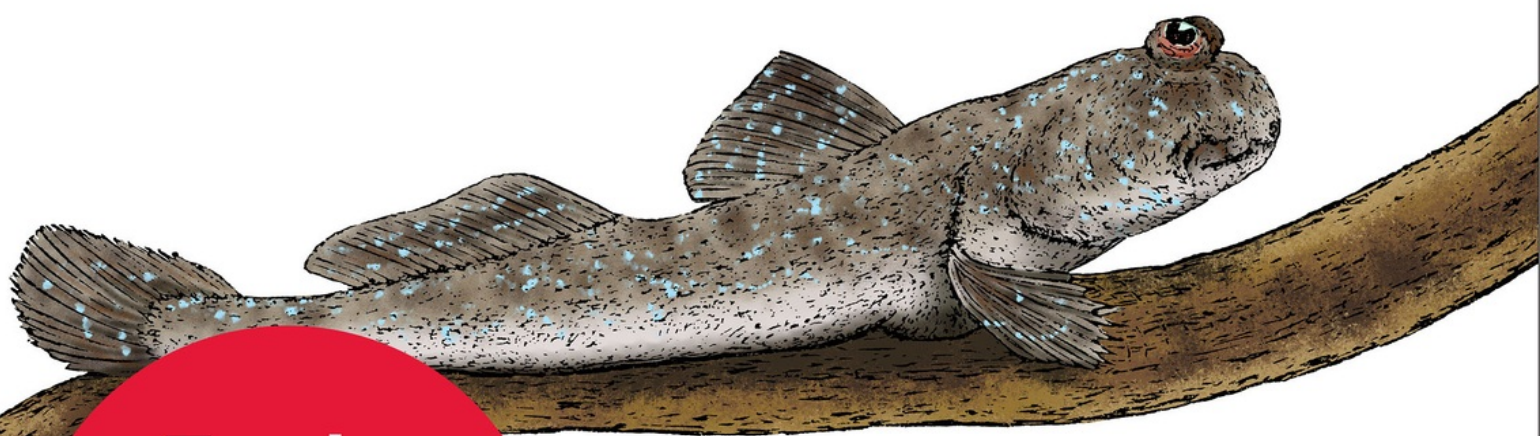


O'REILLY®

Advanced Analytics with PySpark

Patterns for Learning from Data at Scale
Using Python and Spark



**Early
Release**

**RAW &
UNEDITED**

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

Advanced Analytics with PySpark

Patterns for Learning from Data at Scale Using
Python and Spark

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

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

Advanced Analytics with PySpark

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

Copyright © 2022 Akash Tandon. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman

Development Editor: Jeff Bleiel

Production Editor: Christopher Faucher

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media, Inc.

June 2022: First Edition

Revision History for the Early Release

- 2021-04-02: First Release
- 2021-05-11: Second Release
- 2021-06-25: Third Release
- 2021-08-02: Fourth Release

- 2021-09-20: Fifth Release
- 2021-12-08: Sixth Release
- 2022-02-04: Seventh Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781098103651> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Advanced Analytics with PySpark*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10358-3

[LSI]

Chapter 1. Analyzing Big Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

When people say that we live in an age of big data they mean that we have tools for collecting, storing, and processing information at a scale previously unheard of. The following tasks simply could not have been accomplished 10 or 15 years ago:

- Build a model to detect credit card fraud using thousands of features and billions of transactions
- Intelligently recommend millions of products to millions of users
- Estimate financial risk through simulations of portfolios that include millions of instruments
- Easily manipulate data from thousands of human genomes to detect genetic associations with disease
- Assess agricultural land usage and crop yield for improved policymaking by periodically processing millions of satellite images

Sitting behind these capabilities is an ecosystem of open source software that can leverage clusters of servers to process massive amounts of data. Introduction of Apache Hadoop in late 2000s had led to widespread adoption of distributed computing. The big data ecosystem and tooling has

evolved at a rapid pace since then. The past 5 years have also seen introduction and adoption of many open source machine learning and deep learning libraries. These tools aim to leverage vast amounts of data that we now collect and store.

But just as a chisel and a block of stone do not make a statue, there is a gap between having access to these tools and all this data and doing something useful with it. Often, “doing something useful” means placing a schema over tabular data and using SQL to answer questions like “Of the gazillion users who made it to the third page in our registration process, how many are over 25?” The field of how to architect data storage and organize information (data warehouses, data lakes, etc.) to make answering such questions easy is a rich one, but we will mostly avoid its intricacies in this book.

Sometimes, “doing something useful” takes a little extra work. SQL still may be core to the approach, but in order to work around idiosyncrasies in the data or perform complex analysis, we need a programming paradigm that’s more flexible, and with richer functionality in areas like machine learning and statistics. This is where data science comes in and that’s what we are going to talk about in this book.

In this chapter, we will start by introducing big data as a concept, and challenges that arise when working with large datasets. We will then introduce Apache Spark, an open source framework for distributed computing, and its key components. Our focus will be on PySpark, Spark’s Python API, and how it fits within a wider ecosystem. This will be followed by a discussion of the changes brought by Spark 3.0, the framework’s first major release in 4 years. We will finish with a brief note about how PySpark addresses challenges of data science and is a great addition to your skillset.

Previous editions of this book had used Spark’s Scala API for code examples. We decided to use PySpark instead because of Python’s popularity in the data science community and an increased focus by core

Spark team to better support the language. By the end of this chapter, you will hopefully appreciate this decision.

Working with Big Data

Many of our favorite small data tools hit a wall when working with big data. Libraries like Pandas are not equipped to deal with data that can't fit in our RAM. Then, what should an equivalent process look like that can leverage clusters of computers to achieve the same outcomes on large data sets? Challenges of distributed computing require us to rethink many of the basic assumptions that we rely on in single-node systems. For example, because data must be partitioned across many nodes on a cluster, algorithms that have wide data dependencies will suffer from the fact that network transfer rates are orders of magnitude slower than memory accesses. As the number of machines working on a problem increases, the probability of a failure increases. These facts require a programming paradigm that is sensitive to the characteristics of the underlying system: one that discourages poor choices and makes it easy to write code that will execute in a highly parallel manner.

HOW BIG IS BIG DATA?

Without a reference point, the term big data is ambiguous. Moreover, the age-old two-tier definition of small and big data can be confusing. When it comes to data size, a three-tiered definition is more helpful.

Dataset type	Fits in RAM?	Fits on local disk?
Small dataset	Yes	Yes
Medium dataset	No	Yes
Big dataset	No	No

As per the above, if the dataset can fit in memory or disk on a single system, it can not be termed big data. This definition is not perfect but it does act as a good rule of thumb in context of an average machine.

Focus of this book is to enable you to work efficiently with big data. If your dataset is small and can fit in-memory, stay away from distributed systems. To analyze medium-sized datasets, a database or parallelism may be good enough at times. At other times, you may have to set up a cluster and use big data tools. Hopefully, the experience that you will gain in the following chapters will help you take such judgment calls.

Single-machine tools that have come to recent prominence in the software community are not the only tools used for data analysis. Scientific fields like genomics that deal with large data sets have been leveraging parallel-computing frameworks for decades. Most people processing data in these fields today are familiar with a cluster-computing environment called HPC (high-performance computing). Where the difficulties with Python and R lie in their inability to scale, the difficulties with HPC lie in its relatively low level of abstraction and difficulty of use. For example, to process a large

file full of DNA-sequencing reads in parallel, we must manually split it up into smaller files and submit a job for each of those files to the cluster scheduler. If some of these fail, the user must detect the failure and take care of manually resubmitting them. If the analysis requires all-to-all operations like sorting the entire data set, the large data set must be streamed through a single node, or the scientist must resort to lower-level distributed frameworks like MPI, which are difficult to program without extensive knowledge of C and distributed/networked systems.

Tools written for HPC environments often fail to decouple the in-memory data models from the lower-level storage models. For example, many tools only know how to read data from a POSIX filesystem in a single stream, making it difficult to make tools naturally parallelize, or to use other storage backends, like databases. Modern distributed computing frameworks provide abstractions that allow users to treat a cluster of computers more like a single computer—to automatically split up files and distribute storage over many machines, divide work into smaller tasks and execute them in a distributed manner, and recover from failures. They can automate a lot of the hassle of working with large data sets, and are far cheaper than HPC.

A simple way to think about *distributed systems* is that they are a group of independent computers that appear to the end-user as a single computer. They allow for horizontal scaling. That means means adding more computers rather than upgrading a single system (vertical scaling). The latter is relatively expensive and often insufficient for large workloads. Distributed systems are great for scaling and reliability but also introduce complexity when it comes to design, construction, and debugging. One should understand this trade-off before opting for such a tool.

Introducing Apache Spark and PySpark

Enter Apache Spark, an open source framework that combines an engine for distributing programs across clusters of machines with an elegant model for writing programs atop it. Spark originated at the UC Berkeley AMPLab and has since been contributed to the Apache Software Foundation. When

released, it was arguably the first open source software that made distributed programming truly accessible to data scientists.

Components

Spark is comprised of four main components. They are available as distinct libraries:

Spark SQL: module for working with structured data.

MLlib: scalable machine learning library.

Structured Streaming makes it easy to build scalable fault-tolerant streaming applications.

GraphX: Apache Spark's library for graphs and graph-parallel computation.¹

These components are separate from Spark's core computation engine. Spark code written by a user, using either of its APIs, is executed in the workers' JVMs (Java Virtual Machines) across the cluster. (see *Chapter 2*)

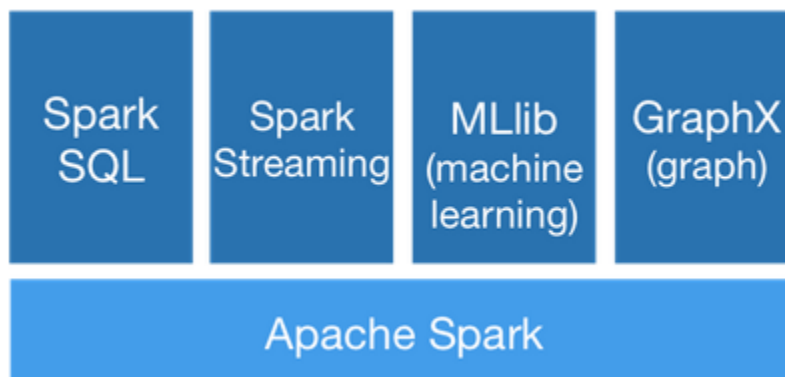


Figure 1-1. Apache Spark components (placeholder)

COMPARISON WITH MAPREDUCE

One illuminating way to understand Spark is in terms of its advances over its predecessor, Apache Hadoop's MapReduce. MapReduce revolutionized computation over huge data sets by offering a simple and resilient model for writing programs that could execute in parallel across hundreds to thousands of machines. It breaks up work into small *tasks* and can gracefully accommodate task failures without compromising the job to which they belong.

Spark maintains MapReduce's linear scalability and fault tolerance, but extends it in three important ways.

- First, rather than relying on a rigid map-then-reduce format, its engine can execute a more general directed acyclic graph (DAG) of operators. This means that in situations where MapReduce must write out intermediate results to the distributed filesystem, Spark can pass them directly to the next step in the pipeline.
- Second, it complements its computational capability with a rich set of transformations that enable users to express computation more naturally. Out-of-the-box functions are provided for various tasks including numerical computation, datetime processing and string manipulation.
- Third, Spark extends its predecessors with in-memory processing. This means that future steps that want to deal with the same data set need not recompute it or reload it from disk. Spark is well suited for highly iterative algorithms as well as adhoc queries.

PySpark

PySpark is Spark's Python API. In simpler words, PySpark is a Python-based wrapper over the core Spark framework which is written primarily in Scala. PySpark provides an intuitive programming environment for data science practitioners, and offers flexibility of Python with the distributed processing capabilities of Spark.

PySpark allows us to work across programming models. For example, a common pattern is to perform large-scale ETL work with Spark and then collect the results to a local machine followed by manipulation using Pandas. We will explore such programming models as we write PySpark code in the upcoming chapters. Here is a code example from the official documentation to give you a glimpse of what's to come:

```
from pyspark.ml.classification import LogisticRegression

# Load training data
training =
spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.t
xt")

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

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))

# We can also use the multinomial family for binary
classification
mlr = LogisticRegression(maxIter=10, regParam=0.3,
elasticNetParam=0.8, family="multinomial")

# Fit the model
mlrModel = mlr.fit(training)

# Print the coefficients and intercepts for logistic regression
with multinomial family
print("Multinomial coefficients: " +
str(mlrModel.coefficientMatrix))
print("Multinomial intercepts: " + str(mlrModel.interceptVector))
```

SPARK VS PYSPARK VS SPARKSQL

The distinction between Spark, PySpark and SparkSQL can confuse beginners. We have introduced the three terms individually. Let's summarize the differences to avoid any confusion going ahead.

- Spark: A distributed processing framework written primarily in the Scala programming language. The framework offers different language APIs on top of the core Scala-based framework.
- PySpark: Spark's Python API. Think of it as a Python-based wrapper on top of core Spark.
- SparkSQL: A Spark module for structured data processing. It is part of the core Spark framework and accessible through all of its language APIs, including PySpark.

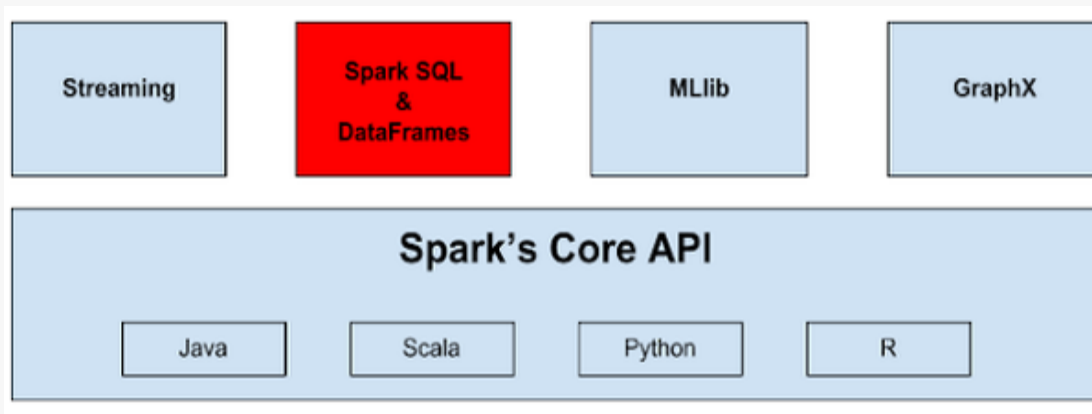


Figure 1-2. Spark, PySpark and SparkSQL (placeholder - highlight PySpark and SparkSQL, remove "& DataFrames")

Ecosystem

Spark is the closest thing akin to a swiss-knife that we have in the big data ecosystem. To top it off, it integrates well with rest of the ecosystem and is extensible. Spark decouples storage and compute unlike Apache Hadoop and HPC systems described previously. That means we can use Spark to

read data stored in many sources—Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMSs, and more—and process it all in memory. Spark’s DataFrameReaders and DataFrameWriters can also be extended to read data from other sources, such as Apache Kafka, Kinesis, Azure Storage, and Amazon S3, on which it can operate. It also supports multiple deployment modes, ranging from local environments to YARN and Kubernetes clusters.

There also exists a wide community around it. This has led to creation of many third-party packages. An official list of such packages can be found here: <https://spark.apache.org/third-party-projects.html>. An even larger community-driven list can be found at <https://spark-packages.org/>.

Spark 3.0

In 2020, Apache Spark made its first major release since 2016 when Spark 2.0 was released - Spark 3.0. This series’ last edition, released in 2017, covered changes brought about my Spark 2.0. Spark 3.0 does not introduce as many major API changes as the last major release. This release focuses on performance and usability improvements without introducing significant backward incompatibility.

The Spark SQL module has seen major performance enhancements in the form of adaptive query execution, and dynamic partition pruning. In simpler terms, they allow Spark to adapt physical execution plan during runtime and skip over data that’s not required in a query’s results respectively. These optimizations address significant effort that users had to put into manual tuning and optimization. Spark 3.0 is almost two-times faster than Spark 2.4 on TPC-DS, an industry-standard analytical processing benchmark. Since most Spark applications are backed by the SQL engine, all the higher-level libraries, including MLlib and structured streaming, and higher level APIs, including SQL and DataFrames have benefitted. Compliance with ANSI SQL standard makes the SQL API more usable.

Python has emerged as the leader in terms of adoption in the data science ecosystem. Consequently, Python is now the most widely used language on Spark. PySpark has more than 5 million monthly downloads on PyPI, the Python Package Index. Spark 3.0 improves its functionalities and usability. Pandas user-defined functions (UDFs) have been redesigned to support Python type hints and iterators as arguments. New pandas UDF types have been included and the error handling is now more pythonic. Python versions below 3.6 have been deprecated.

Over the last 4 years, the data science ecosystem has also changed at a rapid pace. There is an increased focus on putting machine learning models in production. Deep learning has provided remarkable results and Spark team is currently experimenting to allow the project's scheduler to leverage accelerators such as GPUs.

PySpark Addresses Challenges of Data Science

For a system that seeks to enable complex analytics on huge data to be successful, it needs to be informed by—or at least not conflict with—some fundamental challenges faced by data scientists.

- First, the vast majority of work that goes into conducting successful analyses lies in preprocessing data. Data is messy, and cleansing, munging, fusing, mushing, and many other verbs are prerequisites to doing anything useful with it.
- Second, *iteration* is a fundamental part of data science. Modeling and analysis typically require multiple passes over the same data. Popular optimization procedures like stochastic gradient descent involve repeated scans over their inputs to reach convergence. Iteration also matters within the data scientist's own workflow. Choosing the right features, picking the right algorithms, running the right significance tests, and finding the right hyperparameters all require experimentation.

- Third, the task isn't over when a well-performing model has been built. The point of data science is to make data useful to non-data scientists. Uses of data recommendation engines and real-time fraud detection systems culminate in data applications. In such systems, models become part of a production service and may need to be rebuilt periodically or even in real time.

PySpark deals well with the aforementioned challenges of data science, acknowledging that the biggest bottleneck in building data applications is not CPU, disk, or network, but analyst productivity. Collapsing the full pipeline, from preprocessing to model evaluation, into a single programming environment can speed up development. By packaging an expressive programming model with a set of analytic libraries under a REPL, PySpark avoids the roundtrips to IDEs. The more quickly analysts can experiment with their data, the higher likelihood they have of doing something useful with it.

PySpark's core APIs provide a strong foundation for data transformation independent of any functionality in statistics, machine learning, or matrix algebra. When exploring and getting a feel for a data set, data scientists can keep data in memory while they run queries, and easily cache transformed versions of the data as well without suffering a trip to disk. As framework that makes modeling easy but is also a good fit for production systems, it is a huge win for the data science ecosystem.

Where to go from here

Spark spans the gap between systems designed for exploratory analytics and systems designed for operational analytics. It is often quoted that a data scientist is someone who is better at engineering than most statisticians, and better at statistics than most engineers. At the very least, Spark is better at being an operational system than most exploratory systems and better for data exploration than the technologies commonly used in operational systems.

We hope that this chapter was helpful and you are now excited for getting hands-on with PySpark. That's what we will do from next chapter onwards!

-
- 1 GraphFrames is an open-source general graph processing library that is similar to Apache Spark's GraphX but uses DataFrame-based APIs. For graph analytics, GraphFrames is recommended instead of GraphX which lacks Python bindings.

Chapter 2. Introduction to Data Analysis with PySpark

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

Python is the most widely used language for data science tasks. The prospect of being able to do statistical computing and web programming using the same language had contributed to its rise in popularity in early 2010s. This has led to a thriving ecosystem of tools and a helpful community for data analysis, often referred as the PyData ecosystem. This is a big reason for PySpark’s popularity. Being able to leverage distributed computing via Spark in Python helps data science practitioners be more productive because of familiarity with the programming language and presence of a wide community. For that same reason, we have opted to write our examples in PySpark.

It’s difficult to express how transformative it is to do all of your data munging and analysis in a single environment, regardless of where the data itself is stored and processed. It’s the sort of thing that you have to experience to understand, and we wanted to be sure that our examples captured some of that magic feeling we experienced when we first started using PySpark. For example, PySpark provides interoperability with Pandas, which is one of the most popular PyData tools. We will explore this feature further in the chapter.

In this chapter, we will explore PySpark’s powerful Dataframe API via a data cleansing exercise. In PySpark, the `DataFrame` is an abstraction for data sets that have a regular structure in which each record is a row made up of a set of columns, and each column has a well-defined data type. You can think of a data frame as the Spark analogue of a table in a relational database. Even though the naming convention might make you think of a `data.frame` object in a

`pandas.DataFrame` object, Spark's DataFrames are a different beast. This is because they represent distributed data sets on a cluster, not local data where every row in the data is stored on the same machine. Although there are similarities in how you use DataFrames and the role they play inside the Spark ecosystem, there are some things you may be used to doing when working with data frames in Pandas or R that do not apply to Spark, so it's best to think of them as their own distinct entity and try to approach them with an open mind.

As for data cleansing, it is the first step in any data science project, and often the most important. Many clever analyses have been undone because the data analyzed had fundamental quality problems or underlying artifacts that biased the analysis or led the data scientist to see things that weren't really there. Hence, what better way to introduce you to working with data using PySpark and DataFrames than a data cleansing exercise?

First, we will introduce PySpark's fundamentals and practice them using a sample dataset from the UC Irvine Machine Learning Repository. We will reiterate why PySpark is a good choice for Data Science and introduce its programming model. We will then set up PySpark on our system or cluster and analyze our dataset using PySpark's DataFrame API. Most of your time using PySpark for data analysis will center around the DataFrame API so get ready to become intimately familiar with it. This will set us up for the following chapters where we delve into various machine learning algorithms.

You don't need to deeply understand the how Spark works under the hood for performing data science tasks. However, understanding basic concepts about Spark's architecture will make it easier to work with PySpark and take better decisions when writing code. That is what we will cover in the next section.

When using the DataFrame API, your PySpark code should provide comparable performance with Scala. If you're using a UDF (user defined function) or RDDs, you will have a performance impact.

Spark Architecture

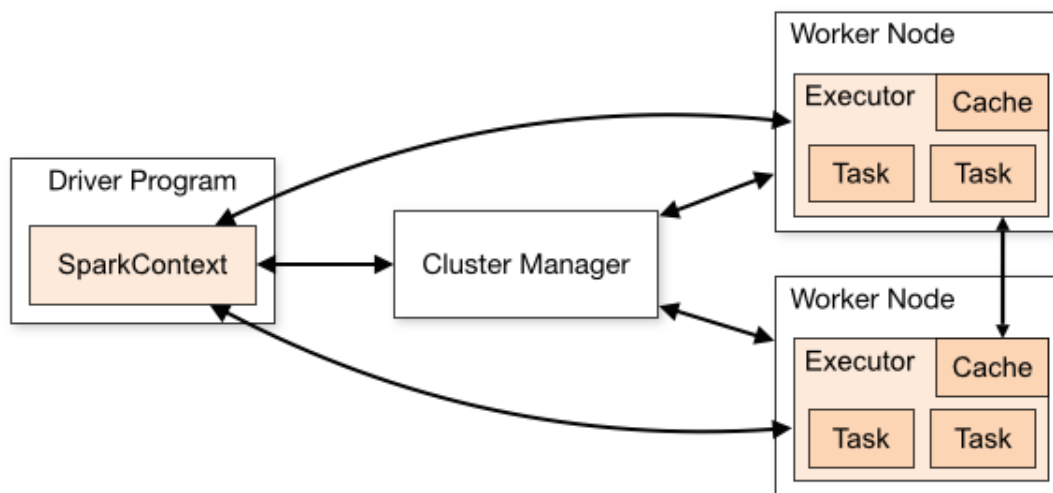


Figure 2-1. Spark architecture diagram (placeholder - replace SparkContext with SparkSession)

Figure 2.1 depicts the Spark architecture through high-level components. Spark applications run as independent sets of processes on a cluster or locally. At a high level, a Spark application is comprised of a driver process, cluster manager and a set of executor processes. The driver program is the central component and responsible for distributing tasks across executor processes. There will always be just one driver process. When we talk about scaling, we mean increasing the number of executors. The cluster manager simply manages resources.

Spark is a distributed, data-parallel compute engine. In the data-parallel model, more data partitions equals more parallelism. Partitioning allows for efficient parallelism. A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth. That is, each executor's core is assigned its own data partition to work on. Remember this whenever a choice related to partitioning comes up.

Spark programming starts with a data set, usually residing in some form of distributed, persistent storage like HDFS and in a format like Parquet. Writing a Spark program typically consists of a few steps:

1. Define a set of transformations on the input data set.
2. Invoke actions that output the transformed data sets to persistent storage or return results to the driver's local memory. These actions will ideally be performed by the worker nodes, as depicted on the right in Figure 2.1.

3. Run local computations that operate on the results computed in a distributed fashion. These can help you decide what transformations and actions to undertake next.

It's important to remember that all of PySpark's higher-level abstractions still rely on the same philosophy that has been present in Spark since the very beginning: the interplay between storage and execution. Understanding these principles will help you make better use of Spark for data analysis.

Next, we will install and set up PySpark on our machine so that we can start performing data analysis. This is a one-time exercise that will help us run the code examples from this and following chapters.

Installing PySpark

The examples and code in this book assume you have Spark 3.0.1 available. For the purpose of following the code examples, install PySpark from the [PyPi repository](#).

```
$ pip install pyspark
```

At the time of writing, PySpark is reported to be incompatible with Python 3.8. Please use Python 3.6 or 3.7 for following the code examples.

If you want SQL, ML, and/or MLlib as extra dependencies, that's an option too. We will need these ahead.

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

Installing from PyPi skips the libraries required to run Scala, Java or R. Full releases can be obtained from the [Spark project site](#). Refer to the [Spark documentation](#) for instructions on setting up a Spark environment, whether on a cluster or simply on your local machine.

Now we're ready to launch the `pyspark-shell`, which is a REPL for the Python language that also has some Spark-specific extensions. This is similar to

Python or IPython shell that you may have used. If you're just running these examples on your personal computer, you can launch a local Spark cluster by specifying `local[N]`, where `N` is the number of threads to run, or `*` to match the number of cores available on your machine. For example, to launch a local cluster that uses eight threads on an eight-core machine:

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

A Spark application itself is often referred to as a Spark *cluster*. That is a logical abstraction and is different from a physical cluster (multiple machines).

If you have a Hadoop cluster that runs a version of Hadoop that supports YARN, you can launch the Spark jobs on the cluster by using the value of `yarn` for the Spark master:

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


The rest of the examples in this book will not show a `--master` argument to `spark-shell`, but you will typically need to specify this argument as appropriate for your environment.

You may need to specify additional arguments to make the Spark shell fully utilize your resources. A list of arguments can be found by executing `pyspark --help`. For example, when running Spark with a local master, you can use `--driver-memory 2g` to let the single local process use 2 GB of memory. YARN memory configuration is more complex, and relevant options like `--executor-memory` are explained in the [Spark on YARN documentation](#).

The Spark framework officially supports 4 cluster deployment modes - standalone, YARN, Kubernetes and Mesos. More details can be found in the [Deploying Spark documentation](#).

After running one of these commands, you will see a lot of log messages from Spark as it initializes itself, but you should also see a bit of ASCII art, followed by some additional log messages and a prompt:

```
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'.
```

You can run the `:help` command in the shell. This will prompt you to either start an interactive help mode or ask for help about specific Python objects. In addition to the note about `:help`, the Spark log messages indicated “SparkSession available as *spark*.” This is a reference to the `SparkSession`, which acts as an entrypoint to all Spark operations and data. Go ahead and type `spark` at the command line:

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

The REPL will print the string form of the object. For the `SparkSession` object, this is simply its name plus the hexadecimal address of the object in memory. (DEADBEEF is a placeholder; the exact value you see here will vary from run to run.) In an interactive Spark shell, the Spark driver instantiates a `SparkSession` for you, while in a Spark application, you create a `SparkSession` object yourself.

In Spark 2.0, the `SparkSession` became a unified channel to all Spark operations and data. Previously used entry points such as `SparkContext`, `SQLContext`, `HiveContext`, `SparkConf`, and `StreamingContext` can be accessed through it too.

What exactly do we do with the `spark` variable? `SparkSession` is an object, so it has methods associated with it. We can see what those methods are in the PySpark shell by typing the name of a variable, followed by a period, followed by tab:

```
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(
```

Out of all the methods provided by `SparkSession`, the ones that we're going to use most often allow us to create *DataFrames*. Now that we have set up PySpark, we can set up our dataset of interest and start using PySpark's DataFrame API to interact with it. That's what we will do in the next section.

Setting up our data

The UC Irvine Machine Learning Repository is a fantastic source for interesting (and free) data sets for research and education. The data set we'll analyze was curated from a record linkage study performed at a German hospital in 2010, and it contains several million pairs of patient records that were matched according to several different criteria, such as the patient's name (first and last), address, and birthday. Each matching field was assigned a numerical score from 0.0 to 1.0 based on how similar the strings were, and the data was then hand-labeled to identify which pairs represented the same person and which did not. The underlying values of the fields that were used to create the data set were removed to protect the privacy of the patients. Numerical identifiers, the match scores for the fields, and the label for each pair (match versus nonmatch) were published for use in record linkage research.

RECORD LINKAGE

The general structure of the a record linkage problem is something like this: we have a large collection of records from one or more source systems, and it is likely that multiple records refer to the same underlying entity, such as a customer, a patient, or the location of a business or an event. Each entity has a number of attributes, such as a name, an address, or a birthday, and we will need to use these attributes to find the records that refer to the same entity. Unfortunately, the values of these attributes aren't perfect: values might have different formatting, typos, or missing information that means that a simple equality test on the values of the attributes will cause us to miss a significant number of duplicate records. For example, let's compare the business listings shown in [Table 2-1](#).

*T
a
b
l
e*

*2
-*
1

*.
T
h
e*

*c
h
a
l
l
e
n
g
e*

*o
f
r
e
c
o
r
d*

*l
i
n
k*

a
g
e

Name	Address	City	State	Phone
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

The first two entries in this table refer to the same small coffee shop, even though a data entry error makes it look as if they are in two different cities (West Hollywood and Hollywood). The second two entries, on the other hand, are actually referring to different business locations of the same chain of coffee shops that happen to share a common address: one of the entries refers to an actual coffee shop, and the other one refers to a local corporate office location. Both of the entries give the official phone number of corporate headquarters in Seattle.

This example illustrates everything that makes record linkage so difficult: even though both pairs of entries look similar to each other, the criteria that we use to make the duplicate/not-duplicate decision is different for each pair. This is the kind of distinction that is easy for a human to understand and identify at a glance, but is difficult for a computer to learn.

Record linkage goes by a lot of different names in the literature and in practice: entity resolution, record deduplication, merge-and-purge, and list washing. For our purposes, we refer to this problem as *record linkage*.

From the shell, let's pull the data from the repository:

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

If you have a Hadoop cluster handy, you can create a directory for the block data in HDFS and copy the files from the data set there:

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

To create a data frame for our record linkage data set, we're going to use the `SparkSession` object. Specifically, we will use the `csv` method on its `Reader API`:

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

By default, every column in a CSV file is treated as a `string` type, and the column names default to `_c0`, `_c1`, `_c2`, We can look at the head of a data frame in the shell by calling its `show` method:

```
prev.show()
```

We can see that the first row of the `DataFrame` is the name of the header columns, as we expected, and that the CSV file has been cleanly split up into its individual columns. We can also see the presence of the `?` strings in some of the columns; we will need to handle these as missing values. In addition to naming each column correctly, it would be ideal if Spark could properly infer the data type of each of the columns for us.

Fortunately, Spark's CSV reader provides all of this functionality for us via options that we can set on the reader API. You can see the full list of options that the API takes in the [pyspark documentation](#). For now, we'll read and parse the linkage data like this:

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

When we call `show` on the `parsed` data, we see that the column names are set correctly and the `?` strings have been replaced by `null` values. To see the inferred type for each column, we can print the schema of the `parsed` `DataFrame` like this:

```
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)
...
```

Each `StructField` instance contains the name of the column, the most specific data type that could handle the type of data contained in each record, and a boolean field that indicates whether or not the column may contain null values, which is true by default. In order to perform the schema inference, Spark must do *two* passes over the data set: one pass to figure out the type of each column, and a second pass to do the actual parsing.

If you know the schema that you want to use for a file ahead of time, you can create an instance of the `pyspark.sql.types.StructType` class and pass it to the Reader API via the `schema` function, which can have a significant performance benefit when the data set is very large, since Spark will not need to perform an extra pass over the data to figure out the data type of each column.

Here is an example of defining a schema using `StructType` and `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("...")
```

Another way to define the schema is using DDL (data definition statements).

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

DATA FORMATS AND DATA SOURCES

Spark ships with built-in support for reading and writing data frames in a variety of formats via the DataFrameReader and DataFrameWriter APIs. In addition to the CSV format discussed here, you can also read and write structured data from the following sources:

parquet

Leading columnar-oriented data storage format (default option in Spark)

orc

Another columnar-oriented data storage format

json

Supports many of the same schema-inference functionality that the CSV format does

jdbc

Connects to a relational database via the JDBC data connection standard

avro

Provides efficient message serialization and deserialization when using a streaming source such as Apache Kafka

text

Maps each line of a file to a data frame with a single column of type `string`

image

Load image files from a directory as a data frame with one column, containing image data stored as image schema

libsvm

Popular text file format for representing labeled observations with sparse features

binary

Reads binary files and converts each file into a single data frame row (new in Spark 3.0)

You access the methods of the `DataFrameReader` API by calling the `read` method on a `SparkSession` instance, and you can load data from a file using either the `format` and `load` methods, or one of the shortcut methods for built-in formats:

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

In this example, `d1` and `d2` reference the same underlying JSON data and will have the same contents. Each of the different file formats has its own set of options that can be set via the same `option` method that we used for CSV files.

To write data out again, you access the `DataFrameWriter` API via the `write` method on any `DataFrame` instance. The `DataFrameWriter` API supports the same built-in formats as the `DataFrameReader` API, so the following two methods are equivalent ways of writing the contents of the `d1` `DataFrame` as a Parquet file:

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

By default, Spark will throw an error if you try to save a data frame to a file that already exists. You can control Spark's behavior in this situation via the `mode` method on the `DataFrameWriter` API to either **Overwrite** the existing file, **Append** the data in the `DataFrame` to the file (if it exists), or **Ignore** the write operation if the file already exists and leave it in place:

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

You can specify the mode as a string literal ("overwrite", "append", "ignore").

DataFrames have a number of methods that allow us to read data from the cluster into the PySpark REPL on our client machine. Perhaps the simplest of these is `first`, which returns the first element of the DataFrame into the client:

```
parsed.first
...
Row(id_1='3148', id_2='8326', cmp_fname_c1='1', cmp_fname_c2=None, ...)
```

The `first` method can be useful for sanity checking a data set, but we're generally interested in bringing back larger samples of a DataFrame into the client for analysis. When we know that a DataFrame only contains a small number of records, we can use the `toPandas`, or `collect` method to return all the contents of a DataFrame to the client as an array. For extremely large DataFrames using these methods can be dangerous, and cause an out-of-memory (OOM) exception. Because we don't know how big the linkage data set is just yet, we'll hold off on doing this right now.

In the next several sections, we'll use a mix of local development and testing and cluster computation to perform more munging and analysis of the record linkage data, but if you need to take a moment to drink in the new world of awesome that you have just entered, we certainly understand.

TRANSFORMATIONS AND ACTIONS

The act of creating a DataFrame does not cause any distributed computation to take place on the cluster. Rather, DataFrames define logical data sets that are intermediate steps in a computation. Spark operations on distributed data can be classified into two types: transformations and actions.

All transformations are evaluated lazily. That is, their results are not computed immediately, but they are recorded as a lineage. This allows Spark to optimize the query plan. Distributed computation occurs upon invoking an *action* on a DataFrame. For example, the `count` action returns the number of objects in an DataFrame:

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

The `collect` action returns an `Array` with all the `Row` objects from the DataFrame. This `Array` resides in local memory, not on the cluster:

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

Actions need not only return results to the local process. The `save` action saves the contents of an to persistent storage, such as HDFS:

```
rdd.write.format("parquet").("hdfs:///user/ds/mynumbers")
```

Remember that `DataFrameReader` can accept a directory of text files as input, meaning that a future Spark job could refer to `mynumbers` as an input directory.

Analyzing Data with the DataFrame API

The DataFrame API comes with a powerful set of tools that will likely be familiar to data scientists who are used to Python, and SQL. In this section, we

will begin to explore these tools and how to apply them to the record linkage data.

If we look at the schema of the `parsed` DataFrame and the first few rows of data, we see this.

