преобразования в MLlib тоже есть подходящие компоненты. Фактически унитарное кодирование строковых функций, таких, как `protocol_type`, на самом деле представляет собой двухэтапный процесс. Сначала строковые значения с помощью `StringIndexer` преобразуются в целые индексы, такие, как 0, 1, 2 и т. д.. Затем с помощью `OneHotEncoder` эти целочисленные индексы кодируются в вектор. Эти два шага можно рассматривать как небольшой конвейер `Pipeline`.

```
from pyspark.ml.feature import OneHotEncoder, StringIndexer

def one_hot_pipeline(input_col):
    indexer = StringIndexer().setInputCol(input_col).\
                              setOutputCol(input_col + "-_indexed")
    encoder = OneHotEncoder().setInputCol(input_col + "indexed").\
                              setOutputCol(input_col + "_vec")
    pipeline = Pipeline().setStages([indexer, encoder])
    return pipeline, input_col + "_vec" ❶
```

❶ Возвращает конвейер и имя столбца, который содержит выходной вектор.

Этот метод создает конвейер, который можно добавить в качестве компонента в общий конвейер кластеризации; конвейеры можно соединять последовательно. Все что осталось, — это добавить новые выходные столбцы вектора в выходные данные `VectorAssembler` и продолжить, как и раньше, масштабирование, кластеризацию и оценку. Исходный код здесь для краткости опущен, но его можно найти в архиве файлов этой книги.

```
(60,39.739250062068685)
(90,15.814341529964691)
(120,3.5008631362395413)
(150,2.2151974068685547)
(180,1.587330730808905)
(210,1.3626704802348888)
(240,1.1202477806210747)
(270,0.9263659836264369)
```

Исходя из этих результатов, можно предположить, что начиная с $k = 180$ отдача от увеличения числа центроидов немного выравнивается. По крайней мере, кластеризация теперь использует все входные признаки.

Использование меток с энтропией

Ранее мы использовали имеющуюся метку каждой точки данных для быстрой проверки качества кластеризации. Этот подход можно формализовать и использовать в качестве альтернативного средства оценки качества кластеризации и, следовательно, выбора k.

Метки говорят нам об истинной природе каждой точки данных. Очевидно, что хорошая кластеризация должна согласовываться с метками, присвоенными человеком. Она должна объединять в кластер точки, которые часто используют одну и ту же метку, а не точки с множеством разных меток. У нас должны получаться кластеры с относительно однородными метками.

Вы, возможно, помните из *разд. «Случайные леса»*, что у нас есть показатели однородности кластера: примесь Джини и энтропия. Это функции соотношения количества меток в каждом кластере, и они дают тем меньшее число, чем меньше разнообразие меток. Здесь для примера мы будем использовать энтропию:

```
from math import log

def entropy(counts):
    values = [c for c in counts if (c > 0)]
    n = sum(values)
    p = [v/n for v in values]
    return sum([-1*(p_v) * log(p_v) for p_v in p])
```

Хороший конвейер кластеризации должен давать кластеры, наборы меток которых однородны и, следовательно, имеют низкую энтропию. Тогда, средневзвешенное значение энтропии можно использовать в качестве оценки кластера:

```
from pyspark.sql import functions as fun
from pyspark.sql import Window

cluster_label = pipeline_model.\
                    transform(data).\
                    select("cluster", "label")❶

df = cluster_label.\
        groupBy("cluster", "label").\
        count().orderBy("cluster")❷

w = Window.partitionBy("cluster")

p_col = df['count'] / fun.sum(df['count']).over(w)
with_p_col = df.withColumn("p_col", p_col)
```

```
result = with_p_col.groupBy("cluster").\
             agg(-fun.sum(col("p_col") * fun.log2(col("p_col")))\
                       .alias("entropy"),
                 fun.sum(col("count"))\
                       .alias("cluster_size"))

result = result.withColumn('weightedClusterEntropy',
                        col('entropy') * col('cluster_size'))❸

weighted_cluster_entropy_avg = result.\
                           agg(fun.sum(
                             col('weightedClusterEntropy'))).\
                           collect()
weighted_cluster_entropy_avg[0][0]/data.count()
```

❶ Предсказываем кластер для каждого элемента данных.

❷ Количество меток на кластер.

❸ Средняя энтропия, взвешенная по размеру кластера.

Как и прежде, этот анализ можно использовать для получения некоторого представления о подходящем значении k. Энтропия не обязательно будет уменьшаться при увеличении k, поэтому можно искать локальное минимальное значение. Здесь снова результаты показывают, что $k = 180$ является разумным выбором, поскольку в этом случае результат на самом деле ниже, чем при 150 и 210:

```
(60,0,03475331900669869)
(90,0,051512668026335535)
(120,0.02020028911919293)
(150,0,019962563512905682)
(180,0,01110240886325257)
(210,0,01259738444250231)
(240,0,01357435960663116)
(270,0,010119881917660544)
```

Кластеризация в действии

Наконец, мы можем с уверенностью подвергнуть кластеризации полный нормализованный набор данных с $k = 180$. И снова мы можем распечатать метки для каждого кластера, чтобы получить некоторое представление о результирующей кластеризации. Похоже, что в каждом кластере преобладает один тип атаки, и они содержат только несколько типов:

```
pipeline_model = fit_pipeline_4(data, 180)❶
count_by_cluster_label = pipeline_model.transform(data).\
                                        select("cluster", "label").\
                                        groupBy("cluster", "label").\
                                      count().orderBy("cluster", "label")
count_by_cluster_label.show()
```

```
...
+-------+----------+------+
|cluster|     label| count|
+-------+----------+------+
|     0|     back.|   324|
|      0|   normal.| 42921|
|      1|  neptune.|  1039|
|      1|portsweep.|     9|
|      1| satan.   |     2|
|      2| neptune. |365375|
|      2|portsweep.|   141|
|      3|portsweep.|     2|
|      3|    satan.| 10627|
|      4|  neptune.|  1033|
|      4|portsweep.|     6|
|      4|    satan.|     1|
...
```

❶ Определение `fit_pipeline_4` приведено в архиве исходного кода книги.

Теперь мы можем сделать настоящий детектор аномалий. Обнаружение аномалии сводится к измерению расстояния новой точки данных до ее ближайшего центроида. Если это расстояние превышает некоторый порог, оно является аномальным. В качестве порога можно выбрать расстояние, скажем, до 100-й самой дальней точки данных среди известных данных:

```
import numpy as np

from pyspark.spark.ml.linalg import Vector, Vectors
from pyspark.sql.functions import udf

k_means_model = pipeline_model.stages[-1]
centroids = k_means_model.clusterCenters

clustered = pipeline_model.transform(data)

def dist_func(cluster, vec):
    return float(np.linalg.norm(centroids[cluster] - vec))
dist = udf(dist_func)

threshold = clustered.select("cluster", "scaledFeatureVector").\
    withColumn("dist_value",
        dist(col("cluster"), col("scaledFeatureVector"))).\
    orderBy(col("dist_value").desc()).take(100)
```

Последним этапом конвейера может быть применение этого порога ко всем новым точкам данных по мере их поступления. Например, при помощи Spark Streaming эту функцию можно применять к небольшим пакетам входных данных, поступаю-

щим из таких источников, как Kafka, или файлов в облачном хранилище. Точки данных, превышающие пороговое значение, могут вызвать действие оповещения, которое отправляет электронное письмо или обновляет базу данных.

Что дальше?

`KMeansModel` сама по себе является частью системы обнаружения аномалий. В предыдущих примерах кода показано, как применить эту модель к данным для обнаружения аномалий. Этот же код можно использовать в Spark Streaming для оценки новых данных по мере их поступления практически в реальном времени и, возможно, для запуска предупреждения или проверки.

MLlib также содержит вариант модели под названием `StreamingKMeans`, который может постепенно обновлять кластеры по мере поступления новых данных в `StreamingKMeansModel`. Мы можем воспользоваться этим вариантом для изучения того, как новые данные влияют на кластеризацию, а не только для оценки новых данных по сравнению с существующими кластерами. Его также можно интегрировать с Spark Streaming. Однако он не был обновлен для новых API на основе DataFrame.

Модель, которую мы построили, является упрощенной. Например, в этом примере используется евклидово расстояние, поскольку в настоящее время это единственная функция расстояния, поддерживаемая Spark MLlib. В будущем можно будет использовать функции расстояния, которые лучше учитывают распределения и корреляции между функциями, такие, как расстояние Махаланобиса.

Существуют также более сложные метрики оценки качества кластера, которые можно применять (даже без меток) для выбора k, — например, *коэффициент силуэта*. Они, как правило, оценивают не только близость точек внутри одного кластера, но и близость точек к другим кластерам. Наконец, вместо простой кластеризации K-средних можно применять другие модели. Например, смешанная модель Гаусса или DBSCAN могут фиксировать более тонкие взаимосвязи между точками данных и центрами кластеров. В Spark MLlib уже есть реализация смешанной модели Гаусса; в скором будущем реализации других моделей могут стать доступными в Spark MLlib или других библиотеках на основе Spark.

Конечно, кластеризация предназначена не только для обнаружения аномалий. На самом деле, она чаще всего связана с задачами, для решения которых применяются существующие кластеры! Например, кластеризацию можно использовать для группировки клиентов в соответствии с их поведением, предпочтениями и атрибутами. Каждый кластер сам по себе может представлять полезный различимый тип клиентов. Это более корректный способ сегментации клиентов на основе данных, а не произвольные общие категории, такие, как «возраст 20–34» и «женщины».

Исследование Википедии с помощью LDA и Spark NLP

С ростом объема неструктурированных текстовых данных в последние годы стало сложно получить актуальную и нужную информацию. Языковые технологии предоставляют мощные методы, которые можно использовать для анализа текстовых данных и извлечения информации, которую мы ищем. В этой главе мы будем использовать PySpark и библиотеку Spark NLP (natural language processing, обработка естественного языка), чтобы применить один из таких методов — *тематическое моделирование*. В частности, мы будем использовать *скрытый алгоритм Дирихле* (latent Dirichlet algorithm, LDA) для понимания набора данных документов Википедии.

Тематическое моделирование, одна из наиболее распространенных задач обработки естественного языка, представляет собой статистический подход к моделированию данных, который помогает обнаружить основные темы, присутствующие в наборе документов. Извлечение распределения тем из миллионов документов может быть полезным во многих отношениях — например, для определения причин жалоб на конкретный продукт или для определения тем в новостных статьях. Наиболее популярным алгоритмом тематического моделирования является LDA. Это генеративная модель, которая предполагает, что документы представлены распределением тем. Темы, в свою очередь, представлены распределением слов. PySpark MLlib предлагает оптимизированную версию LDA, специально предназначенную для работы в распределенной среде. Мы создадим простой конвейер тематического моделирования, используя Spark NLP для предварительной обработки данных и LDA Spark MLlib для извлечения тем.

В этой главе мы приступим к скромной задаче извлечения человеческого знания из латентных (скрытых) тем и взаимосвязей. Мы применим LDA к текстовому корпусу, состоящему из статей, содержащихся в Википедии. Мы начнем со знакомства с основами LDA и рассмотрим его реализацию в PySpark. Затем мы загрузим набор данных и настроим нашу среду программирования, установив Spark NLP. Далее мы выполним предварительную обработку данных. Вы увидите мощь готовых методов библиотеки Spark NLP, которые значительно упрощают задачи NLP.

Затем мы оценим термины в наших документах, используя *метод TF-IDF* (term frequency — inverse document frequency, частота термина — обратная частота документа) и передадим полученный результат в нашу модель LDA. В итоге мы получим темы, назначенные нашей моделью входным документам. Мы должны иметь возможность понять, к какому сегменту принадлежит запись в Википедии без необходимости ее читать. Давайте начнем с изучения основ LDA.

Скрытое распределение Дирихле

Идея скрытого распределения Дирихле заключается в том, что документы генерируются на основе набора тем. В этом процессе мы предполагаем, что каждый документ распределен по темам, а каждая тема распределена по набору терминов. Каждый документ и каждое слово генерируются путем выборки этих распределений. Обучающийся алгоритм LDA работает в обратном направлении и пытается определить распределения, в которых наблюдаемый документ наиболее вероятен.

Алгоритм пытается разделить корпус на набор соответствующих *тем*. Каждая тема фиксирует поток вариаций данных и часто соответствует одной из идей, обсуждаемых в корпусе. Документ может быть частью нескольких тем. Мы также можем рассматривать LDA как способ мягкой кластеризации документов. Если не углубляться в математику, тематическая модель LDA описывает два основных атрибута: вероятность выбора темы при выборке определенного документа и вероятность выбора определенного термина при выборе темы. Например, LDA может обнаружить тему, тесно связанную с терминами «Азимов» и «робот», а также с документами «основная серия» и «научная фантастика». Выбирая только самые важные концепции, LDA может отбросить ненужный шум и объединить сопутствующие цепочки, чтобы получить более простое представление данных.

Мы можем использовать этот механизм в различных задачах. Например, LDA может помочь нам порекомендовать похожие статьи из Википедии, исходя из тематики выбранной статьи. Инкапсулируя модели дисперсии в текстовый корпус, алгоритм может основывать оценки на более глубоком понимании, чем просто на подсчете вхождений и совпадений слов. Далее мы рассмотрим реализацию LDA в PySpark.

LDA в PySpark

PySpark MLlib содержит реализацию LDA в качестве одного из своих алгоритмов кластеризации. Вот пример кода:

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.clustering import LDA

df = spark.createDataFrame([[1, Vectors.dense([0.0, 1.0])],
      [2, Vectors.dense([2.0, 3.0])],],
      ["id", "features"])

lda = LDA(k=2, seed=1)❶
lda.setMaxIter(10)

model = lda.fit(df)

model.vocabSize()
2

model.describeTopics().show()❷
```

```
+-----+-----------+--------------------+
|topic|termIndices|         termWeights|
+-----+-----------+--------------------+
|    0|     [0, 1]|[0.53331100994293...|
|    1|     [1, 0]|[0.50230220117597...|
+-----+-----------+--------------------+
```

❶ Мы применяем LDA к нашему фрейму данных с количеством тем (k), равным 2.

❷ Фрейм данных, описывающий вес вероятности, связанный с каждым термином в наших темах.

Мы рассмотрим реализацию PySpark LDA и связанные с ней параметры в применении к набору данных Википедии. Однако сначала нам нужно загрузить соответствующий набор данных. Этим мы и займемся дальше.

Получение данных

Википедия предоставляет доступ к дампам всех своих статей. Полный дамп находится в одном большом XML-файле. Его можно скачать, а затем разместить в облачном хранилище (например, AWS S3 или GCS, Google Cloud Storage) или HDFS. Для этого воспользуйтесь, например, следующими командами:

```
$ curl -s -L https://dumps.wikimedia.org/enwiki/latest/\
$ enwiki-latest-pages-articles-multistream.xml.bz2 \
$ | bzip2 -cd \
$ | Hadoop fs -put - wikidump.xml
```

Скачивание и размещение в хранилище займет некоторое время.

Использование такого объема данных имеет смысл при работе с кластером из нескольких узлов. Чтобы запустить код примеров этой главы на локальном компьютере, желательно создать дамп меньшего размера с использованием страниц экспорта Википедии. Попробуйте скачать все страницы из нескольких категорий, которые имеют много страниц и мало подкатегорий, таких, как биология, здоровье и геометрия. Чтобы следующий код работал, загрузите полученный дамп в каталог ch06-LDA/ и переименуйте его в wikidump.xml.

Нам нужно преобразовать XML-дамп Википедии в формат, с которым мы можем легко работать в PySpark. При работе на нашей локальной машине мы можем использовать для этого удобный инструмент WikiExtractor. Он извлекает и очищает текст из дампов базы данных Википедии, аналогичных нашему.

Установите инструмент с помощью pip:

```
$ pip3 установить wikiextractor
```

Дальше остается только запустить следующую команду в каталоге, содержащем загруженный файл:

```
$ wikiextractor wikidump.xml
```

Вывод сохраняется в одном или нескольких файлах одинакового размера в заданном каталоге с именем `text`. Каждый файл будет содержать несколько документов в следующем формате:

```
$ mv text wikidump ❶
$ tree wikidump
...
wikidump
└── AA
    └── wiki_00

...
$ head -n 5 wikidump/AA/wiki_00
...

<doc id="18831" url="?curid=18831" title="Mathematics">
Mathematics

Mathematics (from Greek: ) includes the study of such topics as numbers ...
```

❶ Переименовываем каталог `text` в `wikidump`.

Прежде чем мы начнем работать с данными, давайте познакомимся поближе с библиотекой Spark NLP.

Spark NLP

Библиотека Spark NLP была разработана John Snow Labs в начале 2017 г. как встроенная в Spark библиотека аннотаций, позволяющая в полной мере использовать модули Spark SQL и MLlib. На создание этой библиотеки разработчиков вдохновила попытка использовать Spark для распределенного выполнения других библиотек NLP, которые обычно не были реализованы с учетом параллелизма или распределенных вычислений.

Spark NLP основана на тех же идеях, что и любая другая библиотека аннотирования, но отличается способом хранения аннотаций. Большинство библиотек аннотаций хранят их в объекте документа, но Spark NLP создает столбцы для различных типов аннотаций. Аннотаторы реализованы как *преобразователи* (transformer), *оценщики* (estimator) и *модели* (model). Мы рассмотрим их в следующем разделе, когда будем применять их к нашему набору данных для предварительной обработки. Но сперва загрузим и настроим Spark NLP в нашей системе.

Функциональность и аннотации NLP-библиотек

Библиотеки NLP можно разделить на два типа: *функциональные библиотеки* и *библиотеки аннотаций*.

Функциональные библиотеки созданы для конкретных задач и методов NLP. Методы, предоставляемые этими библиотеками, часто автономны. Это означает, что предполагается, что основные задачи, такие, как токенизация, должны выполняться даже при вызове методов распознавания сущностей или разметки частей речи. Это упрощает работу с ними ценой снижения производительности.

Библиотеки аннотаций — это библиотеки, в которых вся функциональность построена вокруг модели аннотаций документов. Они опираются на три основных понятия: *документ*, *аннотация* и *аннотатор*:

Документ

Представление текста, который мы хотим анализировать или с которым работать.

Аннотация

Представление вывода наших функций NLP.

Аннотатор

Объект, содержащий логику использования функций NLP.

По мере того как мы будем использовать Spark NLP в следующих разделах, эти понятия станут нам понятнее.

Подготовка вашей среды

Установите Spark NLP при помощи `pip`:

```
pip3 install spark-nlp==3.2.3
```

Запустите оболочку PySpark:

```
pyspark --packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.4
```

Импортируйте Spark NLP в нашу оболочку PySpark:

```
import sparknlp
spark = sparknlp.start()
```

Теперь вы можете импортировать соответствующие модули Spark NLP, которые мы будем использовать:

```
from sparknlp.base import DocumentAssembler, Finisher
from sparknlp.annotator import (Lemmatizer, Stemmer,
                                Tokenizer, Normalizer,
                                StopWordsCleaner)
from sparknlp.pretrained import PretrainedPipeline
```

Завершив подготовку среды программирования, можно приступить к работе с нашим набором данных. Мы начнем с анализа данных в форме фрейма PySpark.

Парсинг данных

Выходные данные WikiExtractor могут быть разделены на несколько каталогов в зависимости от размера входного дампа. Нам нужно импортировать все данные как один фрейм данных. Начнем с указания входного каталога:

```
data_source = 'wikidump/*/*'
```

Мы импортируем данные, используя метод `fullTextFiles`, доступный через `sparkContext`. Этот метод считывает данные в виде RDD. Мы конвертируем их в объект DataFrame, как и хотели:

```
raw_data = spark.sparkContext.wholeTextFiles(data_source).toDF()
raw_data.show(1, vertical=True)
...

-RECORD 0-------------------
 _1  | file:/home/analyt...
 _2  | <doc id="18831" u...
```

Результирующий фрейм данных будет состоять из двух столбцов. Количество записей будет соответствовать количеству прочитанных файлов. Первый столбец состоит из пути к файлу, а второй содержит соответствующее текстовое содержимое. Текст содержит несколько записей, но нам нужно, чтобы каждая строка соответствовала одной записи. Основываясь на структуре записей, которую мы видели ранее, мы можем разделить записи, используя пару утилит PySpark: `split` (разделить) и `explode` (разбить).

```
from pyspark.sql import functions as fun
df = raw_data.withColumn('content', fun.explode(fun.split(fun.col("_2"),
  "</doc>")))
df = df.drop(fun.col('_2')).drop(fun.col('_1'))

df.show(4, vertical=True)
...
-RECORD 0-----------------------
 content | <doc id="18831" u...
-RECORD 1-----------------------
 content |
<doc id="5627588...
-RECORD 2-----------------------
 content |
<doc id="3354393...
-RECORD 3-----------------------
 content |
<doc id="5999808...
only showing top 4 rows
(показаны только 4 первых строки)
```

Функция split используется для разделения строки фрейма данных Column на массив на основе совпадений введенного шаблона. В предыдущем коде мы разделили объединенную строку XML документа на массив, используя разделитель в виде строки </doc>. У нас получился массив из нескольких документов. Затем мы применили explode, чтобы создать новые строки для каждого элемента в массиве, возвращаемом функцией разделения. На этом шаге у нас получились строки, соответствующие каждому документу.

Проходим по структуре, полученной в столбце content нашей предыдущей операцией:

```
df.show(1, truncate=False, vertical=True)
...
-RECORD 0


-----------------------------------------------------------------
content | <doc id="18831" url="?curid=18831" title="Mathematics">
Mathematics


Mathematics (from Greek: ) includes the study of such topics as numbers...
```

Мы можем дополнительно разделить наш столбец content, извлекая заголовки записей:

```
df = df.withColumn('title', fun.split(fun.col('content'), '\n').getItem(2)) \
   .withColumn('content', fun.split(fun.col('content'), '\n').getItem(4))
df.show(4, vertical=True)
...
-RECORD 0-----------------------
 content | In mathematics, a...
 title | Tertiary ideal
-RECORD 1-----------------------
 content | In algebra, a bin...
 title | Binomial (polynom...
-RECORD 2-----------------------
 content | Algebra (from ) i...
 title | Algebra
-RECORD 3-----------------------
 content | In set theory, th...
 title | Kernel (set theory)
only showing top 4 rows
...
```

Теперь, когда у нас есть подвергнутый парсингу набор данных, можно приступить к предварительной обработке данных с использованием Spark NLP.

Подготовка данных с использованием Spark NLP

Ранее мы упоминали, что библиотека Spark NLP, основанная на модели «документ-аннотатор», опирается на понятие «документы». В PySpark изначально не существует такого понятия. Но мы также знаем, что одним из основных принципов устройства Spark NLP является хорошая совместимость с MLlib. Это достигается путем использования преобразователей, совместимых с DataFrame, которые преобразуют текстовые столбцы в документы, а аннотации — в типы данных PySpark.

Начнем с создания нашего столбца `document` с помощью `DocumentAssembler`:

```
document_assembler = DocumentAssembler() \
    .setInputCol("content") \
    .setOutputCol("document") \
    .setCleanupMode("shrink")
document_assembler.transform(df).select('document').limit(1).collect()
...

Row(document=[Row(annotatorType='document', begin=0, end=289, result='...',
    metadata={'sentence': '0'}, embeddings=[])])
```

В разделе парсинга мы могли бы использовать аннотатор Spark NLP `DocumentNormalizer`.

Мы могли бы преобразовать входной фрейм данных напрямую, как мы сделали в предыдущем коде. Однако вместо этого мы будем использовать `DocumentAssembler` и другие необходимые аннотаторы как часть конвейера машинного обучения.

Итак, в конвейер предварительной обработки мы включим следующие аннотаторы: `Tokenizer`, `Normalizer`, `StopWordsCleaner` и `Stemmer`.

Начнем с аннотатора `Tokenizer` (токенизатор):

```
# Разделяем предложение на токены (массив)
tokenizer = Tokenizer() \
  .setInputCols(["document"]) \
  .setOutputCol("token")
```

`Tokenizer` — это фундаментальный аннотатор. Почти вся обработка текстовых данных начинается с той или иной формы *токенизации*, которая представляет собой процесс разбиения необработанного текста на небольшие фрагменты. Токены могут быть словами, символами или подсловами (*n*-граммами). Большинство классических алгоритмов NLP предполагают, что на вход поступают токены. Сейчас идет разработка множества алгоритмов глубокого обучения, которые в качестве основных входных данных используют символы. Но большинство сегодняшних приложений NLP по-прежнему используют токенизацию.

Далее идет `Normalizer` (нормализатор):

```
# удаляем ненужные символы и мусор
normalizer = Normalizer() \
    .setInputCols(["token"]) \
    .setOutputCol("normalized") \
    .setLowercase(True)
```

Нормализатор очищает токены, полученные на предыдущем шаге, и удаляет все нежелательные символы из текста.

Далее идет `StopWordsCleaner` (удаление стоп-слов):

```
# удаление стоп-слов
stopwords_cleaner = StopWordsCleaner()\
      .setInputCols("normalized")\
      .setOutputCol("cleanTokens")\
      .setCaseSensitive(False)
```

Этот аннотатор удаляет из текста так называемые *стоп-слова*. Такие слова английского языка, как «the», «is» и «at», которые настолько часто встречаются в тексте и настолько малозначащие, что их можно удалить без существенного изменения смысла текста. Удаление стоп-слов полезно, когда нужно иметь дело только с наиболее семантически важными словами в тексте и игнорировать слова с низкой семантической значимостью, такие, как артикли и предлоги.

Эту цепочку завершает `Stemmer` (стеммер, выделитель корневой части слов):

```
# приведение слов к корневой форме.
stemmer = Stemmer() \
    .setInputCols(["cleanTokens"]) \
    .setOutputCol("stem")
```

Стеммер возвращает корни (основные формы) слов с целью извлечения значимой части слова. Соответственно, стемминг — это процесс сведения слова к его корневой основе с целью извлечения значимой части. Например, «выборка», «выборщик» и «предвыборный» имеют корневую форму «выбор».

Мы почти закончили. Прежде чем мы сможем завершить наш конвейер NLP, нам нужно добавить `Finisher`. Когда мы преобразуем каждую строку в кадре данных в документ, Spark NLP использует свою внутреннюю структуру. Финишер имеет решающее значение, потому что он помогает нам вернуть ожидаемую структуру — массив токенов:

```
finisher = Finisher() \
    .setInputCols(["stem"]) \
    .setOutputCols(["tokens"]) \
    .setOutputAsArray(True) \
    .setCleanAnnotations(False)
```

Теперь у нас есть все необходимые детали конвейера. Давайте построим его таким образом, чтобы каждую фазу можно было выполнять последовательно:

```
from pyspark.ml import Pipeline
nlp_pipeline = Pipeline(
```

```
    stages=[document_assembler,
            tokenizer,
            normalizer,
            stopwords_cleaner,
            stemmer,
            finisher])
```

Выполните конвейер и преобразуйте фрейм данных:

```
nlp_model = nlp_pipeline.fit(df)❶
processed_df = nlp_model.transform(df)❷
processed_df.printSchema()
...
root
|-- content: string (nullable = true)
|-- title: string (nullable = true)
|-- document: array (nullable = true)
| |-- element: struct (containsNull = true)
| | |-- annotatorType: string (nullable = true)
| | |-- begin: integer (nullable = false)
| | |-- end: integer (nullable = false)
| | |-- result: string (nullable = true)
| | |-- metadata: map (nullable = true)
| | | |-- key: string
| | | |-- value: string (valueContainsNull = true)
| | |-- embeddings: array (nullable = true)
| | | |-- element: float (containsNull = false)
|-- token: array (nullable = true)
| |-- element: struct (containsNull = true)
| | |-- annotatorType: string (nullable = true)
| | |-- begin: integer (nullable = false)
| | |-- end: integer (nullable = false)
| | |-- result: string (nullable = true)
| | |-- metadata: map (nullable = true)
| | | |-- key: string
| | | |-- value: string (valueContainsNull = true)
| | |-- embeddings: array (nullable = true)
| | | |-- element: float (containsNull = false)
|-- normalized: array (nullable = true)
| |-- element: struct (containsNull = true)
| | |-- annotatorType: string (nullable = true)
| | |-- begin: integer (nullable = false)
| | |-- end: integer (nullable = false)
| | |-- result: string (nullable = true)
| | |-- metadata: map (nullable = true)
| | | |-- key: string
| | | |-- value: string (valueContainsNull = true)
| | |-- embeddings: array (nullable = true)
```

```
| | | |-- element: float (containsNull = false)
|-- cleanTokens: array (nullable = true)
| |-- element: struct (containsNull = true)
| | |-- annotatorType: string (nullable = true)
| | |-- begin: integer (nullable = false)
| | |-- end: integer (nullable = false)
| | |-- result: string (nullable = true)
| | |-- metadata: map (nullable = true)
| | | |-- key: string
| | | |-- value: string (valueContainsNull = true)
| | |-- embeddings: array (nullable = true)
| | | |-- element: float (containsNull = false)
|-- stem: array (nullable = true)
| |-- element: struct (containsNull = true)
| | |-- annotatorType: string (nullable = true)
| | |-- begin: integer (nullable = false)
| | |-- end: integer (nullable = false)
| | |-- result: string (nullable = true)
| | |-- metadata: map (nullable = true)
| | | |-- key: string
| | | |-- value: string (valueContainsNull = true)
|    |    |-- embeddings: array (nullable = true)
|    |    |    |-- element: float (containsNull = false)
|-- tokens: array (nullable = true)
|    |-- element: string (containsNull = true)
```

❶ Обучаем конвейер.

❷ Применяем конвейер для преобразования фрейма данных.

Конвейер NLP создал промежуточные столбцы, которые нам не нужны. Удалим лишние столбцы:

```
tokens_df = processed_df.select('title','tokens')
tokens_df.show(2, vertical=True)
...

