Community Experience Distilled

Mastering Scala Machine Learning

Advance your skills in efficient data analysis and data processing using the powerful tools of Scala, Spark, and Hadoop

Alex Kozlov

[PACKT] PUBLISHING open source* community experience distilled

Mastering Scala Machine Learning

Advance your skills in efficient data analysis and data processing using the powerful tools of Scala, Spark, and Hadoop

Alex Kozlov

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Mastering Scala Machine Learning

First published: June 2016

Production reference: 1220616

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

ISBN 978-1-78588-088-9

www.packtpub.com

Credits

Author
Alex Kozlov

Reviewer
Rok Kralj

Commissioning Editor
Dipika Gaonkar

Acquisition Editor
Kirk D'costa

Content Development Editor
Samantha Gonsalves

Technical Editor
Suwarna Patil

Copy Editor
Vibha Shukla

Project Coordinator
Sanchita Mandal

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Disha Haria

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Alex Kozlov is a multidisciplinary big data scientist. He came to Silicon Valley in 1991, got his Ph.D. from Stanford University under the supervision of Prof. Daphne Koller and Prof. John Hennessy in 1998, and has been around a few computer and data management companies since. His latest stint was with Cloudera, the leader in Hadoop, where he was one of the early employees and ended up heading the solution architects group on the West Coast. Before that, he spent time with an online advertising company, Turn, Inc.; and before that, he had the privilege to work with HP Labs researchers at HP Inc., and on data mining software at SGI, Inc. Currently, Alexander is the chief solutions architect at an enterprise security startup, E8 Security, where he came to understand the intricacies of catching bad guys in the Internet universe.

On the non-professional side, Alexander lives in Sunnyvale, CA, together with his beautiful wife, Oxana, and other important family members, including three daughters, Lana, Nika, and Anna, and a cat and dog. His family also included a hamster and a fish at one point.

Alex is an active participant in Silicon Valley technology groups and meetups, and although he is not an official committer of any open source projects, he definitely contributed to many of them in the form of code or discussions. Alexander is an active coder and publishes his open source code at `https://github.com/alexvk`. Other information can be looked up on his LinkedIn page at `https://www.linkedin.com/in/alexvk`.

Acknowlegement

I had a few chances to write a book in the past, but when Packt called me shortly before my 50th birthday, I agreed almost immediately. Scala? Machine learning? Big data? What could be a worse combination of poorly understood and intensely marketed topics? What followed was eight months of sleep deprived existence, putting my ideas on paper—computer keyboard, actually—during which I was able to experimentally find out that my body needs at least three hours of sleep each night and a larger break once in a while. As a whole, the experience was totally worth it. I really appreciate the help of everyone around me, first of all of my family, who had to deal with a lot of sleepless nights and my temporary lack of attention.

I would like to thank my wife for putting up with a lot of extra load and late night writing sessions. I know it's been very hard. I also give deep thanks to my editors, specifically Samantha Gonsalves, who not only nagged me from time to time to keep me on schedule, but also gave very sound advice and put up with my procrastination. Not least, I am very grateful to my colleagues who filled in for me during some very critical stages of E8 Security product releases—we did go through the GA, and at least a couple of releases during this time. A lot of ideas percolated into the E8 product. Particularly, I would like to thank Jeongho Park, Christophe Briguet, Mahendra Kutare, Srinivas Doddi, and Ravi Devireddy. I am grateful to all my Cloudera colleagues for feedback and discussions, specifically Josh Patterson, Josh Wills, Omer Trajman, Eric Sammer, Don Brown, Phillip Zeyliger, Jonathan Hsieh, and many others. Last, but not least, I would like to thank my Ph.D. mentors Walter A. Harrison, Jaswinder Pal Singh, John Hennessy, and Daphne Koller for bringing me into the world of technology and innovation.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface

This book is about machine learning, the functional approach to programming with Scala being the focus, and big data with Spark being the target. When I was offered to write the book about nine months ago, my first reaction was that, while each of the mentioned subjects have been thoroughly investigated and written about, I've definitely taken part in enough discussions to know that combining any pair of them presents challenges, not to mention combining all three of them in one book. The challenge piqued my interest, and the result is this book. Not every chapter is as smooth as I wished it to be, but in the world where technology makes huge strides every day, this is probably expected. I do have a real job and writing is only one way to express my ideas.

