Advanced Analytics with Spark

In this practical book, four Cloudera data scientists present a set of self-contained patterns for performing large-scale data analysis with Spark. The authors bring Spark, statistical methods, and real-world data sets together to teach you how to approach analytics problems by example.

You’ll start with an introduction to Spark and its ecosystem, and then dive into patterns that apply common techniques—classification, collaborative filtering, and anomaly detection, among others—to fields such as genomics, security, and finance. If you have an entry-level understanding of machine learning and statistics, and you program in Java, Python, or Scala, you’ll find these patterns useful for working on your own data applications.

Patterns include:

- Recommending music and the Audioscrobbler data set
- Predicting forest cover with decision trees
- Anomaly detection in network traffic with K-means clustering
- Understanding Wikipedia with Latent Semantic Analysis
- Analyzing co-occurrence networks with GraphX
- Geospatial and temporal data analysis on the New York City Taxi Trips data
- Estimating financial risk through Monte Carlo simulation
- Analyzing genomics data and the BDG project
- Analyzing neuroimaging data with PySpark and Thunder

Sandy Ryza is a Senior Data Scientist at Cloudera and active contributor to the Apache Spark project.

Uri Laserson is a Senior Data Scientist at Cloudera, where he focuses on Python in the Hadoop ecosystem.

Sean Owen is Director of Data Science for EMEA at Cloudera, and a committer for Apache Spark.

Josh Wills is Senior Director of Data Science at Cloudera and founder of the Apache Crunch project.
In this practical book, four Cloudera data scientists present a set of self-contained patterns for performing large-scale data analysis with Spark. The authors bring Spark, statistical methods, and real-world data sets together to teach you how to approach analytics problems by example.

You’ll start with an introduction to Spark and its ecosystem, and then dive into patterns that apply common techniques—classification, collaborative filtering, and anomaly detection, among others—to fields such as genomics, security, and finance. If you have an entry-level understanding of machine learning and statistics, and you program in Java, Python, or Scala, you’ll find these patterns useful for working on your own data applications.

Patterns include:
- Recommending music and the Audioscrobbler data set
- Predicting forest cover with decision trees
- Anomaly detection in network traffic with K-means clustering
- Understanding Wikipedia with Latent Semantic Analysis
- Analyzing co-occurrence networks with GraphX
- Geospatial and temporal data analysis on the New York City Taxi Trips data
- Estimating financial risk through Monte Carlo simulation
- Analyzing genomics data and the BDG project
- Analyzing neuroimaging data with PySpark and Thunder

Sandy Ryza is a Senior Data Scientist at Cloudera and active contributor to the Apache Spark project.
Uri Laserson is a Senior Data Scientist at Cloudera, where he focuses on Python in the Hadoop ecosystem.
Sean Owen is Director of Data Science for EMEA at Cloudera, and a committer for Apache Spark.
Josh Wills is Senior Director of Data Science at Cloudera and founder of the Apache Crunch project.
Want to read more?

You can buy this book at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code OPC10

All orders over $29.95 qualify for free shipping within the US.

It’s also available at your favorite book retailer, including the iBookstore, the Android Marketplace, and Amazon.com.
Advanced Analytics with Spark

Sandy Ryza, Uri Laserson, Sean Owen, and Josh Wills
# Table of Contents

Foreword ................................................................. vii

Preface ................................................................. ix

1. **Analyzing Big Data** .............................................. 1
   - The Challenges of Data Science ................................. 3
   - Introducing Apache Spark ....................................... 4
   - About This Book .................................................. 6

2. **Introduction to Data Analysis with Scala and Spark** ................. 9
   - Scala for Data Scientists ....................................... 10
   - The Spark Programming Model ................................. 11
   - Record Linkage .................................................... 11
   - Getting Started: The Spark Shell and SparkContext ........ 13
   - Bringing Data from the Cluster to the Client .............. 18
   - Shipping Code from the Client to the Cluster .............. 22
   - Structuring Data with Tuples and Case Classes ............ 23
   - Aggregations ...................................................... 28
   - Creating Histograms ............................................. 29
   - Summary Statistics for Continuous Variables .............. 30
   - Creating Reusable Code for Computing Summary Statistics 31
   - Simple Variable Selection and Scoring ...................... 36
   - Where to Go from Here ........................................... 37