-RECORD 0----------------------
 title | Tertiary ideal
 tokens | [mathemat, tertia...
-RECORD 1----------------------
 title | Binomial (polynom...
 tokens | [algebra, binomi,...
only showing top 2 rows
(показаны только верхние 2 строки)
```

Далее мы изучим основы метода TF-IDF и реализуем его на предварительно обработанном наборе данных `token_df`, который мы получили перед построением модели LDA.

TF-IDF

Перед применением алгоритма LDA нам нужно преобразовать наши данные в числовое представление. Мы получим такое представление, используя метод TF-IDF. В общих чертах TF-IDF используется для определения важности терминов, связанных с определенным документом. Далее в ознакомительных целях показана реализация метода в коде Python. На самом деле мы не будем использовать этот код, потому что PySpark содержит собственную реализацию.

```
import math

def term_doc_weight(term_frequency_in_doc, total_terms_in_doc,
                    term_freq_in_corpus, total_docs):
  tf = term_frequency_in_doc / total_terms_in_doc
  doc_freq = total_docs / term_freq_in_corpus
  idf = math.log(doc_freq)
  tf * idf
}
```

TF-IDF использует два критерия релевантности термина документу. Во-первых, мы ожидаем, что чем чаще термин встречается в документе, тем важнее он для этого документа. Во-вторых, не все термины равны в глобальном смысле. Гораздо важнее встретить в данном документе слово, редко встречающееся во всем корпусе, чем слово, встречающееся в большинстве документов; поэтому мы и говорим, что метрика использует *обратную* частоту появления слова в документах в полном корпусе.

Частотность слов в корпусе склонна к экспоненциальному распределению. Обычное слово может встречаться в десять раз чаще, чем умеренно распространенное слово, которое, в свою очередь, может встречаться в десять или сто раз чаще, чем редкое слово. Если бы метрика основывалась на необработанной обратной частотности документа, то редким словам придавался бы огромный вес, а влияние всех остальных слов практически не учитывалось бы. Чтобы исправить это распределение, схема использует логарифм обратной частотности документа. Логарифм смягчает различия в частотности, преобразуя мультипликативные интервалы между словами в аддитивные.

Модель основана на нескольких предположениях. Она рассматривает каждый документ как *мешок слов* (bag of words), то есть не обращает внимания на порядок слов, структуру предложения или отрицание. Представляя каждый термин один раз, модель с трудом справляется с многозначностью, то есть использованием одного и того же слова в нескольких значениях. Например, модель не может различить использование слова «группа» в предложениях «Radiohead — лучшая группа всех времен» и «У меня редкая группа крови». Если оба предложения часто встречаются в корпусе, может возникнуть ассоциация «Radiohead» со словом «кровь».

Теперь перейдем к рассмотрению реализации TF-IDF с помощью PySpark.

Вычисление TF-IDF

Прежде всего, мы рассчитаем TF (term frequency, частоту каждого термина в документе) с помощью `CountVectorizer`, который отслеживает создаваемый словарь, чтобы мы могли сопоставить наши темы с соответствующими словами. TF создает матрицу, в которую заносятся подсчеты, сколько раз каждое слово словаря встречается в каждом тексте. Затем каждому слову присваивается вес в зависимости от его частоты. Мы получаем подсчеты на этапе преобразования:

```
from pyspark.ml.feature import CountVectorizer
cv = CountVectorizer(inputCol="tokens", outputCol="raw_features")

# обучение модели
cv_model = cv.fit(tokens_df)

# преобразование данных. Столбец выходных данных - raw_features.
vectorized_tokens = cv_model.transform(tokens_df)
```

Затем мы переходим к IDF (обратная частотность для документов, в которых встречается термин), что уменьшает веса часто встречающихся терминов:

```
from pyspark.ml.feature import IDF
idf = IDF(inputCol="raw_features", outputCol="features")

idf_model = idf.fit(vectorized_tokens)

vectorized_df = idf_model.transform(vectorized_tokens)
```

Вот как будет выглядеть результат:

```
vectorized_df = vectorized_df.drop(fun.col('raw_features'))

vectorized_df.show(6)
...

+--------------------+--------------------+--------------------+
|               title|              tokens|            features|
+--------------------+--------------------+--------------------+
|      Tertiary ideal|[mathemat, tertia...|(2451,[1,6,43,56,...|
|Binomial (polynom...|[algebra, binomi,...|(2451,[0,10,14,34...|
|             Algebra|[algebra, on, bro...|(2451,[0,1,5,6,15...|
| Kernel (set theory)|[set, theori, ker...|(2451,[2,3,13,19,...|
|Generalized arith...|[mathemat, gener,...|(2451,[1,2,6,45,4...|
+--------------------+--------------------+--------------------+
```

После завершения предварительной обработки и извлечения признаков мы можем создать нашу модель LDA, чем мы и займемся в следующем разделе.

Создание модели LDA

Ранее мы упоминали, что LDA перерабатывает текстовый корпус в набор релевантных тем. В этом разделе мы рассмотрим примеры таких тем. Перед этим нам нужно определиться с двумя гиперпараметрами, которые требуются для нашей модели LDA. Это количество тем (обозначается *k*) и количество итераций.

Есть несколько способов выбрать *k*. Для этого используются две популярные метрики: *перплексия* (perplexity, переплетенность) и *связность темы*. Первая метрика доступна благодаря реализации PySpark. Основная идея состоит в том, чтобы попытаться определить *k*, при котором улучшения этих показателей становятся незначительными. Если вы знакомы с методом убывающей ценности для нахождения количества кластеров для K-средних, здесь применяется похожий принцип. В зависимости от размера вашего корпуса это может быть ресурсоемкий и трудоемкий процесс, поскольку вам нужно будет построить модель для нескольких значений *k*. В качестве альтернативы вы можете попробовать создать репрезентативную выборку имеющегося набора данных и использовать ее для определения *k*. Попробуйте в качестве упражнения сделать это самостоятельно.

Поскольку вы можете работать локально, мы продолжим и зададим значения (`k = 5` и `max_iter = 50`), которые на данный момент выглядят интуитивно подходящими.

Создадим нашу модель LDA:

```
from pyspark.ml.clustering import LDA

num_topics = 5
max_iter = 50

lda = LDA(k=num_topics, maxIter=max_iter)
model = lda.fit(vectorized_df)

lp = model.logPerplexity(vectorized_df)

print("The upper bound on perplexity: " + str(lp))
...

The upper bound on perplexity: 6.768323190833805
```

Перплексия — это показатель того, насколько хорошо модель предсказывает выборку. Низкий уровень перплексии указывает на то, что распределение вероятностей терминов хорошо предсказывает выборку. При сравнении разных моделей выбирайте ту, у которой меньшее значение перплексии[1].

[1] *Perplexity* дословно означает «растерянность, замешательство». С математической точки зрения отражением перплексии является *коэффициент неопределенности*. Чем меньше неопределенность в предсказании выборки смоделированным распределением, тем лучше. — *Примеч. пер.*

Теперь, когда мы создали нашу модель, будет полезно вывести темы в удобочитаемом виде. Мы получим словарь, сгенерированный на наших этапах предварительной обработки, затем получим темы из модели LDA и сопоставим их между собой.

```
vocab = cv_model.vocabulary ❶

raw_topics = model.describeTopics().collect()❷

topic_inds = [ind.termIndices for ind in raw_topics]❸

topics = []
for topic in topic_inds:
    _topic = []
    for ind in topic:
        _topic.append(vocab[ind])
    topics.append(_topic)
```

❶ Создаем ссылку на наш словарь.

❷ Получаем темы, сгенерированные моделью LDA с помощью `descriptionTopics`, и загружаем их в список Python.

❸ Получаем индексы словарных терминов из наших тем.

Давайте теперь сгенерируем отображения из наших тематических индексов в наш словарь:

```
for i, topic in enumerate(topics, start=1):
    print(f"topic {i}: {topic}")
...

topic 1: ['ислам', 'здоров', 'лекарств', 'импер', 'медицин', 'полит',...
topic 2: ['формул', 'групп', 'алгебр', 'генер', 'измен', ...
topic 3: ['треугол', 'плоск', 'лин', 'точк', 'два', 'тангенс', ...
topic 4: ['лиц', 'терапевт', 'фреймворк', 'частиц', 'интерф', ...
topic 5: ['вычисл', 'полином', 'паттерн', 'интернет', 'сет', ...
```

Этот результат не идеален, но в темах можно заметить некоторые закономерности (для наглядности приведены переводы терминов в корневой форме). Тема 1 в первую очередь связана со здоровьем. Она также содержит ссылки на ислам и империю. Может быть, это из-за того, что они упоминаются в истории медицины и наоборот, или еще по какой-то причине? Темы 2 и 3 связаны с математикой, причем последняя имеет склонность к геометрии. Тема 5 представляет собой смесь вычислений и математики. Даже если вы не читали ни один из документов, вы уже можете с достаточной точностью догадаться об их категориях. Это впечатляет!

Теперь мы можем проверить, с какими темами наши входные документы наиболее тесно связаны. Один документ может иметь несколько значимых тематических ассоциаций. На данный момент мы рассмотрим только наиболее тесно связанные темы.

Выполним операцию преобразования модели LDA для нашего входного фрейма данных:

```
lda_df = model.transform(vectorized_df)
lda_df.select(fun.col('title'), fun.col('topicDistribution')).\
                show(2, vertical=True, truncate=False)
...
-RECORD 0-------------------------------------
 title | Tertiary ideal
 topicDistribution | [5.673953573608612E-4,...
-RECORD 1----------------------------------...
 title | Binomial (polynomial) ...
 topicDistribution | [0.0019374384060205127...
only showing top 2 rows
(показаны только верхние 2 строки)
```

Как видите, с каждым документом связано распределение вероятности темы. Чтобы получить связанную тему для каждого документа, нужно узнать для него индекс темы с наивысшей оценкой вероятности. Затем мы можем сопоставить индексы с темами, которые мы получили ранее.

Напишем PySpark UDF, чтобы найти наивысшую оценку вероятности темы для каждой записи:

```
from pyspark.sql.types import IntegerType
max_index = fun.udf(lambda x: x.tolist().index(max(x)) + 1, IntegerType())
lda_df = lda_df.withColumn('topic_index',
                       max_index(fun.col('topicDistribution')))
lda_df.select('title', 'topic_index').show(10, truncate=False)
...


+----------------------------------+-----------+
|                            title |topic_index|
+----------------------------------+-----------+
|Tertiary ideal                    |2          |
|Binomial (polynomial)             |2          |
|Algebra                           |2          |
|Kernel (set theory)               |2          |
|Generalized arithmetic progression|2          |
|Schur algebra                     |2          |
|Outline of algebra                |2          |
|Recurrence relation               |5          |
|Rational difference equation      |5          |
|Polynomial arithmetic             |2          |
+----------------------------------+-----------+
only showing top 11 rows
```

Тема 2, если помните, была связана с математикой. Результат соответствует нашим ожиданиям. Вы можете просканировать более объемный набор данных, чтобы увидеть, как это работает. Вы можете выбрать определенные темы с помощью команд `where` или `filter` и сравнить их со списком тем, сгенерированным ранее, чтобы лучше понять принцип формирования кластеров. Как и было заявлено в начале главы, мы теперь можем группировать статьи по разным темам, не читая их!

Что дальше?

В этой главе мы применили LDA к текстовому корпусу Википедии. В процессе подготовки данных вы узнали о предварительной обработке текста с использованием удивительной библиотеки Spark NLP и техники TF-IDF. Вы можете продолжить изучение этой темы, улучшив модель за счет лучшей предварительной обработки и настройки гиперпараметров. Кроме того, при наличии пользовательского ввода вы можете попытаться рекомендовать похожие записи на основе схожести документов. Нужную меру подобия можно получить, используя вектор распределения вероятностей, полученный из LDA.

Отметим также, что для понимания больших корпусов текстов существует множество других методов. Например, метод, известный как *латентный семантический анализ* (latent semantic analysis, LSA), полезен в аналогичных ситуациях и использовался в предыдущем издании этой книги для того же набора данных. Глубокое обучение, которое рассматривается в *главе 10*, описывает возможности тематического моделирования. Вы можете изучить их самостоятельно.

ГЛАВА 7

Геопространственный и временнóй анализ данных о поездках на такси

Характерной особенностью *геопространственных данных* (geospatial data) является наличие, в той или иной форме, встроенной информации о местоположении. Такие данные в настоящее время ежедневно генерируются в огромных масштабах миллиардами источников, такими, как мобильные телефоны и датчики. Данные о передвижении людей и машин, а также данные дистанционного зондирования очень важны для нашей экономики и общего благосостояния. Геопространственный анализ предоставляет нам инструменты и методы, необходимые для понимания всех этих данных и использования их для решения прикладных задач.

Когда речь идет о геопространственном анализе, стоит отметить, что за последние несколько лет экосистемы PySpark и PyData очень сильно изменились. Их используют в разных отраслях для обработки данных о местоположении и, в свою очередь, они влияют на нашу повседневную жизнь. Одно из применений геопространственных данных, с которым мы сталкиваемся каждый день, — это пассажирский транспорт. Феномен цифровых сервисов вызова такси, ставших чрезвычайно популярными за последние несколько лет, привел к тому, что мы стали лучше разбираться в геопространственных технологиях. В этой главе вы примените на практике полученные навыки работы с PySpark и анализа данных для работы с набором геопространственных данных о поездках, совершенных такси в Нью-Йорке.

Одним из статистических показателей, важных для понимания экономических основ бизнеса такси, является *коэффициент использования*: доля времени, в течение которого такси находится в пути и занято одним или несколькими пассажирами. Одним из факторов, влияющих на использование, является пункт назначения пассажира: такси, которое высаживает пассажиров возле Юнион-сквер (многолюдная площадь в Манхэттене, Нью-Йорк) в полдень, с гораздо большей вероятностью найдет следующий заказ всего за минуту или две, в то время как водителю такси, который высадил кого-то в 2 часа ночи на Статен-Айленде (самый удаленный от центра и малонаселенный район), возможно, придется проделать весь путь до Манхэттена, прежде чем он найдет следующий заказ. Мы хотели бы количественно оценить влияние пункта назначения и выяснить, сколько времени требуется водителю такси, чтобы найти следующий заказ, в зависимости от района, в котором он высадил своих пассажиров, — Манхэттен, Бруклин, Квинс, Бронкс, Статен-Айленд или ничего из вышеперечисленного (например, если он высадил пассажира где-то за пределами города, например, в международном аэропорту Ньюарк Либерти).

Мы начнем с подготовки нашего набора данных, а затем погрузимся в геопространственный анализ. Вы узнаете о формате GeoJSON и научитесь использовать инструменты экосистемы PyData в сочетании с PySpark. Для работы с геопространственной информацией, такой, как точки долготы и широты и пространственные границы вы будете использовать GeoPandas. Завершая эту главу, вы освоите работу с данными, отражающими дату и время. Это поможет вам понять использование такси Нью-Йорка. PySpark и его DataFrame API предоставляют готовые типы данных и методы для обработки показателей времени.

Давайте начнем с загрузки набора данных и изучения его с помощью PySpark.

Происхождение набора данных о поездках на такси в Нью-Йорке

У жителей Нью-Йорка есть всевозможные приметы, основанные на их неподтвержденном личном опыте, о том, когда и где лучше всего поймать такси, особенно в час пик и во время дождя. Но есть период времени, когда все порекомендуют вам просто сесть на метро — это время пересменки, которая происходит между 16 и 17 часами вечера ежедневно. На это время желтым такси приходится возвращаться в свои диспетчерские центры (часто в Квинсе), чтобы один водитель мог закончить смену, а следующий мог приступить к работе. Опоздавшим водителям приходится платить штраф.

В марте 2014 г. Комиссия по такси и лимузинам Нью-Йорка поделилась в своем аккаунте в Твиттере @nyctaxi инфографикой, в которой было показано количество такси на дороге и доля занятых машин среди них. Конечно же, с 16 до 18 часов на дороге наблюдался заметный провал количества машин, и две трети работавших такси были заняты.

Этот твит привлек внимание самопровозглашенного урбаниста, картографа и фанатика данных Криса Уонга, который отправил твит в аккаунт @nyctaxi, чтобы узнать, можно ли получить свободный доступ к данным, использованным в инфографике. Крис получил доступ ко всем данным о поездках на такси с 1 января по 31 декабря 2013 г. Более того, он разместил все данные о стоимости проезда в Интернете, где они использовались в качестве основы для ряда прекрасных визуализаций работы транспорта в Нью-Йорке. Мы тоже воспользуемся этим набором данных.

Подготовка данных

Для анализа в этой главе мы будем рассматривать только данные о заказах за январь 2013 г., которые после распаковки занимают около 2,5 Гб. Вы можете получить доступ к данным за каждый месяц 2013 г., и, если у вас есть достаточно большой кластер PySpark, вы можете впоследствии проанализировать данные за весь год.

А пока давайте создадим рабочий каталог на нашей клиентской машине и посмотрим на структуру данных о заказах:

```
$ mkdir taxidata
$ cd taxidata
$ curl -O https://storage.googleapis.com/aas-data-sets/trip_data_1.csv.zip
$ unzip trip_data_1.csv.zip
$ head -n 5 trip_data_1.csv

...

medallion,hack_license,vendor_id,rate_code,store_and_fwd_flag,...
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,1,...
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,...
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,...
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,...
```

Каждая строка файла после заголовка представляет одну поездку на такси в формате CSV. Для каждой поездки у нас есть атрибуты такси (хешированная версия номера автомобиля), а также водителя (хешированная версия лицензии на вождение такси), информация о времени начала и конца поездки, а также координаты долготы/широты места, где пассажиров забрали и высадили.

Создадим директорию taxidata и скопируем данные о поездках в хранилище:

```
$ mkdir taxidata
$ mv trip_data_1.csv taxidata/
```

Здесь мы использовали локальную файловую систему, но вы можете поступить иначе. В наше время никого не удивит использование собственной облачной файловой системы, такой, как AWS S3 или GCS. В таком случае вы будете загружать данные в S3 или GCS соответственно.

Теперь запустите оболочку PySpark:

```
$ pyspark
```

После загрузки оболочки PySpark мы можем создать свой рабочий набор данных из данных такси и проверить первые несколько строк, как мы это делали в других главах:

```
taxi_raw = pyspark.read.option("header", "true").csv("taxidata")
taxi_raw.show(1, vertical=True)

...

RECORD 0-----------------------------------
 medallion          | 89D227B655E5C82AE...
 hack_license       | BA96DE419E711691B...
 vendor_id          | CMT
 rate_code          | 1
 store_and_fwd_flag | N
 pickup_datetime    | 2013-01-01 15:11:48
 dropoff_datetime   | 2013-01-01 15:18:10
```

```
passenger_count    | 4
 trip_time_in_secs  | 382
 trip_distance      | 1.0
 pickup_longitude   | -73.978165
 pickup_latitude    | 40.757977
 dropoff_longitude  | -73.989838
 dropoff_latitude   | 40.751171
only showing top 1 row
(показана только первая строка)
...
```

На первый взгляд это выглядит как хорошо отформатированный набор данных. Теперь посмотрим на схему фрейма данных:

```
taxi_raw.printSchema()
...
root
 |-- medallion: string (nullable = true)
 |-- hack_license: string (nullable = true)
 |-- vendor_id: string (nullable = true)
 |-- rate_code: integer (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- pickup_datetime: string (nullable = true)
 |-- dropoff_datetime: string (nullable = true)
 |-- passenger_count: integer (nullable = true)
 |-- trip_time_in_secs: integer (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- pickup_longitude: double (nullable = true)
 |-- pickup_latitude: double (nullable = true)
 |-- dropoff_longitude: double (nullable = true)
 |-- dropoff_latitude: double (nullable = true)
 ...
```

Мы представляем поля `pickup_datetime` и `dropoff_datetime` как `String` и сохраняем отдельные координаты `(x,y)` мест посадки и высадки как данные типа `Double`. Нам нужно, чтобы поля даты и времени были метками времени, поскольку это позволит нам удобно манипулировать ими и анализировать их.

Преобразование строк даты и времени в метки времени

Как упоминалось ранее, PySpark предоставляет готовые методы для обработки данных времени.

В частности, мы будем использовать функцию `to_timestamp` для анализа строк даты и времени и преобразования их в метки времени:

```
from pyspark.sql import functions as fun

taxi_raw = taxi_raw.withColumn('pickup_datetime',
                               fun.to_timestamp(fun.col('pickup_datetime'),
                                                "yyyy-MM-dd HH:mm:ss"))
taxi_raw = taxi_raw.withColumn('dropoff_datetime',
                              fun.to_timestamp(fun.col('dropoff_datetime'),
                                               "yyyy-MM-dd HH:mm:ss"))
```

Давайте еще раз взглянем на схему:

```
taxi_raw.printSchema()
...

root
 |-- medallion: string (nullable = true)
 |-- hack_license: string (nullable = true)
 |-- vendor_id: string (nullable = true)
 |-- rate_code: integer (nullable = true)
 |-- store_and_fwd_flag: string (nullable = true)
 |-- pickup_datetime: timestamp (nullable = true)
 |-- dropoff_datetime: timestamp (nullable = true)
 |-- passenger_count: integer (nullable = true)
 |-- trip_time_in_secs: integer (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- pickup_longitude: double (nullable = true)
 |-- pickup_latitude: double (nullable = true)
 |-- dropoff_longitude: double (nullable = true)
 |-- dropoff_latitude: double (nullable = true)

...
```

Поля `pickup_datetime` и `dropoff_datetime` теперь являются метками времени. Отличная работа!

Мы упоминали, что этот набор данных содержит поездки за январь 2013 г. Однако не следует верить нам на слово. Мы можем подтвердить это, отсортировав поле `pickup_datetime`, чтобы получить самую последнюю дату и время в данных. Для этого мы используем метод `sort` DataFrame в сочетании с методом `desc` столбца PySpark:

```
taxi_raw.sort(fun.col("pickup_datetime").desc()).show(3, vertical=True)
...

-RECORD 0----------------------------------
 medallion          | EA00A64CBDB68C77D...
 hack_license       | 2045C77002FA0F2E0...
```

```
 vendor_id          | CMT
 rate_code          | 1
 store_and_fwd_flag | N
 pickup_datetime    | 2013-01-31 23:59:59
 dropoff_datetime   | 2013-02-01 00:08:39
 passenger_count    | 1
 trip_time_in_secs  | 520
 trip_distance      | 1.5
 pickup_longitude   | -73.970528
 pickup_latitude    | 40.75502
 dropoff_longitude  | -73.981201
 dropoff_latitude   | 40.769104
-RECORD 1-----------------------------------
 medallion          | E3F00BB3F4E710383...
 hack_license       | 10A2B96DE39865918...
 vendor_id          | CMT
 rate_code          | 1
 store_and_fwd_flag | N
 pickup_datetime    | 2013-01-31 23:59:59
 dropoff_datetime   | 2013-02-01 00:05:16
 passenger_count    | 1
 trip_time_in_secs  | 317
 trip_distance      | 1.0
 pickup_longitude   | -73.990685
 pickup_latitude    | 40.719158
 dropoff_longitude  | -74.003288
 dropoff_latitude   | 40.71521
-RECORD 2-----------------------------------
 medallion          | 83D8E776A05EEF731...
 hack_license       | E6D27C8729EF55D20...
 vendor_id          | CMT
 rate_code          | 1
 store_and_fwd_flag | N
 pickup_datetime    | 2013-01-31 23:59:58
 dropoff_datetime   | 2013-02-01 00:04:19
 passenger_count    | 1
 trip_time_in_secs  | 260
 trip_distance      | 0.8
 pickup_longitude   | -73.982452
 pickup_latitude    | 40.77277
 dropoff_longitude  | -73.989227
 dropoff_latitude   | 40.766754
only showing top 3 rows
(показаны только 3 первых строки)
...
```

Теперь мы можем проверить, есть ли какие-либо расхождения фактических данных с ожиданиями.

Обработка недействительных записей

Любой, кто работал с крупномасштабными наборами данных из реального мира, знает, что они неизменно содержат, как минимум, несколько записей, которые не соответствуют ожиданиям человека, написавшего код для их обработки. Многие конвейеры PySpark вышли из строя из-за недопустимых записей, из-за которых логика синтаксического анализа выдавала исключение. При помощи интерактивного анализа мы можем получить представление о потенциальных аномалиях в данных, сосредоточившись на ключевых переменных.

В нашем случае переменные, содержащие геопространственную и временную информацию, заслуживают проверки на несоответствия. Наличие нулевых значений в этих столбцах несомненно должно привлечь наше внимание.

```
geospatial_temporal_colnames = ["pickup_longitude", "pickup_latitude", \
                                "dropoff_longitude", "dropoff_latitude", \
                                "pickup_datetime", "dropoff_datetime"]
taxi_raw.select([fun.count(fun.when(fun.isnull(c), c)).\
                      alias(c) for c in geospatial_temporal_colnames]).\
                show()
...