```
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 | cmp_sex | cmp_b |
| d | cmp_bm | cmp_by | cmp_plz | is_match |
+-----+-----+-----+-----+-----+-----+-----+
| 3148 | 8326 | 1.0 | null | 1.0 | null |
1 | 1 | 1 | 1 | 1 | true |
| 14055 | 94934 | 1.0 | null | 1.0 | null |
1 | 1 | 1 | 1 | 1 | true |
| 33948 | 34740 | 1.0 | null | 1.0 | null |
1 | 1 | 1 | 1 | 1 | true |
| 946 | 71870 | 1.0 | null | 1.0 | null |
1 | 1 | 1 | 1 | 1 | true |
| 64880 | 71676 | 1.0 | null | 1.0 | null |
1 | 1 | 1 | 1 | 1 | true |
```

- The first two fields are integer IDs that represent the patients that were matched in the record.

- The next nine values are (possibly missing) numeric values (either doubles or ints) that represent match scores on different fields of the patient records, such as their names, birthdays, and locations. The fields are stored as integers when the only possible values are match (1) or no-match (0), and doubles whenever partial matches are possible.
- The last field is a boolean value (`true` or `false`) indicating whether or not the pair of patient records represented by the line was a match.

Our goal is to come up with a simple classifier that allows us to predict whether a record will be a match based on the values of the match scores for the patient records. Let's start by getting an idea of the number of records we're dealing with via the `count` method:

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

This is a relatively small data set—certainly small enough to fit in memory on one of the nodes in a cluster or even on your local machine if you don't have a cluster available. Thus far, every time we've processed the data, Spark has reopened the file, reparsed the rows, and then performed the action requested, like showing the first few rows of the data or counting the number of records. When we ask another question, Spark will do these same operations, again and again, even if we have filtered the data down to a small number of records or are working with an aggregated version of the original data set.

This isn't an optimal use of our compute resources. After the data has been parsed once, we'd like to save the data in its parsed form on the cluster so that we don't have to reparse it every time we want to ask a new question of the data. Spark supports this use case by allowing us to signal that a given `DataFrame` should be cached in memory after it is generated by calling the `cache` method on the instance. Let's do that now for the `parsed` `DataFrame`:

```
parsed.cache()
```

CACHING

Although the contents of DataFrames are transient by default, Spark provides a mechanism for persisting the underlying data:

```
cached.cache()  
cached.count()  
cached.take(10)
```

The call to `cache` indicates that the contents of the DataFrame should be stored in memory the next time it's computed. In this example, the call to `count` computes the contents initially, and the `take` action returns the first 10 elements of the DataFrame as a local `Array[Row]`. When `take` is called, it accesses the cached elements of `cached` instead of recomputing them from their dependencies.

Spark defines a few different mechanisms, or `StorageLevel` values, for persisting data. `cache()` is shorthand for `persist(StorageLevel.MEMORY)`, which stores the rows as unserialized Java objects. When Spark estimates that a partition will not fit in memory, it simply will not store it, and it will be recomputed the next time it's needed. This level makes the most sense when the objects will be referenced frequently and/or require low-latency access, because it avoids any serialization overhead. Its drawback is that it takes up larger amounts of memory than its alternatives. Also, holding on to many small objects puts pressure on Java's garbage collection, which can result in stalls and general slowness.

Spark also exposes a `MEMORY_SER` storage level, which allocates large byte buffers in memory and serializes the records into them. When we use the right format (more on this in a bit), serialized data usually takes up two to five times less space than its raw equivalent.

Spark can use disk for caching data as well. The `MEMORY_AND_DISK` and `MEMORY_AND_DISK_SER` are similar to the `MEMORY` and `MEMORY_SER` storage levels, respectively. For the latter two, if a partition will not fit in memory, it is simply not stored, meaning that it must be recomputed from its dependencies the next time an action uses it. For the former, Spark spills partitions that will not fit in memory to disk.

Spark can use the detailed knowledge of the data stored with a data frame available via the DataFrame's schema to persist the data efficiently. Deciding when to cache data can be an art. The decision typically involves trade-offs between space and speed, with the specter of garbage-collecting looming overhead to occasionally confound things further. In general, data should be cached when it is likely to be referenced by multiple actions, is relatively small compared to the amount of memory/disk available on the cluster, and is expensive to regenerate.

Once our data has been cached, the next thing we want to know is the relative fraction of records that were matches versus those that were nonmatches.

```
from pyspark.sql.functions import col

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

Instead of writing a function to extract the `is_match` column, we simply pass its name to the `groupBy` method on the DataFrame, call the `count` method to, well, count the number of records inside each grouping, sort the resulting data in descending order based on the `count` column, and then cleanly render the result of the computation in the REPL with `show`. Under the covers, the Spark engine determines the most efficient way to perform the aggregation and return the results. This represents the clean, fast, and expressive way to do data analysis that Spark provides.

Note that there are two ways we can reference the names of the columns in the DataFrame: either as literal strings, like in `groupBy("is_match")`, or as `Column` objects by using the `"col()"` function that we used on the `count` column. Either approach is valid in most cases, but we needed to use the `col` function to call the `desc` method on the resulting `count` column object.

DATAFRAME AGGREGATION FUNCTIONS

In addition to `count`, we can also compute more complex aggregations like sums, mins, maxes, means, and standard deviation using the `agg` method of the DataFrame API in conjunction with the aggregation functions defined in the `pyspark.sql.functions` collection. For example, to find the mean and standard deviation of the `cmp_sex` field in the overall `parsed` DataFrame, we could type:

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

Note that by default, Spark computes the sample standard deviation; there is also a `stddev_pop` function for computing the population standard deviation.

You may have noticed that the functions on the DataFrame API are similar to the components of a SQL query. This isn't a coincidence, and in fact we have the option of treating any DataFrame we create as if it were a database table and expressing our questions using familiar and powerful SQL syntax. First, we need to tell the Spark SQL execution engine the name it should associate with the `parsed` DataFrame, since the name of the variable itself ("`parsed`") isn't available to Spark:

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

Because the `parsed` DataFrame is only available during the length of this PySpark REPL session, it is a *temporary* table. Spark SQL may also be used to query persistent tables in HDFS if we configure Spark to connect to an Apache Hive metastore that tracks the schemas and locations of structured data sets.

Once our temporary table is registered with the Spark SQL engine, we can query it like this:

```

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|
+-----+-----+

```

You have the option of running Spark using either an ANSI 2003-compliant version of Spark SQL (the default) or in HiveQL mode by calling the `enableHiveSupport` method when you create a `SparkSession` instance via its Builder API.

CONNECTING SPARK SQL TO HIVE

You can connect to a Hive metastore via a *hive-site.xml* file, and you can also use HiveQL in queries by calling the `enableHiveSupport` method on the `SparkSession` Builder API:

```

sparkSession =
SparkSession.builder.master("local[4]").enableHiveSupport().getOrCreate()

```

You can treat any table in the Hive metastore as a data frame, execute Spark SQL queries against tables defined in the metastore, and persist the output of those queries to the metastore so that they can be queried by other tools, including Hive itself, Apache Impala, or Presto.

Should you use Spark SQL or the DataFrame API to do your analysis in PySpark? There are pros and cons to each: SQL has the benefit of being broadly familiar and expressive for simple queries. It is also the best way to quickly read and filter data stored in commonly used columnar file formats like Parquet and ORC. It also lets you query data using JDBC/ODBC connectors from databases such as PostgreSQL or tools such as Tableau. The downside of SQL is that it can be difficult to express complex, multistage analyses in a dynamic, readable,

and testable way—all areas where the DataFrame API shines. Throughout the rest of the book, we use both Spark SQL and the DataFrame API, and leave it as an exercise for the reader to examine the choices we made and translate our computations from one interface to the other.

We can apply functions one-by-one to our DataFrame to obtain statistics such as count and mean. However, PySpark offers a better way to obtain summary statistics for DataFrames and that's what we will cover in the next section.

Fast Summary Statistics for DataFrames

Although there are many kinds of analyses that may be expressed equally well in SQL or with the DataFrame API, there are certain common things that we want to be able to do with data frames that can be tedious to express in SQL. One such analysis that is especially helpful is computing the min, max, mean, and standard deviation of all the non-null values in the numerical columns of a data frame. In PySpark, this function has the same name that it does in Pandas, `describe`:

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

The `summary` DataFrame has one column for each variable in the `parsed` DataFrame, along with another column (also named `summary`) that indicates which metric—`count`, `mean`, `stddev`, `min`, or `max`—is present in the rest of the columns in the row. We can use the `select` method to choose a subset of the columns in order to make the summary statistics easier to read and compare:

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

Note the difference in the value of the `count` variable between `cmp_fname_c1` and `cmp_fname_c2`. While almost every record has a non-null value for `cmp_fname_c1`, less than 2% of the records have a non-null value for `cmp_fname_c2`. To create a useful classifier, we need to rely on variables that are almost always present in the data—unless their missingness indicates something meaningful about whether the record matches.

Once we have an overall feel for the distribution of the variables in our data, we want to understand how the values of those variables are correlated with the value of the `is_match` column. Therefore, our next step is to compute those same summary statistics for just the subsets of the `parsed` DataFrame that correspond to matches and nonmatches. We can filter DataFrames using either SQL-style `where` syntax or with `Column` objects using the DataFrame API and then use `describe` on the resulting DataFrames:

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

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

The logic inside the string we pass to the `where` function can include statements that would be valid inside a `WHERE` clause in Spark SQL. For the filtering condition that uses the DataFrame API, we use the `==` operator on the `"is_match"` column object to check for equality with the boolean object, `False`. Note that the `where` function is an alias for the `filter` function; we could have reversed the `where` and `filter` calls in the above snippet and everything would have worked the same way.

We can now start to compare our `matchSummary` and `missSummary` DataFrames to see how the distribution of the variables changes depending on whether the record is a match or a miss. Although this is a relatively small data set, doing this comparison is still somewhat tedious—what we really want is to transpose the `matchSummary` and `missSummary` DataFrames so that the rows and columns are swapped, which would allow us to join the transposed DataFrames together by variable and analyze the summary statistics, a practice that most data scientists know as “pivoting” or “reshaping” a data set. In the next section, we’ll show you how to perform these transforms.

Pivoting and Reshaping DataFrames

We can transpose the DataFrames entirely using functions provided by PySpark. However, there is another way to perform this task. PySpark allows conversion between Spark and Pandas DataFrames. We will convert the DataFrames in question into Pandas DataFrames, reshape them and convert them back to Spark DataFrames. We can safely do this because of the small size of the `summary`, `matchSummary` and `missSummary` DataFrames since Pandas DataFrames reside in memory. In upcoming chapters, we will rely on Spark operations for such transformations on larger datasets.

Conversion to/from Pandas DataFrames is possible because of the Apache Arrow project, which allows efficient data transfer between JVM and Python processes. The PyArrow library was installed as a dependency of the Spark SQL module when we installed `pyspark[sql]` using `pip`.

Let's convert `summary` into a Pandas DataFrame:

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

We can now use Pandas functions on the `summary_p` DataFrame.

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

We can now perform a transpose operation to swap rows and columns using familiar Pandas methods on the DataFrame.

```
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)
```

We have successfully transposed the `summary_p` Pandas DataFrame. Convert it into a Spark DataFrame using SparkSession's `createDataFrame` method:

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

We are not done yet. Print the schema of the `summaryT` DataFrame.

```
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)
```

In the summary schema, as obtained from the `describe()` method, every field is treated as a string. Since we want to analyze the summary statistics as numbers, we'll need to convert the values from strings to doubles.

```
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)
```

Now that we have figured out how to transpose a summary DataFrame, let's implement our logic into a function that we can reuse on the `matchSummary` and `missSummary` DataFrames.

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

def pivotSummary(desc: DataFrame) -> DataFrame:
    # convert to Pandas dataframe
    desc_p = desc.toPandas()
    # transpose
    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)
    # convert to Spark dataframe
    descT = spark.createDataFrame(desc_p)
    # convert metric columns to double from string
    for c in descT.columns:
        if c == 'field':
            continue
        else:
            descT = descT.withColumn(c, descT[c].cast(DoubleType()))
    return descT
```

Now in your Spark shell, use the `pivotSummary` function on the `matchSummary` and `missSummary` DataFrames:

```
matchSummaryT = pivotSummary(matchSummary)
missSummaryT = pivotSummary(missSummary)
```

Now that we have successfully transposed the summary DataFrames, we can join and compare them. That's what we will do in the next section. Further, we will also select desirable features for building our model.

Joining DataFrames and Selecting Features

So far, we have only used Spark SQL and the DataFrame API to filter and aggregate the records from a data set, but we can also use these tools in order to perform joins (inner, left outer, right outer, or full outer) on DataFrames as well. Although the DataFrame API includes a `join` function, it's often easier to express these joins using Spark SQL, especially when the tables we are joining have a large number of column names in common and we want to be able to clearly indicate which column we are referring to in our select expressions. Let's create temporary views for the `matchSummaryT` and `missSummaryT` DataFrames, join them on the `field` column, and compute some simple summary statistics on the resulting rows:

```
matchSummaryT.createOrReplaceTempView("match_desc")
missSummaryT.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	0.032408185250332844
+-----+-----+-----+-----+		

A good feature has two properties: it tends to have significantly different values for matches and nonmatches (so the difference between the means will be large) and it occurs often enough in the data that we can rely on it to be regularly available for any pair of records. By this measure, `cmp_fname_c2` isn't very useful because it's missing a lot of the time and the difference in the mean value for matches and nonmatches is relatively small—0.09, for a score that ranges from 0 to 1. The `cmp_sex` feature also isn't particularly helpful because even though it's available for any pair of records, the difference in means is just 0.03.

Features `cmp_plz` and `cmp_by`, on the other hand, are excellent. They almost always occur for any pair of records, and there is a very large difference in the mean values (more than 0.77 for both features.) Features `cmp_bd`, `cmp_lname_c1`, and `cmp_bm` also seem beneficial: they are generally available in the data set and the difference, in mean values for matches and nonmatches are substantial.

Features `cmp_fname_c1` and `cmp_lname_c2` are more of a mixed bag: `cmp_fname_c1` doesn't discriminate all that well (the difference in the means is only 0.28) even though it's usually available for a pair of records, whereas `cmp_lname_c2` has a large difference in the means but it's almost always missing. It's not quite obvious under what circumstances we should include these features in our model based on this data.

For now, we're going to use a simple scoring model that ranks the similarity of pairs of records based on the sums of the values of the obviously good features: `cmp_plz`, `cmp_by`, `cmp_bd`, `cmp_lname_c1`, and `cmp_bm`. For the few records where the values of these features are missing, we'll use 0 in place of the `null` value in our sum. We can get a rough feel for the performance of our simple model by creating a data frame of the computed scores and the value of the `is_match` column and evaluating how well the score discriminates between matches and nonmatches at various thresholds.

Scoring And Model Evaluation

For our scoring function, we are going to sum up the value of five fields (`cmp_lname_c1`, `cmp_plz`, `cmp_by`, `cmp_bd`, and `cmp_bm`). We will use `expr` from `pyspark.sql.functions` for doing this. The `expr` function parses an input expression string into the column that it represents. This string can even involve multiple columns.

Let's create the required expression string:

```
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'
```

We can now use the `sum_expression` string for calculating the score. When summing up the values, we will account for and replace null values with 0 using `DataFrame`'s `fillna` method.

```
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|
...

```

The final step in creating our scoring function is to decide on what threshold the score must exceed in order for us to predict that the two records represent a match. If we set the threshold too high, then we will incorrectly mark a matching record as a miss (called the *false-negative* rate), whereas if we set the threshold too low, we will incorrectly label misses as matches (the *false-positive* rate.) For any nontrivial problem, we always have to trade some false positives for some

false negatives, and the question of what the threshold value should be usually comes down to the relative cost of the two kinds of errors in the situation to which the model is being applied.

To help us choose a threshold, it's helpful to create a *contingency table* (which is sometimes called a *cross tabulation*, or *crosstab*) that counts the number of records whose scores fall above/below the threshold value crossed with the number of records in each of those categories that were/were not matches. Since we don't know what threshold value we're going to use yet, let's write a function that takes the `scored` DataFrame and the choice of threshold as parameters and computes the crosstabs using the DataFrame API:

```
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()
```

Note that we are including the `selectExpr` method of the DataFrame API to dynamically determine the value of the field named `above` based on the value of the `t` argument using Python's f-string formatting syntax, which allows us to substitute variables by name if we preface the string literal with the letter `f` (yet another handy bit of Scala implicit magic). Once the `above` field is defined, we create the crosstab with a standard combination of the `groupBy`, `pivot`, and `count` methods that we used before.

Applying a high threshold value of 4.0, meaning that the average of the five features is 0.8, we can filter out almost all of the nonmatches while keeping over 90% of the matches:

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

Applying the lower threshold of 2.0, we can ensure that we capture *all* of the known matching records, but at a substantial cost in terms of false positive (top-right cell):

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

Even though the number of false positives is higher than we want, this more generous filter still removes 90% of the nonmatching records from our consideration while including every positive match. Even though this is pretty good, it's possible to do even better; see if you can find a way to use some of the other values from `MatchData` (both missing and not) to come up with a scoring function that successfully identifies every `true` match at the cost of less than 100 false positives.

Where to Go from Here

If this chapter was your first time carrying out data preparation and analysis with PySpark, we hope that you got a feel for what a powerful foundation these tools provide. If you have been using Python and Spark for a while, we hope that you will pass this chapter along to your friends and colleagues as a way of introducing them to that power as well.

Our goal for this chapter was to provide you with enough knowledge to be able to understand and complete the rest of the examples in this book. If you are the kind of person who learns best through practical examples, your next step is to continue on to the next set of chapters, where we will introduce you to MLlib, the machine learning library designed for Spark.

Chapter 3. Recommending Music and the Audioscrobbler Data Set

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

The recommender engine is one of the most popular example of large-scale machine learning, and most people have seen Amazon’s. It is a common denominator because recommender engines are everywhere, from social networks to video sites to online retailers. We can also directly observe them in action. We’re aware that a computer is picking tracks to play on Spotify, in much the same way we don’t necessarily notice that Gmail is deciding whether inbound email is spam.

The output of a recommender is more intuitively understandable than other machine learning algorithms. It’s exciting, even. For as much as we think that musical taste is personal and inexplicable, recommenders do a surprisingly good job of identifying tracks we didn’t know we would like. For domains like music or movies where recommenders are usually deployed, it’s comparatively easy to reason about why a recommended piece of music fits with someone’s listening history. Not all clustering or classification algorithms match that description. For example, a support vector machine classifier is a set of coefficients, and it’s hard even for

practitioners to articulate what the numbers mean when they make predictions.

It seems fitting to kick off the next three chapters, which will explore key machine learning algorithms on PySpark, with a chapter built around recommender engines, and recommending music in particular. It's an accessible way to introduce real-world use of PySpark and MLlib, and some basic machine learning ideas that will be developed in subsequent chapters.

In this chapter, we will implement a recommender system in PySpark. Specifically, we will use the Alternating Least Squares (ALS) algorithm on an open dataset provided by a music streaming service. We will start off by understanding the dataset and importing it in PySpark. We will then discuss our motivation for choosing the ALS algorithm and its implementation in PySpark. This will be followed by data preparation and building our model using PySpark. We will finish off by making some user recommendations and discussing ways to improve our model through hyperparameter selection.

Setting up the Data

We will use a data set published by Audioscrobbler. Audioscrobbler was the first music recommendation system for [last.fm](#), one of the first internet streaming radio sites, founded in 2002. Audioscrobbler provided an open API for “scrobbling,” or recording listeners’ song plays. last.fm used this information to build a powerful music recommender engine. The system reached millions of users because third-party apps and sites could provide listening data back to the recommender engine.

At that time, research on recommender engines was mostly confined to learning from rating-like data. That is, recommenders were usually viewed as tools that operated on input like “Bob rates Prince 3.5 stars.” The Audioscrobbler data set is interesting because it merely records plays: “Bob played a Prince track.” A play carries less information than a rating. Just because Bob played the track doesn't mean he actually liked it. You or I

may occasionally play a song by an artist we don't care for, or even play an album and walk out of the room.

However, listeners rate music far less frequently than they play music. A data set like this is therefore much larger, covers more users and artists, and contains more total information than a rating data set, even if each individual data point carries less information. This type of data is often called *implicit feedback* data because the user-artist connections are implied as a side effect of other actions, and not given as explicit ratings or thumbs-up.

A snapshot of a data set distributed by last.fm in 2005 can be found [online as a compressed archive](#). Download the archive, and find within it several files. First, the data set's files need to be made available. If you are using a remote cluster, copy all three data files into HDFS. This chapter will assume that the files are available at `/user/ds/`.

Start `pyspark-shell`. Note that the computations in this chapter will take up more memory than simple applications. If you are running locally rather than on a cluster, for example, you will likely need to specify something like `--driver-memory 4g` to have enough memory to complete these computations. The main data set is in the `user_artist_data.txt` file. It contains about 141,000 unique users, and 1.6 million unique artists. About 24.2 million users' plays of artists are recorded, along with their counts. Let's read this data set into a `DataFrame` and have a look at it.

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

```
...  
+-----+  
|               value|  
+-----+  
|    1000002 1 55|  
| 1000002 1000006 33|  
| 1000002 1000007 8|
```

```
|1000002 1000009 144|
|1000002 1000010 314|
+-----+
```

By default on HDFS, the data set will contain one partition for each HDFS block. Because this file consumes about 400 MB on disk, it will split into about three to six partitions given typical HDFS block sizes. Machine learning tasks like ALS are likely to be more compute-intensive than simple text processing. It may be better to break the data into smaller pieces—more partitions—for processing. You can chain a call to `.repartition(n)` after reading the text file to specify a different and larger number of partitions. You might set this higher to match the number of cores in your cluster, for example.

The data set also gives the names of each artist by ID in the *artist_data.txt* file. Note that when plays are scrobbled, the client application submits the name of the artist being played. This name could be misspelled or nonstandard, and this may only be detected later. For example, “The Smiths,” “Smiths, The,” and “the smiths” may appear as distinct artist IDs in the data set even though they are plainly the same. So, the data set also includes *artist_alias.txt*, which maps artist IDs that are known misspellings or variants to the canonical ID of that artist. Let’s read in these two data sets into PySpark too.

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

```
raw_artist_data.show(5)
```

```
...
+-----+
|          value|
+-----+
|1134999        06Crazy Life|
|6821360        Pang Nakarin|
|10113088       Terfel, ...|
|10151459       The Flam...|
|6826647        Bodenstan...|
+-----+
```

```
...
```

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

raw_artist_alias.show(5)
```

```
...
+-----+
|           value |
+-----+
| 1092764         1000311 |
| 1095122         1000557 |
| 6708070         1007267 |
| 10088054        1042317 |
| 1195917         1042317 |
+-----+
```

We now have a basic understanding of the datasets. We will now discuss our requirements from a recommender algorithm and subsequently understand why the Alternating Least Squares algorithm is a good choice.

Our requirements from a recommender system

We need to choose a recommender algorithm that is suitable for our data. Here are our considerations:

- Implicit feedback.

It is comprised entirely of interactions between users and artists' songs. It contains no information about the users, or about the artists other than their names. We need an algorithm that learns without access to user or artist attributes. These are typically called **collaborative filtering** algorithms. For example, deciding that two users might share similar tastes because they are the same age *is not* an example of collaborative filtering. Deciding that two users might both like the same song because they play many other same songs *is* an example.

- Sparsity

Our data set looks large because it contains tens of millions of play counts. But in a different sense, it is small and skimpy, because it is sparse. On average, each user has played songs from about 171 artists—out of 1.6 million. Some users have listened to only one artist. We need an algorithm that can provide decent recommendations to even these users. After all, every single listener must have started with just one play at some point!

- Scalability and real-time predictions

Finally, we need an algorithm that scales, both in its ability to build large models and to create recommendations quickly. Recommendations are typically required in near real time—within a second, not tomorrow.

A broad class of algorithms that may be suitable are **latent-factor** models. They try to explain *observed interactions* between large numbers of users and items through a relatively small number of *unobserved, underlying reasons*. For example, consider a customer who has bought albums by metal bands Megadeth and Pantera, but also classical composer Mozart. It may be difficult to explain why exactly these albums were bought and nothing else. However, it's probably a small window on a much larger set of tastes. Maybe the customer likes a coherent spectrum of music from metal to progressive rock to classical. That explanation is simpler, and as a bonus, suggests many other albums that would be of interest. In this example, “liking metal, progressive rock, and classical” are three latent factors that could explain tens of thousands of individual album preferences.

In our case, we will specifically use a type of **matrix factorization** model. Mathematically, these algorithms treat the user and product data as if it were a large matrix A , where the entry at row i and column j exists if user i has played artist j . A is sparse: most entries of A are 0, because only a few of all possible user-artist combinations actually appear in the data. They factor A as the matrix product of two smaller matrices, X and Y . They are very skinny—both have many rows because A has many rows and columns, but both have just a few columns (k). The k columns correspond to the latent factors that are being used to explain the interaction data.

The factorization can only be approximate because k is small, as shown in Figure 3-1.

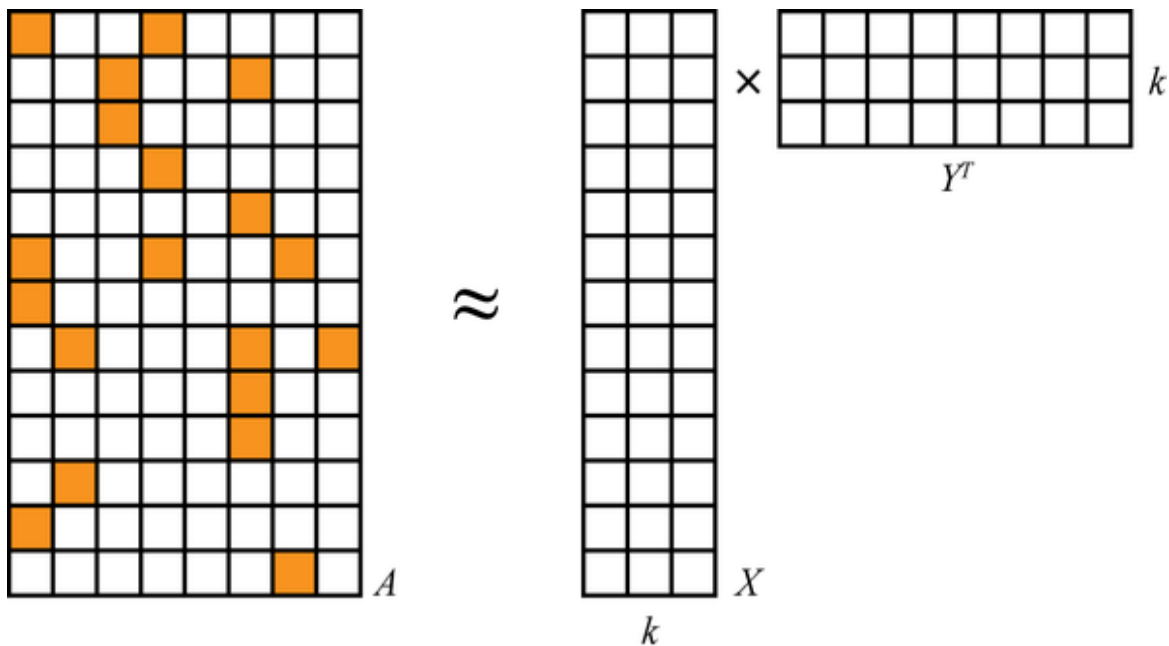


Figure 3-1. Matrix factorization

These algorithms are sometimes called matrix completion algorithms, because the original matrix A may be quite sparse, but the product XY^T is dense. Very few, if any, entries are 0, and therefore the model is only an approximation to A . It is a model in the sense that it produces (“completes”) a value for even the many entries that are missing (that is, 0) in the original A .

This is a case where, happily, the linear algebra maps directly and elegantly to intuition. These two matrices contain a row for each user and each artist, respectively. The rows have few values— k . Each value corresponds to a latent feature in the model. So the rows express how much users and artists associate with these k latent features, which might correspond to tastes or genres. And it is simply the product of a user-feature and feature-artist matrix that yields a complete estimation of the entire, dense user-artist interaction matrix. This product might be thought of as mapping items to their attributes, and then weighting those by user attributes.

The bad news is that $A = XY^T$ generally has no exact solution at all, because X and Y aren't large enough (technically speaking, too low **rank**) to perfectly represent A . This is actually a good thing. A is just a tiny sample of all interactions that *could* happen. In a way, we believe A is a terribly spotty and therefore hard-to-explain view of a simpler underlying reality that is well explained by just some small number of factors, k , of them. Think of a jigsaw puzzle depicting a cat. The final puzzle is simple to describe: a cat. When you're holding just a few pieces, however, the picture you see is quite difficult to describe.

XY^T should still be as close to A as possible. After all, it's all we've got to go on. It will not and should not reproduce it exactly. The bad news again is that this can't be solved directly for both the best X and best Y at the same time. The good news is that it's trivial to solve for the best X if Y is known, and vice versa. But neither is known beforehand!

Fortunately, there are algorithms that can escape this catch-22 and find a decent solution. One such algorithm that's available in PySpark is the ALS algorithm.

Alternating Least Squares algorithm

We will use the **Alternating Least Squares** (ALS) algorithm to compute latent factors from our dataset. This type of approach was popularized around the time of the **Netflix Prize** by papers like “**Collaborative Filtering for Implicit Feedback Datasets**” and “**Large-Scale Parallel Collaborative Filtering for the Netflix Prize**”. PySpark MLlib's ALS implementation draws on ideas from both of these papers and is the only recommender algorithm currently implemented in Spark MLlib.

Here's a code snippet (non-functional) to give you a peek at what lies ahead in the chapter.

```
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)
```

With ALS, we will treat our input data as a large sparse matrix A , and find out X and Y , as discussed in previous section. At the start, Y isn't known, but it can be initialized to a matrix full of randomly chosen row vectors. Then simple linear algebra gives the best solution for X , given A and Y . In fact, it's trivial to compute each row i of X separately as a function of Y and of one row of A . Because it can be done separately, it can be done in parallel, and that is an excellent property for large-scale computation:

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

Equality can't be achieved exactly, so in fact the goal is to minimize $|A_i Y (Y^T Y)^{-1} - X_i|$, or the sum of squared differences between the two matrices' entries. This is where the “least squares” in the name comes from. In practice, this is never solved by actually computing inverses but faster and more directly via methods like the **QR decomposition**. This equation simply elaborates on the theory of how the row vector is computed.

The same thing can be done to compute each Y_j from X . And again, to compute X from Y , and so on. This is where the “alternating” part comes from. There's just one small problem: Y was made up, and random! X was computed optimally, yes, but given a bogus solution for Y . Fortunately, if this process is repeated, X and Y do eventually converge to decent solutions.

When used to factor a matrix representing implicit data, there is a little more complexity to the ALS factorization. It is not factoring the input matrix A directly, but a matrix P of 0s and 1s, containing 1 where A contains a positive value and 0 elsewhere. The values in A are incorporated later as weights. This detail is beyond the scope of this book, but is not necessary to understand how to use the algorithm.

Finally, the ALS algorithm can take advantage of the sparsity of the input data as well. This, and its reliance on simple, optimized linear algebra and its data-parallel nature, make it very fast at large scale.

Next, we will preprocess our dataset and make it suitable for use with the ALS algorithm.

Preparing the Data

The first step in building a model is to understand the data that is available, and parse or transform it into forms that are useful for analysis in Spark.

Spark MLlib's ALS implementation does not strictly require numeric IDs for users and items, but is more efficient when the IDs are in fact representable as 32-bit integers. It's advantageous to use `Int` to represent IDs, but this would mean that the IDs can't exceed `Int.MaxValue`, or 2147483647. Does this data set conform to this requirement already?

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

Each line of the file contains a user ID, an artist ID, and a play count, separated by spaces. To compute statistics on the user ID, we split the line by space characters, and parse the values as integers. The result is conceptually three “columns”: a user ID, artist ID and count as `Int`s. It makes sense to transform this to a data frame with columns named “user”, “artist”, and “count” because it then becomes simple to compute simple statistics like the maximum and minimum:

```

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())) \
.withColumn('artist', split(raw_user_artist_data['value'], '
').getItem(1).cast(IntegerType())) \
.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|
+-----+-----+-----+-----+

```

The maximum user and artist IDs are 2443548 and 10794401, respectively (and their minimums are 90 and 1; no negative values). These are comfortably smaller than 2147483647. No additional transformation will be necessary to use these IDs.

It will be useful later in this example to know the artist names corresponding to the opaque numeric IDs. `raw_artist_data` contains the artist ID and name separated by a tab. PySpark's `split` function accepts **regular expression** values for the `pattern` parameter. We can split using the whitespace character, `\s`:

```

from pyspark.sql.functions import col

artist_by_id = raw_artist_data.withColumn('id',
split(col('value'), '\s+',
2).getItem(0).cast(IntegerType())).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|
+-----+-----+
```

This gives a data frame with the artist ID and name as columns “id” and “name”.

`raw_artist_alias` maps artist IDs that may be misspelled or nonstandard to the ID of the artist’s canonical name. This dataset is relatively small, containing about 200,000 entries. It contains two IDs per line, separated by a tab. We will parse this in a similar manner as we did `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|
+-----+-----+
```

The first entry maps ID 1092764 to 1000311. We can look these up from `artist_by_id` DataFrame:

```
artist_by_id.filter(artist_by_id.id.isin(1092764,
1000311)).show()
```

```
...
```

```
+-----+-----+
```

id	name
1000311	Steve Winwood
1092764	Winwood, Steve

This entry evidently maps “Winwood, Steve” to “Steve Winwood,” which is in fact the correct name for the artist.

Building a First Model

Although the data set is in nearly the right form for use with Spark MLlib’s ALS implementation, it requires a small extra transformation. The aliases data set should be applied to convert all artist IDs to a canonical ID, if a different canonical ID exists.

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

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

train_data.cache()
```

❶ Get artist’s alias if it exists, otherwise get original artist.

We broadcast the `artist_alias` DataFrame created earlier.. This makes Spark send and hold in memory just one copy for *each executor* in the cluster. When there are thousands tasks and many execute in parallel on each executor, this can save significant network traffic and memory. As a rule of thumb, its helpful to broadcast a significantly smaller dataset when performing a join with a very big dataset.

BROADCAST VARIABLES

When Spark runs a stage, it creates a binary representation of all the information needed to run tasks in that stage; this is called the *closure* of the function that needs to be executed. This closure includes all the data structures on the driver referenced in the function. Spark distributes it with every task that is sent to an executor on the cluster.

Broadcast variables are useful when many tasks need access to the same (immutable) data structure. They extend normal handling of task closures to enable:

- Caching data as raw Java objects on each executor, so they need not be deserialized for each task
- Caching data across multiple jobs, stages, and tasks

For example, consider a natural language processing application that requires a large dictionary of English words, and has a `score` function that accepts a line of input and dictionary of words. Broadcasting the dictionary means it is transferred to each executor only once:

`DataFrame` operations can at times also automatically take advantage of broadcasts when performing joins between a large and small table. Just broadcasting the small table is advantageous sometimes. This is called a *broadcast hash join*.

The call to `cache()` suggests to Spark that this `DataFrame` should be temporarily stored after being computed, and furthermore, kept in memory in the cluster. This is helpful because the ALS algorithm is iterative, and will typically need to access this data 10 times or more. Without this, the `DataFrame` could be repeatedly recomputed from the original data each time it is accessed! The Storage tab in the Spark UI will show how much of the `DataFrame` is cached and how much memory it uses, as shown in [Figure 3-2](#). This one consumes about 120 MB across the cluster.

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

Figure 3-2. Storage tab in the Spark UI, showing cached DataFrame memory usage

Note that the label “Deserialized” in the UI above is actually only relevant for RDDs, where “Serialized” means data are stored in memory not as objects, but as serialized bytes. However, `DataFrame` instances like this one perform their own “encoding” of common data types in memory separately.

Actually, 120 MB is surprisingly small. Given that there are about 24 million plays stored here, a quick back-of-the-envelope calculation suggests that this would mean that each user-artist-count entry consumes only 5 bytes on average. However, the three 32-bit integers alone ought to consume 12 bytes. This is one of the advantages of a `DataFrame`. Because the types of data stored are primitive 32-bit integers, their representation can be optimized in memory internally.

SPARK UI

Spark UI is the web interface of a running Spark application to monitor the status and resource consumption of your Spark cluster. By default, it is available at [http://\[driver\]:4040](http://[driver]:4040).

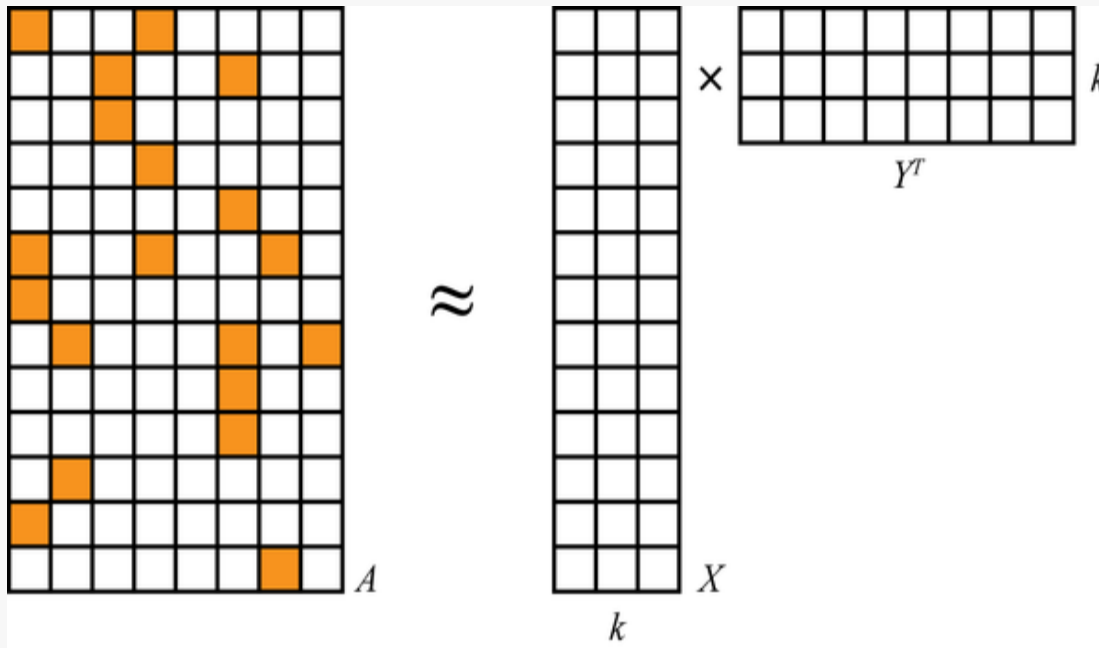


Figure 3-3. Spark UI

It comes with the following tabs:

- Jobs
- Stages
- Storage with RDD size and memory use
- Environment
- Executors
- SQL

Some tabs, such as the ones related to streaming, are created lazily, i.e. when required.

Finally, we can build a model:

```
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)
```

This constructs `model` as an `ALSModel` with some default configuration. The operation will likely take minutes or more depending on your cluster. Compared to some machine learning models, whose final form may consist of just a few parameters or coefficients, this type of model is huge. It contains a feature vector of 10 values for each user and product in the model, and in this case there are more than 1.7 million of them. The model contains these large user-feature and product-feature matrices as `DataFrames` of their own.

The values in your results may be somewhat different. The final model depends on a randomly chosen initial set of feature vectors. The default behavior of this and other components in `MLlib`, however, is to use the same set of random choices every time by defaulting to a fixed seed. This is unlike other libraries, where behavior of random elements is typically not fixed by default. So, here and elsewhere, a random seed is set with (`... seed=0, ...`).

To see some feature vectors, try the following, which displays just one row and does not truncate the wide display of the feature vector:

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

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

The other methods invoked on ALS, like `setAlpha`, set *hyperparameters* whose value can affect the quality of the recommendations that the model makes. These will be explained later. The more important first question is, is the model any good? Does it produce good recommendations? That is what we will try to answer in the next section.

Spot Checking Recommendations

We should first see if the artist recommendations make any intuitive sense, by examining a user, plays, and recommendations for that user. Take, for example, user 2093760. First, let's look at his or her plays to get a sense of the person's tastes. Extract the IDs of artists that this user has listened to and print their names. This means searching the input for artist IDs played by this user, and then filtering the set of artists by these IDs in order to print the names in order:

```
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 Find lines whose user is 2093760.
- ❷ Collect data set of artist ID.
- ❸ Filter in those artists.