3. **Recommending Music and the Audioscrobbler Data Set** ............... 39
   - Data Set ................................................................ 40
   - The Alternating Least Squares Recommender Algorithm .... 41
   - Preparing the Data ............................................... 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building a First Model</td>
<td>46</td>
</tr>
<tr>
<td>Spot Checking Recommendations</td>
<td>48</td>
</tr>
<tr>
<td>Evaluating Recommendation Quality</td>
<td>50</td>
</tr>
<tr>
<td>Computing AUC</td>
<td>51</td>
</tr>
<tr>
<td>Hyperparameter Selection</td>
<td>53</td>
</tr>
<tr>
<td>Making Recommendations</td>
<td>55</td>
</tr>
<tr>
<td>Where to Go from Here</td>
<td>56</td>
</tr>
<tr>
<td>4. Predicting Forest Cover with Decision Trees</td>
<td>59</td>
</tr>
<tr>
<td>Fast Forward to Regression</td>
<td>59</td>
</tr>
<tr>
<td>Vectors and Features</td>
<td>60</td>
</tr>
<tr>
<td>Training Examples</td>
<td>61</td>
</tr>
<tr>
<td>Decision Trees and Forests</td>
<td>62</td>
</tr>
<tr>
<td>Covtype Data Set</td>
<td>65</td>
</tr>
<tr>
<td>Preparing the Data</td>
<td>66</td>
</tr>
<tr>
<td>A First Decision Tree</td>
<td>67</td>
</tr>
<tr>
<td>Decision Tree Hyperparameters</td>
<td>71</td>
</tr>
<tr>
<td>Tuning Decision Trees</td>
<td>73</td>
</tr>
<tr>
<td>Categorical Features Revisited</td>
<td>75</td>
</tr>
<tr>
<td>Random Decision Forests</td>
<td>77</td>
</tr>
<tr>
<td>Making Predictions</td>
<td>79</td>
</tr>
<tr>
<td>Where to Go from Here</td>
<td>79</td>
</tr>
<tr>
<td>5. Anomaly Detection in Network Traffic with K-means Clustering</td>
<td>81</td>
</tr>
<tr>
<td>Anomaly Detection</td>
<td>82</td>
</tr>
<tr>
<td>K-means Clustering</td>
<td>82</td>
</tr>
<tr>
<td>Network Intrusion</td>
<td>83</td>
</tr>
<tr>
<td>KDD Cup 1999 Data Set</td>
<td>84</td>
</tr>
<tr>
<td>A First Take on Clustering</td>
<td>85</td>
</tr>
<tr>
<td>Choosing k</td>
<td>87</td>
</tr>
<tr>
<td>Visualization in R</td>
<td>89</td>
</tr>
<tr>
<td>Feature Normalization</td>
<td>91</td>
</tr>
<tr>
<td>Categorical Variables</td>
<td>94</td>
</tr>
<tr>
<td>Using Labels with Entropy</td>
<td>95</td>
</tr>
<tr>
<td>Clustering in Action</td>
<td>96</td>
</tr>
<tr>
<td>Where to Go from Here</td>
<td>97</td>
</tr>
<tr>
<td>The Term-Document Matrix</td>
<td>100</td>
</tr>
<tr>
<td>Getting the Data</td>
<td>102</td>
</tr>
<tr>
<td>Parsing and Preparing the Data</td>
<td>102</td>
</tr>
<tr>
<td>Lemmatization</td>
<td>104</td>
</tr>
</tbody>
</table>
## Computing the TF-IDFs

105

## Singular Value Decomposition

107

## Finding Important Concepts

109

## Querying and Scoring with the Low-Dimensional Representation

112

## Term-Term Relevance

113

## Document-Document Relevance

115

## Term-Document Relevance

116

## Multiple-Term Queries

117

## Where to Go from Here

119

### 7. Analyzing Co-occurrence Networks with GraphX

121

- The MEDLINE Citation Index: A Network Analysis
  122
- Getting the Data
  123
- Parsing XML Documents with Scala’s XML Library
  125
- Analyzing the MeSH Major Topics and Their Co-occurrences
  127
- Constructing a Co-occurrence Network with GraphX
  129
- Understanding the Structure of Networks
  132
  - Connected Components
    132
  - Degree Distribution
    135
- Filtering Out Noisy Edges
  138
  - Processing EdgeTriplets
    139
  - Analyzing the Filtered Graph
    140
- Small-World Networks
  142
  - Cliques and Clustering Coefficients
    143
  - Computing Average Path Length with Pregel
    144
- Where to Go from Here
  149