+----------------+---------------+----------------
|pickup_longitude|pickup_latitude|dropoff_longitude
+----------------+---------------+----------------
|               0|              0|              86
+----------------+---------------+----------------
+----------------+---------------+----------------+
|dropoff_latitude|pickup_datetime|dropoff_datetime|
+----------------+---------------+----------------+
|              86|              0|               0|
+----------------+---------------+----------------+
```

Теперь удалим нулевые значения из наших данных:

```
taxi_raw = taxi_raw.na.drop(subset=geospatial_temporal_colnames)
```

Другая проверка на основе здравого смысла, которую мы можем сделать, касается записей широты и долготы, где значения равны нулю. Мы знаем, что для интересующего нас региона это будут недопустимые значения:

```
print("Количество нулевых значений широты и долготы координат посадки и высадки")
taxi_raw.groupBy((fun.col("dropoff_longitude") == 0) |
  (fun.col("dropoff_latitude") == 0) |
  (fun.col("pickup_longitude") == 0) |
  (fun.col("pickup_latitude") == 0)).\ ❶
    count().show()
...
```

```
Количество нулевых значений широты и долготы координат посадки и высадки
+---------------+
| ...  |   count|
+------+--------+
| true |  285909|
| false|14490620|
+---------------+
```

❶ Комбинация условий `OR` (`|`) будет истинной, если хотя бы одно из них выполняется для какой-либо записи.

У нас таких записей оказалось довольно много. Если запись выглядит так, будто такси доставило пассажира на Южный полюс, будет разумно полагать, что она недействительна и должна быть исключена из нашего анализа. Но мы не станем их удалять, а вернемся к ним ближе к концу следующего раздела, чтобы посмотреть, как они влияют на результаты анализа.

В повседневной работе мы обрабатываем возникшие исключения по одному, проверяя журналы для отдельных задач, выясняя, какая строка кода вызвала исключение, а затем выясняя, как настроить код, чтобы игнорировать или исправить недопустимые записи. Это утомительный процесс, и часто кажется, что мы играем в «убей крота»: как только мы устраняем одно исключение, при следующем запуске возникает новое.

Одна из стратегий, которую опытные специалисты по данным применяют при работе с новым набором данных, заключается в добавлении блока `try-except` в код синтаксического анализа, чтобы любые недопустимые записи можно было записывать в журнал, не вызывая сбоя всей работы конвейера. Если во всем наборе данных встречается лишь несколько недопустимых записей, мы можем игнорировать их и продолжить наш анализ.

Теперь, когда у нас готов набор данных, можно приступить к проведению геопространственного анализа.

Геопространственный анализ

Существует два основных типа геопространственных данных — векторные и растровые, и для работы с каждым типом существуют разные инструменты. В данном случае у нас есть широта и долгота для записей о поездках на такси, а также векторные данные, хранящиеся в формате GeoJSON, представляющие границы различных районов Нью-Йорка. Нас интересуют значения широты и долготы. Давайте начнем с просмотра данных GeoJSON.

Введение в GeoJSON

Данные, которые мы будем использовать для определения границ районов Нью-Йорка, записаны в формате GeoJSON. Базовая структурная единица GeoJSON на-

зывается *объект* (feature). Он состоит из экземпляра *геометрии* (geometry) и набора пар ключ-значение, называемых *свойствами* (property). Геометрия — это форма, такая, как *точка* (point), *линия* (line) или *полигон* (polygon). Набор объектов называется FeatureCollection. Давайте извлечем данные GeoJSON для карт районов Нью-Йорка и изучим их структуру.

Зайдите в каталог `taxidata` на вашем клиентском компьютере, скачайте данные и назовите файл более коротким именем:

```
$ url="https://nycdatastables.s3.amazonaws.com/\
        2013-08-19T18:15:35.172Z/nyc-borough-boundaries-polygon.geojson"
$ curl -O $url
$ mv nyc-borough-boundaries-polygon.geojson nyc-boroughs.geojson
```

Откройте файл и просмотрите запись объекта. Обратите внимание на свойства и геометрию — в данном случае полигон, представляющий границы района, и свойства, содержащие название района и другую связанную информацию.

```
$ head -n 7 data/trip_data_ch07/nyc-boroughs.geojson
...
{
"type": "FeatureCollection",

"features": [{ "type": "Feature", "id": 0, "properties": { "boroughCode": 5, ...
,
{ "type": "Feature", "id": 1, "properties": { "boroughCode": 5, ...
```

Форматы геопространственных данных

Существуют различные специализированные геопространственные форматы. Кроме того, для сбора данных о местоположении можно использовать такие источники данных, как журналы с геотегами и текстовые данные. К ним относятся:

- векторные форматы, такие, как GeoJSON, KML, shapefile и WKT;
- растровые форматы, такие, как сетка Esri, GeoTIFF, JPEG 2000 и NITF;
- навигационные стандарты, например, используемые устройствами AIS и GPS;
- базы геоданных, доступные через соединения JDBC/ODBC, такие, как PostgreSQL/PostGIS;
- форматы выходных данных датчиков гиперспектральных, мультиспектральных, лидарных и радарных платформ;
- веб-стандарты OGC, такие, как WCS, WFS, WMS и WMTS;
- снабженные геотегами логи, изображения, видео и записи в социальных сетях;
- неструктурированные данные со ссылками на местоположение.

GeoPandas

Первое, что вы должны учитывать при выборе библиотеки для выполнения геопространственного анализа, это определить, с какими данными вам нужно будет работать. Нам нужна библиотека, которая может анализировать данные GeoJSON и способна обрабатывать пространственные отношения, например, определять, содержится ли заданная пара долгота/широта внутри полигона, представляющего границы определенного района. Для этой задачи мы будем использовать библиотеку GeoPandas. GeoPandas — это проект с открытым исходным кодом, упрощающий работу с геопространственными данными в Python. Он расширяет типы данных, используемые библиотекой pandas, которую мы использовали в предыдущих главах, чтобы выполнять пространственные операции с геометрическими типами данных.

Установите пакет GeoPandas с помощью pip:

```
pip3 install geopandas
```

Давайте теперь приступим к изучению геопространственных аспектов данных о поездках. Для каждой поездки у нас есть пары долгота/широта, представляющие, где пассажира забрали и высадили. Нам хотелось бы иметь возможность определить, к какому району принадлежит каждая из этих пар широта/долгота, и идентифицировать любые поездки, которые не начинались или не заканчивались ни в одном из пяти районов Нью-Йорка. Например, если такси доставило пассажиров из Манхэттена в международный аэропорт Ньюарк Либерти, это была бы подходящая поездка, которую было бы интересно проанализировать, поскольку она началась, но не закончилась в одном из пяти районов.

Чтобы выполнить анализ района, нам нужно загрузить данные GeoJSON, которые мы загрузили ранее и сохранили в файле nyc-boroughs.geojson:

```
import geopandas as gdp
gdf = gdp.read_file("./data/trip_data_ch07/nyc-boroughs.geojson")
```

Прежде чем использовать объекты GeoJSON для анализа данных о поездках на такси, мы должны подумать о том, как организовать эти геопространственные данные для максимальной эффективности. Одним из вариантов может быть исследование структур данных, оптимизированных для геопространственного поиска, таких, как деревья квадрантов, а затем поиск или написание собственной реализации. Вместо этого мы попытаемся придумать быструю эвристику, которая позволит нам обойтись без этой части работы.

Мы будем перебирать `gdf`, пока не найдем объект, геометрия которого содержит заданную точку долготы/широты. Большинство поездок на такси в Нью-Йорке начинаются и заканчиваются на Манхэттене, поэтому, если геопространственные объекты, представляющие Манхэттен, находятся в начале последовательности, наши поиски закончатся относительно быстро. Мы можем использовать тот факт, что свойство `boroughCode` каждого объекта можно использовать в качестве ключа сортировки, при этом код для Манхэттена равен 1, а код для Стейтен-Айленда равен 5. Что касается объектов для каждого района, нам нужно, чтобы объекты, связанные с самыми большими полигонами, располагались перед теми, которые связаны с

меньшими полигонами, потому что большинство поездок будет осуществляться в «основной» регион каждого района и обратно.

Мы рассчитаем площадь, связанную с геометрией каждого объекта, и сохраним ее как новый столбец:

```
gdf = gdf.to_crs(3857)

gdf['area'] = gdf.apply(lambda x: x['geometry'].area, axis=1)
gdf.head(5)
...

   boroughCode borough        @id     geometry     area
0  5           Staten Island  http://nyc.pediacities.com/Resource/Borough/St...
1  5           Staten Island  http://nyc.pediacities.com/Resource/Borough/St...
2  5           Staten Island  http://nyc.pediacities.com/Resource/Borough/St...
3  5           Staten Island  http://nyc.pediacities.com/Resource/Borough/St...
4  4           Queens         http://nyc.pediacities.com/Resource/Borough/Qu...
```

Сортировка объектов по комбинации кода района и площади геометрии каждого объекта должна дать нужный результат:

```
gdf = gdf.sort_values(by=['boroughCode', 'area'], ascending=[True, False])
gdf.head(5)
...
    boroughCode  borough    @id     geometry     area
72  1            Manhattan  http://nyc.pediacities.com/Resource/Borough/Ma...
71  1            Manhattan  http://nyc.pediacities.com/Resource/Borough/Ma...
51  1            Manhattan  http://nyc.pediacities.com/Resource/Borough/Ma...
69  1            Manhattan  http://nyc.pediacities.com/Resource/Borough/Ma...
73  1            Manhattan  http://nyc.pediacities.com/Resource/Borough/Ma...
```

Обратите внимание, что мы сортируем строки на основе значения площади в порядке убывания, потому что мы хотим, чтобы самые большие полигоны шли первыми, а `sort_values` по умолчанию сортирует в порядке возрастания.

Теперь мы можем отправить отсортированные объекты в кластер и написать функцию, которая использует эти признаки, чтобы узнать, в каком из пяти районов (если таковые имеются) закончилась конкретная поездка:

```
b_gdf = spark.sparkContext.broadcast(gdf)

def find_borough(latitude,longitude):
    mgdf = b_gdf.value.apply(lambda x: x['borough'] if \
                             x['geometry'].\
                             intersects(gdp.\
                                        points_from_xy(
                                            [longitude], \
                                            [latitude])[0]) \
                             else None, axis=1)
    idx = mgdf.first_valid_index()
    return mgdf.loc[idx] if idx is not None else None

find_borough_udf = fun.udf(find_borough, StringType())
```

UDF или не UDF?

PySpark SQL позволяет очень легко преобразовывать бизнес-логику в функции, как мы это сделали здесь с функцией `find_borough`. Глядя на это, вы можете подумать, что было бы неплохо перенести всю вашу бизнес-логику в UDF (user defined function, пользовательская функция), чтобы упростить повторное использование, тестирование и обслуживание. Однако есть несколько предостережений по использованию UDF, о которых следует помнить, прежде чем вы начнете использовать их по всему коду:

- во-первых, пользовательские функции непрозрачны для планировщика запросов SQL и механизма выполнения PySpark, в отличие от стандартного синтаксиса запросов SQL, поэтому перемещение логики в пользовательские функции вместо использования буквального выражения SQL может снизить производительность запросов;
- во-вторых, обработка значений `null` в PySpark SQL может быстро усложниться, особенно для пользовательских функций, которые принимают несколько аргументов.

Мы можем применить функцию `find_borough` к поездкам в фрейме данных `taxi_raw`, чтобы создать гистограмму поездок по районам:

```
df_with_boroughs = taxi_raw.\
                    withColumn("dropoff_borough", \
                              find_borough_udf(
                                fun.col("dropoff_latitude"),\
                                fun.col('dropoff_longitude')))
df_with_boroughs.groupBy(fun.col("dropoff_borough")).count().show()
...
+-----------------------+--------+
|       dropoff_borough |   count|
+-----------------------+--------+
|                 Queens|  672192|
|                   null| 7942421|
|               Brooklyn|  715252|
|          Staten Island|    3338|
|             Manhattan|12979047|
|                  Bronx|   67434|
+-----------------------+--------+
```

Как мы и ожидали, подавляющее большинство поездок заканчивается в районе Манхэттена, в то время как относительно небольшое количество поездок заканчивается на Стейтен-Айленде. Обращает на себя внимание примечательный факт — это количество поездок, которые заканчиваются за пределами любого района; количество записей `null` значительно больше, чем количество поездок на такси, которые заканчиваются в Бронксе.

Недавно мы говорили об обработке таких недействительных записей, но не стали их удалять. В качестве упражнения вам остается удалить такие записи и построить гистограмму из очищенных данных. Сделав это, вы заметите уменьшение количества пустых записей, оставив гораздо более разумное количество наблюдений.

Поработав с геопространственными аспектами наших данных, давайте теперь углубимся во временную природу наших данных, выполнив сессионизацию с помощью PySpark.

Сессионизация в PySpark

Вид анализа, при котором мы исследуем один объект, переживающий серию событий с течением времени (сессию), называется *сессионизацией* (sessionization, иногда переводят как *сеансизация*) и обычно выполняется с помощью журналов веб-сервера для анализа поведения пользователей веб-сайта. PySpark располагает стандартными функциями библиотеки Window и агрегациями, которые можно использовать для выполнения такого анализа. Это позволяет нам сосредоточиться на логике анализа, вместо того чтобы пытаться реализовать сложные операции с данными и расчеты. Мы воспользуемся этой возможностью в следующем разделе, чтобы лучше проанализировать использование такси, представленное в нашем наборе данных.

Сессионизация может быть очень мощным методом для раскрытия информации в данных и создания новых продуктов на основе данных, которые можно использовать, чтобы помочь людям принимать более правильные решения. Например, механизм исправления орфографии Google построен на основе сеансов активности пользователей, которые Google создает каждый день из зарегистрированных записей о каждом событии (поиски, клики, посещения карт и т. д.), происходящем на его веб-ресурсах. Чтобы определить вероятных кандидатов на исправление орфографии, Google обрабатывает эти сеансы в поисках характерных ситуаций, когда пользователь ввел запрос, но не нажал клавишу ввода, потом через несколько секунд ввел немного исправленный запрос, а затем перешел по ссылке в результатах поиска и не вернулся к поисковику. При этом Google подсчитывает, как часто этот шаблон встречается для любой пары запросов. Если это происходит достаточно часто (например, если каждый раз, когда мы видим запрос «умиление звука», за ним через несколько секунд следует запрос «усиление звука»), то мы предполагаем, что второй запрос является исправлением первого.

Этот анализ использует шаблоны человеческого поведения, представленные в журналах событий, для создания механизма исправления орфографии на основе данных, который является более мощным, чем любой механизм, который можно было бы создать из словаря. Этот подход можно использовать для исправления правописания на любом языке и исправлять слова, которые могут отсутствовать в любом словаре (например, название нового стартапа), или такие запросы, как «умиление звука», в которых ни одно из слов формально не содержит ошибку! Google использует аналогичные методы, чтобы показать рекомендуемые и похожие поисковые запросы, а также решить, какие запросы должны возвращать результат OneBox, ко-

торый дает ответ на запрос на самой странице поиска, не требуя перехода на другую страницу. Есть готовые ответы OneBox для погоды, результатов спортивных мероприятий, адресов и множества других запросов.

Пока что информация о наборе событий, происходящих с каждым объектом, разбросана по фрейму данных, поэтому для анализа нам нужно разместить эти релевантные события рядом друг с другом и в хронологическом порядке. В следующем разделе мы покажем, как эффективно создавать и анализировать сеансы с использованием расширенных функций PySpark.

Создание сессий: вторичные сортировки в PySpark

Наивный способ создания сессий в PySpark — выполнить `groupBy` для идентификатора, для которого мы хотим создать сессию, а затем отсортировать события после перетасовки по идентификатору метки времени. Если у нас есть только небольшое количество событий для каждой сущности, этот подход будет работать достаточно хорошо. Однако, поскольку этот подход требует, чтобы все события для любой конкретной сущности находились в памяти одновременно, он перестанет работать после того, как количество событий для каждой сущности превысит определенный предел. Нам нужен способ построения сессий, который не требует одновременного хранения в памяти всех событий для конкретной сущности.

В MapReduce мы можем выстраивать сессии, выполняя *вторичную сортировку*, при которой мы создаем *составной ключ*, состоящий из идентификатора и значения метки времени, сортируем все записи по составному ключу, а затем используем настраиваемый разделитель и функцию группировки, чтобы все записи для одного и того же идентификатора появлялись в одном и том же выходном разделе. К счастью, PySpark может реализовать такой подход с помощью функций библиотеки `Window`:

```
from pyspark.sql import Window

window_spec = Window.partitionBy("hack_license").\
                     orderBy(fun.col("hack_license"),
                             fun.col("pickup_datetime"))
```

Сначала мы используем метод `partitionBy`, чтобы гарантировать, что все записи с одинаковым значением в столбце `license` попадут в один и тот же раздел. Затем в каждом из этих разделов мы сортируем записи по значению `license` (чтобы все поездки одного и того же водителя отображались вместе), а затем по `pickupTime` (время посадки пассажира), чтобы последовательность поездок отображалась в отсортированном порядке. Теперь, когда мы аггрегировали записи о поездках, мы можем быть уверены, что поездки упорядочены оптимальным образом для анализа сессий. Поскольку эта операция запускает перестановку записей и изрядное количество вычислений, и нам нужно будет использовать результаты более одного раза, мы кэшируем их:

```
window_spec.cache()
```

Выполнение конвейера сессионизации — вычислительно затратная операция, а полученное представление данных может пригодиться для множества различных задач анализа, которые мы, возможно, захотим выполнить позже. В условиях когда кто-то может решить продолжить анализ позже или сотрудничать с другими специалистами по данным, рекомендуется амортизировать стоимость сессионизации большого набора данных, выполнив сеанс только один раз, а затем записав сессионные данные в файловую систему, например, S3 или HDFS, чтобы их можно было использовать для ответа на множество разных вопросов. Однократное выполнение сессионизации — это также хороший способ применить стандартные правила формирования сессии во всей команде специалистов по обработке и анализу данных, чтобы обеспечить в будущем корректное сравнение «яблок с яблоками».

На данный момент мы готовы проанализировать данные наших сессий, чтобы увидеть, через сколько времени после остановки в определенном районе водитель получает следующий заказ. Мы будем использовать функцию `lag` (задержка) вместе с созданным ранее объектом `window_spec`, чтобы взять две поездки и вычислить интервал в секундах между временем высадки первой поездки и временем посадки во второй:

```
df_ with_ borough_durations = df_with_boroughs.\
        withColumn("trip_time_difference", \
        fun.col("pickup_datetime") - fun.lag(fun.col("pickup_datetime"),
                                    1). \
        over(window_spec)).show(50, vertical=True)
```

Теперь мы должны выполнить проверку, чтобы убедиться, что большинство полученных значений неотрицательные:

```
df_with_borough_durations.
  selectExpr("floor(seconds / 3600) as hours").
  groupBy("hours").
  count().
  sort("hours").
  show()
...
+-----+--------+
|hours|   count|
+-----+--------+
|   -3|       2|
|   -2|      16|
|   -1|    4253|
|    0|13359033|
|    1|  347634|
|    2|   76286|
|    3|   24812|
|    4|   10026|
|    5|    4789|
```

Лишь несколько записей имеют отрицательную продолжительность, но похоже между ними нет ничего общего, что позволило бы понять источник ошибочных данных. Если мы исключим эти записи с отрицательными значениями из нашего набора входных данных и посмотрим на среднее значение и стандартное отклонение времени посадки по районам, мы увидим следующее:

```
df_with_borough_durations.
  where("seconds > 0 AND seconds < 60*60*4").
  groupBy("borough").
  agg(avg("seconds"), stddev("seconds")).
  show()
...
+-------------+------------------+--------------------+
|      borough|      avg(seconds)|stddev_samp(seconds)|
+-------------+------------------+--------------------+
|       Queens|2380.6603554494727|  2206.6572799118035|
|           NA|  2006.53571169866|  1997.0891370324784|
|     Brooklyn| 1365.394576250576|  1612.9921698951398|
|Staten Island|         2723.5625|  2395.7745475546385|
|    Manhattan| 631.8473780726746|   1042.919915477234|
|        Bronx|1975.9209786770646|   1704.006452085683|
+-------------+------------------+--------------------+
```

Как и следовало ожидать, данные показывают, что на Манхэттене у водителей наименьшее время простоя — около 10 минут. Поездки на такси, которые заканчиваются в Бруклине, дают время простоя более чем в два раза больше, а относительно редкие поездки, которые заканчиваются в Стейтен-Айленде, отнимают у водителей в среднем почти 45 мин на ожидание следующего заказа.

Как показывают данные, у водителей такси есть серьезный финансовый стимул различать пассажиров в зависимости от их конечного пункта назначения. Высадки в Стейтен-Айленде, в частности, приводят к значительному простою в ожидании заказа. Комиссия по такси и лимузинам Нью-Йорка на протяжении многих лет прилагала большие усилия для выявления этой дискриминации и штрафовала водителей, которые отказывались от пассажиров из-за невыгодного пункта назначения. Было бы интересно попытаться изучить данные о необычно коротких поездках на такси, которые могут свидетельствовать о конфликте между водителем и пассажиром по поводу пункта назначения.

Что дальше?

В этой главе мы работали как с временны́ми, так и с пространственными характеристиками реального набора данных. Навыки геопространственного анализа, которые вы приобрели, можно использовать для освоения таких сред, как Apache

Sedona или GeoMesa. У них более крутая кривая обучения по сравнению с работой с GeoPandas и UDF, но они являются более эффективными. Существует также много интересных возможностей визуализации геопространственных и временны́х данных.

Кроме того, представьте, что вы используете анализ данных о такси, чтобы создать приложение, которое могло бы рекомендовать водителю лучшее место для стоянки такси после высадки пассажира на основе текущих моделей трафика и истории записей лучших мест, содержащихся в этих данных. Вы также можете посмотреть на информацию с точки зрения потенциального пассажира, который пытается поймать такси: учитывая текущее время, место и данные о погоде, какова вероятность того, что я смогу поймать такси на этой части улицы в течение следующих пяти минут? Подобную информацию можно включить в такие приложения, как Google Maps, чтобы помочь путешественникам решить, когда уехать и какой маршрут им следует выбрать.

ГЛАВА 8

Оценка финансового риска

Существует ли способ приблизительно оценить величину потерь при неудачном инвестировании на финансовом рынке? Количество, которое стремится измерить *экономическая стоимость риска* (value at risk, VaR). VaR — это простая мера инвестиционного риска, которая пытается дать разумную оценку максимальной вероятной потери стоимости инвестиционного портфеля за определенный период времени. Статистика VaR зависит от трех параметров: инвестиционного портфеля, периода времени и вероятности. Например, значение VaR, равное 1 млн долларов с вероятностью 5% и периодом в две недели, указывает на то, что вероятность потери портфелем более 1 млн долларов за две недели составляет всего 5%.

С момента своего появления, а это произошло вскоре после краха фондового рынка в 1987 г., критерий VaR получил широкое распространение в организациях, предоставляющих финансовые услуги. Статистика играет жизненно важную роль в управлении этими учреждениями, помогая определить характеристики риска их стратегий.

Многие из самых сложных подходов к оценке этой статистики основаны на ресурсоемких вычислениях моделей поведения рынков в случайных условиях. Техника, лежащая в основе этих подходов, называемая *моделированием Монте-Карло*, включает в себя воспроизведение тысяч или миллионов случайных рыночных сценариев и наблюдение за тем, как они влияют на портфель. Эти сценарии называются *испытаниями*. PySpark — идеальный инструмент для моделирования методом Монте-Карло. PySpark может использовать тысячи ядер для запуска случайных испытаний и объединения их результатов. Являясь механизмом преобразования данных общего назначения, он также способен выполнять этапы предварительной и последующей обработки, связанные с моделированием. Он может преобразовывать необработанные финансовые данные в параметры модели, необходимые для моделирования, а также выполнять специальный анализ результатов. Его простая модель программирования значительно сокращает время разработки по сравнению с более традиционными подходами, использующими среды высокопроизводительных вычислений.

Мы также обсудим, как рассчитать связанную статистику, называемую *условной стоимостью риска* (conditional value at risk, CVaR), известную ещё как *ожидаемые потери* (expected shortfall), которую Базельский комитет по банковскому надзору предложил несколько лет назад в качестве лучшей меры риска, чем VaR. Статистика CVaR имеет те же три параметра, что и статистика VaR, но рассматривает ожи-

даемый средний убыток, а не предоставляет значение вероятности заданного убытка. CVaR в размере 5 млн долларов США при 5%-ном *q-значении* на периоде в две недели свидетельствует об уверенности, что средний убыток в худших 5% результатов составляет 5 млн долларов США.

В процессе моделирования VaR вы освоите несколько различных концепций, подходов и пакетов. Мы начнем с базовой финансовой терминологии, которую будем использовать на протяжении всей главы, а затем расскажем о методах, используемых для расчета VaR, включая метод моделирования Монте-Карло. После этого мы загрузим и подготовим наш набор данных с помощью PySpark и pandas. В качестве примера мы будем использовать данные фондового рынка за конец 2000-х и начало 2010-х годов, включая рыночные индикаторы, такие, как цены на казначейские облигации и стоимость акций различных компаний. Завершив предварительную обработку данных, мы создадим модель линейной регрессии для расчета изменения стоимости акций за определенный период времени. Затем мы придумаем способ генерировать образцы значений рыночных индикаторов для использования в тестировании модели, выполненной по методу Монте-Карло. Наконец, мы выполним симуляцию с помощью PySpark и рассмотрим наши результаты.

Начнем с определения основных финансовых терминов, которые мы будем использовать.

Терминология

В этой главе встречаются следующие термины, принятые в финансовой отрасли:

Инструмент

Торговый актив, такой, как облигация, кредит, опцион или инвестиция в акции. В любой конкретный момент считается, что инструмент имеет *стоимость*, которая представляет собой цену, по которой он может быть продан.

Портфель

Набор инструментов, принадлежащих финансовому учреждению.

Доходность

Изменение стоимости инструмента или портфеля за определенный период времени.

Убыток

Отрицательная доходность.

Индекс

Воображаемый портфель инструментов. Например, составной индекс NASDAQ включает около 3000 акций и аналогичных инструментов крупных американских и международных компаний.

Рыночный фактор

Значение, которое можно использовать в качестве индикатора макроаспектов финансового климата в определенное время, например, значение индекса, валового внутреннего продукта Соединенных Штатов или обменного курса между долларом и евро. Для удобства будем называть рыночные факторы просто *факторами*.

Методы расчета VaR

До сих пор наше определение VaR было довольно размытым. Для получения этого показателя необходимо предложить модель функционирования инвестиционного портфеля и выбрать вероятностное распределение его доходности. Учреждения используют различные подходы для расчета VaR, каждый из которых, как правило, относится к одному из общепринятых методов.

Дисперсия-ковариация

Метод *дисперсии-ковариации*, безусловно, является самым простым и наименее ресурсоемким методом. Его модель предполагает нормальное распределение доходности каждого инструмента, что позволяет получить оценку чисто аналитически.

Ретроспективное моделирование

Ретроспективное моделирование (иногда говорят *историческое моделирование*) экстраполирует риск из ретроспективных данных, напрямую используя его распределение, а не сводную статистику. Например, чтобы определить 95% VaR для портфеля, мы можем посмотреть на доходность этого портфеля за последние 100 дней и оценить показатель, взяв пятое снизу значение доходности. Недостатком этого метода является то, что сценарии ретроспективных данных могут быть ограничены и не включают в себя ситуации «что если». Например, что если в истории инструментов в нашем портфеле отсутствуют обвалы рынка, а мы хотим смоделировать, что происходит с нашим портфелем в таких ситуациях? Существуют методы, позволяющие сделать ретроспективное моделирование устойчивым к этим проблемам, например, введение «шоков» в данные, но мы не будем их здесь рассматривать.

Моделирование методом Монте-Карло

Моделирование методом Монте-Карло, которому будет посвящена оставшаяся часть этой главы, пытается ослабить допущения предыдущих методов путем моделирования портфеля в случайных условиях. Когда мы не можем представить распределение вероятностей в виде аналитической формулы, но можем оценить его функцию плотности вероятности, повторно выбирая более простые случайные ве-

личины, от которых оно зависит, и наблюдая, как оно работает в совокупности. В самом общем виде этот метод:

- определяет взаимосвязь между рыночными условиями и доходностью каждого инструмента. Эта взаимосвязь представлена в виде модели, адаптированной к ретроспективным данным;
- определяет распределения для рыночных условий, из которых можно легко сделать выборку. Эти распределения подогнаны к ретроспективным данным;
- ставит испытания, состоящие из случайных рыночных условий;
- вычисляет общие потери портфеля для каждого испытания и использует эти потери для определения эмпирического распределения потерь. Это означает, что если мы проведем 100 испытаний и хотим оценить 5% VaR, мы выберем его как убыток от испытания с пятым по величине убытком. Чтобы рассчитать 5% CVaR, мы должны найти средние потери по пяти худшим испытаниям.

Конечно, метод Монте-Карло тоже не идеален. Он использует модели для создания условий испытаний и для вывода характеристик инструмента, и эти модели должны делать упрощающие допущения. Если эти допущения не соответствуют действительности, то и итоговое распределение вероятностей не будет соответствовать действительности.

Наша модель

Модель риска Монте-Карло обычно выражает доходность каждого инструмента с точки зрения набора рыночных факторов. Общими рыночными факторами могут быть значения таких индексов, как S&P 500, ВВП страны или обменные курсы валют. Значит, нам нужна модель, которая предсказывает доходность каждого инструмента на основе этих рыночных условий. Мы будем использовать простую линейную модель. Согласно нашему предыдущему определению доходности, *доходность фактора* (factor return) представляет собой изменение стоимости рыночного фактора в течение определенного времени. Например, если стоимость S&P 500 вырастет с 2000 до 2100 за рассматриваемый период времени, его доходность составит 100. Мы получим набор признаков из простых преобразований доходности факторов. То есть вектор рыночных факторов m_t для испытания t преобразуется некоторой функцией ϕ и дает нам вектор признаков f_t:

$$f_t = \phi(m_t).$$

Для каждого инструмента мы будем обучать модель, которая присваивает вес каждому признаку. Чтобы вычислить r_{it}, доходность инструмента i в испытании t, мы используем c_i — член пересечения линии регрессии для инструмента; w_{ij} — вес регрессии для признака j инструмента i; и f_{tj} — случайно сгенерированное значение признака j в испытании t:

$$r_{it} = c_i + \sum_{j=1}^{|w_i|} w_{ij} \cdot f_{tj}.$$

Это означает, что доходность каждого инструмента рассчитывается как сумма доходностей рыночных факторов, умноженных на их веса для этого инструмента. Мы можем настроить линейную модель для каждого инструмента, используя ретроспективные данные (иначе говоря, выполнить линейную регрессию). Если горизонт расчета VaR составляет две недели, регрессия рассматривает каждый (перекрывающийся) двухнедельный интервал в истории как выделенную точку.

Стоит также отметить, что мы могли бы выбрать более сложную модель. Например, модель не обязательно должна быть линейной: это может быть дерево регрессии или модель, явным образом включающая знания, относящиеся к предметной области.

Теперь, когда у нас есть модель для расчета потерь инструментов от рыночных факторов, нам нужен процесс моделирования поведения рыночных факторов. Простое предположение состоит в том, что доходность каждого рыночного фактора следует нормальному распределению. Чтобы зафиксировать тот факт, что рыночные факторы часто коррелируют друг с другом — когда NASDAQ падает, индекс Доу-Джонса, вероятно, также страдает, — мы можем использовать многомерное нормальное распределение с недиагональной ковариационной матрицей:

$$m_t \sim \mathcal{N}(\mu, \Sigma).$$

где μ — вектор эмпирических средних доходностей факторов, а Σ — эмпирическая ковариационная матрица доходностей факторов.

Как и прежде, мы могли бы выбрать более сложный метод моделирования рынка или предположить другой тип распределения для каждого рыночного фактора, возможно, используя распределения с более толстыми хвостами.

Получение данных

Скачайте ретроспективный набор данных о ценах на акции и поместите его в каталог `data/stocks/`:

```
$ mkdir stocks && cd stocks
$ url="https://raw.githubusercontent.com/ \
        sryza/aas/master/ch09-risk/data/stocks.zip"
$ wget $url
$ unzip stocks.zip
```

Довольно трудно найти большие объемы хорошо отформатированных ретроспективных ценовых данных. Набор данных, используемый в этой главе, был загружен с Yahoo!

Нам также нужны ретроспективные данные о факторах риска. Для наших коэффициентов мы будем использовать следующие значения:

- ETF 20-летних и более кредитных облигаций США iShares (NASDAQ: TLT);
- ETF кредитных облигаций США iShares (NYSEArca: CRED);
- Золотой фонд SPDR (NYSEArca: GLD).

Скачайте и сохраните данные о факторах:

```
$ cd .. && mkdir factors && cd factors
$ url2 = "https://raw.githubusercontent.com/ \
          sryza/aas/master/ch09-risk/data/factors.zip"
$ wget $url2
$ unzip factors.zip
$ ls factors
...

NASDAQ%3ATLT.csv NYSEARCA%3ACRED.csv NYSEARCA%3AGLD.csv
```

Давайте посмотрим, что представляет собой один из наших факторов:

```
$ !head -n 5 data/factors/NASDAQ%3ATLT.csv
...

Date,Open,High,Low,Close,Volume
31-Dec-13,102.29,102.55,101.17,101.86,7219195
30-Dec-13,102.15,102.58,102.08,102.51,4491711
27-Dec-13,102.07,102.31,101.69,101.81,4755262
26-Dec-13,102.35,102.36,102.01,102.10,4645323
24-Dec-13,103.23,103.35,102.80,102.83,4897009
```

Загрузив набор данных, займемся его подготовкой.

Подготовка данных

Первые несколько строк данных в формате Yahoo! для GOOGL выглядят следующим образом:

```
$ !head -n 5 data/stocks/GOOGL.csv
...

Date,Open,High,Low,Close,Volume
31-Dec-13,556.68,561.06,553.68,560.92,1358300
30-Dec-13,560.73,560.81,555.06,555.28,1236709
27-Dec-13,560.56,560.70,557.03,559.76,1570140
26-Dec-13,557.56,560.06,554.90,559.29,1338507
24-Dec-13,558.04,558.18,554.60,556.48,734170
```

Запустите оболочку PySpark:

```
$ pyspark --driver-memory 4g
```

Прочитайте набор данных инструментов как DataFrame:

```
stocks = spark.read.csv("data/stocks/", header='true',
                        inferSchema='true')

stocks.show(2)
...
```

```
+----------+----+----+----+-----+------+
|      Date|Open|High| Low|Close|Volume|
+----------+----+----+----+-----+------+
|2013-12-31|4.40|4.48|3.92| 4.07|561247|
|2013-12-30|3.93|4.42|3.90| 4.38|550358|
+----------+----+----+----+-----+------+
```

В фрейме данных отсутствует обозначение инструмента (так называемый биржевой символ). Добавим, его используя в качестве источника имена файлов, соответствующие каждой строке:

```
from pyspark.sql import functions as fun

stocks = stocks.withColumn("Symbol", fun.input_file_name()).\
                withColumn("Symbol",
                  fun.element_at(fun.split("Symbol", "/"), -1)).\
                withColumn("Symbol",
                  fun.element_at(fun.split("Symbol", "\."), 1))

stocks.show(2)
...
+---------+-------+-------+-------+-------+------+------+
|     Date|   Open|   High|    Low|  Close|Volume|Symbol|
+---------+-------+-------+-------+-------+------+------+
|31-Dec-13|1884.00|1900.00|1880.00| 1900.0|   546|  CLDN|
|30-Dec-13|1889.00|1900.00|1880.00| 1900.0|  1656|  CLDN|
+---------+-------+-------+-------+-------+------+------+
```

Прочитаем и обработаем набор данных факторов аналогичным образом:

```
factors = spark.read.csv("data/factors", header='true', inferSchema='true')
factors = factors.withColumn("Symbol", fun.input_file_name()).\
                  withColumn("Symbol",
                    fun.element_at(fun.split("Symbol", "/"), -1)).\
                  withColumn("Symbol",
                    fun.element_at(fun.split("Symbol", "\."), 1))
```

Отфильтруем инструменты с менее чем пятилетней историей:

```
from pyspark.sql import Window

stocks = stocks.withColumn('count', fun.count('Symbol').\
                over(Window.partitionBy('Symbol'))).\
                filter(fun.col('count') > 260*5 + 10)
```

Различные типы инструментов могут торговаться в разные дни, или данные могут иметь отсутствующие значения по другим причинам, поэтому важно убедиться, что разные ретроспективные данные совпадают. Сначала нам нужно обрезать все наши временны́е ряды до одного и того же периода времени. Для этого мы сначала преобразуем тип столбца `Date` из строкового в дату:

```
stocks = stocks.withColumn('Date',
                  fun.to_date(fun.to_timestamp(fun.col('Date'),
                                               'dd-MM-yy')))
```

```
stocks.printSchema()
...
root
 |-- Date: date (nullable = true)
 |-- Open: string (nullable = true)
 |-- High: string (nullable = true)
 |-- Low: string (nullable = true)
 |-- Close: double (nullable = true)
 |-- Volume: string (nullable = true)
 |-- Symbol: string (nullable = true)
 |-- count: long (nullable = false)
```

Обрежем временные периоды инструментов для совмещения:

```
from datetime import datetime

stocks = stocks.filter(fun.col('Date') >= datetime(2009, 10, 23)).\
                filter(fun.col('Date') <= datetime(2014, 10, 23))
```

Преобразуем тип столбца `Date` и обрежем период времени в нашем фрейме данных факторов:

```
factors = factors.withColumn('Date',
                             fun.to_date(fun.to_timestamp(fun.col('Date'),
                                                          'dd-MMM-yy')))

factors = factors.filter(fun.col('Date') >= datetime(2009, 10, 23)).\
                  filter(fun.col('Date') <= datetime(2014, 10, 23))
```

Ретроспективные данные нескольких тысяч инструментов и трех факторов достаточно малы, чтобы их можно было прочитать и обработать локально. Это справедливо даже для больших симуляций с сотнями тысяч инструментов и тысячами факторов. Несмотря на то что до сих пор мы использовали PySpark для предварительной обработки наших данных, при фактическом запуске моделирования потребности распределенной системы, такой, как PySpark, могут привести к огромному объему вычислений для каждого инструмента. Мы можем преобразовать наш объект PySpark DataFrame в pandas DataFrame и по-прежнему легко продолжать с ним работать, выполняя операции в памяти.

```
stocks_pd_df = stocks.toPandas()
factors_pd_df = factors.toPandas()

factors_pd_df.head(5)
...
        Date Open High Low Close Volume Symbol
0       2013-12-31 102.29 102.55 101.17 101.86 7219195
    NASDAQ%253ATLT
1       2013-12-30 102.15 102.58 102.08 102.51 4491711
    NASDAQ%253ATLT
2       2013-12-27 102.07 102.31 101.69 101.81 4755262
    NASDAQ%253ATLT
```

```
3       2013-12-26 102.35 102.36 102.01 102.10 4645323
   NASDAQ%253ATLT
4       2013-12-24 103.23 103.35 102.80 102.83 4897009
   NASDAQ%253ATLT
```

Мы будем использовать эти фреймы данных pandas в следующем разделе, когда попытаемся подобрать модель линейной регрессии для прогнозирования доходности инструмента на основе доходности факторов.

Определение весов факторов

Напомним, что VaR имеет дело с убытками *за определенный период времени*. Нас интересуют не абсолютные цены инструментов, а то, как эти цены меняются в течение заданного промежутка времени. В наших расчетах мы примем длительность этого периода равной двум неделям. Следующая функция использует метод `rolling` библиотеки pandas для преобразования временнóго ряда цен в перекрывающуюся последовательность ценовых изменений за двухнедельные интервалы. Обратите внимание, что мы используем 10 вместо 14 для определения окна данных, потому что финансовые данные не включают выходные дни:

```
n_steps = 10
def my_fun(x):
    return ((x.iloc[-1] - x.iloc[0]) / x.iloc[0])

stock_returns = stocks_pd_df.groupby('Symbol').Close.\
                                    rolling(window=n_steps).apply(my_fun)
factors_returns = factors_pd_df.groupby('Symbol').Close.\\
                                    rolling(window=n_steps).apply(my_fun)

stock_returns = stock_returns.reset_index().\
                              sort_values('level_1').\
                              reset_index()
factors_returns = factors_returns.reset_index().\
                                  sort_values('level_1').\
                                  reset_index()
```

Получив эти истории доходности, мы можем перейти к нашей цели — обучению прогностических моделей доходности инструмента. Для каждого инструмента нам нужна модель, которая предсказывает его двухнедельную доходность на основе доходности факторов за тот же период времени. Для простоты мы будем использовать модель линейной регрессии.

Чтобы смоделировать тот факт, что доходность инструмента может быть нелинейной функцией доходности фактора, мы можем включить в нашу модель некоторые дополнительные признаки, которые мы получаем из нелинейных преобразований доходности фактора. В качестве примера мы добавим один дополнительный признак для каждого коэффициента доходности: возведение в квадрат. Наша модель по-прежнему является линейной моделью в том смысле, что переменная отклика

является линейной функцией признаков. Некоторые признаки просто определяются нелинейными функциями доходности факторов. Имейте в виду, что это конкретное преобразование признака предназначено для демонстрации некоторых доступных вариантов — его не следует воспринимать как передовую практику прогнозного финансового моделирования.

```
# Создаем комбинированный фрейм данных инструментов
stocks_pd_df_with_returns = stocks_pd_df.\
                              assign(stock_returns = \
                                     stock_returns['Close'])

# Создаем комбинированный фрейм данных факторов
factors_pd_df_with_returns = factors_pd_df.\
                              assign(factors_returns = \
                                     factors_returns['Close'],
                                     factors_returns_squared = \
                                     factors_returns['Close']**2)

factors_pd_df_with_returns = factors_pd_df_with_returns.\
                              pivot(index='Date',
                                     columns='Symbol',
                                     values=['factors_returns', \
                                             'factors_returns_squared'])❶

factors_pd_df_with_returns.columns = factors_pd_df_with_returns.\
                                        columns.\
                                        to_series().\
                                        str.\
                                        join('_').\
                                        reset_index()[0] ❷

factors_pd_df_with_returns = factors_pd_df_with_returns.\
                                reset_index()

print(factors_pd_df_with_returns.head(1))
...
0        Date factors_returns_NASDAQ%253ATLT \
0  2009-10-23                       0.01834

0  factors_returns_NYSEARCA%253ACRED
0                          -0.006594

0  factors_returns_NYSEARCA%253AGLD \
0                       - 0.032623

0  factors_returns_squared_NASDAQ%253ATLT \
0                                0.000336
```

```
0  factors_returns_squared_NYSEARCA%253ACRED \
0                                   0.000043

0 factors_returns_squared_NYSEARCA%253AGLD
0                                  0.001064
...

print(factors_pd_df_with_returns.columns)
...
Index(['Date', 'factors_returns_NASDAQ%253ATLT',
       'factors_returns_NYSEARCA%253ACRED',
       'factors_returns_NYSEARCA%253AGLD',
       'factors_returns_squared_NASDAQ%253ATLT',
       'factors_returns_squared_NYSEARCA%253ACRED',
       'factors_returns_squared_NYSEARCA%253AGLD'],
      dtype='object', name=0)
...
```

❶ Преобразование фрейма данных факторов из формата long в wide, чтобы все факторы за период времени находились в одной строке.

❷ Выравнивание многоиндексного фрейма данных и исправление имен столбцов.

Несмотря на то что мы будем выполнять много регрессий — по одной для каждого инструмента — количество признаков и точек данных в каждой регрессии невелико, а это означает, что нам не нужно использовать возможности распределенного линейного моделирования PySpark. Вместо этого мы будем использовать обычную регрессию методом наименьших квадратов, предлагаемую пакетом `scikit-learn`:

```
from sklearn.linear_model import LinearRegression

# Для каждого инструмента создаем входной фрейм данных
# для обучения модели линейной регрессии

stocks_factors_combined_df = pd.merge(stocks_pd_df_with_returns,
                                      factors_pd_df_with_returns,
                                      How="left", on="Date")

feature_columns = list(stocks_factors_combined_df.columns[-6:])

with pd.option_context('mode.use_inf_as_na', True):
    stocks_factors_combined_df = stocks_factors_combined_df.\
                                     dropna(subset=feature_columns \
                                            + ['stock_returns'])

def find_ols_coef(df):
    y = df[['stock_returns']].values
    X = df[feature_columns]
```

```
    regr = LinearRegression()
    regr_output = regr.fit(X, y)

    return list(df[['Symbol']].values[0]) + \
                list(regr_output.coef_[0])

coefs_per_stock = stocks_factors_combined_df.\
                      groupby('Symbol').\
                      apply(find_ols_coef)

coefs_per_stock = pd.DataFrame(coefs_per_stock).reset_index()
coefs_per_stock.columns = ['symbol', 'factor_coef_list']

coefs_per_stock = pd.DataFrame(coefs_per_stock.\
                                factor_coef_list.tolist(),
                                index=coefs_per_stock.index,
                                columns = ['Symbol'] + feature_columns)

coefs_per_stock
```

Теперь у нас есть фрейм данных, где каждая строка представляет собой набор параметров модели (коэффициенты, веса, коварианты, регрессоры или пр.) для инструмента.

На этом этапе в любом реальном конвейере было бы полезно понять, насколько хорошо эти модели соответствуют данным. Поскольку точки данных взяты из временны́х рядов, и особенно из-за того, что временны́е интервалы перекрываются, очень вероятно, что выборки автокоррелированы. Это означает, что обычные меры, такие, как R^2, скорее всего, будут завышать оценку того, насколько хорошо модели соответствуют данным. Для оценки этих эффектов обычно применяют тест Бреуша-Годфри. Разделите временно́й ряд на два набора, оставляя в середине достаточно точек данных, чтобы последние точки в более раннем наборе не автокоррелировали с первыми точками в более позднем наборе. Затем обучите модель на одном наборе и оцените ее ошибку на другом.

Итак, у нас есть модели, которые сопоставляют доходность фактора с доходностью инструмента, и теперь нам нужна процедура для имитации рыночных условий путем создания случайной доходности фактора. Этому посвящен следующий раздел.

Выборка

Чтобы придумать способ генерации случайных доходностей факторов, нам нужно определиться с распределением вероятностей по векторам доходностей факторов и произвести выборку из него. Но как понять, какому распределению соответствуют фактические данные? Часто бывает полезно начать с визуального представления данных.

Хороший способ визуализировать распределение вероятностей по непрерывным данным — это график плотности, который отображает область распределения в зависимости от его функции плотности вероятности. Поскольку мы не знаем распределения, управляющего данными, у нас нет уравнения, которое могло бы дать нам его плотность в произвольной точке, но мы можем аппроксимировать его с помощью метода, называемого *ядерной оценкой плотности* (kernel density estimation, KDE). В широком смысле ядерная оценка плотности — это способ сглаживания гистограммы. Он центрирует распределение вероятностей (обычно нормальное распределение) в каждой точке данных. Таким образом, набор двухнедельных выборок дает несколько нормальных распределений, каждое из которых имеет свое среднее значение. Чтобы оценить плотность вероятности в данной точке, оценивают плотность распределения вероятностей (probability density function, PDF) всех нормальных распределений в этой точке и берут их среднее значение. Гладкость графика, полученного методом KDE, зависит от его полосы пропускания — стандартного отклонения каждого из нормальных распределений.

Мы будем использовать один из встроенных методов pandas для расчета и построения графика KDE. Следующий фрагмент кода создает график плотности распределения для одного из наших факторов:

```
samples = factors_returns.loc[factors_returns.Symbol == \
                              factors_returns.Symbol.unique()[0]]['Close']
samples.plot.kde()
```

На рис. 8.1 показано распределение (функция плотности вероятности) двухнедельной доходности казначейских облигаций со сроком погашения более 20 лет (20+ Year Treasury Bond ETF) из нашего набора данных.

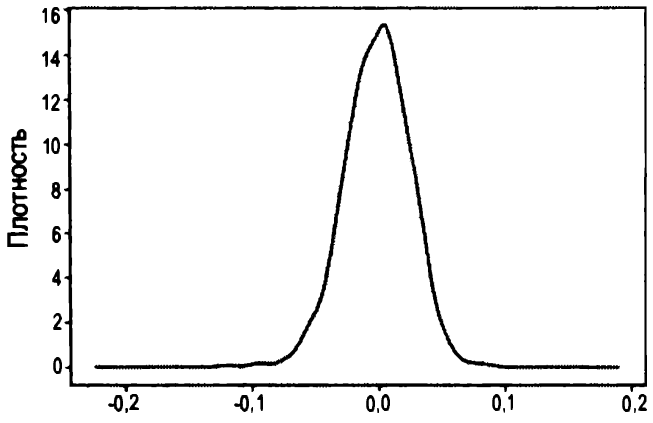


Рис. 8.1. Распределение двухнедельной доходности казначейских облигаций со сроком погашения более 20 лет

На рис. 8.2 то же самое распределение для двухнедельной доходности кредитных облигаций США (US Credit Bond ETF).

Мы подгоним нормальное распределение к доходам каждого фактора. Часто имеет смысл поиск более экзотического распределения, возможно, с более толстыми хвостами, которое точнее соответствует данным. Однако ради простоты мы не рассматриваем здесь такие распределения.

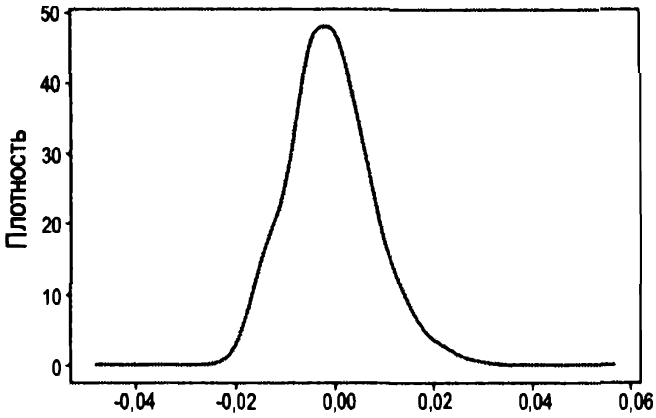


Рис. 8.2. Распределение двухнедельной доходности кредитных облигаций США

В качестве простейшего способа выборки доходности факторов можно было бы подобрать нормальное распределение для каждого из факторов и провести независимую выборку из этих распределений. Однако при этом игнорируется тот факт, что рыночные факторы часто взаимосвязаны. Например, если Treasury Bond ETF упадет, Credit Bond ETF, скорее всего, тоже упадет. Неспособность принять во внимание эти корреляции может дать нам гораздо более радужную картину нашего профиля риска, чем в реальности. Коррелируют ли доходности наших факторов? Реализация корреляции Пирсона в pandas поможет нам ответить на этот вопрос:

```
f_1 = factors_returns.loc[factors_returns.Symbol == \
                          factors_returns.Symbol.unique()[0]]['Close']
f_2 = factors_returns.loc[factors_returns.Symbol == \
                          factors_returns.Symbol.unique()[1]]['Close']
f_3 = factors_returns.loc[factors_returns.Symbol == \
                          factors_returns.Symbol.unique()[2]]['Close']
pd.DataFrame({'f1': list(f_1), 'f2': list(f_2), 'f3': list(f_3)}).corr()
...
```

	f1	f2	f3
f1	1.000000	0.530550	0.074578
f2	0.530550	1.000000	0.206538
f3	0.074578	0.206538	1.000000

Поскольку у нас есть ненулевые элементы вне диагоналей, похоже, что это не так.

Многомерное нормальное распределение

С учетом информации о корреляции между факторами нам может помочь многомерное нормальное распределение. Каждая выборка из такого распределения является вектором. Если взять какое-то одно измерение, распределение значений по этому измерению является нормальным. Но в своем совместном распределении переменные не являются независимыми.

Многомерное нормальное распределение параметризовано средним значением по каждому измерению и матрицей, описывающей ковариации между каждой парой измерений. При N измерениях ковариационная матрица имеет размерность N на N, потому

что нам нужно зафиксировать ковариации между каждой парой измерений. Когда ковариационная матрица является диагональной, многомерное нормальное распределение сводится к выборке независимо по каждому измерению, но наличие ненулевых значений вне диагонали помогает установить отношения между переменными.

В литературе по VaR часто описывают этап, на котором веса факторов преобразуются (декоррелируются), чтобы можно было продолжить выборку. Обычно это достигается с помощью *разложения Холецкого* или *разложения по собственным векторам* (eigendecomposition). `MultivariateNormalDistribution` из пакета NumPy позаботится об этом шаге за нас, применяя разложение по собственным векторам.

Чтобы подогнать многомерное нормальное распределение к нашим данным, сначала нам нужно найти его выборочные средние значения и ковариации:

```
factors_returns_cov = pd.DataFrame({'f1': list(f_1),
                                    'f2': list(f_2[:-1]),
                                    'f3': list(f_3[:-2])})\
                                    .cov().to_numpy()
factors_returns_mean = pd.DataFrame({'f1': list(f_1),
                                     'f2': list(f_2[:-1]),
                                     'f3': list(f_3[:-2])}).\
                                     mean()
```

Затем мы можем просто создать распределение, параметризованное с их помощью, и выбрать из него набор рыночных условий:

```
from numpy.random import multivariate_normal

multivariate_normal(factors_returns_mean, factors_returns_cov)
...
array([ 0.02234821, 0.01838763, -0.01107748])
```

Теперь у нас есть модели для каждого инвестиционного инструмента и процедура выборки факторного дохода, и мы можем приступить к испытанию моделей. Давайте запустим симуляцию рынка и проведем испытания.

Испытание моделей

Поскольку испытание моделей требует больших вычислительных ресурсов, мы обратимся к PySpark, чтобы он помог распараллелить вычисления. В каждом испытании нам нужно выбрать набор факторов риска, использовать их для прогнозирования доходности каждого инструмента и суммировать все эти доходности, чтобы найти полные убытки по испытанию. Чтобы добиться репрезентативного распределения, мы должны провести тысячи или миллионы таких испытаний.

У нас есть несколько вариантов распараллеливания моделирования. Мы можем распараллелить испытания, инструменты или и то, и другое. Чтобы распараллелить оба процесса, мы должны создать набор данных инструментов и набор данных параметров испытаний, а затем использовать преобразование `crossJoin` для создания набора данных всех пар. Это самый очевидный и универсальный подход, но он

имеет пару недостатков. Во-первых, требуется явное создание фрейма данных параметров испытаний, чего можно избежать, используя некоторые приемы со случайными начальными значениями. Во-вторых, требуется операция перемешивания.

Разбиение по инструментам будет выглядеть примерно так:

```
random_seed = 1496
instruments_dF = ...
def trialLossesForInstrument(seed, instrument):
  ...

instruments_DF.rdd.\
  flatMap(trialLossesForInstrument(random_seed, _)).\
  reduceByKey(_ + _)
```

При таком подходе данные распределяются по фрейму данных инструментов, и для каждого инструмента преобразование `flatMap` вычисляет и выдает потери по каждому испытанию. Использование одного и того же начального числа случайной последовательности для всех задач означает, что мы будем генерировать одну и ту же последовательность испытаний. `reduceByKey` суммирует все потери, соответствующие одним и тем же испытаниям. Недостатком этого подхода является то, что он по-прежнему требует перетасовки данных $O(|\text{инструменты}| * |\text{испытания}|)$.

Данные наших моделей для нескольких тысяч инструментов достаточно малы, чтобы поместиться в памяти каждой машины, и прикидочные расчеты показывают, что это будет справедливо даже для миллионов инструментов и сотней факторов. Миллион инструментов, умноженный на 500 факторов и умноженный на 8 байтов, необходимых для переменной двойной точности, в которой хранится вес каждого фактора, составляет примерно 4 ГБ, что по сегодняшним меркам достаточно мало, и позволяет разместить данные в каждом исполнителе на большинстве современных кластерных машин. Это означает, что хорошим вариантом является рассылка данных инструмента в широковещательной переменной. Преимущество наличия у каждого исполнителя полной копии данных инструмента заключается в том, что общие потери для каждого испытания могут быть рассчитаны на одной машине. Агрегация не требуется. Мы также рассылаем некоторые другие данные, необходимые для расчета доходности в испытании.