The artists look like a mix of mainstream pop and hip-hop. A Jurassic 5 fan? Remember, it's 2005. In case you're wondering, the Saw Doctors are a very Irish rock band popular in Ireland.

Now, it's simple to make recommendations for a user, though computing them this way will take a few moments. It's suitable for batch scoring but not real-time use cases:

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

The resulting recommendations contain lists comprised of artist ID and of course, “predictions.” For this type of ALS algorithm, the prediction is an opaque value normally between 0 and 1, where higher values mean a better recommendation. It is not a probability, but can be thought of as an estimate of a 0/1 value indicating whether the user won't or will interact with the artist, respectively.

After extracting the artist IDs for the recommendations, we can look up artist names in a similar way:

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

The result is all hip-hop. This doesn't look like a great set of recommendations, at first glance. While these are generally popular artists, they don't appear to be personalized to this user's listening habits.

Evaluating Recommendation Quality

Of course, that's just one subjective judgment about one user's results. It's hard for anyone but that user to quantify how good the recommendations are. Moreover, it's infeasible to have any human manually score even a small sample of the output to evaluate the results.

It's reasonable to assume that users tend to play songs from artists who are appealing, and not play songs from artists who aren't appealing. So, the plays for a user give a partial picture of "good" and "bad" artist recommendations. This is a problematic assumption but about the best that can be done without any other data. For example, presumably user 2093760 likes many more artists than the 5 listed previously, and among the 1.7 million other artists not played, a few are of interest and not all are "bad" recommendations.

What if a recommender were evaluated on its ability to rank good artists high in a list of recommendations? This is one of several generic metrics that can be applied to a system that ranks things, like a recommender. The problem is that "good" is defined as "artists the user has listened to," and the recommender system has already received all of this information as input. It could trivially return the user's previously listened-to artists as top recommendations and score perfectly. But this is not useful, especially

because the recommender's role is to recommend artists that the user has never listened to.

To make this meaningful, some of the artist play data can be set aside and hidden from the ALS model-building process. Then, this held-out data can be interpreted as a collection of good recommendations for each user but one that the recommender has not already been given. The recommender is asked to rank all items in the model, and the ranks of the held-out artists are examined. Ideally, the recommender places all of them at or near the top of the list.

We can then compute the recommender's score by comparing all held-out artists' ranks to the rest. (In practice, we compute this by examining only a sample of all such pairs, because a potentially huge number of such pairs may exist.) The fraction of pairs where the held-out artist is ranked higher is its score. A score of 1.0 is perfect, 0.0 is the worst possible score, and 0.5 is the expected value achieved from randomly ranking artists.

This metric is directly related to an information retrieval concept called the **receiver operating characteristic** (ROC) curve. The metric in the preceding paragraph equals the area under this ROC curve, and is indeed known as AUC, or Area Under the Curve. AUC may be viewed as the probability that a randomly chosen good recommendation ranks above a randomly chosen bad recommendation.

The AUC metric is also used in the evaluation of classifiers. It is implemented, along with related methods, in the MLlib class `BinaryClassificationMetrics`. For recommenders, we will compute AUC *per user* and average the result. The resulting metric is slightly different, and might be called "mean AUC." We will implement this, because it is not (quite) implemented in PySpark.

Other evaluation metrics that are relevant to systems that rank things are implemented in `RankingMetrics`. These include metrics like precision, recall, and **mean average precision** (MAP). MAP is also frequently used and focuses more narrowly on the quality of the top recommendations.

However, AUC will be used here as a common and broad measure of the quality of the entire model output.

In fact, the process of holding out some data to select a model and evaluate its accuracy is common practice in all of machine learning. Typically, data is divided into three subsets: training, cross-validation (CV), and test sets. For simplicity in this initial example, only two sets will be used: training and CV. This will be sufficient to choose a model. In Chapter 4, this idea will be extended to include the test set.

Computing AUC

An implementation of mean AUC is provided in the source code accompanying this book. It is not reproduced here, but is explained in some detail in comments in the source code. It accepts the CV set as the “positive” or “good” artists for each user, and a prediction function. This function translates a data frame containing each user-artist pair into a data frame that also contains its estimated strength of interaction as a “prediction,” a number wherein higher values mean higher rank in the recommendations.

In order to use it, we must split the input data into a training and CV set. The ALS model will be trained on the training data set only, and the CV set will be used to evaluate the model. Here, 90% of the data is used for training and the remaining 10% for cross-validation:

```
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)

```

Note that `areaUnderCurve()` accepts a *function* as its third argument. Here, the `transform` method from `ALSModel` is passed in, but it will shortly be swapped out for an alternative.

The result is about 0.879. Is this good? It is certainly higher than the 0.5 that is expected from making recommendations randomly, and it's close to 1.0, which is the maximum possible score. Generally, an AUC over 0.9 would be considered high.

But is it an accurate evaluation? This evaluation could be repeated with a different 90% as the training set. The resulting AUC values' average might be a better estimate of the algorithm's performance on the data set. In fact, one common practice is to divide the data into k subsets of similar size, use $k - 1$ subsets together for training, and evaluate on the remaining subset. We can repeat this k times, using a different set of subsets each time. This is called *k-fold cross-validation*. This won't be implemented in examples here, for simplicity, but some support for this technique exists in MLlib in its `CrossValidator` API. The validation API will be revisited in “Random Forests”.

It's helpful to benchmark this against a simpler approach. For example, consider recommending the globally most-played artists to every user. This is not personalized, but it is simple and may be effective. Define this simple prediction function and evaluate its AUC score:

```

from pyspark.sql.functions import sum as _sum

def predict_most_listened(train):
    listen_counts =
train.groupBy("artist").agg(_sum("count").alias("prediction")).se
lect("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))

```

The result is also about 0.880. This suggests that nonpersonalized recommendations are already fairly effective according to this metric. However, we’d expect the “personalized” recommendations to score better in comparison. Clearly, the model needs some tuning. Can it be made better?

Hyperparameter Selection

So far, the hyperparameter values used to build the `ALSModel` were simply given without comment. They are not learned by the algorithm and must be chosen by the caller. The configured hyperparameters were:

setRank(10)

The number of latent factors in the model, or equivalently, the number of columns k in the user-feature and product-feature matrices. In nontrivial cases, this is also their rank.

setMaxIter(5)

The number of iterations that the factorization runs. More iterations take more time but may produce a better factorization.

setRegParam(0.01)

A standard overfitting parameter, also usually called *lambda*. Higher values resist overfitting, but values that are too high hurt the

factorization's accuracy.

`setAlpha(1.0)`

Controls the relative weight of observed versus unobserved user-product interactions in the factorization.

`rank`, `regParam`, and `alpha` can be considered *hyperparameters* to the model. (`maxIter` is more of a constraint on resources used in the factorization.) These are not values that end up in the matrices inside the `ALSModel`—those are simply its *parameters* and are chosen by the algorithm. These hyperparameters are instead parameters to the process of building itself.

The values used in the preceding list are not necessarily optimal. Choosing good hyperparameter values is a common problem in machine learning. The most basic way to choose values is to simply try combinations of values and evaluate a metric for each of them, and choose the combination that produces the best value of the metric.

In the following example, eight possible combinations are tried: `rank = 5` or `30`, `regParam = 4.0` or `0.0001`, and `alpha = 1.0` or `40.0`. These values are still something of a guess, but are chosen to cover a broad range of parameter values. The results are printed in order by top AUC score:

```
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").set
ItemCol("artist").setRatingCol("count").setPredictionCol("predict
ion").\
    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 Free up model resources immediately.
- ❷ Sort by first value (AUC), descending, and print.

The differences are small in absolute terms, but are still somewhat significant for AUC values. Interestingly, the parameter `alpha` seems consistently better at 40 than 1. (For the curious, 40 was a value proposed as a default in one of the original ALS papers mentioned earlier.) This can be interpreted as indicating that the model is better off focusing far more on what the user did listen to than what he or she did not listen to.

A higher `regParam` looks better too. This suggests the model is somewhat susceptible to overfitting, and so needs a higher `regParam` to resist trying to fit the sparse input given from each user too exactly. Overfitting will be revisited in more detail in “Random Forests”.

As expected, 5 features is pretty low for a model of this size, and underperforms the model that uses 30 features to explain tastes. It's possible that the best number of features is actually higher than 30, and that these values are alike in being too small.

Of course, this process can be repeated for different ranges of values or more values. It is a brute-force means of choosing hyperparameters. However, in a world where clusters with terabytes of memory and hundreds of cores are not uncommon, and with frameworks like Spark that can exploit parallelism and memory for speed, it becomes quite feasible.

It is not strictly required to understand what the hyperparameters mean, although it is helpful to know what normal ranges of values are like in order to start the search over a parameter space that is neither too large nor too tiny.

This was a fairly manual way to loop over hyperparameters, build models, and evaluate them. In Chapter 4, after learning more about the Spark ML API, we'll find that there is a more automated way to compute this using `Pipelines` and `TrainValidationSplit`.

Making Recommendations

Proceeding for the moment with the best set of hyperparameters, what does the new model recommend for user 2093760?

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

Anecdotally, this makes a bit more sense for this user, being dominated by pop rock instead of all hip-hop. `[unknown]` is plainly not an artist.

Querying the original data set reveals that it occurs 429,447 times, putting it nearly in the top 100! This is some default value for plays without an artist, maybe supplied by a certain scrobbling client. It is not useful information and we should discard it from the input before starting again. It is an example of how the practice of data science is often iterative, with discoveries about the data occurring at every stage.

This model can be used to make recommendations for all users. This could be useful in a batch process that recomputes a model and recommendations for users every hour or even less, depending on the size of the data and speed of the cluster.

At the moment, however, Spark MLlib's ALS implementation does not support a method to recommend to all users. It is possible to recommend to one user at a time, as shown above, although each will launch a short-lived distributed job that takes a few seconds. This may be suitable for rapidly recomputing recommendations for small groups of users. Here, recommendations are made to 100 users taken from the data and printed:

```
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 Subset of 100 distinct users.

Here, the recommendations are just printed. They could just as easily be written to an external store like **HBase**, which provides fast lookup at runtime.

Where to Go from Here

Naturally, it's possible to spend more time tuning the model parameters, and finding and fixing anomalies in the input, like the [unknown] artist. For example, a quick analysis of play counts reveals that user 2064012 played artist 4468 an astonishing 439,771 times! Artist 4468 is the implausibly successful alterna-metal band **System of a Down**, who turned up earlier in recommendations. Assuming an average song length of 4 minutes, this is over 33 years of playing hits like “Chop Suey!” and “B.Y.O.B.” Because the band started making records in 1998, this would require playing four or five tracks at once for seven years. It must be spam or a data error, and another example of the types of real-world data problems that a production system would have to address.

ALS is not the only possible recommender algorithm, but at this time, it is the only one supported by Spark MLlib. However, MLlib also supports a variant of ALS for non-implicit data. Its use is identical, except that ALS is configured with `setImplicitPrefs(false)`. This is appropriate when data is rating-like, rather than count-like. For example, it is appropriate when the data set is user ratings of artists on a 1–5 scale. The resulting `prediction` column returned from `ALSModel.transform` recommendation methods then really is an estimated rating. In this case, the simple RMSE (root mean squared error) metric is appropriate for evaluating the recommender.

Later, other recommender algorithms may be available in Spark MLlib or other libraries.

In production, recommender engines often need to make recommendations in real time, because they are used in contexts like ecommerce sites where recommendations are requested frequently as customers browse product pages. Precomputing and storing recommendations in a NoSQL store, as mentioned previously, is a reasonable way to make recommendations available at scale. One disadvantage of this approach is that it requires precomputing recommendations for all users who might need recommendations soon, which is potentially any of them. For example, if

only 10,000 of 1 million users visit a site in a day, precomputing all million users' recommendations each day is 99% wasted effort.

It would be nicer to compute recommendations on the fly, as needed. While we can compute recommendations for one user using the `ALSModel`, this is necessarily a distributed operation that takes several seconds, because `ALSModel` is uniquely large and therefore actually a distributed data set. This is not true of other models, which afford much faster scoring. Projects like [Oryx 2](#) attempt to implement real-time on-demand recommendations with libraries like MLlib by efficiently accessing the model data in memory.

Chapter 4. Predicting Forest Cover with Decision Trees

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

Classification and regression are the oldest and most well-studied types of predictive analytics. Most algorithms you will likely encounter in analytics packages and libraries are classification or regression techniques, like support vector machines, logistic regression, neural networks, and deep learning. The common thread linking regression and classification is that both involve predicting one (or more) values given one (or more) other values. To do so, both require a body of inputs and outputs to learn from. They need to be fed both questions and known answers. For this reason, they are known as types of *supervised learning*.

PySpark MLlib offers implementations of a number of classification and regression algorithms. These include **decision trees**, **naïve bayes**, **logistic regression**, and **linear regression**. The exciting thing about these algorithms is that, with all due respect to Mr. Bohr, they can help predict the future—or at least, predict the things we don’t yet know for sure, like the likelihood you will buy a car based on your online behavior, whether an email is spam given its words, or which acres of land are likely to grow the most crops given their location and soil chemistry.

In this chapter, we will focus on a popular and flexible type of algorithm for both classification and regression: decision trees, and the algorithm’s extension, random decision forests. First, we will understand the basics of Decision

Trees and Forests, and introduce the former's PySpark implementation. This will be followed by preparation of our dataset, and creating our first decision tree. Then we will tune our decision tree model. We will finish off by training a random forest model on our processed dataset and making predictions.

Although PySpark's decision tree implementation is easy to get started with, it is helpful to understand the fundamentals of Decision Tree and Random Forest algorithms. That is what we will cover in the next section.

REGRESSION TO THE MEAN

In the late nineteenth century, the English scientist Sir Francis Galton was busy measuring things like peas and people. He found that large peas (and people) had larger-than-average offspring. This isn't surprising. However, the offspring were, on average, smaller than their parents. In terms of people: the child of a seven-foot-tall basketball player is likely to be taller than the global average but still more likely than not to be less than seven feet tall.

As almost a side effect of his study, Galton plotted child versus parent size and noticed there was a roughly linear relationship between the two. Large parent peas had large children, but slightly smaller than themselves; small parents had small children, but generally a bit larger than themselves. The line's slope was therefore positive but less than 1, and Galton described this phenomenon as we do today, as *regression to the mean*.

Although maybe not perceived this way at the time, this line was, to me, an early example of a predictive model. The line links the two values, and implies that the value of one suggests a lot about the value of the other. Given the size of a new pea, this relationship could lead to a more accurate estimate of its offspring's size than simply assuming the offspring would be like the parent or like every other pea.

More than a century of statistics later, and since the advent of modern machine learning and data science, we still talk about the idea of predicting a value from other values as **regression**, even though it has nothing to do with slipping back toward a mean value, or indeed moving backward at all. Regression techniques also relate to **classification** techniques. Generally, *regression* refers to predicting a numeric quantity like size or income or temperature, whereas *classification* refers to predicting a label or category, like "spam" or "picture of a cat."

Decision Trees and Forests

Decision trees are a family of algorithms that can naturally handle both categorical and numeric features. Building a single tree can be done in parallel, and many trees can be built in parallel at once. They are robust to outliers in the data, meaning that a few extreme and possibly erroneous data points might not affect predictions at all. They can consume data of different types and on different scales without the need for preprocessing or normalization, which is an issue that will reappear in chapter 7.

The PySpark implementation of decision trees supports binary and multiclass classification, and regression. The implementation partitions data by rows, allowing distributed training with millions or even billions of instances. Decision tree–based algorithms have the further advantage of being comparatively intuitive to understand and reason about. In fact, we all probably use the same reasoning embodied in decision trees, implicitly, in everyday life. For example, I sit down to have morning coffee with milk. Before I commit to that milk and add it to my brew, I want to predict: is the milk spoiled? I don't know for sure. I might check if the use-by date has passed. If not, I predict **no**, it's not spoiled. If the date has passed, but that was three or fewer days ago, I take my chances and predict **no**, it's not spoiled. Otherwise, I sniff the milk. If it smells funny, I predict **yes**, and otherwise **no**.

This series of yes/no decisions that lead to a prediction are what decision trees embody. Each decision leads to one of two results, which is either a prediction or another decision, as shown in **Figure 4-1**. In this sense, it is natural to think of the process as a tree of decisions, where each internal node in the tree is a decision, and each leaf node is a final answer.

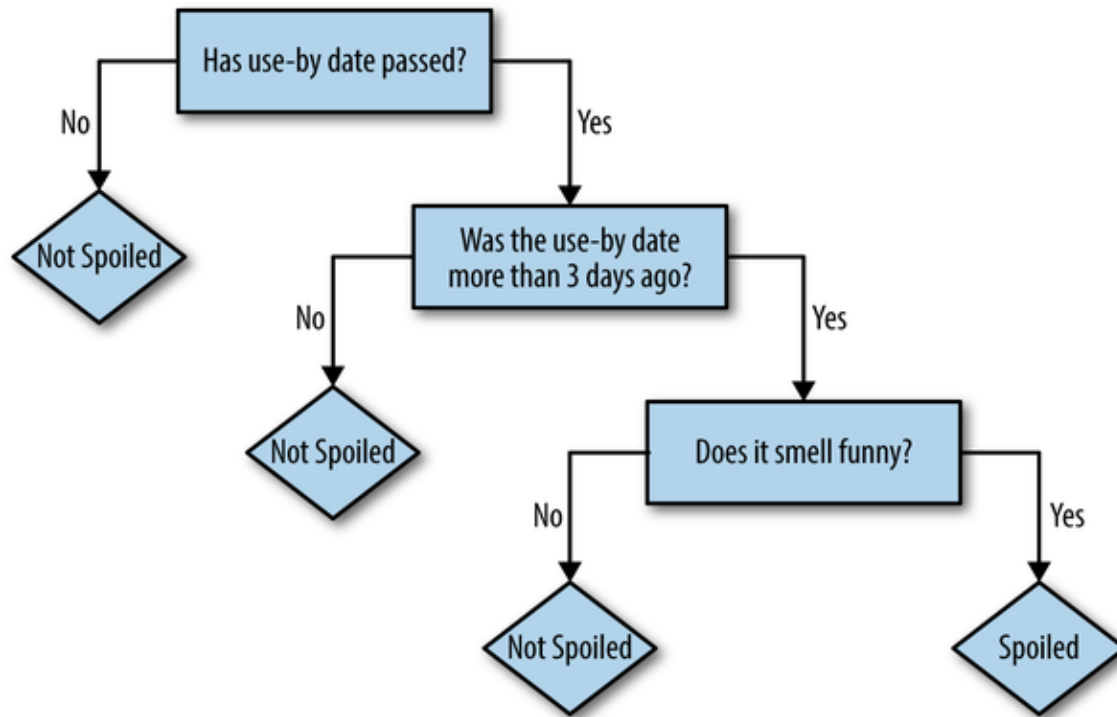


Figure 4-1. Decision tree: is it spoiled?

That is a simplistic decision tree, and was not built with any rigor. To elaborate, consider another example. A robot has taken a job in an exotic pet store. It wants to learn, before the shop opens, which animals in the shop would make a good pet for a child. The owner lists nine pets that would and wouldn't be suitable before hurrying off. The robot compiles the information found in [Table 4-1](#) from examining the animals

*T
a
b
l
e*

*4
-
1
.
E
x
o
t
i
c*

*p
e
t
s
t
o
r
e*

*“
f
e
a
t
u
r
e*

v

e
c
t
o
r
s
”

Name	Weight (kg)	# Legs	Color	Good pet?
Fido	20.5	4	Brown	Yes
Mr. Slither	3.1	0	Green	No
Nemo	0.2	0	Tan	Yes
Dumbo	1390.8	4	Gray	No
Kitty	12.1	4	Gray	Yes
Jim	150.9	2	Tan	No
Millie	0.1	100	Brown	No
McPigeon	1.0	2	Gray	No
Spot	10.0	4	Brown	Yes

Although a name is given, it will not be included as a feature. There is little reason to believe the name alone is predictive; “Felix” could name a cat or a poisonous tarantula, for all the robot knows. So, there are two numeric features (weight, number of legs) and one categorical feature (color) predicting a categorical target (is/is not a good pet for a child).

The robot might try to fit a simple decision tree to this training data to start, consisting of a single decision based on weight, as shown in [Figure 4-2](#).

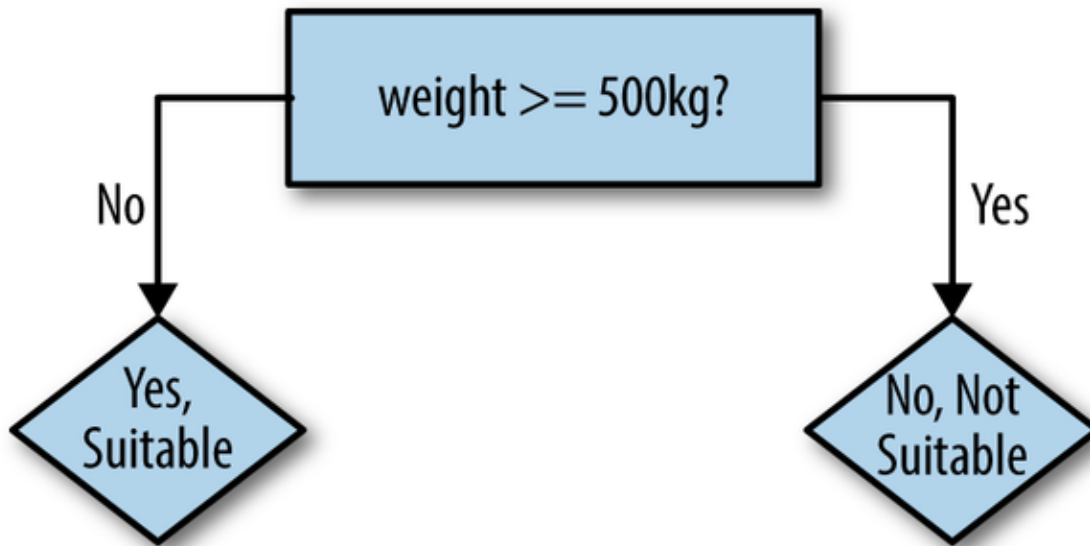


Figure 4-2. Robot's first decision tree

The logic of the decision tree is easy to read and make some sense of: 500kg animals certainly sound unsuitable as pets. This rule predicts the correct value in five of nine cases. A quick glance suggests that we could improve the rule by lowering the weight threshold to 100kg. This gets six of nine examples correct. The heavy animals are now predicted correctly; the lighter animals are only partly correct.

So, a second decision can be constructed to further refine the prediction for examples with weights less than 100kg. It would be good to pick a feature that changes some of the incorrect YES predictions to NO. For example, there is one small green animal, sounding suspiciously like a snake, that the robot could predict correctly by deciding on color, as shown in [Figure 4-3](#).

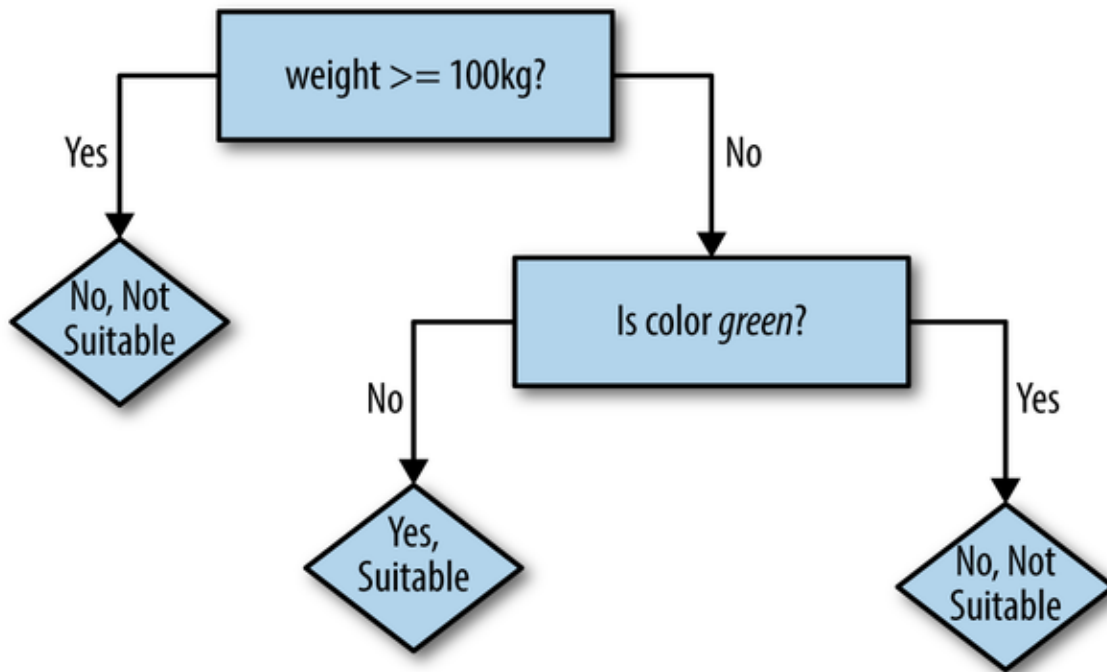


Figure 4-3. Robot's next decision tree

Now, seven of nine examples are correct. Of course, decision rules could be added until all nine were correctly predicted. The logic embodied in the resulting decision tree would probably sound implausible when translated into common speech: “If the animal’s weight is less than 100kg, and its color is brown instead of green, and it has fewer than 10 legs, then yes it is a suitable pet.” While perfectly fitting the given examples, a decision tree like this would fail to predict that a small, brown, four-legged wolverine is not a suitable pet. Some balance is needed to avoid this phenomenon, known as *overfitting*.

Decision trees generalize into a more powerful algorithm, called *random forests*. Random forests combine many decision trees in order to reduce the risk of overfitting and train the decision trees separately. The algorithm injects randomness into the training process so that each decision tree is a bit different. Combining the predictions reduces the variance of the predictions, makes the resulting model more generalizable and improves performance on test data.

This is enough of an introduction to decision trees and random forests for us to begin using them with PySpark. In the next section, we will introduce the dataset that we’ll work with and prepare it for use in PySpark.

Preparing the Data

The data set used in this chapter is the well-known Covtype data set, available [online](#) as a compressed CSV-format data file, *covtype.data.gz*, and accompanying info file, *covtype.info*.

The data set records the types of forest-covering parcels of land in Colorado, USA. It's only a coincidence that the data set concerns real-world forests! Each example contains several features describing each parcel of land—like its elevation, slope, distance to water, shade, and soil type—along with the known forest type covering the land. The forest cover type is to be predicted from the rest of the features, of which there are 54 in total.

This data set has been used in research and even a [Kaggle competition](#). It is an interesting data set to explore in this chapter because it contains both categorical and numeric features. There are 581,012 examples in the data set, which does not exactly qualify as big data but is large enough to be manageable as an example and still highlight some issues of scale.

Thankfully, the data is already in a simple CSV format and does not require much cleansing or other preparation to be used with PySpark MLlib. Later, it will be of interest to explore some transformations of the data, but it can be used as is to start.

The *covtype.data* file should be extracted and copied into HDFS. Start `pyspark-shell`. You may find it helpful to give the shell a healthy amount of memory to work with, as building decision forests can be resource-intensive. If you have the memory, specify `--driver-memory 8g` or similar.

CSV files contain fundamentally tabular data, organized into rows of columns. Sometimes these columns are given names in a header line, although that's not the case here. The column names are given in the companion file, *covtype.info*. Conceptually, each column of a CSV file has a type as well—a number, a string—but a CSV file doesn't specify this.

It's natural to parse this data as a data frame because this is PySpark's abstraction for tabular data, with a defined column schema, including column names and types. PySpark has built-in support for reading CSV data, in fact:

```

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)
...

```

This code reads the input as CSV and does not attempt to parse the first line as a header of column names. It also requests that the type of each column be inferred by examining the data. It correctly infers that all of the columns are numbers, and more specifically, integers. Unfortunately it can only name the columns “_c0” and so on.

Looking at the column names, it’s clear that some features are indeed numeric. “Elevation” is an elevation in meters; “Slope” is measured in degrees. However, “Wilderness_Area” is something different, because it is said to span four columns, each of which is a 0 or 1. In reality, “Wilderness_Area” is a categorical value, not a numeric one.

These four columns are actually a **one-hot** or 1-of-n encoding, in which one categorical feature that takes on N distinct values becomes N numeric features, each taking on the value 0 or 1. Exactly one of the N values has value 1, and the others are 0. For example, a categorical feature for weather that can be `cloudy`, `rainy`, or `clear` would become three numeric features, where `cloudy` is represented by `1, 0, 0`; `rainy` by `0, 1, 0`; and so on. These three numeric features might be thought of as `is_cloudy`, `is_rainy`, and `is_clear` features. Likewise, 40 of the columns are really one `Soil_Type` categorical feature.

This isn’t the only possible way to encode a categorical feature as a number. Another possible encoding simply assigns a distinct numeric value to each possible value of the categorical feature. For example, `cloudy` may become 1.0, `rainy` 2.0, and so on. The target itself, “Cover_Type”, is a categorical value encoded as a value 1 to 7.

Be careful when encoding a categorical feature as a single numeric feature. The original categorical values have no ordering, but when encoded as a number, they appear to. Treating the encoded feature as numeric leads to meaningless results because the algorithm is effectively pretending that `rainy` is somehow greater than, and two times larger than, `cloudy`. It's OK as long as the encoding's numeric value is not used as a number.

So we see both types of encodings of categorical features. It would have, perhaps, been simpler and more straightforward to not encode such features (and in two ways, no less), and instead simply include their values directly like “Rawah Wilderness Area.” This may be an artifact of history; the data set was released in 1998. For performance reasons or to match the format expected by libraries of the day, which were built more for regression problems, data sets often contain data encoded in these ways.

In any event, before proceeding, it is useful to add column names to this DataFrame in order to make it easier to work with:

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

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

data.head()
...
Row(Elevation=2596, Aspect=51, Slope=3,
Horizontal_Distance_To_Hydrology=258,
Vertical_Distance_To_Hydrology=0,
Horizontal_Distance_To_Roadways=510, Hillshade_9am=221,
Hillshade_Noon=232, Hillshade_3pm=148, ...)
```

❶ ++ concatenates collections

The wilderness- and soil-related columns are named “Wilderness_Area_0”, “Soil_Type_0”, and a bit of Python can generate these 44 names without having to type them all out. Finally, the target “Cover_Type” column is cast to a `double` value upfront, because it will actually be necessary to consume it as a `double` rather than `int` in all PySpark MLlib APIs. This will become apparent later.

You can call `data.show()` to see some rows of the data set, but the display is so wide that it will be difficult to read at all. `data.head()` displays it as a raw `Row` object, which will be more readable in this case.

Now that we’re familiar with our dataset and have processed it, we can train a decision tree model in the next section.

Our First Decision Tree

In chapter 3, we built a recommender model right away on all of the available data. This created a recommender that could be sense-checked by anyone with some knowledge of music: looking at a user’s listening habits and recommendations, we got some sense that it was producing good results. Here, that is not possible. We would have no idea how to make up a new 54-feature description of a new parcel of land in Colorado or what kind of forest cover to expect from such a parcel.

Instead, we must jump straight to holding out some data for purposes of evaluating the resulting model. Before, the AUC metric was used to assess the agreement between held-out listening data and predictions from recommendations. The principle is the same here, although the evaluation metric will be different: *accuracy*. The majority—90%—of the data will again be used for training, and later, we’ll see that a subset of this training set will be held out for cross-validation (the CV set). The other 10% held out here is actually a third subset, a proper test set.

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

The data needs a little more preparation to be used with a classifier in MLlib. The input DataFrame contains many columns, each holding one feature that could be used to predict the target column. MLlib requires all of the inputs to be collected into *one* column, whose value is a vector. This class is an abstraction for vectors in the linear algebra sense, and contains only numbers. For most intents and purposes, they work like a simple array of `double` values (floating-point numbers). Of course, some of the input features are conceptually categorical, even if they're all represented with numbers in the input.

Fortunately, the `VectorAssembler` class can do this work:

```
from pyspark.ml.feature import VectorAssembler

input_cols = colnames[:-1] ❶
vector_assembler =
VectorAssembler(inputCols=input_cols, outputCol="featureVector")

assembled_train_data = vector_assembler.transform(train_data)

assembled_train_data.select("featureVector").show(truncate = False)
...
+-----+
...
|featureVector|
...
+-----+
...
|(54,[0,1,2,3,4,5,6,7,8,9,13,15],[1863.0,37.0,17.0,120.0,18.0,90.0,2
...
|(54,[0,1,2,5,6,7,8,9,13,18],[1874.0,18.0,14.0,90.0,208.0,209.0,135.
...
|(54,[0,1,2,3,4,5,6,7,8,9,13,18],[1879.0,28.0,19.0,30.0,12.0,95.0,20
...
...
...

```

❶ Excludes the label, `Cover_Type`

Its key parameters are the columns to combine into the feature vector, and the name of the new column containing the feature vector. Here, all columns—*except*—the target, of course—are included as input features. The resulting DataFrame has a new “featureVector” column, as shown.

The output doesn't look exactly like a sequence of numbers, but that's because this shows a raw representation of the vector, represented as a `SparseVector` instance to save storage. Because most of the 54 values are 0, it only stores nonzero values and their indices. This detail won't matter in classification.

FEATURE VECTOR

Consider predicting tomorrow's high temperature given today's weather. There is nothing wrong with this idea, but "today's weather" is a casual concept that requires structuring before it can be fed into a learning algorithm. Really, it is certain *features* of today's weather that may predict tomorrow's temperature, such as high temperature, low temperature, average humidity, whether it's cloudy, rainy, or clear today, and the number of weather forecasters predicting a cold snap tomorrow.

These features are also sometimes called *dimensions*, *predictors*, or just *variables*. Each of these features can be quantified. For example, high and low temperatures are measured in degrees Celsius, humidity can be measured as a fraction between 0 and 1, and weather type can be labeled `cloudy`, `rainy`, or `clear`. The number of forecasters is, of course, an integer count. Today's weather might therefore be reduced to a list of values like `13.1, 19.0, 0.73, cloudy, 1`.

These five features together, in order, are known as a *feature vector*, and can describe any day's weather. This usage bears some resemblance to use of the term *vector* in linear algebra, except that a vector in this sense can conceptually contain nonnumeric values, and even lack some values. These features are not all of the same type. The first two features are measured in degrees Celsius, but the third is unitless, a fraction. The fourth is not a number at all, and the fifth is a number that is always a nonnegative integer.

A learning algorithm needs to train on data in order to make predictions. Feature vectors provide an organized way to describe input to a learning algorithm (here: `12.5, 15.5, 0.10, clear, 0`). The output, or *target*, of the prediction can also be thought of as a feature. Here, it is a numeric feature: `17.2`.

It's not uncommon to simply include the target as another feature in the feature vector. The entire training example might be thought of as `12.5, 15.5, 0.10, clear, 0, 17.2`. The collection of all of these examples is known as the *training set*.

VectorAssembler is an example of Transformer within the current MLlib “Pipelines” API. It transforms another DataFrame into a DataFrame, and is composable with other transformations into a pipeline. Later in this chapter, these transformations will be connected into an actual Pipeline. Here, the transformation is just invoked directly, which is sufficient to build a first decision tree classifier model.