Let's start with machine learning. Machine learning went through a head-spinning transformation; it was an offspring of AI and statistics somewhere in the 1990s and later gave birth to data science in or slightly before 2010. There are many definitions of data science, but the most popular one is probably from Josh Wills, with whom I had the privilege to work at Cloudera, which is depicted in *Figure 1*. While the details may be argued about, the truth is that data science is always on the intersection of a few disciplines, and a data scientist is not necessarily is an expert on any one of them. Arguably, the first data scientists worked at Facebook, according to Jeff Hammerbacher, who was also one of the Cloudera founders and an early Facebook employee. Facebook needed interdisciplinary skills to extract value from huge amounts of social data at the time. While I call myself a big data scientist, for the purposes of this book, I'd like to use the term machine learning or ML to keep the focus, as I am mixing too much already here.

One other aspect of ML that came about recently and is actively discussed is that the quantity of data beats the sophistication of the models. One can see this in this book in the example of some Spark MLlib implementations, and word2vec for NLP in particular. Speedier ML models that can respond to new environments faster also often beat the more complex models that take hours to build. Thus, ML and big data make a good match.

Last but not least is the emergence of microservices. I spent a great deal of time on the topic of machine and application communication in this book, and Scala with the Akka actors model comes very naturally here.

Functional programming, at least for a good portion of practical programmers, is more about the style of programming than a programming language itself. While Java 8 started having lambda expressions and streams, which came out of functional programming, one can still write in a functional style without these mechanisms or even write a Java-style code in Scala. The two big ideas that brought Scala to prominence in the big data world are lazy evaluation, which greatly simplifies data processing in a multi-threaded or distributed world, and immutability. Scala has two different libraries for collections: one is mutable and another is immutable. While the distinction is subtle from the application user point of view, immutability greatly increases the options from a compiler perspective, and lazy evaluation cannot be a better match for big data, where REPL postpones most of the number crunching towards later stages of the pipeline, increasing interactivity.

Figure 1: One of the possible definitions of a data scientist

Finally, big data. Big data has definitely occupied the headlines for a couple of years now, and a big reason for this is that the amount of data produced by machines today greatly surpasses anything that a human cannot even produce, but even comprehend, without using the computers. The social network companies, such as Facebook, Google, Twitter, and so on, have demonstrated that enough information can be extracted from these blobs of data to justify the tools specifically targeted towards processing big data, such as Hadoop, MapReduce, and Spark.

We will touch on what Hadoop does later in the book, but originally, it was a Band-Aid on top of commodity hardware to be able to deal with a vast amount of information, which the traditional relational DBs at the time were not equipped to handle (or were able, but at a prohibitive price). While big data is probably too big a subject for me to handle in this book, Spark is the focus and is another implementation of Hadoop MapReduce that removes a few inefficiencies of having to deal with persisting data on disk. Spark is a bit more expensive as it consumes more memory in general and the hardware has to be more reliable, but it is more interactive. Furthermore, Spark works on top of Scala—other languages such as Java and Python too—but Scala is the primary API language, and it found certain synergies in how it expresses data pipelines in Scala.

What this book covers

Chapter 1, *Exploratory Data Analysis,* covers how every data analyst begins with an exploratory data analysis. There is nothing new here, except that the new tools allow you to look into larger datasets—possibly spread across multiple computers, as easily as if they were just on a local machine. This, of course, does not prevent you from running the pipeline on a single machine, but even then, the laptop I am writing this on has four cores and about 1,377 threads running at the same time. Spark and Scala (parallel collections) allow you to transparently use this entire dowry, sometimes without explicitly specifying the parallelism. Modern servers may have up to 128 hyper-threads available to the OS. This chapter will show you how to start with the new tools, maybe by exploring your old datasets.

Chapter 2, *Data Pipelines and Modeling*, explains that while data-driven processes existed long before Scala/Spark, the new age demonstrated the emergence of a fully data-driven enterprise where the business is optimized by the feedback from multiple data-generating machines. Big data requires new techniques and architectures to accommodate the new decision making process. Borrowing from a number of academic fields, this chapter proceeds to describe a generic architecture of a data-driven business, where most of the workers' task is monitoring and tuning the data pipelines (or enjoying the enormous revenue per worker that these enterprises can command).