```
b_coefs_per_stock = spark.sparkContext.broadcast(coefs_per_stock)
b_feature_columns = spark.sparkContext.broadcast(feature_columns)
b_factors_returns_mean = spark.sparkContext.broadcast(factors_returns_mean)
b_factors_returns_cov = spark.sparkContext.broadcast(factors_returns_cov)
```

Взяв на вооружение подход разделения по испытаниям (который мы будем использовать), мы начинаем с фрейма данных начальных значений рандомизатора. В каждом разделе должно быть свое начальное число, чтобы каждый раздел генерировал разные испытания:

```
from pyspark.sql.types import IntegerType

parallelism = 1000
num_trials = 1000000
base_seed = 1496
```

```
seeds = [b for b in range(base_seed,
                          base_seed + parallelism)]
seedsDF = spark.createDataFrame(seeds, IntegerType())

seedsDF = seedsDF.repartition(parallelism)
```

Генерация случайных чисел — процесс, требующий много времени и ресурсов процессора. Хотя мы не используем здесь этот трюк, но бывает полезно заранее сгенерировать набор случайных чисел и использовать его в нескольких заданиях. Однако помните, что одни и те же случайные числа не должны использоваться в рамках одного задания, поскольку это нарушит предположение Монте-Карло о независимом распределении случайных значений.

Для каждого начального значения мы должны создать набор испытательных параметров и наблюдать за влиянием этих параметров на все инструменты. Мы напишем функцию, которая вычисляет полную доходность инструментов для нескольких испытаний. Начнем с применения линейной модели, которую мы обучили ранее для каждого инструмента. Затем мы находим простое среднее значение доходности всех инструментов. Это предполагает, что мы располагаем одинаковой стоимостью каждого инструмента в портфеле. Если бы у нас в портфеле были акции на разные суммы, то мы использовали бы не среднее, а средневзвешенное значение доходности. Наконец, нам нужно сгенерировать множество испытаний в каждой задаче. Поскольку выбор случайных чисел является существенной частью процесса, важно использовать мощный генератор случайных чисел. Встроенная библиотека случайных чисел Python включает в себя реализацию алгоритма Mersenne Twister (Вихрь Мерсенна), который подходит для этой задачи. Мы используем его для выборки из многомерного нормального распределения, как описано ранее:

```
import random

from pyspark.sql.types import LongType, ArrayType
from pyspark.sql.functions import udf

def calculate_trial_return(x):
#     return x
    trial_return_list = []

    for i in range(num_trials/parallelism):
        random_int = random.randint(0, num_trials*num_trials)

        seed(x)

       random_factors = multivariate_normal(b_factors_returns_mean.value,
         b_factors_returns_cov.value)

        coefs_per_stock_df = b_coefs_per_stock.value
        returns_per_stock = coefs_per_stock_df[b_feature_columns.value] *
          (list(random_factors) + list(random_factors**2))
```

```
        trial_return_list.append(float(returns_per_stock.sum(axis=1)
                                                          .sum()/b_coefs_
          per_stock.value.size))

    return trial_return_list

udf_return = udf(calculate_trial_return, ArrayType(DoubleType()))
```

Завершив разработку всех элементов кода, мы можем использовать его для вычисления фрейма данных, где каждый элемент представляет собой совокупный доход от одного испытания:

```
from pyspark.sql.functions import col, explode

trials = seedsDF.withColumn("trial_return", udf_return(col("value")))
trials = trials.select('value', explode('trial_return'))❶

trials.cache()
```

❶ Разделение массива доходов на отдельные строки фрейма данных.

Как вы помните, единственная причина, по которой мы возились со всеми этими числами, заключается в том, чтобы рассчитать VaR. Фрейм `trials` теперь содержит эмпирическое распределение доходности портфеля. Чтобы рассчитать 5% VaR, в нашем эмпирическом распределении нужно найти такое значение, относительно которого 5% испытаний хуже, а 95% испытаний лучше. Мы можем добиться этого, просто отделив подмножество из 5% худших испытаний. Наш VaR — результат лучшего испытания в этом подмножестве:

```
trials.approxQuantile('trial_return', [0.05], 0.0)
...
-0.010831826593164014
```

Мы можем найти CVaR почти идентичным способом. Вместо того чтобы брать наилучший доход в подмножестве из 5% худших испытаний, мы берем средний доход от этого набора испытаний:

```
trials.orderBy(col('trial_return').asc()).\
  limit(int(trials.count()/20)).\
  agg(fun.avg(col("trial_return"))).show()
...
+--------------------+
|   avg(trial_return)|
+--------------------+
|-0.09002629251426077|
+--------------------+
```

Визуализация распределения доходов

В дополнение к расчету VaR (с определенным уровнем достоверности) может быть полезным взглянуть на более полную картину распределения доходов. Соответствует ли она нормальному распределению? Нет ли выбросов на концах? Мы можем построить график функции плотности вероятности для совместного распределения вероятностей, используя ядерную оценку плотности (рис. 8.3), как мы делали это раньше для факторов:

```
trials.plot.kde()
```

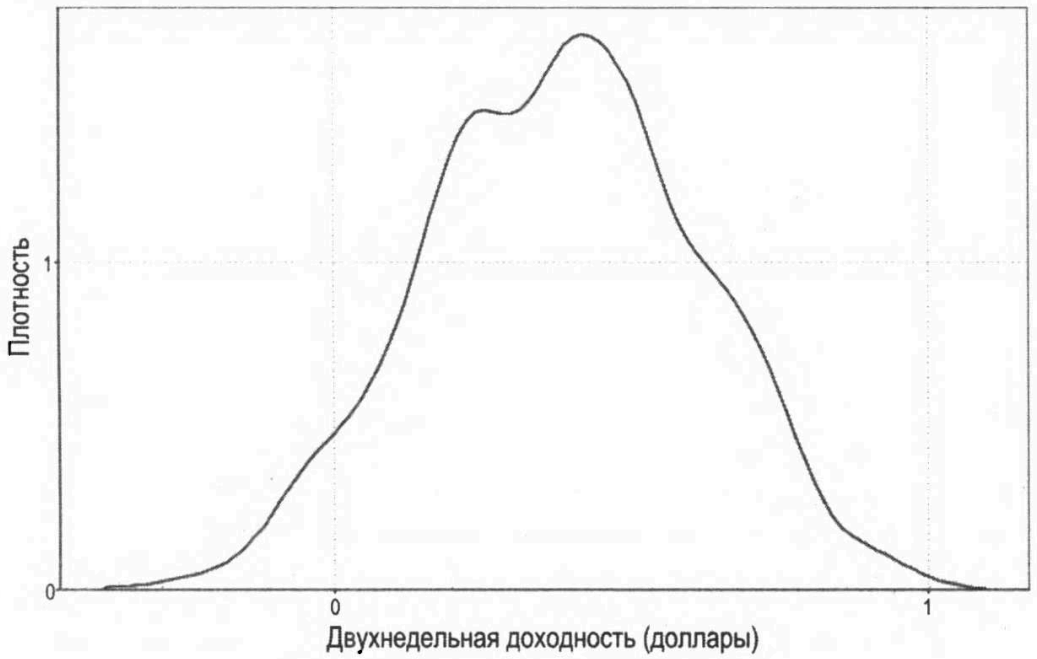


Рис. 8.3. Распределение доходов за двухнедельный период

Что дальше?

Модель, представленная в этом упражнении, представляет собой очень приближенную и упрощенную версию того, что можно было бы использовать в реальном финансовом учреждении. Мы упустили несколько очень важных шагов, необходимых для построения точной модели VaR. Ручное формирование ограниченного набора рыночных факторов может создать или разрушить модель, а финансовые учреждения нередко включают сотни факторов в свои модели. Выбор этих факторов требует как проведения многочисленных экспериментов с ретроспективными данными, так и изрядного творческого подхода. Не менее важен и выбор прогностической модели, которая сопоставляет рыночные факторы с доходностью инструментов. Хотя мы использовали простую линейную модель, во многих расчетах используются нелинейные функции или моделируется путь по принципу броуновского движения.

Наконец, стоит уделить внимание распределению, которое мы используем для моделирования доходности факторов. Для проверки нормальности эмпирического

распределения полезно применять критерии Колмогорова-Смирнова и хи-квадрат (критерий Пирсона). Для визуального сравнения распределений можно использовать диаграмму «квантиль-квантиль». Надо заметить, что обычно финансовый риск лучше отражает распределение с более толстыми хвостами, чем нормальное распределение, которое мы использовали. Смеси нормальных распределений — хороший способ получить эти более толстые хвосты. Статья Маркуса Хааса и Кристиана Пигорша «Распределения с толстым хвостом в финансовой экономике»[1] содержит полезную справочную информацию по некоторым другим распределениям с толстым хвостом.

Банки тоже используют PySpark и крупномасштабные платформы обработки данных для расчета VaR с использованием ретроспективных методов. В статье «Оценка моделей стоимости под риском с использованием ретроспективных данных»[2] Дэррилла Хендрикса приведен хороший обзор и сравнение эффективности ретроспективных методов VaR.

Моделирование рисков методом Монте-Карло применимо не только для расчета одиночного статистического показателя. Результаты можно использовать для упреждающего снижения риска портфеля путем формирования инвестиционных решений. Например, если в испытаниях с самой низкой доходностью мы видим, что определенный набор инструментов регулярно приводит к потере дохода, стоит задуматься об исключении этих инструментов из портфеля или добавлении инструментов, которым во время колебаний рынка свойственно двигаться в противоположном направлении.

---

[1] URL: **https://link.springer.com/referenceworkentry/10.1007/978-0-387-30440-3_204**.

[2] URL: **https://www.newyorkfed.org/medialibrary/media/research/epr/96v02n1/9604hend.pdf**.

ГЛАВА 9

Анализ данных геномики и проект BDG

Появление технологии секвенирования ДНК нового поколения (next-generation sequencing, NGS) быстро превратило науки о жизни в отрасль, основанную на данных. Однако попытки эффективного использования этих данных упираются в традиционную вычислительную экосистему, основанную на сложных в использовании низкоуровневых примитивах для распределенных вычислений и запутанных джунглях плохо структурированных текстовых форматов файлов.

Эта глава будет служить двум основным целям. Во-первых, мы рассмотрим набор популярных форматов сериализации и файлов (Avro и Parquet), которые упрощают многие проблемы в управлении данными. Эти технологии сериализации позволяют нам преобразовывать данные в компактные, удобные для машин двоичные представления. Благодаря этому становится намного легче перемещать данные по сетям и обеспечивать кросс-совместимость между языками программирования. Хотя мы будем применять методы сериализации к данным геномики, эти подходы пригодятся вам при обработке больших объемов любых данных.

Во-вторых, мы покажем, как решать типичные задачи геномики в экосистеме PySpark. В частности, мы будем использовать PySpark и библиотеку ADAM с открытым исходным кодом, чтобы оперировать большими объемами геномных данных из нескольких источников и сформировать набор данных для прогнозирования *сайтов связывания*[1] *транскрипционных факторов*[2] (transcription factor, TF). Для этого мы соединим аннотации генома из набора данных ENCODE. Эта глава будет служить учебным пособием по работе с проектом ADAM, который включает в себя набор специфичных для геномики схем Avro, API-интерфейсы на основе PySpark и инструменты командной строки для крупномасштабного анализа геномных данных. Помимо других приложений, ADAM содержит встроенную реализацию набора инструментов для анализа генома (Genome Analysis Toolkit, GATK) с использованием PySpark.

Мы начнем с обсуждения различных форматов данных, используемых в области биоинформатики, связанных с ними проблем и того, как нам могут помочь правильно выбранные форматы сериализации. После этого мы установим проект ADAM и изучим его API, используя пример набора данных. Затем мы приступим к

[1] Последовательность ДНК, с которой связываются траснкрипционные факторы. — *Примеч. пер.*

[2] Белки́, контролирующие процесс синтеза мРНК на матрице ДНК (транскрипцию). — *Примеч. пер.*

работе с несколькими наборами данных геномики и подготовим набор данных, который можно использовать для прогнозирования сайтов связывания в последовательностях ДНК для конкретного типа белка — фактора транскрипции CTCF. Рабочие наборы данных будут получены из общедоступного набора данных ENCODE. Поскольку геном подразумевает одномерную систему координат, многие геномные операции носят пространственный характер. Мы будем использовать ориентированный на задачи геномики API, который предоставляет проект ADAM, для выполнения распределенных пространственных соединений.

Для желающих получить более глубокие знания в области биологии отличным материалом для изучения станет курс EdX Эрика Ландера (**https://www.edx.org/course/introduction-to-biology-the-secret-of-life-3**). Введение в биоинформатику хорошо представлено в книге Артура Леска «Введение в биоинформатику» (Arthur Lesk, *Introduction to Bioinformatics*, Oxford University Press).

Разделение хранения и моделирования

Биоинформатики тратят непропорционально много времени на возню с форматами файлов — .fasta, .fastq, .sam, .bam, .vcf, .gvcf, .bcf, .bed, .gff, .gtf, .narrowPeak, .wig, .bigWig, .bigBed, .ped и .tped, и это лишь часть из них! Некоторые ученые считают необходимым придумать собственный формат для своего пользовательского инструмента. Кроме того, многие спецификации формата являются неполными или неоднозначными (что затрудняет обеспечение согласованности или взаимосоответствия реализаций) и вдобавок хранят данные в кодировке ASCII. Данные ASCII очень распространены в биоинформатике, но они неэффективны и относительно плохо сжимаются. Кроме того, данные всегда приходится подвергать парсингу, что создает дополнительную вычислительную нагрузку.

Этот момент вызывает особое беспокойство, потому что все эти форматы файлов по существу хранят только несколько общих типов объектов: *сопоставленную прочитанную последовательность*, *вызываемый генотип*, *признак последовательности* (sequence feature) и *фенотип*. (Термин «признак последовательности» в геномике имеет несколько расширенное толкование[3], но в этой главе мы используем его в смысле элемента из инструментов геномного анализа трека в окне UCSC Genome Browser.) Такие библиотеки, как `biopython`, популярны среди ученых, потому что они битком набиты парсерами (например, `Bio.SeqIO`), которые пытаются читать все форматы файлов и сводить их в небольшое количество моделей в памяти (например, `Bio.Seq`, `Bio.SeqRecord`, `Bio.SeqFeature`).

Мы можем решить все эти проблемы одним махом, используя фреймворк сериализации, такой, как Apache Avro. Все дело в том, что Avro отделяет модель данных (то есть явную схему) от базового формата файла хранилища, а также представления в

[3] *Признаки последовательности* — это группы аминокислот, которые придают определенные характеристики белку и могут быть важны для его общей функции. — *Примеч. пер.*

памяти, присущего языку. Avro указывает, как данные определенного типа должны передаваться между процессами, будь то передача через Интернет между запущенными процессами или процессом, пытающимся записать данные в файл определенного формата и файловой системой. Например, программа на Java, использующая Avro, может записывать данные в несколько базовых форматов файлов, совместимых с моделью данных Avro. Это позволяет каждому процессу не беспокоиться о совместимости с несколькими форматами файлов: процессу нужно только знать, как читать Avro, а файловой системе нужно знать, как предоставлять Avro данные.

Возьмем в качестве примера признак последовательности. Начнем с указания желаемой схемы для объекта с помощью языка определения интерфейса Avro (interface definition language, IDL):

```
enum Strand {
  Forward,
  Reverse,
  Independent
}

record SequenceFeature {
  string featureId;
  string featureType; ❶
  string chromosome;
  long startCoord;
  long endCoord;
  Strand strand;
  double value;
  map<string> attributes;
}
```

❶ Например, «conservation» (сохранение), «centipede» (многоножка), «gene» (ген).

Этот тип данных можно использовать для кодирования, например, уровня сохранения, наличия промотора или сайта соединения рибосомы, сайта связывания TF и т. д. в определенном месте генома. Этот формат можно рассматривать как двоичную версию JSON, но более ограниченную и с более высоким быстродействием. Спецификация Avro определяет точную двоичную кодировку для объекта с учетом схемы данных, чтобы его можно было легко передавать между процессами (даже если они написаны на разных языках программирования), по сети или на диск для хранения. Проект Avro содержит модули для обработки данных в кодировке Avro на многих языках, включая Java, C/C++, Python и Perl; далее код на любом из этих языков может хранить объект в памяти любым удобным для него способом. Отделение моделирования данных от формата хранения обеспечивает еще один уровень гибкости/абстракции. Данные Avro можно хранить в виде сериализованных двоичных объектов Avro (файл-контейнер Avro), в формате файла столбцов для быстрых запросов (файл Parquet) или в виде текстовых данных JSON для максимальной гибкости (минимальное быстродействие). Наконец, Avro поддерживает развитие схе-

мы, позволяя пользователю добавлять новые поля по мере необходимости, в то время как программное обеспечение одинаково хорошо работает с новыми и старыми версиями схемы.

В целом, Avro — это эффективный механизм двоичного кодирования, позволяющий определять изменяемые схемы данных, обрабатывать одни и те же данные из многих языков программирования и хранить данные в различных форматах. Использование схем Avro освобождает вас от необходимости заботиться о совместимости со все большим количеством пользовательских форматов данных, одновременно повышая производительность ваших вычислений.

Платформы сериализации/RPC

В экосистеме работы с данными существует большое количество фреймворков для сериализации и удаленного вызова процедур. Наиболее часто используемые фреймворки в сообществе больших данных — это Apache Avro и Protocol Buffers от Google (protobuf). По сути, все они предоставляют язык определения интерфейса для указания схем типов объектов/сообщений, и все они компилируются в различные языки программирования. (Среда Google RPC для protobuf теперь является проектом с открытым исходным кодом gRPC.) Помимо IDL и RPC, Avro добавляет спецификацию формата файла для хранения данных на диске.

Трудно делать выводы о том, какая платформа подходит для каких обстоятельств, потому что все они поддерживают разные языки и демонстрируют разную производительность с этими языками.

Конкретная модель `SequenceFeature`, используемая в предыдущем примере, немного упрощена относительно реальных данных, а проект Big Data Genomics (BDG) уже определил схемы Avro для представления нижеперечисленных объектов, а также многих других:

- `AlignmentRecord` для чтения;
- `Variant` для известных вариантов генома и метаданных;
- `Genotype` для названного генотипа в конкретном локусе;
- `Feature` для признака последовательности (аннотация сегмента генома).

Актуальные схемы можно найти в репозитории `bdg-formats` на GitHub (**https://github.com/bigdatagenomics/bdg-formats**). Форматы BDG могут служить заменой вездесущим «устаревшим» форматам (таким, как BAM и VCF), но чаще функционируют как высокопроизводительные «промежуточные» форматы. (Первоначальная цель этих форматов BDG состояла в том, чтобы полностью вытеснить BAM и VCF, но их повсеместное распространение и упрямство ученых показали, что эта цель труднодостижима.) Avro обеспечивает различные преимущества в производительности и моделировании данных по сравнению с традиционными пользовательскими ASCII-форматами.

В оставшейся части главы мы воспользуемся некоторыми схемами BDG для выполнения типичных задач геномики. Прежде чем мы сможем это сделать, нам нужно установить проект ADAM, как показано в следующем разделе.

Установка ADAM

Основной набор инструментов BDG для геномики называется ADAM. Его ядро включает в себя инструменты, которые могут выполнять дублирование меток, перекалибровку базовой оценки качества, повторное выравнивание делеций[4], определение вариантов, а также другие задачи. ADAM также содержит интерфейс командной строки, служащий оберткой ядра для простоты использования. В отличие от традиционных инструментов HPC, ADAM может автоматически выполнять распараллеливание в кластере без необходимости разбивать файлы или планировать задания вручную.

Начнем с установки ADAM с помощью pip:

```
pip3 install bdgenomics.adam
```

Альтернативные способы установки можно найти на странице GitHub (**https://github.com/bigdatagenomics/adam**).

ADAM также поставляется со сценарием отправки, который облегчает взаимодействие со скриптом `spark-submit`:

```
adam-submit
...

Using ADAM_MAIN=org.bdgenomics.adam.cli.ADAMMain
Using spark-submit=/home/analytical-monk/miniconda3/envs/pyspark/bin/spark-submit
       e        888~-_         e            e    e
      d8b       888   \       d8b          d8b  d8b
     /Y88b      888    |     /Y88b        d888bdY88b
    /  Y88b     888    |    /  Y88b      / Y88Y Y888b
   /____Y88b    888   /    /____Y88b    /   YY   Y888b
  /      Y88b   888_-~    /      Y88b  /          Y888b

Usage: adam-submit [<spark-args> --] <adam-args>

Choose one of the following commands:

ADAM ACTIONS
          countKmers : Counts the k-mers/q-mers from a read dataset...
     countSliceKmers : Counts the k-mers/q-mers from a slice dataset...
 transformAlignments : Convert SAM/BAM to ADAM format and optionally...
   transformFeatures : Convert a file with sequence features into...
  transformGenotypes : Convert a file with genotypes into correspondi...
  transformSequences : Convert a FASTA file as sequences into corresp...
```

[4] Хромосомная перестройка, при которой происходит потеря участка хромосомы. — *Примеч. пер.*

```
     transformSlices : Convert a FASTA file as slices into correspond...
   transformVariants : Convert a file with variants into correspondin...
         mergeShards : Merges the shards of a fil...
            coverage : Calculate the coverage from a given ADAM fil...
CONVERSION OPERATION
          adam2fastq : Convert BAM to FASTQ file
  transformFragments : Convert alignments into fragment records
PRIN
               print : Print an ADAM formatted fil
            flagstat : Print statistics on reads in an ADAM file...
                view : View certain reads from an alignment-record file.
```

На этом этапе вы сможете запустить ADAM из командной строки и получить сообщение с кратким руководством об использовании. Как указано в этом сообщении, аргументы Spark задаются перед аргументами, предназначенными для ADAM.

После установки ADAM мы можем приступить к работе с геномными данными. Далее мы изучим API ADAM, работая с образцом набора данных.

Введение в работу с геномными данными с использованием ADAM

Мы начнем с того, что возьмем файл .bam, содержащий некоторые сопоставленные *чтения*[5] NGS, преобразуем их в соответствующий формат BDG (в данном случае `AlignedRecord`) и сохраним их в HDFS. Сначала мы получаем файл .bam:

```
# Примечание: этот файл имеет размер 16 ГБ.
curl -O ftp://ftp.ncbi.nlm.nih.gov/1000genomes/ftp/phase3/data\
/HG00103/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam

# или вместо этого используйте Aspera (что *намного* быстрее)
ascp -i path/to/asperaweb_id_dsa.openssh -QTr -l 10G \
anonftp@ftp.ncbi.nlm.nih.gov:/1000genomes/ftp/phase3/data\
/HG00103/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam
```

Переместите загруженный файл в каталог, где мы будем хранить все данные для этой главы:

```
mv HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam data/genomics
```

Далее мы будем использовать интерфейс командной строки ADAM CLI.

[5] Чтения (риды, reads) — это фрагменты ДНК (длиной от 25 до 20 000 нуклеотидов), считываемые с генома при помощи специальной машины-секвенатора. — *Примеч. пер.*

Преобразование формата файла с помощью ADAM CLI

Затем мы можем использовать команду `transform` ADAM для преобразования файла `.bam` в формат Parquet (его описание приведено в *разд. «Формат Parquet и столбчатое хранилище»*). Эта команда будет работать как в кластере, так и в локальном режиме:

```
adam-submit \
  --master yarn \ ❶
  --deploy-mode client \
  --driver-memory 8G \
  --num-executors 6 \
  --executor-cores 4 \
  --executor-memory 12G \
  -- \
  transform \ ❷
  data/genomics/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam \
  data/genomics/HG00103
```

❶ Пример аргументов Spark для запуска в YARN.

❷ Сама подкоманда ADAM.

Запуск этой команды должен привести к довольно большому выводу на консоль, включая URL-адрес для отслеживания хода выполнения задания.

Результирующий набор данных представляет собой объединение всех файлов в каталоге data/genomics/reads/HG00103/, где каждый файл part-*.parquet является результатом выполнения одной из задач PySpark. Вы также заметите, что данные были сжаты более эффективно, чем исходный файл .bam (который сжат архиватором Gzip) благодаря столбцовому хранилищу Parquet.

```
$ du -sh data/genomics/HG00103*bam
16G data/genomics/HG00103. [...] .bam

$ du -sh data/genomics/HG00103/
13G data/genomics/HG00103
```

Давайте посмотрим, как выглядит один из этих объектов в интерактивном сеансе.

Получение геномных данных с помощью PySpark и ADAM

Запустите оболочку PySpark с помощью вспомогательной команды ADAM. Она загружает все необходимые JAR-файлы.

```
pyadam

...

[...]
Welcome to
```

```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.2.1
      /_/

Using Python version 3.6.12 (default, Sep 8 2020 23:10:56)
Spark context Web UI available at http://192.168.29.60:4040
Spark context available as 'sc'.
SparkSession available as 'spark'.

>>>
```

В некоторых случаях вы можете столкнуться с ошибкой `TypeError` с упоминанием об отсутствии объекта JavaPackage при попытке использовать ADAM с PySpark. Это известная проблема, описанная на GitHub (**https://github.com/bigdatagenomics/adam/issues/2122**).

В таком случае попробуйте решения, предложенные на GitHub. Можно выполнить следующую команду, чтобы запустить оболочку PySpark с ADAM:

```
!pyspark --conf spark.serializer=org.apache.spark.
serializer.KryoSerializer --conf spark.kryo.registrator=
org.bdgenomics.adam.serialization.ADAMKryoRegistrator
--jars `find-adam-assembly.sh` --driver-class-path
`find-adam-assembly.sh`
```

Теперь загрузите совмещенные данные чтения как `AlignmentDataset`:

```
from bdgenomics.adam.adamContext import ADAMContext

ac = ADAMContext(spark)

readsData = ac.loadAlignments("data/HG00103")

readsDataDF = readsData.toDF()
readsDataDF.show(1, vertical=True)

...

