Tomasz Drabas, Denny Lee

Foreword by: Holden Karau
Principal Software Engineer at IBM Spark Technology Center

Learning PySpark

Build data-intensive applications locally and deploy at scale using the combined powers of Python and Spark 2.0

Packt

Learning PySpark

Build data-intensive applications locally and deploy at scale using the combined powers of Python and Spark 2.0

Tomasz Drabas

Denny Lee

Packt

BIRMINGHAM - MUMBAI

Learning PySpark

First published: February 2017

Production reference: 1220217

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78646-370-8

`www.packtpub.com`

Credits

Authors
Tomasz Drabas
Denny Lee

Reviewer
Holden Karau

Commissioning Editor
Amey Varangaonkar

Acquisition Editor
Prachi Bisht

Content Development Editor
Amrita Noronha

Technical Editor
Akash Patel

Copy Editor
Safis Editing

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Aishwarya Gangawane

Graphics
Disha Haria

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

Foreword

Thank you for choosing this book to start your PySpark adventures, I hope you are as excited as I am. When Denny Lee first told me about this new book I was delighted-one of the most important things that makes Apache Spark such a wonderful platform, is supporting both the Java/Scala/JVM worlds and Python (and more recently R) worlds. Many of the previous books for Spark have been focused on either all of the core languages, or primarily focused on JVM languages, so it's great to see PySpark get its chance to shine with a dedicated book from such experienced Spark educators. By supporting both of these different worlds, we are able to more effectively work together as Data Scientists and Data Engineers, while stealing the best ideas from each other's communities.

It has been a privilege to have the opportunity to review early versions of this book, which has only increased my excitement for the project. I've had the privilege of being at some of the same conferences and meetups and watching the authors introduce new concepts in the world of Spark to a variety of audiences (from first timers to old hands), and they've done a great job distilling their experience for this book. The experience of the authors shines through with everything from their explanations to the topics covered. Beyond simply introducing PySpark they have also taken the time to look at *up and coming* packages from the community, such as GraphFrames and TensorFrames.

I think the community is one of those often-overlooked components when deciding what tools to use, and Python has a great community and I'm looking forward to you joining the Python Spark community. So, enjoy your adventure; I know you are in good hands with Denny Lee and Tomek Drabas. I truly believe that by having a diverse community of Spark users we will be able to make better tools useful for everyone, so I hope to see you around at one of the conferences, meetups, or mailing lists soon :)

Holden Karau

P.S.

I owe Denny a beer; if you want to buy him a Bud Light lime (or lime-a-rita) for me I'd be much obliged (although he might not be quite as amused as I am).

About the Authors

Tomasz Drabas is a Data Scientist working for Microsoft and currently residing in the Seattle area. He has over 13 years of experience in data analytics and data science in numerous fields: advanced technology, airlines, telecommunications, finance, and consulting he gained while working on three continents: Europe, Australia, and North America. While in Australia, Tomasz has been working on his PhD in Operations Research with a focus on choice modeling and revenue management applications in the airline industry.

At Microsoft, Tomasz works with big data on a daily basis, solving machine learning problems such as anomaly detection, churn prediction, and pattern recognition using Spark.

Tomasz has also authored the *Practical Data Analysis Cookbook* published by Packt Publishing in 2016.

I would like to thank my family: Rachel, Skye, and Albert—you are the love of my life and I cherish every day I spend with you! Thank you for always standing by me and for encouraging me to push my career goals further and further. Also, to my family and my in-laws for putting up with me (in general).

There are many more people that have influenced me over the years that I would have to write another book to thank them all. You know who you are and I want to thank you from the bottom of my heart!

However, I would not have gotten through my PhD if it was not for Czesia Wieruszewska; Czesiu - dziękuję za Twoją pomoc bez której nie rozpocząłbym mojej podróży po Antypodach. Along with Krzys Krzysztoszek, you guys have always believed in me! Thank you!

Denny Lee is a Principal Program Manager at Microsoft for the Azure DocumentDB team—Microsoft's blazing fast, planet-scale managed document store service. He is a hands-on distributed systems and data science engineer with more than 18 years of experience developing Internet-scale infrastructure, data platforms, and predictive analytics systems for both on-premise and cloud environments.

He has extensive experience of building greenfield teams as well as turnaround/ change catalyst. Prior to joining the Azure DocumentDB team, Denny worked as a Technology Evangelist at Databricks; he has been working with Apache Spark since 0.5. He was also the Senior Director of Data Sciences Engineering at Concur, and was on the incubation team that built Microsoft's Hadoop on Windows and Azure service (currently known as HDInsight). Denny also has a Masters in Biomedical Informatics from Oregon Health and Sciences University and has architected and implemented powerful data solutions for enterprise healthcare customers for the last 15 years.

I would like to thank my wonderful spouse, Hua-Ping, and my awesome daughters, Isabella and Samantha. You are the ones who keep me grounded and help me reach for the stars!

About the Reviewer

Holden Karau is transgender Canadian, and an active open source contributor. When not in San Francisco working as a software development engineer at IBM's Spark Technology Center, Holden talks internationally on Spark and holds office hours at coffee shops at home and abroad. Holden is a co-author of numerous books on Spark including High Performance Spark (which she believes is the gift of the season for those with expense accounts) & Learning Spark. Holden is a Spark committer, specializing in PySpark and Machine Learning. Prior to IBM she worked on a variety of distributed, search, and classification problems at Alpine, Databricks, Google, Foursquare, and Amazon. She graduated from the University of Waterloo with a Bachelor of Mathematics in Computer Science. Outside of software she enjoys playing with fire, welding, scooters, poutine, and dancing.

Customer Feedback

Table of Contents

Preface

It is estimated that in 2013 the whole world produced around 4.4 zettabytes of data; that is, 4.4 *billion* terabytes! By 2020, we (as the human race) are expected to produce ten times that. With data getting larger literally by the second, and given the growing appetite for making sense out of it, in 2004 Google employees Jeffrey Dean and Sanjay Ghemawat published the seminal paper *MapReduce: Simplified Data Processing on Large Clusters*. Since then, technologies leveraging the concept started growing very quickly with Apache Hadoop initially being the most popular. It ultimately created a Hadoop ecosystem that included abstraction layers such as Pig, Hive, and Mahout – all leveraging this simple concept of map and reduce.

However, even though capable of chewing through petabytes of data daily, MapReduce is a fairly restricted programming framework. Also, most of the tasks require reading and writing to disk. Seeing these drawbacks, in 2009 Matei Zaharia started working on Spark as part of his PhD. Spark was first released in 2012. Even though Spark is based on the same MapReduce concept, its advanced ways of dealing with data and organizing tasks make it 100x faster than Hadoop (for in-memory computations).

In this book, we will guide you through the latest incarnation of Apache Spark using Python. We will show you how to read structured and unstructured data, how to use some fundamental data types available in PySpark, build machine learning models, operate on graphs, read streaming data, and deploy your models in the cloud. Each chapter will tackle different problem, and by the end of the book we hope you will be knowledgeable enough to solve other problems we did not have space to cover here.

What this book covers

Chapter 1, *Understanding Spark*, provides an introduction into the Spark world with an overview of the technology and the jobs organization concepts.

Chapter 2, *Resilient Distributed Datasets*, covers RDDs, the fundamental, schema-less data structure available in PySpark.

Chapter 3, *DataFrames*, provides a detailed overview of a data structure that bridges the gap between Scala and Python in terms of efficiency.

Chapter 4, *Prepare Data for Modeling*, guides the reader through the process of cleaning up and transforming data in the Spark environment.

Chapter 5, *Introducing MLlib*, introduces the machine learning library that works on RDDs and reviews the most useful machine learning models.

Chapter 6, *Introducing the ML Package*, covers the current mainstream machine learning library and provides an overview of all the models currently available.

Chapter 7, *GraphFrames*, will guide you through the new structure that makes solving problems with graphs easy.

Chapter 8, *TensorFrames*, introduces the bridge between Spark and the Deep Learning world of TensorFlow.

Chapter 9, *Polyglot Persistence with Blaze*, describes how Blaze can be paired with Spark for even easier abstraction of data from various sources.

Chapter 10, *Structured Streaming*, provides an overview of streaming tools available in PySpark.

Chapter 11, *Packaging Spark Applications*, will guide you through the steps of modularizing your code and submitting it for execution to Spark through command-line interface.

For more information, we have provided two bonus chapters as follows:

Installing Spark: `https://www.packtpub.com/sites/default/files/downloads/InstallingSpark.pdf`

Free Spark Cloud Offering: `https://www.packtpub.com/sites/default/files/downloads/FreeSparkCloudOffering.pdf`

What you need for this book

For this book you need a personal computer (can be either Windows machine, Mac, or Linux). To run Apache Spark, you will need Java 7+ and an installed and configured Python 2.6+ or 3.4+ environment; we use the Anaconda distribution of Python in version 3.5, which can be downloaded from `https://www.continuum.io/downloads`.

The Python modules we randomly use throughout the book come preinstalled with Anaconda. We also use GraphFrames and TensorFrames that can be loaded dynamically while starting a Spark instance: to load these you just need an Internet connection. It is fine if some of those modules are not currently installed on your machine – we will guide you through the installation process.

Who this book is for

This book is for everyone who wants to learn the fastest-growing technology in big data: Apache Spark. We hope that even the more advanced practitioners from the field of data science can find some of the examples refreshing and the more advanced topics interesting.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

A block of code is set as follows:

```
data = sc.parallelize(
    [('Amber', 22), ('Alfred', 23), ('Skye',4), ('Albert', 12),
     ('Amber', 9)])
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
rdd1 = sc.parallelize([('a', 1), ('b', 4), ('c',10)])
rdd2 = sc.parallelize([('a', 4), ('a', 1), ('b', '6'), ('d', 15)])
rdd3 = rdd1.leftOuterJoin(rdd2)
```

Any command-line input or output is written as follows:

```
java -version
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

All the code is also available on GitHub: `https://github.com/drabastomek/learningPySpark`.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Learning-PySpark`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/LearningPySpark_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1 Understanding Spark

Apache Spark is a powerful open source processing engine originally developed by Matei Zaharia as a part of his PhD thesis while at UC Berkeley. The first version of Spark was released in 2012. Since then, in 2013, Zaharia co-founded and has become the CTO at Databricks; he also holds a professor position at Stanford, coming from MIT. At the same time, the Spark codebase was donated to the Apache Software Foundation and has become its flagship project.

Apache Spark is fast, easy to use framework, that allows you to solve a wide variety of complex data problems whether semi-structured, structured, streaming, and/or machine learning / data sciences. It also has become one of the largest open source communities in big data with more than 1,000 contributors from 250+ organizations and with 300,000+ Spark Meetup community members in more than 570+ locations worldwide.

In this chapter, we will provide a primer to understanding Apache Spark. We will explain the concepts behind Spark Jobs and APIs, introduce the Spark 2.0 architecture, and explore the features of Spark 2.0.

The topics covered are:

- What is Apache Spark?
- Spark Jobs and APIs
- Review of Resilient Distributed Datasets (RDDs), DataFrames, and Datasets
- Review of Catalyst Optimizer and Project Tungsten
- Review of the Spark 2.0 architecture

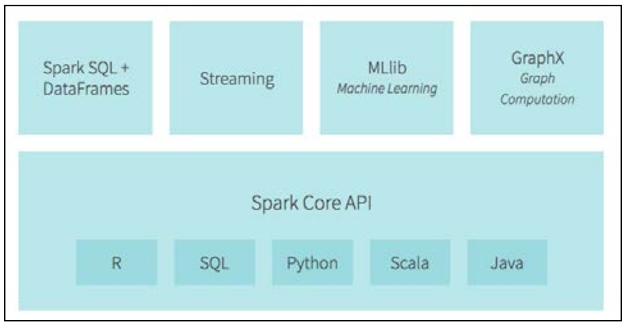
What is Apache Spark?

Apache Spark is an open-source powerful distributed querying and processing engine. It provides flexibility and extensibility of MapReduce but at significantly higher speeds: Up to 100 times faster than Apache Hadoop when data is stored in memory and up to 10 times when accessing disk.

Apache Spark allows the user to read, transform, and aggregate data, as well as train and deploy sophisticated statistical models with ease. The Spark APIs are accessible in Java, Scala, Python, R and SQL. Apache Spark can be used to build applications or package them up as libraries to be deployed on a cluster or perform *quick* analytics interactively through notebooks (like, for instance, Jupyter, Spark-Notebook, Databricks notebooks, and Apache Zeppelin).

Apache Spark exposes a host of libraries familiar to data analysts, data scientists or researchers who have worked with Python's `pandas` or R's `data.frames` or `data.tables`. It is important to note that while Spark DataFrames will be *familiar* to `pandas` or `data.frames` / `data.tables` users, there are some differences so please temper your expectations. Users with more of a SQL background can use the language to shape their data as well. Also, delivered with Apache Spark are several already implemented and tuned algorithms, statistical models, and frameworks: MLlib and ML for machine learning, GraphX and GraphFrames for graph processing, and Spark Streaming (DStreams and Structured). Spark allows the user to combine these libraries seamlessly in the same application.

Apache Spark can easily run locally on a laptop, yet can also easily be deployed in standalone mode, over YARN, or Apache Mesos - either on your local cluster or in the cloud. It can read and write from a diverse data sources including (but not limited to) HDFS, Apache Cassandra, Apache HBase, and S3:



Source: Apache Spark is the smartphone of Big Data `http://bit.ly/1QsgaNj`

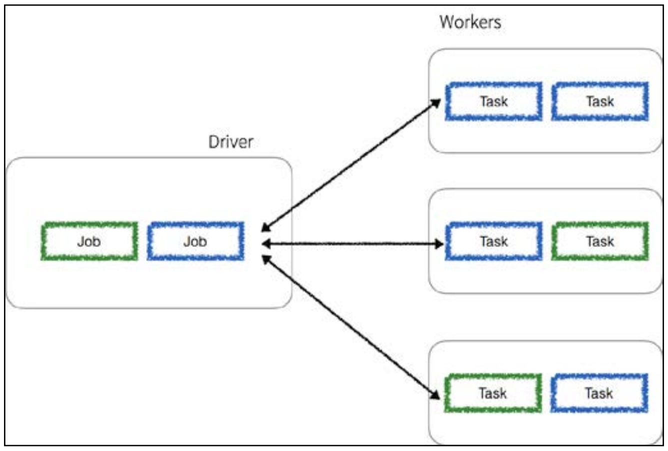
For more information, please refer to: Apache Spark is the Smartphone of Big Data at `http://bit.ly/1QsgaNj`

Spark Jobs and APIs

In this section, we will provide a short overview of the Apache Spark Jobs and APIs. This provides the necessary foundation for the subsequent section on Spark 2.0 architecture.

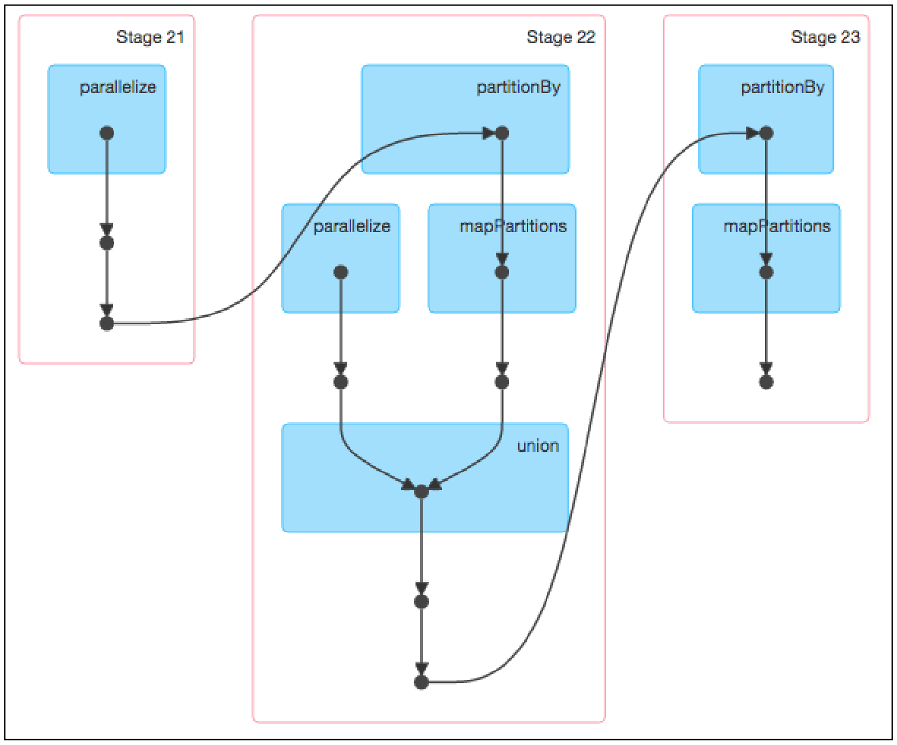
Execution process

Any Spark application spins off a single driver process (that can contain multiple jobs) on the *master* node that then directs executor processes (that contain multiple tasks) distributed to a number of *worker* nodes as noted in the following diagram:



The driver process determines the number and the composition of the task processes directed to the executor nodes based on the graph generated for the given job. Note, that any worker node can execute tasks from a number of different jobs.

A Spark job is associated with a chain of object dependencies organized in a direct acyclic graph (DAG) such as the following example generated from the Spark UI. Given this, Spark can optimize the scheduling (for example, determine the number of tasks and workers required) and execution of these tasks:



For more information on the DAG scheduler, please refer to `http://bit.ly/29WTiK8`.

Resilient Distributed Dataset

Apache Spark is built around a distributed collection of immutable Java Virtual Machine (JVM) objects called Resilient Distributed Datasets (RDDs for short). As we are working with Python, it is important to note that the Python data is stored within these JVM objects. More of this will be discussed in the subsequent chapters on RDDs and DataFrames. These objects allow any job to perform calculations very quickly. RDDs are calculated against, cached, and stored in-memory: a scheme that results in orders of magnitude faster computations compared to other traditional distributed frameworks like Apache Hadoop.

At the same time, RDDs expose some coarse-grained transformations (such as `map(...)`, `reduce(...)`, and `filter(...)` which we will cover in greater detail in Chapter 2, *Resilient Distributed Datasets*), keeping the flexibility and extensibility of the Hadoop platform to perform a wide variety of calculations. RDDs apply and log transformations to the data in parallel, resulting in both increased speed and fault-tolerance. By registering the transformations, RDDs provide data lineage - a form of an ancestry tree for each intermediate step in the form of a graph. This, in effect, guards the RDDs against data loss - if a partition of an RDD is lost it still has enough information to recreate that partition instead of simply depending on replication.

If you want to learn more about data lineage check this link `http://ibm.co/2ao9B1t`.

RDDs have two sets of parallel operations: *transformations* (which return pointers to new RDDs) and *actions* (which return values to the driver after running a computation); we will cover these in greater detail in later chapters.

For the latest list of transformations and actions, please refer to the Spark Programming Guide at `http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations`.

RDD transformation operations are *lazy* in a sense that they do not compute their results immediately. The transformations are only computed when an action is executed and the results need to be returned to the driver. This delayed execution results in more fine-tuned queries: Queries that are optimized for performance. This optimization starts with Apache Spark's DAGScheduler – the stage oriented scheduler that transforms using *stages* as seen in the preceding screenshot. By having separate RDD *transformations* and *actions*, the DAGScheduler can perform optimizations in the query including being able to avoid *shuffling*, the data (the most resource intensive task).

For more information on the DAGScheduler and optimizations (specifically around narrow or wide dependencies), a great reference is the *Narrow vs. Wide Transformations* section in *High Performance Spark* in *Chapter 5, Effective Transformations* (`https://smile.amazon.com/High-Performance-Spark-Practices-Optimizing/dp/1491943203`).

DataFrames

DataFrames, like RDDs, are immutable collections of data distributed among the nodes in a cluster. However, unlike RDDs, in DataFrames data is organized into named columns.

[If you are familiar with Python's `pandas` or R `data.frames`, this is a similar concept.]

DataFrames were designed to make large data sets processing even easier. They allow developers to formalize the structure of the data, allowing higher-level abstraction; in that sense DataFrames resemble tables from the relational database world. DataFrames provide a domain specific language API to manipulate the distributed data and make Spark accessible to a wider audience, beyond specialized data engineers.

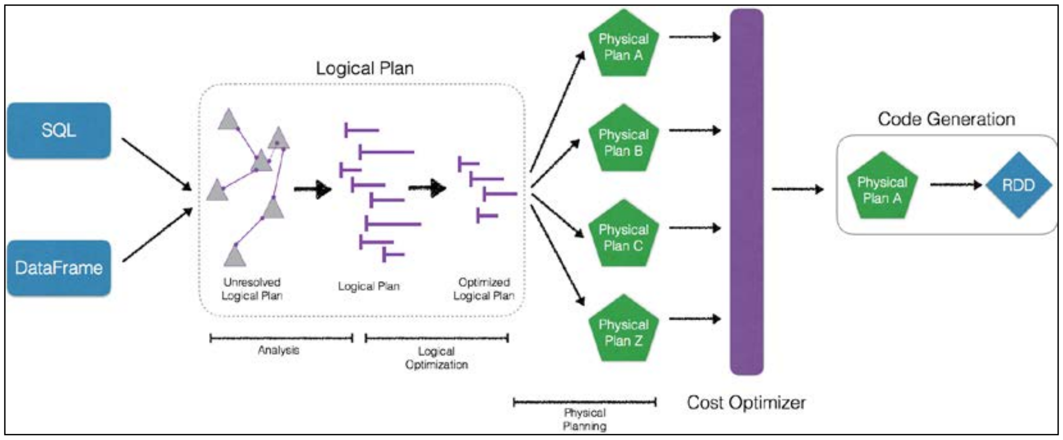
One of the major benefits of DataFrames is that the Spark engine initially builds a logical execution plan and executes generated code based on a physical plan determined by a cost optimizer. Unlike RDDs that can be significantly slower on Python compared with Java or Scala, the introduction of DataFrames has brought performance parity across all the languages.

Datasets

Introduced in Spark 1.6, the goal of Spark Datasets is to provide an API that allows users to easily express transformations on domain objects, while also providing the performance and benefits of the robust Spark SQL execution engine. Unfortunately, at the time of writing this book Datasets are only available in Scala or Java. When they are available in PySpark we will cover them in future editions.

Catalyst Optimizer

Spark SQL is one of the most technically involved components of Apache Spark as it powers both SQL queries and the DataFrame API. At the core of Spark SQL is the Catalyst Optimizer. The optimizer is based on functional programming constructs and was designed with two purposes in mind: To ease the addition of new optimization techniques and features to Spark SQL and to allow external developers to extend the optimizer (for example, adding data source specific rules, support for new data types, and so on):



For more information, check out *Deep Dive into Spark SQL's Catalyst Optimizer* (`http://bit.ly/271I7Dk`) and *Apache Spark DataFrames: Simple and Fast Analysis of Structured Data* (`http://bit.ly/29QbcOV`)

Project Tungsten

Tungsten is the codename for an umbrella project of Apache Spark's execution engine. The project focuses on improving the Spark algorithms so they use memory and CPU more efficiently, pushing the performance of modern hardware closer to its limits.

The efforts of this project focus, among others, on:

- Managing memory explicitly so the overhead of JVM's object model and garbage collection are eliminated
- Designing algorithms and data structures that exploit the memory hierarchy
- Generating code in runtime so the applications can exploit modern compliers and optimize for CPUs
- Eliminating virtual function dispatches so that multiple CPU calls are reduced
- Utilizing low-level programming (for example, loading immediate data to CPU registers) to speed up the memory access and optimizing Spark's engine to efficiently compile and execute simple loops

For more information, please refer to

Project Tungsten: Bringing Apache Spark Closer to Bare Metal (`https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`)

Deep Dive into Project Tungsten: Bringing Spark Closer to Bare Metal [SSE 2015 Video and Slides] (`https://spark-summit.org/2015/events/deep-dive-into-project-tungsten-bringing-spark-closer-to-bare-metal/`) and

Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop (`https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`)

Spark 2.0 architecture

The introduction of Apache Spark 2.0 is the recent major release of the Apache Spark project based on the key learnings from the last two years of development of the platform:

Source: Apache Spark 2.0: Faster, Easier, and Smarter `http://bit.ly/2ap7qd5`

The three overriding themes of the Apache Spark 2.0 release surround performance enhancements (via Tungsten Phase 2), the introduction of structured streaming, and unifying Datasets and DataFrames. We will describe the Datasets as they are part of Spark 2.0 even though they are currently only available in Scala and Java.

Refer to the following presentations by key Spark committers for more information about Apache Spark 2.0:

Reynold Xin's Apache Spark 2.0: Faster, Easier, and Smarter webinar `http://bit.ly/2ap7qd5`

Michael Armbrust's Structuring Spark: DataFrames, Datasets, and Streaming `http://bit.ly/2ap7qd5`

Tathagata Das' A Deep Dive into Spark Streaming `http://bit.ly/2aHt1w0`

Joseph Bradley's Apache Spark MLlib 2.0 Preview: Data Science and Production `http://bit.ly/2aHrOVN`

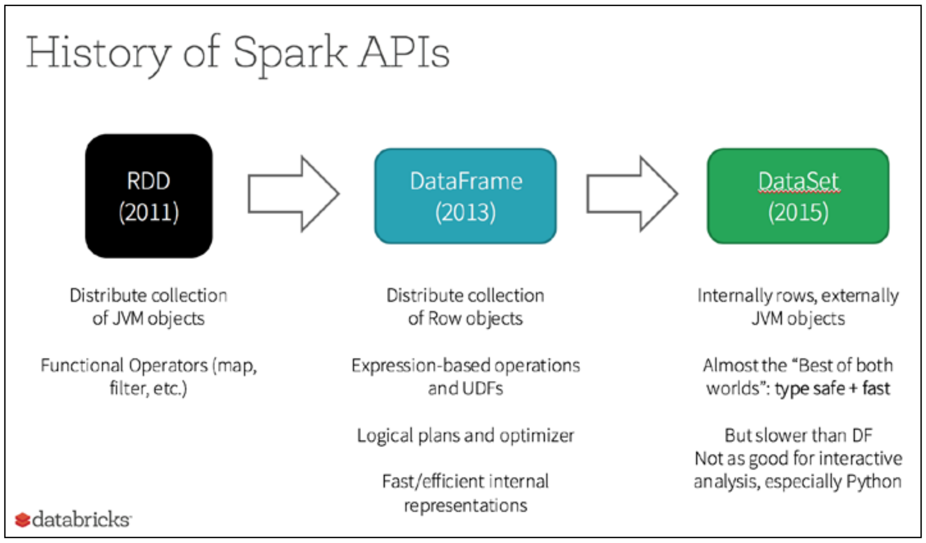
Unifying Datasets and DataFrames

In the previous section, we stated out that Datasets (at the time of writing this book) are only available in Scala or Java. However, we are providing the following context to better understand the direction of Spark 2.0.

Datasets were introduced in 2015 as part of the Apache Spark 1.6 release. The goal for datasets was to provide a type-safe, programming interface. This allowed developers to work with semi-structured data (like JSON or key-value pairs) with compile time type safety (that is, production applications can be checked for errors before they run). Part of the reason why Python does not implement a Dataset API is because Python is not a type-safe language.

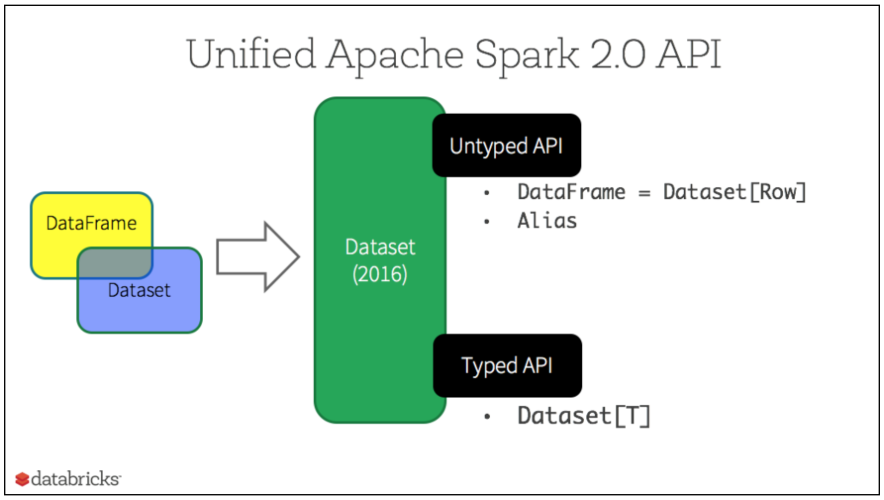
Just as important, the Datasets API contain high-level domain specific language operations such as `sum()`, `avg()`, `join()`, and `group()`. This latter trait means that you have the flexibility of traditional Spark RDDs but the code is also easier to express, read, and write. Similar to DataFrames, Datasets can take advantage of Spark's catalyst optimizer by exposing expressions and data fields to a query planner and making use of Tungsten's fast in-memory encoding.

The history of the Spark APIs is denoted in the following diagram noting the progression from RDD to DataFrame to Dataset:


History of Spark APIs
RDD (2011)
DataFrame (2013)
DataSet (2015)
Distribute collection of JVM objects
Functional Operators (map, filter, etc.)
Distribute collection of Row objects
Expression-based operations and UDFs
Logical plans and optimizer
Fast/efficient internal representations
Internally rows, externally JVM objects
Almost the "Best of both worlds": type safe + fast
But slower than DF
Not as good for interactive analysis, especially Python
databricks

Source: From Webinar Apache Spark 1.5: What is the difference between a DataFrame and a RDD? `http://bit.ly/29JPJSA`

The unification of the DataFrame and Dataset APIs has the potential of creating breaking changes to backwards compatibility. This was one of the main reasons Apache Spark 2.0 was a major release (as opposed to a 1.x minor release which would have minimized any breaking changes). As you can see from the following diagram, DataFrame and Dataset both belong to the new Dataset API introduced as part of Apache Spark 2.0:



Source: A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets `http://bit.ly/2accSNA`

As noted previously, the Dataset API provides a type-safe, object-oriented programming interface. Datasets can take advantage of the Catalyst optimizer by exposing expressions and data fields to the query planner and Project Tungsten's Fast In-memory encoding. But with DataFrame and Dataset now unified as part of Apache Spark 2.0, DataFrame is now an alias for the Dataset Untyped API. More specifically:

```
DataFrame = Dataset[Row]
```

Introducing SparkSession

In the past, you would potentially work with `SparkConf`, `SparkContext`, `SQLContext`, and `HiveContext` to execute your various Spark queries for configuration, Spark context, SQL context, and Hive context respectively. The `SparkSession` is essentially the combination of these contexts including `StreamingContext`.

For example, instead of writing:

```
df = sqlContext.read \
    .format('json').load('py/test/sql/people.json')
```

now you can write:

```
df = spark.read.format('json').load('py/test/sql/people.json')
```

or:

```
df = spark.read.json('py/test/sql/people.json')
```

The `SparkSession` is now the entry point for reading data, working with metadata, configuring the session, and managing the cluster resources.

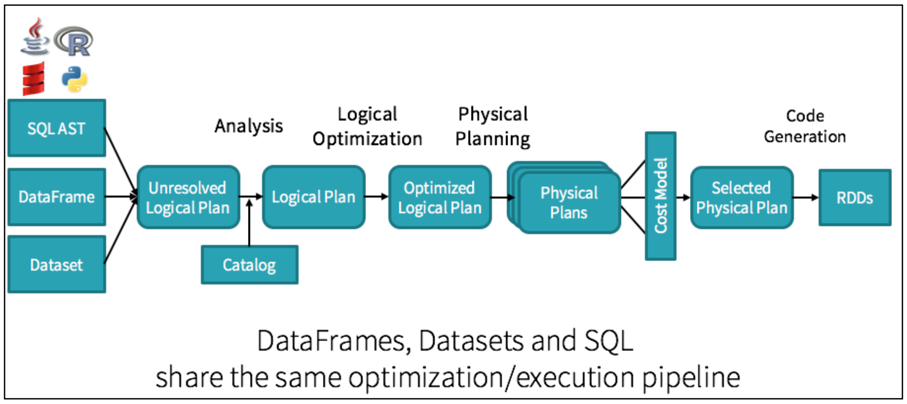
Tungsten phase 2

The fundamental observation of the computer hardware landscape when the project started was that, while there were improvements in *price per performance* in RAM memory, disk, and (to an extent) network interfaces, the *price per performance* advancements for CPUs were not the same. Though hardware manufacturers could put more cores in each socket (i.e. improve performance through parallelization), there were no significant improvements in the actual core speed.

Project Tungsten was introduced in 2015 to make significant changes to the Spark engine with the focus on improving performance. The first phase of these improvements focused on the following facets:

- **Memory Management and Binary Processing**: Leveraging application semantics to manage memory explicitly and eliminate the overhead of the JVM object model and garbage collection
- **Cache-aware computation**: Algorithms and data structures to exploit memory hierarchy
- **Code generation**: Using code generation to exploit modern compilers and CPUs

The following diagram is the updated Catalyst engine to denote the inclusion of Datasets. As you see at the right of the diagram (right of the Cost Model), **Code Generation** is used against the selected physical plans to generate the underlying RDDs:



Source: Structuring Spark: DataFrames, Datasets, and Streaming `http://bit.ly/2cJ508x`

As part of Tungsten Phase 2, there is the push into *whole-stage* code generation. That is, the Spark engine will now generate the byte code at compile time for the entire Spark stage instead of just for specific jobs or tasks. The primary facets surrounding these improvements include:

- **No virtual function dispatches**: This reduces multiple CPU calls that can have a profound impact on performance when dispatching billions of times
- **Intermediate data in memory vs CPU registers**: Tungsten Phase 2 places intermediate data into CPU registers. This is an order of magnitude reduction in the number of cycles to obtain data from the CPU registers instead of from memory
- **Loop unrolling and SIMD**: Optimize Apache Spark's execution engine to take advantage of modern compilers and CPUs' ability to efficiently compile and execute simple `for` loops (as opposed to complex function call graphs)

For a more in-depth review of Project Tungsten, please refer to:

- *Apache Spark Key Terms, Explained* `https://databricks.com/blog/2016/06/22/apache-spark-key-terms-explained.html`
- *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop* `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`

- *Project Tungsten: Bringing Apache Spark Closer to Bare Metal* `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`

Structured Streaming

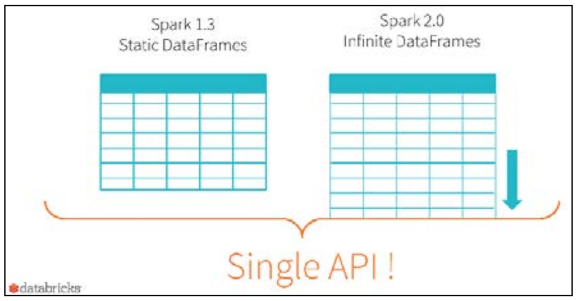
As quoted by Reynold Xin during Spark Summit East 2016:

> "*The simplest way to perform streaming analytics is not having to reason about streaming.*"

This is the underlying foundation for building Structured Streaming. While streaming is powerful, one of the key issues is that streaming can be difficult to build and maintain. While companies such as Uber, Netflix, and Pinterest have Spark Streaming applications running in production, they also have dedicated teams to ensure the systems are highly available.

> For a high-level overview of Spark Streaming, please review Spark Streaming: What Is It and Who's Using It? `http://bit.ly/1Qb10f6`

As implied previously, there are many things that can go wrong when operating Spark Streaming (and any streaming system for that matter) including (but not limited to) late events, partial outputs to the final data source, state recovery on failure, and/or distributed reads/writes:

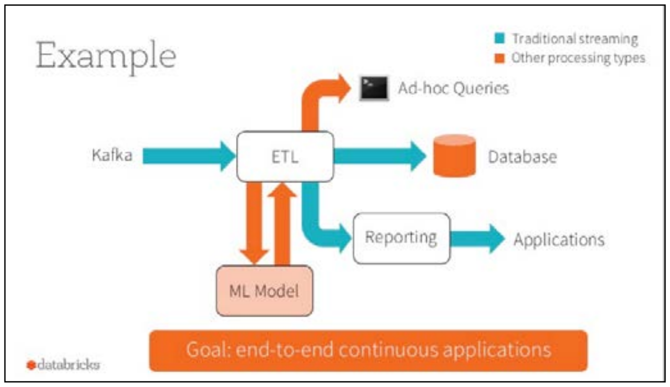


Source: A Deep Dive into Structured Streaming `http://bit.ly/2aHt1w0`

Therefore, to simplify Spark Streaming, there is now a single API that addresses both batch and streaming within the Apache Spark 2.0 release. More succinctly, the high-level streaming API is now built on top of the Apache Spark SQL Engine. It runs the same queries as you would with Datasets/DataFrames providing you with all the performance and optimization benefits as well as benefits such as event time, windowing, sessions, sources, and sinks.

Continuous applications

Altogether, Apache Spark 2.0 not only unified DataFrames and Datasets but also unified streaming, interactive, and batch queries. This opens a whole new set of use cases including the ability to aggregate data into a stream and then serving it using traditional JDBC/ODBC, to change queries at run time, and/or to build and apply ML models in for many scenario in a variety of latency use cases:



Source: Apache Spark Key Terms, Explained `https://databricks.com/blog/2016/06/22/apache-spark-key-terms-explained.html`.

Together, you can now build end-to-end **continuous applications**, in which you can issue the same queries to batch processing as to real-time data, perform ETL, generate reports, update or track specific data in the stream.

For more information on continuous applications, please refer to Matei Zaharia's blog post *Continuous Applications: Evolving Streaming in Apache Spark 2.0 - A foundation for end-to-end real-time applications* `http://bit.ly/2aJaSOr`.

Summary

In this chapter, we reviewed what is Apache Spark and provided a primer on Spark Jobs and APIs. We also provided a primer on Resilient Distributed Datasets (RDDs), DataFrames, and Datasets; we will dive further into RDDs and DataFrames in subsequent chapters. We also discussed how DataFrames can provide faster query performance in Apache Spark due to the Spark SQL Engine's Catalyst Optimizer and Project Tungsten. Finally, we also provided a high-level overview of the Spark 2.0 architecture including the Tungsten Phase 2, Structured Streaming, and Unifying DataFrames and Datasets.

In the next chapter, we will cover one of the fundamental data structures in Spark: The Resilient Distributed Datasets, or RDDs. We will show you how to create and modify these schema-less data structures using transformers and actions so your journey with PySpark can begin.

Before we do that, however, please, check the link `http://www.tomdrabas.com/site/book` for the Bonus Chapter 1 where we outline instructions on how to install Spark locally on your machine (unless you already have it installed). Here's a direct link to the manual: `https://www.packtpub.com/sites/default/files/downloads/InstallingSpark.pdf`.

2 Resilient Distributed Datasets

Resilient Distributed Datasets (RDDs) are a distributed collection of immutable JVM objects that allow you to perform calculations very quickly, and they are the *backbone* of Apache Spark.

As the name suggests, the dataset is distributed; it is split into chunks based on some key and distributed to executor nodes. Doing so allows for running calculations against such datasets very quickly. Also, as already mentioned in *Chapter 1, Understanding Spark*, RDDs keep track (log) of all the transformations applied to each chunk to speed up the computations and provide a fallback if things go wrong and that portion of the data is lost; in such cases, RDDs can recompute the data. This data lineage is another line of defense against data loss, a complement to data replication.

The following topics are covered in this chapter:

- Internal workings of an RDD
- Creating RDDs
- Global versus local scopes
- Transformations
- Actions

Internal workings of an RDD

RDDs operate in parallel. This is the strongest advantage of working in Spark: Each transformation is executed in parallel for enormous increase in speed.

The transformations to the dataset are lazy. This means that any transformation is only executed when an action on a dataset is called. This helps Spark to optimize the execution. For instance, consider the following very common steps that an analyst would normally do to get familiar with a dataset:

1. Count the occurrence of distinct values in a certain column.
2. Select those that start with an `A`.
3. Print the results to the screen.

As simple as the previously mentioned steps sound, if only items that start with the letter `A` are of interest, there is no point in counting distinct values for all the other items. Thus, instead of following the execution as outlined in the preceding points, Spark could only count the items that start with `A`, and then print the results to the screen.

Let's break this example down in code. First, we order Spark to map the values of `A` using the `.map(lambda v: (v, 1))` method, and then select those records that start with an `'A'` (using the `.filter(lambda val: val.startswith('A'))` method). If we call the `.reduceByKey(operator.add)` method it will reduce the dataset and *add* (in this example, count) the number of occurrences of each key. All of these steps **transform** the dataset.

Second, we call the `.collect()` method to execute the steps. This step is an **action** on our dataset - it finally counts the distinct elements of the dataset. In effect, the action might reverse the order of transformations and filter the data first before mapping, resulting in a smaller dataset being passed to the reducer.

Do not worry if you do not understand the previous commands yet - we will explain them in detail later in this chapter.

Creating RDDs

There are two ways to create an RDD in PySpark: you can either `.parallelize(...)` a collection (`list` or an `array` of some elements):

```
data = sc.parallelize(
    [('Amber', 22), ('Alfred', 23), ('Skye',4), ('Albert', 12),
     ('Amber', 9)])
```

Or you can reference a file (or files) located either locally or somewhere externally:

```
data_from_file = sc.\
    textFile(
        '/Users/drabast/Documents/PySpark_Data/VS14MORT.txt.gz',
        4)
```

We downloaded the Mortality dataset `VS14MORT.txt` file from (accessed on July 31, 2016) `ftp://ftp.cdc.gov/pub/Health_Statistics/NCHS/Datasets/DVS/mortality/mort2014us.zip`; the record schema is explained in this document `http://www.cdc.gov/nchs/data/dvs/Record_Layout_2014.pdf`. We selected this dataset on purpose: The encoding of the records will help us to explain how to use UDFs to transform your data later in this chapter. For your convenience, we also host the file here: `http://tomdrabas.com/data/VS14MORT.txt.gz`

The last parameter in `sc.textFile(..., n)` specifies the number of partitions the dataset is divided into.

A rule of thumb would be to break your dataset into two-four partitions for each in your cluster.

Spark can read from a multitude of filesystems: Local ones such as NTFS, FAT, or Mac OS Extended (HFS+), or distributed filesystems such as HDFS, S3, Cassandra, among many others.

Be wary where your datasets are read from or saved to: The path cannot contain special characters `[]`. Note, that this also applies to paths stored on Amazon S3 or Microsoft Azure Data Storage.

Multiple data formats are supported: Text, parquet, JSON, Hive tables, and data from relational databases can be read using a JDBC driver. Note that Spark can automatically work with compressed datasets (like the Gzipped one in our preceding example).

Depending on how the data is read, the object holding it will be represented slightly differently. The data read from a file is represented as `MapPartitionsRDD` instead of `ParallelCollectionRDD` when we `.paralellize(...)` a collection.

Schema

RDDs are *schema-less* data structures (unlike DataFrames, which we will discuss in the next chapter). Thus, parallelizing a dataset, such as in the following code snippet, is perfectly fine with Spark when using RDDs:

```
data_heterogenous = sc.parallelize([
    ('Ferrari', 'fast'),
    {'Porsche': 100000},
    ['Spain','visited', 4504]
]).collect()
```

So, we can mix almost anything: a `tuple`, a `dict`, or a `list` and Spark will not complain.

Once you `.collect()` the dataset (that is, run an action to bring it back to the driver) you can access the data in the object as you would normally do in Python:

```
data_heterogenous[1]['Porsche']
```

It will produce the following:

```
100000
```

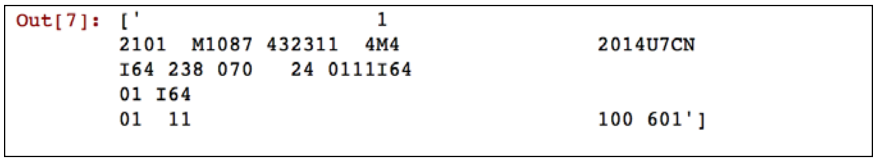
The `.collect()` method returns all the elements of the RDD to the driver where it is serialized as a list.

We will talk more about the caveats of using `.collect()` later in this chapter.

Reading from files

When you read from a text file, each row from the file forms an element of an RDD.

The `data_from_file.take(1)` command will produce the following (somewhat unreadable) output:



```
Out[7]: ['                    1
        2101  M1087 432311  4M4                2014U7CN
        I64 238 070   24 0111I64
        01 I64
        01  11                                 100 601']
```

To make it more readable, let's create a list of elements so each line is represented as a list of values.

Lambda expressions

In this example, we will extract the useful information from the cryptic looking record of `data_from_file`.

> Please refer to our GitHub repository for this book for the details of this method. Here, due to space constraints, we will only present an abbreviated version of the full method, especially where we create the Regex pattern. The code can be found here: `https://github.com/drabastomek/learningPySpark/tree/master/Chapter03/LearningPySpark_Chapter03.ipynb`.

First, let's define the method with the help of the following code, which will parse the unreadable row into something that we can use:

```
def extractInformation(row):
    import re
    import numpy as np
    selected_indices = [
         2,4,5,6,7,9,10,11,12,13,14,15,16,17,18,
         ...
         77,78,79,81,82,83,84,85,87,89
    ]
    record_split = re\
        .compile(
            r'([\s]{19})([0-9]{1})([\s]{40})
            ...
            ([\s]{33})([0-9\s]{3})([0-9\s]{1})([0-9\s]{1})')
    try:
        rs = np.array(record_split.split(row))[selected_indices]
    except:
        rs = np.array(['-99'] * len(selected_indices))
    return rs
```

A word of caution here is necessary. Defining pure Python methods can slow down your application as Spark needs to continuously switch back and forth between the Python interpreter and JVM. Whenever you can, you should use built-in Spark functions.

Next, we import the necessary modules: The `re` module as we will use regular expressions to parse the record, and `NumPy` for ease of selecting multiple elements at once.

Finally, we create a `Regex` object to extract the information as specified and parse the row through it.

We will not be delving into details here describing Regular Expressions. A good compendium on the topic can be found here `https://www.packtpub.com/application-development/mastering-python-regular-expressions`.

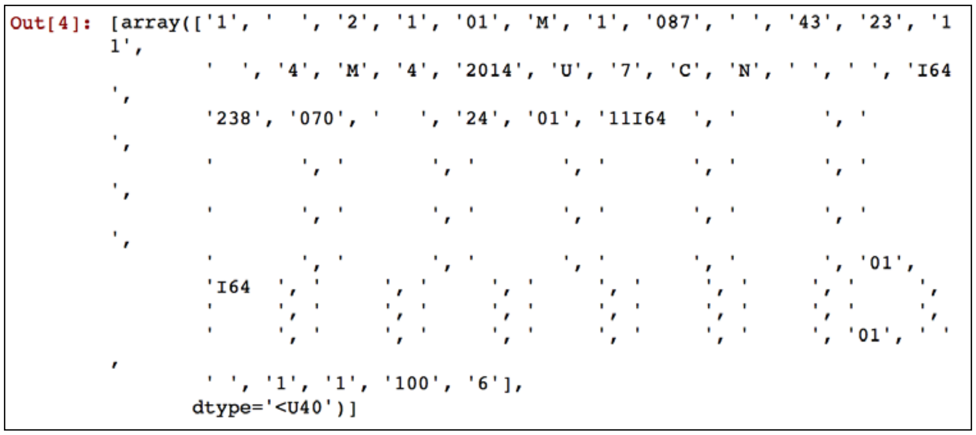
Once the record is parsed, we try to convert the list into a `NumPy` array and return it; if this fails we return a list of default values `-99` so we know this record did not parse properly.

We could implicitly filter out the malformed records by using `.flatMap(...)` and return an empty list `[]` instead of `-99` values. Check this for details: `http://stackoverflow.com/questions/34090624/remove-elements-from-spark-rdd`

Now, we will use the `extractInformation(...)` method to split and convert our dataset. Note that we pass only the method signature to `.map(...)`: the method will *hand over* one element of the RDD to the `extractInformation(...)` method at a time in each partition:

```
data_from_file_conv = data_from_file.map(extractInformation)
```

Running `data_from_file_conv.take(1)` will produce the following result (abbreviated):



```
Out[4]: [array(['1', '  ', '2', '1', '01', 'M', '1', '087', ' ', '43', '23', '1
        1',
               '  ', '4', 'M', '4', '2014', 'U', '7', 'C', 'N', ' ', ' ', 'I64
        ',
               '238', '070', '   ', '24', '01', '11I64  ', '        ', '
        ',
               '        ', '        ', '        ', '        ', '        ', '
        ',
               '        ', '        ', '        ', '        ', '        ', '
        ',
               '        ', '        ', '        ', '        ', '        ', '01',
               'I64   ', '       ', '       ', '       ', '       ', '       ',
               '      ', '       ', '       ', '       ', '       ', '       ',
               '      ', '       ', '       ', '       ', '       ', '01', ' '
        ,
               ' ', '1', '1', '100', '6'],
              dtype='<U40')]
```

Global versus local scope

One of the things that you, as a prospective PySpark user, need to get used to is the inherent parallelism of Spark. Even if you are proficient in Python, executing scripts in PySpark requires shifting your thinking a bit.

Spark can be run in two modes: Local and cluster. When you run Spark locally your code might not differ to what you are currently used to with running Python: Changes would most likely be more syntactic than anything else but with an added twist that data and code can be copied between separate worker processes.

However, taking the same code and deploying it to a cluster might cause a lot of head-scratching if you are not careful. This requires understanding how Spark executes a job on the cluster.

In the cluster mode, when a job is submitted for execution, the job is sent to the driver (or a master) node. The driver node creates a DAG (see *Chapter 1, Understanding Spark*) for a job and decides which executor (or worker) nodes will run specific tasks.

The driver then instructs the workers to execute their tasks and return the results to the driver when done. Before that happens, however, the driver prepares each task's closure: A set of variables and methods present on the driver for the worker to execute its task on the RDD.

This set of variables and methods is inherently *static* within the executors' context, that is, each executor gets a *copy* of the variables and methods from the driver. If, when running the task, the executor alters these variables or overwrites the methods, it does so **without** affecting either other executors' copies or the variables and methods of the driver. This might lead to some unexpected behavior and runtime bugs that can sometimes be really hard to track down.

Check out this discussion in PySpark's documentation for a more hands-on example: `http://spark.apache.org/docs/latest/programming-guide.html#local-vs-cluster-modes`.

Transformations

Transformations shape your dataset. These include mapping, filtering, joining, and transcoding the values in your dataset. In this section, we will showcase some of the transformations available on RDDs.

Due to space constraints we include only the most often used transformations and actions here. For a full set of methods available we suggest you check PySpark's documentation on RDDs `http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD`.

Since RDDs are schema-less, in this section we assume you know the schema of the produced dataset. If you cannot remember the positions of information in the parsed dataset we suggest you refer to the definition of the `extractInformation(...)` method on GitHub, code for `Chapter 03`.

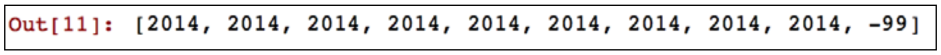
The .map(...) transformation

It can be argued that you will use the `.map(...)` transformation most often. The method is applied to each element of the RDD: In the case of the `data_from_file_conv` dataset, you can think of this as a transformation of each row.

In this example, we will create a new dataset that will convert year of death into a numeric value:

```
data_2014 = data_from_file_conv.map(lambda row: int(row[16]))
```

Running `data_2014.take(10)` will yield the following result:



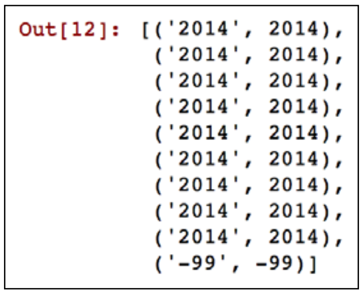
```
Out[11]: [2014, 2014, 2014, 2014, 2014, 2014, 2014, 2014, 2014, -99]
```

> If you are not familiar with `lambda` expressions, please refer to this resource: `https://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/`.

You can of course bring more columns over, but you would have to package them into a `tuple`, `dict`, or a `list`. Let's also include the 17th element of the row along so that we can confirm our `.map(...)` works as intended:

```
data_2014_2 = data_from_file_conv.map(
    lambda row: (row[16], int(row[16]):)
data_2014_2.take(5)
```

The preceding code will produce the following result:



```
Out[12]: [('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('-99', -99)]
```

The .filter(...) transformation

Another most often used transformation is the `.filter(...)` method, which allows you to select elements from your dataset that fit specified criteria. As an example, from the `data_from_file_conv` dataset, let's count how many people died in an accident in 2014:

```
data_filtered = data_from_file_conv.filter(
    lambda row: row[16] == '2014' and row[21] == '0')
data_filtered.count()
```

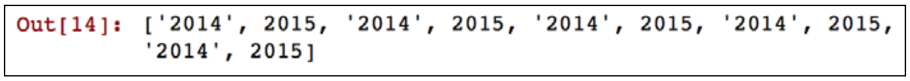
Note that the preceding command might take a while depending on how fast your computer is. For us, it took a little over two minutes to return a result.

The .flatMap(...) transformation

The `.flatMap(...)` method works similarly to `.map(...)`, but it returns a flattened result instead of a list. If we execute the following code:

```
data_2014_flat = data_from_file_conv.flatMap(lambda row: (row[16],
int(row[16]) + 1))
data_2014_flat.take(10)
```

It will yield the following output:



```
Out[14]: ['2014', 2015, '2014', 2015, '2014', 2015, '2014', 2015,
          '2014', 2015]
```

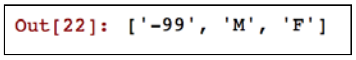
You can compare this result with the results of the command that generated `data_2014_2` previously. Note, also, as mentioned earlier, that the `.flatMap(...)` method can be used to filter out some malformed records when you need to parse your input. Under the hood, the `.flatMap(...)` method treats each row as a list and then simply *adds* all the records together; by passing an empty list the malformed records is dropped.

The .distinct(...) transformation

This method returns a list of distinct values in a specified column. It is extremely useful if you want to get to know your dataset or validate it. Let's check if the `gender` column contains only males and females; that would verify that we parsed the dataset properly. Let's run the following code:

```
distinct_gender = data_from_file_conv.map(
    lambda row: row[5]).distinct()
distinct_gender.collect()
```

This code will produce the following output:



```
Out[22]: ['-99', 'M', 'F']
```

First, we extract only the column that contains the gender. Next, we use the `.distinct()` method to select only the distinct values in the list. Lastly, we use the `.collect()` method to return the print of the values on the screen.

> Note that this is an expensive method and should be used sparingly and only when necessary as it shuffles the data around.

The .sample(...) transformation

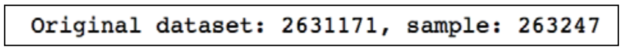
The `.sample(...)` method returns a randomized sample from the dataset. The first parameter specifies whether the sampling should be with a replacement, the second parameter defines the fraction of the data to return, and the third is seed to the pseudo-random numbers generator:

```
fraction = 0.1
data_sample = data_from_file_conv.sample(False, fraction, 666)
```

In this example, we selected a randomized sample of 10% from the original dataset. To confirm this, let's print the sizes of the datasets:

```
print('Original dataset: {0}, sample: {1}'\
.format(data_from_file_conv.count(), data_sample.count()))
```

The preceding command produces the following output:



```
Original dataset: 2631171, sample: 263247
```

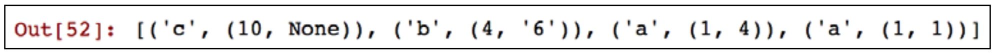
We use the `.count()` action that counts all the records in the corresponding RDDs.

The .leftOuterJoin(...) transformation

`.leftOuterJoin(...)`, just like in the SQL world, joins two RDDs based on the values found in both datasets, and returns records from the left RDD with records from the right one appended in places where the two RDDs match:

```
rdd1 = sc.parallelize([('a', 1), ('b', 4), ('c',10)])
rdd2 = sc.parallelize([('a', 4), ('a', 1), ('b', '6'), ('d', 15)])
rdd3 = rdd1.leftOuterJoin(rdd2)
```

Running `.collect(...)` on the `rdd3` will produce the following:



```
Out[52]: [('c', (10, None)), ('b', (4, '6')), ('a', (1, 4)), ('a', (1, 1))]
```

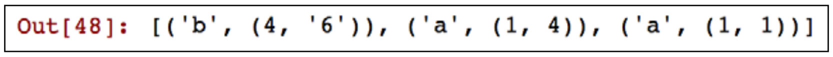
This is another expensive method and should be used sparingly and only when necessary as it shuffles the data around causing a performance hit.

What you can see here are all the elements from RDD `rdd1` and their corresponding values from RDD `rdd2`. As you can see, the value `'a'` shows up two times in `rdd3` and `'a'` appears twice in the RDD `rdd2`. The value `b` from the `rdd1` shows up only once and is joined with the value `'6'` from the `rdd2`. There are two things *missing*: Value `'c'` from `rdd1` does not have a corresponding key in the `rdd2` so the value in the returned tuple shows as `None`, and, since we were performing a left outer join, the value `'d'` from the `rdd2` disappeared as expected.

If we used the `.join(...)` method instead we would have got only the values for `'a'` and `'b'` as these two values intersect between these two RDDs. Run the following code:

```
rdd4 = rdd1.join(rdd2)
rdd4.collect()
```

It will result in the following output:

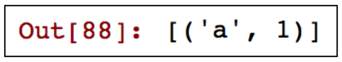


```
Out[48]: [('b', (4, '6')), ('a', (1, 4)), ('a', (1, 1))]
```

Another useful method is `.intersection(...)`, which returns the records that are equal in both RDDs. Execute the following code:

```
rdd5 = rdd1.intersection(rdd2)
rdd5.collect()
```

The output is as follows:



```
Out[88]: [('a', 1)]
```

The .repartition(...) transformation

Repartitioning the dataset changes the number of partitions that the dataset is divided into. This functionality should be used sparingly and only when really necessary as it shuffles the data around, which in effect results in a significant hit in terms of performance:

```
rdd1 = rdd1.repartition(4)
len(rdd1.glom().collect())
```

The preceding code prints out `4` as the new number of partitions.

The `.glom()` method, in contrast to `.collect()`, produces a list where each element is another list of all elements of the dataset present in a specified partition; the main list returned has as many elements as the number of partitions.

Actions

Actions, in contrast to transformations, execute the scheduled task on the dataset; once you have finished transforming your data you can execute your transformations. This might contain no transformations (for example, `.take(n)` will just return `n` records from an RDD even if you did not do any transformations to it) or execute the whole chain of transformations.

The .take(...) method

This is most arguably the most useful (and used, such as the `.map(...)` method). The method is preferred to `.collect(...)` as it only returns the `n` top rows from a single data partition in contrast to `.collect(...)`, which returns the whole RDD. This is especially important when you deal with large datasets:

```
data_first = data_from_file_conv.take(1)
```

If you want somewhat randomized records you can use `.takeSample(...)` instead, which takes three arguments: First whether the sampling should be with replacement, the second specifies the number of records to return, and the third is a seed to the pseudo-random numbers generator:

```
data_take_sampled = data_from_file_conv.takeSample(False, 1, 667)
```

The .collect(...) method

This method returns all the elements of the RDD to the driver. As we have just provided a caution about it, we will not repeat ourselves here.

The .reduce(...) method

The `.reduce(...)` method reduces the elements of an RDD using a specified method.

You can use it to sum the elements of your RDD:

```
rdd1.map(lambda row: row[1]).reduce(lambda x, y: x + y)
```

This will produce the sum of `15`.

We first create a list of all the values of the `rdd1` using the `.map(...)` transformation, and then use the `.reduce(...)` method to process the results. The `reduce(...)` method, on each partition, runs the summation method (here expressed as a `lambda`) and returns the sum to the driver node where the final aggregation takes place.

A word of caution is necessary here. The functions passed as a reducer need to be **associative**, that is, when the order of elements is changed the result does not, and **commutative**, that is, changing the order of operands does not change the result either.

The example of the associativity rule is *(5 + 2) + 3 = 5 + (2 + 3)*, and of the commutative is *5 + 2 + 3 = 3 + 2 + 5*. Thus, you need to be careful about what functions you pass to the reducer.

If you ignore the preceding rule, you might run into trouble (assuming your code runs at all). For example, let's assume we have the following RDD (with one partition only!):

```
data_reduce = sc.parallelize([1, 2, .5, .1, 5, .2], 1)
```

If we were to reduce the data in a manner that we would like to divide the current result by the subsequent one, we would expect a value of `10`:

```
works = data_reduce.reduce(lambda x, y: x / y)
```

However, if you were to partition the data into three partitions, the result will be wrong:

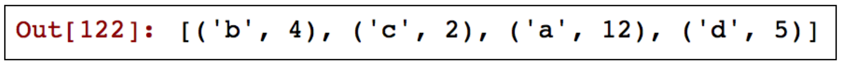
```
data_reduce = sc.parallelize([1, 2, .5, .1, 5, .2], 3)

data_reduce.reduce(lambda x, y: x / y)
```

It will produce `0.004`.

The `.reduceByKey(...)` method works in a similar way to the `.reduce(...)` method, but it performs a reduction on a key-by-key basis:

```
data_key = sc.parallelize(
    [('a', 4),('b', 3),('c', 2),('a', 8),('d', 2),('b', 1),
     ('d', 3)],4)
data_key.reduceByKey(lambda x, y: x + y).collect()
```

The preceding code produces the following:



```
Out[122]: [('b', 4), ('c', 2), ('a', 12), ('d', 5)]
```

The .count(...) method

The `.count(...)` method counts the number of elements in the RDD. Use the following code:

```
data_reduce.count()
```

This code will produce `6`, the exact number of elements in the `data_reduce` RDD.

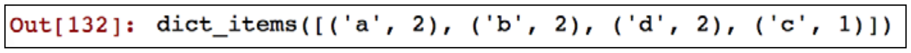
The `.count(...)` method produces the same result as the following method, but it does not require moving the whole dataset to the driver:

```
len(data_reduce.collect()) # WRONG -- DON'T DO THIS!
```

If your dataset is in a key-value form, you can use the `.countByKey()` method to get the counts of distinct keys. Run the following code:

```
data_key.countByKey().items()
```

This code will produce the following output:



```
Out[132]: dict_items([('a', 2), ('b', 2), ('d', 2), ('c', 1)])
```

The .saveAsTextFile(...) method

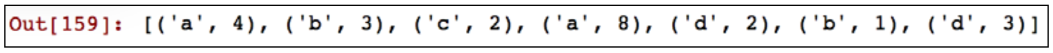
As the name suggests, the `.saveAsTextFile(...)` the RDD and saves it to text files: Each partition to a separate file:

```
data_key.saveAsTextFile(
'/Users/drabast/Documents/PySpark_Data/data_key.txt')
```

To read it back, you need to parse it back as all the rows are treated as strings:

```
def parseInput(row):
    import re
    pattern = re.compile(r'\(\'([a-z])\', ([0-9])\)')
    row_split = pattern.split(row)
    return (row_split[1], int(row_split[2]))

data_key_reread = sc \
    .textFile(
        '/Users/drabast/Documents/PySpark_Data/data_key.txt') \
    .map(parseInput)
data_key_reread.collect()
```

The list of keys read matches what we had initially:



```
Out[159]: [('a', 4), ('b', 3), ('c', 2), ('a', 8), ('d', 2), ('b', 1), ('d', 3)]
```

The .foreach(...) method

This is a method that applies the same function to each element of the RDD in an iterative way; in contrast to `.map(..)`, the `.foreach(...)` method applies a defined function to each record in a one-by-one fashion. It is useful when you want to save the data to a database that is not natively supported by PySpark.

Here, we'll use it to print (to CLI - not the Jupyter Notebook) all the records that are stored in `data_key` RDD:

```
def f(x):
    print(x)

data_key.foreach(f)
```

If you now navigate to CLI you should see all the records printed out. Note, that every time the order will most likely be different.

Summary

RDDs are the backbone of Spark; these schema-less data structures are the most fundamental data structures that we will deal with within Spark.

In this chapter, we presented ways to create RDDs from text files, by means of the `.parallelize(...)` method as well as by reading data from text files. Also, some ways of processing unstructured data were shown.

Transformations in Spark are lazy - they are only applied when an action is called. In this chapter, we discussed and presented the most commonly used transformations and actions; the PySpark documentation contains many more `http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD`.

One major distinction between Scala and Python RDDs is speed: Python RDDs can be much slower than their Scala counterparts.

In the next chapter we will walk you through a data structure that made PySpark applications perform *on par* with those written in Scala - the DataFrames.

3
DataFrames

A DataFrame is an immutable distributed collection of data that is organized into named columns analogous to a table in a relational database. Introduced as an experimental feature within Apache Spark 1.0 as `SchemaRDD`, they were renamed to `DataFrames` as part of the Apache Spark 1.3 release. For readers who are familiar with Python Pandas `DataFrame` or R `DataFrame`, a Spark DataFrame is a similar concept in that it allows users to easily work with structured data (for example, data tables); there are some differences as well so please temper your expectations.

By imposing a structure onto a distributed collection of data, this allows Spark users to query structured data in Spark SQL or using expression methods (instead of lambdas). In this chapter, we will include code samples using both methods. By structuring your data, this allows the Apache Spark engine – specifically, the Catalyst Optimizer – to significantly improve the performance of Spark queries. In earlier APIs of Spark (that is, RDDs), executing queries in Python could be significantly slower due to communication overhead between the Java JVM and Py4J.

If you are familiar with working with DataFrames in previous versions of Spark (that is Spark 1.x), you will notice that in Spark 2.0 we are using SparkSession instead of `SQLContext`. The various Spark contexts: `HiveContext`, `SQLContext`, `StreamingContext`, and `SparkContext` have merged together in SparkSession. This way you will be working with this session only as an entry point for reading data, working with metadata, configuration, and cluster resource management.

For more information, please refer to *How to use SparkSession in Apache Spark 2.0*(`http://bit.ly/2br0Fr1`).

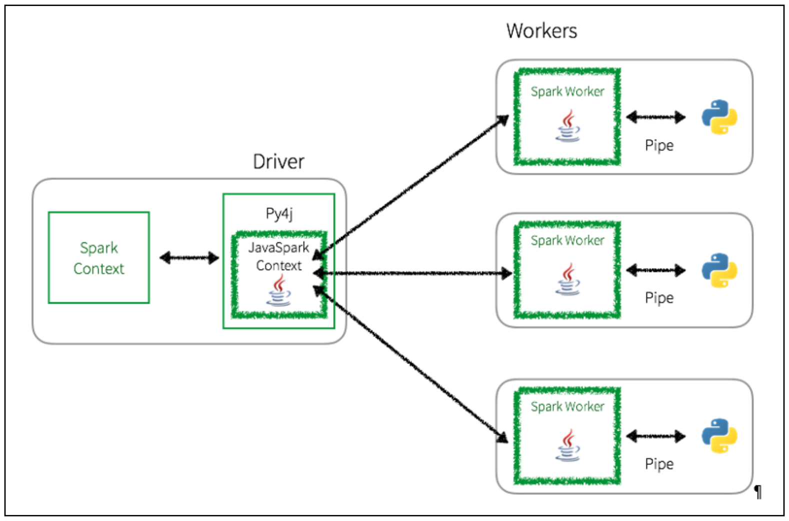
In this chapter, you will learn about the following:

- Python to RDD communications
- A quick refresh of Spark's Catalyst Optimizer
- Speeding up PySpark with DataFrames
- Creating DataFrames
- Simple DataFrame queries
- Interoperating with RDDs
- Querying with the DataFrame API
- Querying with Spark SQL
- Using DataFrames for an on-time flight performance

Python to RDD communications

Whenever a PySpark program is executed using RDDs, there is a potentially large overhead to execute the job. As noted in the following diagram, in the PySpark driver, the `Spark Context` uses `Py4j` to launch a JVM using the `JavaSparkContext`. Any RDD transformations are initially mapped to `PythonRDD` objects in Java.

Once these tasks are pushed out to the Spark Worker(s), `PythonRDD` objects launch Python `subprocesses` using pipes to send *both code and data* to be processed within Python:

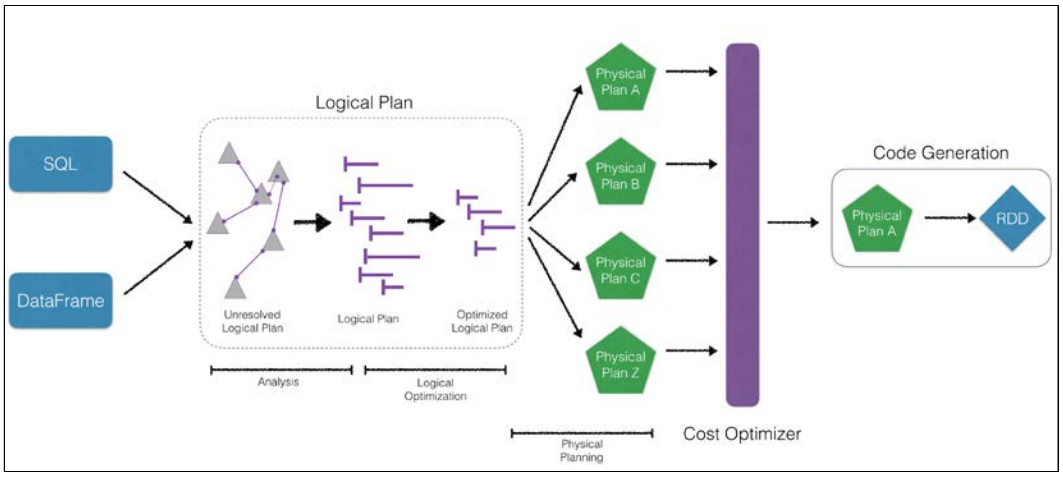


While this approach allows PySpark to distribute the processing of the data to multiple Python subprocesses on multiple workers, as you can see, there is a lot of context switching and communications overhead between Python and the JVM.

An excellent resource on PySpark performance is Holden Karau's *Improving PySpark Performance: Spark performance beyond the JVM*: `http://bit.ly/2bx89bn`.

Catalyst Optimizer refresh

As noted in *Chapter 1*, *Understanding Spark*, one of the primary reasons the Spark SQL engine is so fast is because of the **Catalyst Optimizer**. For readers with a database background, this diagram looks similar to the logical/physical planner and cost model/cost-based optimization of a **relational database management system** (**RDBMS**):



The significance of this is that, as opposed to immediately processing the query, the Spark engine's Catalyst Optimizer compiles and optimizes a logical plan and has a cost optimizer that determines the most efficient physical plan generated.

As noted in earlier chapters, while the Spark SQL Engine has both rules-based and cost-based optimizations that include (but are not limited to) predicate push down and column pruning. Targeted for the Apache Spark 2.2 release, the jira item *[SPARK-16026] Cost-based Optimizer Framework* at `https://issues.apache.org/jira/browse/SPARK-16026` is an umbrella ticket to implement a cost-based optimizer framework beyond broadcast join selection. For more information, please refer to the *Design Specification of Spark Cost-Based Optimization* at `http://bit.ly/2li1t4T`.

As part of **Project Tungsten**, there are further improvements to performance by generating byte code (code generation or `codegen`) instead of interpreting each row of data. Find more details on Tungsten in the *Project Tungsten* section in *Chapter 1, Understanding Spark*.

As previously noted, the optimizer is based on functional programming constructs and was designed with two purposes in mind: to ease the adding of new optimization techniques and features to Spark SQL, and to allow external developers to extend the optimizer (for example, adding data-source-specific rules, support for new data types, and so on).

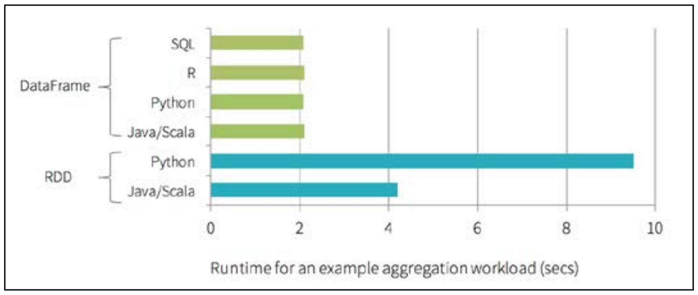
For more information, please refer to Michael Armbrust's excellent presentation, *Structuring Spark: SQL DataFrames, Datasets, and Streaming* at `http://bit.ly/2cJ508x`.

For further understanding of the *Catalyst Optimizer*, please refer to *Deep Dive into Spark SQL's Catalyst Optimizer* at `http://bit.ly/2bDVB1T`.

Also, for more information on *Project Tungsten*, please refer to *Project Tungsten: Bringing Apache Spark Closer to Bare Metal* at `http://bit.ly/2bQIlKY`, and *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop* at `http://bit.ly/2bDWtnc`.

Speeding up PySpark with DataFrames

The significance of DataFrames and the *Catalyst Optimizer* (and *Project Tungsten*) is the increase in performance of PySpark queries when compared to non-optimized RDD queries. As shown in the following figure, prior to the introduction of DataFrames, Python query speeds were often twice as slow as the same Scala queries using RDD. Typically, this slowdown in query performance was due to the communications overhead between Python and the JVM:



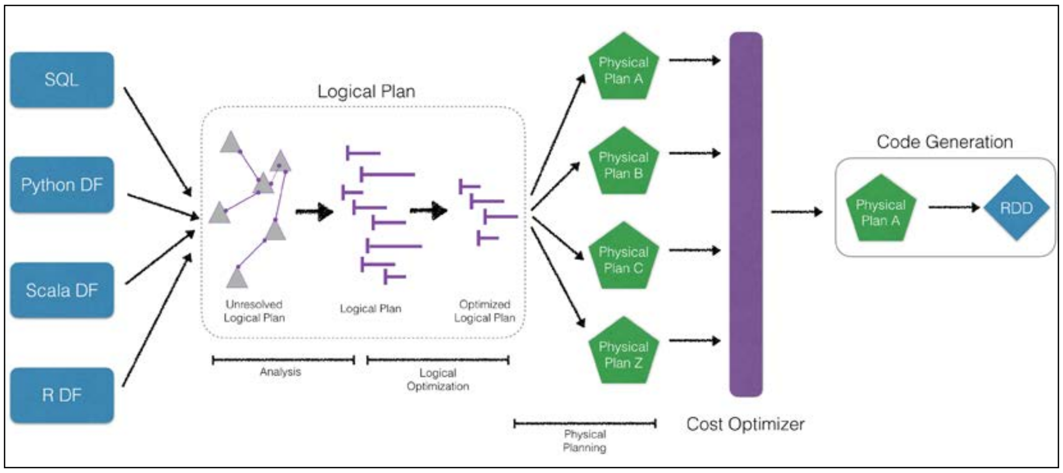
Source: *Introducing DataFrames in Apache-spark for Large Scale Data Science* at `http://bit.ly/2blDBI1`

With DataFrames, not only was there a significant improvement in Python performance, there is now performance parity between Python, Scala, SQL, and R.

> It is important to note that while, with DataFrames, PySpark is often significantly faster, there are some exceptions. The most prominent one is the use of Python UDFs, which results in round-trip communication between Python and the JVM. Note, this would be the worst-case scenario which would be similar if the compute was done on RDDs.

Python can take advantage of the performance optimizations in Spark even while the codebase for the Catalyst Optimizer is written in Scala. Basically, it is a Python wrapper of approximately 2,000 lines of code that allows PySpark DataFrame queries to be significantly faster.

Altogether, Python DataFrames (as well as SQL, Scala DataFrames, and R DataFrames) are all able to make use of the Catalyst Optimizer (as per the following updated diagram):



For more information, please refer to the blog post *Introducing DataFrames in Apache Spark for Large Scale Data Science* at `http://bit.ly/2blDBI1`, as well as Reynold Xin's Spark Summit 2015 presentation, *From DataFrames to Tungsten: A Peek into Spark's Future* at `http://bit.ly/2bQN92T`.

Creating DataFrames

Typically, you will create DataFrames by importing data using SparkSession (or calling `spark` in the PySpark shell).

In Spark 1.x versions, you typically had to use `sqlContext`.

In future chapters, we will discuss how to import data into your local file system, **Hadoop Distributed File System** (**HDFS**), or other cloud storage systems (for example, S3 or WASB). For this chapter, we will focus on generating your own DataFrame data directly within Spark or utilizing the data sources already available within Databricks Community Edition.

For instructions on how to sign up for the Community Edition of Databricks, see the bonus chapter, *Free Spark Cloud Offering*.

First, instead of accessing the file system, we will create a DataFrame by generating the data. In this case, we'll first create the `stringJSONRDD` RDD and then convert it into a DataFrame. This code snippet creates an RDD comprised of swimmers (their ID, name, age, and eye color) in JSON format.

Generating our own JSON data

Below, we will generate initially generate the `stringJSONRDD` RDD:

```
stringJSONRDD = sc.parallelize(("""
  { "id": "123",
"name": "Katie",
"age": 19,
"eyeColor": "brown"
  }""",
"""{
"id": "234",
"name": "Michael",
"age": 22,
"eyeColor": "green"
  }""",
"""{
"id": "345",
"name": "Simone",
"age": 23,
"eyeColor": "blue"
  }""")
)
```

Now that we have created the RDD, we will convert this into a DataFrame by using the SparkSession `read.json` method (that is, `spark.read.json(...)`). We will also create a temporary table by using the `.createOrReplaceTempView` method.

> In Spark 1.x, this method was `.registerTempTable`, which is being deprecated as part of Spark 2.x.

Creating a DataFrame

Here is the code to create a DataFrame:

```
swimmersJSON = spark.read.json(stringJSONRDD)
```

Creating a temporary table

Here is the code for creating a temporary table:

```
swimmersJSON.createOrReplaceTempView("swimmersJSON")
```

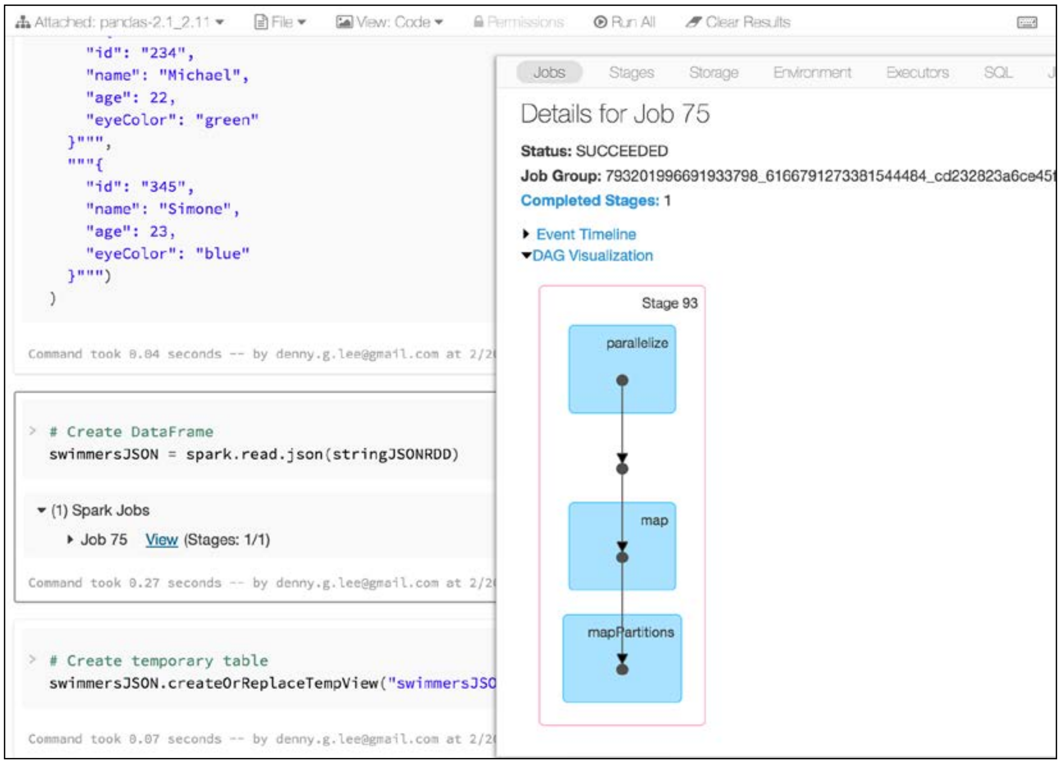
As noted in the previous chapters, many RDD operations are transformations, which are not executed until an action operation is executed. For example, in the preceding code snippet, the `sc.parallelize` is a transformation that is executed when converting from an RDD to a DataFrame by using `spark.read.json`. Notice that, in the screenshot of this code snippet notebook (near the bottom left), the Spark job is not executed until the second cell containing the `spark.read.json` operation.

> These are screenshots from Databricks Community Edition, but all the code samples and Spark UI screenshots can be executed/viewed in any flavor of Apache Spark 2.x.

To further emphasize the point, in the right pane of the following figure, we present the DAG graph of execution.

> A great resource to better understand the Spark UI DAG visualization is the blog post *Understanding Your Apache Spark Application Through Visualization* at `http://bit.ly/2cSemkv`.

In the following screenshot, you can see the Spark job' `sparallelize` operation is from the first cell generating the RDD `stringJSONRDD`, while the `map` and `mapPartitions` operations are the operations required to create the DataFrame:



```
    "id": "234",
    "name": "Michael",
    "age": 22,
    "eyeColor": "green"
  }""",
  """{
    "id": "345",
    "name": "Simone",
    "age": 23,
    "eyeColor": "blue"
  }""")
)

Command took 0.84 seconds -- by denny.g.lee@gmail.com at 2/2

# Create DataFrame
swimmersJSON = spark.read.json(stringJSONRDD)

(1) Spark Jobs
  Job 75  View (Stages: 1/1)

Command took 0.27 seconds -- by denny.g.lee@gmail.com at 2/2

# Create temporary table
swimmersJSON.createOrReplaceTempView("swimmersJSO

Command took 0.07 seconds -- by denny.g.lee@gmail.com at 2/2
```

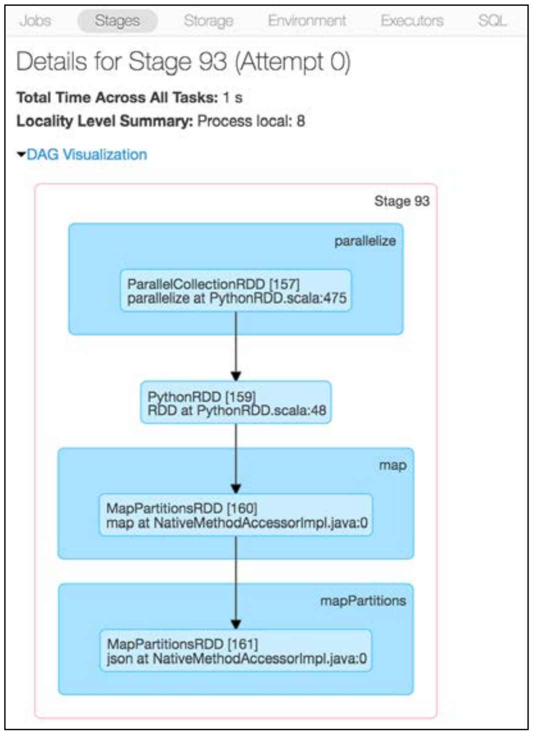
Details for Job 75

Status: SUCCEEDED
Job Group: 793201996691933798_6166791273381544484_cd232823a6ce45
Completed Stages: 1

Event Timeline
DAG Visualization

Spark UI of the DAG visualization of the spark.read.json(stringJSONRDD) job.

In the following screenshot, you can see the *stages* for the `parallelize` operation are from the first cell generating the RDD `stringJSONRDD`, while the `map` and `mapPartitions` operations are the operations required to create the DataFrame:


Jobs
Stages
Storage
Environment
Executors
SQL
Details for Stage 93 (Attempt 0)
Total Time Across All Tasks: 1 s
Locality Level Summary: Process local: 8
DAG Visualization
Stage 93
parallelize
ParallelCollectionRDD [157]
parallelize at PythonRDD.scala:475
PythonRDD [159]
RDD at PythonRDD.scala:48
map
MapPartitionsRDD [160]
map at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [161]
json at NativeMethodAccessorImpl.java:0

Spark UI of the DAG visualization of the stages within the spark.read.json(stringJSONRDD) job.

It is important to note that `parallelize`, `map`, and `mapPartitions` are all RDD *transformations*. Wrapped within the DataFrame operation, `spark.read.json` (in this case), are not only the RDD transformations, but also the *action* which converts the RDD into a DataFrame. This is an important call out, because even though you are executing DataFrame *operations*, to debug your operations you will need to remember that you will be making sense of *RDD operations* within the Spark UI.

Note that creating the temporary table is a DataFrame transformation and not executed until a DataFrame action is executed (for example, in the SQL query to be executed in the following section).

> DataFrame transformations and actions are similar to RDD transformations and actions in that there is a set of operations that are lazy (transformations). But, in comparison to RDDs, DataFrames operations are not as lazy, primarily due to the Catalyst Optimizer. For more information, please refer to Holden Karau and Rachel Warren's book *High Performance Spark*, `http://highperformancespark.com/`.

Simple DataFrame queries

Now that you have created the `swimmersJSON` DataFrame, we will be able to run the DataFrame API, as well as SQL queries against it. Let's start with a simple query showing all the rows within the DataFrame.

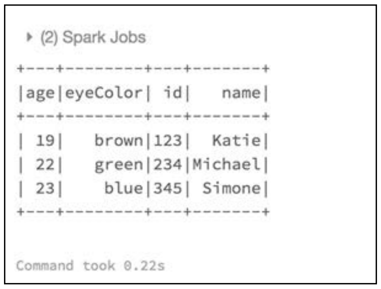
DataFrame API query

To do this using the DataFrame API, you can use the `show(<n>)` method, which prints the first `n` rows to the console:

Running the `.show()` method will default to present the first 10 rows.

```
# DataFrame API
swimmersJSON.show()
```

This gives the following output:



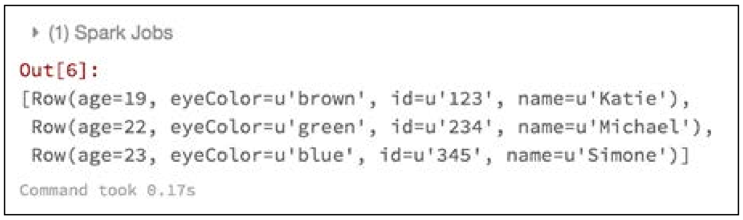
```
▸ (2) Spark Jobs
+---+--------+---+-------+
|age|eyeColor| id|   name|
+---+--------+---+-------+
| 19|   brown|123|  Katie|
| 22|   green|234|Michael|
| 23|    blue|345| Simone|
+---+--------+---+-------+

Command took 0.22s
```

SQL query

If you prefer writing SQL statements, you can write the following query:

```
spark.sql("select * from swimmersJSON").collect()
```

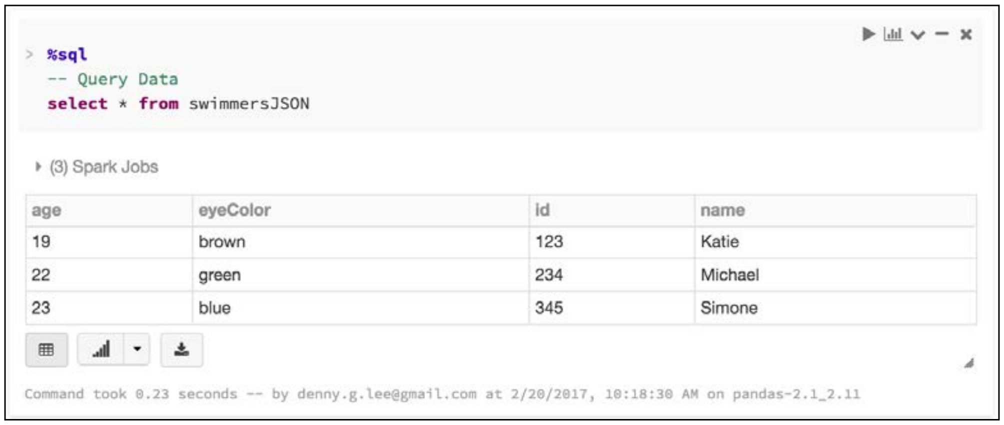
This will give the following output:



```
▸ (1) Spark Jobs
Out[6]:
[Row(age=19, eyeColor=u'brown', id=u'123', name=u'Katie'),
 Row(age=22, eyeColor=u'green', id=u'234', name=u'Michael'),
 Row(age=23, eyeColor=u'blue', id=u'345', name=u'Simone')]
Command took 0.17s
```

We are using the `.collect()` method, which returns all the records as a list of **Row** objects. Note that you can use either the `collect()` or `show()` method for both DataFrames and SQL queries. Just make sure that if you use `.collect()`, this is for a small DataFrame, since it will return all of the rows in the DataFrame and move them back from the executors to the driver. You can instead use `take(<n>)` or `show(<n>)`, which allow you to limit the number of rows returned by specifying `<n>`:

Note that, if you are using Databricks, you can use the `%sql` command and run your SQL statement directly within a notebook cell, as noted.