### 8. Geospatial and Temporal Data Analysis on the New York City Taxi Trip Data

151

- Getting the Data
  152
- Working with Temporal and Geospatial Data in Spark
  153
- Temporal Data with JodaTime and NScalaTime
  153
- Geospatial Data with the Esri Geometry API and Spray
  155
  - Exploring the Esri Geometry API
    155
  - Intro to GeoJSON
    157
- Preparing the New York City Taxi Trip Data
  159
  - Handling Invalid Records at Scale
    160
  - Geospatial Analysis
    164
- Sessionization in Spark
  167
  - Building Sessions: Secondary Sorts in Spark
    168
- Where to Go from Here
  171
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Estimating Financial Risk through Monte Carlo Simulation</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>Terminology</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>Methods for Calculating VaR</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Variance-Covariance</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Historical Simulation</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Monte Carlo Simulation</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>Our Model</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td>Getting the Data</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>Preprocessing</td>
<td>178</td>
</tr>
<tr>
<td></td>
<td>Determining the Factor Weights</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>Sampling</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>The Multivariate Normal Distribution</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>Running the Trials</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td>Visualizing the Distribution of Returns</td>
<td>189</td>
</tr>
<tr>
<td></td>
<td>Evaluating Our Results</td>
<td>190</td>
</tr>
<tr>
<td></td>
<td>Where to Go from Here</td>
<td>192</td>
</tr>
<tr>
<td>10</td>
<td>Analyzing Genomics Data and the BDG Project</td>
<td>195</td>
</tr>
<tr>
<td></td>
<td>Decoupling Storage from Modeling</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>Ingesting Genomics Data with the ADAM CLI</td>
<td>198</td>
</tr>
<tr>
<td></td>
<td>Parquet Format and Columnar Storage</td>
<td>204</td>
</tr>
<tr>
<td></td>
<td>Predicting Transcription Factor Binding Sites from ENCODE Data</td>
<td>206</td>
</tr>
<tr>
<td></td>
<td>Querying Genotypes from the 1000 Genomes Project</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>Where to Go from Here</td>
<td>214</td>
</tr>
<tr>
<td>11</td>
<td>Analyzing Neuroimaging Data with PySpark and Thunder</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>Overview of PySpark</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>PySpark Internals</td>
<td>219</td>
</tr>
<tr>
<td></td>
<td>Overview and Installation of the Thunder Library</td>
<td>221</td>
</tr>
<tr>
<td></td>
<td>Loading Data with Thunder</td>
<td>222</td>
</tr>
<tr>
<td></td>
<td>Thunder Core Data Types</td>
<td>229</td>
</tr>
<tr>
<td></td>
<td>Categorizing Neuron Types with Thunder</td>
<td>231</td>
</tr>
<tr>
<td></td>
<td>Where to Go from Here</td>
<td>236</td>
</tr>
<tr>
<td>A</td>
<td>Deeper into Spark</td>
<td>237</td>
</tr>
<tr>
<td>B</td>
<td>Upcoming MLlib Pipelines API</td>
<td>247</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>253</td>
</tr>
</tbody>
</table>
CHAPTER 1

Analyzing Big Data

Sandy Ryza

[Data applications] are like sausages. It is better not to see them being made.
—Otto von Bismarck

• Build a model to detect credit card fraud using thousands of features and billions of transactions.
• Intelligently recommend millions of products to millions of users.
• Estimate financial risk through simulations of portfolios including millions of instruments.
• Easily manipulate data from thousands of human genomes to detect genetic associations with disease.

These are tasks that simply could not be accomplished 5 or 10 years ago. When people say that we live in an age of “big data,” they mean that we have tools for collecting, storing, and processing information at a scale previously unheard of. Sitting behind these capabilities is an ecosystem of open source software that can leverage clusters of commodity computers to chug through massive amounts of data. Distributed systems like Apache Hadoop have found their way into the mainstream and have seen widespread deployment at organizations in nearly every field.