Chapter 3, *Working with Spark and MLlib*, focuses on the internal architecture of Spark, which we mentioned earlier as a replacement for and/or complement to Hadoop MapReduce. We will specifically stop on a few ML algorithms, which are grouped under the MLlib tag. While this is still a developing topic and many of the algorithms are being moved using a different package now, we will provide a few examples of how to run standard ML algorithms in the `org.apache.spark.mllib` package. We will also explain the modes that Spark can be run under and touch on Spark performance tuning.

Chapter 4, *Supervised and Unsupervised Learning*, explains that while Spark MLlib may be a moving target, general ML principles have been solidly established. Supervised/unsupervised learning is a classical division of ML algorithms that work on row-oriented data—most of the data, really. This chapter is a classic part of any ML book, but we spiced it up a bit to make it more Scala/Spark-oriented.

Chapter 5, *Regression and Classification*, introduces regression and classification, which is another classic subdivision of the ML algorithms, even if it has been shown that classification can be used to regress, and regression to classify, still these are the two classes that use different techniques, precision metrics, and ways to regularize the models. This chapter will take a practical approach while showing you practical examples of regression and classification analysis

Chapter 6, *Working with Unstructured Data,* covers how one of the new features that social data brought with them and brought traditional DBs to their knees is nested and unstructured data. Working with unstructured data requires new techniques and formats, and this chapter is dedicated to the ways to present, store, and evolve these types of data. Scala becomes a big winner here, as it has a natural way to deal with complex data structures in the data pipelines.

Chapter 7, *Working with Graph Algorithms*, explains how graphs present another challenge to the traditional row-oriented DBs. Lately, there has been a resurgence of graph DBs. We will cover two different libraries in this chapter: one is Scala-graph from Assembla, which is a convenient tool to represent and reason with graphs, and the other is Spark's graph class with a few graph algorithms implemented on top of it.

Chapter 8, *Integrating Scala with R and Python*, covers how even though Scala is cool, many people are just too cautious to leave their old libraries behind. In this chapter, I will show how to transparently refer to the legacy code written in R and Python, a request I hear too often. In short, there are too mechanisms: one is using Unix pipelines and another way is to launch R or Python in JVM.

Chapter 9, *NLP in Scala*, focuses on how natural language processing has deal with human-computer interaction and computer's understanding of our often-substandard ways to communicate. I will focus on a few tools that Scala specifically provide for NLP, topic association, and dealing with large amounts of textual information (Spark).

Chapter 10, *Advanced Model Monitoring*, introduces how developing data pipelines usually means that someone is going to use and debug them. Monitoring is extremely important not only for the end user data pipeline, but also for the developer or designer who is looking for the ways to either optimize the execution or further the design. We cover the standard tools for monitoring systems and distributed clusters of machines as well as how to design a service that has enough hooks to look into its functioning without attaching a debugger. I will also touch on the new emerging field of statistical model monitoring.

What you need for this book

This book is based on open source software. First, it's Java. One can download Java from Oracle's Java Download page. You have to accept the license and choose an appropriate image for your platform. Don't use OpenJDK—it has a few problems with Hadoop/Spark.

Second, Scala. If you are using Mac, I recommend installing Homebrew:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

Multiple open source packages will also be available to you. To install Scala, run `brew install scala`. Installation on a Linux platform requires downloading an appropriate Debian or RPM package from the `http://www.scala-lang.org/download/` site. We will use the latest version at the time, that is, 2.11.7.

Spark distributions can be downloaded from `http://spark.apache.org/downloads.html`. We use pre-build for Hadoop 2.6 and later image. As it's Java, you need to just unzip the package and start using the scripts from the `bin` subdirectory.

R and Python packages are available at `http://cran.r-project.org/bin` and `http://python.org/ftp/python/$PYTHON_VERSION/Python-$PYTHON_VERSION.tar.xz` sites respectively. The text has specific instruction on how to configure them. Although our use of the packages should be version agnostic, I used R version 3.2.3 and Python version 2.7.11 in this book.