-RECORD 0-----------------------------------------
 referenceName | hs37d5
 start | 21810734
 originalStart | null
 end | 21810826
 mappingQuality | 0
 readName | SRR062640.14600566
 sequence | TCCATTCCACTCAGTTT...
 qualityScores | /MOONNCRQPIQIKRGL...
```

```
 cigar | 92M8S
 originalCigar | null
 basesTrimmedFromStart | 0
 basesTrimmedFromEnd | 0
 readPaired | true
 properPair | false
 readMapped | false
 mateMapped | true
 failedVendorQualityChecks | false
 duplicateRead | false
 readNegativeStrand | false
 mateNegativeStrand | false
 primaryAlignment | true
 secondaryAlignment | false
 supplementaryAlignment | false
 mismatchingPositions | null
 originalQualityScores | null
 readGroupId | SRR062640
 readGroupSampleId | HG00103
 mateAlignmentStart | 21810734
 mateReferenceName | hs37d5
 insertSize | null
 readInFragment | 1
 attributes | RG:Z:SRR062640\tX...
only showing top 1 row
(показана только первая строка)
```

Вы можете получить другой вывод, потому что разделение данных в вашей системе может быть другим, и неизвестно, какие из них будут выведены первыми.

Теперь мы можем в интерактивном режиме задавать вопросы о нашем наборе данных, одновременно выполняя вычисления в кластере в фоновом режиме. Сколько чтений у нас есть в этом наборе данных?

```
readsData.toDF().count()
...
160397565
```

Поступают ли чтения в этом наборе данных из всех хромосом человека?

```
unique_chr = readsDataDF.select('referenceName').distinct().collect()
unique_chr = [u.referenceName for u in unique_chr]

unique_chr.sort()
...
1
10
11
12
```

```
GL000249.1
MT
NC_007605
X
Y
hs37d5
```

Да, мы наблюдаем считывание хромосом с 1 по 22, X и Y, а также некоторых других фрагментов хромосом, которые не являются частью «основных» хромосом или расположение которых неизвестно. Давайте проанализируем код более внимательно:

```
readsData = ac.loadAlignments("data/HG00103") ❶

readsDataDF = readsData.toDF() ❷

unique_chr = readsDataDF.select('referenceName').distinct(). \ ❸
             collect() ❹
```

❶ `AlignmentDataset`: тип ADAM, содержащий все наши данные.

❷ `DataFrame`: базовый объект Spark DataFrame.

❸ Объединит все отдельные имена контигов[6].

❹ Запускает вычисление и возвращает данные в DataFrame в клиентское приложение (оболочку).

В качестве более наглядного примера предположим, что мы тестируем геном человека, чтобы проверить, является ли он носителем каких-либо отклонений, которые создают риск рождения ребенка с муковисцидозом (MB). В нашем генетическом тесте используется секвенирование ДНК нового поколения для получения чтений нескольких определенных генов, таких, как ген CFTR (мутации которого могут вызывать муковисцидоз). Пропустив эти данные через наш конвейер генотипирования, мы определили, что ген CFTR, по-видимому, имеет преждевременный стоп-кодон, который нарушает его функционирование. Однако об этой мутации никогда ранее не сообщалось ни в базе данных мутаций генов человека, ни в базе данных Sickkids CFTR, которая объединяет мутации генов, ведущие к заболеванию MB. Мы хотим вернуться к необработанным данным секвенирования, чтобы увидеть, не является ли потенциально опасный фрагмент генотипа ложным срабатыванием. Для этого нам нужно вручную проанализировать все чтения, которые соответствуют этому вариантному локусу, скажем, хромосоме 7 по адресу 117149189 (рис. 9.1):

```
from pyspark.sql import functions as fun
cftr_reads = readsDataDF.where("referenceName == 7").\
```

[6] *Контиг* (contig) — группа из нескольких последовательно соединенных секвенированных участков ДНК, т. е. секвенированный без пропусков фрагмент ДНК. — *Примеч. пер.*

```
            where(fun.col("start") <= 117149189).\
            where(fun.col("end") > 117149189)
cftr_reads.count()
...

9
```

Теперь можно вручную проверить эти девять чтений или обработать их, например, с помощью пользовательского согласователя и проверить, является ли заявленный патогенный вариант ложноположительным.

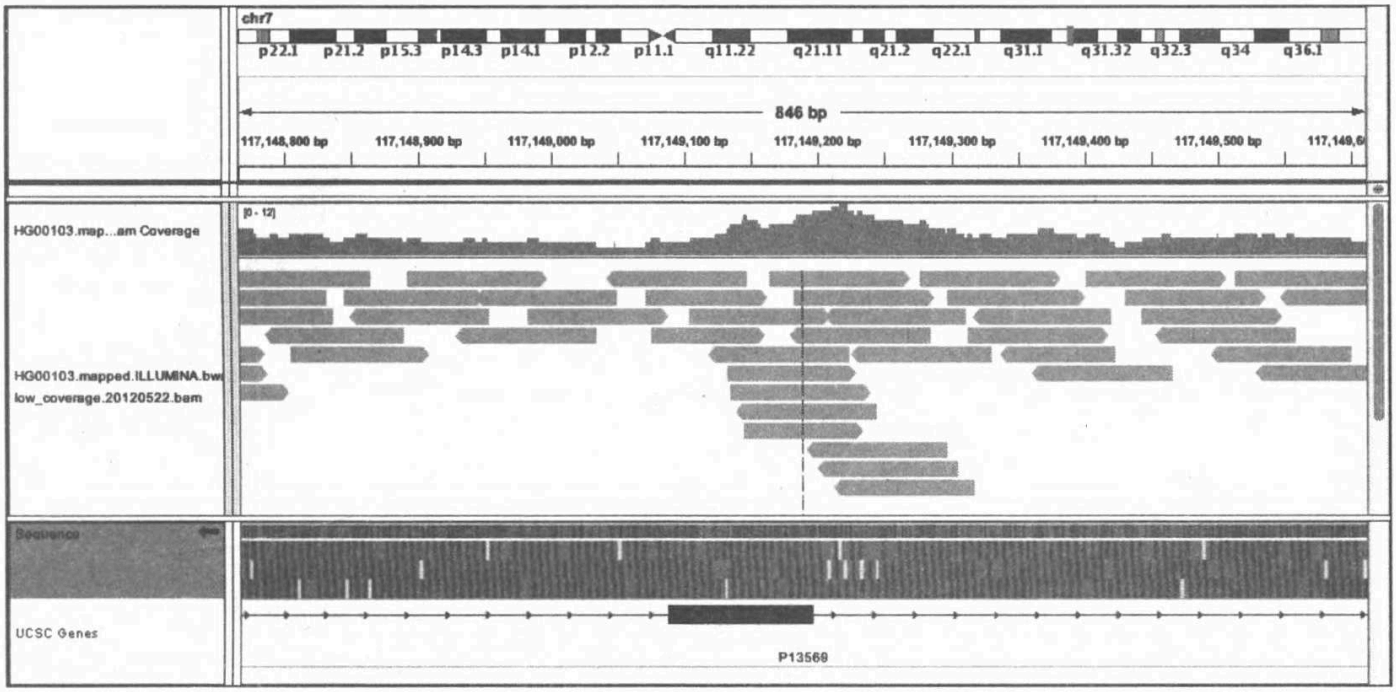


Рис. 9.1. Визуализация HG00103 в chr7:117149189 в гене CFTR с помощью Integrative Genomic Viewer

Скажем, у нас есть клиническая лаборатория, которая проводит такой скрининг фрагментов в качестве услуги для лечащих врачей. Архивирование необработанных данных с использованием облачной системы хранения, такой, как AWS S3, гарантирует, что данные остаются относительно «теплыми» (по сравнению, скажем, с ленточным архивом). Помимо наличия надежной системы для фактического выполнения обработки данных, мы можем легко получить доступ ко всем прошлым данным для контроля качества или в случаях, когда требуется ручное вмешательство, как в примере CFTR, представленном ранее. В дополнение к быстрому доступу ко всей совокупности данных централизованность также упрощает проведение крупных аналитических исследований, таких, как популяционная генетика, крупномасштабные анализы контроля качества и т. д.

Теперь, когда мы знакомы с API ADAM, давайте начнем работу над созданием нашего набора данных для прогнозирования фактора транскрипции.

Формат Parquet и столбчатое хранилище

В предыдущем разделе вы увидели, что можно работать с потенциально очень большим объемом данных секвенирования, не беспокоясь о специфике базового хранилища или распараллеливания выполнения. Однако стоит отметить, что в проекте ADAM используется формат файла Parquet, обеспечивающий значительные преимущества в производительности, о которых мы хотели бы рассказать.

Parquet — это открытая спецификация формата файла и набор реализаций чтения/записи, которые мы рекомендуем для постоянного использования при работе с данными, которые будут использоваться в аналитических запросах (запись один раз, чтение много раз). Он в значительной степени основан на базовом формате хранения данных, используемом в системе Google Dremel (см. Melnik et al., «Dremel: Interactive Analysis of Web-scale Datasets», Proc. VLDB, 2010)[7], и имеет модель данных, совместимую с Avro, Thrift и Protocol Buffers. В частности, он поддерживает большинство распространенных типов баз данных (int, double, string и т. д.), а также массивы и записи, включая вложенные типы. Примечательно, что это столбчатый формат файла, означающий, что значения для определенного столбца из многих записей хранятся на диске непрерывно (рис. 9.2).

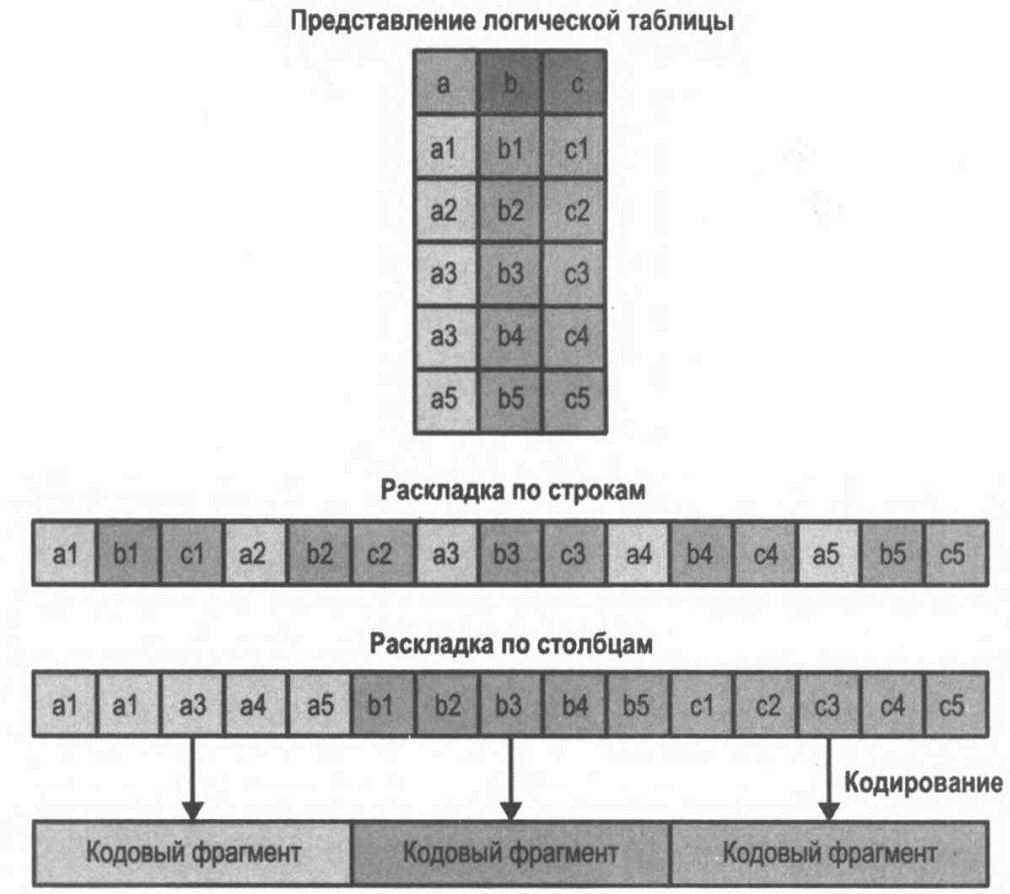


Рис. 9.2. Различия между расположением данных по строкам и по столбцам

[7] URL: **https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36632.pdf**.

Такая физическая структура данных обеспечивает гораздо более эффективное кодирование/сжатие данных и значительно сокращает время выполнения запросов за счет сведения к минимуму объема данных, которые необходимо прочитать/десериализовать. Parquet поддерживает применение различных схем сжатия для каждого столбца, а также схемы кодирования для конкретных столбцов, такие как кодирование длин серий, кодирование по словарю и дельта-кодирование.

Еще одна полезная функция Parquet для повышения быстродействия — *проталкивание предикатов* (predicate pushdown). *Предикат* — это некоторое выражение или функция, которая оценивается как истина или ложь на основе записи данных (что эквивалентно выражению в операторе WHERE в языке SQL) (аналогично оператору `WHERE` в SQL). В нашем предыдущем запросе CFTR Spark должен был десериализовать/материализовать каждую запись `AlignmentRecord`, прежде чем решить, прошла ли она предикат. Это приводит к значительным потерям вычислительных ресурсов и времени ввода-вывода. Реализации считывателей Parquet позволяют нам предоставить класс предикатов, который десериализует только столбцы, необходимые для принятия решения, прежде чем материализовать полную запись.

Прогнозирование сайтов связывания транскрипционных факторов на основе данных ENCODE

В этом примере мы будем использовать общедоступные данные о признаках последовательности, чтобы построить простую модель связывания транскрипционных факторов (TF). TF представляют собой белки, которые связываются со специфическими последовательностями ДНК в геноме и помогают контролировать экспрессию различных генов. Поэтому они имеют решающее значение в определении фенотипа конкретной клетки и участвуют во многих физиологических и болезненных процессах. *ChIP-seq* представляет собой анализ на основе NGS, который позволяет охарактеризовать сайты связывания определенного TF в масштабах всего генома для конкретного типа клеток/тканей. Однако в дополнение к стоимости и технической сложности такой анализ требует отдельного эксперимента для каждой пары ткань/TF. Напротив, *DNase-seq* представляет собой анализ, который находит области открытого хроматина по всему геному и должен выполняться только один раз для каждого типа ткани. Вместо того чтобы анализировать сайты связывания TF путем проведения эксперимента ChIP-seq для каждой комбинации ткань/TF, мы хотели бы предсказывать сайты связывания TF в новом типе ткани, предполагая только наличие данных DNase-seq.

В частности, мы предскажем сайты связывания для CTCF TF, используя данные DNase-seq вместе с известными данными о *мотивах последовательности* (sequence

motif) из HT-SELEX и другими данными из общедоступного набора данных ENCODE (**https://www.encodeproject.org/**). Мы выбрали шесть различных типов клеток, которые имеют доступные для обучения данные DNase-seq и CTCF ChIP-seq. Обучающим примером будет пик гиперчувствительности к ДНКазе (сегмент генома), а бинарная метка того, связан или несвязан TF, будет получена из данных ChIP-seq.

Обобщим поток данных: основные примеры обучения/тестирования будут получены из данных DNase-seq. Каждая область открытого хроматина (интервал в геноме) будет использоваться для предсказания того, будет ли там связываться конкретный TF в конкретном типе ткани. Для этого мы пространственно присоединяем данные ChIP-seq к данным DNase-seq; каждое перекрытие является положительной меткой для объектов DNase-seq. Наконец, для повышения точности предсказания мы генерируем дополнительный признак на каждом интервале в данных DNase-seq — расстояние до сайта начала транскрипции (используя набор данных GENCODE). Признак добавляется в обучающие примеры путем выполнения пространственного соединения (с возможной агрегацией).

Мы будем использовать данные из следующих клеточных линий:

GM12878

Обычно изучаемая линия лимфобластоидных клеток.

K562

Женский хронический миелогенный лейкоз.

BJ

Фибробласт кожи.

HEK293

Первичная почка (вольфово тело).

H54

Глиобластома.

HepG2

Гепатоцеллюлярная карцинома.

Сначала мы загружаем данные ДНКазы для каждой клеточной линии в формате .narrowPeak:

```
mkdir data/genomics/dnase

curl -O -L "https://www.encodeproject.org/ \
              files/ENCFF001UVC/@@download/ENCFF001UVC.bed.gz" | \
              gunzip > data/genomics/dnase/GM12878.DNase.narrowPeak ❶
curl -O -L "https://www.encodeproject.org/ \
              files/ENCFF001UWQ/@@download/ENCFF001UWQ.bed.gz" | \
              gunzip > data/genomics/dnase/K562.DNase.narrowPeak
```

```
curl -O -L "https://www.encodeproject.org/ \
             files/ENCFF001WEI/@@download/ENCFF001WEI.bed.gz" | \
             gunzip > data/genomics/dnase/BJ.DNase.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
             files/ENCFF001UVQ/@@download/ENCFF001UVQ.bed.gz" | \
             gunzip > data/genomics/dnase/HEK293.DNase.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
             files/ENCFF001SOM/@@download/ENCFF001SOM.bed.gz" | \
             gunzip > data/genomics/dnase/H54.DNase.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
             files/ENCFF001UVU/@@download/ENCFF001UVU.bed.gz" | \
             gunzip > data/genomics/dnase/HepG2.DNase.narrowPeak
[...]
```

❶ Потоковая распаковка.

Затем мы загружаем данные ChIP-seq для CTCF TF, также в формате .narrowPeak, и данные GENCODE в формате GTF:

```
mkdir data/genomics/chip-seq

curl -O -L "https://www.encodeproject.org/ \
         files/ENCFF001VED/@@download/ENCFF001VED.bed.gz" | \
         gunzip > data/genomics/chip-seq/GM12878.ChIP-seq.CTCF.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
         files/ENCFF001VMZ/@@download/ENCFF001VMZ.bed.gz" | \
         gunzip > data/genomics/chip-seq/K562.ChIP-seq.CTCF.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
         files/ENCFF001XMU/@@download/ENCFF001XMU.bed.gz" | \
         gunzip > data/genomics/chip-seq/BJ.ChIP-seq.CTCF.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
         files/ENCFF001XQU/@@download/ENCFF001XQU.bed.gz" | \
         gunzip > data/genomics/chip-seq/HEK293.ChIP-seq.CTCF.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
         files/ENCFF001USC/@@download/ENCFF001USC.bed.gz" | \
         gunzip> data/genomics/chip-seq/H54.ChIP-seq.CTCF.narrowPeak
curl -O -L "https://www.encodeproject.org/ \
         files/ENCFF001XRC/@@download/ENCFF001XRC.bed.gz" | \
         gunzip> data/genomics/chip-seq/HepG2.ChIP-seq.CTCF.narrowPeak
curl -s -L "http://ftp.ebi.ac.uk/pub/databases/gencode/\
         Gencode_human/release_18/gencode.v18.annotation.gtf.gz" | \
         gunzip > data/genomics/gencode.v18.annotation.gtf
[...]
```

Обратите внимание, как мы распаковываем поток данных с помощью `gunzip` на пути к его размещению в нашей файловой системе.

Из всех этих необработанных данных мы хотим создать обучающий набор со схемой, подобной следующей:

1. Хромосома.
2. Начало.
3. Конец.
4. Расстояние до ближайшего сайта начала транскрипции (TSS).
5. Идентификатор TF (в данном случае всегда «CTCF»).
6. Клеточная линия.
7. Статус связывания TF (логическое значение; целевая переменная).

Этот набор данных можно легко преобразовать в фрейм данных типа DataFrame для переноса в библиотеку машинного обучения. Поскольку нам нужно сгенерировать данные для нескольких строк ячеек, мы определим DataFrame для каждой строки ячеек отдельно и соединим их в конце:

```
cell_lines = ["GM12878", "K562", "BJ", "HEK293", "H54", "HepG2"]
for cell in cell_lines:
## Для каждой клеточной линии...
## ...генерируем подходящий фрейм данных
## Объединяем фреймы данных и передаем, например, в MLlib
```

Определяем служебную функцию и широковещательную переменную, которые будем использовать для генерации признаков:

```
local_prefix = "data/genomics"
import pyspark.sql.functions as fun

## UDF для поиска ближайшего сайта начала транскрипции
## Это примитивное решение;
## упражнение для читателей: заставьте код работать быстрее
def distance_to_closest(loci, query):
  return min([abs(x - query) for x in loci])
distance_to_closest_udf = fun.udf(distance_to_closest)

## Создаем структуру в памяти для вычисления расстояния до TSS
## по сути, мы реализуем здесь всенаправленное объединение
tss_data = ac.loadFeatures("data/genomics/gencode.v18.annotation.gtf")
tss_df = tss_data.toDF().filter(fun.col("featureType") == 'transcript')
b_tss_df = spark.sparkContext.broadcast(tss_df.groupBy('referenceName').\
                agg(fun.collect_list("start").alias("start_sites")))
```

Теперь, когда загружены данные, необходимые для определения наших обучающих примеров, мы определяем тело «цикла» для вычисления данных по каждой клеточной линии. Обратите внимание, как мы читаем текстовые представления данных ChIP-seq и DNase, поскольку наборы данных не настолько велики, чтобы снижать быстродействие.

Для этого мы загружаем данные ДНКазы и ChIP-seq:

```
current_cell_line = cell_lines[0]

dnase_path = f'data/genomics/dnase/{current_cell_line}.DNase.narrowPeak'
dnase_data = ac.loadFeatures(dnase_path) ❶
dnase_data.toDF().columns ❷
...
['featureId', 'sampleId', 'name', 'source', 'featureType', 'referenceName',
'start', 'end', 'strand', 'phase', 'frame', 'score', 'geneId', 'transcriptId',
'exonId', 'proteinId', 'aliases', 'parentIds', 'target', 'gap', 'derivesFrom',
'notes', 'dbxrefs', 'ontologyTerms', 'circular', 'attributes']

...

chip_seq_path = f'data/genomics/chip-seq/ \
                 {current_cell_line}.ChIP-seq.CTCF.narrowPeak'
chipseq_data = ac.loadFeatures(chipseq_path)❶
```

❶ `FeatureDataset`.

❷ Столбцы в фрейме данных `Dnase`.

Сайты, которые перекрывают пик ChIP-seq, как определено `ReferenceRegion` в `chipseq_data`, имеют сайты связывания TF и поэтому помечаются как `true` (истинные), в то время как остальные сайты помечаются как `false` (ложные). Мы делаем это с помощью примитивов пространственного объединения 1D, предоставляемых API ADAM. Для функции объединения требуется RDD с ключом `ReferenceRegion`, который будет создавать кортежи с перекрывающимися областями в соответствии с обычной семантикой присоединения (например, внутреннее или внешнее).

```
dnase_with_label = dnase_data.leftOuterShuffleRegionJoin(chipseq_data)
dnase_with_label_df = dnase_with_label.toDF()
...

-RECORD 0-----------------------------------------------------------------..
 _1  | {null, null, chr1.1, null, null, chr1, 713841, 714424, INDEPENDENT, null..
 _2  | {null, null, null, null, null, chr1, 713945, 714492, INDEPENDENT, null, ..
-RECORD 1-----------------------------------------------------------------..
 _1  | {null, null, chr1.2, null, null, chr1, 740179, 740374, INDEPENDENT, null..
 _2  | {null, null, null, null, null, chr1, 740127, 740310, INDEPENDENT, null, ..
-RECORD 2-----------------------------------------------------------------..
 _1  | {null, null, chr1.3, null, null, chr1, 762054, 763213, INDEPENDENT, null..
 _2  | null...
only showing top 3 rows
(показаны первые 3 строки)
...

dnase_with_label_df = dnase_with_label_df.\
                      withColumn("label", \
                                 ~fun.col("_2").isNull())
dnase_with_label_df.show(5)
```

Теперь мы вычисляем окончательный набор признаков для каждого пика ДНКазы:

```
## формируем окончательный обучающий фрейм
training_df = dnase_with_label_df.withColumn(
    "contig", fun.col("_1").referenceName).withColumn(
    "start", fun.col("_1").start).withColumn(
    "end", fun.col("_1").end).withColumn(
    "tf", fun.lit("CTCF")).withColumn(
    "cell_line", fun.lit(current_cell_line)).drop("_1", "_2")

training_df = training_df.join(b_tss_df,
                               training_df.contig == b_tss_df.referenceName,
                               "inner") ❶

training_df.withColumn("closest_tss",
                       fun.least(distance_to_closest_udf(fun.col("start_sites"),
                                                         fun.col("start")),
                                 distance_to_closest_udf(fun.col("start_sites"),
                                                         fun.col("end")))) ❷
```

❶ Левое соединение с созданным ранее `tss_df`.

❷ Получение ближайшего расстояния TSS.

Этот окончательный фрейм данных вычисляется при каждом проходе цикла по линиям ячеек. Наконец, мы объединяем каждый фрейм из каждой клеточной линии и кэшируем эти данные в памяти для подготовки к обучению моделей:

```
preTrainingData = data_by_cellLine.union(...)
preTrainingData.cache()

preTrainingData.count()
preTrainingData.filter(fun.col("label") == true).count()
```

На этом этапе данные в `preTrainingData` могут быть нормализованы и преобразованы в DataFrame для обучения классификатора, как описано в *разд. «Случайные леса» главы 4*. Обратите внимание, что вы должны выполнить перекрестную проверку, где в каждом проходе удерживаются данные из одной из клеточных линий.

Что дальше?

Многие вычисления в геномике прекрасно вписываются в вычислительную парадигму PySpark. Когда вы выполняете специальный анализ, наиболее ценным достоинством таких проектов, как ADAM, является набор схем Avro, которые представляют базовые аналитические объекты (наряду с инструментами преобразования). Мы видели, как после преобразования данных в соответствующие схемы Avro многие крупномасштабные вычисления становятся относительно легко реализуемыми и распространяемыми.

Хотя инструментов для проведения научных исследований в PySpark все еще недостаточно много, существует несколько проектов, которые могли бы помочь избежать изобретения велосипеда. В этой книге мы изучили основные функциональные возможности, реализованные в ADAM, но в проекте уже есть реализации для всех передовых методов GATK, включая повторное сопоставление делеций и дедупликацию. В дополнение к ADAM Институт Брода в настоящее время разрабатывает крупные программные проекты с использованием Spark, включая новейшую версию GATK4 и проект под названием Hail (**https://github.com/hail-is/hail**) для крупномасштабных вычислений в области популяционной генетики. Все эти инструменты имеют открытый исходный код, поэтому, если вы начнете использовать их в своей работе, подумайте о том, чтобы внести свои улучшения!

ГЛАВА 10

Обнаружение сходства изображений с помощью глубокого обучения и PySpark LSH

Изображения в социальных сетях или интернет-магазинах давно являются неотъемлемой частью нашей цифровой жизни. Но настоящую революцию в области глубокого обучения вызвал набор изображений ImageNet. Модель классификации изображений, разработанная командой Джеффри Хинтона и победившая на конкурсе ImageNet 2012 благодаря потрясающей точности, стала важной вехой и привлекла внимание всего сообщества машинного обучения. Поэтому вы наверняка, в качестве специалиста по науке о данных, уже сталкивались или непременно столкнетесь с наборами изображений.

В этой главе вы получите опыт масштабирования рабочего процесса глубокого обучения для задачи компьютерного зрения, а именно, обнаружения сходства изображений, с помощью PySpark. Задача идентификации изображений, похожих друг на друга, интуитивно понятна людям, но это сложная вычислительная задача. При масштабировании она становится еще сложнее. В этой главе мы представим приблизительный метод поиска похожих элементов, называемый *хешированием с учетом локальных особенностей* (locality sensitive hashing, LSH)[1], и применим его к изображениям. Мы будем использовать глубокое обучение для преобразования данных изображения в числовое векторное представление. Алгоритм PySpark LSH будет применен к результирующим векторам, что позволит нам найти похожие изображения с учетом нового входного изображения.

Пример обнаружения сходства изображений, рассмотренный в этой главе, отражает один из подходов, используемых приложениями для обмена фотографиями, такими, как Instagram и Pinterest. Это помогает их пользователям ориентироваться в гигантском массиве визуальных данных, накопленных на платформах. Он также показывает, как рабочий процесс глубокого обучения может выиграть от масштабируемости PySpark.

Мы начнем с краткого знакомства с фреймворком глубокого обучения PyTorch. В последние годы он приобрел известность благодаря относительной легкости освоения по сравнению с другими популярными низкоуровневыми библиотеками

[1] Также иногда называют *хешированием с учетом локалей* или, для краткости, *локальным хешированием*. — *Примеч. пер.*

глубокого обучения. Затем мы загрузим и подготовим наш набор данных. Мы будем использовать набор данных Cars, опубликованный в 2013 г. Стэнфордской лабораторией искусственного интеллекта. PyTorch будет работать на этапе предварительной обработки изображений. За этим последует преобразование наших входных данных изображения в векторное представление (*встраивание* изображений).

Далее мы импортируем полученные векторные представления в PySpark и преобразуем их с помощью алгоритма LSH. Мы завершим главу тем, что возьмем новое изображение и выполним поиск ближайших соседей, используя наш преобразованный LSH набор данных, чтобы найти похожие изображения.

Давайте начнем с установки PyTorch.

PyTorch

PyTorch — это фреймворк для создания проектов глубокого обучения. Он обладает высокой гибкостью и позволяет выражать модели глубокого обучения характерными средствами языка Python. Первое применение этот фреймворк нашел в исследовательском сообществе. В последнее время, благодаря простоте использования, он превратился в один из самых известных инструментов глубокого обучения для широкого круга приложений. Наряду с TensorFlow на данный момент это самая популярная библиотека для глубокого обучения.

Простой и гибкий интерфейс PyTorch позволяет выполнять быстрые эксперименты. Вы можете загружать данные, применять преобразования и создавать модели с помощью нескольких строк кода. У вас есть возможность создавать индивидуальные циклы обучения, проверки и тестирования и с легкостью развертывать обученные модели. PyTorch постоянно используют в профессиональных задачах для реальной, критически важной работы. Возможность использовать GPU (графические процессоры) для обучения ресурсоемких моделей стала важным фактором популярности глубокого обучения. Хотя для нашей задачи это не и понадобится, отметим, что PyTorch обеспечивает отличную поддержку GPU.

Установка PyTorch

На веб-сайте PyTorch вы можете легко получить инструкции по установке в зависимости от конфигурации вашей системы, как показано на рис. 10.1.

Выполните следующую команду и следуйте инструкциям для настройки вашей конфигурации:

```
$ pip3 install torch torchvision
```

Мы не будем полагаться на GPU и, следовательно, выберем CPU в качестве вычислительной платформы. Если у вас есть графический процессор, который вы хотите использовать, выберите соответствующие параметры, чтобы получить необходимые инструкции. Для этой главы нам также не понадобится Torchaudio, поэтому мы пропустим его установку.

PyTorch Build	Stable (1.10)	Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python	C++ / Java		
Compute Platform	~~CUDA 10.2~~	~~CUDA 11.3~~	~~ROCm 4.2 (beta)~~	CPU
Run this Command:	pip3 install torch torchvision torchaudio			

Рис. 10.1. Установка PyTorch в зависимости от конфигурации системы

Подготовка данных

Мы будем использовать набор данных Stanford Cars. Он был опубликован Джонатаном Краузе, Майклом Старком, Цзя Денгом и Ли Фей-Фей в рамках доклада ICCV 2013 г. «Трехмерные представления объектов для мелкозернистой категоризации».

Вы можете скачать изображения с Kaggle или по ссылке на источник, предоставленной Stanford AI Lab.