But just as a chisel and a block of stone do not make a statue, there is a gap between having access to these tools and all this data, and doing something useful with it. This is where “data science” comes in. As sculpture is the practice of turning tools and raw material into something relevant to nonsculptors, data science is the practice of turning tools and raw data into something that nondata scientists might care about.

Often, “doing something useful” means placing a schema over it and using SQL to answer questions like “of the gazillion users who made it to the third page in our
registration process, how many are over 25?” The field of how to structure a data
warehouse and organize information to make answering these kinds of questions
easy is a rich one, but we will mostly avoid its intricacies in this book.

Sometimes, “doing something useful” takes a little extra. SQL still may be core to the
approach, but to work around idiosyncrasies in the data or perform complex analysis,
we need a programming paradigm that’s a little bit more flexible and a little closer to
the ground, and with richer functionality in areas like machine learning and statistics.
These are the kinds of analyses we are going to talk about in this book.

For a long time, open source frameworks like R, the PyData stack, and Octave have
made rapid analysis and model building viable over small data sets. With fewer than
10 lines of code, we can throw together a machine learning model on half a data set
and use it to predict labels on the other half. With a little more effort, we can impute
missing data, experiment with a few models to find the best one, or use the results of
a model as inputs to fit another. What should an equivalent process look like that can
leverage clusters of computers to achieve the same outcomes on huge data sets?

The right approach might be to simply extend these frameworks to run on multiple
machines, to retain their programming models and rewrite their guts to play well in
distributed settings. However, the challenges of distributed computing require us to
rethink many of the basic assumptions that we rely on in single-node systems. For
example, because data must be partitioned across many nodes on a cluster, algorithms
that have wide data dependencies will suffer from the fact that network transfer rates
are orders of magnitude slower than memory accesses. As the number of machines
working on a problem increases, the probability of a failure increases. These facts
require a programming paradigm that is sensitive to the characteristics of the under‐
lying system: one that discourages poor choices and makes it easy to write code that
will execute in a highly parallel manner.

Of course, single-machine tools like PyData and R that have come to recent promi‐
nence in the software community are not the only tools used for data analysis. Sci‐
tific fields like genomics that deal with large data sets have been leveraging parallel
computing frameworks for decades. Most people processing data in these fields today
are familiar with a cluster-computing environment called HPC (high-performance
computing). Where the difficulties with PyData and R lie in their inability to scale,
the difficulties with HPC lie in its relatively low level of abstraction and difficulty of
use. For example, to process a large file full of DNA sequencing reads in parallel, we
must manually split it up into smaller files and submit a job for each of those files to
the cluster scheduler. If some of these fail, the user must detect the failure and take
care of manually resubmitting them. If the analysis requires all-to-all operations like
sorting the entire data set, the large data set must be streamed through a single node,
or the scientist must resort to lower-level distributed frameworks like MPI, which are
difficult to program without extensive knowledge of C and distributed/networked
systems. Tools written for HPC environments often fail to decouple the in-memory data models from the lower-level storage models. For example, many tools only know how to read data from a POSIX filesystem in a single stream, making it difficult to make tools naturally parallelize, or to use other storage backends, like databases. Recent systems in the Hadoop ecosystem provide abstractions that allow users to treat a cluster of computers more like a single computer—to automatically split up files and distribute storage over many machines, to automatically divide work into smaller tasks and execute them in a distributed manner, and to automatically recover from failures. The Hadoop ecosystem can automate a lot of the hassle of working with large data sets, and is far cheaper than HPC.

The Challenges of Data Science

A few hard truths come up so often in the practice of data science that evangelizing these truths has become a large role of the data science team at Cloudera. For a system that seeks to enable complex analytics on huge data to be successful, it needs to be informed by, or at least not conflict with, these truths.

First, the vast majority of work that goes into conducting successful analyses lies in preprocessing data. Data is messy, and cleansing, munging, fusing, mushing, and many other verbs are prerequisites to doing anything useful with it. Large data sets in particular, because they are not amenable to direct examination by humans, can require computational methods to even discover what preprocessing steps are required. Even when it comes time to optimize model performance, a typical data pipeline requires spending far more time in feature engineering and selection than in choosing and writing algorithms.

For example, when building a model that attempts to detect fraudulent purchases on a website, the data scientist must choose from a wide variety of potential features: any fields that users are required to fill out, IP location info, login times, and click logs as users navigate the site. Each of these comes with its own challenges in converting to vectors fit for machine learning algorithms. A system needs to support more flexible transformations than turning a 2D array of doubles into a mathematical model.