```
%sql
-- Query Data
select * from swimmersJSON
```

(3) Spark Jobs

age	eyeColor	id	name
19	brown	123	Katie
22	green	234	Michael
23	blue	345	Simone

```
Command took 0.23 seconds -- by denny.g.lee@gmail.com at 2/20/2017, 10:18:30 AM on pandas-2.1_2.11
```

Interoperating with RDDs

There are two different methods for converting existing RDDs to DataFrames (or Datasets[T]): inferring the schema using reflection, or programmatically specifying the schema. The former allows you to write more concise code (when your Spark application already knows the schema), while the latter allows you to construct DataFrames when the columns and their data types are only revealed at run time. Note, **reflection** is in reference to *schema reflection* as opposed to Python `reflection`.

Inferring the schema using reflection

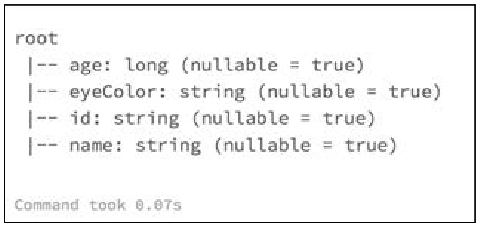
In the process of building the DataFrame and running the queries, we skipped over the fact that the schema for this DataFrame was automatically defined. Initially, row objects are constructed by passing a list of key/value pairs as `**kwargs` to the row class. Then, Spark SQL converts this RDD of row objects into a DataFrame, where the keys are the columns and the data types are inferred by sampling the data.

The `**kwargs` construct allows you to pass a variable number of parameters to a method at runtime.

Going back to the code, after initially creating the `swimmersJSON` DataFrame, without specifying the schema, you will notice the schema definition by using the `printSchema()` method:

```
# Print the schema
swimmersJSON.printSchema()
```

This gives the following output:



```
root
 |-- age: long (nullable = true)
 |-- eyeColor: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)

Command took 0.07s
```

But what if we want to specify the schema because, in this example, we know that the `id` is actually a `long` instead of a `string`?

Programmatically specifying the schema

In this case, let's programmatically specify the schema by bringing in Spark SQL data types (`pyspark.sql.types`) and generate some `.csv` data for this example:

```
# Import types
from pyspark.sql.types import *

# Generate comma delimited data
stringCSVRDD = sc.parallelize([
(123, 'Katie', 19, 'brown'),
(234, 'Michael', 22, 'green'),
(345, 'Simone', 23, 'blue')
])
```

First, we will encode the schema as a string, per the `[schema]` variable below. Then we will define the schema using `StructType` and `StructField`:

```
# Specify schema
schema = StructType([
StructField("id", LongType(), True),
StructField("name", StringType(), True),
StructField("age", LongType(), True),
StructField("eyeColor", StringType(), True)
])
```

Note, the `StructField` class is broken down in terms of:

- `name`: The name of this field
- `dataType`: The data type of this field
- `nullable`: Indicates whether values of this field can be null

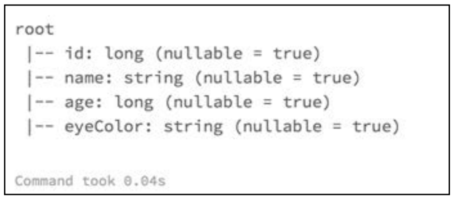
Finally, we will apply the schema (`schema`) we created to the `stringCSVRDD` RDD (that is, the generated `.csv` data) and create a temporary view so we can query it using SQL:

```
# Apply the schema to the RDD and Create DataFrame
swimmers = spark.createDataFrame(stringCSVRDD, schema)

# Creates a temporary view using the DataFrame
swimmers.createOrReplaceTempView("swimmers")
```

With this example, we have finer-grain control over the schema and can specify that `id` is a `long` (as opposed to a string in the previous section):

```
swimmers.printSchema()
```

This gives the following output:



```
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- eyeColor: string (nullable = true)

Command took 0.04s
```

In many cases, the schema can be inferred (as per the previous section) and you do not need to specify the schema, as in this preceding example.

Querying with the DataFrame API

As noted in the previous section, you can start off by using `collect()`, `show()`, or `take()` to view the data within your DataFrame (with the last two including the option to limit the number of returned rows).

Number of rows

To get the number of rows within your DataFrame, you can use the `count()` method:

```
swimmers.count()
```

This gives the following output:

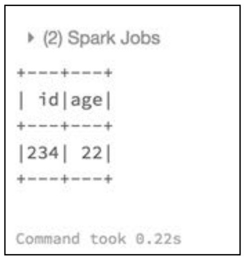
```
Out[13]: 3
```

Running filter statements

To run a filter statement, you can use the `filter` clause; in the following code snippet, we are using the `select` clause to specify the columns to be returned as well:

```
# Get the id, age where age = 22
swimmers.select("id", "age").filter("age = 22").show()

# Another way to write the above query is below
swimmers.select(swimmers.id, swimmers.age).filter(swimmers.age == 22).
show()
```

The output of this query is to choose only the `id` and `age` columns, where `age` = `22`:

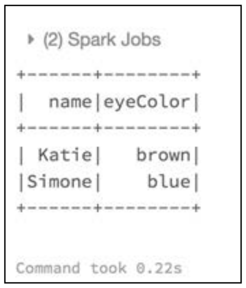


```
+---+---+
| id|age|
+---+---+
|234| 22|
+---+---+

Command took 0.22s
```

If we only want to get back the name of the swimmers who have an eye color that begins with the letter `b`, we can use a SQL-like syntax, `like`, as shown in the following code:

```
# Get the name, eyeColor where eyeColor like 'b%'
swimmers.select("name", "eyeColor").filter("eyeColor like 'b%'").
show()
```

The output is as follows:



```
▸ (2) Spark Jobs
+------+--------+
|  name|eyeColor|
+------+--------+
| Katie|   brown|
|Simone|    blue|
+------+--------+

Command took 0.22s
```

Querying with SQL

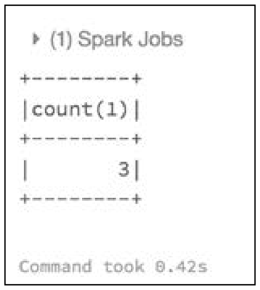
Let's run the same queries, except this time, we will do so using SQL queries against the same DataFrame. Recall that this DataFrame is accessible because we executed the `.createOrReplaceTempView` method for `swimmers`.

Number of rows

The following is the code snippet to get the number of rows within your DataFrame using SQL:

```
spark.sql("select count(1) from swimmers").show()
```

The output is as follows:



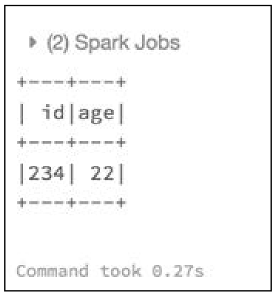
```
▸ (1) Spark Jobs
+--------+
|count(1)|
+--------+
|       3|
+--------+

Command took 0.42s
```

Running filter statements using the where Clauses

To run a filter statement using SQL, you can use the `where` clause, as noted in the following code snippet:

```
# Get the id, age where age = 22 in SQL
spark.sql("select id, age from swimmers where age = 22").show()
```

The output of this query is to choose only the `id` and `age` columns where `age` = `22`:



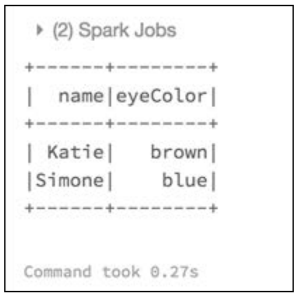
▸ (2) Spark Jobs

```
+---+---+
| id|age|
+---+---+
|234| 22|
+---+---+

Command took 0.27s
```

As with the DataFrame API querying, if we want to get back the name of the swimmers who have an eye color that begins with the letter `b` only, we can use the `like` syntax as well:

```
spark.sql(
"select name, eyeColor from swimmers where eyeColor like 'b%'").show()
```

The output is as follows:



▸ (2) Spark Jobs

```
+------+--------+
|  name|eyeColor|
+------+--------+
| Katie|   brown|
|Simone|    blue|
+------+--------+

Command took 0.27s
```

For more information, please refer to the *Spark SQL, DataFrames, and Datasets Guide* at `http://bit.ly/2cd1wyx`.

An important note when working with Spark SQL and DataFrames is that while it is easy to work with CSV, JSON, and a variety of data formats, the most common storage format for Spark SQL analytics queries is the *Parquet* file format. It is a columnar format that is supported by many other data processing systems and Spark SQL supports both reading and writing Parquet files that automatically preserves the schema of the original data. For more information, please refer to the latest *Spark SQL Programming Guide* > *Parquet Files* at: `http://spark.apache.org/docs/latest/sql-programming-guide.html#parquet-files`. Also, there are many performance optimizations that pertain to Parquet, including (but not limited to) *Automatic Partition Discovery and Schema Migration for Parquet* at `https://databricks.com/blog/2015/03/24/spark-sql-graduates-from-alpha-in-spark-1-3.html` and *How Apache Spark performs a fast count using the parquet metadata* at `https://github.com/dennyglee/databricks/blob/master/misc/parquet-count-metadata-explanation.md`.

DataFrame scenario – on-time flight performance

To showcase the types of queries you can do with DataFrames, let's look at the use case of on-time flight performance. We will analyze the *Airline On-Time Performance and Causes of Flight Delays: On-Time Data* (`http://bit.ly/2ccJPPM`), and join this with the airports dataset, obtained from the *Open Flights Airport, airline, and route data* (`http://bit.ly/2ccK5hw`), to better understand the variables associated with flight delays.

For this section, we will be using Databricks Community Edition (a free offering of the Databricks product), which you can get at `https://databricks.com/try-databricks`. We will be using visualizations and pre-loaded datasets within Databricks to make it easier for you to focus on writing the code and analyzing the results.

If you would prefer to run this on your own environment, you can find the datasets available in our GitHub repository for this book at `https://github.com/drabastomek/learningPySpark`.

Preparing the source datasets

We will first process the source airports and flight performance datasets by specifying their file path location and importing them using SparkSession:

```
# Set File Paths
flightPerfFilePath =
"/databricks-datasets/flights/departuredelays.csv"
airportsFilePath =
"/databricks-datasets/flights/airport-codes-na.txt"

# Obtain Airports dataset
airports = spark.read.csv(airportsFilePath, header='true',
inferSchema='true', sep='\t')
airports.createOrReplaceTempView("airports")

# Obtain Departure Delays dataset
flightPerf = spark.read.csv(flightPerfFilePath, header='true')
flightPerf.createOrReplaceTempView("FlightPerformance")

# Cache the Departure Delays dataset
flightPerf.cache()
```

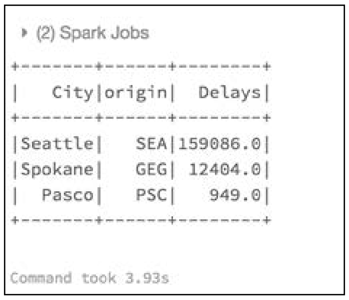
Note that we're importing the data using the CSV reader (`com.databricks.spark.csv`), which works for any specified delimiter (note that the airports data is tab-delimited, while the flight performance data is comma-delimited). Finally, we cache the flight dataset so subsequent queries will be faster.

Joining flight performance and airports

One of the more common tasks with DataFrames/SQL is to join two different datasets; it is often one of the more demanding operations (from a performance perspective). With DataFrames, a lot of the performance optimizations for these joins are included by default:

```
# Query Sum of Flight Delays by City and Origin Code
# (for Washington State)
spark.sql("""
select a.City,
f.origin,
sum(f.delay) as Delays
from FlightPerformance f
join airports a
on a.IATA = f.origin
where a.State = 'WA'
group by a.City, f.origin
order by sum(f.delay) desc"""
).show()
```

In our scenario, we are querying the total delays by city and origin code for the state of Washington. This will require joining the flight performance data with the airports data by **International Air Transport Association** (**IATA**) code. The output of the query is as follows:

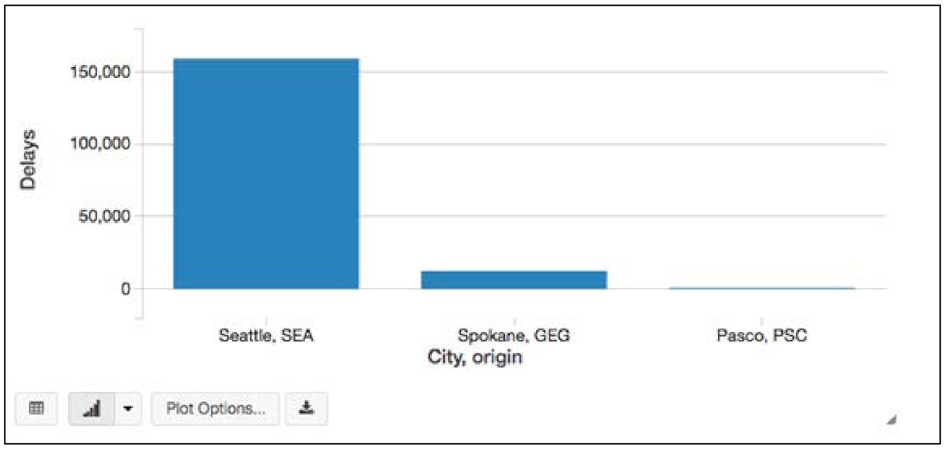


```
▸ (2) Spark Jobs
+-------+------+--------+
|   City|origin|  Delays|
+-------+------+--------+
|Seattle|   SEA|159086.0|
|Spokane|   GEG| 12404.0|
|  Pasco|   PSC|   949.0|
+-------+------+--------+

Command took 3.93s
```

Using notebooks (such as Databricks, iPython, Jupyter, and Apache Zeppelin), you can more easily execute and visualize your queries. In the following examples, we will be using the Databricks notebook. Within our Python notebook, we can use the `%sql` function to execute SQL statements within that notebook cell:

```
%sql
-- Query Sum of Flight Delays by City and Origin Code (for Washington
State)
select a.City, f.origin, sum(f.delay) as Delays
  from FlightPerformance f
    join airports a
      on a.IATA = f.origin
 where a.State = 'WA'
 group by a.City, f.origin
 order by sum(f.delay) desc
```

This is the same as the previous query, but due to formatting, easier to read. In our Databricks notebook example, we can quickly visualize this data into a bar chart:

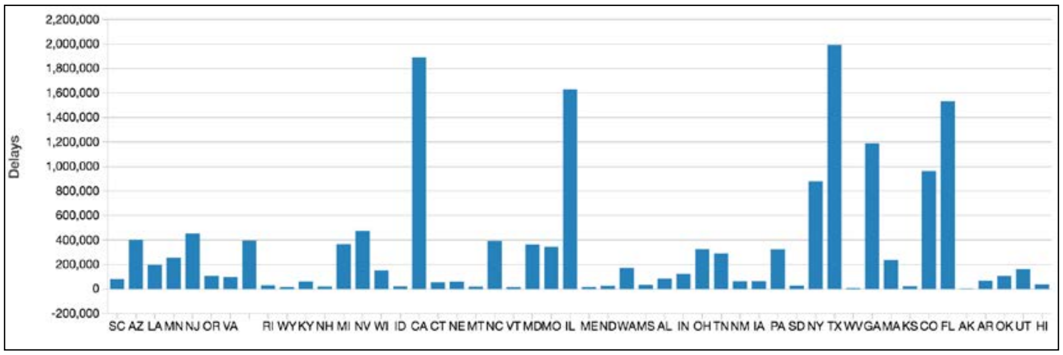


Visualizing our flight-performance data

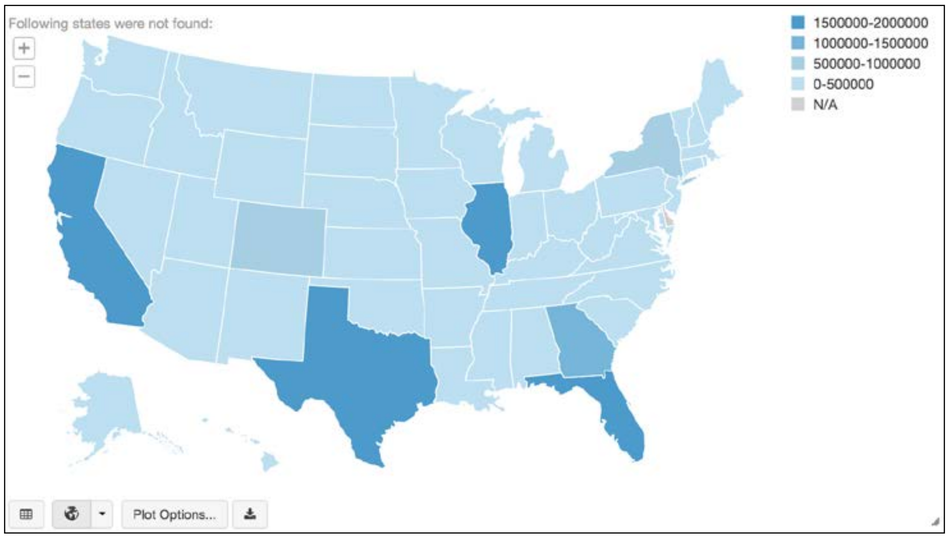
Let's continue visualizing our data, but broken down by all states in the continental US:

```
%sql
-- Query Sum of Flight Delays by State (for the US)
select a.State, sum(f.delay) as Delays
  from FlightPerformance f
    join airports a
      on a.IATA = f.origin
 where a.Country = 'USA'
 group by a.State
```

The output bar chart is as follows:



But, it would be cooler to view this data as a map; click on the bar chart icon at the bottom-left of the chart, and you can choose from many different native navigations, including a map:

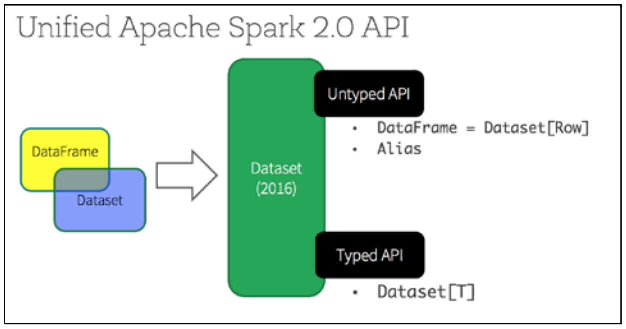


One of the key benefits of DataFrames is that the information is structured similar to a table. Therefore, whether you are using notebooks or your favorite BI tool, you will be able to quickly visualize your data.

> You can find the full list of `pyspark.sql.DataFrame` methods at `http://bit.ly/2bkUGnT`.
>
> You can find the full list of `pyspark.sql.functions` at `http://bit.ly/2bTAzLT`.

Spark Dataset API

After this discussion about Spark DataFrames, let's have a quick recap of the Spark Dataset API. Introduced in Apache Spark 1.6, the goal of Spark Datasets was to provide an API that allows users to easily express transformations on domain objects, while also providing the performance and benefits of the robust Spark SQL execution engine. As part of the Spark 2.0 release (and as noted in the diagram below), the DataFrame APIs is merged into the Dataset API thus unifying data processing capabilities across all libraries. Because of this unification, developers now have fewer concepts to learn or remember, and work with a single high-level and *type-safe* API – called Dataset:



Conceptually, the Spark DataFrame is an *alias* for a collection of generic objects Dataset[Row], where a Row is a generic *untyped* JVM object. Dataset, by contrast, is a collection of *strongly-typed* JVM objects, dictated by a case class you define, in Scala or Java. This last point is particularly important as this means that the Dataset API is *not supported* by PySpark due to the lack of benefit from the type enhancements. Note, for the parts of the Dataset API that are not available in PySpark, they can be accessed by converting to an RDD or by using UDFs. For more information, please refer to the jira [SPARK-13233]: Python Dataset at `http://bit.ly/2dbfoFT`.

Summary

With Spark DataFrames, Python developers can make use of a simpler abstraction layer that is also potentially significantly faster. One of the main reasons Python is initially slower within Spark is due to the communication layer between Python sub-processes and the JVM. For Python DataFrame users, we have a Python wrapper around Scala DataFrames that avoids the Python sub-process/JVM communication overhead. Spark DataFrames has many performance enhancements through the Catalyst Optimizer and Project Tungsten which we have reviewed in this chapter. In this chapter, we also reviewed how to work with Spark DataFrames and worked on an on-time flight performance scenario using DataFrames.

In this chapter, we created and worked with DataFrames by generating the data or making use of existing datasets.

In the next chapter, we will discuss how to transform and understand your own data.

4
Prepare Data for Modeling

All data is dirty, irrespective of what the source of the data might lead you to believe: it might be your colleague, a telemetry system that monitors your environment, a dataset you download from the web, or some other source. Until you have tested and proven to yourself that your data is in a clean state (we will get to what clean state means in a second), you should neither trust it nor use it for modeling.

Your data can be stained with duplicates, missing observations and outliers, non-existent addresses, wrong phone numbers and area codes, inaccurate geographical coordinates, wrong dates, incorrect labels, mixtures of upper and lower cases, trailing spaces, and many other more subtle problems. It is your job to clean it, irrespective of whether you are a data scientist or data engineer, so you can build a statistical or machine learning model.

Your dataset is considered technically clean if none of the aforementioned problems can be found. However, to clean the dataset for modeling purposes, you also need to check the distributions of your features and confirm they fit the predefined criteria.

As a data scientist, you can expect to spend 80-90% of your time *massaging* your data and getting familiar with all the features. This chapter will guide you through that process, leveraging Spark capabilities.

In this chapter, you will learn how to do the following:

- Recognize and handle duplicates, missing observations, and outliers
- Calculate descriptive statistics and correlations
- Visualize your data with matplotlib and Bokeh

Checking for duplicates, missing observations, and outliers

Until you have fully tested the data and proven it worthy of your time, you should neither trust it nor use it. In this section, we will show you how to deal with duplicates, missing observations, and outliers.

Duplicates

Duplicates are observations that appear as distinct rows in your dataset, but which, upon closer inspection, look the same. That is, if you looked at them side by side, all the features in these two (or more) rows would have exactly the same values.

On the other hand, if your data has some form of an ID to distinguish between records (or associate them with certain users, for example), then what might initially appear as a duplicate may not be; sometimes systems fail and produce erroneous IDs. In such a situation, you need to either check whether the same ID is a real duplicate, or you need to come up with a new ID system.

Consider the following example:

```
df = spark.createDataFrame([
        (1, 144.5, 5.9, 33, 'M'),
        (2, 167.2, 5.4, 45, 'M'),
        (3, 124.1, 5.2, 23, 'F'),
        (4, 144.5, 5.9, 33, 'M'),
        (5, 133.2, 5.7, 54, 'F'),
        (3, 124.1, 5.2, 23, 'F'),
        (5, 129.2, 5.3, 42, 'M'),
    ], ['id', 'weight', 'height', 'age', 'gender'])
```

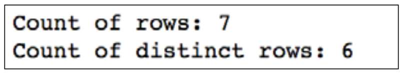
As you can see, we have several issues here:

- We have two rows with IDs equal to `3` and they are exactly the same
- Rows with IDs `1` and `4` are the same — the only thing that's different is their IDs, so we can safely assume that they are the same person
- We have two rows with IDs equal to `5`, but that seems to be a recording issue, as they do not seem to be the same person

This is a very easy dataset with only seven rows. What do you do when you have millions of observations? The first thing I normally do is to check if I have any duplicates: I compare the counts of the full dataset with the one that I get after running a `.distinct()` method:

```
print('Count of rows: {0}'.format(df.count()))
print('Count of distinct rows: {0}'.format(df.distinct().count()))
```

Here's what you get back for our DataFrame:

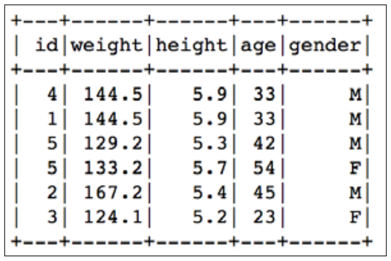


```
Count of rows: 7
Count of distinct rows: 6
```

If these two numbers differ, then you know you have, what I like to call, pure duplicates: rows that are exact copies of each other. We can drop these rows by using the `.dropDuplicates(...)` method:

```
df = df.dropDuplicates()
```

Our dataset will then look as follows (once you run `df.show()`):

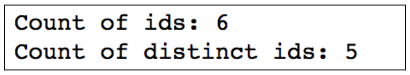


```
+---+------+------+---+------+
| id|weight|height|age|gender|
+---+------+------+---+------+
|  4| 144.5|   5.9| 33|     M|
|  1| 144.5|   5.9| 33|     M|
|  5| 129.2|   5.3| 42|     M|
|  5| 133.2|   5.7| 54|     F|
|  2| 167.2|   5.4| 45|     M|
|  3| 124.1|   5.2| 23|     F|
+---+------+------+---+------+
```

We dropped one of the rows with ID `3`. Now let's check whether there are any duplicates in the data irrespective of ID. We can quickly repeat what we have done earlier, but using only columns other than the ID column:

```
print('Count of ids: {0}'.format(df.count()))
print('Count of distinct ids: {0}'.format(
    df.select([
        c for c in df.columns if c != 'id'
    ]).distinct().count())
)
```

We should see one more row that is a duplicate:

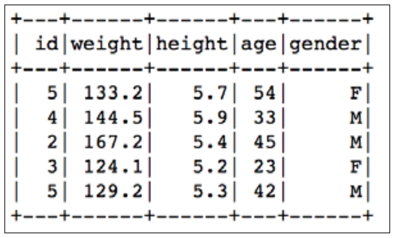


```
Count of ids: 6
Count of distinct ids: 5
```

We can still use the `.dropDuplicates(...)`, but will add the `subset` parameter that specifies only the columns other than the `id` column:

```
df = df.dropDuplicates(subset=[
    c for c in df.columns if c != 'id'
])
```

The `subset` parameter instructs the `.dropDuplicates(...)` method to look for duplicated rows using only the columns specified via the `subset` parameter; in the preceding example, we will drop the duplicated records with the same `weight`, `height`, `age`, and `gender` but not `id`. Running the `df.show()`, we get the following cleaner dataset as we dropped the row with `id = 1` since it was identical to the record with `id = 4`:

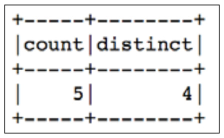


```
+---+------+------+---+------+
| id|weight|height|age|gender|
+---+------+------+---+------+
|  5| 133.2|   5.7| 54|     F|
|  4| 144.5|   5.9| 33|     M|
|  2| 167.2|   5.4| 45|     M|
|  3| 124.1|   5.2| 23|     F|
|  5| 129.2|   5.3| 42|     M|
+---+------+------+---+------+
```

Now that we know there are no full rows duplicated, or any identical rows differing only by ID, let's check if there are any duplicated IDs. To calculate the total and distinct number of IDs in one step, we can use the `.agg(...)` method:

```
import pyspark.sql.functions as fn

df.agg(
    fn.count('id').alias('count'),
    fn.countDistinct('id').alias('distinct')
).show()
```

Here's the output of the preceding code:



```
+-----+--------+
|count|distinct|
+-----+--------+
|    5|       4|
+-----+--------+
```

In the previous example, we first import all the functions from the `pyspark.sql` module.

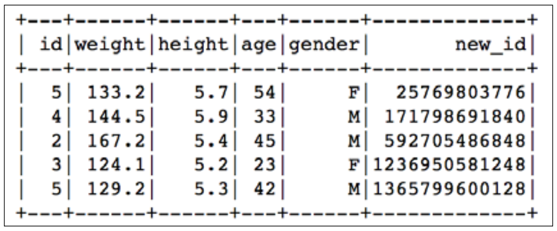
This gives us access to a vast array of various functions, too many to list here. However, we strongly encourage you to study the PySpark's documentation at `http://spark.apache.org/docs/2.0.0/api/python/pyspark.sql.html#module-pyspark.sql.functions`.

Next, we use the `.count(...)` and `.countDistinct(...)` to, respectively, calculate the number of rows and the number of distinct `ids` in our DataFrame. The `.alias(...)` method allows us to specify a friendly name to the returned column.

As you can see, we have five rows in total, but only four distinct IDs. Since we have already dropped all the duplicates, we can safely assume that this might just be a fluke in our ID data, so we will give each row a unique ID:

```
df.withColumn('new_id', fn.monotonically_increasing_id()).show()
```

The preceding code snippet produced the following output:



```
+---+------+------+---+------+-------------+
| id|weight|height|age|gender|       new_id|
+---+------+------+---+------+-------------+
|  5| 133.2|   5.7| 54|     F|  25769803776|
|  4| 144.5|   5.9| 33|     M| 171798691840|
|  2| 167.2|   5.4| 45|     M| 592705486848|
|  3| 124.1|   5.2| 23|     F|1236950581248|
|  5| 129.2|   5.3| 42|     M|1365799600128|
+---+------+------+---+------+-------------+
```

The `.monotonicallymonotonically_increasing_id()` method gives each record a unique and increasing ID. According to the documentation, as long as your data is put into less than roughly 1 billion partitions with less than 8 billions records in each, the ID is guaranteed to be unique.

A word of caution: in earlier versions of Spark the `.monotonicallymonotonically_increasing_id()` method would not necessarily return the same IDs across multiple evaluations of the same DataFrame. This, however, has been fixed in Spark 2.0.

Missing observations

You will frequently encounter datasets with *blanks* in them. The missing values can happen for a variety of reasons: systems failure, people error, data schema changes, just to name a few.

The simplest way to deal with missing values, if your data can afford it, is to drop the whole observation when any missing value is found. You have to be careful not to drop too many: depending on the distribution of the missing values across your dataset it might severely affect the usability of your dataset. If, after dropping the rows, I end up with a very small dataset, or find that the reduction in data size is more than 50%, I start checking my data to see what features have the most holes in them and perhaps exclude those altogether; if a feature has most of its values missing (unless a missing value bears a meaning), from a modeling point of view, it is fairly useless.

The other way to deal with the observations with missing values is to impute some value in place of those `Nones`. Given the type of your data, you have several options to choose from:

- If your data is a discrete Boolean, you can turn it into a categorical variable by adding a third category — `Missing`
- If your data is already categorical, you can simply extend the number of levels and add the `Missing` category as well
- If you're dealing with ordinal or numerical data, you can impute either mean, median, or some other predefined value (for example, first or third quartile, depending on the distribution shape of your data)

Consider a similar example to the one we presented previously:

```
df_miss = spark.createDataFrame([
        (1, 143.5, 5.6, 28,   'M',  100000),
        (2, 167.2, 5.4, 45,   'M',  None),
        (3, None , 5.2, None, None, None),
        (4, 144.5, 5.9, 33,   'M',  None),
        (5, 133.2, 5.7, 54,   'F',  None),
        (6, 124.1, 5.2, None, 'F',  None),
        (7, 129.2, 5.3, 42,   'M',  76000),
    ], ['id', 'weight', 'height', 'age', 'gender', 'income'])
```

In our example, we deal with a number of missing values categories.

Analyzing *rows*, we see the following:

- The row with ID `3` has only one useful piece of information—the `height`
- The row with ID `6` has only one missing value—the `age`

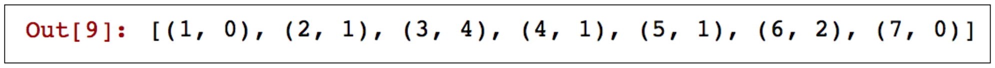
Analyzing *columns*, we can see the following:

- The `income` column, since it is a very personal thing to disclose, has most of its values missing
- The `weight` and `gender` columns have only one missing value each
- The `age` column has two missing values

To find the number of missing observations per row, we can use the following snippet:

```
df_miss.rdd.map(
    lambda row: (row['id'], sum([c == None for c in row]))
).collect()
```

It produces the following output:



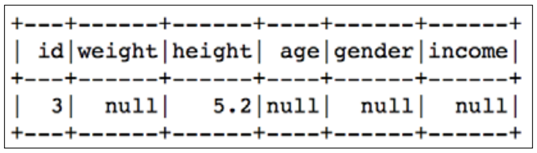
```
Out[9]: [(1, 0), (2, 1), (3, 4), (4, 1), (5, 1), (6, 2), (7, 0)]
```

It tells us that, for example, the row with ID `3` has four missing observations, as we observed earlier.

Let's see what values are missing so that when we count missing observations in columns, we can decide whether to drop the observation altogether or impute some of the observations:

```
df_miss.where('id == 3').show()
```

Here's what we get:

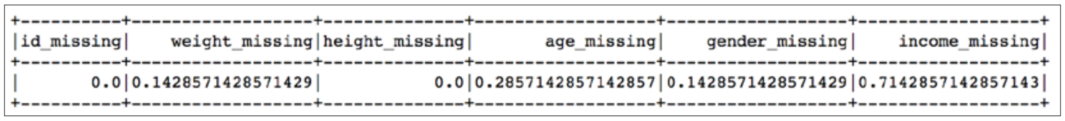


```
+---+------+------+----+------+------+
| id|weight|height| age|gender|income|
+---+------+------+----+------+------+
|  3|  null|   5.2|null|  null|  null|
+---+------+------+----+------+------+
```

Let's now check what percentage of missing observations are there in each column:

```
df_miss.agg(*[
    (1 - (fn.count(c) / fn.count('*'))).alias(c + '_missing')
    for c in df_miss.columns
]).show()
```

This generates the following output:



```
+----------+------------------+--------------+------------------+------------------+------------------+
|id_missing|    weight_missing|height_missing|       age_missing|    gender_missing|    income_missing|
+----------+------------------+--------------+------------------+------------------+------------------+
|       0.0|0.1428571428571429|           0.0|0.2857142857142857|0.1428571428571429|0.7142857142857143|
+----------+------------------+--------------+------------------+------------------+------------------+
```

[The `*` argument to the `.count(...)` method (in place of a column name) instructs the method to count all rows. On the other hand, the `*` preceding the list declaration instructs the `.agg(...)` method to treat the list as a set of separate parameters passed to the function.]

So, we have 14% of missing observations in the `weight` and `gender` columns, twice as much in the `height` column, and almost 72% of missing observations in the `income` column. Now we know what to do.

First, we will drop the `'income'` feature, as most of its values are missing.

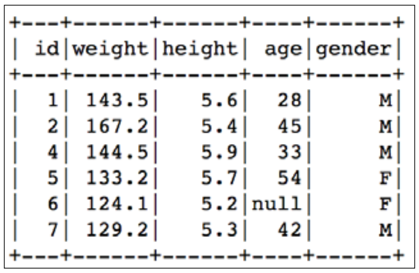
```
df_miss_no_income = df_miss.select([
    c for c in df_miss.columns if c != 'income'
])
```

We now see that we do not need to drop the row with ID `3` as the coverage in the `'weight'` and `'age'` columns has enough observations (in our simplified example) to calculate the mean and impute it in the place of the missing values.

However, if you decide to drop the observations instead, you can use the `.dropna(...)` method, as shown here. Here, we will also use the `thresh` parameter, which allows us to specify a threshold on the number of missing observations per row that would qualify the row to be dropped. This is useful if you have a dataset with tens or hundreds of features and you only want to drop those rows that exceed a certain threshold of missing values:

```
df_miss_no_income.dropna(thresh=3).show()
```

The preceding code produces the following output:



```
+---+------+------+----+------+
| id|weight|height| age|gender|
+---+------+------+----+------+
|  1| 143.5|   5.6|  28|     M|
|  2| 167.2|   5.4|  45|     M|
|  4| 144.5|   5.9|  33|     M|
|  5| 133.2|   5.7|  54|     F|
|  6| 124.1|   5.2|null|     F|
|  7| 129.2|   5.3|  42|     M|
+---+------+------+----+------+
```

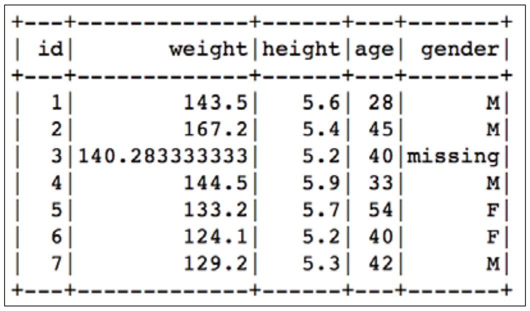
On the other hand, if you wanted to impute the observations, you can use the `.fillna(...)` method. This method accepts a single integer (long is also accepted), float, or string; all missing values in the whole dataset will then be filled in with that value. You can also pass a dictionary of a form `{'<colName>': <value_to_impute>}`. This has the same limitation, in that, as the `<value_to_impute>`, you can only pass an integer, float, or string.

If you want to impute a mean, median, or other calculated value, you need to first calculate the value, create a dictionary with such values, and then pass it to the `.fillna(...)` method.

Here's how we do it:

```
means = df_miss_no_income.agg(
    *[fn.mean(c).alias(c)
        for c in df_miss_no_income.columns if c != 'gender']
).toPandas().to_dict('records')[0]

means['gender'] = 'missing'

df_miss_no_income.fillna(means).show()
```

The preceding code will produce the following output:



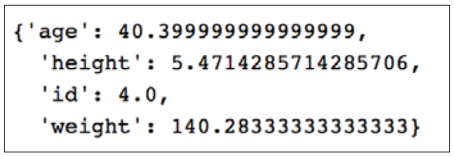
```
+---+-------------+------+---+-------+
| id|       weight|height|age| gender|
+---+-------------+------+---+-------+
|  1|        143.5|   5.6| 28|      M|
|  2|        167.2|   5.4| 45|      M|
|  3|140.283333333|   5.2| 40|missing|
|  4|        144.5|   5.9| 33|      M|
|  5|        133.2|   5.7| 54|      F|
|  6|        124.1|   5.2| 40|      F|
|  7|        129.2|   5.3| 42|      M|
+---+-------------+------+---+-------+
```

We omit the `gender` column as one cannot calculate a mean of a categorical variable, obviously.

We use a double conversion here. Taking the output of the `.agg(...)` method (a PySpark DataFrame), we first convert it into a pandas' DataFrame and then once more to a dictionary.

Note that calling the `.toPandas()` can be problematic, as the method works essentially in the same way as `.collect()` in RDDs. It collects all the information from the workers and brings it over to the driver. It is unlikely to be a problem with the preceding dataset, unless you have thousands upon thousands of features.

The `records` parameter to the `.to_dict(...)` method of pandas instructs it to create the following dictionary:



```
{'age': 40.399999999999999,
  'height': 5.4714285714285706,
  'id': 4.0,
  'weight': 140.28333333333333}
```

Since we cannot calculate the average (or any other numeric metric of a categorical variable), we added the `missing` category to the dictionary for the `gender` feature. Note that, even though the mean of the age column is 40.40, when imputed, the type of the `df_miss_no_income.age` column was preserved—it is still an integer.

Outliers

Outliers are those observations that deviate significantly from the distribution of the rest of your sample. The definitions of *significance* vary, but in the most general form, you can accept that there are no outliers if all the values are roughly within the Q1−1.5IQR and Q3+1.5IQR range, where IQR is the interquartile range; the IQR is defined as a difference between the upper- and lower-quartiles, that is, the 75th percentile (the Q3) and 25th percentile (the Q1), respectively.

Let's, again, consider a simple example:

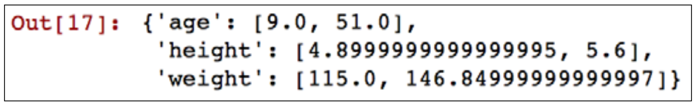
```
df_outliers = spark.createDataFrame([
        (1, 143.5, 5.3, 28),
        (2, 154.2, 5.5, 45),
        (3, 342.3, 5.1, 99),
        (4, 144.5, 5.5, 33),
        (5, 133.2, 5.4, 54),
        (6, 124.1, 5.1, 21),
        (7, 129.2, 5.3, 42),
    ], ['id', 'weight', 'height', 'age'])
```

Now we can use the definition we outlined previously to flag the outliers.

First, we calculate the lower and upper cut off points for each feature. We will use the `.approxQuantile(...)` method. The first parameter specified is the name of the column, the second parameter can be either a number between `0` or `1` (where `0.5` means to calculated median) or a list (as in our case), and the third parameter specifies the acceptable level of an error for each metric (if set to `0`, it will calculate an exact value for the metric, but it can be really expensive to do so):

```
cols = ['weight', 'height', 'age']
bounds = {}

for col in cols:
    quantiles = df_outliers.approxQuantile(
        col, [0.25, 0.75], 0.05
    )

    IQR = quantiles[1] - quantiles[0]

    bounds[col] = [
        quantiles[0] - 1.5 * IQR,
        quantiles[1] + 1.5 * IQR
]
```

The `bounds` dictionary holds the lower and upper bounds for each feature:

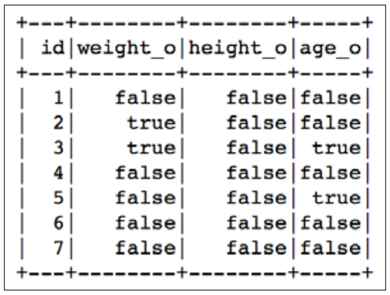


```
Out[17]: {'age': [9.0, 51.0],
          'height': [4.8999999999999995, 5.6],
          'weight': [115.0, 146.84999999999997]}
```

Let's now use it to flag our outliers:

```
outliers = df_outliers.select(*['id'] + [
    (
        (df_outliers[c] < bounds[c][0]) |
        (df_outliers[c] > bounds[c][1])
    ).alias(c + '_o') for c in cols
])
outliers.show()
```

The preceding code produces the following output:

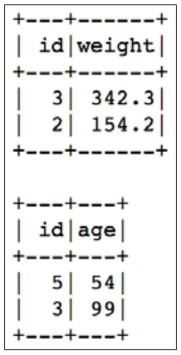


```
+---+--------+--------+-----+
| id|weight_o|height_o|age_o|
+---+--------+--------+-----+
|  1|   false|   false|false|
|  2|    true|   false|false|
|  3|    true|   false| true|
|  4|   false|   false|false|
|  5|   false|   false| true|
|  6|   false|   false|false|
|  7|   false|   false|false|
+---+--------+--------+-----+
```

We have two outliers in the `weight` feature and two in the `age` feature. By now you should know how to extract these, but here is a snippet that lists the values significantly differing from the rest of the distribution:

```
df_outliers = df_outliers.join(outliers, on='id')
df_outliers.filter('weight_o').select('id', 'weight').show()
df_outliers.filter('age_o').select('id', 'age').show()
```

The preceding code will give you the following output:



```
+---+------+
| id|weight|
+---+------+
|  3| 342.3|
|  2| 154.2|
+---+------+

+---+---+
| id|age|
+---+---+
|  5| 54|
|  3| 99|
+---+---+
```

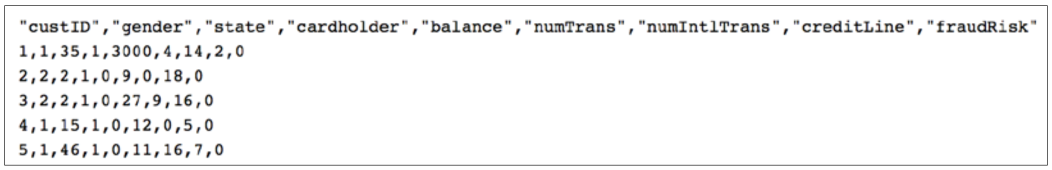
Equipped with the methods described in this section, you can quickly clean up even the biggest of datasets.

Getting familiar with your data

Although we would strongly discourage such behavior, you can build a model without knowing your data; it will most likely take you longer, and the quality of the resulting model might be less than optimal, but it is doable.

> In this section, we will use the dataset we downloaded from `http://packages.revolutionanalytics.com/datasets/ccFraud.csv`. We did not alter the dataset itself, but it was GZipped and uploaded to `http://tomdrabas.com/data/LearningPySpark/ccFraud.csv.gz`. Please download the file first and save it in the same folder that contains your notebook for this chapter.

The head of the dataset looks as follows:



```
"custID","gender","state","cardholder","balance","numTrans","numIntlTrans","creditLine","fraudRisk"
1,1,35,1,3000,4,14,2,0
2,2,2,1,0,9,0,18,0
3,2,2,1,0,27,9,16,0
4,1,15,1,0,12,0,5,0
5,1,46,1,0,11,16,7,0
```

Thus, any serious data scientist or data modeler will become acquainted with the dataset before starting any modeling. As a first thing, we normally start with some descriptive statistics to get a feeling for what we are dealing with.

Descriptive statistics

Descriptive statistics, in the simplest sense, will tell you the basic information about your dataset: how many non-missing observations there are in your dataset, the mean and the standard deviation for the column, as well as the min and max values.

However, first things first—let's load our data and convert it to a Spark DataFrame:

```
import pyspark.sql.types as typ
```

First, we load the only module we will need. The `pyspark.sql.types` exposes all the data types we can use, such as `IntegerType()` or `FloatType()`.

> For a full list of available types check `http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.types`.

Next, we read the data in and remove the header line using the `.filter(...)` method. This is followed by splitting the row on each comma (since this is a `.csv` file) and converting each element to an integer:

```
fraud = sc.textFile('ccFraud.csv.gz')
header = fraud.first()

fraud = fraud \
    .filter(lambda row: row != header) \
    .map(lambda row: [int(elem) for elem in row.split(',')])
```

Next, we create the schema for our DataFrame:

```
fields = [
    *[
        typ.StructField(h[1:-1], typ.IntegerType(), True)
        for h in header.split(',')
    ]
]
schema = typ.StructType(fields)
```

Finally, we create our DataFrame:

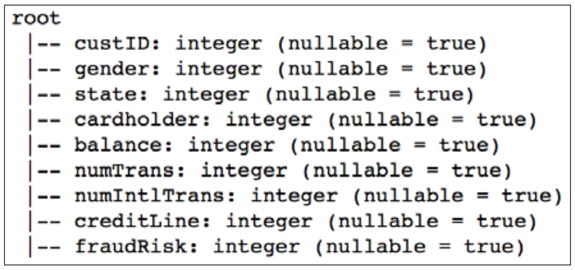
```
fraud_df = spark.createDataFrame(fraud, schema)
```

Having created our `fraud_df` DataFrame, we can calculate the basic descriptive statistics for our dataset. However, you need to remember that even though all of our features appear as numeric in nature, some of them are categorical (for example, `gender` or `state`).

Here's the schema of our DataFrame:

```
fraud_df.printSchema()
```

The representation is shown here:



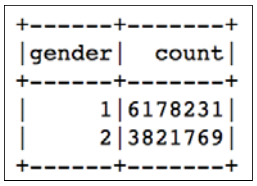
```
root
 |-- custID: integer (nullable = true)
 |-- gender: integer (nullable = true)
 |-- state: integer (nullable = true)
 |-- cardholder: integer (nullable = true)
 |-- balance: integer (nullable = true)
 |-- numTrans: integer (nullable = true)
 |-- numIntlTrans: integer (nullable = true)
 |-- creditLine: integer (nullable = true)
 |-- fraudRisk: integer (nullable = true)
```

Also, no information would be gained from calculating the mean and standard deviation of the `custId` column, so we will not be doing that.

For a better understanding of categorical columns, we will count the frequencies of their values using the `.groupby(...)` method. In this example, we will count the frequencies of the `gender` column:

```
fraud_df.groupby('gender').count().show()
```

The preceding code will produce the following output:



```
+------+-------+
|gender|  count|
+------+-------+
|     1|6178231|
|     2|3821769|
+------+-------+
```

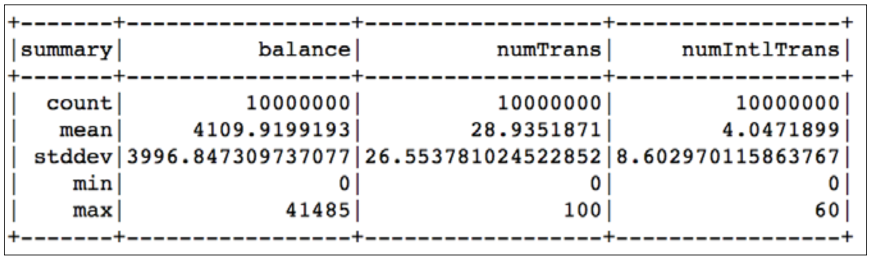
As you can see, we are dealing with a fairly imbalanced dataset. What you would expect to see is an equal distribution for both genders.

It goes beyond the scope of this chapter, but if we were building a statistical model, you would need to take care of these kinds of biases. You can read more at `http://www.va.gov/VETDATA/docs/SurveysAndStudies/SAMPLE_WEIGHT.pdf`.

For the truly numerical features, we can use the `.describe()` method:

```
numerical = ['balance', 'numTrans', 'numIntlTrans']
desc = fraud_df.describe(numerical)
desc.show()
```

The `.show()` method will produce the following output:



```
+-------+-----------------+------------------+-----------------+
|summary|          balance|          numTrans|     numIntlTrans|
+-------+-----------------+------------------+-----------------+
|  count|         10000000|          10000000|         10000000|
|   mean|     4109.9199193|        28.9351871|        4.0471899|
| stddev|3996.847309737077|26.553781024522852|8.602970115863767|
|    min|                0|                 0|                0|
|    max|            41485|               100|               60|
+-------+-----------------+------------------+-----------------+
```

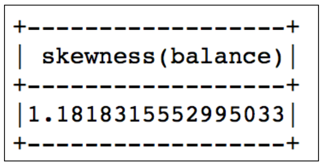
Even from these relatively few numbers we can tell quite a bit:

- All of the features are positively skewed. The maximum values are a number of times larger than the average.
- The coefficient of variation (the ratio of mean to standard deviation) is very high (close or greater than `1`), suggesting a wide spread of observations.

Here's how you check the `skeweness` (we will do it for the `'balance'` feature only):

```
fraud_df.agg({'balance': 'skewness'}).show()
```

The preceding code produces the following output:



```
+------------------+
| skewness(balance)|
+------------------+
|1.1818315552995033|
+------------------+
```

A list of aggregation functions (the names are fairly self-explanatory) includes: `avg()`, `count()`, `countDistinct()`, `first()`, `kurtosis()`, `max()`, `mean()`, `min()`, `skewness()`, `stddev()`, `stddev_pop()`, `stddev_samp()`, `sum()`, `sumDistinct()`, `var_pop()`, `var_samp()` and `variance()`.

Correlations

Another highly useful measure of mutual relationships between features is correlation. Your model would normally include only those features that are highly correlated with your target. However, it is almost equally important to check the correlation between the features; including features that are highly correlated among them (that is, are *collinear*) may lead to unpredictable behavior of your model, or might unnecessarily complicate it.

I talk more about multicollinearity in my other book, *Practical Data Analysis Cookbook, Packt Publishing* (`https://www.packtpub.com/big-data-and-business-intelligence/practical-data-analysis-cookbook`), in *Chapter 5, Introducing MLlib*, under the section titled *Identifying and tackling multicollinearity*.

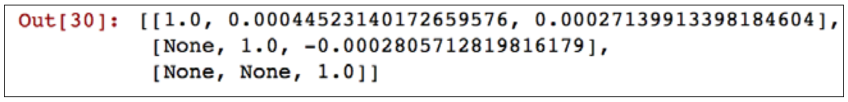
Calculating correlations in PySpark is very easy once your data is in a DataFrame form. The only difficulties are that the `.corr(...)` method supports the Pearson correlation coefficient at the moment, and it can only calculate pairwise correlations, such as the following:

```
fraud_df.corr('balance', 'numTrans')
```

In order to create a correlations matrix, you can use the following script:

```
n_numerical = len(numerical)

corr = []

for i in range(0, n_numerical):
    temp = [None] * i

    for j in range(i, n_numerical):
        temp.append(fraud_df.corr(numerical[i], numerical[j]))
    corr.append(temp)
```

The preceding code will create the following output:



```
Out[30]: [[1.0, 0.00044523140172659576, 0.00027139913398184604],
          [None, 1.0, -0.00028057128198161791],
          [None, None, 1.0]]
```

As you can see, the correlations between the numerical features in the credit card fraud dataset are pretty much non-existent. Thus, all these features can be used in our models, should they turn out to be statistically sound in explaining our target.

Having checked the correlations, we can now move on to visually inspecting our data.

Visualization

There are multiple visualization packages, but in this section we will be using `matplotlib` and Bokeh exclusively to give you the best tools for your needs.

Both of the packages come preinstalled with Anaconda. First, let's load the modules and set them up:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

```
import bokeh.charts as chrt
from bokeh.io import output_notebook

output_notebook()
```

The `%matplotlib inline` and the `output_notebook()` commands will make every chart generated with `matplotlib` or Bokeh, respectively, appear within the notebook and not as a separate window.

Histograms

Histograms are by far the easiest way to visually gauge the distribution of your features. There are three ways you can generate histograms in PySpark (or a Jupyter notebook):

- Aggregate the data in workers and return an aggregated list of bins and counts in each bin of the histogram to the driver
- Return all the data points to the driver and allow the plotting libraries' methods to do the job for you
- Sample your data and then return them to the driver for plotting.

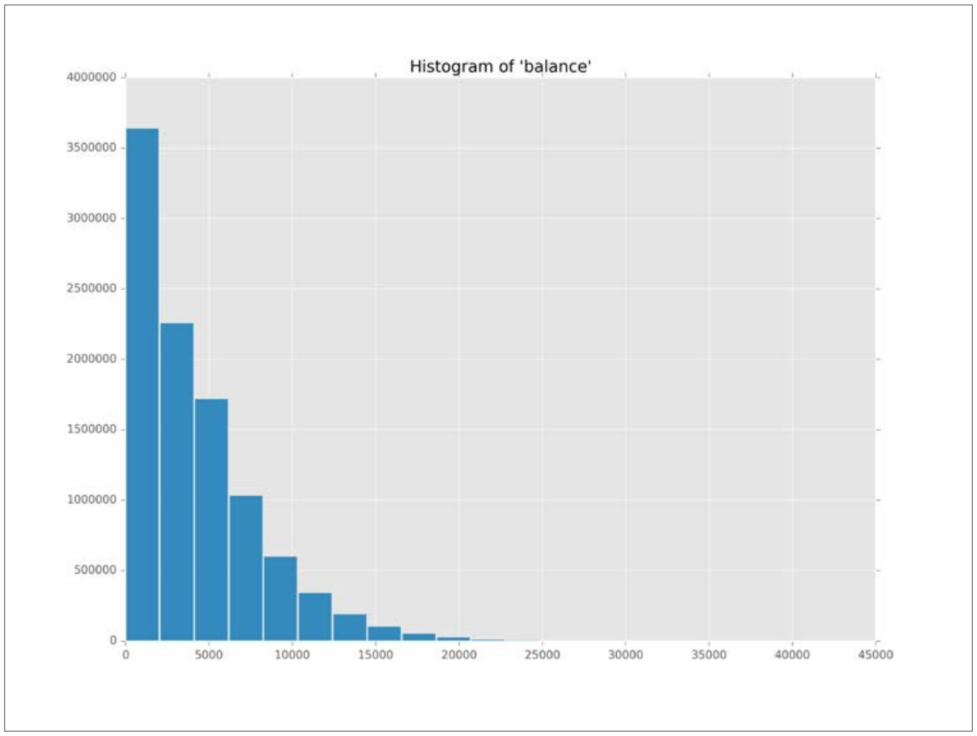
If the number of rows in your dataset is counted in billions, then the second option might not be attainable. Thus, you need to aggregate the data first:

```
hists = fraud_df.select('balance').rdd.flatMap(
    lambda row: row
).histogram(20)
```

To plot the histogram, you can simply call `matplotlib`, as shown in the following code:

```
data = {
    'bins': hists[0][:-1],
    'freq': hists[1]
}
plt.bar(data['bins'], data['freq'], width=2000)
plt.title('Histogram of \'balance\'')
```

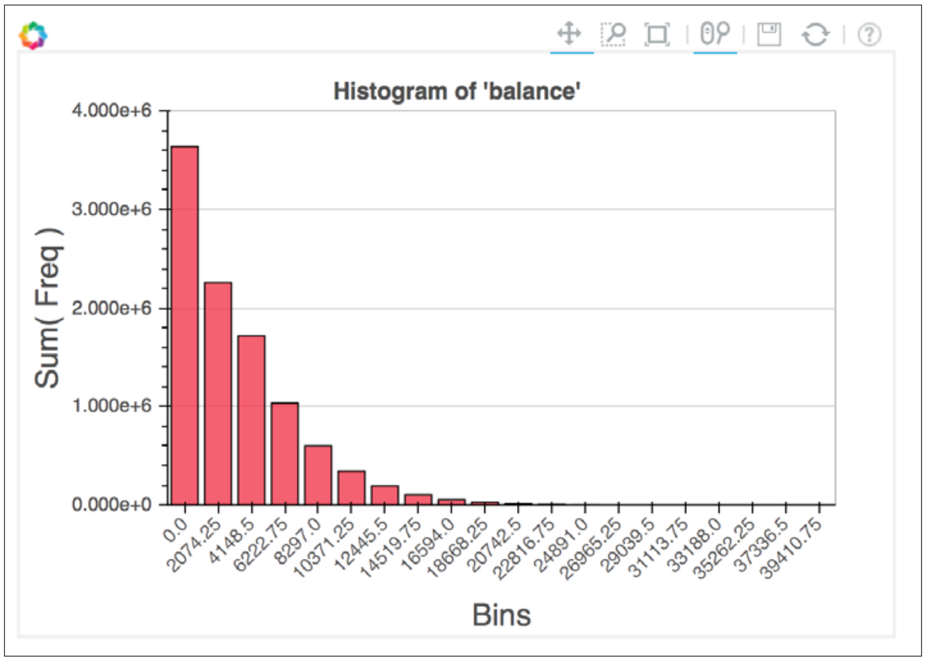
This will produce the following chart:



In a similar manner, a histogram can be created with Bokeh:

```
b_hist = chrt.Bar(
    data,
    values='freq', label='bins',
    title='Histogram of \'balance\'')
chrt.show(b_hist)
```

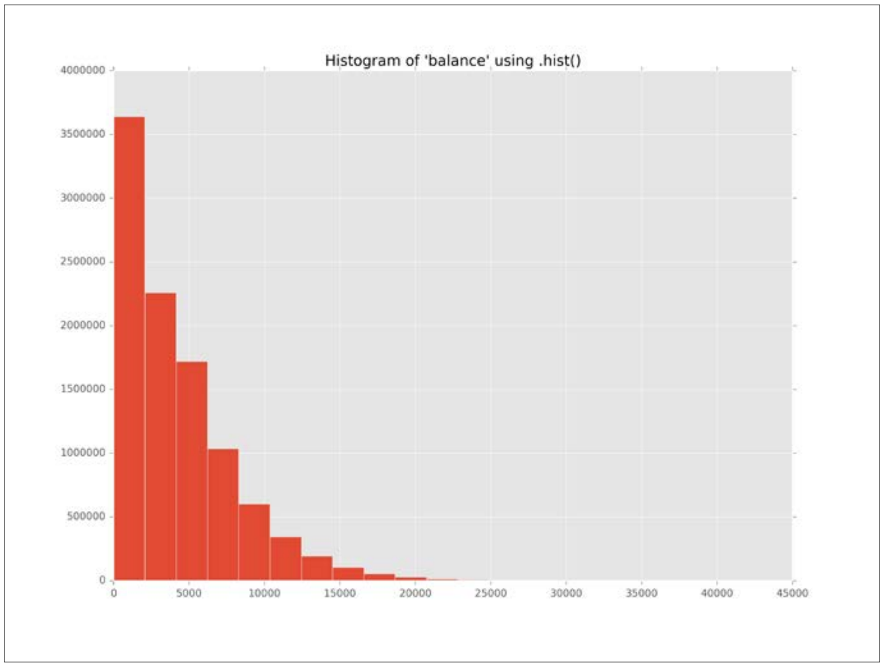
Since Bokeh uses D3.js in the background, the resulting chart is interactive:



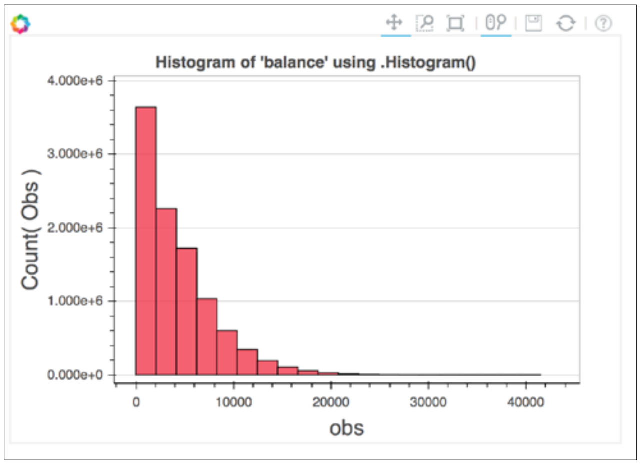
If your data is small enough to fit on the driver (although we would argue it would normally be faster to use the previous method), you can bring the data and use the `.hist(...)` (from `matplotlib`) or `.Histogram(...)` (from Bokeh) methods:

```
data_driver = {
    'obs': fraud_df.select('balance').rdd.flatMap(
        lambda row: row
    ).collect()
}
plt.hist(data_driver['obs'], bins=20)
plt.title('Histogram of \'balance\' using .hist()')
b_hist_driver = chrt.Histogram(
    data_driver, values='obs',
    title='Histogram of \'balance\' using .Histogram()',
    bins=20
)
chrt.show(b_hist_driver)
```

This will produce the following chart for `matplotlib`:



For Bokeh, the following chart will be generated:



Interactions between features

Scatter charts allow us to visualize interactions between up to three variables at a time (although we will be only presenting a 2D interaction in this section).

You should rarely revert to 3D visualizations unless you are dealing with some temporal data and you want to observe changes over time. Even then, we would rather discretize the time data and present a series of 2D charts, as interpreting 3D charts is somewhat more complicated and (most of the time) confusing.

Since PySpark does not offer any visualization modules on the server side, and trying to plot billions of observations at the same time would be highly impractical, in this section we will sample the dataset at 0.02% (roughly 2,000 observations).

Unless you chose a stratified sampling, you should create at least three to five samples at a predefined sampling fraction so you can check if your sample is somewhat representative of your dataset—that is, that the differences between your samples are not big.

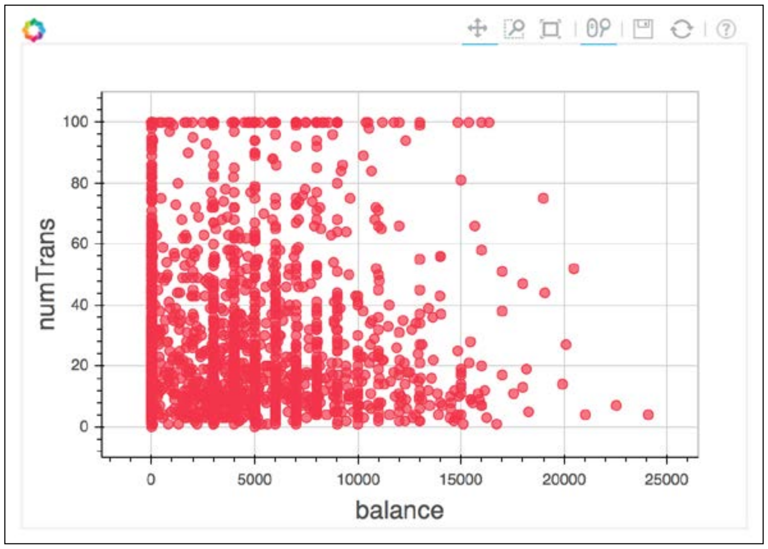
In this example, we will sample our fraud dataset at 0.02% given `'gender'` as a strata:

```
data_sample = fraud_df.sampleBy(
    'gender', {1: 0.0002, 2: 0.0002}
).select(numerical)
```

To put multiple 2D charts in one go, you can use the following code:

```
data_multi = dict([
    (elem, data_sample.select(elem).rdd \
        .flatMap(lambda row: row).collect())
    for elem in numerical
])
sctr = chrt.Scatter(data_multi, x='balance', y='numTrans')
chrt.show(sctr)
```

The preceding code will produce the following chart:



As you can see, there are plenty of fraudulent transactions that had 0 balance but many transactions—that is, a fresh card and big spike of transactions. However, no specific pattern can be shown apart from some *banding* occurring at $1,000 intervals.

Summary

In this chapter, we looked at how to clean and prepare your dataset for modeling by identifying and tackling datasets with missing values, duplicates, and outliers. We also looked at how to get a bit more familiar with your data using tools from PySpark (although this is by no means a full manual on how to analyze your datasets). Finally, we showed you how to chart your data.

We will use these (and more) techniques in the next two chapters, where we will be building machine learning models.

5
Introducing MLlib

In the previous chapter, we learned how to prepare the data for modeling. In this chapter, we will actually use some of that learning to build a classification model using the MLlib package of PySpark.

MLlib stands for Machine Learning Library. Even though MLlib is now in a maintenance mode, that is, it is not actively being developed (and will most likely be deprecated later), it is warranted that we cover at least some of the features of the library. In addition, MLlib is currently the only library that supports training models for streaming.

Starting with Spark 2.0, ML is the main machine learning library that operates on DataFrames instead of RDDs as is the case for MLlib.

The documentation for `MLlib` can be found here: `http://spark.apache.org/docs/latest/api/python/pyspark.mllib.html`.

In this chapter, you will learn how to do the following:

- Prepare the data for modeling with MLlib
- Perform statistical testing
- Predict survival chances of infants using logistic regression
- Select the most predictable features and train a random forest model

Overview of the package

At the high level, MLlib exposes three core machine learning functionalities:

- **Data preparation**: Feature extraction, transformation, selection, hashing of categorical features, and some natural language processing methods
- **Machine learning algorithms**: Some popular and advanced regression, classification, and clustering algorithms are implemented
- **Utilities**: Statistical methods such as descriptive statistics, chi-square testing, linear algebra (sparse and dense matrices and vectors), and model evaluation methods

As you can see, the palette of available functionalities allows you to perform almost all of the fundamental data science tasks.

In this chapter, we will build two classification models: a linear regression and a random forest. We will use a portion of the US 2014 and 2015 birth data we downloaded from `http://www.cdc.gov/nchs/data_access/vitalstatsonline.htm`; from the total of 300 variables we selected 85 features that we will use to build our models. Also, out of the total of almost 7.99 million records, we selected a balanced sample of 45,429 records: 22,080 records where infants were reported dead and 23,349 records with infants alive.

[The dataset we will use in this chapter can be downloaded from `http://www.tomdrabas.com/data/LearningPySpark/births_train.csv.gz`.]

Loading and transforming the data

Even though MLlib is designed with RDDs and DStreams in focus, for ease of transforming the data we will read the data and convert it to a DataFrame.

[The DStreams are the basic data abstraction for Spark Streaming (see `http://bit.ly/2jIDT2A`)]

Just like in the previous chapter, we first specify the schema of our dataset.

Note that here (for brevity), we only present a handful of features. You should always check our GitHub account for this book for the latest version of the code: `https://github.com/drabastomek/learningPySpark`.

Here's the code:

```
import pyspark.sql.types as typ
labels = [
    ('INFANT_ALIVE_AT_REPORT', typ.StringType()),
    ('BIRTH_YEAR', typ.IntegerType()),
    ('BIRTH_MONTH', typ.IntegerType()),
    ('BIRTH_PLACE', typ.StringType()),
    ('MOTHER_AGE_YEARS', typ.IntegerType()),
    ('MOTHER_RACE_6CODE', typ.StringType()),
    ('MOTHER_EDUCATION', typ.StringType()),
    ('FATHER_COMBINED_AGE', typ.IntegerType()),
    ('FATHER_EDUCATION', typ.StringType()),
    ('MONTH_PRECARE_RECODE', typ.StringType()),
    ...
    ('INFANT_BREASTFED', typ.StringType())
]
schema = typ.StructType([
        typ.StructField(e[0], e[1], False) for e in labels
    ])
```

Next, we load the data. The `.read.csv(...)` method can read either uncompressed or (as in our case) GZipped comma-separated values. The `header` parameter set to `True` indicates that the first row contains the header, and we use the `schema` to specify the correct data types:

```
births = spark.read.csv('births_train.csv.gz',
                        header=True,
                        schema=schema)
```

There are plenty of features in our dataset that are strings. These are mostly categorical variables that we need to somehow convert to a numeric form.

You can glimpse over the original file schema specification here: `ftp://ftp.cdc.gov/pub/Health_Statistics/NCHS/Dataset_Documentation/DVS/natality/UserGuide2015.pdf`.

We will first specify our recode dictionary:

```
recode_dictionary = {
    'YNU': {
        'Y': 1,
        'N': 0,
        'U': 0
    }
}
```

Our goal in this chapter is to predict whether the `'INFANT_ALIVE_AT_REPORT'` is either `1` or `0`. Thus, we will drop all of the features that relate to the infant and will try to predict the infant's chances of surviving only based on the features related to its mother, father, and the place of birth:

```
selected_features = [
    'INFANT_ALIVE_AT_REPORT',
    'BIRTH_PLACE',
    'MOTHER_AGE_YEARS',
    'FATHER_COMBINED_AGE',
    'CIG_BEFORE',
    'CIG_1_TRI',
    'CIG_2_TRI',
    'CIG_3_TRI',
    'MOTHER_HEIGHT_IN',
    'MOTHER_PRE_WEIGHT',
    'MOTHER_DELIVERY_WEIGHT',
    'MOTHER_WEIGHT_GAIN',
    'DIABETES_PRE',
    'DIABETES_GEST',
    'HYP_TENS_PRE',
    'HYP_TENS_GEST',
    'PREV_BIRTH_PRETERM'
]
births_trimmed = births.select(selected_features)
```

In our dataset, there are plenty of features with Yes/No/Unknown values; we will only code `Yes` to `1`; everything else will be set to `0`.

There is also a small problem with how the number of cigarettes smoked by the mother was coded: as 0 means the mother smoked no cigarettes before or during the pregnancy, between 1-97 states the actual number of cigarette smoked, 98 indicates either 98 or more, whereas 99 identifies the unknown; we will assume the unknown is 0 and recode accordingly.

So next we will specify our recoding methods:

```
import pyspark.sql.functions as func
def recode(col, key):
    return recode_dictionary[key][col]
def correct_cig(feat):
    return func \
        .when(func.col(feat) != 99, func.col(feat))\
        .otherwise(0)
rec_integer = func.udf(recode, typ.IntegerType())
```

The `recode` method looks up the correct key from the `recode_dictionary` (given the `key`) and returns the corrected value. The `correct_cig` method checks when the value of the feature `feat` is not equal to 99 and (for that situation) returns the value of the feature; if the value is equal to 99, we get 0 otherwise.

We cannot use the `recode` function directly on a `DataFrame`; it needs to be converted to a UDF that Spark will understand. The `rec_integer` is such a function: by passing our specified `recode` function and specifying the return value data type, we can use it then to encode our Yes/No/Unknown features.

So, let's get to it. First, we'll correct the features related to the number of cigarettes smoked:

```
births_transformed = births_trimmed \
    .withColumn('CIG_BEFORE', correct_cig('CIG_BEFORE'))\
    .withColumn('CIG_1_TRI', correct_cig('CIG_1_TRI'))\
    .withColumn('CIG_2_TRI', correct_cig('CIG_2_TRI'))\
    .withColumn('CIG_3_TRI', correct_cig('CIG_3_TRI'))
```

The `.withColumn(...)` method takes the name of the column as its first parameter and the transformation as the second one. In the previous cases, we do not create new columns, but reuse the same ones instead.

Now we will focus on correcting the Yes/No/Unknown features. First, we will figure out which these are with the following snippet:

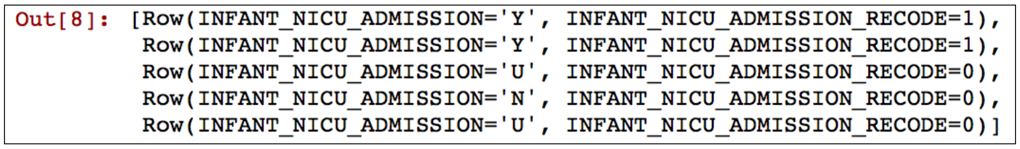
```
cols = [(col.name, col.dataType) for col in births_trimmed.schema]
YNU_cols = []
for i, s in enumerate(cols):
    if s[1] == typ.StringType():
        dis = births.select(s[0]) \
            .distinct() \
            .rdd \
            .map(lambda row: row[0]) \
            .collect()
        if 'Y' in dis:
            YNU_cols.append(s[0])
```

First, we created a list of tuples (`cols`) that hold column names and corresponding data types. Next, we loop through all of these and calculate distinct values of all string columns; if a `'Y'` is within the returned list, we append the column name to the `YNU_cols` list.

DataFrames can transform the features in bulk while selecting features. To present the idea, consider the following example:

```
births.select([
        'INFANT_NICU_ADMISSION',
        rec_integer(
            'INFANT_NICU_ADMISSION', func.lit('YNU')
        ) \
        .alias('INFANT_NICU_ADMISSION_RECODE')]
     ).take(5)
```

Here's what we get in return:



```
Out[8]: [Row(INFANT_NICU_ADMISSION='Y', INFANT_NICU_ADMISSION_RECODE=1),
         Row(INFANT_NICU_ADMISSION='Y', INFANT_NICU_ADMISSION_RECODE=1),
         Row(INFANT_NICU_ADMISSION='U', INFANT_NICU_ADMISSION_RECODE=0),
         Row(INFANT_NICU_ADMISSION='N', INFANT_NICU_ADMISSION_RECODE=0),
         Row(INFANT_NICU_ADMISSION='U', INFANT_NICU_ADMISSION_RECODE=0)]
```

We select the `'INFANT_NICU_ADMISSION'` column and we pass the name of the feature to the `rec_integer` method. We also alias the newly transformed column as `'INFANT_NICU_ADMISSION_RECODE'`. This way we will also confirm that our UDF works as intended.

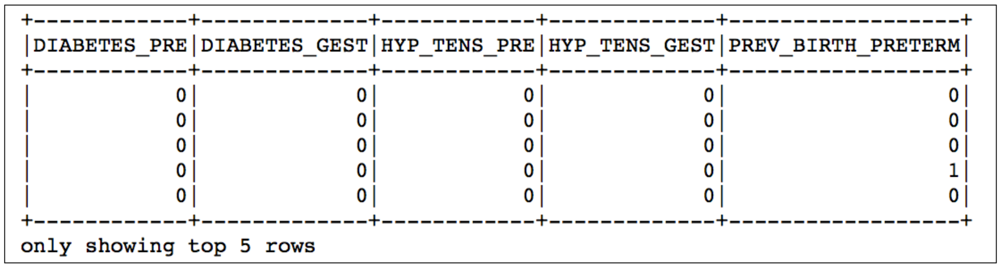
So, to transform all the `YNU_cols` in one go, we will create a list of such transformations, as shown here:

```
exprs_YNU = [
    rec_integer(x, func.lit('YNU')).alias(x)
    if x in YNU_cols
    else x
    for x in births_transformed.columns
]
births_transformed = births_transformed.select(exprs_YNU)
```

Let's check if we got it correctly:

```
births_transformed.select(YNU_cols[-5:]).show(5)
```

Here's what we get:



```
+------------+-------------+------------+-------------+------------------+
|DIABETES_PRE|DIABETES_GEST|HYP_TENS_PRE|HYP_TENS_GEST|PREV_BIRTH_PRETERM|
+------------+-------------+------------+-------------+------------------+
|           0|            0|           0|            0|                 0|
|           0|            0|           0|            0|                 0|
|           0|            0|           0|            0|                 0|
|           0|            0|           0|            0|                 1|
|           0|            0|           0|            0|                 0|
+------------+-------------+------------+-------------+------------------+
only showing top 5 rows
```

Looks like everything worked as we wanted it to work, so let's get to know our data better.

Getting to know your data

In order to build a statistical model in an informed way, an intimate knowledge of the dataset is necessary. Without knowing the data it is possible to build a successful model, but it is then a much more arduous task, or it would require more technical resources to test all the possible combinations of features. Therefore, after spending the required 80% of the time cleaning the data, we spend the next 15% getting to know it!

Descriptive statistics

I normally start with descriptive statistics. Even though the DataFrames expose the `.describe()` method, since we are working with `MLlib`, we will use the `.colStats(...)` method.

A word of warning: the `.colStats(...)` calculates the descriptive statistics based on a sample. For real world datasets this should not really matter but if your dataset has less than 100 observations you might get some strange results.

The method takes an `RDD` of data to calculate the descriptive statistics of and return a `MultivariateStatisticalSummary` object that contains the following descriptive statistics:

- `count()`: This holds a row count
- `max()`: This holds maximum value in the column
- `mean()`: This holds the value of the mean for the values in the column

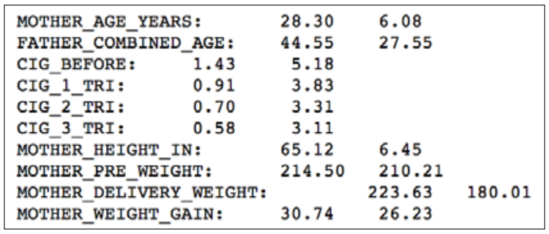
- `min()`: This holds the minimum value in the column
- `normL1()`: This holds the value of the L1-Norm for the values in the column
- `normL2()`: This holds the value of the L2-Norm for the values in the column
- `numNonzeros()`: This holds the number of nonzero values in the column
- `variance()`: This holds the value of the variance for the values in the column

[You can read more about the L1- and L2-norms here `http://bit.ly/2jJJPJ0`]

We recommend checking the documentation of Spark to learn more about these. The following is a snippet that calculates the descriptive statistics of the numeric features:

```
import pyspark.mllib.stat as st
import numpy as np
numeric_cols = ['MOTHER_AGE_YEARS','FATHER_COMBINED_AGE',
                'CIG_BEFORE','CIG_1_TRI','CIG_2_TRI','CIG_3_TRI',
                'MOTHER_HEIGHT_IN','MOTHER_PRE_WEIGHT',
                'MOTHER_DELIVERY_WEIGHT','MOTHER_WEIGHT_GAIN'
               ]
numeric_rdd = births_transformed\
                       .select(numeric_cols)\
                       .rdd \
                       .map(lambda row: [e for e in row])
mllib_stats = st.Statistics.colStats(numeric_rdd)
for col, m, v in zip(numeric_cols,
                     mllib_stats.mean(),
                     mllib_stats.variance()):
    print('{0}: \t{1:.2f} \t {2:.2f}'.format(col, m, np.sqrt(v)))
```

The preceding code produces the following result:



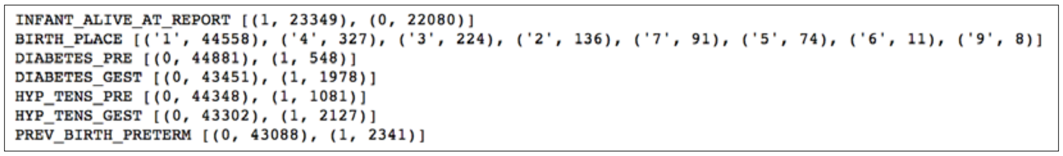
```
MOTHER_AGE_YEARS:        28.30    6.08
FATHER_COMBINED_AGE:     44.55    27.55
CIG_BEFORE:      1.43     5.18
CIG_1_TRI:       0.91     3.83
CIG_2_TRI:       0.70     3.31
CIG_3_TRI:       0.58     3.11
MOTHER_HEIGHT_IN:        65.12    6.45
MOTHER_PRE_WEIGHT:       214.50   210.21
MOTHER_DELIVERY_WEIGHT:         223.63   180.01
MOTHER_WEIGHT_GAIN:      30.74    26.23
```

As you can see, mothers, compared to fathers, are younger: the average age of mothers was 28 versus over 44 for fathers. A good indication (at least for some of the infants) was that many mothers quit smoking while being pregnant; it is horrifying, though, that there still were some that continued smoking.

For the categorical variables, we will calculate the frequencies of their values:

```
categorical_cols = [e for e in births_transformed.columns
                    if e not in numeric_cols]
categorical_rdd = births_transformed\
                       .select(categorical_cols)\
                       .rdd \
                       .map(lambda row: [e for e in row])
for i, col in enumerate(categorical_cols):
    agg = categorical_rdd \
        .groupBy(lambda row: row[i]) \
        .map(lambda row: (row[0], len(row[1])))
    print(col, sorted(agg.collect(),
                      key=lambda el: el[1],
                      reverse=True))
```

Here is what the results look like:



```
INFANT_ALIVE_AT_REPORT [(1, 23349), (0, 22080)]
BIRTH_PLACE [('1', 44558), ('4', 327), ('3', 224), ('2', 136), ('7', 91), ('5', 74), ('6', 11), ('9', 8)]
DIABETES_PRE [(0, 44881), (1, 548)]
DIABETES_GEST [(0, 43451), (1, 1978)]
HYP_TENS_PRE [(0, 44348), (1, 1081)]
HYP_TENS_GEST [(0, 43302), (1, 2127)]
PREV_BIRTH_PRETERM [(0, 43088), (1, 2341)]
```

Most of the deliveries happened in hospital (`BIRTH_PLACE` equal to `1`). Around 550 deliveries happened at home: some intentionally (`'BIRTH_PLACE'` equal to `3`), and some not (`'BIRTH_PLACE'` equal to `4`).

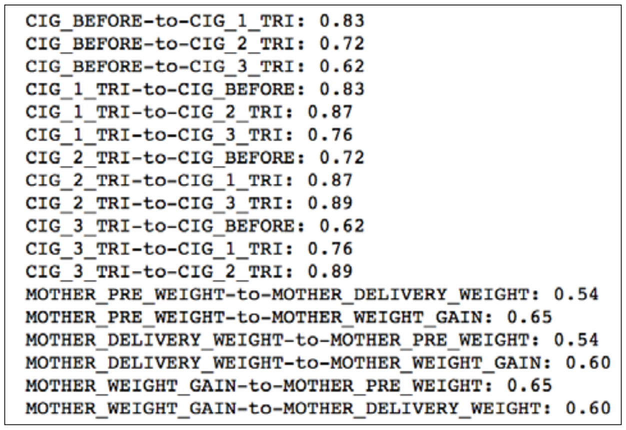
Correlations

Correlations help to identify collinear numeric features and handle them appropriately. Let's check the correlations between our features:

```
corrs = st.Statistics.corr(numeric_rdd)
for i, el in enumerate(corrs > 0.5):
    correlated = [
        (numeric_cols[j], corrs[i][j])
        for j, e in enumerate(el)
        if e == 1.0 and j != i]
    if len(correlated) > 0:
        for e in correlated:
            print('{0}-to-{1}: {2:.2f}' \
                  .format(numeric_cols[i], e[0], e[1]))
```

The preceding code will calculate the correlation matrix and will print only those features that have a correlation coefficient greater than `0.5`: the `corrs > 0.5` part takes care of that.

Here's what we get:



```
CIG_BEFORE-to-CIG_1_TRI: 0.83
CIG_BEFORE-to-CIG_2_TRI: 0.72
CIG_BEFORE-to-CIG_3_TRI: 0.62
CIG_1_TRI-to-CIG_BEFORE: 0.83
CIG_1_TRI-to-CIG_2_TRI: 0.87
CIG_1_TRI-to-CIG_3_TRI: 0.76
CIG_2_TRI-to-CIG_BEFORE: 0.72
CIG_2_TRI-to-CIG_1_TRI: 0.87
CIG_2_TRI-to-CIG_3_TRI: 0.89
CIG_3_TRI-to-CIG_BEFORE: 0.62
CIG_3_TRI-to-CIG_1_TRI: 0.76
CIG_3_TRI-to-CIG_2_TRI: 0.89
MOTHER_PRE_WEIGHT-to-MOTHER_DELIVERY_WEIGHT: 0.54
MOTHER_PRE_WEIGHT-to-MOTHER_WEIGHT_GAIN: 0.65
MOTHER_DELIVERY_WEIGHT-to-MOTHER_PRE_WEIGHT: 0.54
MOTHER_DELIVERY_WEIGHT-to-MOTHER_WEIGHT_GAIN: 0.60
MOTHER_WEIGHT_GAIN-to-MOTHER_PRE_WEIGHT: 0.65
MOTHER_WEIGHT_GAIN-to-MOTHER_DELIVERY_WEIGHT: 0.60
```

As you can see, the `'CIG_...'` features are highly correlated, so we can drop most of them. Since we want to predict the survival chances of an infant as soon as possible, we will keep only the `'CIG_1_TRI'`. Also, as expected, the weight features are also highly correlated and we will only keep the `'MOTHER_PRE_WEIGHT'`:

```
features_to_keep = [
    'INFANT_ALIVE_AT_REPORT',
    'BIRTH_PLACE',
    'MOTHER_AGE_YEARS',
    'FATHER_COMBINED_AGE',
    'CIG_1_TRI',
    'MOTHER_HEIGHT_IN',
    'MOTHER_PRE_WEIGHT',
    'DIABETES_PRE',
    'DIABETES_GEST',
    'HYP_TENS_PRE',
    'HYP_TENS_GEST',
    'PREV_BIRTH_PRETERM'
]
births_transformed = births_transformed.select([e for e in features_
to_keep])
```

Statistical testing

We cannot calculate correlations for the categorical features. However, we can run a Chi-square test to determine if there are significant differences.

Here's how you can do it using the `.chiSqTest(...)` method of `MLlib`:

```
import pyspark.mllib.linalg as ln
for cat in categorical_cols[1:]:
    agg = births_transformed \
        .groupby('INFANT_ALIVE_AT_REPORT') \
        .pivot(cat) \
        .count()
    agg_rdd = agg \
        .rdd \
        .map(lambda row: (row[1:])) \
        .flatMap(lambda row:
                 [0 if e == None else e for e in row]) \
        .collect()
    row_length = len(agg.collect()[0]) - 1
    agg = ln.Matrices.dense(row_length, 2, agg_rdd)

    test = st.Statistics.chiSqTest(agg)
    print(cat, round(test.pValue, 4))
```

We loop through all the categorical variables and pivot them by the `'INFANT_ALIVE_AT_REPORT'` feature to get the counts. Next, we transform them into an RDD, so we can then convert them into a matrix using the `pyspark.mllib.linalg` module. The first parameter to the `.Matrices.dense(...)` method specifies the number of rows in the matrix; in our case, it is the length of distinct values of the categorical feature.

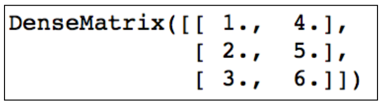
The second parameter specifies the number of columns: we have two as our `'INFANT_ALIVE_AT_REPORT'` target variable has only two values.

The last parameter is a list of values to be transformed into a matrix.

Here's an example that shows this more clearly:

```
print(ln.Matrices.dense(3,2, [1,2,3,4,5,6]))
```

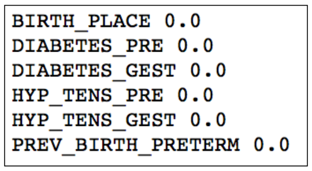
The preceding code produces the following matrix:



```
DenseMatrix([[ 1.,  4.],
             [ 2.,  5.],
             [ 3.,  6.]])
```

Once we have our counts in a matrix form, we can use the `.chiSqTest(...)` to calculate our test.

Here's what we get in return:



```
BIRTH_PLACE 0.0
DIABETES_PRE 0.0
DIABETES_GEST 0.0
HYP_TENS_PRE 0.0
HYP_TENS_GEST 0.0
PREV_BIRTH_PRETERM 0.0
```

Our tests reveal that all the features should be significantly different and should help us predict the chance of survival of an infant.

Creating the final dataset

Therefore, it is time to create our final dataset that we will use to build our models. We will convert our DataFrame into an RDD of `LabeledPoints`.

A `LabeledPoint` is a MLlib structure that is used to train the machine learning models. It consists of two attributes: `label` and `features`.

The `label` is our target variable and `features` can be a NumPy `array`, `list`, `pyspark.mllib.linalg.SparseVector`, `pyspark.mllib.linalg.DenseVector`, or `scipy.sparse` column matrix.

Creating an RDD of LabeledPoints

Before we build our final dataset, we first need to deal with one final obstacle: our `'BIRTH_PLACE'` feature is still a string. While any of the other categorical variables can be used as is (as they are now dummy variables), we will use a hashing trick to encode the `'BIRTH_PLACE'` feature:

```
import pyspark.mllib.feature as ft
import pyspark.mllib.regression as reg
hashing = ft.HashingTF(7)
births_hashed = births_transformed \
    .rdd \
    .map(lambda row: [
            list(hashing.transform(row[1]).toArray())
                if col == 'BIRTH_PLACE'
                else row[i]
            for i, col
            in enumerate(features_to_keep)]) \
    .map(lambda row: [[e] if type(e) == int else e
                      for e in row]) \
    .map(lambda row: [item for sublist in row
                      for item in sublist]) \
    .map(lambda row: reg.LabeledPoint(
            row[0],
            ln.Vectors.dense(row[1:]))
        )
```

First, we create the hashing model. Our feature has seven levels, so we use as many features as that for the hashing trick. Next, we actually use the model to convert our `'BIRTH_PLACE'` feature into a `SparseVector`; such a data structure is preferred if your dataset has many columns but in a row only a few of them have non-zero values. We then combine all the features together and finally create a `LabeledPoint`.

Splitting into training and testing

Before we move to the modeling stage, we need to split our dataset into two sets: one we'll use for training and the other for testing. Luckily, RDDs have a handy method to do just that: `.randomSplit(...)`. The method takes a list of proportions that are to be used to randomly split the dataset.

Here is how it is done:

```
births_train, births_test = births_hashed.randomSplit([0.6, 0.4])
```

That's it! Nothing more needs to be done.

Predicting infant survival

Finally, we can move to predicting the infants' survival chances. In this section, we will build two models: a linear classifier—the logistic regression, and a non-linear one—a random forest. For the former one, we will use all the features at our disposal, whereas for the latter one, we will employ a `ChiSqSelector(...)` method to select the top four features.

Logistic regression in MLlib

Logistic regression is somewhat a benchmark to build any classification model. MLlib used to provide a logistic regression model estimated using a **stochastic gradient descent** (**SGD**) algorithm. This model has been deprecated in Spark 2.0 in favor of the `LogisticRegressionWithLBFGS` model.

The `LogisticRegressionWithLBFGS` model uses the **Limited-memory Broyden–Fletcher–Goldfarb–Shanno** (**BFGS**) optimization algorithm. It is a quasi-Newton method that approximates the BFGS algorithm.

> For those of you who are mathematically adept and interested in this, we suggest perusing this blog post that is a nice walk-through of the optimization algorithms: `http://aria42.com/blog/2014/12/understanding-lbfgs`.

First, we train the model on our data:

```
from pyspark.mllib.classification \
    import LogisticRegressionWithLBFGS
LR_Model = LogisticRegressionWithLBFGS \
    .train(births_train, iterations=10)
```

Training the model is very simple: we just need to call the `.train(...)` method. The required parameters are the RDD with `LabeledPoints`; we also specified the number of `iterations` so it does not take too long to run.

Having trained the model using the `births_train` dataset, let's use the model to predict the classes for our testing set:

```
LR_results = (
        births_test.map(lambda row: row.label) \
        .zip(LR_Model \
             .predict(births_test\
                      .map(lambda row: row.features)))
    ).map(lambda row: (row[0], row[1] * 1.0))
```

The preceding snippet creates an RDD where each element is a tuple, with the first element being the actual label and the second one, the model's prediction.

MLlib provides an evaluation metric for classification and regression. Let's check how well or how bad our model performed:

```
import pyspark.mllib.evaluation as ev
LR_evaluation = ev.BinaryClassificationMetrics(LR_results)
print('Area under PR: {0:.2f}' \
      .format(LR_evaluation.areaUnderPR))
print('Area under ROC: {0:.2f}' \
      .format(LR_evaluation.areaUnderROC))
LR_evaluation.unpersist()
```

Here's what we got:

```
Area under PR: 0.85
Area under ROC: 0.63
```

The model performed reasonably well! The 85% area under the Precision-Recall curve indicates a good fit. In this case, we might be getting slightly more predicted deaths (true and false positives). In this case, this is actually a good thing as it would allow doctors to put the expectant mother and the infant under special care.

The area under **Receiver-Operating Characteristic** (**ROC**) can be understood as a probability of the model ranking higher than a randomly chosen positive instance compared to a randomly chosen negative one. A 63% value can be thought of as acceptable.

For more on these metrics, we point interested readers to `http://stats.stackexchange.com/questions/7207/roc-vs-precision-and-recall-curves` and `http://gim.unmc.edu/dxtests/roc3.htm`.

Selecting only the most predictable features

Any model that uses less features to predict a class accurately should always be preferred to a more complex one. MLlib allows us to select the most predictable features using a Chi-Square selector.

Here's how you do it:

```
selector = ft.ChiSqSelector(4).fit(births_train)
topFeatures_train = (
        births_train.map(lambda row: row.label) \
        .zip(selector \
             .transform(births_train \
                        .map(lambda row: row.features)))
    ).map(lambda row: reg.LabeledPoint(row[0], row[1]))
topFeatures_test = (
        births_test.map(lambda row: row.label) \
        .zip(selector \
             .transform(births_test \
                        .map(lambda row: row.features)))
    ).map(lambda row: reg.LabeledPoint(row[0], row[1]))
```

We asked the selector to return the four most predictive features from the dataset and train the selector using the `births_train` dataset. We then used the model to extract only those features from our training and testing datasets.

The `.ChiSqSelector(...)` method can only be used for numerical features; categorical variables need to be either hashed or dummy coded before the selector can be used.

Random forest in MLlib

We are now ready to build the random forest model.

The following code shows you how to do it:

```
from pyspark.mllib.tree import RandomForest
RF_model = RandomForest \
    .trainClassifier(data=topFeatures_train,
                     numClasses=2,
                     categoricalFeaturesInfo={},
                     numTrees=6,
                     featureSubsetStrategy='all',
                     seed=666)
```

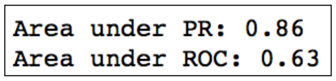
The first parameter to the `.trainClassifier(...)` method specifies the training dataset. The `numClasses` one indicates how many classes our target variable has. As the third parameter, you can pass a dictionary where the key is the index of a categorical feature in our RDD and the value for the key indicates the number of levels that the categorical feature has. The `numTrees` specifies the number of trees to be in the forest. The next parameter tells the model to use all the features in our dataset instead of keeping only the most descriptive ones, while the last one specifies the seed for the stochastic part of the model.

Let's see how well our model did:

```
RF_results = (
        topFeatures_test.map(lambda row: row.label) \
        .zip(RF_model \
             .predict(topFeatures_test \
                      .map(lambda row: row.features)))
    )
RF_evaluation = ev.BinaryClassificationMetrics(RF_results)
print('Area under PR: {0:.2f}' \
      .format(RF_evaluation.areaUnderPR))
```

```
print('Area under ROC: {0:.2f}' \
      .format(RF_evaluation.areaUnderROC))
model_evaluation.unpersist()
```

Here are the results:

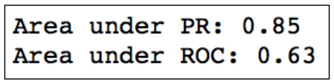


```
Area under PR: 0.86
Area under ROC: 0.63
```

As you can see, the Random Forest model with fewer features performed even better than the logistic regression model. Let's see how the logistic regression would perform with a reduced number of features:

```
LR_Model_2 = LogisticRegressionWithLBFGS \
    .train(topFeatures_train, iterations=10)
LR_results_2 = (
        topFeatures_test.map(lambda row: row.label) \
        .zip(LR_Model_2 \
             .predict(topFeatures_test \
                      .map(lambda row: row.features)))
    ).map(lambda row: (row[0], row[1] * 1.0))
LR_evaluation_2 = ev.BinaryClassificationMetrics(LR_results_2)
print('Area under PR: {0:.2f}' \
      .format(LR_evaluation_2.areaUnderPR))
print('Area under ROC: {0:.2f}' \
      .format(LR_evaluation_2.areaUnderROC))
LR_evaluation_2.unpersist()
```

The results might surprise you:



```
Area under PR: 0.85
Area under ROC: 0.63
```

As you can see, both models can be simplified and still attain the same level of accuracy. Having said that, you should always opt for a model with fewer variables.

Summary

In this chapter, we looked at the capabilities of the `MLlib` package of PySpark. Even though the package is currently in a maintenance mode and is not actively being worked on, it is still good to know how to use it. Also, for now it is the only package available to train models while streaming data. We used `MLlib` to clean up, transform, and get familiar with the dataset of infant deaths. Using that knowledge we then successfully built two models that aimed at predicting the chance of infant survival given the information about its mother, father, and place of birth.

In the next chapter, we will revisit the same problem, but using the newer package that is currently the Spark recommended package for machine learning.

6
Introducing the ML Package

In the previous chapter, we worked with the MLlib package in Spark that operated strictly on RDDs. In this chapter, we move to the ML part of Spark that operates strictly on DataFrames. Also, according to the Spark documentation, the primary machine learning API for Spark is now the DataFrame-based set of models contained in the `spark.ml` package.

So, let's get to it!

In this chapter, we will reuse a portion of the dataset we played within the previous chapter. The data can be downloaded from `http://www.tomdrabas.com/data/LearningPySpark/births_transformed.csv.gz`.

In this chapter, you will learn how to do the following:

- Prepare transformers, estimators, and pipelines
- Predict the chances of infant survival using models available in the ML package
- Evaluate the performance of the model
- Perform parameter hyper-tuning
- Use other machine-learning models available in the package

Overview of the package

At the top level, the package exposes three main abstract classes: a `Transformer`, an `Estimator`, and a `Pipeline`. We will shortly explain each with some short examples. We will provide more concrete examples of some of the models in the last section of this chapter.

Transformer

The `Transformer` class, like the name suggests, *transforms* your data by (normally) appending a new column to your DataFrame.

At the high level, when deriving from the `Transformer` abstract class, each and every new `Transformer` needs to implement a `.transform(...)` method. The method, as a first and normally the only obligatory parameter, requires passing a DataFrame to be transformed. This, of course, varies *method-by-method* in the ML package: other *popular* parameters are `inputCol` and `outputCol`; these, however, frequently default to some predefined values, such as, for example, `'features'` for the `inputCol` parameter.

There are many `Transformers` offered in the `spark.ml.feature` and we will briefly describe them here (before we use some of them later in this chapter):

- `Binarizer`: Given a threshold, the method takes a continuous variable and transforms it into a binary one.
- `Bucketizer`: Similar to the `Binarizer`, this method takes a list of thresholds (the `splits` parameter) and transforms a continuous variable into a multinomial one.
- `ChiSqSelector`: For the categorical target variables (think classification models), this feature allows you to select a predefined number of features (parameterized by the `numTopFeatures` parameter) that explain the variance in the target the best. The selection is done, as the name of the method suggests, using a Chi-Square test. It is one of the two-step methods: first, you need to `.fit(...)` your data (so the method can calculate the Chi-square tests). Calling the `.fit(...)` method (you pass your DataFrame as a parameter) returns a `ChiSqSelectorModel` object that you can then use to transform your DataFrame using the `.transform(...)` method.

More information on Chi-squares can be found here: `http://ccnmtl.columbia.edu/projects/qmss/the_chisquare_test/about_the_chisquare_test.html`.

- `CountVectorizer`: This is useful for a tokenized text (such as `[['Learning', 'PySpark', 'with', 'us'],['us', 'us', 'us']]`). It is one of two-step methods: first, you need to `.fit(...)`, that is, learn the patterns from your dataset, before you can `.transform(...)` with the `CountVectorizerModel` returned by the `.fit(...)` method. The output from this transformer, for the tokenized text presented previously, would look similar to this: `[(4, [0, 1, 2, 3], [1.0, 1.0, 1.0, 1.0]),(4, [3], [3.0])]`.

- `DCT`: The Discrete Cosine Transform takes a vector of real values and returns a vector of the same length, but with the sum of cosine functions oscillating at different frequencies. Such transformations are useful to extract some underlying frequencies in your data or in data compression.
- `ElementwiseProduct`: A method that returns a vector with elements that are products of the vector passed to the method, and a vector passed as the `scalingVec` parameter. For example, if you had a `[10.0, 3.0, 15.0]` vector and your `scalingVec` was `[0.99, 3.30, 0.66]`, then the vector you would get would look as follows: `[9.9, 9.9, 9.9]`.
- `HashingTF`: A hashing trick transformer that takes a list of tokenized text and returns a vector (of predefined length) with counts. From PySpark's documentation:

  "*Since a simple modulo is used to transform the hash function to a column index, it is advisable to use a power of two as the numFeatures parameter; otherwise the features will not be mapped evenly to the columns.*"

- `IDF`: This method computes an **Inverse Document Frequency** for a list of documents. Note that the documents need to already be represented as a vector (for example, using either the `HashingTF` or `CountVectorizer`).
- `IndexToString`: A complement to the `StringIndexer` method. It uses the encoding from the `StringIndexerModel` object to reverse the string index to original values. As an aside, please note that this sometimes does not work and you need to specify the values from the `StringIndexer`.
- `MaxAbsScaler`: Rescales the data to be within the `[-1.0, 1.0]` range (thus, it does not shift the center of the data).
- `MinMaxScaler`: This is similar to the `MaxAbsScaler` with the difference that it scales the data to be in the `[0.0, 1.0]` range.
- `NGram`: This method takes a list of tokenized text and returns *n-grams*: pairs, triples, or *n-mores* of subsequent words. For example, if you had a `['good', 'morning', 'Robin', 'Williams']` vector you would get the following output: `['good morning', 'morning Robin', 'Robin Williams']`.
- `Normalizer`: This method scales the data to be of unit norm using the p-norm value (by default, it is L2).
- `OneHotEncoder`: This method encodes a categorical column to a column of binary vectors.
- `PCA`: Performs the data reduction using principal component analysis.

- `PolynomialExpansion`: Performs a polynomial expansion of a vector. For example, if you had a vector symbolically written as `[x, y, z]`, the method would produce the following expansion: `[x, x*x, y, x*y, y*y, z, x*z, y*z, z*z]`.
- `QuantileDiscretizer`: Similar to the `Bucketizer` method, but instead of passing the splits parameter, you pass the `numBuckets` one. The method then decides, by calculating approximate quantiles over your data, what the splits should be.
- `RegexTokenizer`: This is a string tokenizer using regular expressions.
- `RFormula`: For those of you who are avid R users, you can pass a formula such as `vec ~ alpha * 3 + beta` (assuming your `DataFrame` has the `alpha` and `beta` columns) and it will produce the `vec` column given the expression.
- `SQLTransformer`: Similar to the previous, but instead of R-like formulas, you can use SQL syntax.

> The `FROM` statement should be selecting from `__THIS__`, indicating you are accessing the DataFrame. For example: `SELECT alpha * 3 + beta AS vec FROM __THIS__`.

- `StandardScaler`: Standardizes the column to have a 0 mean and standard deviation equal to 1.
- `StopWordsRemover`: Removes stop words (such as `'the'` or `'a'`) from a tokenized text.
- `StringIndexer`: Given a list of all the words in a column, this will produce a vector of indices.
- `Tokenizer`: This is the default tokenizer that converts the string to lower case and then splits on space(s).
- `VectorAssembler`: This is a highly useful transformer that collates multiple numeric (vectors included) columns into a single column with a vector representation. For example, if you had three columns in your DataFrame:

  ```
  df = spark.createDataFrame(
      [(12, 10, 3), (1, 4, 2)],
      ['a', 'b', 'c'])
  ```

  The output of calling:

  ```
  ft.VectorAssembler(inputCols=['a', 'b', 'c'],
          outputCol='features')\
      .transform(df) \
      .select('features')\
      .collect()
  ```

It would look as follows:

```
[Row(features=DenseVector([12.0, 10.0, 3.0])),
 Row(features=DenseVector([1.0, 4.0, 2.0]))]
```

- `VectorIndexer`: This is a method for indexing categorical columns into a vector of indices. It works in a *column-by-column* fashion, selecting distinct values from the column, sorting and returning an index of the value from the map instead of the original value.
- `VectorSlicer`: Works on a feature vector, either dense or sparse: given a list of indices, it extracts the values from the feature vector.
- `Word2Vec`: This method takes a sentence (string) as an input and transforms it into a map of `{string, vector}` format, a representation that is useful in natural language processing.

Note that there are many methods in the ML package that have an E letter next to it; this means the method is currently in beta (or Experimental) and it sometimes might fail or produce erroneous results. Beware.

Estimators

Estimators can be thought of as statistical models that need to be estimated to make predictions or classify your observations.

If deriving from the abstract `Estimator` class, the new model has to implement the `.fit(...)` method that fits the model given the data found in a DataFrame and some default or user-specified parameters.

There are a lot of estimators available in PySpark and we will now shortly describe the models available in Spark 2.0.

Classification

The ML package provides a data scientist with seven classification models to choose from. These range from the simplest ones (such as logistic regression) to more sophisticated ones. We will provide short descriptions of each of them in the following section:

- `LogisticRegression`: The benchmark model for classification. The logistic regression uses a logit function to calculate the probability of an observation belonging to a particular class. At the time of writing, the PySpark ML supports only binary classification problems.

- `DecisionTreeClassifier`: A classifier that builds a decision tree to predict a class for an observation. Specifying the `maxDepth` parameter limits the depth the tree grows, the `minInstancePerNode` determines the minimum number of observations in the tree node required to further split, the `maxBins` parameter specifies the maximum number of bins the continuous variables will be split into, and the `impurity` specifies the metric to measure and calculate the information gain from the split.
- `GBTClassifier`: A **Gradient Boosted Trees** model for classification. The model belongs to the family of ensemble models: models that combine multiple weak predictive models to form a strong one. At the moment, the `GBTClassifier` model supports binary labels, and continuous and categorical features.
- `RandomForestClassifier`: This model produces multiple decision trees (hence the name—forest) and uses the `mode` output of those decision trees to classify observations. The `RandomForestClassifier` supports both binary and multinomial labels.
- `NaiveBayes`: Based on the Bayes' theorem, this model uses conditional probability theory to classify observations. The `NaiveBayes` model in PySpark ML supports both binary and multinomial labels.
- `MultilayerPerceptronClassifier`: A classifier that mimics the nature of a human brain. Deeply rooted in the Artificial Neural Networks theory, the model is a black-box, that is, it is not easy to interpret the internal parameters of the model. The model consists, at a minimum, of three, fully connected `layers` (a parameter that needs to be specified when creating the model object) of artificial neurons: the input layer (that needs to be equal to the number of features in your dataset), a number of hidden layers (at least one), and an output layer with the number of neurons equal to the number of categories in your label. All the neurons in the input and hidden layers have a sigmoid activation function, whereas the activation function of the neurons in the output layer is softmax.
- `OneVsRest`: A reduction of a multiclass classification to a binary one. For example, in the case of a multinomial label, the model can train multiple binary logistic regression models. For example, if `label == 2`, the model will build a logistic regression where it will convert the `label == 2` to `1` (all remaining label values would be set to `0`) and then train a binary model. All the models are then scored and the model with the highest probability wins.

Regression

There are seven models available for regression tasks in the PySpark ML package. As with classification, these range from some basic ones (such as the obligatory linear regression) to more complex ones:

- `AFTSurvivalRegression`: Fits an Accelerated Failure Time regression model. It is a parametric model that assumes that a marginal effect of one of the features accelerates or decelerates a life expectancy (or process failure). It is highly applicable for the processes with well-defined stages.
- `DecisionTreeRegressor`: Similar to the model for classification with an obvious distinction that the label is continuous instead of binary (or multinomial).
- `GBTRegressor`: As with the `DecisionTreeRegressor`, the difference is the data type of the label.
- `GeneralizedLinearRegression`: A family of linear models with differing kernel functions (link functions). In contrast to the linear regression that assumes normality of error terms, the GLM allows the label to have different error term distributions: the `GeneralizedLinearRegression` model from the PySpark ML package supports `gaussian`, `binomial`, `gamma`, and `poisson` families of error distributions with a host of different link functions.
- `IsotonicRegression`: A type of regression that fits a free-form, non-decreasing line to your data. It is useful to fit the datasets with ordered and increasing observations.
- `LinearRegression`: The most simple of regression models, it assumes a linear relationship between features and a continuous label, and normality of error terms.
- `RandomForestRegressor`: Similar to either `DecisionTreeRegressor` or `GBTRegressor`, the `RandomForestRegressor` fits a continuous label instead of a discrete one.

Clustering

Clustering is a family of unsupervised models that are used to find underlying patterns in your data. The PySpark ML package provides the four most popular models at the moment:

- `BisectingKMeans`: A combination of the k-means clustering method and hierarchical clustering. The algorithm begins with all observations in a single cluster and iteratively splits the data into `k` clusters.

[Check out this website for more information on pseudo-algorithms: `http://minethedata.blogspot.com/2012/08/bisecting-k-means.html`.]

- `KMeans`: This is the famous k-mean algorithm that separates data into `k` clusters, iteratively searching for centroids that minimize the sum of square distances between each observation and the centroid of the cluster it belongs to.
- `GaussianMixture`: This method uses `k` Gaussian distributions with unknown parameters to dissect the dataset. Using the Expectation-Maximization algorithm, the parameters for the Gaussians are found by maximizing the log-likelihood function.

[Beware that for datasets with many features this model might perform poorly due to the curse of dimensionality and numerical issues with Gaussian distributions.]

- `LDA`: This model is used for topic modeling in natural language processing applications.

There is also one recommendation model available in PySpark ML, but we will refrain from describing it here.

Pipeline

A `Pipeline` in PySpark ML is a concept of an *end-to-end* transformation-estimation process (with distinct stages) that ingests some raw data (in a DataFrame form), performs the necessary data carpentry (transformations), and finally estimates a statistical model (estimator).

[A `Pipeline` can be purely transformative, that is, consisting of `Transformers` only.]

A `Pipeline` can be thought of as a chain of multiple discrete stages. When a `.fit(...)` method is executed on a `Pipeline` object, all the stages are executed in the order they were specified in the `stages` parameter; the `stages` parameter is a list of `Transformer` and `Estimator` objects. The `.fit(...)` method of the `Pipeline` object executes the `.transform(...)` method for the `Transformers` and the `.fit(...)` method for the `Estimators`.

Normally, the output of a preceding stage becomes the input for the following stage: when deriving from either the `Transformer` or `Estimator` abstract classes, one needs to implement the `.getOutputCol()` method that returns the value of the `outputCol` parameter specified when creating an object.

Predicting the chances of infant survival with ML

In this section, we will use the portion of the dataset from the previous chapter to present the ideas of PySpark ML.

If you have not yet downloaded the data while reading the previous chapter, it can be accessed here: `http://www.tomdrabas.com/data/LearningPySpark/births_transformed.csv.gz`.

In this section, we will, once again, attempt to predict the chances of the survival of an infant.

Loading the data

First, we load the data with the help of the following code:

```
import pyspark.sql.types as typ
labels = [
    ('INFANT_ALIVE_AT_REPORT', typ.IntegerType()),
    ('BIRTH_PLACE', typ.StringType()),
    ('MOTHER_AGE_YEARS', typ.IntegerType()),
    ('FATHER_COMBINED_AGE', typ.IntegerType()),
    ('CIG_BEFORE', typ.IntegerType()),
    ('CIG_1_TRI', typ.IntegerType()),
    ('CIG_2_TRI', typ.IntegerType()),
    ('CIG_3_TRI', typ.IntegerType()),
    ('MOTHER_HEIGHT_IN', typ.IntegerType()),
    ('MOTHER_PRE_WEIGHT', typ.IntegerType()),
    ('MOTHER_DELIVERY_WEIGHT', typ.IntegerType()),
    ('MOTHER_WEIGHT_GAIN', typ.IntegerType()),
    ('DIABETES_PRE', typ.IntegerType()),
    ('DIABETES_GEST', typ.IntegerType()),
    ('HYP_TENS_PRE', typ.IntegerType()),
    ('HYP_TENS_GEST', typ.IntegerType()),
    ('PREV_BIRTH_PRETERM', typ.IntegerType())
]
schema = typ.StructType([
    typ.StructField(e[0], e[1], False) for e in labels
])
births = spark.read.csv('births_transformed.csv.gz',
                        header=True,
                        schema=schema)
```

We specify the schema of the DataFrame; our severely limited dataset now only has 17 columns.

Creating transformers

Before we can use the dataset to estimate a model, we need to do some transformations. Since statistical models can only operate on numeric data, we will have to encode the `BIRTH_PLACE` variable.

Before we do any of this, since we will use a number of different feature transformations later in this chapter, let's import them all:

```
import pyspark.ml.feature as ft
```

To encode the `BIRTH_PLACE` column, we will use the `OneHotEncoder` method. However, the method cannot accept `StringType` columns; it can only deal with numeric types so first we will cast the column to an `IntegerType`:

```
births = births \
    .withColumn('BIRTH_PLACE_INT', births['BIRTH_PLACE'] \
    .cast(typ.IntegerType()))
```

Having done this, we can now create our first `Transformer`:

```
encoder = ft.OneHotEncoder(
    inputCol='BIRTH_PLACE_INT',
    outputCol='BIRTH_PLACE_VEC')
```

Let's now create a single column with all the features collated together. We will use the `VectorAssembler` method:

```
featuresCreator = ft.VectorAssembler(
    inputCols=[
        col[0]
        for col
        in labels[2:]] + \
    [encoder.getOutputCol()],
    outputCol='features'
)
```

The `inputCols` parameter passed to the `VectorAssembler` object is a list of all the columns to be combined together to form the `outputCol`—the `'features'`. Note that we use the output of the encoder object (by calling the `.getOutputCol()` method), so we do not have to remember to change this parameter's value should we change the name of the output column in the encoder object at any point.

It's now time to create our first estimator.

Creating an estimator

In this example, we will (once again) use the logistic regression model. However, later in the chapter, we will showcase some more complex models from the `.classification` set of PySpark ML models, so we load the whole section:

```
import pyspark.ml.classification as cl
```

Once loaded, let's create the model by using the following code:

```
logistic = cl.LogisticRegression(
    maxIter=10,
    regParam=0.01,
    labelCol='INFANT_ALIVE_AT_REPORT')
```

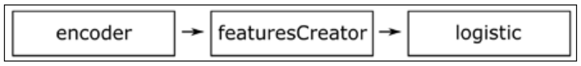
We would not have to specify the `labelCol` parameter if our target column had the name `'label'`. Also, if the output of our `featuresCreator` was not called `'features'`, we would have to specify the `featuresCol` by (most conveniently) calling the `getOutputCol()` method on the `featuresCreator` object.

Creating a pipeline

All that is left now is to create a `Pipeline` and fit the model. First, let's load the `Pipeline` from the ML package:

```
from pyspark.ml import Pipeline
```

Creating a `Pipeline` is really easy. Here's how our pipeline should look like conceptually:



Converting this structure into a `Pipeline` is a *walk in the park*:

```
pipeline = Pipeline(stages=[
        encoder,
        featuresCreator,
        logistic
    ])
```

That's it! Our `pipeline` is now created so we can (finally!) estimate the model.

Fitting the model

Before you fit the model, we need to split our dataset into training and testing datasets. Conveniently, the DataFrame API has the `.randomSplit(...)` method:

```
births_train, births_test = births \
    .randomSplit([0.7, 0.3], seed=666)
```

The first parameter is a list of dataset proportions that should end up in, respectively, `births_train` and `births_test` subsets. The `seed` parameter provides a seed to the randomizer.

> You can also split the dataset into more than two subsets as long as the elements of the list sum up to 1, and you unpack the output into as many subsets.
>
> For example, we could split the births dataset into three subsets like this:
>
> ```
> train, test, val = births.\
>     randomSplit([0.7, 0.2, 0.1], seed=666)
> ```
>
> The preceding code would put a random 70% of the births dataset into the `train` object, 20% would go to the `test`, and the `val` DataFrame would hold the remaining 10%.

Now it is about time to finally run our pipeline and estimate our model:

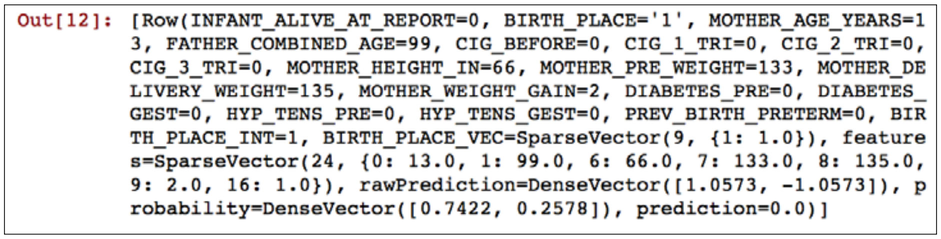
```
model = pipeline.fit(births_train)
test_model = model.transform(births_test)
```

The `.fit(...)` method of the pipeline object takes our training dataset as an input. Under the hood, the `births_train` dataset is passed first to the `encoder` object. The DataFrame that is created at the `encoder` stage then gets passed to the `featuresCreator` that creates the `'features'` column. Finally, the output from this stage is passed to the `logistic` object that estimates the final model.

The `.fit(...)` method returns the `PipelineModel` object (the `model` object in the preceding snippet) that can then be used for prediction; we attain this by calling the `.transform(...)` method and passing the testing dataset created earlier. Here's what the `test_model` looks like in the following command:

```
test_model.take(1)
```

It generates the following output:



```
Out[12]: [Row(INFANT_ALIVE_AT_REPORT=0, BIRTH_PLACE='1', MOTHER_AGE_YEARS=1
         3, FATHER_COMBINED_AGE=99, CIG_BEFORE=0, CIG_1_TRI=0, CIG_2_TRI=0,
         CIG_3_TRI=0, MOTHER_HEIGHT_IN=66, MOTHER_PRE_WEIGHT=133, MOTHER_DE
         LIVERY_WEIGHT=135, MOTHER_WEIGHT_GAIN=2, DIABETES_PRE=0, DIABETES_
         GEST=0, HYP_TENS_PRE=0, HYP_TENS_GEST=0, PREV_BIRTH_PRETERM=0, BIR
         TH_PLACE_INT=1, BIRTH_PLACE_VEC=SparseVector(9, {1: 1.0}), feature
         s=SparseVector(24, {0: 13.0, 1: 99.0, 6: 66.0, 7: 133.0, 8: 135.0,
         9: 2.0, 16: 1.0}), rawPrediction=DenseVector([1.0573, -1.0573]), p
         robability=DenseVector([0.7422, 0.2578]), prediction=0.0)]
```

As you can see, we get all the columns from the `Transfomers` and `Estimators`. The logistic regression model outputs several columns: the `rawPrediction` is the value of the linear combination of features and the β coefficients, the `probability` is the calculated probability for each of the classes, and finally, the `prediction` is our final class assignment.

Evaluating the performance of the model

Obviously, we would like to now test how well our model did. PySpark exposes a number of evaluation methods for classification and regression in the `.evaluation` section of the package:

```
import pyspark.ml.evaluation as ev
```

We will use the `BinaryClassficationEvaluator` to test how well our model performed:

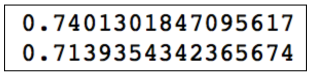
```
evaluator = ev.BinaryClassificationEvaluator(
    rawPredictionCol='probability',
    labelCol='INFANT_ALIVE_AT_REPORT')
```

The `rawPredictionCol` can either be the `rawPrediction` column produced by the estimator or the `probability`.

Let's see how well our model performed:

```
print(evaluator.evaluate(test_model,
    {evaluator.metricName: 'areaUnderROC'}))
print(evaluator.evaluate(test_model,
   {evaluator.metricName: 'areaUnderPR'}))
```

The preceding code produces the following result:



```
0.7401301847095617
0.7139354342365674
```

The area under the ROC of 74% and area under PR of 71% shows a well-defined model, but nothing out of extraordinary; if we had other features, we could drive this up, but this is not the purpose of this chapter (nor the book, for that matter).

Saving the model

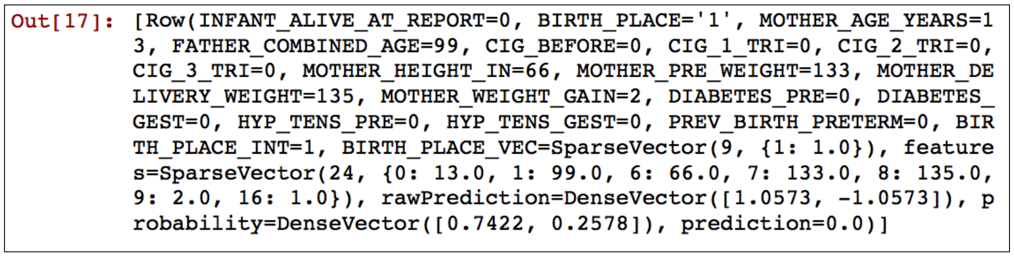
PySpark allows you to save the `Pipeline` definition for later use. It not only saves the pipeline structure, but also all the definitions of all the `Transformers` and `Estimators`:

```
pipelinePath = './infant_oneHotEncoder_Logistic_Pipeline'
pipeline.write().overwrite().save(pipelinePath)
```

So, you can load it up later and use it straight away to `.fit(...)` and predict:

```
loadedPipeline = Pipeline.load(pipelinePath)
loadedPipeline \
    .fit(births_train)\
    .transform(births_test)\
    .take(1)
```

The preceding code produces the same result (as expected):



```
Out[17]: [Row(INFANT_ALIVE_AT_REPORT=0, BIRTH_PLACE='1', MOTHER_AGE_YEARS=1
         3, FATHER_COMBINED_AGE=99, CIG_BEFORE=0, CIG_1_TRI=0, CIG_2_TRI=0,
         CIG_3_TRI=0, MOTHER_HEIGHT_IN=66, MOTHER_PRE_WEIGHT=133, MOTHER_DE
         LIVERY_WEIGHT=135, MOTHER_WEIGHT_GAIN=2, DIABETES_PRE=0, DIABETES_
         GEST=0, HYP_TENS_PRE=0, HYP_TENS_GEST=0, PREV_BIRTH_PRETERM=0, BIR
         TH_PLACE_INT=1, BIRTH_PLACE_VEC=SparseVector(9, {1: 1.0}), feature
         s=SparseVector(24, {0: 13.0, 1: 99.0, 6: 66.0, 7: 133.0, 8: 135.0,
         9: 2.0, 16: 1.0}), rawPrediction=DenseVector([1.0573, -1.0573]), p
         robability=DenseVector([0.7422, 0.2578]), prediction=0.0)]
```

If you, however, want to save the estimated model, you can also do that; instead of saving the `Pipeline`, you need to save the `PipelineModel`.

Note, that not only the `PipelineModel` can be saved: virtually all the models that are returned by calling the `.fit(...)` method on an `Estimator` or `Transformer` can be saved and loaded back to be reused.

To save your model, see the following the example:

```
from pyspark.ml import PipelineModel

modelPath = './infant_oneHotEncoder_Logistic_PipelineModel'
model.write().overwrite().save(modelPath)

loadedPipelineModel = PipelineModel.load(modelPath)
test_reloadedModel = loadedPipelineModel.transform(births_test)
```

The preceding script uses the `.load(...)` method, a class method of the `PipelineModel` class, to reload the estimated model. You can compare the result of `test_reloadedModel.take(1)` with the output of `test_model.take(1)` we presented earlier.

Parameter hyper-tuning

Rarely, our first model would be the best we can do. By simply looking at our metrics and accepting the model because it passed our pre-conceived performance thresholds is hardly a scientific method for finding the best model.

A concept of parameter hyper-tuning is to find the best parameters of the model: for example, the maximum number of iterations needed to properly estimate the logistic regression model or maximum depth of a decision tree.

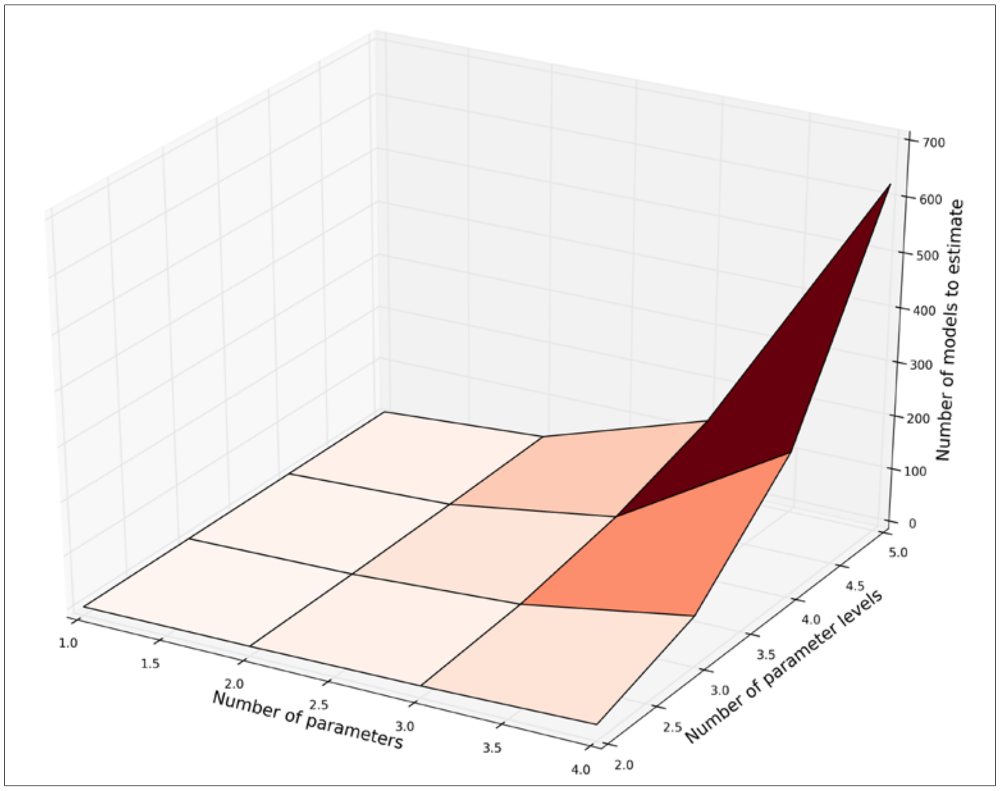
In this section, we will explore two concepts that allow us to find the best parameters for our models: grid search and train-validation splitting.

Grid search

Grid search is an exhaustive algorithm that loops through the list of defined parameter values, estimates separate models, and chooses the best one given some evaluation metric.

A note of caution should be stated here: if you define too many parameters you want to optimize over, or too many values of these parameters, it might take a lot of time to select the best model as the number of models to estimate would grow very quickly as the number of parameters and parameter values grow.

For example, if you want to fine-tune two parameters with two parameter values, you would have to fit four models. Adding one more parameter with two values would require estimating eight models, whereas adding one more additional value to our two parameters (bringing it to three values for each) would require estimating nine models. As you can see, this can quickly get out of hand if you are not careful. See the following chart to inspect this visually:



After this cautionary tale, let's get to fine-tuning our parameters space. First, we load the `.tuning` part of the package:

```
import pyspark.ml.tuning as tune
```

Next, let's specify our model and the list of parameters we want to loop through:

```
logistic = cl.LogisticRegression(
    labelCol='INFANT_ALIVE_AT_REPORT')
grid = tune.ParamGridBuilder() \
    .addGrid(logistic.maxIter,
             [2, 10, 50]) \
    .addGrid(logistic.regParam,
             [0.01, 0.05, 0.3]) \
    .build()
```

First, we specify the model we want to optimize the parameters of. Next, we decide which parameters we will be optimizing, and what values for those parameters to test. We use the `ParamGridBuilder()` object from the `.tuning` subpackage, and keep adding the parameters to the grid with the `.addGrid(...)` method: the first parameter is the parameter object of the model we want to optimize (in our case, these are `logistic.maxIter` and `logistic.regParam`), and the second parameter is a list of values we want to loop through. Calling the `.build()` method on the `.ParamGridBuilder` builds the grid.

Next, we need some way of comparing the models:

```
evaluator = ev.BinaryClassificationEvaluator(
    rawPredictionCol='probability',
    labelCol='INFANT_ALIVE_AT_REPORT')
```

So, once again, we'll use the `BinaryClassificationEvaluator`. It is time now to create the logic that will do the validation work for us:

```
cv = tune.CrossValidator(
    estimator=logistic,
    estimatorParamMaps=grid,
    evaluator=evaluator
)
```

The `CrossValidator` needs the `estimator`, the `estimatorParamMaps`, and the `evaluator` to do its job. The model loops through the grid of values, estimates the models, and compares their performance using the `evaluator`.

We cannot use the data straight away (as the `births_train` and `births_test` still have the `BIRTHS_PLACE` column not encoded) so we create a purely transforming `Pipeline`:

```
pipeline = Pipeline(stages=[encoder ,featuresCreator])
data_transformer = pipeline.fit(births_train)
```

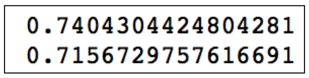
Having done this, we are ready to find the optimal combination of parameters for our model:

```
cvModel = cv.fit(data_transformer.transform(births_train))
```

The `cvModel` will return the best model estimated. We can now use it to see if it performed better than our previous model:

```
data_train = data_transformer \
    .transform(births_test)
results = cvModel.transform(data_train)
print(evaluator.evaluate(results,
     {evaluator.metricName: 'areaUnderROC'}))
print(evaluator.evaluate(results,
     {evaluator.metricName: 'areaUnderPR'}))
```

The preceding code will produce the following result:



```
0.7404304424804281
0.7156729757616691
```

As you can see, we got a slightly better result. What parameters does the best model have? The answer is a little bit convoluted, but here's how you can extract it:

```
results = [
    (
        [
            {key.name: paramValue}
            for key, paramValue
            in zip(
                params.keys(),
                params.values())
        ], metric
    )
    for params, metric
    in zip(
        cvModel.getEstimatorParamMaps(),
        cvModel.avgMetrics
    )
]
sorted(results,
       key=lambda el: el[1],
       reverse=True)[0]
```

The preceding code produces the following output:

```
Out[27]: ([{'maxIter': 50}, {'regParam': 0.01}], 2.2158632176362274)
```

Train-validation splitting

The `TrainValidationSplit` model, to select the best model, performs a random split of the input dataset (the training dataset) into two subsets: smaller training and validation subsets. The split is only performed once.

In this example, we will also use the `ChiSqSelector` to select only the top five features, thus limiting the complexity of our model:

```
selector = ft.ChiSqSelector(
    numTopFeatures=5,
    featuresCol=featuresCreator.getOutputCol(),
    outputCol='selectedFeatures',
    labelCol='INFANT_ALIVE_AT_REPORT'
)
```

The `numTopFeatures` specifies the number of features to return. We will put the selector after the `featuresCreator`, so we call the `.getOutputCol()` on the `featuresCreator`.

We covered creating the `LogisticRegression` and `Pipeline` earlier, so we will not explain how these are created again here:

```
logistic = cl.LogisticRegression(
    labelCol='INFANT_ALIVE_AT_REPORT',
    featuresCol='selectedFeatures'
)
pipeline = Pipeline(stages=[encoder, featuresCreator, selector])
data_transformer = pipeline.fit(births_train)
```

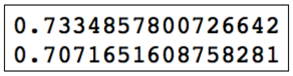
The `TrainValidationSplit` object gets created in the same fashion as the `CrossValidator` model:

```
tvs = tune.TrainValidationSplit(
    estimator=logistic,
    estimatorParamMaps=grid,
    evaluator=evaluator
)
```

As before, we fit our data to the model, and calculate the results:

```
tvsModel = tvs.fit(
    data_transformer \
        .transform(births_train)
)
data_train = data_transformer \
    .transform(births_test)
results = tvsModel.transform(data_train)
print(evaluator.evaluate(results,
     {evaluator.metricName: 'areaUnderROC'}))
print(evaluator.evaluate(results,
     {evaluator.metricName: 'areaUnderPR'}))
```

The preceding code prints out the following output:



```
0.7334857800726642
0.7071651608758281
```

Well, the model with less features certainly performed worse than the full model, but the difference was not that great. Ultimately, it is a performance trade-off between a more complex model and the less sophisticated one.

Other features of PySpark ML in action

At the beginning of this chapter, we described most of the features of the PySpark ML library. In this section, we will provide examples of how to use some of the `Transformers` and `Estimators`.

Feature extraction

We have used quite a few models from this submodule of PySpark. In this section, we'll show you how to use the most useful ones (in our opinion).

NLP - related feature extractors

As described earlier, the `NGram` model takes a list of tokenized text and produces pairs (or n-grams) of words.

In this example, we will take an excerpt from PySpark's documentation and present how to clean up the text before passing it to the `NGram` model. Here's how our dataset looks like (abbreviated for brevity):

For the full view of how the following snippet looks like, please download the code from our GitHub repository: `https://github.com/drabastomek/learningPySpark`.

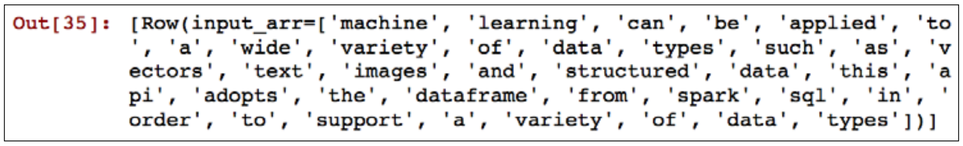
We copied these four paragraphs from the description of the DataFrame usage in `Pipelines`: `http://spark.apache.org/docs/latest/ml-pipeline.html#dataframe`.

```
text_data = spark.createDataFrame([
    ['''Machine learning can be applied to a wide variety
        of data types, such as vectors, text, images, and
        structured data. This API adopts the DataFrame from
        Spark SQL in order to support a variety of data
        types.'''],
    (...)
    ['''Columns in a DataFrame are named. The code examples
        below use names such as "text," "features," and
        "label."''']
], ['input'])
```

Each row in our single-column DataFrame is just a bunch of text. First, we need to tokenize this text. To do so we will use the `RegexTokenizer` instead of just the `Tokenizer` as we can specify the pattern(s) we want the text to be broken at:

```
tokenizer = ft.RegexTokenizer(
    inputCol='input',
    outputCol='input_arr',
    pattern='\s+|[,.\"]')
```

The pattern here splits the text on any number of spaces, but also removes commas, full stops, backslashes, and quotation marks. A single row from the output of the `tokenizer` looks similar to this:



```
Out[35]: [Row(input_arr=['machine', 'learning', 'can', 'be', 'applied', 'to
         ', 'a', 'wide', 'variety', 'of', 'data', 'types', 'such', 'as', 'v
         ectors', 'text', 'images', 'and', 'structured', 'data', 'this', 'a
         pi', 'adopts', 'the', 'dataframe', 'from', 'spark', 'sql', 'in', '
         order', 'to', 'support', 'a', 'variety', 'of', 'data', 'types'])]
```

As you can see, the `RegexTokenizer` not only splits the sentences in to words, but also normalizes the text so each word is in small-caps.

However, there is still plenty of junk in our text: words such as `be`, `a`, or `to` normally provide us with nothing useful when analyzing a text. Thus, we will remove these so called `stopwords` using nothing else other than the `StopWordsRemover(...)`:

```
stopwords = ft.StopWordsRemover(
    inputCol=tokenizer.getOutputCol(),
    outputCol='input_stop')
```

The output of the method looks as follows:

```
Out[37]: [Row(input_stop=['machine', 'learning', 'applied', 'wide', 'variet
         y', 'data', 'types', 'vectors', 'text', 'images', 'structured', 'd
         ata', 'api', 'adopts', 'dataframe', 'spark', 'sql', 'order', 'supp
         ort', 'variety', 'data', 'types'])]
```

Now we only have the useful words. So, let's build our `NGram` model and the `Pipeline`:

```
ngram = ft.NGram(n=2,
    inputCol=stopwords.getOutputCol(),
    outputCol="nGrams")
pipeline = Pipeline(stages=[tokenizer, stopwords, ngram])
```

Now that we have the `pipeline`, we follow in a very similar fashion as before:

```
data_ngram = pipeline \
    .fit(text_data) \
    .transform(text_data)
data_ngram.select('nGrams').take(1)
```

The preceding code produces the following output:

```
Out[39]: [Row(nGrams=['machine learning', 'learning applied', 'applied wide
         ', 'wide variety', 'variety data', 'data types', 'types vectors',
         'vectors text', 'text images', 'images structured', 'structured da
         ta', 'data api', 'api adopts', 'adopts dataframe', 'dataframe spar
         k', 'spark sql', 'sql order', 'order support', 'support variety',
         'variety data', 'data types'])]
```

That's it. We have got our n-grams and we can now use them in further NLP processing.

Discretizing continuous variables

Ever so often, we deal with a continuous feature that is highly non-linear and really hard to fit in our model with only one coefficient.

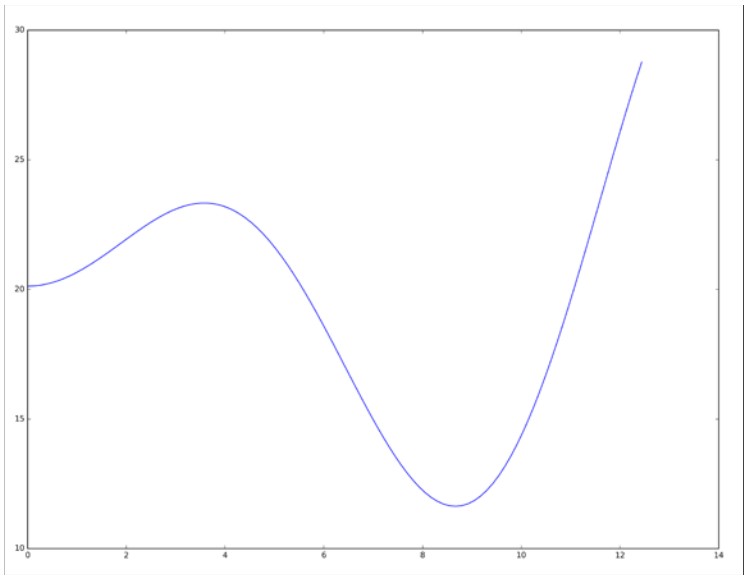
In such a situation, it might be hard to explain the relationship between such a feature and the target with just one coefficient. Sometimes, it is useful to band the values into discrete buckets.

First, let's create some fake data with the help of the following code:

```
import numpy as np
x = np.arange(0, 100)
x = x / 100.0 * np.pi * 4
y = x * np.sin(x / 1.764) + 20.1234
```

Now, we can create a DataFrame by using the following code:

```
schema = typ.StructType([
    typ.StructField('continuous_var', 
                    typ.DoubleType(), 
                    False
   )
])
data = spark.createDataFrame(
    [[float(e), ] for e in y], 
    schema=schema)
```



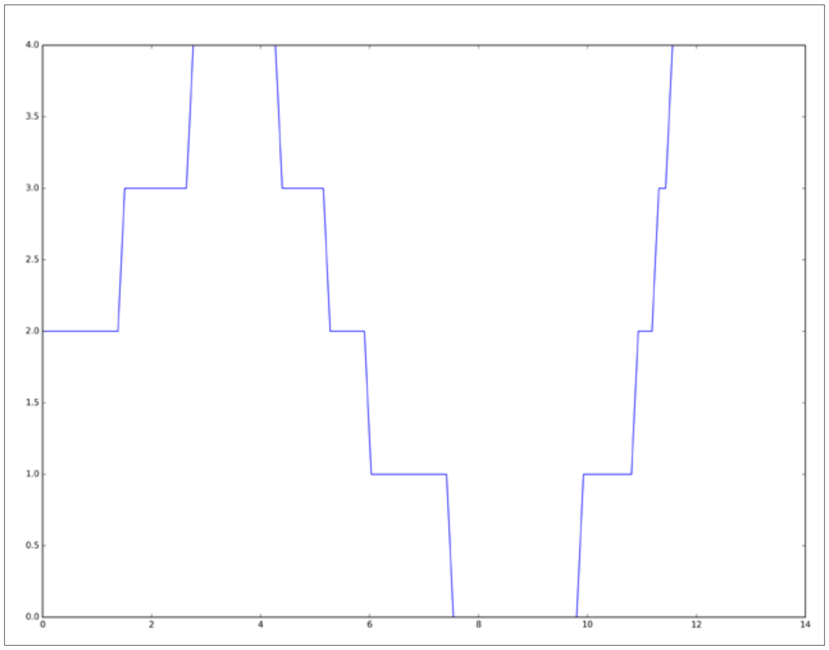
Next, we will use the `QuantileDiscretizer` model to split our continuous variable into five buckets (the `numBuckets` parameter):

```
discretizer = ft.QuantileDiscretizer(
    numBuckets=5,
    inputCol='continuous_var',
    outputCol='discretized')
```

Let's see what we have got:

```
data_discretized = discretizer.fit(data).transform(data)
```

Our function now looks as follows:



We can now treat this variable as categorical and use the `OneHotEncoder` to encode it for future use.

Standardizing continuous variables

Standardizing continuous variables helps not only in better understanding the relationships between the features (as interpreting the coefficients becomes easier), but it also aids computational efficiency and protects from running into some numerical traps. Here's how you do it with PySpark ML.

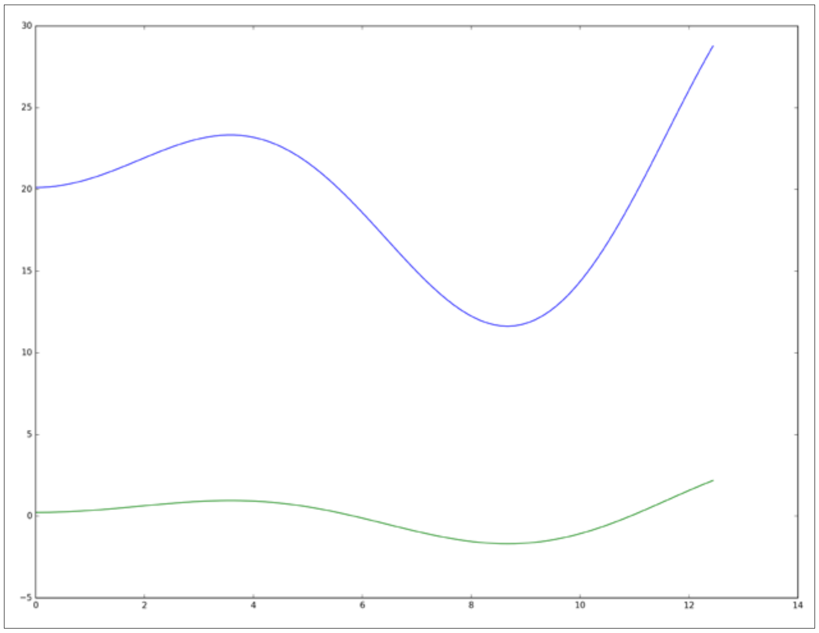
First, we need to create a vector representation of our continuous variable (as it is only a single float):

```
vectorizer = ft.VectorAssembler(
    inputCols=['continuous_var'],
    outputCol= 'continuous_vec')
```

Next, we build our `normalizer` and the `pipeline`. By setting the `withMean` and `withStd` to `True`, the method will remove the mean and scale the variance to be of unit length:

```
normalizer = ft.StandardScaler(
    inputCol=vectorizer.getOutputCol(),
    outputCol='normalized',
    withMean=True,
    withStd=True
)
pipeline = Pipeline(stages=[vectorizer, normalizer])
data_standardized = pipeline.fit(data).transform(data)
```

Here's what the transformed data would look like:



As you can see, the data now oscillates around 0 with the unit variance (the green line).

Classification

So far we have only used the `LogisticRegression` model from PySpark ML. In this section, we will use the `RandomForestClassfier` to, once again, model the chances of survival for an infant.

Before we can do that, though, we need to cast the `label` feature to `DoubleType`:

```
import pyspark.sql.functions as func
births = births.withColumn(
    'INFANT_ALIVE_AT_REPORT',
    func.col('INFANT_ALIVE_AT_REPORT').cast(typ.DoubleType())
)
births_train, births_test = births \
    .randomSplit([0.7, 0.3], seed=666)
```

Now that we have the label converted to double, we are ready to build our model. We progress in a similar fashion as before with the distinction that we will reuse the `encoder` and `featureCreator` from earlier in the chapter. The `numTrees` parameter specifies how many decision trees should be in our random forest, and the `maxDepth` parameter limits the depth of the trees:

```
classifier = cl.RandomForestClassifier(
    numTrees=5,
    maxDepth=5,
    labelCol='INFANT_ALIVE_AT_REPORT')
pipeline = Pipeline(
    stages=[
        encoder,
        featuresCreator,
        classifier])
model = pipeline.fit(births_train)
test = model.transform(births_test)
```

Let's now see how the `RandomForestClassifier` model performs compared to the `LogisticRegression`:

```
evaluator = ev.BinaryClassificationEvaluator(
    labelCol='INFANT_ALIVE_AT_REPORT')
print(evaluator.evaluate(test,
    {evaluator.metricName: "areaUnderROC"}))
print(evaluator.evaluate(test,
    {evaluator.metricName: "areaUnderPR"}))
```

We get the following results:

```
0.7736428008521183
0.7415879154340478
```

Well, as you can see, the results are better than the logistic regression model by roughly 3 percentage points. Let's test how well would a model with one tree do:

```
classifier = cl.DecisionTreeClassifier(
    maxDepth=5,
    labelCol='INFANT_ALIVE_AT_REPORT')
pipeline = Pipeline(stages=[
    encoder,
    featuresCreator,
    classifier])
model = pipeline.fit(births_train)
test = model.transform(births_test)
evaluator = ev.BinaryClassificationEvaluator(
    labelCol='INFANT_ALIVE_AT_REPORT')
print(evaluator.evaluate(test,
    {evaluator.metricName: "areaUnderROC"}))
print(evaluator.evaluate(test,
    {evaluator.metricName: "areaUnderPR"}))
```

The preceding code gives us the following:

```
0.7582781726635287
0.7787580540118526
```

Not bad at all! It actually performed better than the random forest model in terms of the precision-recall relationship and only slightly worse in terms of the area under the ROC. We just might have found a winner!

Clustering

Clustering is another big part of machine learning: quite often, in the real world, we do not have the luxury of having the target feature, so we need to revert to an unsupervised learning paradigm, where we try to uncover patterns in the data.

Finding clusters in the births dataset

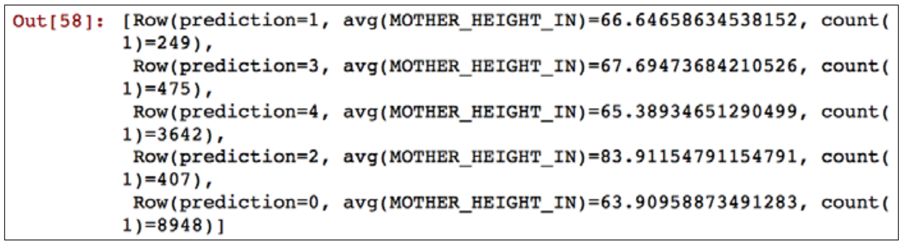
In this example, we will use the `k-means` model to find similarities in the births data:

```
import pyspark.ml.clustering as clus
kmeans = clus.KMeans(k = 5,
    featuresCol='features')
pipeline = Pipeline(stages=[
        assembler,
        featuresCreator,
        kmeans]
)
model = pipeline.fit(births_train)
```

Having estimated the model, let's see if we can find some differences between clusters:

```
test = model.transform(births_test)
test \
    .groupBy('prediction') \
    .agg({
        '*': 'count',
        'MOTHER_HEIGHT_IN': 'avg'
    }).collect()
```

The preceding code produces the following output:



```
Out[58]: [Row(prediction=1, avg(MOTHER_HEIGHT_IN)=66.64658634538152, count(
         1)=249),
          Row(prediction=3, avg(MOTHER_HEIGHT_IN)=67.69473684210526, count(
         1)=475),
          Row(prediction=4, avg(MOTHER_HEIGHT_IN)=65.38934651290499, count(
         1)=3642),
          Row(prediction=2, avg(MOTHER_HEIGHT_IN)=83.91154791154791, count(
         1)=407),
          Row(prediction=0, avg(MOTHER_HEIGHT_IN)=63.90958873491283, count(
         1)=8948)]
```

Well, the `MOTHER_HEIGHT_IN` is significantly different in cluster 2. Going through the results (which we will not do here for obvious reasons) would most likely uncover more differences and allow us to understand the data better.

Topic mining

Clustering models are not limited to numeric data only. In the field of NLP, problems such as topic extraction rely on clustering to detect documents with similar topics. We will go through such an example.

First, let's create our dataset. The data is formed from randomly selected paragraphs found on the Internet: three of them deal with topics of nature and national parks, the remaining three cover technology.

The code snippet is abbreviated again, for obvious reasons. Refer to the source file on GitHub for full representation.

```
text_data = spark.createDataFrame([
    ['''To make a computer do anything, you have to write a
    computer program. To write a computer program, you have
    to tell the computer, step by step, exactly what you want
    it to do. The computer then "executes" the program,
    following each step mechanically, to accomplish the end
    goal. When you are telling the computer what to do, you
    also get to choose how it's going to do it. That's where
    computer algorithms come in. The algorithm is the basic
    technique used to get the job done. Let's follow an
    example to help get an understanding of the algorithm
    concept.'''],
    (...),
    ['''Australia has over 500 national parks. Over 28
    million hectares of land is designated as national
    parkland, accounting for almost four per cent of
    Australia's land areas. In addition, a further six per
    cent of Australia is protected and includes state
    forests, nature parks and conservation reserves.National
    parks are usually large areas of land that are protected
    because they have unspoilt landscapes and a diverse
    number of native plants and animals. This means that
    commercial activities such as farming are prohibited and
    human activity is strictly monitored.''']
], ['documents'])
```

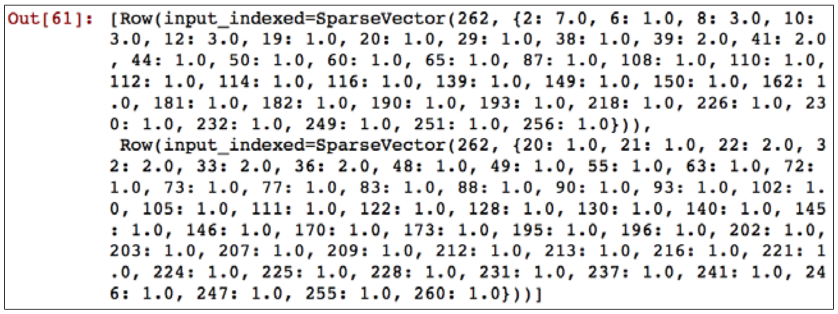
First, we will once again use the `RegexTokenizer` and the `StopWordsRemover` models:

```
tokenizer = ft.RegexTokenizer(
    inputCol='documents',
    outputCol='input_arr',
    pattern='\s+|[,.\"]')
stopwords = ft.StopWordsRemover(
    inputCol=tokenizer.getOutputCol(),
    outputCol='input_stop')
```

Next in our pipeline is the `CountVectorizer`: a model that counts words in a document and returns a vector of counts. The length of the vector is equal to the total number of distinct words in all the documents, which can be seen in the following snippet:

```
stringIndexer = ft.CountVectorizer(
    inputCol=stopwords.getOutputCol(),
    outputCol="input_indexed")
tokenized = stopwords \
    .transform(
        tokenizer\
            .transform(text_data)
    )

stringIndexer \
    .fit(tokenized)\
    .transform(tokenized)\
    .select('input_indexed')\
    .take(2)
```

The preceding code will produce the following output:



```
Out[61]: [Row(input_indexed=SparseVector(262, {2: 7.0, 6: 1.0, 8: 3.0, 10:
         3.0, 12: 3.0, 19: 1.0, 20: 1.0, 29: 1.0, 38: 1.0, 39: 2.0, 41: 2.0
         , 44: 1.0, 50: 1.0, 60: 1.0, 65: 1.0, 87: 1.0, 108: 1.0, 110: 1.0,
         112: 1.0, 114: 1.0, 116: 1.0, 139: 1.0, 149: 1.0, 150: 1.0, 162: 1
         .0, 181: 1.0, 182: 1.0, 190: 1.0, 193: 1.0, 218: 1.0, 226: 1.0, 23
         0: 1.0, 232: 1.0, 249: 1.0, 251: 1.0, 256: 1.0})),
          Row(input_indexed=SparseVector(262, {20: 1.0, 21: 1.0, 22: 2.0, 3
         2: 2.0, 33: 2.0, 36: 2.0, 48: 1.0, 49: 1.0, 55: 1.0, 63: 1.0, 72:
         1.0, 73: 1.0, 77: 1.0, 83: 1.0, 88: 1.0, 90: 1.0, 93: 1.0, 102: 1.
         0, 105: 1.0, 111: 1.0, 122: 1.0, 128: 1.0, 130: 1.0, 140: 1.0, 145
         : 1.0, 146: 1.0, 170: 1.0, 173: 1.0, 195: 1.0, 196: 1.0, 202: 1.0,
         203: 1.0, 207: 1.0, 209: 1.0, 212: 1.0, 213: 1.0, 216: 1.0, 221: 1
         .0, 224: 1.0, 225: 1.0, 228: 1.0, 231: 1.0, 237: 1.0, 241: 1.0, 24
         6: 1.0, 247: 1.0, 255: 1.0, 260: 1.0}))]
```

As you can see, there are 262 distinct words in the text, and each document is now represented by a count of each word occurrence.

It's now time to start predicting the topics. For that purpose we will use the `LDA` model—the **Latent Dirichlet Allocation** model:

```
clustering = clus.LDA(k=2,
    optimizer='online',
    featuresCol=stringIndexer.getOutputCol())
```

The `k` parameter specifies how many topics we expect to see, the `optimizer` parameter can be either `'online'` or `'em'` (the latter standing for the Expectation Maximization algorithm).

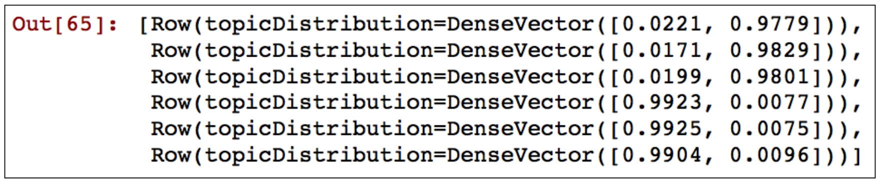
Putting these puzzles together results in, so far, the longest of our pipelines:

```
pipeline = ml.Pipeline(stages=[
        tokenizer,
        stopwords,
        stringIndexer,
        clustering]
)
```

Have we properly uncovered the topics? Well, let's see:

```
topics = pipeline \
    .fit(text_data) \
    .transform(text_data)
topics.select('topicDistribution').collect()
```

Here's what we get:



```
Out[65]: [Row(topicDistribution=DenseVector([0.0221, 0.9779])),
          Row(topicDistribution=DenseVector([0.0171, 0.9829])),
          Row(topicDistribution=DenseVector([0.0199, 0.9801])),
          Row(topicDistribution=DenseVector([0.9923, 0.0077])),
          Row(topicDistribution=DenseVector([0.9925, 0.0075])),
          Row(topicDistribution=DenseVector([0.9904, 0.0096]))]
```

Looks like our method discovered all the topics properly! Do not get used to seeing such good results though: sadly, real world data is seldom that kind.

Regression

We could not finish a chapter on a machine learning library without building a regression model.

In this section, we will try to predict the `MOTHER_WEIGHT_GAIN` given some of the features described here; these are contained in the features listed here:

```
features = ['MOTHER_AGE_YEARS','MOTHER_HEIGHT_IN',
            'MOTHER_PRE_WEIGHT','DIABETES_PRE',
            'DIABETES_GEST','HYP_TENS_PRE',
            'HYP_TENS_GEST', 'PREV_BIRTH_PRETERM',
            'CIG_BEFORE','CIG_1_TRI', 'CIG_2_TRI',
            'CIG_3_TRI'
           ]
```