Who this book is for

Professional and emerging data scientists who want to sharpen their skills and see practical examples of working with big data: a data analyst who wants to effectively extract actionable information from large amounts of data and an aspiring statistician who is willing to get beyond the existing boundaries and become a data scientist.

The book style is pretty much hands-on, I don't delve into mathematical proofs or validations, with a few exceptions, and there are more in-depth texts that I recommend throughout the book. However, I will try my best to provide code samples and tricks that you can start using for the standard techniques and libraries as soon as possible.

Conventions

In this book, you will find a number of text styles 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: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
import scala.util.hashing.MurmurHash3._

val markLow = 0
val markHigh = 4096
val seed = 12345

def consistentFilter(s: String): Boolean = {
  val hash = stringHash(s.split(" ")(0), seed) >>> 16
  hash >= markLow && hash < markHigh
}

val w = new java.io.FileWriter(new java.io.File("out.txt"))
val lines = io.Source.fromFile("chapter01/data/iris/in.txt").getLines
lines.filter(consistentFilter).foreach { s =>
     w.write(s + Properties.lineSeparator)
}
```

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

```
akozlov@Alexanders-MacBook-Pro]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala> import scala.util.Random
import scala.util.Random
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Run all cells at once by navigating to **Cell** | **Run All**."

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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in 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 at `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 this book 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.

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/Mastering-Scala-Machine-Learning`. 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/MasteringScalaMachineLearning_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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Exploratory Data Analysis

Before I dive into more complex methods to analyze your data later in the book, I would like to stop at basic data exploratory tasks on which almost all data scientists spend at least 80-90% of their productive time. The data preparation, cleansing, transforming, and joining the data alone is estimated to be a $44 billion/year industry alone (*Data Preparation in the Big Data Era* by *Federico Castanedo* and *Best Practices for Data Integration*, *O'Reilly Media*, *2015*). Given this fact, it is surprising that people only recently started spending more time on the science of developing best practices and establishing good habits, documentation, and teaching materials for the whole process of data preparation (*Beautiful Data: The Stories Behind Elegant Data Solutions*, edited by *Toby Segaran* and *Jeff Hammerbacher*, *O'Reilly Media*, *2009* and *Advanced Analytics with Spark: Patterns for Learning from Data at Scale* by *Sandy Ryza et al.*, *O'Reilly Media*, *2015*).

Few data scientists would agree on specific tools and techniques—and there are multiple ways to perform the exploratory data analysis, ranging from Unix command line to using very popular open source and commercial ETL and visualization tools. The focus of this chapter is how to use Scala and a laptop-based environment to benefit from techniques that are commonly referred as a functional paradigm of programming. As I will discuss, these techniques can be transferred to exploratory analysis over distributed system of machines using Hadoop/Spark.

What has functional programming to do with it? Spark was developed in Scala for a good reason. Many basic principles that lie at the foundation of functional programming, such as lazy evaluation, immutability, absence of side effects, list comprehensions, and monads go really well with processing data in distributed environments, specifically, when performing the data preparation and transformation tasks on big data. Thanks to abstractions, these techniques work well on a local workstation or a laptop. As mentioned earlier, this does not preclude us from processing very large datasets up to dozens of TBs on modern laptops connected to distributed clusters of storage/processing nodes. We can do it one topic or focus area at the time, but often we even do not have to sample or filter the dataset with proper partitioning. We will use Scala as our primary tool, but will resort to other tools if required.

While Scala is complete in the sense that everything that can be implemented in other languages can be implemented in Scala, Scala is fundamentally a high-level, or even a scripting, language. One does not have to deal with low-level details of data structures and algorithm implementations that in their majority have already been tested by a plethora of applications and time, in, say, Java or C++—even though Scala has its own collections and even some basic algorithm implementations today. Specifically, in this chapter, I'll be focusing on using Scala/Spark only for high-level tasks.