```
wget http://ai.stanford.edu/~jkrause/car196/car_ims.tgz
```

После загрузки разархивируйте каталоги обучающих и тестовых изображений и поместите их в каталог с именем cars_data:

```
data_directory = "cars_data"
train_images = "cars_data/cars_train/cars_train"
```

Вы можете скачать файл CSV, содержащий метки для набора обучающих данных, на GitHub (**https://github.com/BotechEngineering/StanfordCarsDatasetCSV**). Загрузите его, переименуйте в cars_train_data.csv и поместите в каталог данных. Давайте посмотрим на содержимое файла:

```
import pandas as pd

train_df = pd.read_csv(data_directory+"/cars_train_data.csv")

train_df.head()
...

    Unnamed: 0 x1        y1          x2        y2    Class     image
0               0        39        116       569   375    14       00001.jpg
1               1        36        116       868   587    3        00002.jpg
2               2        85        109       601   381    91       00003.jpg
3               3        621       393       1484  1096   134      00004.jpg
4               4        14        36        133   99     106      00005.jpg
```

Игнорируйте все столбцы, кроме `Class` и `Image`. Другие столбцы относятся к исходному исследовательскому проекту, из которого был получен этот набор данных, и не понадобятся для нашей задачи.

Изменение размера изображений с помощью PyTorch

Прежде чем мы двинемся дальше, нам нужно обработать наши изображения. Предварительная обработка данных очень часто применяется в машинном обучении, поскольку модели глубокого обучения (нейронные сети) ожидают, что ввод будет соответствовать определенным требованиям.

Нам нужно применить ряд шагов предварительной обработки, называемых *преобразованиями*, чтобы преобразовать входные изображения в правильный формат для моделей. В нашем случае нужно, чтобы это были изображения в формате JPEG размером 224 × 224 пикселя, поскольку таковы требования модели ResNet-18, которую мы будем использовать в следующем разделе. Мы выполняем это преобразование с помощью пакета Torchvision PyTorch в следующем коде:

```
import os
from PIL import Image
from torchvision import transforms

# входной размер, нужный для CNN
input_dim = (224,224)
input_dir_cnn = data_directory + "/images/input_images_cnn"

os.makedirs(input_dir_cnn, exist_ok = True)

transformation_for_cnn_input =
                        transforms.Compose([transforms.Resize(input_dim)])

for image_name in os.listdir(train_images):
    I = Image.open(os.path.join(train_images, image_name))
    newI = transformation_for_cnn_input(I)

    newI.save(os.path.join(input_dir_cnn, image_name))

    newI.close()
    I.close()
```

Здесь мы используем единственное преобразование, которое изменяет размер изображения, чтобы оно соответствовало нейронным сетям. Однако мы можем использовать преобразование `Compose`, чтобы определить серию преобразований, используемых для предварительной обработки нашего изображения.

Теперь наш набор данных готов. В следующем разделе мы преобразуем данные изображения в векторное представление, подходящее для использования с алгоритмом PySpark LSH.

Модель глубокого обучения для векторного представления изображений

Сверточные нейронные сети, или CNN, — это стандартные архитектуры нейронных сетей, используемые для прогнозирования, когда входными наблюдениями являются изображения. Мы будем использовать их не для прогнозирования, а для создания векторного представления изображений. В частности, мы будем использовать архитектуру ResNet-18.

Остаточная сеть Residual Network (Residual Network, ResNet) была представлена в 2015 г. (Shaoqing Ren, Kaiming He, Jian Sun, Xiangyu Zhang, *Deep Residual Learning for Image Recognition*). Число 18 в названии ResNet-18 означает количество слоев, существующих в архитектуре нейронной сети. Другие популярные варианты ResNet включают 34 и 50 слоев. Большее количество слоев приводит к повышению точности за счет увеличения объема вычислений.

Встраивание изображений

Встраивание изображения — это представление изображения в векторном пространстве. Основная идея заключается в том, что если одно изображение похоже на другое, их векторы тоже будут похожи и расположены рядом в пространственном измерении.

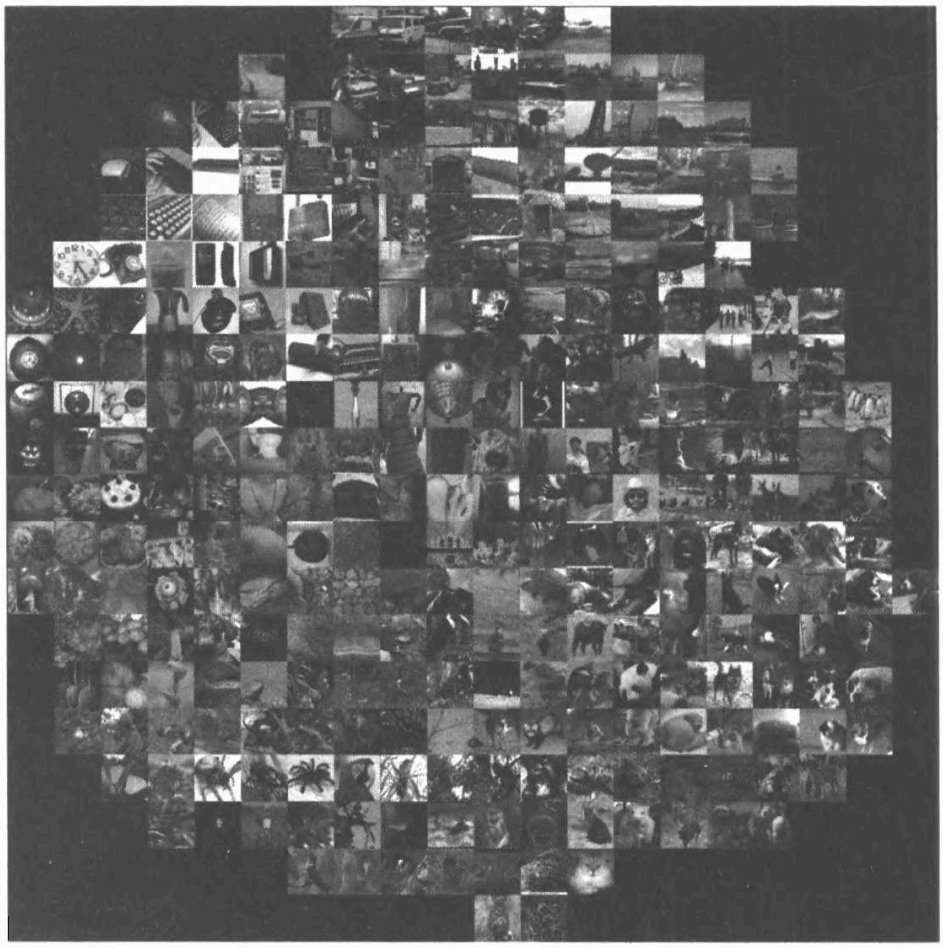


Рис. 10.2. Встраивание изображений ILSVRC 2012 в двумерное пространство

Изображение на рис. 10.2, опубликованное Андреем Карпати, показывает, как изображения могут быть представлены в пространстве более низкого измерения. Например, вы можете заметить, что транспортные средства сосредоточены вверху, а птицы внизу слева.

Мы можем получить встраивания изображений из сети ResNet-18, взяв выходные данные ее предпоследнего, полностью связанного слоя, который имеет размерность 512. Затем мы создаем класс, который при наличии изображения может возвращать его представление в форме числового вектора.

```
import torch
from torchvision import models

class Img2VecResnet18():
    def __init__(self):
        self.device = torch.device("cpu")
        self.numberFeatures = 512
        self.modelName = "resnet-18"
        self.model, self.featureLayer = self.getFeatureLayer()
        self.model = self.model.to(self.device)
        self.model.eval()
        self.toTensor = transforms.ToTensor() ❶
        self.normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])❷

    def getFeatureLayer(self):
        cnnModel = models.resnet18(pretrained=True)
        layer = cnnModel._modules.get('avgpool')
        self.layer_output_size = 512

        return cnnModel, layer

    def getVec(self, img):
        image =
        self.normalize(self.toTensor(img)).unsqueeze(0).to(self.device)
        embedding = torch.zeros(1, self.numberFeatures, 1, 1)
        def copyData(m, i, o): embedding.copy_(o.data)
        h = self.featureLayer.register_forward_hook(copyData)
        self.model(image)
        h.remove()
        return embedding.numpy()[0, :, 0, 0]
```

❶ Преобразование изображений в тензорный формат PyTorch.

❷ Приведение значений пикселей к интервалу от 0 до 1. Среднее значение и стандартное отклонение (std) были предварительно рассчитаны на основе данных, использованных для обучения модели. Нормализация изображения повышает точность классификатора.

Теперь мы инициализируем класс `Img2VecResnet18` и применяем метод `getVec` ко всем изображениям, чтобы получить их векторные представления.

```
import tqdm

img2vec = Img2VecResnet18()
allVectors = {}
for image in tqdm(os.listdir(input_dir_cnn)):
    I = Image.open(os.path.join(input_dir_cnn, image))
    vec = img2vec.getVec(I)
    allVectors[image] = vec
    I.close()
```

Работая с большими наборами данных, вы можете последовательно записать векторный вывод в файл, а не хранить его в памяти, чтобы избежать ошибки нехватки памяти. Здесь можно управлять данными, поэтому мы создаем словарь, который сохраняем в виде CSV-файла на следующем шаге:

```
import pandas as pd

pd.DataFrame(allVectors).transpose().\
    to_csv(data_folder + '/input_data_vectors.csv')
```

Поскольку мы работаем локально, мы выбрали для сохранения векторного вывода формат CSV. Однако формат Parquet больше подходит для данных такого рода. Вы можете легко сохранить данные в формате Parquet, заменив `to_csv` на `to_parquet` в предыдущем коде.

Теперь, когда у нас есть необходимые встраивания изображений, мы можем импортировать их в PySpark.

Импорт встраиваний изображений в PySpark

Запустите оболочку PySpark:

```
$ pyspark --driver-memory 4g
```

Импортируйте встраивания изображений:

```
input_df = spark.read.option('inferSchema', True).\
                    csv(data_directory + '/input_data_vectors.csv')
input_df.columns
...

['_c0',
'_c1',
'_c2',
'_c3',
'_c4',
[...]
'_c509',
```

```
'_c510',
'_c511',
'_c512']
```

Для реализации PySpark LSH требуется столбец векторов в качестве входных данных. Мы можем создать его, объединив соответствующие столбцы в нашем фрейме данных с помощью преобразования `VectorAssembler`:

```
from pyspark.ml.feature import VectorAssembler

vector_columns = input_df.columns[1:]
assembler = VectorAssembler(inputCols=vector_columns,
                            outputCol="features")

output = assembler.transform(input_df)
output = output.select('_c0', 'features')

output.show(1, vertical=True)
...

-RECORD 0------------------------
 _c0 | 01994.jpg
 features | [0.05640895,2.709...
...

output.printSchema()
...

root
 |-- _c0: string (nullable = true)
 |-- features: vector (nullable = true)
```

В следующем разделе мы будем использовать алгоритм LSH, чтобы найти похожие изображения в нашем наборе данных.

Поиск схожих изображений с использованием PySpark LSH

Хеширование с учетом локальных особенностей — это важный класс методов хеширования, который обычно используется при кластеризации, приближенном поиске ближайшего соседа и обнаружении выбросов в больших наборах данных. Функции, чувствительные к локальным особенностям, берут две точки данных и решают, должны ли они быть парой-кандидатом.

Общая идея LSH состоит в том, чтобы использовать семейство функций («семейство LSH») для хеширования точек данных в *сегменты* таким образом, чтобы точки данных, которые находятся близко друг к другу, с высокой вероятностью находились в одних и тех же сегментах, и наоборот. Две точки данных, которые сопоставлены с одним и тем же сегментом, считаются парой-кандидатом.

В PySpark различные семейства LSH реализованы в отдельных классах (например, `MinHash` и `BucketedRandomProjection`), и в каждом классе предусмотрены API для преобразования признаков, приближенного соединения по сходству и приближенного ближайшего соседа.

Мы будем использовать реализацию LSH `BucketedRandomProjection`. Сначала создадим наш объект модели:

```
from pyspark.ml.feature import BucketedRandomProjectionLSH

brp = BucketedRandomProjectionLSH(inputCol="features",
                                  outputCol="hashes",
                                  numHashTables=200, bucketLength=2.0)
model = brp.fit(output)
```

В реализации LSH `BucketedRandomProjection` длину сегмента можно использовать для управления средним размером сегментов хеширования (и, следовательно, количеством сегментов). Большая длина сегмента (т. е. меньшее количество сегментов) увеличивает вероятность того, что признаки будут хешированы в один и тот же сегмент (увеличивая количество истинно- и ложноположительных срабатываний).

Теперь мы преобразуем входной фрейм данных, используя только что созданный объект модели LSH. Результирующий фрейм будет содержать столбец `hashes`, содержащий хешированное представление встраиваний изображений:

```
lsh_df = model.transform(output)
lsh_df.show(5)

...
+---------+--------------------+--------------------+
|      _c0|            features|              hashes|
+---------+--------------------+--------------------+
|01994.jpg|[0.05640895,2.709...|[[0.0], [-2.0], [...|
|07758.jpg|[2.1690884,3.4647...|[[0.0], [-1.0], [...|
|05257.jpg|[0.7666548,3.7960...|[[-1.0], [-1.0], ...|
|07642.jpg|[0.86353475,2.993...|[[-1.0], [-1.0], ...|
|00850.jpg|[0.49161428,2.172...|[[-1.0], [-2.0], ...|
+---------+--------------------+--------------------+
only showing top 5 rows
(показаны только верхние 5 строк)
```

Закончив преобразование набора данных с помощью LSH, мы переходим к проверке нашей работы в следующем разделе.

Поиск ближайшего соседа

Попробуем найти похожее изображение для нового входного изображения. Выберем случайным образом одно изображение из набора входных данных (рис. 10.3):

```
from IPython.display import display
from PIL import Image
```

```
input_dir_cnn = data_folder + "/images/input_images_cnn"

test_image = os.listdir(input_dir_cnn)[0]
test_image = os.path.join(input_dir_cnn, test_image)
print(test_image)
display(Image.open(test_image))
...
cars_data/images/input_images_cnn/01994.jpg
```

Рис. 10.3. Случайно выбранное изображение автомобиля из нашего набора данных

Сначала нам нужно преобразовать входное изображение в векторный формат, используя класс `Img2VecResnet18`:

```
img2vec = Img2VecResnet18()
I = Image.open(test_image)
test_vec = img2vec.getVec(I)
I.close()

print(len(test_vec))
print(test_vec)
...

512
[5.64089492e-02 2.70972490e+00 2.15519500e+00 1.43926993e-01
 2.47581363e+00 1.36641121e+00 1.08204508e+00 7.62105465e-01
[...]
5.62133253e-01 4.33687061e-01 3.95899676e-02 1.47889364e+00
 2.89110214e-01 6.61322474e-01 1.84713617e-01 9.42268595e-02]
...

test_vector = Vectors.dense(test_vec)
```

Теперь выполним приблизительный поиск ближайшего соседа:

```
print("Approximately searching lsh_df for 5 nearest neighbors \
       of input vector:")
result = model.approxNearestNeighbors(lsh_df, test_vector, 5)

result.show()
...
```

```
+---------+--------------------+--------------------+--------------------+
|      _c0|            features|              hashes|             distCol|
+---------+--------------------+--------------------+--------------------+
|01994.jpg|[0.05640895,2.709...|[[0.0], [-2.0], [...|3.691941786298668...|
|00046.jpg|[0.89430475,1.992...|[[0.0], [-2.0], [...|   10.16105522433224|
|04232.jpg|[0.71477133,2.582...|[[-1.0], [-2.0], ...|  10.255391011678762|
|05146.jpg|[0.36903867,3.410...|[[-1.0], [-2.0], ...|  10.264572173322843|
|00985.jpg|[0.530428,2.87453...|[[-1.0], [-2.0], ...|  10.474841359816633|
+---------+--------------------+--------------------+--------------------+
```

Вы можете проверить результаты поиска, представленные на рис. 10.4–10.8, чтобы убедиться, что модель делает свою работу достаточно правильно:

```
for i in list(result.select('_c0').toPandas()['_c0']):
    display(Image.open(os.path.join(input_dir_cnn, i)))
```

Как и следовало ожидать, входное изображение находится вверху списка.

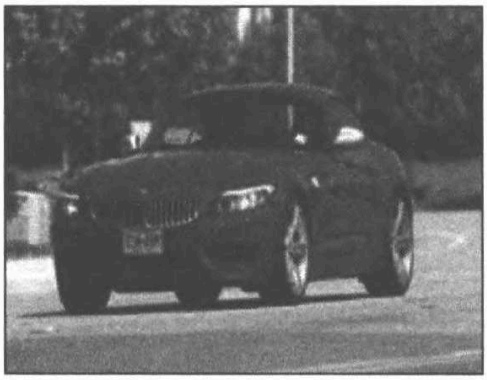


Рис. 10.4. Изображение результата 1

Рис. 10.6. Изображение результата 3

Рис. 10.5. Изображение результата 2

Рис. 10.7. Изображение результата 4

Рис. 10.8. Изображение результата 5

Что дальше?

В этой главе вы узнали, как PySpark можно комбинировать с современной инфраструктурой глубокого обучения для обнаружения сходства изображений в больших наборах данных.

Есть несколько способов улучшить эту реализацию. Вы можете попробовать использовать лучшую модель или улучшить предварительную обработку, чтобы получить лучшее качество встраивания. Кроме того, модель LSH можно настроить. В реальных условиях вам может потребоваться постоянно обновлять эталонный набор данных, чтобы учитывать новые изображения, поступающие в систему. Самый простой способ сделать это — периодически запускать пакетное задание для создания новых моделей LSH. Вы можете изучить все эти вопросы в зависимости от ваших потребностей и интересов.

Управление жизненным циклом машинного обучения с помощью MLflow

По мере того как машинное обучение становится все более популярным в различных отраслях и развертывается в производственных средах, возрастает и уровень сложности связанных с ним процессов. К счастью, появились платформы и инструменты, помогающие структурированно управлять жизненным циклом машинного обучения. Одной из таких платформ, которая хорошо работает с PySpark, является MLflow. В этой главе мы продемонстрируем использование MLflow с PySpark. Попутно представив ключевые приемы, которые вы сможете добавить в свой рабочий процесс обработки данных.

Вместо того чтобы начинать с нуля, мы будем опираться на работу, проделанную в *главе 4*. Мы вернемся к нашей реализации дерева решений и набору данных Covtype. Только на этот раз мы будем использовать MLflow для управления жизненным циклом машинного обучения.

Мы начнем с объяснения проблем и процессов, которые охватывают жизненный цикл машинного обучения. Затем мы представим MLflow и его компоненты, а также расскажем о поддержке MLflow для PySpark. За этим последует введение в отслеживание прогонов машинного обучения с использованием MLflow. Затем мы расскажем, как управлять моделями машинного обучения с помощью моделей MLflow. Потом мы обсудим развертывание нашей модели PySpark и реализуем ее. А закончим главу созданием проекта MLflow. На этом примере вы увидите, как можно сделать вашу работу воспроизводимой для коллег и единомышленников. Давайте начнем с обсуждения жизненного цикла машинного обучения.

Жизненный цикл машинного обучения

Существует несколько способов описания жизненного цикла машинного обучения. Один из простых способов — разбить его на различные компоненты или этапы, как показано на рис. 11.1. Эти шаги не обязательно расположены в таком порядке для каждого проекта, но жизненный цикл чаще всего цикличен:

- определение бизнес-проекта и согласование с заинтересованными сторонами;
- сбор и исследование данных;

- моделирование данных;
- интерпретация и обсуждение результатов;
- реализация и развертывание модели.

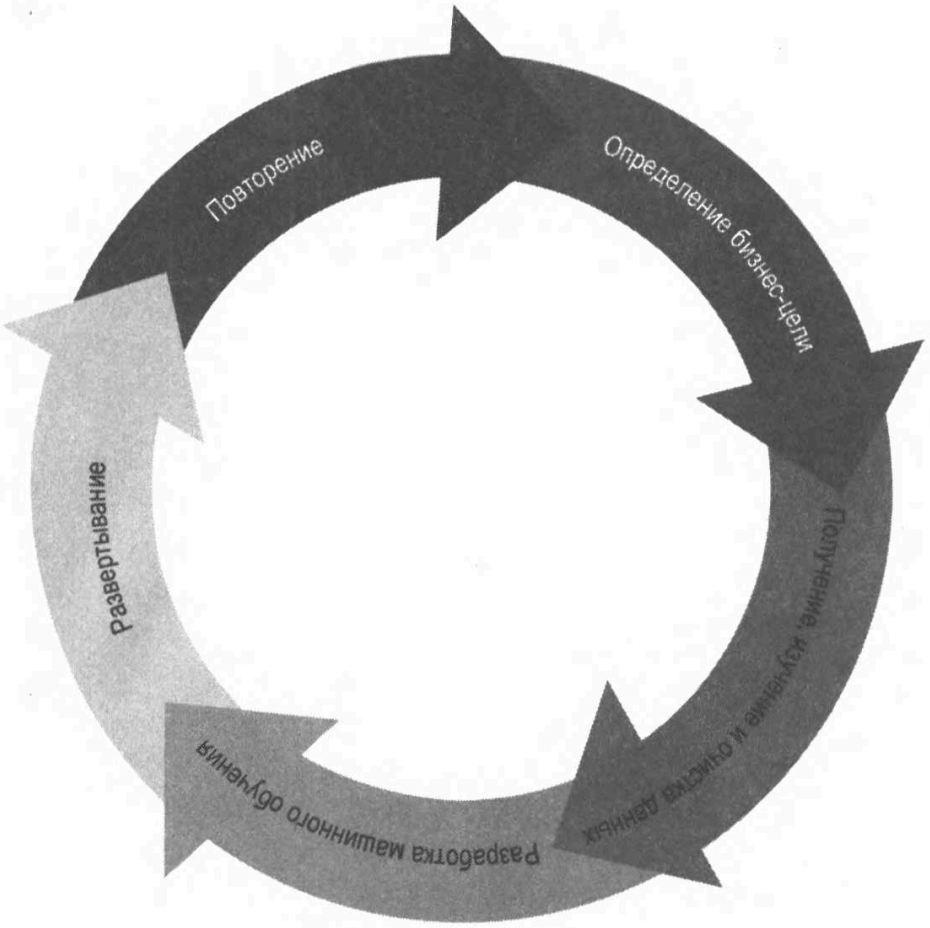


Рис. 11.1. Жизненный цикл машинного обучения

Скорость, с которой вы можете повторять жизненный цикл машинного обучения, влияет на то, насколько быстро вы сможете использовать свою работу на практике. Например, реализованная модель может устареть из-за изменения базовых данных. В этом случае вам нужно будет вернуться к прошлой работе и обновить модель.

Примеры проблем, которые могут возникнуть в течение жизненного цикла проекта машинного обучения:

Отсутствие воспроизводимости

Специалисты по данным в одной команде могут быть не в состоянии воспроизвести результаты друг друга, даже если код и параметры модели совпадают. Это может быть связано с тем, что среда выполнения (конфигурация системы или зависимости библиотек) отличается.

Отсутствие стандартизации моделей

Разные команды могут использовать разные библиотеки и разные соглашения для разработки и хранения моделей машинного обучения. Это может стать большой проблемой при распределении работы между командами.

Попытка структурировать свою работу во время прохождения жизненного цикла машинного обучения быстро становится непосильной. Это распространенная проблема, для решения которой разработаны несколько открытых и проприетарных платформ. Одной из ведущих платформ с открытым исходным кодом является MLflow, о которой мы расскажем в следующем разделе.

MLflow

MLflow — это платформа с открытым исходным кодом для управления сквозным жизненным циклом машинного обучения. Она помогает нам воспроизводить эксперименты и делиться ими, управлять моделями и развертывать модели для конечных пользователей. В дополнение к REST API и CLI она также предоставляет API для Python, R и Java/Scala.

Он состоит из четырех основных компонентов, как показано на рис. 11.2:

Отслеживание MLflow

Этот компонент записывает параметры, метрики, версии кода, модели и артефакты, такие, как графики и текст.

Проекты MLflow

Этот компонент предоставляет многоразовый воспроизводимый формат, которым можно поделиться с другими специалистами по данным или передать в производство. Он поможет вам управлять процессом обучения модели.

Модели MLflow

Этот компонент позволяет упаковывать модели для развертывания на различных платформах обслуживания моделей и вывода. Он предоставляет согласованный API для загрузки и применения моделей независимо от базовой библиотеки, используемой для построения модели.

Реестр MLflow

Этот компонент позволяет совместно отслеживать родословную модели, версии модели, переходы между этапами и аннотации в центральном хранилище.

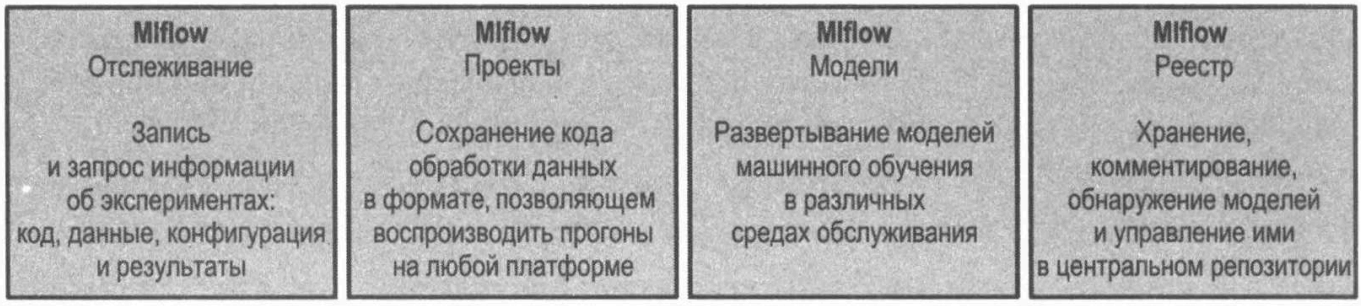


Рис. 11.2. Компоненты Mlflow

Давайте установим MLflow. Это легко делается с помощью pip:

```
$ pip3 install mlflow
```

Вот и все!

MLflow интегрируется со многими популярными платформами машинного обучения, такими, как Spark, TensorFlow, PyTorch и другими. В следующих нескольких разделах мы будем использовать встроенную поддержку Spark. Чтобы импортировать компонент MLflow, специфичный для Spark, достаточно добавить в код строку `import mlflow.spark`.

В следующем разделе мы представим компонент отслеживания MLflow Tracking и добавим его в наш код дерева решений из *главы 4*.

Отслеживание экспериментов

Типичный проект машинного обучения включает эксперименты с несколькими алгоритмами и моделями, предназначенными для решения проблемы. Необходимо отслеживать связанные наборы данных, гиперпараметры и метрики. Как правило, отслеживание экспериментов осуществляется кустарно с помощью импровизированных инструментов, таких, как электронные таблицы, и может быть неэффективным или, что еще хуже, ненадежным.

MLflow Tracking — это API и пользовательский интерфейс для регистрации параметров, версий кода, метрик и артефактов при выполнении кода машинного обучения и для последующей визуализации результатов. Вы можете использовать MLflow Tracking в любой среде (например, в автономном скрипте или блокноте) для записи результатов в локальные файлы или на сервер, а затем сравнивать несколько прогонов. Компонент не зависит от библиотеки и интегрируется с несколькими платформами.

MLflow Tracking организован вокруг концепции *прогонов* (run), которые представляют собой выполнение некоторой части кода обработки данных. MLflow Tracking предоставляет пользовательский интерфейс, который позволяет визуализировать, искать и сравнивать прогоны, а также загружать артефакты прогона или метаданные для анализа в других инструментах. Он поддерживает следующие ключевые функции:

- список прогонов по экспериментам и их сравнение;
- поиск прогонов по параметру или значению метрики;
- визуализация показателей прогона;
- загрузка результатов прогона.

Добавим MLflow Tracking в наш код дерева решений в оболочке PySpark. Предполагается, что вы загрузили набор данных Covtype и знакомы с ним. Набор данных Covtype доступен онлайн в виде сжатого файла данных в формате CSV, covtype.data.gz, и сопутствующего информационного файла, covtype.info.

Запустите `pyspark-shell`. Как упоминалось ранее, построение деревьев решений может быть ресурсоемким. Если у вашего компьютера достаточный объем памяти, выделите память командой `--driver-memory 8g` или аналогичной.

Начнем с подготовки конвейера данных и машинного обучения:

```
from pyspark.ml import Pipeline
from pyspark.sql.functions import col
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier

data_without_header = spark.read.option("inferSchema", True).\
                                option("header", False).\
                                csv("data/covtype.data")

colnames = ["Elevation", "Aspect", "Slope",
            "Horizontal_Distance_To_Hydrology",
            "Vertical_Distance_To_Hydrology",
            "Horizontal_Distance_To_Roadways",
            "Hillshade_9am", "Hillshade_Noon",
            "Hillshade_3pm", "Horizontal_Distance_To_Fire_Points"] + \
[f"Wilderness_Area_{i}" for i in range(4)] + \
[f"Soil_Type_{i}" for i in range(40)] + \
["Cover_Type"]

data = data_without_header.toDF(*colnames).\
                           withColumn("Cover_Type",
                                      col("Cover_Type").\
                                      cast(DoubleType()))

(train_data, test_data) = data.randomSplit([0.9, 0.1])

input_cols = colnames[:-1]
vector_assembler =
         VectorAssembler(inputCols=input_cols,outputCol="featureVector")

classifier = DecisionTreeClassifier(seed = 1234,
                                    labelCol="Cover_Type",
                                    featuresCol="featureVector",
                                    predictionCol="prediction")

pipeline = Pipeline(stages=[vector_assembler, classifier])
```

Чтобы начать ведение журнала с помощью MLflow, мы запускаем прогон с помощью `mlflow.start_run`. Мы будем использовать оператор `with` для автоматического завершения выполнения в конце блока:

```
import mlflow
import mlflow.spark
```

```
import pandas as pd
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

with mlflow.start_run(run_name="decision-tree"):
    # Log param: max_depth
    mlflow.log_param("max_depth", classifier.getMaxDepth())
    # Log model
    pipeline_model = pipeline.fit(train_data)
    mlflow.spark.log_model(pipeline_model, "model")
    # Log metrics: Accuracy and F1
    pred_df = pipeline_model.transform(test_data)
    evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type",
                                                  predictionCol="prediction")
    accuracy = evaluator.setMetricName("accuracy").evaluate(pred_df)
    f1 = evaluator.setMetricName("f1").evaluate(pred_df)
    mlflow.log_metrics({"accuracy": accuracy, "f1": f1})
    # Log artifact: feature importance scores
    tree_model = pipeline_model.stages[-1]
    feature_importance_df = (pd.DataFrame(list(
                                zip(vector_assembler.getInputCols(),
                                tree_model.featureImportances)),
                            columns=["feature", "importance"])
                        .sort_values(by="importance", ascending=False))
    feature_importance_df.to_csv("feature-importance.csv", index=False)
    mlflow.log_artifact("feature-importance.csv")
```

Теперь мы можем получить доступ к данным нашего эксперимента через пользовательский интерфейс отслеживания. Запустите его при помощи команды `mlflow ui`. По умолчанию он запускается на порте 5000. Значение порта по умолчанию можно изменить при помощи параметра `-p <port_name>`. После успешного запуска пользовательского интерфейса перейдите по адресу **http://localhost:5000/**. Вы увидите страницу пользовательского интерфейса, как показано на рис. 11.3. Вы можете выполнять поиск по всем прогонам, фильтровать те, которые соответствуют определенным критериям, сравнивать прогоны рядом и т. д. При желании вы также можете экспортировать содержимое в виде CSV-файла для локального анализа. В окне пользовательского интерфейса кликните на прогон с именем `decision-tree`.

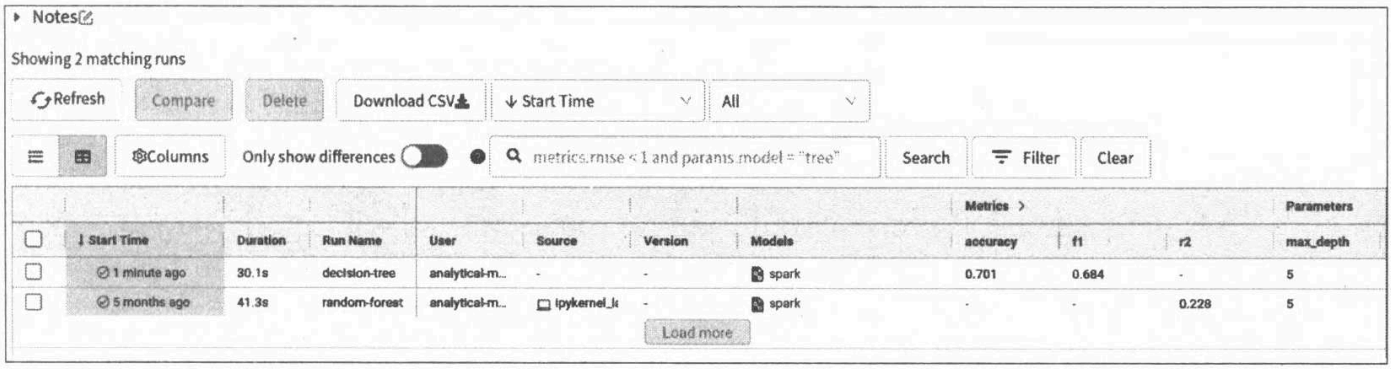


Рис. 11.3. Пользовательский интерфейс MLflow (UI 1)

При просмотре отдельного прогона, как показано на рис. 11.4, вы заметите, что MLflow сохраняет все соответствующие параметры, метрики и т. д. Вы можете добавлять примечания об этом запуске в произвольном тексте, а также теги.

Рис. 11.4. Пользовательский интерфейс MLflow (UI 2)

Теперь мы можем отслеживать и воспроизводить наши эксперименты. Далее мы обсудим управление нашими моделями с помощью MLflow.

Использование MLflow Model

MLflow Model — это стандартный формат для упаковки моделей машинного обучения, который можно использовать в различных нижестоящих инструментах, например, для обслуживания в режиме реального времени через REST API или пакетного вывода в Apache Spark. Формат определяет соглашение, позволяющее сохранять модель в разных *вариантах* (flavors), которые могут быть поняты разными библиотеками.

Варианты — ключевая концепция, которая придает моделям MLflow особую мощь. Они позволяют создавать инструменты, работающие с моделями из любой библиотеки машинного обучения, без необходимости интегрировать каждый инструмент с каждой библиотекой. MLflow определяет несколько стандартных вариантов, которые поддерживают все его встроенные инструменты развертывания, например, вариант «функция Python», описывающий запуск модели как функции Python. Однако библиотеки также могут определять и использовать другие варианты. Например, библиотека MLflow `mlflow.sklearn` позволяет загружать модели в виде объекта конвейера `scikit-learn` для использования в коде, поддерживающем `scikit-learn`, или в качестве универсальной функции Python для использования в инструментах, кото-

рым просто нужно применить модель (для например, инструмент `mlflow.sagemaker` для развертывания моделей в Amazon SageMaker).

MLflow Model — это каталог, содержащий набор файлов. Ранее мы зарегистрировали нашу модель с помощью API `log_model`. При этом был создан файл с именем MLmodel. Откройте прогон дерева решений и прокрутите вниз до раздела Artifacts **Artifacts** (Артефакты). Откройте файл MLmodel. Его содержимое должно быть похоже на то, что изображено на рис. 11.5.

```
artifact_path: model
flavors:
  python_function:
    data: sparkml
    env: conda.yaml
    loader_module: mlflow.spark
    python_version: 3.6.12
  spark:
    model_data: sparkml
    pyspark_version: 3.0.1
run_id: 0433bb047f514e28a73109bbab767222
utc_time_created: '2021-11-06 12:34:27.758915'
```

Рис. 11.5. Окно MLflow Model

Файл содержит метаданные, сигнатуру и варианты нашей модели. Сигнатура определяет схему входов и выходов модели.

Наш файл модели имеет два варианта: `python_function` и `spark`. Вариант `python_function` позволяет инструментам развертывания и обслуживания модели MLflow работать с любой моделью Python, независимо от того какая библиотека ML обучила модель. В результате любую модель Python можно легко реализовать в различных средах выполнения.

Вариант модели spark позволяет экспортировать модели Spark MLlib как MLflow Model. Например, чтобы делать прогнозы для Spark DataFrame с использованием имеющейся модели:

```
import mlflow

run_id = "0433bb047f514e28a73109bbab767222" ❶
logged_model = f'runs:/{run_id}/model' ❷

# Load model as a Spark UDF.
loaded_model = mlflow.spark.load_model(model_uri=logged_model)

# Predict on a Spark DataFrame.
preds = loaded_model.transform(test_data)
preds.select('Cover_Type', 'rawPrediction', 'probability',
             'prediction').\
              show(1, vertical=True)
...
```

```
-RECORD 0-----------------------------
Cover_Type     | 6.0
 rawPrediction | [0.0,0.0,605.0,15...
 probability   | [0.0,0.0,0.024462...
 prediction    | 3.0
only showing top 1 row
(показана только первая строка)
```

❶ Этот идентификатор можно получить из пользовательского интерфейса отслеживания в соответствующем файле Mlmodel.

❷ Мы используем f-строки Python для добавления соответствующего идентификатора прогона.

Мы также можем использовать инструмент командной строки `mlflow serve` для обслуживания модели, соответствующей конкретному идентификатору прогона.

```
$ mlflow models serve --model-uri
        runs:/0433bb047f514e28a73109bbab767222/model \
        -p 7000
...

2021/11/13 12:13:49 INFO mlflow.models.cli: Selected backend for...
2021/11/13 12:13:52 INFO mlflow.utils.conda: === Creating conda ...
Collecting package metadata (repodata.json): done
Solving environment: done ...
```

Вы успешно развернули свою модель как REST API!

Теперь мы можем использовать эту конечную точку для выполнения логического вывода. Давайте подготовим и отправим запрос на конечную точку, чтобы увидеть вывод в действии. Для этого воспользуемся библиотекой requests. Сначала установите ее с помощью pip, если она еще не установлена:

```
pip3 install requests
```

Теперь отправим запрос, содержащий объект JSON в версии pandas-split, на сервер модели:

```
import requests

host = '0.0.0.0'
port = '7001'

url = f'http://{host}:{port}/invocations'

headers = {
    'Content-Type': 'application/json;',
    'format': 'pandas-split';
}

http_data = '{"columns":["Elevation","Aspect","Slope", \
    "Horizontal_Distance_To_Hydrology", \
```

```
    "Vertical_Distance_To_Hydrology","Horizontal_Distance_To_Roadways", \
    "Hillshade_9am","Hillshade_Noon","Hillshade_3pm",\
    "Horizontal_Distance_To_Fire_Points",\
    "Wilderness_Area_0","Wilderness_Area_1","Wilderness_Area_2",\
    "Wilderness_Area_3","Soil_Type_0","Soil_Type_1","Soil_Type_2",\
    "Soil_Type_3","Soil_Type_4","Soil_Type_5","Soil_Type_6",\
    "Soil_Type_7","Soil_Type_8","Soil_Type_9","Soil_Type_10",\
    "Soil_Type_11","Soil_Type_12","Soil_Type_13",\
    "Soil_Type_14","Soil_Type_15","Soil_Type_16",\
    "Soil_Type_17","Soil_Type_18","Soil_Type_19",\
    "Soil_Type_20","Soil_Type_21","Soil_Type_22",\
    "Soil_Type_23","Soil_Type_24","Soil_Type_25",\
    "Soil_Type_26","Soil_Type_27","Soil_Type_28",\
    "Soil_Type_29","Soil_Type_30","Soil_Type_31",\
    "Soil_Type_32","Soil_Type_33","Soil_Type_34",\
    "Soil_Type_35","Soil_Type_36","Soil_Type_37",\
    "Soil_Type_38","Soil_Type_39","Cover_Type"],\
    "index":[0],\
    "data":[[2596,51,3,258,0,510,221,232,148,6279,1,\
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,\
            0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,5.0]]'\

r = requests.post(url=url, headers=headers, data=http_data)

print(f'Predictions: {r.text}')
...
Predictions: [2.0]
```

Мы не только воспроизвели сохраненную модель, но и развернули ее как REST API и выполнили вывод в реальном времени!

Давайте теперь посмотрим, как создать проект MLflow для работы, которую мы проделали раньше.

Создание и использование проектов MLflow

MLflow Projects — это стандартный формат многоразовой и воспроизводимой упаковки кода. Это автономный модуль, который объединяет весь машинный код и зависимости, необходимые для выполнения рабочего процесса машинного обучения, и позволяет вам создавать конкретную модель, работающую в любой системе или среде. MLflow Projects предоставляет API и инструменты командной строки для запуска проектов. Его также можно использовать для объединения проектов в рабочие процессы.

Каждый проект — это просто каталог файлов или репозиторий Git, содержащий ваш код. MLflow может запускать определенные проекты на основе соглашения о размещении файлов в этом каталоге (например, файл conda.yml рассматривается

как среда Conda), но вы можете описать свой проект более подробно, добавив файл MLproject, который представляет собой текстовый файл в формате YAML.

В настоящее время MLflow поддерживает следующие среды проекта: среда Conda, среда контейнера Docker и среда системы. По умолчанию MLflow использует системный путь для поиска и запуска двоичного файла Conda.

Создать типовой проект MLflow несложно. Необходимые шаги перечислены на рис. 11.6.

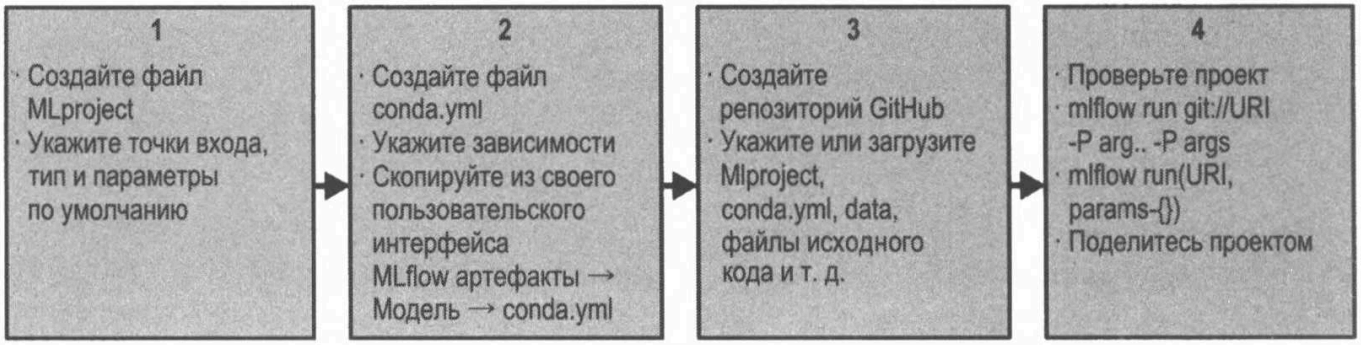


Рис. 11.6. Как создать типовой проект MLflow

Мы начнем с создания каталога нашего проекта с именем decision_tree_project:

```
mkdir decision_tree_project
cd decision_tree_project
```

Далее мы сначала создадим файл MLproject:

```
name: decision_tree_project

conda_env: conda.yml

entry_points:
  main:
    command: "python train.py"
```

Теперь нам нужен файл conda.yml. Мы можем получить его из пользовательского интерфейса MLflow, представленного в предыдущем разделе. Войдите в прогон дерева решений, который мы видели ранее. Прокрутите вниз до артефактов, кликните на файл Conda YAML и скопируйте его содержимое в conda.yml в каталоге нашего проекта:

```
channels:
- conda-forge
dependencies:
- python=3.6.12
- pip
- pip:
  - mlflow
  - pyspark==3.2.1
  - scipy==1.5.3
name: mlflow-env
```

Далее создайте скрипт Python, который будет применяться для обучения модели дерева решений при выполнении проекта MLflow. Для этого пригодится код из предыдущего раздела:

```
from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.sql.functions import col
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

spark = SparkSession.builder.appName("App").getOrCreate()

def main():
    data_without_header = spark.read.option("inferSchema", True).\
                                    option("header", False).\
                                    csv("../data/covtype.data") ❶

    colnames = ["Elevation", "Aspect", "Slope",
                "Horizontal_Distance_To_Hydrology",
                "Vertical_Distance_To_Hydrology",
                "Horizontal_Distance_To_Roadways",
                "Hillshade_9am", "Hillshade_Noon",
                "Hillshade_3pm",
                "Horizontal_Distance_To_Fire_Points"] + \
    [f"Wilderness_Area_{i}" for i in range(4)] + \
    [f"Soil_Type_{i}" for i in range(40)] + \
    ["Cover_Type"]

    data = data_without_header.toDF(*colnames).\
                               withColumn("Cover_Type",
                                          col("Cover_Type").\
                                          cast(DoubleType()))

    (train_data, test_data) = data.randomSplit([0.9, 0.1])

    input_cols = colnames[:-1]
    vector_assembler = VectorAssembler(inputCols=input_cols,
                               outputCol="featureVector")

    classifier = DecisionTreeClassifier(seed = 1234,
                                        labelCol="Cover_Type",
                                        featuresCol="featureVector",
                                        predictionCol="prediction")

    pipeline = Pipeline(stages=[vector_assembler, classifier])
```

```
    pipeline_model = pipeline.fit(train_data)
    # Log metrics: Accuracy and F1
    pred_df = pipeline_model.transform(test_data)
    evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type",
                                                  predictionCol="prediction")
    accuracy = evaluator.setMetricName("accuracy").evaluate(pred_df)
    f1 = evaluator.setMetricName("f1").evaluate(pred_df)
    print({"accuracy": accuracy, "f1": f1})

if __name__ == "__main__":
    main()
```

❶ Предполагается, что данные находятся на один уровень выше каталога выполняемого проекта Mlflow.

Данные также могут быть включены непосредственно в проект MLflow. В данном случае мы этого не делаем из-за большого размера. В таком случае данные можно совместно использовать с помощью облачного хранилища, такого, как AWS S3 или GCS.

Вы также можете локально смоделировать, как это будет работать для вашего коллеги, прежде чем делиться с ним. Это можно сделать с помощью команды `mlflow run`.

```
mlflow run decision_tree_project
...
[...]
{'accuracy': 0.6988990605087336, 'f1': 0.6805617730220171}
```

Теперь у нас есть воспроизводимый проект MLflow. Мы можем загрузить его в репозиторий GitHub и поделиться с коллегой, который сможет воспроизвести нашу работу.

Что дальше?

В этой главе был представлен проект MLflow и показана его реализация для простого проекта. В самом проекте MLflow есть что исследовать. Вы сможете найти больше информации в официальной документации. Существуют и другие инструменты, которые могут служить альтернативой MLflow. К ним относятся проекты с открытым исходным кодом, такие, как Metaflow и Kubeflow, а также проприетарные предложения крупных облачных провайдеров, включая Amazon SageMaker и платформу Databricks.

Конечно, инструменты — это только часть решения проблем, которые сопровождают любой реальный проект машинного обучения. Процессы должны определять люди, работающие над проектом. Мы надеемся, что вы воспользуетесь знаниями, полученными в этой главе, и внесете свой вклад в разработку успешных проектов машинного обучения в реальных условиях.

Об авторах

Акаш Тандон — соучредитель и технический директор Loopрanel. Работал главным инженером по данным в Atlan.

Сэнди Райза — ведущий разработчик проекта Dagster и участник проекта Apach Spark.

Ури Ласерсон — учредитель и технический директор Patch Biosciences. Ранее работал с большими данным геномики в Cloudera.

Шон Оуэн — главный архитектор решений, специализирующийся на машинном обучении и науке о данных в Databricks, является участником проекта Apache Spark и членом PMC.

Джош Уиллс — инженер-программист в WeaveGrid и бывший руководитель отдела обработки данных в Slack.

Колофон

Животное, изображенное на обложке книги — атлантический илистый прыгун (Periophthalmus barbarus), рыба-амфибия, чаще всего встречающаяся в мангровых болотах и илистых отмелях на западном побережье Африки.

Этот илистый прыгун окрашен в оливково-коричневый цвет и часто имеет синие отметины. Сильные грудные плавники позволяют легко передвигаться по суше и воде. Особое расположение лягушачьих глаз позволяет охотиться, находясь практически полностью погруженным в грязь или воду.

Атлантические илистые прыгуны — хищники, которые охотятся из засады на мелких насекомымых и ракообразных. Они всасывают воду за счет резкого расширения головы. Самцы этого вида охраняют свою территорию и, чтобы разграничить зоны (площадью примерно 10 квадратных футов), создают грязевые гребни.

Хотя атлантический прыгун занесен в список Международного союза охраны природы, как вызывающий наименьшее беспокойство, многие животные на обложках книг O'Reilly находятся под угрозой исчезновения; ведь все без исключения животные важны для окружающего мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе старинной гравюры из «Естественной истории» Джонсона.

Предметный указатель

O'REILLY®

Расширенная аналитика с PySpark

В современном мире накоплен ошеломляющий объем данных, и он продолжает расти. Один из основных инструментов анализа данных — Apache Spark, фреймворк с открытым исходным кодом для распределенной обработки неструктурированных и слабоструктурированных данных. Это практическое руководство объединяет Spark, статистические методы и наборы данных из реального мира. Авторы научат вас решать задачи анализа с помощью системы с открытым исходным кодом PySpark, применяемой для распределенной обработки больших данных с использованием Python, а также других передовых методов программирования Spark.

Специалисты по обработке данных знакомят читателей с экосистемой Spark, а затем приводят примеры базовых методов анализа, включая классификацию, кластеризацию, совместную фильтрацию и обнаружение аномалий в таких областях, как геномика, безопасность и финансы. Отдельные главы посвящены обработке изображений и библиотеке Spark NLP.

Если у вас имеются базовые знания о машинном обучении и статистике и вы программируете на Python, книга поможет вам освоить анализ больших данных.

- **Ознакомьтесь с моделью программирования и экосистемой Spark**
- **Изучите общие подходы в науке о данных**
- **Изучите практические примеры анализа больших наборов данных**
- **Узнайте, какие инструменты машинного обучения подходят для решения конкретных задач**
- **Напишите код, который можно адаптировать для многих целей**

Акаш Тандон
Соучредитель и технический директор компании Loopanel. Ранее работал главным инженером по данным в компании Atlan, специализирующейся на обработке данных и аналитике.

Сэнди Райза
Ведущий разработчик проекта Dagster, облачного оркестратора для анализа данных, и участник проекта Apache Spark.

Ури Ласерсон
Учредитель и технический директор компании Patch Biosciences, специализирующейся в области геномики и расшифровки ДНК. Ранее работал с большими данными в Cloudera, создателе дистрибутивов Apache Hadoop.

Шон Оуэн
Главный архитектор решений, специализирующийся на машинном обучении и науке о данных в Databricks — компании по разработке корпоративного программного обеспечения, основанной создателями Apache Spark. Участник проекта Apache Spark.

Джош Уиллс
Инженер-программист в компании WeaveGrid, развивающей «зеленые» технологии в сфере энергетики, и бывший руководитель отдела обработки данных в компании Slack Tecnologies, разработавшей одноименный корпоративный мессенджер.

ISBN 978-5-9775-1770-6

9 785977 517706

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®