Second, iteration is a fundamental part of the data science. Modeling and analysis typically require multiple passes over the same data. One aspect of this lies within machine learning algorithms and statistical procedures. Popular optimization procedures like stochastic gradient descent and expectation maximization involve repeated scans over their inputs to reach convergence. Iteration also matters within the data scientist's own workflow. When data scientists are initially investigating and trying to get a feel for a data set, usually the results of a query inform the next query that should run. When building models, data scientists do not try to get it right in one try. Choosing the right features, picking the right algorithms, running the right significance tests, and finding the right hyperparameters all require experimentation. A
framework that requires reading the same data set from disk each time it is accessed adds delay that can slow down the process of exploration and limit the number of things we get to try.

Third, the task isn't over when a well-performing model has been built. If the point of data science is making data useful to nondata scientists, then a model stored as a list of regression weights in a text file on the data scientist’s computer has not really accomplished this goal. Uses of data recommendation engines and real-time fraud detection systems culminate in data applications. In these, models become part of a production service and may need to be rebuilt periodically or even in real time.

For these situations, it is helpful to make a distinction between analytics in the lab and analytics in the factory. In the lab, data scientists engage in exploratory analytics. They try to understand the nature of the data they are working with. They visualize it and test wild theories. They experiment with different classes of features and auxiliary sources they can use to augment it. They cast a wide net of algorithms in the hopes that one or two will work. In the factory, in building a data application, data scientists engage in operational analytics. They package their models into services that can inform real-world decisions. They track their models’ performance over time and obsess about how they can make small tweaks to squeeze out another percentage point of accuracy. They care about SLAs and uptime. Historically, exploratory analytics typically occurs in languages like R, and when it comes time to build production applications, the data pipelines are rewritten entirely in Java or C++.

Of course, everybody could save time if the original modeling code could be actually used in the app for which it is written, but languages like R are slow and lack integration with most planes of the production infrastructure stack, and languages like Java and C++ are just poor tools for exploratory analytics. They lack Read-Evaluate-Print Loop (REPL) environments for playing with data interactively and require large amounts of code to express simple transformations. A framework that makes modeling easy but is also a good fit for production systems is a huge win.

**Introducing Apache Spark**

Enter Apache Spark, an open source framework that combines an engine for distributing programs across clusters of machines with an elegant model for writing programs atop it. Spark, which originated at the UC Berkeley AMPLab and has since been contributed to the Apache Software Foundation, is arguably the first open source software that makes distributed programming truly accessible to data scientists.

One illuminating way to understand Spark is in terms of its advances over its predecessor, MapReduce. MapReduce revolutionized computation over huge data sets by offering a simple model for writing programs that could execute in parallel across
hundreds to thousands of machines. The MapReduce engine achieves near linear scalability—as the data size increases, we can throw more computers at it and see jobs complete in the same amount of time—and is resilient to the fact that failures that occur rarely on a single machine occur all the time on clusters of thousands. It breaks up work into small tasks and can gracefully accommodate task failures without compromising the job to which they belong.

Spark maintains MapReduce’s linear scalability and fault tolerance, but extends it in three important ways. First, rather than relying on a rigid map-then-reduce format, its engine can execute a more general directed acyclic graph (DAG) of operators. This means that, in situations where MapReduce must write out intermediate results to the distributed filesystem, Spark can pass them directly to the next step in the pipeline. In this way, it is similar to Dryad, a descendant of MapReduce that originated at Microsoft Research. Second, it complements this capability with a rich set of transformations that enable users to express computation more naturally. It has a strong developer focus and streamlined API that can represent complex pipelines in a few lines of code.

Third, Spark extends its predecessors with in-memory processing. Its Resilient Distributed Dataset (RDD) abstraction enables developers to materialize any point in a processing pipeline into memory across the cluster, meaning that future steps that want to deal with the same data set need not recompute it or reload it from disk. This capability opens up use cases that distributed processing engines could not previously approach. Spark is well suited for highly iterative algorithms that require multiple passes over a data set, as well as reactive applications that quickly respond to user queries by scanning large in-memory data sets.