```
from pyspark.ml.classification import DecisionTreeClassifier

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

model = classifier.fit(assembledTrainData)
print(model.toDebugString)
...
DecisionTreeClassificationModel:
uid=DecisionTreeClassifier_d44ee1f6587c, depth=5, numNodes=41,
numClasses=8, numFeatures=54
If (feature 0 <= 3036.5)
  If (feature 0 <= 2545.5)
    If (feature 10 <= 0.5)
      If (feature 0 <= 2414.5)
        If (feature 3 <= 15.0)
          Predict: 4.0
        Else (feature 3 > 15.0)
          Predict: 3.0
      Else (feature 0 > 2414.5)
        Predict: 3.0
    ...
```

Again, the essential configuration for the classifier consists of column names: the column containing the input feature vectors and the column containing the target value to predict. Because the model will later be used to predict new values of the target, it is given the name of a column to store predictions.

Printing a representation of the model shows some of its tree structure. It consists of a series of nested decisions about features, comparing feature values to thresholds. (Here, for historical reasons, the features are only referred to by number, not name, unfortunately.)

Decision trees are able to assess the importance of input features as part of their building process. That is, they can estimate how much each input feature

contributes to making correct predictions. This information is simple to access from the model.

```
import pandas as pd

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

```

	importance
Elevation	0.826946
Hillshade_Noon	0.029365
Soil_Type_1	0.028276
Soil_Type_3	0.026778
Horizontal_Distance_To_Hydrology	0.024833
Wilderness_Area_0	0.024618
Soil_Type_31	0.018371
Wilderness_Area_2	0.012516
Horizontal_Distance_To_Roadways	0.003654
Hillshade_9am	0.002906
Horizontal_Distance_To_Fire_Points	0.001737
Soil_Type_28	0.000000

```
...
```

This pairs importance values (higher is better) with column names and prints them in order from most to least important. Elevation seems to dominate as the most important feature; most features are estimated to have virtually no importance when predicting the cover type!

The resulting `DecisionTreeClassificationModel` is itself a transformer because it can transform a data frame containing feature vectors into a data frame also containing predictions.

For example, it might be interesting to see what the model predicts on the *training* data, and compare its prediction with the known correct cover type.

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

```

```

+-----+-----+-----+-----+
+--- ...
| 6.0      | 3.0      |
| 0.0, 0.0, 0.02477728285077951, 0.6444877505567929, ...
| 6.0      | 4.0      |
| 0.0, 0.0, 0.03204807210816224, 0.2989484226339509, ...
| 6.0      | 3.0      |
| 0.0, 0.0, 0.02477728285077951, 0.6444877505567929, ...
| 6.0      | 3.0      |
| 0.0, 0.0, 0.02477728285077951, 0.6444877505567929, ...
...

```

Interestingly, the output also contains a “probability” column that gives the model’s estimate of how likely it is that each possible outcome is correct. This shows that in these instances, it’s fairly sure the answer is 3 in several cases and quite sure the answer isn’t 1.

Eagle-eyed readers might note that the probability vectors actually have eight values even though there are only seven possible outcomes. The vector’s values at indices 1 to 7 do contain the probability of outcomes 1 to 7. However, there is also a value at index 0, which always shows as probability 0.0. This can be ignored, as 0 isn’t even a valid outcome, as this says. It’s a quirk of representing this information as a vector that’s worth being aware of.

Based on this snippet, it looks like the model could use some work. Its predictions look like they are often wrong. As with the ALS implementation in chapter 3, the `DecisionTreeClassifier` implementation has several hyperparameters for which a value must be chosen, and they’ve all been left to defaults here. Here, the test set can be used to produce an unbiased evaluation of the expected accuracy of a model built with these default hyperparameters.

`MulticlassClassificationEvaluator` can compute accuracy and other metrics that evaluate the quality of the model’s predictions. It’s an example of an evaluator in MLlib, which is responsible for assessing the quality of an output `DataFrame` in some way.

```

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

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

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

```

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

```
...
0.699417706840366
0.6825807291644509
```

After being given the column containing the “label” (target, or known correct output value) and the name of the column containing the prediction, it finds that the two match about 70% of the time. This is the accuracy of this classifier. It can compute other related measures, like the **F1 score**. For purposes here, accuracy will be used to evaluate classifiers.

This single number gives a good summary of the quality of the classifier’s output. Sometimes, however, it can be useful to look at the *confusion matrix*. This is a table with a row and a column for every possible value of the target. Because there are seven target category values, this is a 7×7 matrix, where each row corresponds to an actual correct value, and each column to a predicted value, in order. The entry at row *i* and column *j* counts the number of times an example with true category *i* was predicted as category *j*. So, the correct predictions are the counts along the diagonal and the predictions are everything else.

It’s possible to calculate something like a confusion matrix directly with the DataFrame API, using its more general operators.

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

confusion_matrix.show()
```

```
...

+-----+-----+-----+-----+-----+-----+-----+
|Cover_Type|      1|      2|      3|      4|      5|      6|      7|
+-----+-----+-----+-----+-----+-----+-----+
|          | 1.0|133529| 51669|  111|   0|   0| 5119|
|          | 2.0| 56784|192506| 4859|  64|   0|  735|
|          | 3.0|   0|  3357|28225| 597|   0|   0|
|          | 4.0|   0|   0| 1487| 948|   0|   0|
|          | 5.0|   0|  8270|  271|   0|   0|   0|
|          | 6.0|   0|  3407|11829| 388|   0|   0|
```

7.0	8057	77	0	0	0	0	10300
-----	------	----	---	---	---	---	-------

1 Replace null with 0

Spreadsheet users may have recognized the problem as just like that of computing a **pivot table**. A pivot table groups values by two dimensions whose values become rows and columns of the output, and compute some aggregation within those groupings, like a count here. This is also available as a PIVOT function in several databases, and is supported by Spark SQL. It's arguably more elegant and powerful to compute it this way.

Although 70% accuracy sounds decent, it's not immediately clear whether it is outstanding or poor. How well would a simplistic approach do to establish a baseline? Just as a broken clock is correct twice a day, randomly guessing a classification for each example would also occasionally produce the correct answer.

We could construct such a random "classifier" by picking a class at random in proportion to its prevalence in the training set. For example, if 30% of the training set were cover type 1, then the random classifier would guess "1" 30% of the time. Each classification would be correct in proportion to its prevalence in the test set. If 40% of the test set were cover type 1, then guessing "1" would be correct 40% of the time. Cover type 1 would then be guessed correctly 30% x 40% = 12% of the time and contribute 12% to overall accuracy. Therefore, we can evaluate the accuracy by summing these products of probabilities:

```
from pyspark.sql import DataFrame

total = data.count()
def class_probabilities(data):
    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 = classProbabilities(train_data)
test_prior_probabilities = classProbabilities(test_data)

train_prior_probabilities
```

```

...

[Row(count_proportion=0.36457304055823236),
Row(count_proportion=0.4877308528103696),
Row(count_proportion=0.06148812265515851),
Row(count_proportion=0.004766610030522358),
Row(count_proportion=0.01626266429419196),
Row(count_proportion=0.02986871754490915),
Row(count_proportion=0.03530999210661611)]

...

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
...

0.3765186220733993

```

- ❶ Count by category
- ❷ Order counts by category
- ❸ Sum products of pairs in training, test sets

Random guessing achieves 37% accuracy then, which makes 70% seem like a good result after all. But this result was achieved with default hyperparameters. We can do even better by exploring what the hyperparameters actually mean for the tree-building process. That is what we will do in the next section.

Decision Tree Hyperparameters

In chapter 3, the ALS algorithm exposed several hyperparameters whose values we had to choose by building models with various combinations of values and then assessing the quality of each result using some metric. The process is the same here, although the metric is now multiclass accuracy instead of AUC. The hyperparameters controlling how the tree's decisions are chosen will be quite different as well: maximum depth, maximum bins, impurity measure, and minimum information gain.

Maximum depth simply limits the number of levels in the decision tree. It is the maximum number of chained decisions that the classifier will make to classify an example. It is useful to limit this to avoid overfitting the training data, as illustrated previously in the pet store example.

The decision tree algorithm is responsible for coming up with potential decision rules to try at each level, like the `weight >= 100` or `weight >= 500` decisions in the pet store example. Decisions are always of the same form: for numeric features, decisions are of the form `feature >= value`; and for categorical features, they are of the form `feature in (value1, value2, ...)`. So, the set of decision rules to try is really a set of values to plug in to the decision rule. These are referred to as “bins” in the PySpark MLlib implementation. A larger number of bins requires more processing time but might lead to finding a more optimal decision rule.

What makes a decision rule good? Intuitively, a good rule would meaningfully distinguish examples by target category value. For example, a rule that divides the Covtype data set into examples with only categories 1–3 on the one hand and 4–7 on the other would be excellent because it clearly separates some categories from others. A rule that resulted in about the same mix of all categories as are found in the whole data set doesn’t seem helpful. Following either branch of such a decision leads to about the same distribution of possible target values, and so doesn’t really make progress toward a confident classification.

Put another way, good rules divide the training data’s target values into relatively homogeneous, or “pure,” subsets. Picking a best rule means minimizing the impurity of the two subsets it induces. There are two commonly used measures of impurity: **Gini impurity** and **entropy**.

Gini impurity is directly related to the accuracy of the random-guess classifier. Within a subset, it is the probability that a randomly chosen classification of a randomly chosen example (both according to the distribution of classes in the subset) is *incorrect*. This is the sum of products of proportions of classes, but with themselves and subtracted from 1. If a subset has N classes and p_i is the proportion of examples of class i , then its Gini impurity is given in the Gini impurity equation:

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

If the subset contains only one class, this value is 0 because it is completely “pure.” When there are N classes in the subset, this value is larger than 0 and is largest when the classes occur the same number of times—maximally impure.

Entropy is another measure of impurity, borrowed from information theory. Its nature is more difficult to explain, but it captures how much uncertainty the collection of target values in the subset implies about predictions for data that falls in that subset. A subset containing one class suggests that the outcome for the subset is completely certain and has 0 entropy—no uncertainty. A subset containing one of each possible class, on the other hand, suggests a lot of uncertainty about predictions for that subset because data have been observed with all kinds of target values. This has high entropy. Hence, low entropy, like low Gini impurity, is a good thing. Entropy is defined by the entropy equation:

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

Interestingly, uncertainty has units. Because the logarithm is the natural log (base e), the units are *nats*, the base- e counterpart to more familiar *bits* (which we can obtain by using log base 2 instead). It really is measuring information, so it’s also common to talk about the *information gain* of a decision rule when using entropy with decision trees.

One or the other measure may be a better metric for picking decision rules in a given data set. They are, in a way, similar. Both involve a weighted average: a sum over values weighted by p_i . The default in PySpark's implementation is Gini impurity.

Finally, minimum information gain is a hyperparameter that imposes a minimum information gain, or decrease in impurity, for candidate decision rules. Rules that do not improve the subsets impurity enough are rejected. Like a lower maximum depth, this can help the model resist overfitting because decisions that barely help divide the training input may in fact not helpfully divide future data at all.

Now that we understand the relevant hyperparameters of a decision tree algorithm, we will tune our model in the next section to improve its performance.

Tuning Decision Trees

It's not obvious from looking at the data which impurity measure leads to better accuracy, or what maximum depth or number of bins is enough without being excessive. Fortunately, as in chapter 3, it's simple to let PySpark try a number of combinations of these values and report the results.

First, it's necessary to set up a pipeline encapsulating the same two steps above. Creating the `VectorAssembler` and `DecisionTreeClassifier` and chaining these two `Transformers` together results in a single `Pipeline` object that represents these two operations together as one operation:

```
from pyspark.ml import Pipeline

assembler = VectorAssembler(inputCols=inputCols,
                             outputCol="featureVector")
classifier = DecisionTreeClassifier(seed=1234,
                                    labelCol="Cover_Type", featuresCol="featureVector",
                                    predictionCol="prediction")

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

Naturally, pipelines can be much longer and more complex. This is about as simple as it gets. Now we can also define the combinations of hyperparameters that should be tested using the PySpark ML API's built-in support, `ParamGridBuilder`. It's also time to define the evaluation metric that will be used to pick the “best” hyperparameters, and that is again `MulticlassClassificationEvaluator` here.

```
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")
```

This means that a model will be built and evaluated for two values of four hyperparameters. That's 16 models. They'll be evaluated by multiclass accuracy. Finally, `TrainValidationSplit` brings these components together—the pipeline that makes models, model evaluation metrics, and hyperparameters to try—and can run the evaluation on the training data. It's worth noting that `CrossValidator` could be used here as well to perform full k -fold cross-validation, but it is k times more expensive and doesn't add as much value in the presence of big data. So, `TrainValidationSplit` is used here.

```
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)
```

This will take minutes or more, depending on your hardware, because it's building and evaluating many models. Note the train ratio parameter is set to 0.9. This means that the training data is actually further subdivided by `TrainValidationSplit` into 90%/10% subsets. The former is used for training each model. The remaining 10% of the input is held out as a cross-validation set to evaluate the model. If it's already holding out some data for evaluation, then why did we hold out 10% of the original data as a test set?

If the purpose of the CV set was to evaluate *parameters* that fit to the *training* set, then the purpose of the test set is to evaluate *hyperparameters* that were "fit" to the CV set. That is, the test set ensures an unbiased estimate of the accuracy of the final, chosen model and its hyperparameters.

Say that the best model chosen by this process exhibits 90% accuracy on the CV set. It seems reasonable to expect it will exhibit 90% accuracy on future data. However, there's an element of randomness in how these models are built. By chance, this model and evaluation could have turned out unusually well. The top model and evaluation result could have benefited from a bit of luck, so its accuracy estimate is likely to be slightly optimistic. Put another way, hyperparameters can overfit too.

To really assess how well this best model is likely to perform on future examples, we need to evaluate it on examples that were not used to train it. But we also need to avoid examples in the CV set that were used to evaluate it. That is why a third subset, the test set, was held out.

The result of the validator contains the best model it found. This itself is a representation of the best overall *pipeline* it found, because we provided an instance of a pipeline to run. In order to query the parameters chosen by `DecisionTreeClassifier`, it's necessary to manually extract `DecisionTreeClassificationModel` from the resulting `PipelineModel`, which is the final stage in the pipeline.

```
from pprint import pprint

best_model = validator_model.bestModel
pprint(best_model.stages[1].extractParamMap())

...
{Param(parent='DecisionTreeClassifier_2b5c68a71772',
```

```

name='minInstancesPerNode', doc='Minimum number of instances each
child must have after split. If a split causes the left or right
child to have fewer than minInstancesPerNode, the split will be
discarded as invalid. Should be >= 1.'): 1,
    Param(parent='DecisionTreeClassifier_2b5c68a71772',
name='minWeightFractionPerNode', doc='Minimum fraction of the
weighted sample count that each child must have after split. If a
split causes the fraction of the total weight in the left or right
child to be less than minWeightFractionPerNode, the split will be
discarded as invalid. Should be in interval [0.0, 0.5).'): 0.0,
    Param(parent='DecisionTreeClassifier_2b5c68a71772',
name='cacheNodeIds', doc='If false, the algorithm will pass trees to
executors to match instances with nodes. If true, the algorithm will
cache node IDs for each instance. Caching can speed up training of
deeper trees. Users can set how often should the cache be
checkpointed or disable it by setting checkpointInterval.'): False,
    ...
}

```

This contains a lot of information about the fitted model, but it also tells us that “entropy” apparently worked best as the impurity measure and that a max depth of 20 was not surprisingly better than 1. It might be surprising that the best model was fit with just 40 bins, but this is probably a sign that 40 was “plenty” rather than “better” than 300. Lastly, no minimum information gain was better than a small minimum, which could imply that the model is more prone to underfit than overfit.

You may wonder if it is possible to see the accuracy that each of the models achieved for each combination of hyperparameters. The hyperparameters as well as the evaluations are exposed by `getEstimatorParamMaps` and `validationMetrics`, respectively. They can be combined to display all of the parameter combinations sorted by metric value:

```

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

...
0.9138483377774368
{

```

```

        dtc_3e3b8bb692d1-impurity: entropy,
        dtc_3e3b8bb692d1-maxBins: 40,
        dtc_3e3b8bb692d1-maxDepth: 20,
        dtc_3e3b8bb692d1-minInfoGain: 0.0
    }

0.9122369506416774
{
    dtc_3e3b8bb692d1-impurity: entropy,
    dtc_3e3b8bb692d1-maxBins: 300,
    dtc_3e3b8bb692d1-maxDepth: 20,
    dtc_3e3b8bb692d1-minInfoGain: 0.0
}
...

```

What was the accuracy that this model achieved on the CV set? And finally, what accuracy does the model achieve on the test set?

```

metrics.sort(reverse=True)
print(metrics[0])
multiclassEval.evaluate(best_model.transform(test_data)) ❶

...
0.9138483377774368
0.9139978718291971

```

❶ bestModel is a complete pipeline.

The results are both about 91%. It happens that the estimate from the CV set was pretty fine to begin with. In fact, it is not usual for the test set to show a very different result.

This is an interesting point at which to revisit the issue of overfitting. As discussed previously, it's possible to build a decision tree so deep and elaborate that it fits the given training examples very well or perfectly but fails to generalize to other examples because it has fit the idiosyncrasies and noise of the training data too closely. This is a problem common to most machine learning algorithms, not just decision trees.

When a decision tree has overfit, it will exhibit high accuracy when run on the same training data that it fit the model to, but low accuracy on other examples. Here, the final model's accuracy was about 91% on other, new examples. Accuracy can just as easily be evaluated over the same data that the model was

trained on, `trainData`. This gives an accuracy of about 95%. The difference is not large but suggests that the decision tree has overfit the training data to some extent. A lower maximum depth might be a better choice.

So far, we've implicitly treated all input features, including categoricals, as if they're numeric. Can we improve our model's performance further by treating categorical features as exactly that? We will explore this next.

Categorical Features Revisited

The categorical features in our dataset are one-hot encoded as several binary 0/1 values. Treating these individual features as numeric turns out to be fine, because any decision rule on the “numeric” features will choose thresholds between 0 and 1, and all are equivalent since all values are 0 or 1.

Of course, this encoding forces the decision tree algorithm to consider the values of the underlying categorical features individually. Because features like soil type are broken down into many features, and because decision trees treat features individually, it is harder to relate information about related soil types.

For example, nine different soil types are actually part of the Leighcan family, and they may be related in ways that the decision tree can exploit. If soil type were encoded as a single categorical feature with 40 soil values, then the tree could express rules like “if the soil type is one of the nine Leighton family types” directly. However, when encoded as 40 features, the tree would have to learn a sequence of nine decisions on soil type to do the same, this expressiveness may lead to better decisions and more efficient trees.

However, having 40 numeric features represent one 40-valued categorical feature increases memory usage and slows things down.

What about undoing the one-hot encoding? This would replace, for example, the four columns encoding wilderness type with one column that encodes the wilderness type as a number between 0 and 3, like “Cover_Type”.

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

```

ness")

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(
n("soil", unhot_udf(col("soil"))))

return with_soil

```

- ❶ Note UDF definition
- ❷ Drop one-hot columns; no longer needed

Here `VectorAssembler` is deployed to combine the 4 and 40 wilderness and soil type columns into two `Vector` columns. The values in these `Vectors` are all 0, except for one location that has a 1. There's no simple `DataFrame` function for this, so we have to define our own UDF that can be used to operate on columns. This turns these two new columns into numbers of just the type we need.

From here, nearly the same process as above can be used to tune the hyperparameters of a decision tree model built on this data and to choose and evaluate a best model. There's one important difference, however. The two new numeric columns have nothing about them that indicates they're actually an encoding of categorical values. To treat them as numbers would be wrong, as their ordering is meaningless. However, it would silently succeed; the information in these features would be all but lost though.

Internally `MLlib` can store additional metadata about each column. The details of this data are generally hidden from the caller, but includes information such as whether the column encodes a categorical value and how many distinct values it takes on. In order to add this metadata, it's necessary to put the data through `VectorIndexer`. Its job is to turn input into properly labeled categorical feature columns. Although we did much of the work already to turn

the categorical features into 0-indexed values, `VectorIndexer` will take care of the metadata.

We need to add this stage to the `Pipeline`:

```
import org.apache.spark.ml.feature.VectorIndexer

unencTrainData = unencodeOneHot(trainData)

cols = unencTrainData.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` because soil has 40 values

The approach assumes that the training set contains all possible values of each of the categorical features at least once. That is, it works correctly only if all 4 soil values and all 40 wilderness values appear in the training set so that all possible values get a mapping. Here, that happens to be true, but may not be for small training sets of data in which some labels appear very infrequently. In those cases, it could be necessary to manually create and add a `VectorIndexerModel` with the complete value mapping supplied manually.

Aside from that, the process is the same as before. You should find that it chose a similar best model but that accuracy on the test set is about 93%. By treating categorical features as actual categorical features, the classifier improved its accuracy by almost 2%.

We have trained and tuned a decision tree. Now, we will move on to random forests, a more powerful algorithm. As we will see in the next section, implementing them using PySpark will be surprisingly straightforward at this point.

Random Forests

If you have been following along with the code examples, you may have noticed that your results differ slightly from those presented in the code listings in the book. That is because there is an element of randomness in building decision trees, and the randomness comes into play when you're deciding what data to use and what decision rules to explore.

The algorithm does not consider every possible decision rule at every level. To do so would take an incredible amount of time. For a categorical feature over N values, there are $2^N - 2$ possible decision rules (every subset except the empty set and entire set). For even moderately large N , this would create billions of candidate decision rules.

Instead, decision trees use several heuristics to determine which few rules to actually consider. The process of picking rules also involves some randomness; only a few features picked at random are looked at each time, and only values from a random subset of the training data. This trades a bit of accuracy for a lot of speed, but it also means that the decision tree algorithm won't build the same tree every time. This is a good thing.

It's good for the same reason that the "wisdom of the crowds" usually beats individual predictions. To illustrate, take this quick quiz: How many black taxis operate in London?

Don't peek at the answer; guess first.

I guessed 10,000, which is well off the correct answer of about 19,000. Because I guessed low, you're a bit more likely to have guessed higher than I did, and so the average of our answers will tend to be more accurate. There's that regression to the mean again. The average guess from an informal poll of 13 people in the office was indeed closer: 11,170.

A key to this effect is that the guesses were independent and didn't influence one another. (You didn't peek, did you?) The exercise would be useless if we had all agreed on and used the same methodology to make a guess, because the guesses would have been the same answer—the same potentially quite wrong answer. It would even have been different and worse if I'd merely influenced you by stating my guess upfront.

It would be great to have not one tree, but many trees, each producing reasonable but different and independent estimations of the right target value. Their collective average prediction should fall close to the true answer, more than any individual tree's does. It's the *randomness* in the process of building that helps create this independence. This is the key to *random forests*.

Randomness is injected by building many trees, each of which sees a different random subset of data—and even of features. This makes the forest as a whole less prone to overfitting. If a particular feature contains noisy data or is deceptively predictive only in the *training* set, then most trees will not consider this problem feature most of the time. Most trees will not fit the noise and will tend to “outvote” the trees that have fit the noise in the forest.

The prediction of a random forest is simply a weighted average of the trees' predictions. For a categorical target, this can be a majority vote or the most probable value based on the average of probabilities produced by the trees. Random forests, like decision trees, also support regression, and the forest's prediction in this case is the average of the number predicted by each tree.

While random forests are a more powerful and complex classification technique, the good news is that it's virtually no different to use it in the pipeline that has been developed in this chapter. Simply drop in a `RandomForestClassifier` in place of `DecisionTreeClassifier` and proceed as before. There's really no more code or API to understand in order to use it.

```
from pyspark.ml.classification import RandomForestClassifier

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

Note that this classifier has another hyperparameter: the number of trees to build. Like the max bins hyperparameter, higher values should give better results up to a point. The cost, however, is that building many trees of course takes many times longer than building one.

The accuracy of the best random forest model produced from a similar tuning process is 95% off the bat—about 2% better already, although viewed another way, that’s a 28% reduction in the error rate over the best decision tree built previously, from 7% down to 5%. You may do better with further tuning.

Incidentally, at this point we have a more reliable picture of feature importance:

```
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)
(0.028408483578137605, Hillshade_9am)
(0.027185325937200706, Aspect)
(0.027075578474331806, Hillshade_3pm)
(0.015317564027809389, Slope)
```

Random forests are appealing in the context of big data because trees are supposed to be built independently, and big data technologies like Spark and MapReduce inherently need *data-parallel* problems, where parts of the overall solution can be computed independently on parts of the data. The fact that trees can, and should, train on only a subset of features or input data makes it trivial to parallelize building the trees.

Making Predictions

Building a classifier, while an interesting and nuanced process, is not the end goal. The goal is to make predictions. This is the payoff, and it is comparatively quite easy.

The resulting “best model” is actually a whole pipeline of operations, which encapsulate how input is transformed for use with the model and includes the model itself, which can make predictions. It can operate on a data frame of new input. The only difference from the `data` DataFrame we started with is that it lacks the “Cover_Type” column. When we’re making predictions—especially about the future, says Mr. Bohr—the output is of course not known.

To prove it, try dropping the “Cover_Type” from the test data input and obtaining a prediction:

```
unenc_test_data = unencode_one_hot(test_data)
bestModel.transform(unenc_test_data.drop("Cover_Type")).select("prediction").show()
```

```
...
+-----+
|prediction|
+-----+
|         6.0|
+-----+
```

The result should be 6.0, which corresponds to class 7 (the original feature was 1-indexed) in the original Covtype data set. The predicted cover type for the land described in this example is Krummholz.

Where to Go from Here

This chapter introduced two related and important types of machine learning, classification and regression, along with some foundational concepts in building and tuning models: features, vectors, training, and cross-validation. It demonstrated how to predict a type of forest cover from things like location and soil type using the Covtype data set, with decision trees and forests implemented in PySpark.

As with recommenders in chapter 3, it could be useful to continue exploring the effect of hyperparameters on accuracy. Most decision tree hyperparameters

trade time for accuracy: more bins and trees generally produce better accuracy but hit a point of diminishing returns.

The classifier here turned out to be very accurate. It's unusual to achieve more than 95% accuracy. In general, you will achieve further improvements in accuracy by including more features or transforming existing features into a more predictive form. This is a common, repeated step in iteratively improving a classifier model. For example, for this data set, the two features encoding horizontal and vertical distance-to-surface-water features could produce a third feature: straight-line distance-to-surface-water features. This might turn out to be more useful than either original feature. Or, if it were possible to collect more data, we might try adding new information like soil moisture in order to improve classification.

Of course, not all prediction problems in the real world are exactly like the Covtype data set. For example, some problems require predicting a continuous numeric value, not a categorical value. Much of the same analysis and code applies to this type of *regression* problem; the `RandomForestRegressor` class will be of use in this case.

Furthermore, decision trees and forests are not the only classification or regression algorithms, and not the only ones implemented in PySpark, as we'd pointed out at the beginning of the chapter. Each of these algorithms operates quite differently from decision trees and forests. However, many elements are the same: they plug into a `Pipeline` and operate on columns in a data frame, and have hyperparameters that you must select using training, cross-validation, and test subsets of the input data. The same general principles, with these other algorithms, can also be deployed to model classification and regression problems.

These have been examples of supervised learning. What happens when some, or all, of the target values are unknown? The following chapter will explore what can be done in this situation.

Chapter 5. Anomaly Detection in Network Traffic with K-means Clustering

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

Classification and regression are powerful, well-studied techniques in machine learning. Chapter 4 demonstrated using a classifier as a predictor of unknown values. But there was a catch: in order to predict unknown values for new data, we had to know the target values for many previously seen examples. Classifiers can only help if we, the data scientists, know what we are looking for and can provide plenty of examples where input produced a known output. These were collectively known as *supervised learning* techniques, because their learning process receives the correct output value for each example in the input.

However, sometimes the correct output is unknown for some or all examples. Consider the problem of dividing up an ecommerce site’s customers by their shopping habits and tastes. The input features are their purchases, clicks, demographic information, and more. The output should be groupings of customers: perhaps one group will represent fashion-conscious buyers, another will turn out to correspond to price-sensitive bargain hunters, and so on.

If you were asked to determine this target label for each new customer, you would quickly run into a problem in applying a supervised learning technique like a classifier: you don't know *a priori* who should be considered fashion-conscious, for example. In fact, you're not even sure if "fashion-conscious" is a meaningful grouping of the site's customers to begin with!

Fortunately, *unsupervised learning* techniques can help. These techniques do not learn to predict a target value, because none is available. They can, however, learn structure in data and find groupings of similar inputs, or learn what types of input are likely to occur and what types are not. This chapter will introduce unsupervised learning using clustering implementations in MLlib.

Anomaly Detection

The inherent problem of anomaly detection is, as its name implies, that of finding unusual things. If we already knew what "anomalous" meant for a data set, we could easily detect anomalies in the data with supervised learning. An algorithm would receive inputs labeled "normal" and "anomaly", and learn to distinguish the two. However, the nature of anomalies is that they are unknown unknowns. Put another way, an anomaly that has been observed and understood is no longer an anomaly.

Anomaly detection is often used to find fraud, detect network attacks, or discover problems in servers or other sensor-equipped machinery. In these cases, it's important to be able to find new types of anomalies that have never been seen before—new forms of fraud, intrusions, and failure modes for servers.

Unsupervised learning techniques are useful in these cases because they can learn what input data normally looks like, and therefore detect when new data is unlike past data. Such new data is not necessarily attacks or fraud; it is simply unusual, and therefore, worth further investigation.

K-means Clustering

Clustering is the best-known type of unsupervised learning. Clustering algorithms try to find natural groupings in data. Data points that are like one another but unlike others are likely to represent a meaningful grouping, so clustering algorithms try to put such data into the same cluster.

K-means clustering may be the most widely used clustering algorithm. It attempts to detect k clusters in a data set, where k is given by the data scientist. k is a hyperparameter of the model, and the right value will depend on the data set. In fact, choosing a good value for k will be a central plot point in this chapter.

What does “like” mean when the data set contains information like customer activity? Or transactions? K-means requires a notion of distance between data points. It is common to use simple Euclidean distance to measure distance between data points with K-means, and as it happens, this is the only distance function supported by Spark MLlib as of this writing. The Euclidean distance is defined for data points whose features are all numeric. “Like” points are those whose intervening distance is small.

To K-means, a cluster is simply a point: the center of all the points that make up the cluster. These are, in fact, just feature vectors containing all numeric features, and can be called vectors. However, it may be more intuitive to think of them as points here, because they are treated as points in a Euclidean space.

This center is called the cluster *centroid*, and is the arithmetic mean of the points—hence the name *K-means*. To start, the algorithm picks some data points as the initial cluster centroids. Then each data point is assigned to the nearest centroid. Then for each cluster, a new cluster centroid is computed as the mean of the data points just assigned to that cluster. This process is repeated.

Enough about K-means for now. Some more interesting details will emerge in the use case to follow.

Network Intrusion

Cyberattacks are increasingly visible in the news. Some attacks attempt to flood a computer with network traffic to crowd out legitimate traffic. But in other cases, attacks attempt to exploit flaws in networking software to gain unauthorized access to a computer. While it's quite obvious when a computer is being bombarded with traffic, detecting an exploit can be like searching for a needle in an incredibly large haystack of network requests.

Some exploit behaviors follow known patterns. For example, accessing every port on a machine in rapid succession is not something any normal software program should ever need to do. However, it is a typical first step for an attacker looking for services running on the computer that may be exploitable.

If you were to count the number of distinct ports accessed by a remote host in a short time, you would have a feature that probably predicts a port-scanning attack quite well. A handful is probably normal; hundreds indicates an attack. The same goes for detecting other types of attacks from other features of network connections—number of bytes sent and received, TCP errors, and so forth.

But what about those unknown unknowns? The biggest threat may be the one that has never yet been detected and classified. Part of detecting potential network intrusions is detecting anomalies. These are connections that aren't known to be attacks but do not resemble connections that have been observed in the past.

Here, unsupervised learning techniques like K-means can be used to detect anomalous network connections. K-means can cluster connections based on statistics about each of them. The resulting clusters themselves aren't interesting per se, but they collectively define types of connections that are like past connections. Anything not close to a cluster could be anomalous. Clusters are interesting insofar as they define regions of normal connections; everything else outside is unusual and potentially anomalous.

KDD Cup 1999 Data Set

The **KDD Cup** was an annual data mining competition organized by a special interest group of the Association for Computing Machinery (ACM). Each year, a machine learning problem was posed, along with a data set, and researchers were invited to submit a paper detailing their best solution to the problem. It was like **Kaggle** before there was Kaggle. In 1999, the topic was network intrusion, and the data set is **still available**. The remainder of this chapter will walk through building a system to detect anomalous network traffic using Spark, by learning from this data.

Don't use this data set to build a real network intrusion system! The data did not necessarily reflect real network traffic at the time—even if it did, it reflects traffic patterns from 17 years ago.

Fortunately, the organizers had already processed raw network packet data into summary information about individual network connections. The data set is about 708 MB in size and contains about 4.9 million connections. This is large, if not massive, and is certainly sufficient for our purposes here. For each connection, the data set contains information like the number of bytes sent, login attempts, TCP errors, and so on. Each connection is one line of CSV-formatted data set, containing 38 features, like this:

```
0, tcp, http, SF, 215, 45076,  
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1,  
0.00, 0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0, 0, 0.00,  
0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, normal.
```

This connection, for example, was a TCP connection to an HTTP service—215 bytes were sent and 45,706 bytes were received. The user was logged in, and so on. Many features are counts, like `num_file_creations` in the 17th column.

Many features take on the value 0 or 1, indicating the presence or absence of a behavior, like `su_attempted` in the 15th column. They look like the

one-hot encoded categorical features from chapter 4, but are not grouped and related in the same way. Each is like a yes/no feature, and is therefore arguably a categorical feature. It is not always valid to translate categorical features as numbers and treat them as if they had an ordering. However, in the special case of a binary categorical feature, in most machine learning algorithms, mapping these to a numeric feature taking on values 0 and 1 will work well.

The rest are ratios like `dst_host_srv_error_rate` in the next-to-last column, and take on values from 0.0 to 1.0, inclusive.

Interestingly, a label is given in the last field. Most connections are labeled `normal`, but some have been identified as examples of various types of network attacks. These would be useful in learning to distinguish a known attack from a normal connection, but the problem here is anomaly detection and finding potentially new and unknown attacks. This label will be mostly set aside for our purposes.

A First Take on Clustering

Unzip the `kddcup.data.gz` data file and copy it into HDFS. This example, like others, will assume the file is available at `/user/ds/kddcup.data`. Open the `spark-shell`, and load the CSV data as a data frame. It's a CSV file again, but without header information. It's necessary to supply column names as given in the accompanying `kddcup.names` file.

```
data_without_header = spark.read.option("inferSchema",
True).option("header", False).csv("data/kdd_1999_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)
```

Begin by exploring the data set. What labels are present in the data, and how many are there of each? The following code simply counts by label and prints the results in descending order by count.

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

```

There are 23 distinct labels, and the most frequent are `smurf.` and `neptune.` attacks.

Note that the data contains nonnumeric features. For example, the second column may be `tcp`, `udp`, or `icmp`, but K-means clustering requires numeric features. The final label column is also nonnumeric. To begin, these will simply be ignored.

Aside from this, creating a K-means clustering of the data follows the same pattern as was seen in chapter 4. A `VectorAssembler` creates a feature vector, a `KMeans` implementation creates a model from the feature vectors,

and a `Pipeline` stitches it all together. From the resulting model, it's possible to extract and examine the cluster centers.

```
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([1.09990000e+04, 0.00000000e+00, 1.3099374e+09,
0.00000000e+00,
        0.00000000e+00, 0.00000000e+00, ...])]
```

It's not easy to interpret the numbers intuitively, but each of these represents the center (also known as centroid) of one of the clusters that the model produced. The values are the coordinates of the centroid in terms of each of the numeric input features.

Two vectors are printed, meaning K-means was fitting $k=2$ clusters to the data. For a complex data set that is known to exhibit at least 23 distinct types of connections, this is almost certainly not enough to accurately model the distinct groupings within the data.

This is a good opportunity to use the given labels to get an intuitive sense of what went into these two clusters by counting the labels within each cluster.

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

The result shows that the clustering was not at all helpful. Only one data point ended up in cluster 1!

Choosing k

Two clusters are plainly insufficient. How many clusters are appropriate for this data set? It's clear that there are 23 distinct patterns in the data, so it seems that k could be at least 23, or likely even more. Typically, many values of k are tried to find the best one. But what is “best”?

A clustering could be considered good if each data point were near its closest centroid, where “near” is defined by the Euclidean distance. This is a simple, common way to evaluate the quality of a clustering, by the mean of these distances over all points, or sometimes, the mean of the distances squared. In fact, `KMeansModel` offers a `computeCost` method that

computes the sum of squared distances and can easily be used to compute the mean squared distance.

Unfortunately, there is no simple `Evaluator` implementation to compute this measure, not like those available to compute multiclass classification metrics. It's simple enough to manually evaluate the clustering cost for several values of k . Note that this code could take 10 minutes or more to run.

```
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]).s
    etOutputCol("featureVector")
    kmeans =
    KMeans().setSeed(randint(100, 100000)).setK(k).setPredictionCol("c
    luster").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))

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

The printed result shows that the score decreases as k increases. Note that scores are shown in scientific notation; the first value is over 10^7 , not just a

bit over 6.

Again, your values will be somewhat different. The clustering depends on a randomly chosen initial set of centroids.