In this chapter, we will cover the following topics:

- Installing Scala
- Learning simple techniques for initial data exploration
- Learning how to downsample the original dataset for faster turnover
- Discussing the implementation of basic data transformation and aggregations in Scala
- Getting familiar with big data processing tools such as Spark and Spark Notebook
- Getting code for some basic visualization of datasets

Getting started with Scala

If you have already installed Scala, you can skip this paragraph. One can get the latest Scala download from `http://www.scala-lang.org/download/`. I used Scala version 2.11.7 on Mac OS X El Capitan 10.11.5. You can use any other version you like, but you might face some compatibility problems with other packages such as Spark, a common problem in open source software as the technology adoption usually lags by a few released versions.

In most cases, you should try to maintain precise match between the recommended versions as difference in versions can lead to obscure errors and a lengthy debugging process.

If you installed Scala correctly, after typing `scala`, you should see something similar to the following:

```
[akozlov@Alexanders-MacBook-Pro ~]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

This is a Scala **read-evaluate-print-loop** (**REPL**) prompt. Although Scala programs can be compiled, the content of this chapter will be in REPL, as we are focusing on interactivity with, maybe, a few exceptions. The `:help` command provides a some utility commands available in REPL (note the colon at the start):

```
[15:05:20 2.6.0-cdh5.5.0 akozlov@Alexanders-MacBook-Pro chapter01(master)]$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_40).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :help
All commands can be abbreviated, e.g., :he instead of :help.
:edit <id>|<line>        edit history
:help [command]          print this summary or command-specific help
:history [num]           show the history (optional num is commands to show)
:h? <string>             search the history
:imports [name name ...] show import history, identifying sources of names
:implicits [-v]          show the implicits in scope
:javap <path|class>      disassemble a file or class name
:line <id>|<line>        place line(s) at the end of history
:load <path>             interpret lines in a file
:paste [-raw] [path]     enter paste mode or paste a file
:power                   enable power user mode
:quit                    exit the interpreter
:replay [options]        reset the repl and replay all previous commands
:require <path>          add a jar to the classpath
:reset [options]         reset the repl to its initial state, forgetting all session entries
:save <path>             save replayable session to a file