Perhaps most importantly, Spark fits well with the aforementioned hard truths of data science, acknowledging that the biggest bottleneck in building data applications is not CPU, disk, or network, but analyst productivity. It perhaps cannot be overstated how much collapsing the full pipeline, from preprocessing to model evaluation, into a single programming environment can speed up development. By packaging an expressive programming model with a set of analytic libraries under a REPL, it avoids the round trips to IDEs required by frameworks like MapReduce and the challenges of subsampling and moving data back and forth from HDFS required by frameworks like R. The more quickly analysts can experiment with their data, the higher likelihood they have of doing something useful with it.

With respect to the pertinence of munging and ETL, Spark strives to be something closer to the Python of big data than the Matlab of big data. As a general-purpose computation engine, its core APIs provide a strong foundation for data transformation independent of any functionality in statistics, machine learning, or matrix algebra. Its Scala and Python APIs allow programming in expressive general-purpose languages, as well as access to existing libraries.
Spark’s in-memory caching makes it ideal for iteration both at the micro and macro level. Machine learning algorithms that make multiple passes over their training set can cache it in memory. When exploring and getting a feel for a data set, data scientists can keep it in memory while they run queries, and easily cache transformed versions of it as well without suffering a trip to disk.

Last, Spark spans the gap between systems designed for exploratory analytics and systems designed for operational analytics. It is often quoted that a data scientist is someone who is better at engineering than most statisticians and better at statistics than most engineers. At the very least, Spark is better at being an operational system than most exploratory systems and better for data exploration than the technologies commonly used in operational systems. It is built for performance and reliability from the ground up. Sitting atop the JVM, it can take advantage of many of the operational and debugging tools built for the Java stack.

Spark boasts strong integration with the variety of tools in the Hadoop ecosystem. It can read and write data in all of the data formats supported by MapReduce, allowing it to interact with the formats commonly used to store data on Hadoop like Avro and Parquet (and good old CSV). It can read from and write to NoSQL databases like HBase and Cassandra. Its stream processing library, Spark Streaming, can ingest data continuously from systems like Flume and Kafka. Its SQL library, SparkSQL, can interact with the Hive Metastore, and a project that is in progress at the time of this writing seeks to enable Spark to be used as an underlying execution engine for Hive, as an alternative to MapReduce. It can run inside YARN, Hadoop’s scheduler and resource manager, allowing it to share cluster resources dynamically and to be managed with the same policies as other processing engines like MapReduce and Impala.

Of course, Spark isn’t all roses and petunias. While its core engine has progressed in maturity even during the span of this book being written, it is still young compared to MapReduce and hasn’t yet surpassed it as the workhorse of batch processing. Its specialized subcomponents for stream processing, SQL, machine learning, and graph processing lie at different stages of maturity and are undergoing large API upgrades. For example, MLlib’s pipelines and transformer API model is in progress while this book is being written. Its statistics and modeling functionality comes nowhere near that of single machine languages like R. Its SQL functionality is rich, but still lags far behind that of Hive.

**About This Book**

The rest of this book is not going to be about Spark’s merits and disadvantages. There are a few other things that it will not be either. It will introduce the Spark programming model and Scala basics, but it will not attempt to be a Spark reference or provide a comprehensive guide to all its nooks and crannies. It will not try to be a
machine learning, statistics, or linear algebra reference, although many of the chapters will provide some background on these before using them.

Instead, it will try to help the reader get a feel for what it’s like to use Spark for complex analytics on large data sets. It will cover the entire pipeline: not just building and evaluating models, but cleansing, preprocessing, and exploring data, with attention paid to turning results into production applications. We believe that the best way to teach this is by example, so, after a quick chapter describing Spark and its ecosystem, the rest of the chapters will be self-contained illustrations of what it looks like to use Spark for analyzing data from different domains.

When possible, we will attempt not to just provide a “solution,” but to demonstrate the full data science workflow, with all of its iterations, dead ends, and restarts. This book will be useful for getting more comfortable with Scala, more comfortable with Spark, and more comfortable with machine learning and data analysis. However, these are in service of a larger goal, and we hope that most of all, this book will teach you how to approach tasks like those described at the beginning of this chapter. Each chapter, in about 20 measly pages, will try to get as close as possible to demonstrating how to build one of these pieces of data applications.
O’Reilly ebooks.
Your bookshelf on your devices.

When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we’ve updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O’Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.