However, this much is obvious. As more clusters are added, it should always be possible to put data points closer to the nearest centroid. In fact, if k is chosen to equal the number of data points, the average distance will be 0 because every point will be its own cluster of one!

Worse, in the preceding results, the distance for $k=80$ is higher than for $k=60$. This shouldn't happen because higher k always permits at least as good a clustering as a lower k . The problem is that K-means is not necessarily able to find the optimal clustering for a given k . Its iterative process can converge from a random starting point to a local minimum, which may be good but is not optimal.

This is still true even when more intelligent methods are used to choose initial centroids. K-means++ and **K-means||** are variants of selection algorithms that are more likely to choose diverse, separated centroids and lead more reliably to a good clustering. Spark MLlib, in fact, implements K-means||. However, all still have an element of randomness in selection and can't guarantee an optimal clustering.

The random starting set of clusters chosen for $k=80$ perhaps led to a particularly suboptimal clustering, or it may have stopped early before it reached its local optimum.

We can improve it by running the iteration longer. The algorithm has a threshold via `setTol()` that controls the minimum amount of cluster centroid movement considered significant; lower values mean the K-means algorithm will let the centroids continue to move longer. Increasing the maximum number of iterations with `setMaxIter()` also prevents it from potentially stopping too early at the cost of possibly more computation.


```

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]).s
etOutputCol("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 Increase from default 20
- ❷ Decrease from default 1.0e-4

This time, at least the scores decrease consistently:

```

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

```

We want to find a point past which increasing k stops reducing the score much—or an “elbow” in a graph of k versus score, which is generally decreasing but eventually flattens out. Here, it seems to be decreasing notably past 100. The right value of k may be past 100.

Visualization with SparkR

At this point, it could be useful to step back and understand more about the data before clustering again. In particular, looking at a plot of the data points could be helpful.

Spark itself has no tools for visualization, but the popular open source statistical environment **R** has libraries for both data exploration and data visualization. Furthermore, Spark also provides some basic integration with R via **SparkR**. This brief section will demonstrate using R and SparkR to cluster the data and explore the clustering.

SparkR is a variant of the `spark-shell` used throughout this book, and is invoked with the command `sparkR`. It runs a local R interpreter, like `spark-shell` runs a variant of the Scala shell as a local process. The machine that runs `sparkR` needs a local installation of R, which is not included with Spark. This can be installed, for example, with `sudo apt-get install r-base` on Linux distributions like Ubuntu, or `brew install R` with **Homebrew** on macOS.

`sparkR` is a command-line shell environment, like R. To view visualizations, it's necessary to run these commands within an IDE-like environment that can display images. **RStudio** is an IDE for R (and works with SparkR); it runs on a desktop operating system so it will only be usable here if you are experimenting with Spark locally rather than on a cluster.

If you are running Spark locally, **download** the free version of RStudio and install it. If not, then most of the rest of this example can still be run with `sparkR` on a command line; for example, on a cluster. It won't be possible to display visualizations this way though.

If running via RStudio, launch the IDE and configure `SPARK_HOME` and `JAVA_HOME`, if your local environment does not already set them, to point to the Spark and JDK installation directories, respectively.

```
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 Replace with actual paths, of course.

Note that these steps aren't needed if you are running `sparkR` on the command line. Instead, it accepts command-line configuration parameters like `--driver-memory`, just like `spark-shell`.

SparkR is an R-language wrapper around the same `DataFrame` and `MLlib` APIs that have been demonstrated in this chapter. It's therefore possible to recreate a K-means simple clustering of the data:

```
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",
  "error_rate", "srv_error_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_error_rate", "dst_host_srv_error_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 Replace with path to *kddcup.data*.
- ❷ Name columns.
- ❸ Drop nonnumeric columns again.

④ `~ .` means all columns.

From here, it's straightforward to assign a cluster to each data point. The operations above show usage of the SparkR APIs, which naturally correspond to core Spark APIs but are expressed as R libraries in R-like syntax. The actual clustering is executed using the same JVM-based, Scala-language implementation in MLlib. These operations are effectively a *handle* or remote control to distributed operations that are not executing in R.

R has its own rich set of libraries for analysis, and its own similar concept of a data frame. It is sometimes useful, therefore, to pull some data down into the R interpreter in order to be able to use these native R libraries, which are unrelated to Spark.

Of course, R and its libraries are not distributed, and so it's not feasible to pull the whole data set of 4,898,431 data points into R. However, it's easy to pull only a sample:

```
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 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 0 ...
...
 $ prediction    : int  33 33 33 0 0 0 0 0 33 33 ...
```

❶ 1% sample without replacement

`clustering_sample` is actually a local R data frame, not a Spark `DataFrame`, so it can be manipulated like any other data in R. Above,

`str()` shows the structure of the data frame.

For example, it's possible to extract the cluster assignment and then show statistics about the distribution of assignments:

```
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 Only the clustering assignment column
- ❷ Everything but the clustering assignment

For example, this shows that most points fell into cluster 0. Although much more could be done with this data in R, further coverage of this is beyond the scope of this book.

To visualize the data, a library called `rgl` is required. It will only be functional if running this example in RStudio. First, install (once) and load the library:

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

Note that R may prompt you to download other packages or compiler tools to complete installation, because installing the package means compiling its source code.

This data set is 38-dimensional. It will have to be projected down into at most three dimensions in order to visualize it with a *random projection*:

```
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 Make a random 3D projection and normalize
- ❷ Project and make a new data frame

This creates a 3D data set out of a 38D data set by choosing three random unit vectors and projecting the data onto them. This is a simplistic, rough-and-ready form of dimension reduction. Of course, there are more sophisticated dimension reduction algorithms, like **principal component analysis (PCA)** or the **singular value decomposition (SVD)**. These are available in R but take much longer to run. For purposes of visualization in this example, a random projection achieves much the same result, faster.

Finally, the clustered points can be plotted in an interactive 3D visualization:

```

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

```

Note that this will require running RStudio in an environment that supports the `rgl` library and graphics. For example, on macOS, it requires that X11 from Apple's Developer Tools be installed.

The resulting visualization in **Figure 5-1** shows data points in 3D space. Many points fall on top of one another, and the result is sparse and hard to interpret. However, the dominant feature of the visualization is its L shape. The points seem to vary along two distinct dimensions, and little in other dimensions.

This makes sense because the data set has two features that are on a much larger scale than the others. Whereas most features have values between 0 and 1, the bytes-sent and bytes-received features vary from 0 to tens of thousands. The Euclidean distance between points is therefore almost

completely determined by these two features. It's almost as if the other features don't exist! So it's important to normalize away these differences in scale to put features on near-equal footing.

Feature Normalization

We can normalize each feature by converting it to a **standard score**. This means subtracting the mean of the feature's values from each value, and dividing by the standard deviation, as shown in the standard score equation:

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

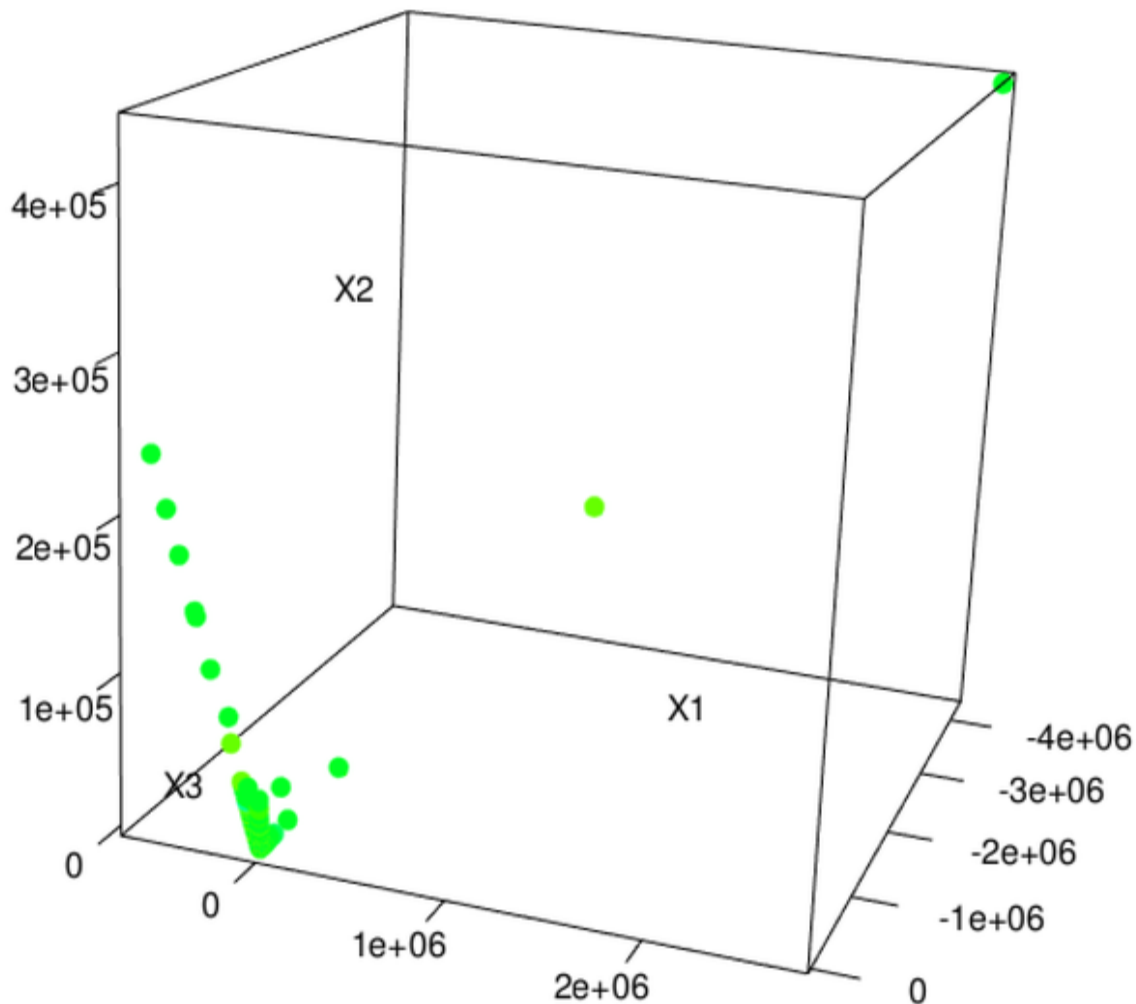


Figure 5-1. Random 3D projection

In fact, subtracting means has no effect on the clustering because the subtraction effectively shifts all the data points by the same amount in the same directions. This does not affect interpoint Euclidean distances.

MLlib provides `StandardScaler`, a component that can perform this kind of standardization and be easily added to the clustering pipeline.

We can run the same test with normalized data on a higher range of 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))

```

This has helped put dimensions on more equal footing, and the absolute distances between points (and thus the cost) is much smaller in absolute terms. However, there isn't yet an obvious value of k beyond which increasing it does little to improve the cost:

```

(60, 1.2454250178069293)
(90, 0.7767730051608682)
(120, 0.5070473497003614)
(150, 0.4077081720067704)
(180, 0.3344486714980788)
(210, 0.276237617334138)
(240, 0.24571877339169032)
(270, 0.21818167354866858)

```

Another 3D visualization of the normalized data points reveals a richer structure, as expected. Some points are spaced in regular, discrete intervals in a direction; these are likely projections of discrete dimensions in the data, like counts. With 100 clusters, it's hard to make out which points come from which clusters. One large cluster seems to dominate, and many

clusters correspond to small compact subregions (some of which are omitted from this zoomed detail of the entire 3D visualization). The result, shown in **Figure 5-2**, does not necessarily advance the analysis but is an interesting sanity check.

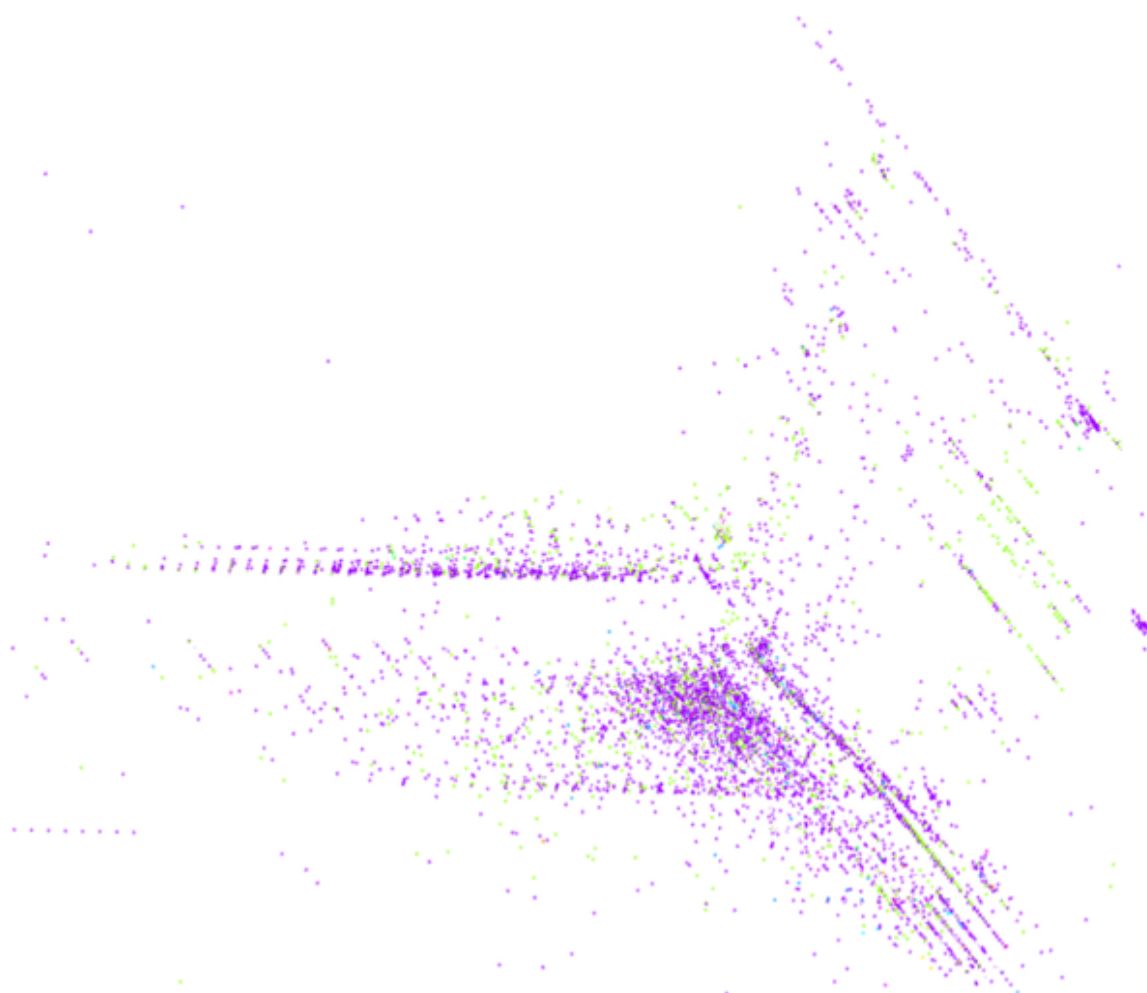


Figure 5-2. Random 3D projection, normalized

Categorical Variables

Normalization was a valuable step forward, but more can be done to improve the clustering. In particular, several features have been left out entirely because they aren't numeric. This is throwing away valuable information. Adding them back in some form should produce a better-informed clustering.

Earlier, three categorical features were excluded because nonnumeric features can't be used with the Euclidean distance function that K-means uses in MLlib. This is the reverse of the issue noted in “Random Forests”, where numeric features were used to represent categorical values but a categorical feature was desired.

The categorical features can translate into several binary indicator features using one-hot encoding, which can be viewed as numeric dimensions. For example, the second column contains the protocol type: `tcp`, `udp`, or `icmp`. This feature could be thought of as *three* features, as if features “is TCP,” “is UDP,” and “is ICMP” were in the data set. The single feature value `tcp` might become `1, 0, 0`; `udp` might be `0, 1, 0`; and so on.

Here again, MLlib provides components that implement this transformation. In fact, one-hot-encoding string-valued features like `protocol_type` is actually a two-step process. First, the string values are converted to integer indices like 0, 1, 2, and so on using `StringIndexer`. Then these integer indices are encoded into a vector with `OneHotEncoder`. These two steps can be thought of as a small `Pipeline` in themselves.

```
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 Return Pipeline and name of output vector column

This method produces a `Pipeline` that can be added as a component in the overall clustering pipeline; pipelines can be composed. All that is left is to make sure to add the new vector output columns into `VectorAssembler`'s output and proceed as before with scaling,

clustering, and evaluation. The source code is omitted for brevity here, but can be found in the repository accompanying this chapter.

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

These sample results suggest, possibly, $k=180$ as a value where the score flattens out a bit. At least the clustering is now using all input features.

Using Labels with Entropy

Earlier, we used the given label for each data point to create a quick sanity check of the quality of the clustering. This notion can be formalized further and used as an alternative means of evaluating clustering quality, and therefore, of choosing k .

The labels tell us something about the true nature of each data point. A good clustering, it seems, should agree with these human-applied labels. It should put together points that share a label frequently and not lump together points of many different labels. It should produce clusters with relatively homogeneous labels.

You may recall from “Random Forests” that we have metrics for homogeneity: Gini impurity and entropy. These are functions of the proportions of labels in each cluster, and produce a number that is low when the proportions are skewed toward few, or one, label. Entropy will be used here for illustration.

```
from math import log

def entropy(counts):
    values = [v for v in values if (v > 0)]
    n = sum(values)
```

```
p = [v/n for v in values]
return sum([-1*(p_v) * log(p_v) for p_v in p])
```

A good clustering would have clusters whose collections of labels are homogeneous and so have low entropy. A weighted average of entropy can therefore be used as a cluster score:

```
from pyspark.sql.functions import collect_list

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

weighted_cluster_entropy =
cluster_label.groupBy(col("cluster")).agg(collect_list("label")).
\ ❷

sum(weighted_cluster_entropy) / data.count() ❸ ❹
```

- ❶ Predict cluster for each datum
- ❷ Extract collections of labels, per cluster
- ❸ Count labels in collections
- ❹ Average entropy weighted by cluster size

As before, this analysis can be used to obtain some idea of a suitable value of k . Entropy will not necessarily decrease as k increases, so it is possible to look for a local minimum value. Here again, results suggest $k=180$ is a reasonable choice because its score is actually lower than 150 *and* 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)
```

Clustering in Action

Finally, with confidence, we can cluster the full normalized data set with $k=180$. Again, we can print the labels for each cluster to get some sense of the resulting clustering. Clusters do seem to be dominated by one type of attack each, and contain only a few types.

```
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 See accompanying source code for `fit_pipeline_4()` definition

Now we can make an actual anomaly detector. Anomaly detection amounts to measuring a new data point's distance to its nearest centroid. If this distance exceeds some threshold, it is anomalous. This threshold might be chosen to be the distance of, say, the 100th-farthest data point from among known data:

```
from pyspark.sql.linalg import Vector, Vectors

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

```
clustered = pipeline_model.transform(data)
threshold = clustered.select("cluster",
    "scaledFeatureVector").withColumn("dist_values",
    Vectors.squared_distance(centroids(col("cluster")),
    col("scaledFeatureVector"))).orderBy(col("value").desc()).last
```

The final step is to apply this threshold to all new data points as they arrive. For example, Spark Streaming can be used to apply this function to small batches of input data arriving from sources like Flume, Kafka, or files on HDFS. Data points exceeding the threshold might trigger an alert that sends an email or updates a database.

As an example, we will apply it to the original data set, to see some of the data points that are, we might believe, most anomalous within the input.

```
val originalCols = data.columns
val anomalies = clustered.filter { row =>
    val cluster = row.getAs[Int]("cluster")
    val vec = row.getAs[Vector]("scaledFeatureVector")
    Vectors.sqdist(centroids(cluster), vec) >= threshold
}.select(originalCols.head, originalCols.tail:_* ) ❶

anomalies.first() ❷

...
[9,tcp,telnet,SF,307,2374,0,0,1,0,0,1,0,1,0,1,3,1,0,0,0,0,1,1,
0.0,0.0,0.0,0.0,1.0,0.0,0.0,69,4,0.03,0.04,0.01,0.75,0.0,0.0,
0.0,0.0,normal.]
```

- ❶ Note odd (String, String*) signature for selecting columns
- ❷ show() works; hard to read

This example shows a slightly different way of operating on data frames. Pure SQL can't express the computation of squared distance. A UDF could be used, as before, to define a function of two columns that returns a squared distance. However, it's also possible to interact with rows of data programmatically as a Row object, much like in JDBC.

A network security expert would be more able to interpret why this is or is not actually a strange connection. It appears unusual at least because it is

labeled normal, but involves connections to 69 different hosts.

Where to Go from Here

The `KMeansModel` is, by itself, the essence of an anomaly detection system. The preceding code demonstrated how to apply it to data to detect anomalies. This same code could be used within **Spark Streaming** to score new data as it arrives in near real time, and perhaps trigger an alert or review.

MLlib also includes a variation called `StreamingKMeans`, which can update a clustering incrementally as new data arrives in a `StreamingKMeansModel`. We could use this to continue to learn, approximately, how new data affects the clustering, and not just to assess new data against existing clusters. It can be integrated with Spark Streaming as well. However, it has not been updated for the new DataFrame-based APIs.

This model is only a simplistic one. For example, Euclidean distance is used in this example because it is the only distance function supported by Spark MLlib at this time. In the future, it may be possible to use distance functions that can better account for the distributions of and correlations between features, such as the **Mahalanobis distance**.

There are also more sophisticated **cluster-quality evaluation metrics** that could be applied (even without labels) to pick k , such as the **Silhouette coefficient**. These tend to evaluate not just closeness of points within one cluster, but closeness of points to other clusters. Finally, different models could be applied instead of simple K-means clustering; for example, a **Gaussian mixture model** or **DBSCAN** could capture more subtle relationships between data points and the cluster centers. Spark MLlib already implements **Gaussian mixture models**; implementations of others may become available in Spark MLlib or other Spark-based libraries in the future.

Of course, clustering isn't just for anomaly detection. In fact, it's more often associated with use cases where the actual clusters matter! For example, clustering can also be used to group customers according to their behaviors, preferences, and attributes. Each cluster, by itself, might represent a usefully distinguishable type of customer. This is a more data-driven way to segment customers rather than leaning on arbitrary, generic divisions like "age 20–34" and "female."

Chapter 6. Geospatial and Temporal Data Analysis on New York City Taxi Trip Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book.

Geospatial data refers to data that has location information embedded in it in some form. Such data is being generated currently at a massive scale by billions of sources such as mobile phones and sensors every day. Data about movement of humans and machines, and remote sensing detects, are significant for our economy, and general well-being. Geospatial analytics can provide us with the tools and methods we need to make sense of all that data and put it to use in solving problems we face at all scales.

PySpark and PyData ecosystems have evolved considerably over the last few years when it comes to geospatial analysis. They are being used across industries for handling location-rich data and in turn, impacting our daily lives. One daily activity where geospatial data manifests itself in a visible way is local transport. The phenomenon of digital cab hailing services getting popular over the last few years has led to us being more aware of geospatial technology. In this chapter, we will use our PySpark and data analysis skills in this domain. We will work with a dataset containing information about trips taken by cabs in New York City.

One statistic that is important to understanding the economics of taxis is *utilization*: the fraction of time that a cab is on the road and is occupied by one or more passengers. One factor that impacts utilization is the passenger's destination: a cab that drops off passengers near Union Square at midday is much more likely to find its next fare in just a minute or two, whereas a cab that drops someone off at 2AM on Staten Island may have to drive all the way back to Manhattan before it finds its next fare. We'd like to quantify these effects and find out the average time it takes for a cab to find its next fare as a function of the borough in which it dropped its passengers off—Manhattan, Brooklyn, Queens, the Bronx, Staten Island, or none of the above (e.g., if it dropped the passenger off somewhere outside of the city, like Newark International Airport).

To carry out this analysis, we need to deal with *temporal data*, such as dates and times, in addition to *geospatial information*, like points of longitude and latitude and spatial boundaries. PySpark's DataFrame API provides out-of-the-box data types and methods to handle temporal data. We will start by setting up our dataset. Then we will dive into geospatial analysis. We will learn about the GeoJSON format and use tools from the PyData ecosystem in combination with PySpark. To wrap things up, we will perform sessionization and understand utilization of New York City cabs. Let's get going by downloading our dataset and exploring it using PySpark. That's what we will do in the next section.

ORIGIN STORY OF THE NEW YORK CITY TAXI TRIP DATASET

New York City is widely known for its yellow taxis, and hailing one is just as much a part of the experience of visiting the city as eating a hot dog from a street vendor or riding the elevator to the top of the Empire State Building.

Residents of New York City have all kinds of tips based on their anecdotal experiences about the best times and places to catch a cab, especially during rush hour and when it's raining. But there is one time of day when everyone will recommend that you simply take the subway instead: during the shift change that happens between 4 and 5PM every day. During this time, yellow taxis have to return to their dispatch centers (often in Queens) so that one driver can quit for the day and the next one can start, and drivers who are late to return have to pay fines.

In March of 2014, the New York City Taxi and Limousine Commission shared an infographic on its Twitter account, [@nyctaxi](#), that showed the number of taxis on the road and the fraction of those taxis that was occupied at any given time. Sure enough, there was a noticeable dip of taxis on the road from 4 to 6PM, and two-thirds of the taxis that were driving were occupied.

This tweet caught the eye of self-described urbanist, mapmaker, and data junkie Chris Whong, who sent a tweet to the [@nyctaxi](#) account to find out if the data it used in its infographic was publicly available. The taxi commission replied that he could have the data if he filed a Freedom of Information Law (FOIL) request and provided the commission with hard drives that they could copy the data on to. After filling out one PDF form, buying two new 500 GB hard drives, and waiting two business days, Chris had access to all of the data on taxi rides from January 1 through December 31, 2013. Even better, he posted all of the fare data online, where it has been used as the basis for a number of beautiful visualizations of transportation in New York City. This is the dataset that we will leverage as well in this chapter.

Preparing the data

For this analysis, we're only going to consider the fare data from January 2013, which will be about 2.5 GB of data after we uncompress it. You can access the [data for each month of 2013](#), and if you have a sufficiently large PySpark cluster at your disposal, you can re-create the following analysis against all of the data for the year. For now, let's create a working directory on our client machine and take a look at the structure of the fare data:

```
$ 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 10 trip_data_1.csv
```

Each row of the file after the header represents a single taxi ride in CSV format. For each ride, we have some attributes of the cab (a hashed version of the medallion number) as well as the driver (a hashed version of the *hack license*, which is what licenses to drive taxis are called), some temporal information about when the trip started and ended, and the longitude/latitude coordinates for where the passenger(s) were picked up and dropped off.

Let's create a *taxidata* directory in HDFS and copy the trip data into the cluster:

```
$ hadoop fs -mkdir taxidata
$ hadoop fs -put trip_data_1.csv taxidata/
```

Now start the PySpark shell:

```
$ pyspark
```

Once the PySpark shell has loaded, we can create a data set from the taxi data and examine the first few lines, just as we have in other chapters:

```
val taxiRaw = pyspark.read.option("header",
    "true").csv("taxidata")
taxiRaw.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

...
```

This looks like a well-formatted dataset at first glance. Let's have a look at the DataFrame's schema.

```
taxiRaw.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)
...
```

We are representing the `pickup_datetime` and `dropoff_datetime` fields as `Strings`, and storing the individual xy coordinates of the pickup and dropoff locations in their own fields as `Doubles`. We want the datetime fields as timestamps since that will allow us to manipulate and analyse them conveniently.

Converting datetime strings to timestamps

As mentioned previously, PySpark provides out-of-the-box methods for handling temporal data. Specifically, we will use the `to_timestamp` function to parse the datetime strings and convert them into timestamps.

```
from pyspark.sql import functions as fun

taxiRaw = taxiRaw.withColumn('pickup_datetime',
fun.to_timestamp(fun.col('pickup_datetime'), "yyyy-MM-dd
HH:mm:ss"))
taxiRaw = taxiRaw.withColumn('dropoff_datetime',
fun.to_timestamp(fun.col('dropoff_datetime'), "yyyy-MM-dd
HH:mm:ss"))
```

Let's have a look at the schema again.

```
taxiRaw.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)
```

```
...
```

The `pickup_datetime` and `dropoff_datetime` fields are timestamps now. Well done!

We'd mentioned that this dataset contains trips from January 2013. Don't just take our word for this though. We can confirm this by sorting the `pickup_datetime` field to get the latest datetime in the data. For this, we use DataFrame's `sort` method combined with PySpark column's `desc` method.

```
taxiRaw.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
...

```

With our datatypes in place, let's check if there are any inconsistencies in our data.

Handling Invalid Records

Anyone who has been working with large-scale, real-world data sets knows that they invariably contain at least a few records that do not conform to the expectations of the person who wrote the code to handle them. Many PySpark pipelines have failed because of invalid records that caused the parsing logic to throw an exception. When performing interactive analysis, we can get a sense of potential anomalies in the data by focusing on key variables.

In our case, variables containing geospatial and temporal information are worth looking at for inconsistencies. Presence of null values in these columns will definitely throw off our analysis.

```

geospatial_temporal_colnames = ["pickup_longitude",
"pickup_latitude", "dropoff_longitude", "dropoff_latitude",
"pickup_datetime", "dropoff_datetime"]
taxiRaw.select([fun.count(fun.when(fun.isNull(c), c)).alias(c)
for c in geospatial_temporal_colnames]).show()
...

+-----+-----+-----+-----+
|pickup_longitude|pickup_latitude|dropoff_longitude|dropoff_latitude|
|pickup_datetime|dropoff_datetime|
+-----+-----+-----+-----+
|              0|              0|              86|
86|              0|              0|
+-----+-----+-----+-----+

```

Let's remove the null values from our data.

```

taxiRaw = taxiRaw.na.drop(subset=geospatial_temporal_colnames)

```

Another common sense check that we can is for latitude and longitude records where the values are zero. We know that for the region we're concerned with, those would be invalid values.

```

print("Count of zero pickup latitude and longitude records")
taxiRaw.groupBy((fun.col("pickup_longitude") == 0) &
(fun.col("pickup_latitude") == 0)).count().show()

print("Count of zero dropoff latitude and longitude records")
taxiRaw.groupBy((fun.col("dropoff_longitude") == 0) &
(fun.col("dropoff_latitude") == 0)).count().show()

print("Count of zero dropoff, pickoff latitude and longitude
records")
taxiRaw.groupBy((fun.col("dropoff_longitude") == 0) &
(fun.col("dropoff_latitude") == 0) & (fun.col("pickup_longitude")
== 0) & (fun.col("pickup_latitude") == 0)).count().show()
...

```

Count of zero pickup latitude and longitude records

```

+-----+-----+
|((pickup_longitude = 0) AND (pickup_latitude = 0))| count|

```

```
+-----+
|                                     true| 264890|
|                                     false|14511639|
+-----+
```

Count of zero dropoff latitude **and** longitude records

```
+-----+
|(((dropoff_longitude = 0) AND (dropoff_latitude = 0))| count|
+-----+
|                                     true| 273160|
|                                     false|14503369|
+-----+
```

Count of zero dropoff, pickoff latitude **and** longitude records

```
+-----+
+-----+
|((((dropoff_longitude = 0) AND (dropoff_latitude = 0)) AND
(pickup_longitude = 0)) AND (pickup_latitude = 0))| count|
+-----+
+-----+
|
true| 256791|
|
false|14519738|
+-----+
+-----+
```

We have quite a few of these. If it looks as if a taxi took a passenger to the South Pole, we can be reasonably confident that the record is invalid and should be excluded from our analysis.

In production settings, we handle these exceptions one at a time by checking the logs for the individual tasks, figuring out which line of code threw the exception, and then figuring out how to tweak the code to ignore or correct the invalid records. This is a tedious process, and it often feels like we're playing whack-a-mole: just as we get one exception fixed, we discover another one on a record that came later within the partition.

One strategy that experienced data scientists deploy when working with a new data set is to add a `try-catch` block to their parsing code so that any invalid records can be written out to the logs without causing the entire job to fail. If there are only a handful of invalid records in the entire data set, we might be okay with ignoring them and continuing with our analysis.

Now that we have prepared our dataset, let's get started with geospatial analysis.

Geospatial Analysis

There are two major kinds of geospatial data—vector and raster—and there are different tools for working with each type. In our case, we have latitude and longitude for our taxi trip records, and vector data stored in the GeoJSON format that represents the boundaries of the different boroughs of New York. We've looked at the latitude and longitude points. Let's start by having a look at the GeoJSON data.

Intro to GeoJSON

The data we'll use for the boundaries of boroughs in New York City comes written in a format called *GeoJSON*. The core object in GeoJSON is called a *feature*, which is made up of a *geometry* instance and a set of key-value pairs called *properties*. A geometry is a shape like a point, line, or polygon. A set of features is called a **FeatureCollection**. Let's pull down the GeoJSON data for the NYC borough maps and take a look at its structure.

In the *taxidata* directory on your client machine, download the data and rename the file to something a bit shorter:

```
$ curl -O https://nycdatastables.s3.amazonaws.com/2013-08-19T18:15:35.172Z/nyc-borough-boundaries-polygon.geojson
$ mv nyc-borough-boundaries-polygon.geojson nyc-boroughs.geojson
```

Open the file and look at a feature record. Note the properties and the geometry objects—in this case, a polygon representing the boundaries of the borough, and the properties containing the name of the borough and other related information.

```
$ head -n 7 data/trip_data_ch07/nyc-boroughs.geojson
...
{
  "type": "FeatureCollection",
```

```
"features": [{ "type": "Feature", "id": 0, "properties": {  
  "boroughCode": 5, "borough": "Staten Island", "@id":  
  "http:\\\\nyc.pediacities.com\\Resource\\Borough\\Staten_Island"  
}, "geometry": { "type": "Polygon", "coordinates": [ [ [   
-74.050508064032471, 40.566422034160816 ], [ -74.049983525625748,  
40.566395924928273 ], [ -74.049316403620878, 40.565887747780437  
], [ -74.049236298420453, 40.565362736368101 ], [   
-74.050026201586434, 40.565318180621134 ], [ -74.050906017050892,  
40.566094342130597 ], [ -74.050679167486138, 40.566310845736403  
], [ -74.05107159803778, 40.566722493397798 ], [   
-74.050508064032471, 40.566422034160816 ] ] ] } }  
,  
{ "type": "Feature", "id": 1, "properties": { "boroughCode": 5,  
  "borough": "Staten Island", "@id":  
  "http:\\\\nyc.pediacities.com\\Resource\\Borough\\Staten_Island"  
}, "geometry": { "type": "Polygon", "coordinates": [ [ [   
-74.053140368211089, 40.577702715545755 ], [ -74.054060449398747,  
40.577116445238872 ], [ -74.054897782108043, 40.577782440919812  
], [ -74.054693169074866, 40.579691632229434 ], [   
-74.054851346253912, 40.579707593077607 ], [ -74.054845600521745,  
40.579945791427605 ], [ -74.053686291504192, 40.580548759614409  
], [ -74.052931906398172, 40.57990247466585 ], [   
-74.053140368211089, 40.577702715545755 ] ] ] } }
```

GEOSPATIAL DATA FORMATS

There are different specialized geospatial formats. In addition, data sources such as geotagged logs and text data can also be used to harvest location data. These include:

- Vector formats such as GeoJSON, KML, Shapefile, and WKT
- Raster formats such as ESRI Grid, GeoTIFF, JPEG 2000, and NITF
- Navigational standards such as used by AIS and GPS devices
- Geodatabases accessible via JDBC / ODBC connections such as PostgreSQL / PostGIS
- Remote sensor formats from Hyperspectral, Multispectral, Lidar, and Radar platforms
- OGC web standards such as WCS, WFS, WMS, and WMTS
- Geotagged logs, pictures, videos, and social media
- Unstructured data with location references

Geopandas

The first thing you must consider when choosing a library is determine what kind of geospatial data you will need to work with. So we need a library that can parse GeoJSON data and can handle spatial relationships, like detecting whether a given longitude/latitude pair is contained inside a polygon that represents the boundaries of a particular borough. We will use the **Geopandas** library for this task. GeoPandas is an open source project to make working with geospatial data in python easier. It extends the datatypes used by the pandas library, which we used in previous chapters, to allow spatial operations on geometric data types.

Let's start examining the geospatial aspects of the taxi data. For each trip, we have longitude/latitude pairs representing where the passenger was picked up and dropped off. We would like to be able to determine which borough each of these longitude/latitude pairs belongs to, and identify any trips that did not start or end in any of the five boroughs. For example, if a taxi took passengers from Manhattan to Newark International Airport, that would be a valid ride that would be interesting to analyze, even though it would not end within one of the five boroughs.

To perform our borough analysis, we need to load the GeoJSON data we downloaded earlier and stored in the *nyc-boroughs.geojson* file.

TO UDF OR NOT TO UDF?

PySpark SQL makes it very easy to inline business logic into functions that can be used from standard SQL, as we did here with the `hours` function. Given this, you might think that it would be a good idea to move all of your business logic into UDFs in order to make it easy to reuse, test, and maintain. However, there are a few caveats for using UDFs that you should be mindful of before you start sprinkling them throughout your code.

First, UDFs are opaque to PySpark's SQL query planner and execution engine in a way that standard SQL query syntax is not, so moving logic into a UDF instead of using a literal SQL expression could hurt query performance.

Second, handling null values in PySpark SQL can get complicated quickly, especially for UDFs that take multiple arguments. To properly handle nulls, you need to use python's `Option[T]` type or write your UDFs using the Java wrapper types, like `java.lang.Integer` and `java.lang.Double`, instead of the primitive types `Int` and `Double` in python.

Before we use the `features` on the taxi trip data, we should take a moment to think about how to organize this geospatial data for maximum

efficiency. One option would be to research data structures that are optimized for geospatial lookups, such as quad trees, and then find or write our own implementation. But let's see if we can come up with a quick heuristic that will allow us to bypass that bit of work.

The `find` method will iterate through the `FeatureCollection` until it finds a feature whose geometry contains the given `Point` of longitude/latitude. Most taxi rides in NYC begin and end in Manhattan, so if the geospatial features that represent Manhattan are earlier in the sequence, most of the `find` calls will return relatively quickly. We can use the fact that the `boroughCode` property of each feature can be used as a sorting key, with the code for Manhattan equal to 1 and the code for Staten Island equal to 5. Within the features for each borough, we want the features associated with the largest polygons to come before the smaller polygons, because most trips will be to and from the “major” region of each borough. Sorting the features by the combination of the borough code and the `area2D()` of each feature's geometry should do the trick:

```
val areaSortedFeatures = features.sortBy(f => {  
    val borough = f("boroughCode").convertTo[Int]  
    (borough, -f.geometry.area2D())  
})
```

Note that we're sorting based on the negation of the `area2D()` value because we want the largest polygons to come first, and python sorts in ascending order by default.

Now we can broadcast the sorted features in the `areaSortedFeatures` sequence to the cluster and write a function that uses these features to find out in which of the five boroughs (if any) a particular trip ended:

```
val bFeatures = sc.broadcast(areaSortedFeatures)  
  
val bLookup = (x: Double, y: Double) => {  
    val feature: Option[Feature] = bFeatures.value.find(f => {  
        f.geometry.contains(new Point(x, y))  
    })  
    feature.map(f => {
```



```

        f("borough").convertTo[String]
    }).getOrElse("NA")
}
val boroughUDF = udf(bLookup)

```

We can apply `boroughUDF` to the trips in the `taxiClean` RDD to create a histogram of trips by borough:

```

taxiClean.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()

```

```

...
+-----+-----+
|UDF(dropoffX, dropoffY)|    count|
+-----+-----+
|                Queens|    672192|
|                 NA|    7942421|
|             Brooklyn|    715252|
|          Staten Island|     3338|
|             Manhattan|   12979047|
|                 Bronx|     67434|
+-----+-----+

```

As we expected, the vast majority of trips end in the borough of Manhattan, while relatively few trips end in Staten Island. One surprising observation is the number of trips that end outside of any borough; the number of `NA` records is substantially larger than the number of taxi rides that end in the Bronx. Let's grab some examples of this kind of trip from the data:

```

taxiClean.
  where(boroughUDF($"dropoffX", $"dropoffY") === "NA").
  show()

```

When we print out these records, we see that a substantial fraction of them start and end at the point `(0.0, 0.0)`, indicating that the trip location is missing for these records. We should filter these events out of our data set because they won't help us with our analysis:

```

val taxiDone = taxiClean.where(
  "dropoffX != 0 and dropoffY != 0 and pickupX != 0 and pickupY

```

```
!= 0"
).cache()
```

When we rerun our borough analysis on the `taxiDone` RDD, we see this:

```
taxiDone.
  groupBy(boroughUDF($"dropoffX", $"dropoffY")).
  count().
  show()
```

```
...
+-----+-----+
|UDF(dropoffX, dropoffY)|count|
+-----+-----+
|                Queens|670912|
|                 NA|62778|
|             Brooklyn|714659|
|          Staten Island|3333|
|             Manhattan|12971314|
|                 Bronx|67333|
+-----+-----+
```

Our zero-point filter removed a small number of observations from the output boroughs, but it removed a large fraction of the `NA` entries, leaving a much more reasonable number of observations that had dropoffs outside the city.

Sessionization in PySpark

This kind of analysis, in which we want to analyze a single entity as it executes a series of events over time, is called *sessionization*, and is commonly performed over web logs to analyze the behavior of the users of a website.

Sessionization can be a very powerful technique for uncovering insights in data and building new data products that can be used to help people make better decisions. For example, Google's spell-correction engine is built on top of the sessions of user activity that Google builds each day from the logged records of every event (searches, clicks, maps visits, etc.) occurring on its web properties. To identify likely spell-correction candidates, Google

processes those sessions looking for situations where a user typed a query, didn't click anything, typed a slightly different query a few seconds later, and then clicked a result and didn't come back to Google. Then it counts how often this pattern occurs for any pair of queries. If it occurs frequently enough (e.g., if every time we see the query "untied stats," it's followed a few seconds later by the query "united states"), then we assume that the second query is a spell correction of the first.

This analysis takes advantage of the patterns of human behavior that are represented in the event logs to build a spell-correction engine from data that is more powerful than any engine that could be created from a dictionary. The engine can be used to perform spell correction in any language, and can correct words that might not be included in any dictionary (e.g., the name of a new startup), and can even correct queries like "untied stats" where none of the words are misspelled! Google uses similar techniques to show recommended and related searches, as well as to decide which queries should return a OneBox result that gives the answer to a query on the search page itself, without requiring that the user click through to a different page. There are OneBoxes for weather, scores from sports games, addresses, and lots of other kinds of queries.

So far, information about the set of events that occurs to each entity is spread out across the RDD's partitions, so, for analysis, we need to place these relevant events next to each other and in chronological order. In the next section, we'll show how to efficiently construct and analyze sessions using advanced functionality that was introduced in PySpark 3.0.

Building Sessions: Secondary Sorts in PySpark

The naive way to create sessions in PySpark is to perform a `groupBy` on the identifier we want to create sessions for and then sort the events postshuffle by a timestamp identifier. If we only have a small number of events for each entity, this approach will work reasonably well. However, because this approach requires all the events for any particular entity to be in memory at the same time, it will not scale as the number of events for each entity gets larger and larger. We need a way of building sessions that

does not require all of the events for a particular entity to be held in memory at the same time for sorting.

In MapReduce, we can build sessions by performing a *secondary sort*, where we create a composite key made up of an identifier and a timestamp value, sort all of the records on the composite key, and then use a custom partitioner and grouping function to ensure that all of the records for the same identifier appear in the same output partition. Fortunately, PySpark can also support this same secondary sort pattern by combining its `repartition` and a `sortWithinPartitions` transformation; in PySpark 3.0, sessionizing a data set can be done in three lines of code:

```
val sessions = taxiDone.  
  repartition($"license").  
  sortWithinPartitions($"license", $"pickupTime")
```

First, we use the `repartition` method to ensure that all of the `Trip` records that have the same value for the `license` column end up in the same partition. Then, within each of these partitions, we sort the records by their `license` value (so all trips by the same driver appear together) and then by their `pickupTime`, so that the sequence of trips appear in sorted order within the partition. Now when we process the trip records using a method like `mapPartitions`, we can be sure that the trips are ordered in a way that is optimal for sessions analysis. Because this operation triggers a shuffle and a fair bit of computation, and we'll need to use the results more than once, we cache them:

```
sessions.cache()
```

Executing a sessionization pipeline is an expensive operation, and the sessionized data is often useful for many different analysis tasks that we might want to perform. In settings where one might want to pick up on the analysis later or collaborate with other data scientists, it's a good idea to amortize the cost of sessionizing a large data set by only performing the sessionization once, and then writing the sessionized data to HDFS so that it can be used to answer lots of different questions. Performing sessionization

once is also a good way to enforce standard rules for session definitions across the entire data science team, which has the same benefits for ensuring apples-to-apples comparisons of results.

At this point, we are ready to analyze our sessions data to see how long it takes for a driver to find his next fare after a dropoff in a particular borough. We will create a `boroughDuration` method that takes two instances of the `Trip` class and computes both the borough of the first trip and the duration in seconds between the dropoff time of the first trip and the pickup time of the second:

```
def boroughDuration(t1: Trip, t2: Trip): (String, Long) = {  
  val b = bLookup(t1.dropoffX, t1.dropoffY)  
  val d = (t2.pickupTime - t1.dropoffTime) / 1000  
  (b, d)  
}
```

We want to apply our new function to all sequential pairs of trips inside our `sessions` data set. Although we could write a `for` loop to do this, we can also use the `sliding` method of the python Collections API to get the sequential pairs in a more functional way:

```
val boroughDurations: DataFrame =  
  sessions.mapPartitions(trips => {  
    val iter: Iterator[Seq[Trip]] = trips.sliding(2)  
    val viter = iter.  
      filter(_.size == 2).  
      filter(p => p(0).license == p(1).license)  
    viter.map(p => boroughDuration(p(0), p(1)))  
  }).toDF("borough", "seconds")
```

The `filter` call on the result of the `sliding` method ensures that we ignore any sessions that contain only a single trip, or any trip pairs that have different values of the `license` field. The result of our `mapPartitions` over the sessions is a data frame of borough/duration pairs that we can now examine. First, we should do a validation check to ensure that most of the durations are nonnegative:

```
boroughDurations.
  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 |
```

Only a few of the records have a negative duration, and when we examine them more closely, there don't seem to be any common patterns to them that we could use to understand the source of the erroneous data. If we exclude these negative duration records from our input data set and look at the average and standard deviation of the pickup times by borough, we see this:

```
boroughDurations.
  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 |
+-----+-----+-----+
```

As we would expect, the data shows that dropoffs in Manhattan have the shortest amount of downtime for drivers, at around 10 minutes. Taxi rides that end in Brooklyn have a downtime of more than twice that, and the relatively few rides that end in Staten Island take a driver an average of almost 45 minutes to get to his next fare.

As the data demonstrates, taxi drivers have a major financial incentive to discriminate among passengers based on their final destination; dropoffs in Staten Island, in particular, involve an extensive amount of downtime for a driver. The NYC Taxi and Limousine Commission has made a major effort over the years to identify this discrimination and has fined drivers who have been caught rejecting passengers because of where they wanted to go. It would be interesting to attempt to examine the data for unusually short taxi rides that could be indicative of a dispute between the driver and the passenger about where the passenger wanted to be dropped off.

Where to Go from Here

Imagine using this same technique on the taxi data to build an application that could recommend the best place for a cab to go after a dropoff based on current traffic patterns and the historical record of next-best locations contained within this data. You could also look at the information from the perspective of someone trying to catch a cab: given the current time, place, and weather data, what is the probability that I will be able to hail a cab from the street within the next five minutes? This sort of information could be incorporated into applications like Google Maps to help travelers decide when to leave and which travel option they should take.

Chapter 7. Estimating Financial Risk

Sandy Ryza

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book.

Under reasonable circumstances, how much can you expect to lose? This is the quantity that the financial statistic *Value at Risk* (VaR) seeks to measure. Since its development soon after the stock market crash of 1987, VaR has seen widespread use across financial services organizations. The statistic plays a vital role in the management of these institutions by helping to determine how much cash they must hold to meet the credit ratings they seek. In addition, some use it to more broadly understand the risk characteristics of large portfolios, and others compute it before executing trades to help inform immediate decisions.

Many of the most sophisticated approaches to estimating this statistic rely on computationally intensive simulation of markets under random conditions. The technique behind these approaches, called *Monte Carlo simulation*, involves posing thousands or millions of random market scenarios and observing how they tend to affect a portfolio. PySpark is an ideal tool for Monte Carlo simulation, because the technique is naturally massively parallelizable. PySpark can leverage thousands of cores to run random trials and aggregate their results. As a general-purpose data transformation engine, it is also adept at performing the pre- and

postprocessing steps that surround the simulations. It can transform raw financial data into the model parameters needed to carry out the simulations, as well as support ad hoc analysis of the results. Its simple programming model can drastically reduce development time compared to more traditional approaches that use HPC environments.

Let's define "how much can you expect to lose" a little more rigorously. VaR is a simple measure of investment risk that tries to provide a reasonable estimate of the maximum probable loss in value of an investment portfolio over a particular time period. A VaR statistic depends on three parameters: a portfolio, a time period, and a probability. A VaR of \$1 million with a 5% probability and two weeks indicates the belief that the portfolio stands only a 5% chance of losing more than \$1 million over two weeks.

We'll also discuss how to compute a related statistic called *Conditional Value at Risk* (CVaR), sometimes known as expected shortfall, which the Basel Committee on Banking Supervision has recently proposed as a better risk measure than VaR. A CVaR statistic has the same three parameters as a VaR statistic, but considers the expected loss instead of the cutoff value. A CVaR of \$5 million with a 5% *q-value* and two weeks indicates the belief that the average loss in the worst 5% of outcomes is \$5 million.

In service of modeling VaR, we'll introduce a few different concepts, approaches, and packages. We'll cover kernel density estimation and plotting using the *breeze-viz* package, sampling using the multivariate normal distribution, and statistics functions using the Apache Commons Math package.

Terminology

This chapter makes use of a set of terms specific to the finance domain. We'll briefly define them here:

Instrument

A tradable asset, such as a bond, loan, option, or stock investment. At any particular time, an instrument is considered to have a *value*, which is the price for which it could be sold.

Portfolio

A collection of instruments owned by a financial institution.

Return

The change in an instrument or portfolio's value over a time period.

Loss

A negative return.

Index

An imaginary portfolio of instruments. For example, the NASDAQ Composite Index includes about 3,000 stocks and similar instruments for major US and international companies.

Market factor

A value that can be used as an indicator of macroaspects of the financial climate at a particular time—for example, the value of an index, the gross domestic product of the United States, or the exchange rate between the dollar and the euro. We will often refer to market factors as just *factors*.

Methods for Calculating VaR

So far, our definition of VaR has been fairly open-ended. Estimating this statistic requires proposing a model for how a portfolio functions and choosing the probability distribution its returns are likely to take. Institutions employ a variety of approaches for calculating VaR, all of which tend to fall under a few general methods.

Variance-Covariance

Variance-covariance is by far the simplest and least computationally intensive method. Its model assumes that the return of each instrument is normally distributed, which allows deriving a estimate analytically.

Historical Simulation

Historical simulation extrapolates risk from historical data by using its distribution directly instead of relying on summary statistics. For example, to determine a 95% VaR for a portfolio, we might look at that portfolio's performance for the last 100 days and estimate the statistic as its value on the fifth-worst day. A drawback of this method is that historical data can be limited and fails to include what-ifs. For example, what if the history we have for the instruments in our portfolio lacks market collapses, and we want to model what happens to our portfolio in these situations. Techniques exist for making historical simulation robust to these issues, such as introducing "shocks" into the data, but we won't cover them here.

Monte Carlo Simulation

Monte Carlo simulation, which the rest of this chapter will focus on, tries to weaken the assumptions in the previous methods by simulating the portfolio under random conditions. When we can't derive a closed form for a probability distribution analytically, we can often estimate its probability density function (PDF) by repeatedly sampling simpler random variables that it depends on and seeing how it plays out in aggregate. In its most general form, this method:

- Defines a relationship between market conditions and each instrument's returns. This relationship takes the form of a model fitted to historical data.
- Defines distributions for the market conditions that are straightforward to sample from. These distributions are fitted to historical data.

- Poses *trials* consisting of random market conditions.
- Calculates the total portfolio loss for each trial, and uses these losses to define an empirical distribution over losses. This means that, if we run 100 trials and want to estimate the 5% VaR, we would choose it as the loss from the trial with the fifth-greatest loss. To calculate the 5% CVaR, we would find the average loss over the five worst trials.

Of course, the Monte Carlo method isn't perfect either. It relies on models for generating trial conditions and for inferring instrument performance, and these models must make simplifying assumptions. If these assumptions don't correspond to reality, then neither will the final probability distribution that comes out.

Our Model

A Monte Carlo risk model typically phrases each instrument's return in terms of a set of *market factors*. Common market factors might be the value of indexes like the S&P 500, the US GDP, or currency exchange rates. We then need a model that predicts the return of each instrument based on these market conditions. In our simulation, we'll use a simple linear model. By our previous definition of return, a *factor return* is a change in the value of a market factor over a particular time. For example, if the value of the S&P 500 moves from 2,000 to 2,100 over a time interval, its return would be 100. We'll derive a set of features from simple transformations of the factor returns. That is, the market factor vector m_t for a trial t is transformed by some function ϕ to produce a feature vector of possible different length f_t :

$$f_t = \phi(m_t)$$

For each instrument, we'll train a model that assigns a weight to each feature. To calculate r_{it} , the return of instrument i in trial t , we use c_i , the intercept term for the instrument; w_{ij} , the regression weight for feature j on instrument i ; and f_{tj} , the randomly generated value of feature j in trial t :

$$r_{it} = c_i + \sum_{j=1}^{|w_i|} w_{ij} * f_{tj}$$

This means that the return of each instrument is calculated as the sum of the returns of the market factor features multiplied by their weights for that instrument. We can fit the linear model for each instrument using historical data (also known as doing linear regression). If the horizon of the VaR calculation is two weeks, the regression treats every (overlapping) two-week interval in history as a labeled point.

It's also worth mentioning that we could have chosen a more complicated model. For example, the model need not be linear: it could be a regression tree or explicitly incorporate domain-specific knowledge.

Now that we have our model for calculating instrument losses from market factors, we need a process for simulating the behavior of market factors. A simple assumption is that each market factor return follows a normal distribution. To capture the fact that market factors are often correlated—when NASDAQ is down, the Dow is likely to be suffering as well—we can use a multivariate normal distribution with a non-diagonal covariance matrix:

$$m_t \sim \mathcal{N}(\mu, \Sigma)$$

where μ is a vector of the empirical means of the returns of the factors and Σ is the empirical covariance matrix of the returns of the factors.

As before, we could have chosen a more complicated method of simulating the market or assumed a different type of distribution for each market factor, perhaps using distributions with fatter tails.

Getting the Data

It can be difficult to find large volumes of nicely formatted historical price data, but Yahoo! has a variety of stock data available for download in CSV format. The following script, located in the *risk/data* directory of the repo, will make a series of REST calls to download histories for all the stocks included in the NASDAQ index and place them in a *stocks/* directory:

```
$ ./download-all-symbols.sh
```

We also need historical data for risk factors. For our factors, we'll use the values of the S&P 500 and NASDAQ indexes, as well as the prices of 5-year and 30-year US Treasury bonds. These can all be downloaded from Yahoo! as well:

```
$ mkdir factors/  
$ ./download-symbol.sh ^GSPC factors  
$ ./download-symbol.sh ^IXIC factors  
$ ./download-symbol.sh ^TYX factors  
$ ./download-symbol.sh ^FVX factors
```

Preprocessing

The first few rows of the Yahoo!-formatted data for GOOGL looks like:

```
Date,Open,High,Low,Close,Volume,Adj Close  
2014-10-24,554.98,555.00,545.16,548.90,2175400,548.90  
2014-10-23,548.28,557.40,545.50,553.65,2151300,553.65  
2014-10-22,541.05,550.76,540.23,542.69,2973700,542.69  
2014-10-21,537.27,538.77,530.20,538.03,2459500,538.03  
2014-10-20,520.45,533.16,519.14,532.38,2748200,532.38
```

Let's fire up the PySpark shell. In this chapter, we rely on several libraries to make our lives easier. The GitHub repo contains a Maven project that can be used to build a JAR file that packages all these dependencies together:

```
$ cd ch09-risk/  
$ mvn package
```

```
$ cd data/  
$ pyspark-shell --jars ../target/ch09-risk-2.0.0-jar-with-  
dependencies.jar
```

For each instrument and factor, we want to derive a list of (date, closing price) tuples. The `java.time` library contains useful functionality for representing and manipulating dates. We can represent our dates as `LocalDate` objects. We can use the `DateTimeFormatter` to parse dates in the Yahoo date format:

```
import java.time.LocalDate  
import java.time.format.DateTimeFormatter  
  
val format = DateTimeFormatter.ofPattern("yyyy-MM-dd")  
LocalDate.parse("2014-10-24")  
res0: java.time.LocalDate = 2014-10-24
```

The 3,000-instrument histories and 4-factor histories are small enough to read and process locally. This remains the case even for larger simulations with hundreds of thousands of instruments and thousands of factors. The need arises for a distributed system like PySpark when we're actually running the simulations, which can require massive amounts of computation on each instrument.

To read a full Yahoo history from local disk:

```
import java.io.File  
  
def readYahooHistory(file: File): Array[(LocalDate, Double)] = {  
  val formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd")  
  val lines = python.io.Source.fromFile(file).getLines().toSeq  
  lines.tail.map { line =>  
    val cols = line.split(',')  
    val date = LocalDate.parse(cols(0), formatter)  
    val value = cols(1).toDouble  
    (date, value)  
  }.reverse.toArray  
}
```

Notice that `lines.tail` is useful for excluding the header row. We load all the data and filter out instruments with less than five years of history:

```
val start = LocalDate.of(2009, 10, 23)
val end = LocalDate.of(2014, 10, 23)

val stocksDir = new File("stocks/")
val files = stocksDir.listFiles()
val allStocks = files.iterator.flatMap { file =>> ❶
    try {
        Some(readYahooHistory(file))
    } catch {
        case e: Exception => None
    }
}
val rawStocks = allStocks.filter(_.size >= 260 * 5 + 10)

val factorsPrefix = "factors/"
val rawFactors = Array(
    "^GSPC.csv", "^IXIC.csv", "^TYX.csv", "^FVX.csv").
    map(x => new File(factorsPrefix + x)).
    map(readYahooHistory)
```

- ❶ Using `iterator` here allows us to stream over the files instead of loading their full contents in memory all at once.

Different types of instruments may trade on different days, or the data may have missing values for other reasons, so it is important to make sure that our different histories align. First, we need to trim all of our time series to the same region in time. Then we need to fill in missing values. To deal with time series that are missing values at the start and end dates in the time region, we simply fill in those dates with nearby values in the time region:

```
def trimToRegion(history: Array[(LocalDate, Double)],
    start: LocalDate, end: LocalDate)
    : Array[(LocalDate, Double)] = {
    var trimmed = history.dropWhile(_. _1.isBefore(start)).
        takeWhile(x => x._1.isBefore(end) || x._1.isEqual(end))
    if (trimmed.head._1 != start) {
        trimmed = Array((start, trimmed.head._2)) ++ trimmed
    }
    if (trimmed.last._1 != end) {
```



```

        trimmed = trimmed ++ Array((end, trimmed.last._2))
    }
    trimmed
}

```

To deal with missing values within a time series, we use a simple imputation strategy that fills in an instrument's price as its most recent closing price before that day. Because there is no pretty python Collections method that can do this for us, we write it on our own. The **spark-ts** and **flint** libraries are alternatives that contain an array of useful time series manipulations functions.

```

import python.collection.mutable.ArrayBuffer

def fillInHistory(history: Array[(DateTime, Double)],
    start: DateTime, end: DateTime): Array[(DateTime, Double)] =
{
    var cur = history
    val filled = new ArrayBuffer[(DateTime, Double)]()
    var curDate = start
    while (curDate < end) {
        if (cur.tail.nonEmpty && cur.tail.head._1 == curDate) {
            cur = cur.tail
        }

        filled += ((curDate, cur.head._2))

        curDate += 1.days
        // Skip weekends
        if (curDate.dayOfWeek().get > 5) curDate += 2.days
    }
    filled.toArray
}

```

We apply `trimToRegion` and `fillInHistory` to the data:

```

val stocks = rawStocks.
    map(trimToRegion(_, start, end)).
    map(fillInHistory(_, start, end))

val factors = (factors1 ++ factors2).
    map(trimToRegion(_, start, end)).
    map(fillInHistory(_, start, end))

```

Keep in mind that, even though the python APIs used here look very similar to PySpark's API, these operations are executing locally. Each element of `stocks` is an array of values at different time points for a particular stock. `factors` has the same structure. All these arrays should have equal length, which we can verify with:

```
(stocks ++ factors).forall(_.size == stocks(0).size)
res17: Boolean = true
```

Determining the Factor Weights

Recall that VaR deals with losses *over a particular time horizon*. We are not concerned with the absolute prices of instruments but how those prices move over a given length of time. In our calculation, we will set that length to two weeks. The following function makes use of the python Collections' `sliding` method to transform time series of prices into an overlapping sequence of price movements over two-week intervals. Note that we use 10 instead of 14 to define the window because financial data does not include weekends:

```
def twoWeekReturns(history: Array[(LocalDate, Double)])
  : Array[Double] = {
  history.sliding(10).
    map { window =>
      val next = window.last._2
      val prev = window.head._2
      (next - prev) / prev
    }.toArray
}

val stocksReturns = stocks.map(twoWeekReturns).toArray.toSeq ❶
val factorsReturns = factors.map(twoWeekReturns)
```

- ❶ Because of our earlier use of `iterator`, `stocks` is an iterator. `.toArray.toSeq` runs through it and collects the elements in memory into a sequence.

With these return histories in hand, we can turn to our goal of training predictive models for the instrument returns. For each instrument, we want a model that predicts its two-week return based on the returns of the factors over the same time period. For simplicity, we will use a linear regression model.

To model the fact that instrument returns may be nonlinear functions of the factor returns, we can include some additional features in our model that we derive from nonlinear transformations of the factor returns. We will try adding two additional features for each factor return: square and square root. Our model is still a linear model in the sense that the response variable is a linear function of the features. Some of the features just happen to be determined by nonlinear functions of the factor returns. Keep in mind that this particular feature transformation is meant to demonstrate some of the options available—it shouldn't be perceived as a state-of-the-art practice in predictive financial modeling.

Even though we will be carrying out many regressions—one for each instrument—the number of features and data points in each regression is small, meaning that we don't need to make use of PySpark's distributed linear modeling capabilities. Instead, we'll use the ordinary least squares regression offered by the Apache Commons Math package. While our factor data is currently a `Seq` of histories (each an array of `(DateTime, Double)` tuples), `OLSMultipleLinearRegression` expects data as an array of sample points (in our case a two-week interval), so we need to transpose our factor matrix:

```
def factorMatrix(histories: Seq[Array[Double]])
  : Array[Array[Double]] = {
  val mat = new Array[Array[Double]](histories.head.length)
  for (i <- histories.head.indices) {
    mat(i) = histories.map(_(i)).toArray
  }
  mat
}

val factorMat = factorMatrix(factorsReturns)
```

Then we can tack on our additional features:

```
def featurize(factorReturns: Array[Double]): Array[Double] = {
  val squaredReturns = factorReturns.
    map(x => math.signum(x) * x * x)
  val squareRootedReturns = factorReturns.
    map(x => math.signum(x) * math.sqrt(math.abs(x)))
  squaredReturns ++ squareRootedReturns ++ factorReturns
}

val factorFeatures = factorMat.map(featurize)
```

And then we can fit the linear models. In addition, to find the model parameters for each instrument, we can use the `OLSMultipleLinearRegression`'s `estimateRegressionParameters` method:

```
import
org.apache.commons.math3.stat.regression.OLSMultipleLinearRegression

def linearModel(instrument: Array[Double],
  factorMatrix: Array[Array[Double]])
  : OLSMultipleLinearRegression = {
  val regression = new OLSMultipleLinearRegression()
  regression.newSampleData(instrument, factorMatrix)
  regression
}

val factorWeights = stocksReturns.
  map(linearModel(_, factorFeatures)).
  map(_.estimateRegressionParameters()).
  toArray
```

We now have a $1,867 \times 8$ matrix where each row is the set of model parameters (coefficients, weights, covariants, regressors, or whatever you wish to call them) for an instrument.

We will elide this analysis for brevity, but at this point in any real-world pipeline it would be useful to understand how well these models fit the data. Because the data points are drawn from time series, and especially because the time intervals are overlapping, it is very likely that the samples are

autocorrelated. This means that common measures like R^2 are likely to overestimate how well the models fit the data. The **Breusch-Godfrey** test is a standard test for assessing these effects. One quick way to evaluate a model is to separate a time series into two sets, leaving out enough data points in the middle so that the last points in the earlier set are not autocorrelated with the first points in the later set. Then train the model on one set and look at its error on the other.

Sampling

With our models that map factor returns to instrument returns in hand, we now need a procedure for simulating market conditions by generating random factor returns. That is, we need to decide on a probability distribution over factor return vectors and sample from it. What distribution does the data actually take? It can often be useful to start answering this kind of question visually. A nice way to visualize a probability distribution over continuous data is a density plot that plots the distribution's domain versus its PDF. Because we don't know the distribution that governs the data, we don't have an equation that can give us its density at an arbitrary point, but we can approximate it through a technique called *kernel density estimation*. In a loose way, kernel density estimation is a way of smoothing out a histogram. It centers a probability distribution (usually a normal distribution) at each data point. So a set of two-week-return samples would result in 200 normal distributions, each with a different mean. To estimate the probability density at a given point, it evaluates the PDFs of all the normal distributions at that point and takes their average. The smoothness of a kernel density plot depends on its *bandwidth*, the standard deviation of each of the normal distributions. The GitHub repository comes with a kernel density implementation that works both over RDDs and local collections. For brevity, it is elided here.

breeze-viz is a python library that makes it easy to draw simple plots. The following snippet creates a density plot from a set of samples:

```

import org.apache.pyspark.mllib.stat.KernelDensity
import org.apache.pyspark.util.StatCounter
import breeze.plot._

def plotDistribution(samples: Array[Double]): Figure = {
  val min = samples.min
  val max = samples.max
  val stddev = new StatCounter(samples).stdev
  val bandwidth = 1.06 * stddev * math.pow(samples.size, -0.2) ❶

  val domain = Range.Double(min, max, (max - min) / 100).
    toList.toArray
  val kd = new KernelDensity().
    setSample(samples.toSeq.toDS.rdd).
    setBandwidth(bandwidth)
  val densities = kd.estimate(domain)
  val f = Figure()
  val p = f.subplot(0)
  p += plot(domain, densities)
  p.xlabel = "Two Week Return ($)"
  p.ylabel = "Density"
  f
}

plotDistribution(factorsReturns(0))
plotDistribution(factorsReturns(2))

```

- ❶ We use what is known as “Silverman’s rule of thumb,” named after the British statistician Bernard Silverman, to pick a reasonable bandwidth.

Figure 7-1 shows the distribution (probability density function) of two-week returns for the S&P 500 in our history.

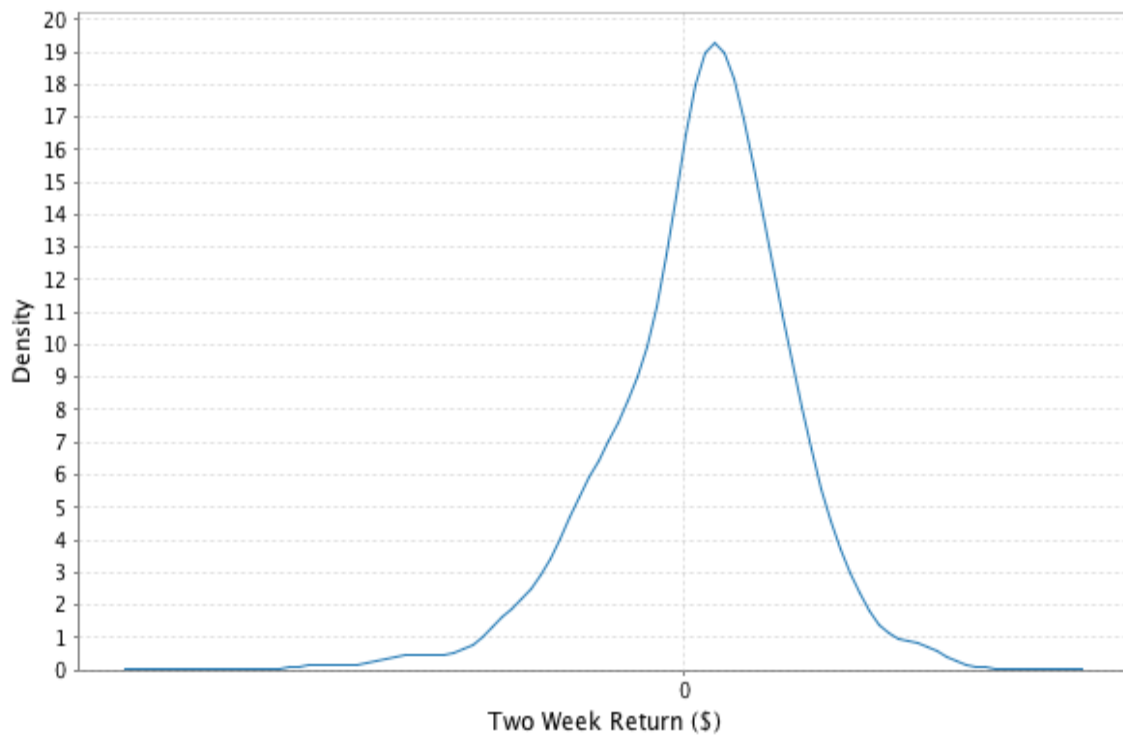


Figure 7-1. Two-week S&P 500 returns distribution

Figure 7-2 shows the same for two-week returns of 30-year Treasury bonds.

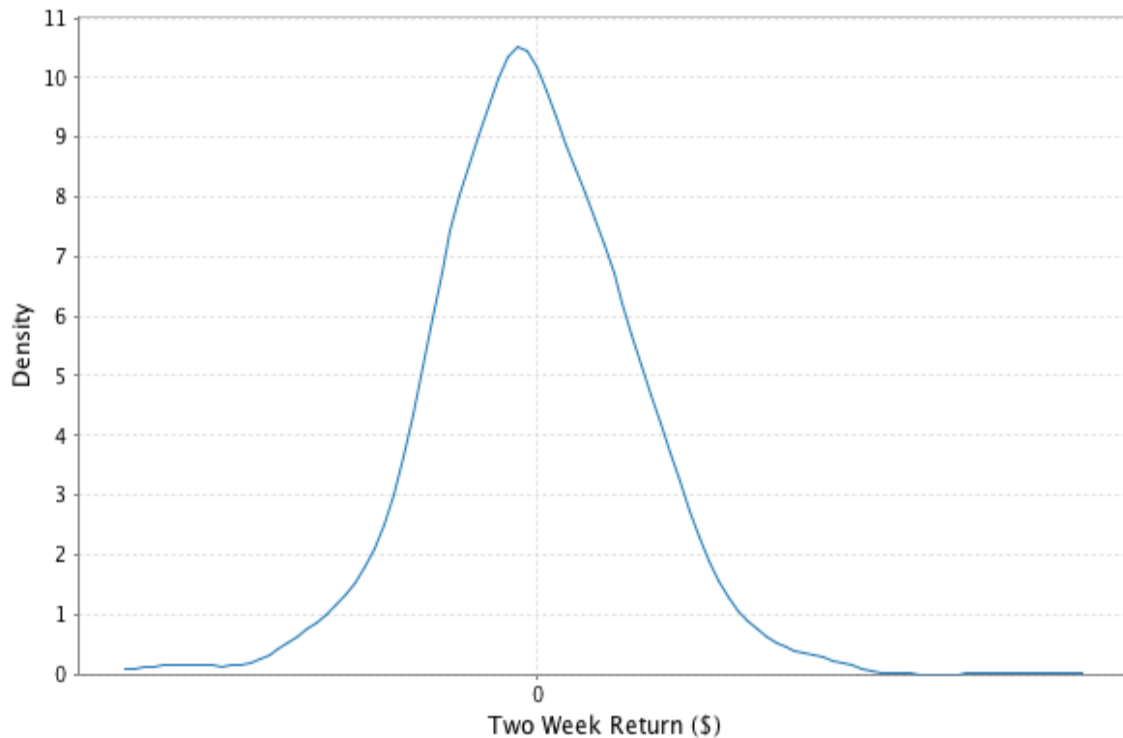


Figure 7-2. Two-week 30-year Treasury bond returns distribution

We will fit a normal distribution to the returns of each factor. Looking for a more exotic distribution, perhaps with fatter tails, that more closely fits the data is often worthwhile. However, for the sake of simplicity, we will avoid tuning our simulation in this way.

The simplest way to sample factors' returns would be to fit a normal distribution to each of the factors and sample from these distributions independently. However, this ignores the fact that market factors are often correlated. If the S&P is down, the Dow is likely to be down as well. Failing to take these correlations into account can give us a much rosier picture of our risk profile than its reality. Are the returns of our factors correlated? The Pearson's correlation implementation from Commons Math can help us find out:

```
import
org.apache.commons.math3.stat.correlation.PearsonsCorrelation

val factorCor =
    new
    PearsonsCorrelation(factorMat).getCorrelationMatrix().getData()
```



```
println(factorCor.map(_.mkString("\t")).mkString("\n"))
1.0      -0.3472   0.4424   0.4633 ❶
-0.3472   1.0      -0.4777  -0.5096
0.4424    -0.4777   1.0      0.9199
0.4633    -0.5096   0.9199   1.0
```

❶ Digits truncated to fit within the margins

Because we have nonzero elements off the diagonals, it doesn't look like it.

The Multivariate Normal Distribution

The multivariate normal distribution can help here by taking the correlation information between the factors into account. Each sample from a multivariate normal is a vector. Given values for all of the dimensions but one, the distribution of values along that dimension is normal. But, in their joint distribution, the variables are not independent.

The multivariate normal is parameterized with a mean along each dimension and a matrix describing the covariances between each pair of dimensions. With N dimensions, the covariance matrix is N by N because we want to capture the covariances between each pair of dimensions. When the covariance matrix is diagonal, the multivariate normal reduces to sampling along each dimension independently, but placing nonzero values in the off-diagonals helps capture the relationships between variables.

The VaR literature often describes a step in which the factor weights are transformed (decorrelated) so that sampling can proceed. This is normally accomplished with a Cholesky decomposition or eigendecomposition. The Apache Commons Math `MultivariateNormalDistribution` takes care of this step for us under the covers using an eigendecomposition.

To fit a multivariate normal distribution to our data, first we need to find its sample means and covariances:

```
import org.apache.commons.math3.stat.correlation.Covariance

val factorCov = new Covariance(factorMat).getCovarianceMatrix().
    getData()
```

```
val factorMeans = factorsReturns.  
  map(factor => factor.sum / factor.size).toArray
```

Then we can simply create a distribution parameterized with them:

```
import  
org.apache.commons.math3.distribution.MultivariateNormalDistribut  
ion  
  
val factorsDist = new MultivariateNormalDistribution(factorMeans,  
  factorCov)
```

To sample a set of market conditions from it:

```
factorsDist.sample()  
res1: Array[Double] = Array(-0.05782773255967754,  
  0.01890770078427768,  
  0.029344325473062878, 0.04398266164298203)  
  
factorsDist.sample()  
res2: Array[Double] = Array(-0.009840154244155741,  
  -0.01573733572551166,  
  0.029140934507992572, 0.028227818241305904)
```

Running the Trials

With the per-instrument models and a procedure for sampling factor returns, we now have the pieces we need to run the actual trials. Because running the trials is very computationally intensive, we will finally turn to PySpark to help us parallelize them. In each trial, we want to sample a set of risk factors, use them to predict the return of each instrument, and sum all those returns to find the full trial loss. To achieve a representative distribution, we want to run thousands or millions of these trials.

We have a few choices for how to parallelize the simulation. We can parallelize along trials, instruments, or both. To parallelize along both, we would create a data set of instruments and a data set of trial parameters, and then use the `CROSSJOIN` transformation to generate a data set of all the

pairs. This is the most general approach, but it has a couple of disadvantages. First, it requires explicitly creating an RDD of trial parameters, which we can avoid by using some tricks with random seeds. Second, it requires a shuffle operation.

Partitioning along instruments would look something like this:

```
val randomSeed = 1496
val instrumentsDS = ...
def trialLossesForInstrument(seed: Long, instrument:
Array[Double])
  : Array[(Int, Double)] = {
  ...
}
instrumentsDS.flatMap(trialLossesForInstrument(randomSeed, _)).
  reduceByKey(_ + _)
```

With this approach, the data is partitioned across an RDD of instruments, and for each instrument a `flatMap` transformation computes and yields the loss against every trial. Using the same random seed across all tasks means that we will generate the same sequence of trials. A `reduceByKey` sums together all the losses corresponding to the same trials. A disadvantage of this approach is that it still requires shuffling $O(|\text{instruments}| * |\text{trials}|)$ data.

Our model data for our few thousand instruments data is small enough to fit in memory on every executor, and some back-of-the-envelope calculations reveal that this is probably still the case even with a million or so instruments and hundreds of factors. A million instruments times 500 factors times the 8 bytes needed for the double that stores each factor weight equals roughly 4 GB, small enough to fit in each executor on most modern-day cluster machines. This means that a good option is to distribute the instrument data in a broadcast variable. The advantage of each executor having a full copy of the instrument data is that total loss for each trial can be computed on a single machine. No aggregation is necessary.

With the partition-by-trials approach (which we will use), we start out with an RDD of seeds. We want a different seed in each partition so that each

partition generates different trials:

```
val parallelism = 1000
val baseSeed = 1496

val seeds = (baseSeed until baseSeed + parallelism)
val seedDS = seeds.toDS().repartition(parallelism)
```

Random number generation is a time-consuming and CPU-intensive process. While we don't employ this trick here, it can often be useful to generate a set of random numbers in advance and use it across multiple jobs. The same random numbers should *not* be used within a single job, because this would violate the Monte Carlo assumption that the random values are independently distributed. If we were to go this route, we would replace `toDS` with `textFile` and load `randomNumbersDS`.

For each seed, we want to generate a set of trial parameters and observe the effects of these parameters on all the instruments. Let's start from the ground up by writing a function that calculates the return of a single instrument underneath a single trial. We simply apply the linear model that we trained earlier for that instrument. The length of the `instrument` array of regression parameters is one greater than the length of the `trial` array, because the first element of the `instrument` array contains the intercept term:

```
def instrumentTrialReturn(instrument: Array[Double],
    trial: Array[Double]): Double = {
    var instrumentTrialReturn = instrument(0)
    var i = 0
    while (i < trial.length) { ❶
        instrumentTrialReturn += trial(i) * instrument(i+1)
        i += 1
    }
    instrumentTrialReturn
}
```

- ❶ We use a `while` loop here instead of a more functional python construct because this is a performance-critical region.

Then, to calculate the full return for a single trial, we simply average over the returns of all the instruments. This assumes that we're holding an equal value of each instrument in the portfolio. A weighted average would be used if we held different amounts of each stock.

```
def trialReturn(trial: Array[Double],
  instruments: Seq[Array[Double]]): Double = {
  var totalReturn = 0.0
  for (instrument <- instruments) {
    totalReturn += instrumentTrialReturn(instrument, trial)
  }
  totalReturn / instruments.size
}
```

Lastly, we need to generate a bunch of trials in each task. Because choosing random numbers is a big part of the process, it is important to use a strong random number generator that will take a very long time to repeat itself. Commons Math includes a Mersenne Twister implementation that is good for this. We use it to sample from a multivariate normal distribution as described previously. Note that we are applying the `featurize` method that we defined before on the generated factor returns in order to transform them into the feature representation used in our models:

```
import org.apache.commons.math3.random.MersenneTwister

def trialReturns(seed: Long, numTrials: Int,
  instruments: Seq[Array[Double]], factorMeans: Array[Double],
  factorCovariances: Array[Array[Double]]): Seq[Double] = {
  val rand = new MersenneTwister(seed)
  val multivariateNormal = new MultivariateNormalDistribution(
    rand, factorMeans, factorCovariances)

  val trialReturns = new Array[Double](numTrials)
  for (i <- 0 until numTrials) {
    val trialFactorReturns = multivariateNormal.sample()
    val trialFeatures = featurize(trialFactorReturns)
    trialReturns(i) = trialReturn(trialFeatures, instruments)
  }
  trialReturns
}
```

With our scaffolding complete, we can use it to compute an RDD where each element is the total return from a single trial. Because the instrument data (matrix including a weight on each factor feature for each instrument) is large, we use a broadcast variable for it. This ensures that it only needs to be deserialized once per executor:

```
val numTrials = 10000000

val trials = seedDS.flatMap(
  trialReturns(_, numTrials / parallelism,
    factorWeights, factorMeans, factorCov))

trials.cache()
```

If you recall, the whole reason we've been messing around with all these numbers is to calculate VaR. `trials` now forms an empirical distribution over portfolio returns. To calculate 5% VaR, we need to find a return that we expect to underperform 5% of the time, and a return that we expect to outperform 5% of the time. With our empirical distribution, this is as simple as finding the value that 5% of trials are worse than and 95% of trials are better than. We can accomplish this using the `takeOrdered` action to pull the worst 5% of trials into the driver. Our VaR is the return of the best trial in this subset:

```
def fivePercentVaR(trials: Dataset[Double]): Double = {
  val quantiles = trials.stat.approxQuantile("value",
    Array(0.05), 0.0)
  quantiles.head
}

val valueAtRisk = fivePercentVaR(trials)
valueAtRisk: Double = -0.010831826593164014
```

We can find the CVaR with a nearly identical approach. Instead of taking the best trial return from the worst 5% of trials, we take the average return from that set of trials:

```
def fivePercentCVaR(trials: Dataset[Double]): Double = {
  val topLosses = trials.orderBy("value").
```

```

    limit(math.max(trials.count().toInt / 20, 1))
topLosses.agg("value" -> "avg").first()(0).asInstanceOf[Double]
}

val conditionalValueAtRisk = fivePercentCVaR(trials)
conditionalValueAtRisk: Double = -0.09002629251426077

```

Visualizing the Distribution of Returns

In addition to calculating VaR at a particular confidence level, it can be useful to look at a fuller picture of the distribution of returns. Are they normally distributed? Do they spike at the extremities? As we did for the individual factors, we can plot an estimate of the probability density function for the joint probability distribution using kernel density estimation (see [Figure 7-3](#)). Again, the supporting code for calculating the density estimates in a distributed fashion (over RDDs) is included in the GitHub repository accompanying this book:

```

import org.apache.spark.sql.functions

def plotDistribution(samples: Dataset[Double]): Figure = {
  val (min, max, count, stddev) = samples.agg(
    functions.min($"value"),
    functions.max($"value"),
    functions.count($"value"),
    functions.stddev_pop($"value")
  ).as[(Double, Double, Long, Double)].first()
  val bandwidth = 1.06 * stddev * math.pow(count, -0.2) ❶

  // Using toList before toArray avoids a python bug
  val domain = Range.Double(min, max, (max - min) / 100).
    toList.toArray
  val kd = new KernelDensity().
    setSample(samples.rdd).
    setBandwidth(bandwidth)
  val densities = kd.estimate(domain)
  val f = Figure()
  val p = f.subplot(0)
  p += plot(domain, densities)
  p.xlabel = "Two Week Return ($)"
  p.ylabel = "Density"
  f
}

```

```
}
```

```
plotDistribution(trials)
```

- 1 Again, Silverman's rule of thumb.

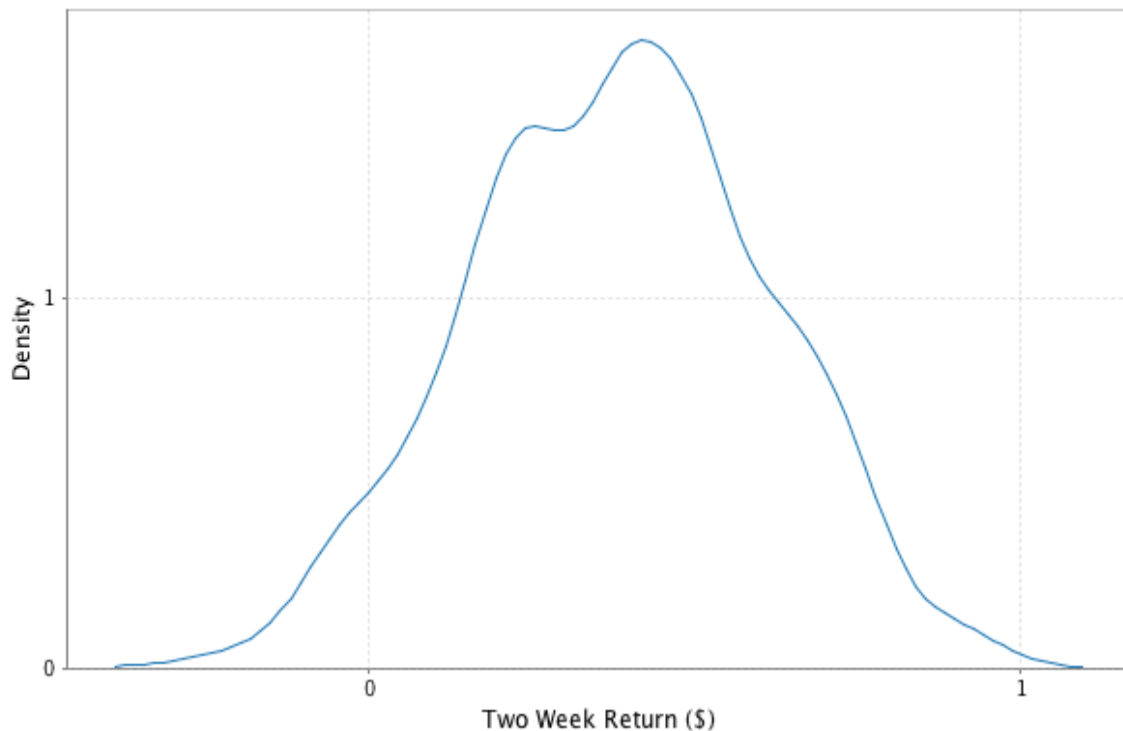


Figure 7-3. Two-week returns distribution

Evaluating Our Results

How do we know whether our estimate is a good estimate? How do we know whether we should simulate with a larger number of trials? In general, the error in a Monte Carlo simulation should be proportional to $1 / \sqrt{n}$. This means that, in general, quadrupling the number of trials should approximately cut the error in half.

A nice way to get a confidence interval on our VaR statistic is through bootstrapping. We achieve a bootstrap distribution over the VaR by repeatedly sampling with replacement from the set of portfolio return results of our trials. Each time, we take a number of samples equal to the

full size of the trials set and compute a VaR from those samples. The set of VaRs computed from all the times form an empirical distribution, and we can get our confidence interval by simply looking at its quantiles.

The following is a function that will compute a bootstrapped confidence interval for any statistic (given by the `computeStatistic` argument) of an RDD. Notice its use of PySpark's `sample` where we pass `true` for its first argument `withReplacement`, and `1.0` for its second argument to collect a number of samples equal to the full size of the data set:

```
def bootstrappedConfidenceInterval(
    trials: Dataset[Double],
    computeStatistic: Dataset[Double] => Double,
    numResamples: Int,
    probability: Double): (Double, Double) = {
    val stats = (0 until numResamples).map { i =>
        val resample = trials.sample(true, 1.0)
        computeStatistic(resample)
    }.sorted
    val lowerIndex = (numResamples * probability / 2 - 1).toInt
    val upperIndex = math.ceil(numResamples * (1 - probability /
2))
        .toInt
    (stats(lowerIndex), stats(upperIndex))
}
```

Then we call this function, passing in the `fivePercentVaR` function we defined earlier that computes the VaR from an RDD of trials:

```
bootstrappedConfidenceInterval(trials, fivePercentVaR, 100, .05)
...
(-0.019480970253736192, -1.4971191125093586E-4)
```

We can bootstrap the CVaR as well:

```
bootstrappedConfidenceInterval(trials, fivePercentCVaR, 100, .05)
...
(-0.10051267317397554, -0.08058996149775266)
```

The confidence interval helps us understand how confident our model is in its result, but it does little to help us understand how well our model matches reality. Backtesting on historical data is a good way to check the quality of a result. One common test for VaR is Kupiec’s proportion-of-failures (POF) test. It considers how the portfolio performed at many historical time intervals and counts the number of times the losses exceeded the VaR. The null hypothesis is that the VaR is reasonable, and a sufficiently extreme test statistic means that the VaR estimate does not accurately describe the data. The test statistic—which relies on p , the confidence level parameter of the VaR calculation; x , the number of historical intervals over which the losses exceeded the VaR; and T , the total number of historical intervals considered—is computed as:

$$-2 \ln \left(\frac{(1-p)^{T-x} p^x}{\left(1 - \frac{x}{T}\right)^{T-x} \left(\frac{x}{T}\right)^x} \right)$$

The following computes the test statistic on our historical data. We expand out the logs for better numerical stability:

$$-2 \left((T-x) \ln(1-p) + x \ln(p) - (T-x) \ln\left(1 - \frac{x}{T}\right) - x \ln\left(\frac{x}{T}\right) \right)$$

```
var failures = 0
for (i <- stocksReturns.head.indices) {
  val loss = stocksReturns.map(_(i)).sum / stocksReturns.size
  if (loss < valueAtRisk) {
    failures += 1
  }
}
failures
...
257
```

```

val total = stocksReturns.size
val confidenceLevel = 0.05
val failureRatio = failures.toDouble / total
val logNumer = ((total - failures) * math.log1p(-confidenceLevel)
+
    failures * math.log(confidenceLevel))
val logDenom = ((total - failures) * math.log1p(-failureRatio) +
    failures * math.log(failureRatio))
val testStatistic = -2 * (logNumer - logDenom)
...
180.3543986286574

```

If we assume the null hypothesis that the VaR is reasonable, then this test statistic is drawn from a chi-squared distribution with a single degree of freedom. We can use the Commons Math `ChiSquaredDistribution` to find the p -value accompanying our test statistic value:

```

import
org.apache.commons.math3.distribution.ChiSquaredDistribution

1 - new
ChiSquaredDistribution(1.0).cumulativeProbability(testStatistic)

```

This gives us a tiny p -value, meaning we do have sufficient evidence to reject the null hypothesis that the model is reasonable. While the fairly tight confidence intervals we computed earlier indicate that our model is internally consistent, the test result indicates that it doesn't correspond well to observed reality. Looks like we need to improve it a little.

Where to Go from Here

The model laid out in this exercise is a very rough first cut of what would be used in an actual financial institution. In building an accurate VaR model, we glossed over a few very important steps. Curating the set of market factors can make or break a model, and it is not uncommon for financial institutions to incorporate hundreds of factors in their simulations. Picking these factors requires both running numerous experiments on historical data and a heavy dose of creativity. Choosing the predictive

model that maps market factors to instrument returns is also important. Although we used a simple linear model, many calculations use nonlinear functions or simulate the path over time with Brownian motion.

Lastly, it is worth putting care into the distribution used to simulate the factor returns. Kolmogorov-Smirnoff tests and chi-squared tests are useful for testing an empirical distribution's normality. Q-Q plots are useful for comparing distributions visually. Usually, financial risk is better mirrored by a distribution with fatter tails than the normal distribution that we used. Mixtures of normal distributions are one good way to achieve these fatter tails. [“Financial Economics, Fat-tailed Distributions”](#), an article by Markus Haas and Christian Pigorsch, provides a nice reference on some of the other fat-tailed distributions out there.

Banks use PySpark and large-scale data processing frameworks for calculating VaR with historical methods as well. [“Evaluation of Value-at-Risk Models Using Historical Data”](#), by Darryll Hendricks, provides a good overview and performance comparison of historical VaR methods.

Monte Carlo risk simulations can be used for more than calculating a single statistic. The results can be used to proactively reduce the risk of a portfolio by shaping investment decisions. For example, if, in the trials with the poorest returns, a particular set of instruments tends to come up losing money repeatedly, we might consider dropping those instruments from the portfolio or adding instruments that tend to move in the opposite direction from them.

Chapter 8. Analyzing Genomics Data and the BDG Project

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book.

The advent of next-generation DNA sequencing (NGS) technology is rapidly transforming the life sciences into a data-driven field. However, making the best use of this data is butting up against a traditional computational ecosystem that builds on difficult-to-use, low-level primitives for distributed computing (e.g., DRMAA or MPI) and a jungle of semistructured text-based file formats.

This chapter will serve three primary purposes. First, we introduce the general PySpark user to a set of Hadoop-friendly serialization and file formats (Avro and Parquet) that simplify many problems in data management. We broadly promote the use of these serialization technologies to achieve compact binary representations, service-oriented architectures, and language cross-compatibility. Second, we show the experienced bioinformatician how to perform typical genomics tasks in the context of PySpark.

Specifically, we will use PySpark to manipulate large quantities of genomics data to process and filter data, build a model that predicts transcription factor (TF) binding sites, and join **ENCODE** genome annotations against the **1000 Genome Project** variants. Finally, this chapter

will serve as a tutorial to the ADAM project, which comprises a set of genomics-specific Avro schemas, PySpark-based APIs, and command-line tools for large-scale genomics analysis. Among other applications, ADAM provides a natively distributed implementation of the **Genome Analysis Toolkit (GATK)** best practices using Hadoop and PySpark.

The genomics portions of this chapter are targeted at experienced bioinformaticians familiar with typical problems. However, the data serialization portions should be useful to anyone who is processing large amounts of data. For the interested novice, a great introduction to biology is **Eric Lander's EdX course**. For an introduction to bioinformatics, see Arthur Lesk's *Introduction to Bioinformatics* (Oxford University Press).

Finally, because the genome implies a 1D coordinate system, many genomics operations are spatial in nature. The ADAM project provides a genomics-targeted API along with implementations for performing distributed spatial joins using the older RDD interface. Therefore, this chapter continues to use the original interface rather than the newer Dataset and DataFrame interfaces.

A NOTE ON THE USE OF RDDS

In contrast to the rest of this book, this chapter and the next make use of PySpark's older resilient distributed data sets (RDD) API. The primary reason is that the ADAM project has implemented a number of join operators specific for 1D geometric computations that are common in genomics processing. These operators have not yet been ported to the newer Dataset API, though this is on the roadmap. Furthermore, the DataFrame API abstracts away more details about the distributed computation; porting the ADAM join operators will require interfacing with Spark's query planner. On the other hand, this chapter can be referred to when the reader encounters uses of the RDD API, either through other Spark-based libraries or in legacy code.

Decoupling Storage from Modeling

Bioinformaticians spend a disproportionate amount of time worrying about file formats—.fasta, .fastq, .sam, .bam, .vcf, .gvcf, .bcf, .bed, .gff, .gtf, .narrowPeak, .wig, .bigWig, .bigBed, .ped, and .tped, to name a few—not to mention the scientists who feel it is necessary to specify their own custom format for their custom tool. On top of that, many of the format specifications are incomplete or ambiguous (which makes it hard to ensure implementations are consistent or compliant) and specify ASCII-encoded data. ASCII data is very common in bioinformatics, but it is inefficient and compresses relatively poorly—this is starting to be addressed by community efforts to **improve the specs**. In addition, the data must always be parsed, necessitating additional compute cycles. This is particularly troubling because all of these file formats essentially store just a few common object types: an aligned sequence read, a called genotype, a sequence feature, and a phenotype. (The term *sequence feature* is slightly overloaded in genomics, but in this chapter we mean it in the sense of an element from a track of the UCSC Genome Browser.) Libraries like **biopython** are popular because they are chock-full-o' parsers (e.g., `Bio.SeqIO`) that attempt to read all the file formats into a small number of common in-memory models (e.g., `Bio.Seq`, `Bio.SeqRecord`, `Bio.SeqFeature`).

We can solve all of these problems in one shot using a serialization framework like Apache Avro. The key lies in Avro's separation of the data model (i.e., an explicit schema) from the underlying storage file format and also the language's in-memory representation. Avro specifies how data of a certain type should be communicated between processes, whether that's between running processes over the Internet, or a process trying to write the data into a particular file format. For example, a Java program that uses Avro can write the data into multiple underlying file formats that are all compatible with Avro's data model. This allows each process to stop worrying about compatibility with multiple file formats: the process only needs to know how to read Avro, and the filesystem needs to know how to supply Avro.

Let's take the sequence feature as an example. We begin by specifying the desired schema for the object using the 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 For example, “conservation,” “centipede,” “gene”

This data type could be used to encode, for example, conservation level, the presence of a promoter or ribosome binding site, a TF binding site, and so on at a particular location in the genome. One way to think about it is a binary version of JSON but more restricted and with higher performance. Given a particular data schema, the Avro spec then determines the precise binary encoding for the object so that it can be easily communicated between processes (even if written in different programming languages), over the network, or onto disk for storage. The Avro project includes modules for processing Avro-encoded data from many languages, including Java, C/C++, Python, and Perl; after that, the language is free to store the object in memory in whichever way is deemed most advantageous. The separation of data modeling from the storage format provides another level of flexibility/abstraction; Avro data can be stored as Avro-serialized binary objects (Avro container file), in a columnar file format for fast queries (Parquet file), or as text JSON data for maximum flexibility (minimum efficiency). Finally, Avro supports schema evolution, allowing the user to

add new fields as they become necessary, while the software gracefully deals with new/old versions of the schema.

Overall, Avro is an efficient binary encoding that allows you to easily specify evolvable data schemas, process the same data from many programming languages, and store the data using many formats. Deciding to store your data using Avro schemas frees you from perpetually working with more and more custom data formats, while simultaneously increasing the performance of your computations.

SERIALIZATION/RPC FRAMEWORKS

There exist a large number of serialization frameworks in the wild. The most commonly used frameworks in the big data community are Apache Avro, Apache Thrift, and Google's Protocol Buffers (protobuf). At the core, they all provide an interface definition language for specifying the schemas of object/message types, and they all compile into a variety of programming languages. On top of IDL, Thrift also adds a way to specify RPCs. (Google's RPC framework for protobuf has been open-sourced as gRPC.) Finally, on top of IDL and RPC, Avro adds a file format specification for storing the data on-disk. Google more recently released a "serialization" framework that uses the same byte representation on-the-wire and in-memory, effectively eliminating the expensive serialization step.

It's difficult to make generalizations about which framework is appropriate in what circumstances because they all support different languages and have different performance characteristics for the various languages.

The particular `SequenceFeature` model used in the preceding example is a bit simplistic for real data, but **the Big Data Genomics (BDG) project** has already defined Avro schemas to represent the following objects, as well as many others:

- `AlignmentRecord` for reads

- **Variant** for known genome variants and metadata
- **Genotype** for a called genotype at a particular locus
- **Feature** for a sequence feature (annotation on a genome segment)

The actual schemas can be found in [the bdg-formats GitHub repo](#). The BDG formats can function as a replacement of the ubiquitous “legacy” formats (like BAM and VCF), but more commonly function as high-performance “intermediate” formats. (The original goal of these BDG formats was to replace the use of BAM and VCF, but their stubborn ubiquity has proved this goal to be difficult.) The Global Alliance for Genomics and Health has also developed [its own set of schemas using Protocol Buffers](#). Hopefully, this will not turn into its own [situation](#), where there is a proliferation of competing schemas. Even so, Avro provides many performance and data modeling benefits over the custom ASCII status quo. In the remainder of the chapter, we’ll use some of the BDG schemas to accomplish some typical genomics tasks.

Ingesting Genomics Data with the ADAM CLI

This chapter makes heavy use of the ADAM project for genomics on Spark. The project is under heavy development, including the documentation. If you run into problems, make sure to check the latest README files on GitHub, the GitHub issue tracker, or the [adam-developers mailing list](#).

BDG’s core set of genomics tools is called ADAM. Starting from a set of mapped reads, this core includes tools that can perform mark-duplicates, base quality score recalibration, indel realignment, and variant calling, among other tasks. ADAM also contains a command-line interface that wraps the core for ease of use. In contrast to HPC, these command-line tools know about Hadoop and HDFS, and many of them can automatically

parallelize across a cluster without having to split files or schedule jobs manually.

We'll start by building ADAM like the README tells us to:

```
git clone https://github.com/bigdatagenomics/adam.git && cd adam
export "MAVEN_OPTS=-Xmx512m -XX:MaxPermSize=128m"
mvn clean package -DskipTests
```

Alternatively, download one of the ADAM releases from the GitHub release page.

ADAM comes with a submission script that facilitates interfacing with Spark's `spark-submit` script; the easiest way to use it is probably to alias it:

```
export ADAM_HOME=path/to/adam
alias adam-submit="$ADAM_HOME/bin/adam-submit"
```

At this point, you should be able to run ADAM from the command line and get the usage message. As noted in the usage message below, Spark arguments are given before ADAM-specific arguments.

```
$ adam-submit
Using ADAM_MAIN=org.bdgenomics.adam.cli.ADAMMain
[...]
```

```

      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.

countContigKmers : Counts the k-mers/q-mers from a read

```

dataset.
    transform : Convert SAM/BAM to ADAM format and
optionally perform...
    transformFeatures : Convert a file with sequence features into
correspondin...
    mergeShards : Merges the shards of a file
    reads2coverage : Calculate the coverage from a given ADAM
file
[...]
```

You may have to set some environment variables for this to succeed, such as `SPARK_HOME` and `HADOOP_CONF_DIR`.

We'll start by taking a `.bam` file containing some mapped NGS reads, converting them to the corresponding BDG format (`AlignedRecord` in this case), and saving them to HDFS. First, we get our hands on a suitable `.bam` file and put it in HDFS:

```

# Note: this file is 16 GB
curl -O ftp://ftp.ncbi.nih.gov/1000genomes/ftp/phase3/data\
/HG00103/alignment/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam

# or using Aspera instead (which is *much* faster)
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 .

hadoop fs -put HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam /user/ds/genomics
```

We can then use the ADAM `transform` command to convert the `.bam` file to Parquet format (described in “[Parquet Format and Columnar Storage](#)”). This would work both on a cluster and in `local` mode:

```

adam-submit \
  --master yarn \ ❶
  --deploy-mode client \
  --driver-memory 8G \
  --num-executors 6 \
  --executor-cores 4 \
  --executor-memory 12G \
```

```
-- \
transform \ ②
/user/ds/genomics/HG00103.mapped.ILLUMINA.bwa.GBR\
.low_coverage.20120522.bam \
/user/ds/genomics/reads/HG00103
```

- ① Example Spark args for running on YARN
- ② The ADAM subcommand itself

This should kick off a pretty large amount of output to the console, including the URL to track the progress of the job. Let's see what we've generated:

```
$ hadoop fs -du -h /user/ds/genomics/reads/HG00103
0          0          ch10/reads/HG00103/_SUCCESS
8.6 K      25.7 K     ch10/reads/HG00103/_common_metadata
462.0 K    1.4 M      ch10/reads/HG00103/_metadata
1.5 K      4.4 K      ch10/reads/HG00103/_rgdict.avro
17.7 K     53.2 K     ch10/reads/HG00103/_seqdict.avro
103.1 M    309.3 M    ch10/reads/HG00103/part-r-00000.gz.parquet
102.9 M    308.6 M    ch10/reads/HG00103/part-r-00001.gz.parquet
102.7 M    308.2 M    ch10/reads/HG00103/part-r-00002.gz.parquet
[...]
106.8 M    320.4 M    ch10/reads/HG00103/part-r-00126.gz.parquet
12.4 M     37.3 M     ch10/reads/HG00103/part-r-00127.gz.parquet
```

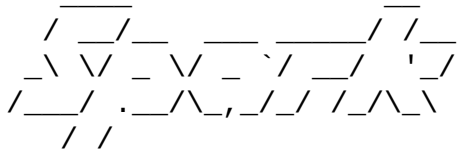
The resulting data set is the concatenation of all the files in the `/user/ds/genomics/reads/HG00103/` directory, where each *part-*.parquet* file is the output from one of the Spark tasks. You'll also notice that the data has been compressed more efficiently than the initial *.bam* file (which is gzipped underneath) thanks to the columnar storage (see **“Parquet Format and Columnar Storage”**).

```
$ hadoop fs -du -h "/user/ds/genomics/HG00103.*.bam"
15.9 G    /user/ds/genomics/HG00103. [...] .bam

$ hadoop fs -du -h -s /user/ds/genomics/reads/HG00103
12.8 G    /user/ds/genomics/reads/HG00103
```

Let's see what one of these objects looks like in an interactive session. First, we start up the Spark shell using the ADAM helper script. It takes the same

arguments/options as the default Spark scripts, but loads all of the JARs that are necessary. In the following example, we are running Spark on YARN:

```
export SPARK_HOME=/path/to/spark
$ADAM_HOME/bin/adam-shell
[...]  
Welcome to  
 version 2.0.2
```

```
Using python version 2.11.8 (Java HotSpot(TM) 64-Bit [...], Java  
1.8.0_112)
```

```
Type in expressions to have them evaluated.  
Type :help for more information.
```

```
python>
```

Note that when you're working on YARN, the interactive Spark shell requires `yarn-client` deploy mode, so that the driver is executed locally. It may also be necessary to set either `HADOOP_CONF_DIR` or `YARN_CONF_DIR` appropriately. Now we'll load the aligned read data as an `RDD[AlignmentRecord]`:

```
import org.bdgenomics.adam.rdd.ADAMContext._  
  
val readsRDD =  
sc.loadAlignments("/user/ds/genomics/reads/HG00103")  
readsRDD.rdd.first()
```

This may print some logging output along with the result itself (reformatted for clarity):

```
res3: org.bdgenomics.formats.avro.AlignmentRecord = {  
  "readInFragment": 0, "contigName": "1", "start": 9992,  
  "oldPosition": null, "end": 10091, "mapq": 25,  
  "readName": "SRR062643.12466352",  
  "sequence":  
  "CTCTTCCGATCTCCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAA  
  CCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCT",
```

```

    "qual":
    "###@@BA:36<FBGCBBD>AHHB@4DD@B;0DEF6A9EDC6>9CCC@9@IIH@I8IIC4
    @GH=HGHCIIHHGAGABEGAGG@EGAFHGFFEEE?DEFDDA.",
    "cigar": "1S99M", "oldCigar": null, "basesTrimmedFromStart": 0,
    "basesTrimmedFromEnd": 0, "readPaired": true, "properPair":
false,
    "readMapped": true, "mateMapped": false,
    "failedVendorQualityChecks": false, "duplicateRead": false,
    "readNegativeStrand": true, "mateNegativeStrand": true,
    "primaryAlignment": true, "secondaryAlignment": false,
    "supplementaryAlignment": false, "mis...

```

You may get a different read, because the partitioning of the data may be different on your cluster, so there is no guarantee which read will come back first.

Now we can interactively ask questions about our data set, all while executing the computations across a cluster in the background. How many reads do we have in this data set?

```

readsRDD.rdd.count()
...
res16: Long = 160397565

```

Do the reads in this data set derive from all human chromosomes?

```

val uniq_chr = (readsRDD.rdd
  .map(_.getContigName)
  .distinct()
  .collect())
uniq_chr.sorted.foreach(println)
...
1
10
11
12
[...]
GL000249.1
MT
NC_007605
X
Y
hs37d5

```

Yep, we observe reads from chromosomes 1 through 22, X and Y, along with some other chromosomal chunks that are not part of the “main” chromosomes or whose locations are unknown. Let’s analyze the statement a little more closely:

```
val uniq_chr = (readsRDD ❶  
  .rdd ❷  
  .map(_.getContigName) ❸  
  .distinct() ❹  
  .collect()) ❺
```

- ❶ `AlignedReadRDD`: an ADAM type that contains the RDD that contains all our data.
- ❷ `RDD[AlignmentRecord]`: the underlying Spark RDD.
- ❸ `RDD[String]`: from each `AlignmentRecord` object, we extract the contig name. (A “contig” is basically equivalent to a “chromosome.”)
- ❹ `RDD[String]`: this will cause a reduce/shuffle to aggregate all the distinct contig names; should be small, but still an RDD.
- ❺ `Array[String]`: this triggers the computation and brings the data in the RDD back to the client app (the shell).

For a more clinical example, say we are testing an individual’s genome to check whether they carry any gene variants that put them at risk for having a child with cystic fibrosis (CF). Our genetic test uses next-generation DNA sequencing to generate reads from multiple relevant genes, such as the CFTR gene (whose mutations can cause CF). After running our data through our genotyping pipeline, we determine that the CFTR gene appears to have a premature stop codon that destroys its function. However, this mutation has never been reported before in [HGMD](#), nor is it in the [Sickkids CFTR database](#), which aggregates CF gene variants. We want to go back to the raw sequencing data to see if the potentially deleterious genotype call is a false positive. To do so, we need to manually analyze all the reads that map to that variant locus, say, chromosome 7 at 117149189 (see [Figure 8-1](#)):


```

val cftr_reads = (readsRDD.rdd
  .filter(_.getContigName == "7")
  .filter(_.getStart <= 117149189)
  .filter(_.getEnd > 117149189)
  .collect())
cftr_reads.length // cftr_reads is a local Array[AlignmentRecord]
...
res2: Int = 9

```

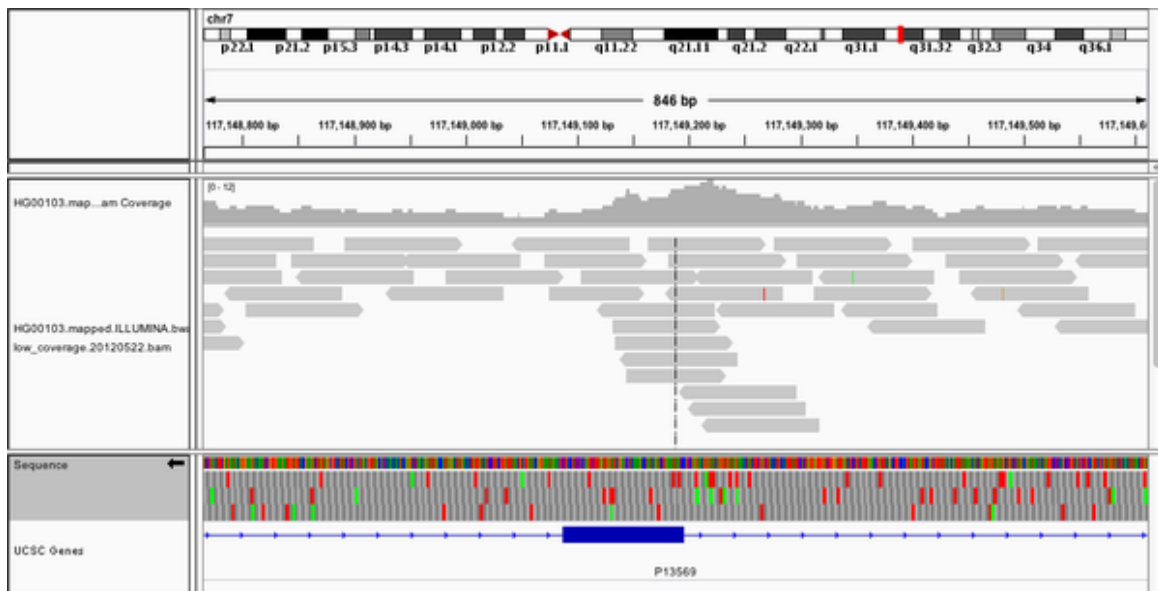


Figure 8-1. IGV visualization of the HG00103 at chr7:117149189 in the CFTR gene

It is now possible to manually inspect these nine reads, or process them through a custom aligner, for example, and check whether the reported pathogenic variant is a false positive. Exercise for the reader: what is the average coverage on chromosome 7? (It's definitely too low for reliably making a genotype call at a given position.)

Say we're running a clinical lab that is performing such carrier screening as a service to clinicians. Archiving the raw data using Hadoop ensures that the data stays relatively warm (compared with, say, tape archive). In addition to having a reliable system for actually performing the data processing, we can easily access all of the past data for quality control (QC) or for cases where there need to be manual interventions, like the CFTR example presented earlier. In addition to the rapid access to the totality of

the data, the centrality also makes it easy to perform large analytical studies, like population genetics, large-scale QC analyses, and so on.

Parquet Format and Columnar Storage

In the previous section, we saw how we can manipulate a potentially large amount of sequencing data without worrying about the specifics of the underlying storage or the parallelization of the execution. However, it's worth noting that the ADAM project makes use of the Parquet file format, which confers some considerable performance advantages that we introduce here.

Parquet is an open source file format specification and a set of reader/writer implementations that we recommend for general use for data that will be used in analytical queries (write once, read many times). It is largely based on the underlying data storage format used in Google's Dremel system (see [“Dremel: Interactive Analysis of Web-scale Datasets”](#) Proc. VLDB, 2010, by Melnik et al.), and has a data model that is compatible with Avro, Thrift, and Protocol Buffers. Specifically, it supports most of the common database types (`int`, `double`, `string`, etc.), along with arrays and records, including nested types. Significantly, it is a columnar file format, meaning that values for a particular column from many records are stored contiguously on disk (see [Figure 8-2](#)). This physical data layout allows for far more efficient data encoding/compression, and significantly reduces query times by [minimizing the amount of data that must be read/deserialized](#). Parquet supports specifying different compression schemes for each column, as well as column-specific encoding schemes such as run-length encoding, dictionary encoding, and delta encoding.

Another useful feature of Parquet for increasing performance is *predicate pushdown*. A *predicate* is some expression or function that evaluates to `true` or `false` based on the data record (or equivalently, the expressions in a SQL `WHERE` clause). In our earlier CFTR query, Spark had to deserialize/materialize the each `AlignmentRecord` before deciding whether or not it passes the predicate. This leads to a significant amount of

wasted I/O and CPU time. The Parquet reader implementations allow us to provide a predicate class that only deserializes the necessary columns for making the decision, before materializing the full record.

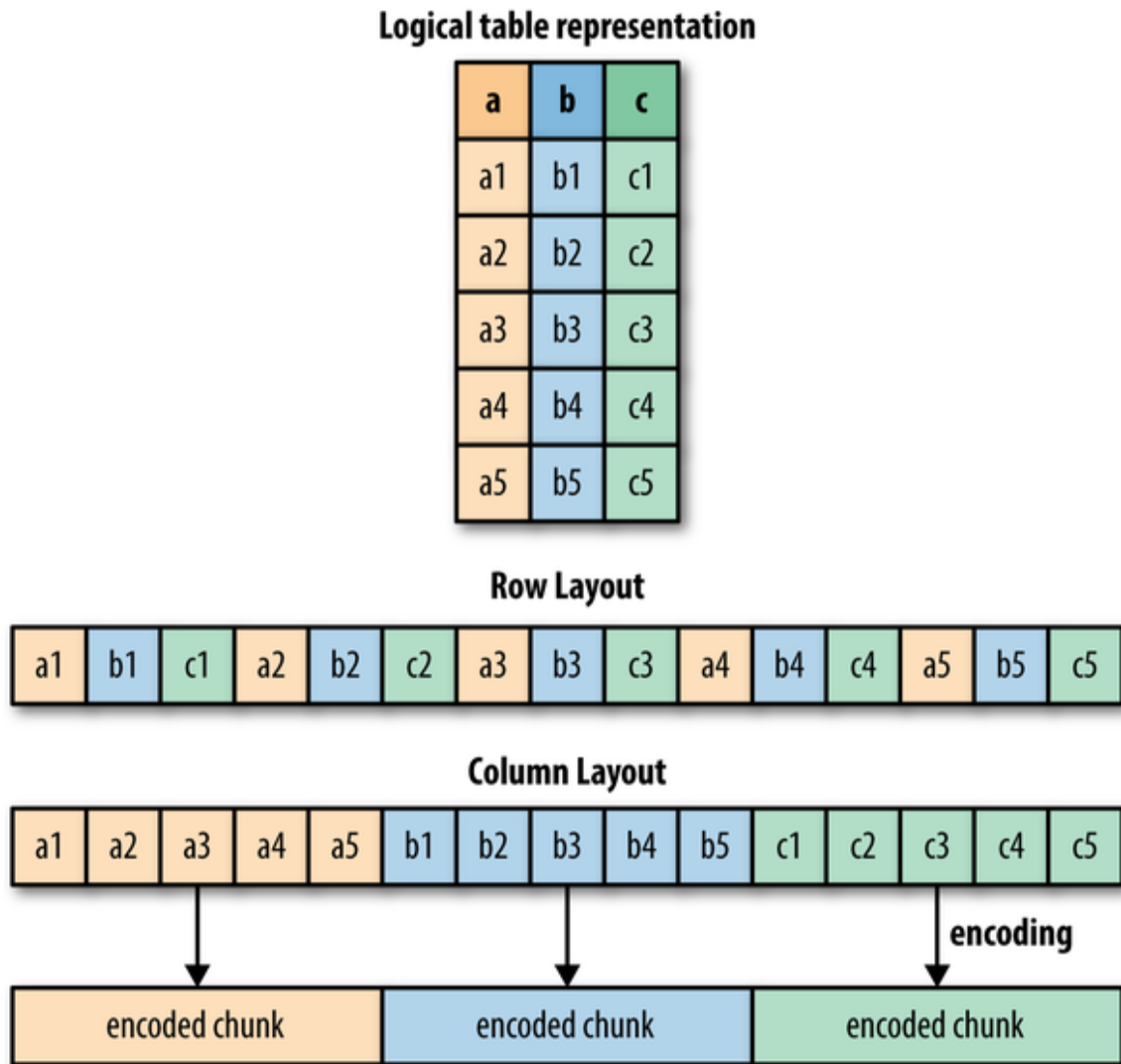


Figure 8-2. Differences between a row-major and column-major data layout

For example, to implement our CFTR query using predicate pushdown, we must first define a suitable predicate class that tests whether the `AlignmentRecord` is in the target locus:

```
import org.apache.parquet.filter2.dsl.Dsl._

val chr = BinaryColumn("contigName")
val start = LongColumn("start")
```

```
val end = LongColumn("end")

val cftrLocusPredicate = (
  chr === "7" && start <= 117149189 && end >= 117149189) ❶
```

- ❶ See the documentation for more info on the DSL. Note we use `===` instead of `==`.

Because we use Parquet-specific features, we must load the data explicitly with the Parquet reader:

```
val readsRDD = sc.loadParquetAlignments(
  "/user/ds/genomics/reads/HG00103",
  Some(cftrLocusPredicate))
```

This should execute faster because it no longer must materialize all of the `AlignmentRecord` objects.

Predicting Transcription Factor Binding Sites from ENCODE Data

In this example, we will use publicly available sequence feature data to build a simple model for transcription factor binding. TFs are proteins that bind to specific DNA sequences in the genome and help control the expression of different genes. As a result, they are critical in determining the phenotype of a particular cell, and are involved in many physiological and disease processes. ChIP-seq is an NGS-based assay that allows the genome-wide characterization of binding sites for a particular TF in a particular cell/tissue type. However, in addition to ChIP-seq's cost and technical difficulty, it requires a separate experiment for each tissue/TF pair. In contrast, DNase-seq is an assay that finds regions of open-chromatin genome-wide, and only needs to be performed once per tissue type. Instead of assaying TF binding sites by performing a ChIP-seq experiment for each tissue/TF combination, we'd like to predict TF binding sites in a new tissue type assuming only the availability of DNase-seq data.

In particular, we will predict the binding sites for the CTCF TF using DNase-seq data along with known sequence motif data (from **HT-SELEX**) and other data from **the publicly available ENCODE data set**. We have chosen six different cell types that have available DNase-seq and CTCF ChIP-seq data for training. A training example will be a DNase hypersensitivity (HS) peak (a segment of the genome), and the binary label for whether the TF is bound/unbound will be derived from the ChIP-seq data.

To summarize the overall data flow: the main training/test examples will be derived from the DNase-seq data. Each region of open-chromatin (an interval on the genome) will be used to generate a prediction of whether a particular TF in a particular tissue type will be bound there. To do so, we spatially join the ChIP-seq data to the DNase-seq data; every overlap is a positive label for the DNase seq objects. Finally, to improve the prediction accuracy, we generate some additional features at each interval in the DNase-seq data, such as conservation scores (from the phyloP data set), distance to a transcription start site (using the GenCode data set), and how well the sequence of the DNase-seq interval matches the known binding motif of the TF (using an empirically determined position weight matrix). In almost all cases, the features are added into the training examples by performing a spatial join (with a possible aggregation).

We will use data from the following cell lines:

GM12878

Commonly studied lymphoblastoid cell line

K562

Female chronic myelogenous leukemia

BJ

Skin fibroblast

HEK293

Embryonic kidney

H54

Glioblastoma

HepG2

Hepatocellular carcinoma

First, we download the DNase data for each cell line in *.narrowPeak* format:

```
hadoop fs -mkdir /user/ds/genomics/dnase
curl -s -L <...DNase URL...> \ ❶
| gunzip \ ❷
| hadoop fs -put -
/user/ds/genomics/dnase/sample.DNase.narrowPeak
[...]
```

- ❶ See accompanying code repo for actual curl commands.
- ❷ Streaming decompression.

Next, we download the ChIP-seq data for the CTCF TF, also in *.narrowPeak* format, and the GENCODE data, in GTF format:

```
hadoop fs -mkdir /user/ds/genomics/chip-seq
curl -s -L <...ChIP-seq URL...> \ ❶
| gunzip \
| hadoop fs -put - /user/ds/genomics/chip-
seq/samp.CTCF.narrowPeak
[...]
```

- ❶ See accompanying code repo for actual curl commands

Note how we unzip the stream of data with `gunzip` on the way to depositing it in HDFS. Now we download the actual human genome sequence, which is used to evaluate a position weight matrix to generate one of the features:

```
# the hg19 human genome reference sequence
curl -s -L -O \

"http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit
"
```

Finally, the conservation data is available in fixed wiggle format, which is difficult to read as a splittable file. Specifically, it is not “splittable” because it is very difficult to enter the file at an arbitrary location and start reading records. This is because the wiggle format intersperses data records with some metadata that describes the current genome location. It is not possible to predict how far back in a file a particular task must read in order to obtain the metadata about the contig coordinates. Therefore, we convert the *.wigFix* data to BED format using the BEDOPS tool on the way into HDFS as well:

```
hadoop fs -mkdir /user/ds/genomics/phylop
for i in $(seq 1 22); do
    curl -s -L <...phyloP.chr$i URL...> \ ❶
    | gunzip \
    | wig2bed -d \ ❷
    | hadoop fs -put -
"/user/ds/genomics/phylop/chr$i.phyloP.bed"
done
[...]
```

- ❶ See accompanying code repo for actual curl commands
- ❷ This command is from the BEDOPS CLI. It’s easy to write your own Python script as well

Finally, we perform a one-time conversion of the phyloP data from the text-based *.bed* format to Parquet in a Spark shell:

```
import org.bdgenomics.adam.rdd.ADAMContext._
(sc
  .loadBed("/user/ds/genomics/phylop_text")
  .saveAsParquet("/user/ds/genomics/phylop"))
```

From all of this raw data, we want to generate a training set with a schema like the following:

1. Chromosome
2. Start
3. End
4. Highest TF motif position weight matrix (PWM) score
5. Average phyloP conservation score
6. Minimum phyloP conservation score
7. Maximum phyloP conservation score
8. Distance to closest transcription start site (TSS)
9. TF identity (always “CTCF” in this case)
10. Cell line
11. TF binding status (boolean; the target variable)

This data set can easily be converted into an `RDD[LabeledPoint]` to carry into a machine learning library. Since we need to generate the data for multiple cell lines, we will define an RDD for each cell line individually and concatenate them at the end:

```
val cellLines = Vector(
  "GM12878", "K562", "BJ", "HEK293", "H54", "HepG2")
val dataByCellLine = cellLines.map(cellLine => { // For each cell
  line...
  // ...generate an RDD suitable for conversion
  // to RDD[LabeledPoint]
})
// Concatenate the RDDs and carry through into MLlib, for example
```

Before we start, we load some data that will be used throughout the computation, including conservation, transcription start sites, the human genome reference sequence, and the CTCF PWM as derived from **HT-SELEX**. We also define a couple of utility functions that will be used to generate the PWM and TSS features:


```

val hdfsPrefix = "/user/ds/genomics"
val localPrefix = "/user/ds/genomics"

// Set up broadcast variables for computing features along with
some
// utility functions

// Load the human genome reference sequence
val bHg19Data = sc.broadcast(
  new TwoBitFile(
    new LocalFileByteAccess(
      new File(Paths.get(localPrefix, "hg19.2bit").toString))))

// fn for finding closest transcription start site
// naive; exercise for reader: make this faster
def distanceToClosest(loci: Vector[Long], query: Long): Long = {
  loci.map(x => math.abs(x - query)).min
}

// CTCF PWM from https://dx.doi.org/10.1016/j.cell.2012.12.009
// generated with genomics/src/main/python/pwm.py
val bPwmData = sc.broadcast(Vector(
  Map('A' -> 0.4553, 'C' -> 0.0459, 'G' -> 0.1455, 'T' -> 0.3533),
  Map('A' -> 0.1737, 'C' -> 0.0248, 'G' -> 0.7592, 'T' -> 0.0423),
  Map('A' -> 0.0001, 'C' -> 0.9407, 'G' -> 0.0001, 'T' -> 0.0591),
  Map('A' -> 0.0051, 'C' -> 0.0001, 'G' -> 0.9879, 'T' -> 0.0069),
  Map('A' -> 0.0624, 'C' -> 0.9322, 'G' -> 0.0009, 'T' -> 0.0046),
  Map('A' -> 0.0046, 'C' -> 0.9952, 'G' -> 0.0001, 'T' -> 0.0001),
  Map('A' -> 0.5075, 'C' -> 0.4533, 'G' -> 0.0181, 'T' -> 0.0211),
  Map('A' -> 0.0079, 'C' -> 0.6407, 'G' -> 0.0001, 'T' -> 0.3513),
  Map('A' -> 0.0001, 'C' -> 0.9995, 'G' -> 0.0002, 'T' -> 0.0001),
  Map('A' -> 0.0027, 'C' -> 0.0035, 'G' -> 0.0017, 'T' -> 0.9921),
  Map('A' -> 0.7635, 'C' -> 0.0210, 'G' -> 0.1175, 'T' -> 0.0980),
  Map('A' -> 0.0074, 'C' -> 0.1314, 'G' -> 0.7990, 'T' -> 0.0622),
  Map('A' -> 0.0138, 'C' -> 0.3879, 'G' -> 0.0001, 'T' -> 0.5981),
  Map('A' -> 0.0003, 'C' -> 0.0001, 'G' -> 0.9853, 'T' -> 0.0142),
  Map('A' -> 0.0399, 'C' -> 0.0113, 'G' -> 0.7312, 'T' -> 0.2177),
  Map('A' -> 0.1520, 'C' -> 0.2820, 'G' -> 0.0082, 'T' -> 0.5578),
  Map('A' -> 0.3644, 'C' -> 0.3105, 'G' -> 0.2125, 'T' -> 0.1127)))

// compute a motif score based on the TF PWM
def scorePWM(ref: String): Double = {
  val score1 = (ref.sliding(bPwmData.value.length)
    .map(s => {
      s.zipWithIndex.map(p => bPwmData.value(p._2)
        (p._1)).product})
    .max)
  val rc = Alphabet.dna.reverseComplementExact(ref)

```

```

    val score2 = (rc.sliding(bPwmData.value.length)
      .map(s => {
        s.zipWithIndex.map(p => bPwmData.value(p._2)
      (p._1)).product}))
      .max)
    math.max(score1, score2)
  }

  // build in-memory structure for computing distance to TSS
  // we are essentially manually implementing a broadcast join here
  val tssRDD = (
    sc.loadFeatures(
      Paths.get(hdfsPrefix,
"encode.v18.annotation.gtf").toString).rdd
      .filter(_.getFeatureType == "transcript")
      .map(f => (f.getContigName, f.getStart))).rdd
      .filter(_.getFeatureType == "transcript")
      .map(f => (f.getContigName, f.getStart)))
  // this broadcast variable will hold the broadcast side of the
  "join"
  val bTssData = sc.broadcast(tssRDD
    // group by contig name
    .groupBy(_._1)
    // create Vector of TSS sites for each chromosome
    .map(p => (p._1, p._2.map(_._2.toLong).toVector))
    // collect into local in-memory structure for broadcasting
    .collect().toMap)

  // load conservation data; independent of cell line
  val phyloRDD = (
    sc.loadParquetFeatures(Paths.get(hdfsPrefix,
"phylo").toString).rdd
    // clean up a few irregularities in the phylo data
    .filter(f => f.getStart <= f.getEnd)
    .map(f => (ReferenceRegion.unstranded(f), f))).rdd
    // clean up a few irregularities in the phylo data
    .filter(f => f.getStart <= f.getEnd)
    .map(f => (ReferenceRegion.unstranded(f), f)))

```

Now that we've loaded the data necessary for defining our training examples, we define the body of the “loop” for computing the data on each cell line. Note how we read the text representations of the ChIP-seq and DNase data, because the data sets are not so large that they will hurt performance.

To do so, we load the DNase and ChIP-seq data as RDDs:

```
val dnasePath = (
  Paths.get(hdfsPrefix, s"dnase/$cellLine.DNase.narrowPeak")
    .toString)
val dnaseRDD = (sc.loadFeatures(dnasePath).rdd
  .map(f => ReferenceRegion.unstranded(f))
  .map(r => (r, r))) ❶

val chipseqPath = (
  Paths.get(hdfsPrefix, s"chip-seq/$cellLine.ChIP-
  seq.CTCF.narrowPeak")
    .toString)
val chipseqRDD = (sc.loadFeatures(chipseqPath).rdd
  .map(f => ReferenceRegion.unstranded(f))
  .map(r => (r, r))) ❶
```

❶ RDD[(ReferenceRegion, ReferenceRegion)]

The core object is a DNase hypersensitivity site, as defined by a `ReferenceRegion` object in `dnaseRDD`. Sites that overlap a ChIP-seq peak, as defined by a `ReferenceRegion` in `chipseqRDD`, have TF binding sites and are therefore labeled `true`, while the rest of the sites are labeled `false`. This is accomplished using the 1D spatial join primitives provided in the ADAM API. The join functionality requires an RDD that is keyed by a `ReferenceRegion`, and will produce tuples that have overlapping regions, according to usual join semantics (e.g., inner versus outer).

```
val dnaseWithLabelRDD = (
  LeftOuterShuffleRegionJoin(bHg19Data.value.sequences, 10000000,
    sc)
    .partitionAndJoin(dnaseRDD, chipseqRDD) ❶
    .map(p => (p._1, p._2.size)) ❷
    .reduceByKey(_ + _) ❸
    .map(p => (p._1, p._2 > 0)) ❹
    .map(p => (p._1, p)) ❺
```

❶ RDD[(ReferenceRegion, Option[ReferenceRegion])]:
there is an `Option` because we are employing a left outer join

- ② RDD[(ReferenceRegion, Int)]: 0 for None and 1 for a successful match
- ③ Aggregate all possible TF binding sites overlaying DNase site
- ④ Positive values indicate an overlap between the data sets and therefore a TF binding site
- ⑤ Prepare RDD for next join by keying it with a ReferenceRegion

Separately, we compute the conservation features by joining the DNase data to the phyloP data:

```
// given phyloP values on a site, compute some stats
def aggPhyloP(values: Vector[Double]) = {
  val avg = values.sum / values.length
  val m = values.min
  val M = values.max
  (avg, m, M)
}
val dnaseWithPhyloP RDD = (
  LeftOuterShuffleRegionJoin(bHg19Data.value.sequences, 1000000,
  sc)
  .partitionAndJoin(dnaseRDD, phyloP RDD) ①
  .filter(!_._2.isEmpty) ②
  .map(p => (p._1, p._2.get.getScore.doubleValue))
  .groupByKey() ③
  .map(p => (p._1, aggPhyloP(p._2.toVector)))) ④
```

- ① RDD[(ReferenceRegion, Option[Feature])]
- ② Filter out sites for which there is missing phyloP data
- ③ Aggregate together all phyloP values for each site
- ④ RDD[(ReferenceRegion, (Double, Double, Double))]

Now we compute the final set of features on each DNase peak by joining together the two RDDs from above and adding a few additional features by mapping over the sites:

```
// build final training example RDD
val examplesRDD = (
  InnerShuffleRegionJoin(bHg19Data.value.sequences, 1000000, sc)
  ①
  .partitionAndJoin(dnaseWithLabelRDD, dnaseWithPhyloP RDD)
  .map(tup => {
```

```

    val seq = bHg19Data.value.extract(tup._1._1) ❷
    (tup._1, tup._2, seq)})
.filter(!_.3.contains("N")) ❸
.map(tup => { ❹
    val region = tup._1._1
    val label = tup._1._2
    val contig = region.referenceName
    val start = region.start
    val end = region.end
    val phyloAvg = tup._2._1
    val phyloMin = tup._2._2
    val phyloMax = tup._2._3
    val seq = tup._3
    val pwmScore = scorePWM(seq)
    val closestTss = math.min(
        distanceToClosest(bTssData.value(contig), start),
        distanceToClosest(bTssData.value(contig), end))
    val tf = "CTCF"
    (contig, start, end, pwmScore, phyloAvg, phyloMin,
phyloMax,
        closestTss, tf, cellLine, label)))

```

- ❶ Inner join to ensure we get well-defined feature vectors
- ❷ Extract the genome sequence corresponding to this site in the genome and attach it to the tuple
- ❸ Drop any site where the genome sequence is ambiguous
- ❹ Here we build the final feature vector

This final RDD is computed in each pass of the loop over the cell lines. Finally, we union each RDD from each cell line, and cache this data in memory in preparation for training models off of it:

```

val preTrainingData = dataByCellLine.reduce(_ ++ _)
preTrainingData.cache()

preTrainingData.count() // 802059
preTrainingData.filter(_.12 == true).count() // 220344

```

At this point, the data in `preTrainingData` can be normalized and converted into an `RDD[LabeledPoint]` for training a classifier, as described in “**Random Forests**”. Note that you should perform cross-

validation, where in each fold, you hold out the data from one of the cell lines.

Querying Genotypes from the 1000 Genomes Project

In this example, we will be ingesting the full 1000 Genomes genotype data set. First, we will download the raw data directly into HDFS, unzipping in-flight, and then run an ADAM job to convert the data to Parquet. The following example command should be executed for all chromosomes, and can be parallelized across the cluster:

```
curl -s -L ftp://.../1000genomes/.../chr1.vcf.gz \ ❶  
| gunzip \  
| hadoop fs -put - /user/ds/genomics/1kg/vcf/chr1.vcf ❷  
  
adam/bin/adam-submit --master yarn --deploy-mode client \  
--driver-memory 8G --num-executors 192 --executor-cores 4 \  
--executor-memory 16G \  
-- \  
vcf2adam /user/ds/genomics/1kg/vcf  
/user/ds/genomics/1kg/parquet
```

- ❶ See the accompanying repo for the actual curl commands
- ❷ Copy the text VCF file into Hadoop

From an ADAM shell, we load and inspect an object like so:

```
import org.bdggenomics.adam.rdd.ADAMContext._  
  
val genotypesRDD =  
sc.loadGenotypes("/user/ds/genomics/1kg/parquet")  
val gt = genotypesRDD.rdd.first()  
...
```

Say we want to compute the minor allele frequency across all our samples for each variant genome-wide that overlaps a CTCF binding site. We essentially must join our CTCF data from the previous section with the

genotype data from the 1000 Genomes project. In the previous TF binding site example, we showed how to use the ADAM join machinery directly. However, when we load data through ADAM, in many cases we obtain an object that implements the `GenomicRDD` trait, which has some built-in methods for filtering and joining, as we show below:

```
import org.bdgenomics.adam.models.ReferenceRegion
import org.bdgenomics.adam.rdd.InnerTreeRegionJoin
val ctcfRDD = (sc.loadFeatures(
  "/user/ds/genomics/chip-seq/GM12878.ChIP-
seq.CTCF.narrowPeak").rdd
  .map(f => { ❶
    f.setContigName(f.getContigName.stripPrefix("chr"))
    f
  })
  .map(f => (ReferenceRegion.unstranded(f), f)))
val keyedGenotypesRDD = genotypesRDD.rdd.map(f =>
  (ReferenceRegion(f), f))
val filteredGenotypesRDD = ( ❷
  InnerTreeRegionJoin().partitionAndJoin(ctcfRDD,
  keyedGenotypesRDD)
  .map(_._2))
filteredGenotypesRDD.cache() ❸
filteredGenotypesRDD.count() // 408107700
```

- ❶ We must perform this mapping because the CTCF data uses “chr1” whereas the genotype data uses “1” to refer to the same chromosome.
- ❷ The inner join performs the filtering. We broadcast the CTCF data because it is relatively small.
- ❸ Because the computation is large, we cache the resulting filtered data to avoid recomputing it.

We also need a function that will take a `Genotype` and compute the counts of the reference/alternate alleles:

```
import python.collection.JavaConverters._
import org.bdgenomics.formats.avro.{Genotype, Variant,
GenotypeAllele}
def genotypeToAlleleCounts(gt: Genotype): (Variant, (Int, Int)) =
{
  val counts = gt.getAlleles.aspython.map(allele => { allele
  match {
```

```

        case GenotypeAllele.REF => (1, 0)
        case GenotypeAllele.ALT => (0, 1)
        case _ => (0, 0)
    })).reduce((x, y) => (x._1 + y._1, x._2 + y._2))
    (gt.getVariant, (counts._1, counts._2))
}

```

Finally, we generate the `RDD[(Variant, (Int, Int))]` and perform the aggregation:

```

val counts = filteredGenotypesRDD.map(gt => { ❶
    val counts = gt.getAlleles.aspython.map(allele => { allele
match {
    case GenotypeAllele.REF => (1, 0)
    case GenotypeAllele.ALT => (0, 1)
    case _ => (0, 0)
}}}
    })).reduce((x, y) => (x._1 + y._1, x._2 + y._2))
    (gt.getVariant, (counts._1, counts._2))
})
val countsByVariant = counts.reduceByKey(
    (x, y) => (x._1 + y._1, x._2 + y._2))
val mafByVariant = countsByVariant.map(tup => {
    val (v, (r, a)) = tup
    val n = r + a
    (v, math.min(r, a).toDouble / n)
})

```

- ❶ We write this function anonymously because of a potential problem with closure serialization when working in a shell. Spark will try to serialize everything in the shell, which can cause errors. When running as a submitted job, the named function should work fine.

The `countsByVariant` RDD stores objects of type `(Variant, (Int, Int))`, where the first member of the tuple is the specific genome variant and the second member is a pair of counts: the first is the number of reference alleles seen, while the second is the number of alternate alleles seen. The `mafByVariant` RDD stores objects of type `(Variant, Double)`, where the second member is the computed minor allele frequency from the pairs in `countsByVariant`. As an example:


```
python> countsByVariant.first._2
res21: (Int, Int) = (1655,4)

python> val mafExample = mafByVariant.first
mafExample: (org.bdgenomics.formats.avro.Variant, Double) = [...]

python> mafExample._1.getContigName -> mafExample._1.getStart
res17: (String, Long) = (X,149849811)

python> mafExample._2
res18: Double = 0.0024110910186859553
```

Traversing the entire data set is a sizable operation. Because we're only accessing a few fields from the genotype data, it would certainly benefit from predicate pushdown and projection, which we leave as an exercise to the reader. Also try running the computation on a subset of the data files if you cannot access a suitable cluster.

Where to Go from Here

Many computations in genomics fit nicely into the Spark computational paradigm. When you're performing ad hoc analysis, the most valuable contribution that projects like ADAM provide is the set of Avro schemas that represents the underlying analytical objects (along with the conversion tools). We saw how once data is converted into the corresponding Avro schemas, many large-scale computations become relatively easy to express and distribute.

While there may still be a relative dearth of tools for performing scientific research on Hadoop/Spark, there do exist a few projects that could help avoid reinventing the wheel. We explored the core functionality implemented in ADAM, but the project already has implementations for the entire GATK best-practices pipeline, including BQSR, indel realignment, and deduplication. In addition to ADAM, many institutions have signed on to the Global Alliance for Genomics and Health, which has started to generate schemas of its own for genomics analysis. The Broad Institute is now developing major software projects using Spark, including the newest

version of the **GATK4** and a new project called **Hail** for large-scale population genetics computations. The Hammerbacher lab at Mount Sinai School of Medicine has also developed **Guacamole**, a suite of tools mainly aimed at somatic variant calling for cancer genomics. All of these tools are open source, so if you start using them in your own work, please consider contributing improvements!

Chapter 9. Image similarity detection with LSH and Deep Learning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

Be it social media or e-commerce stores, images are integral to our digital lives. In fact, it was an image dataset - ImageNet - which was a key component for sparking the current deep learning (DL) revolution. A remarkable performance by a classification model in the ImageNet 2012 challenge was an important milestone and led to widespread attention. It is no wonder then that you are likely to encounter image data at some point as a data science practitioner. In this chapter, you will gain experience scaling a deep learning workflow for a visual task, namely image similarity detection, with PySpark. We covered the basics of deep learning in *chapter 6* and you can go over them before continuing.

Finding similar images comes intuitively to humans but is a complex computational task. At scale, it becomes even more difficult. In this chapter, we will introduce an approximate method for finding similar items called Locality Sensitive Hashing or LSH. We will use deep learning to convert image data into a numerical vector representation. PySpark's LSH algorithm will be applied to the resulting vectors which will allow us to find similar images given a new input image.

On a high-level, this example mirrors one of the approaches used by photo-sharing apps such as Instagram and Pinterest for image similarity detection. This helps their users make sense of the deluge of visual data that exists on their platforms. This also depicts how a deep learning workflow can benefit from PySpark's scalability.

We will start by briefly introducing PyTorch, a deep learning framework. It has gained prominence in recent years for its relatively easier learning curve compared to other major low-level DL libraries. We will then download and prepare our dataset. The dataset being used for our task is the Landmarks dataset by Google. PyTorch will be used for image preprocessing. This will be followed by conversion of our input image data into a vector representation (image embeddings). We will then import the resulting embeddings into PySpark and transform them using the LSH algorithm. We will finish off by taking a new image and performing a nearest neighbors search using our LSH-transformed dataset to find similar images.

Let's start by introducing and setting up PyTorch.

PyTorch

PyTorch is a library for building deep learning projects. It emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python. It found early adopters in the research community. Recently, it has grown into one of the most prominent deep learning tools across a broad range of applications due its ease of use. Along with Tensorflow, it is the most popular library for deep learning as of now.

PyTorch's simple and flexible interface enables fast experimentation. You can load data, apply transforms, and build models with a few lines of code. Then, you have the flexibility to write customized training, validation, and test loops and deploy trained models with ease. It is consistently being used in professional contexts for real-world, mission-critical work. Being able to use GPUs (graphical processing units) for training resource-intensive models has been a big factor for making deep learning popular. PyTorch provides great GPU support although we won't need that for our task.

Installation

On the [PyTorch website](#), you can easily obtain the installation instructions based on your system configuration, as shown in the below figure. We will use . Execute the provided command and follow the instructions for your configuration.

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			

```
$ pip3 install torch torchvision
```

We will not be relying on a GPU and hence, will choose CPU as a compute platform. In case you have a GPU setup that you want to utilise, choose options accordingly to obtain required instructions. We will not be needing torchaudio for this chapter either so we skip its installation.

Preparing the data

We will be using the [Google-Landmarks dataset](#). It can be downloaded using the script [available here](#).

Once downloaded, place them in a directory called `landmark_data`.

```
data_folder = "./landmark_data"
```

Resizing images using PyTorch

Before we head further, we'll need to preprocess our images. Preprocessing data is very common in machine learning since deep learning models (neural networks) expect the input to meet certain requirements.

We need to apply a series of preprocessing steps, called transforms, to convert input images into the proper format for the models. In our case, we need them to be 224 x 224-pixel JPEG-formatted images. We do this using PyTorch's Torchvision package in the following code:

```
from PIL import Image
from torchvision import transforms

# needed input dimensions for the CNN
input_dim = (224, 224)
input_dir_cnn = data_folder + "/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()
```

We use the Compose() transform to define a series of transforms used to preprocess our image. Here, we resize the image to fit within the neural networks.

Our dataset is in place now. In the next section, we will convert our image data into a vector representation fit for use with PySpark's LSH algorithm.

Deep Learning model for vector representation of images

Convolutional Neural Networks or CNNs are the standard neural network architecture used for prediction when the input observations are images. We won't be using them for any prediction task but for generating a vector representation of images. Specifically, we will use the ResNet-18 architecture.

Residual Network (ResNet) was introduced by Shaoqing Ren, Kaiming He, Jian Sun, and Xiangyu Zhang in their paper. The paper was named "Deep Residual Learning for Image Recognition" in 2015. The 18 in ResNet-18 stands for the number of layers that exist in the neural network architecture. Other popular variants of ResNet include 34 and 50 layers. More number of layers result in improved performance at the cost of computational costs.

Image embeddings

An image embedding, in this case, is a representation of an image in a vector space. The basic idea is that if given image is close to another image, their embedding will also be similar and close in the spatial dimension.

We can obtain image embeddings from a ResNet-34 by taking the output of its second last Fully-connected layer which has a dimension of 512. Below, we create a class that provided an image can return its numeric vector form representation.

```
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 convert images into the PyTorch tensor format
- ❷ Rescale the range of pixel values between 0 and 1. The values for the mean and standard deviation (std) were precomputed based on the data used to train the model. Normalizing the image improves the accuracy of the classifier.

We now initialize the above class and apply the `getVec` method to all of the images to obtain their respective image embeddings.

```

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()

```

For a larger dataset, you may want to sequentially write the vector output into a file rather than keeping them in memory to avoid an out-of-memory error. The data is manageable here so we create a dictionary which we save as a CSV file in the next step.

```

import pandas as pd

```



```
pd.DataFrame(allVectors).transpose().to_csv(data_folder +
'/input_data_vectors.csv')
```

Now that we have the required image embeddings, we can import them into PySpark.

Import image embeddings into PySpark

Start the PySpark shell.

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

Import the image embeddings.

```
input_df = spark.read.option('inferSchema', True).csv(data_folder
+ '/input_data_vectors.csv')
```

PySpark's LSH implementation requires a vector column as an input. We can create one by combining the relevant columns in our dataframe using the VectorAssembler transform.

```
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.printSchema()

...

root
|-- _c0: string (nullable = true)
|-- features: vector (nullable = true)
```

In the next section, we will use the LSH algorithm to create a way for us to find similar images from our dataset.

Image similarity search using LSH

We will use MinHash implementation of LSH from PySpark.

Let's first create our model object.

```
from pyspark.ml.feature import MinHashLSH

mh = MinHashLSH(inputCol="features", outputCol="hashes",
numHashTables=10)
model = mh.fit(output)
```

We now transform the input DataFrame using the newly created LSH model object. The resulting DataFrame will contain a *hashes* column containing hashed representation of the image embeddings.

```
lsh_df = model.transform(output)
lsh_df.show(3)
```

...

With our LSH-transformed dataset ready, we will now put our work to the test in next section.

Nearest neighbor search

Let's try to find a similar image using a new image. For now, we will pick one from the input dataset itself.

First, we will need to convert the input image into a vector format. We use our `Img2VecResnet18` class to do that.

```
input_dir_cnn = data_folder + "/images/input_images_cnn"

test_image = os.listdir(input_dir_cnn)[5577]
test_image = os.path.join(input_dir_cnn, test_image)
print(test_image)
display(Image.open(test_image))

img2vec = Img2VecResnet18()
I = Image.open(test_image)
```

```
test_vec = img2vec.getVec(I)
I.close()

test_vector = Vectors.dense(test_vec)
```

Now we perform an approximate nearest neighbor search.

```
print("Approximately searching lsh_df for 5 nearest neighbors of
the input vector:")
result = model.approxNearestNeighbors(lsh_df, test_vector, 5)

result.show()
```

You can check the images to see that the model gets it somewhat right already. The input image is on top of the list as one would expect.

What's next

In this chapter, we learnt how PySpark can be combined with a modern deep learning framework to scale an image similarity detection workflow.

There are multiple ways to improve this implementation. You can try using a better model or improving the preprocessing to get better quality of embeddings. Further, the LSH model can be tweaked. In a real-life setting, you may need to update the reference dataset consistently to account for new images coming into the system. The simplest way to do this is by running a batch job at periodic intervals to create new LSH models. You can explore all of these depending on your need and interest.

About the Authors

Sandy Ryza develops algorithms for public transit at Remix. Before that, he was a senior data scientist at Cloudera and Clover Health. He is an Apache Spark committer, Apache Hadoop PMC member, and founder of the Time Series for Spark project. He holds the Brown University computer science department's 2012 Twining award for "Most Chill."

Uri Laserson is an Assistant Professor of Genetics at the Icahn School of Medicine at Mount Sinai, where he develops scalable technology for genomics and immunology using the Hadoop ecosystem.

Sean Owen is Director of Data Science at Cloudera. He is an Apache Spark committer and PMC member, and was an Apache Mahout committer.

Josh Wills is the Head of Data Engineering at Slack, the founder of the Apache Crunch project, and wrote a tweet about data scientists once.



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se

singlelogin.re

go-to-zlibrary.se

single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>