:sh <command line>       run a shell command (result is implicitly => List[String])
:settings <options>      update compiler options, if possible; see reset
:silent                  disable/enable automatic printing of results
:type [-v] <expr>        display the type of an expression without evaluating it
:kind [-v] <expr>        display the kind of expression's type
:warnings                show the suppressed warnings from the most recent line which had any
```

Distinct values of a categorical field

Now, you have a dataset and a computer. For convenience, I have provided you a small anonymized and obfuscated sample of clickstream data with the book repository that you can get at `https://github.com/alexvk/ml-in-scala.git`. The file in the `chapter01/data/clickstream` directory contains lines with timestamp, session ID, and some additional event information such as URL, category information, and so on at the time of the call. The first thing one would do is apply transformations to find out the distribution of values for different columns in the dataset.

Figure 01-1 shows screenshot shows the output of the dataset in the terminal window of the `gzcat chapter01/data/clickstream/clickstream_sample.tsv.gz | less -U` command. The columns are tab (`^I`) separated. One can notice that, as in many real-world big data datasets, many values are missing. The first column of the dataset is recognizable as the timestamp. The file contains complex data such as arrays, structs, and maps, another feature of big data datasets.

Unix provides a few tools to dissect the datasets. Probably, **less**, **cut**, **sort**, and **uniq** are the most frequently used tools for text file manipulations. **Awk**, **sed**, **perl**, and **tr** can do more complex transformations and substitutions. Fortunately, Scala allows you to transparently use command-line tools from within Scala REPL, as shown in the following screenshot:

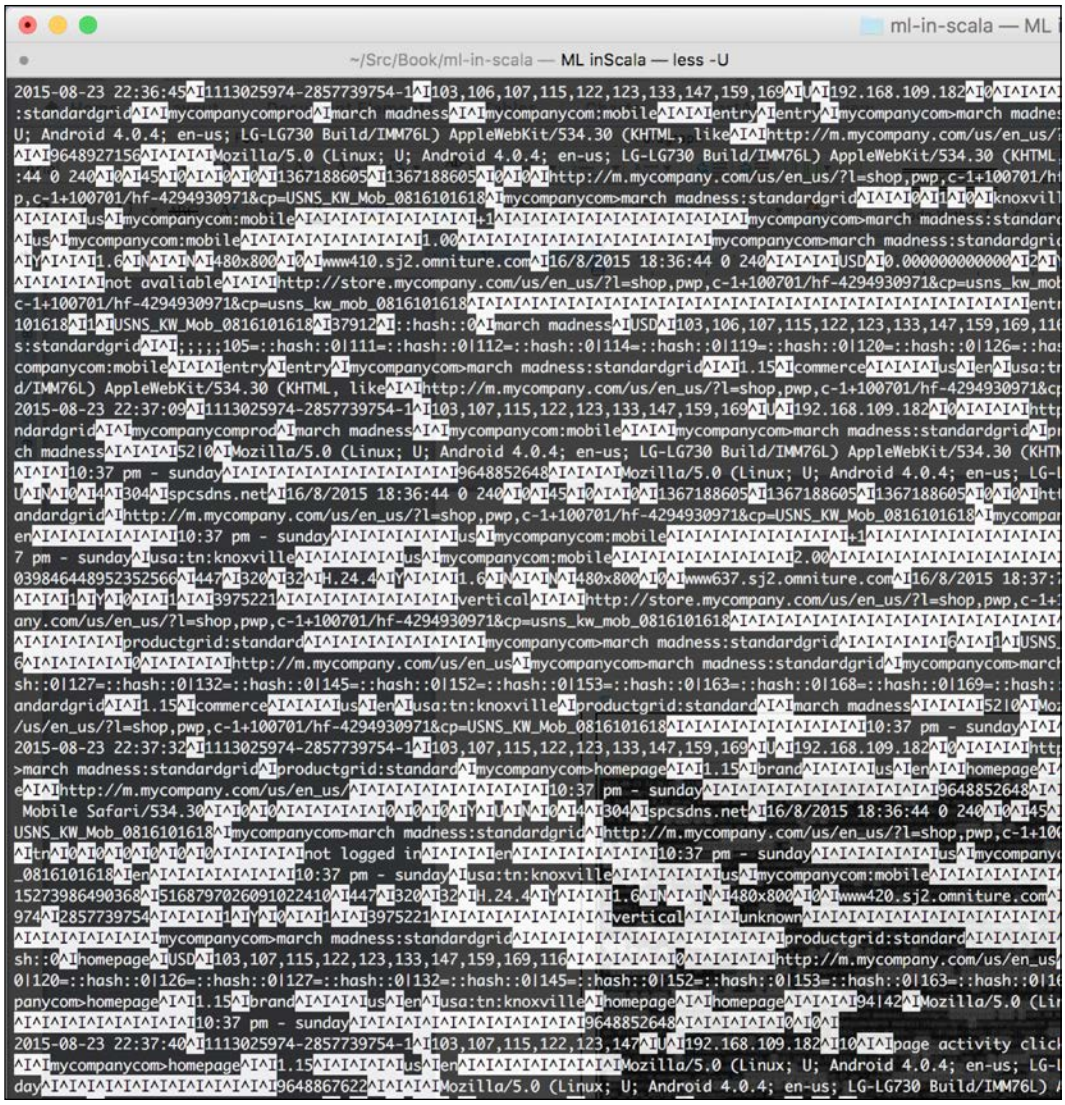

Figure 01-1. The clickstream file as an output of the less -U Unix command

Fortunately, Scala allows you to transparently use command-line tools from within Scala REPL:

```
[akozlov@Alexanders-MacBook-Pro]$ scala
…
scala> import scala.sys.process._
import scala.sys.process._
```

```
scala> val histogram = ( "gzcat chapter01/data/clickstream/clickstream_
sample.tsv.gz"  #|  "cut -f 10" #| "sort" #|  "uniq -c" #| "sort -k1nr"
).lineStream

histogram: Stream[String] = Stream(7731 http://www.mycompany.com/us/en_
us/, ?)

scala> histogram take(10) foreach println

7731 http://www.mycompany.com/us/en_us/

3843 http://mycompanyplus.mycompany.com/plus/

2734 http://store.mycompany.com/us/en_us/?l=shop,men_shoes

2400 http://m.mycompany.com/us/en_us/

1750 http://store.mycompany.com/us/en_us/?l=shop,men_mycompanyid

1556 http://www.mycompany.com/us/en_us/c/mycompanyid?sitesrc=id_redir

1530 http://store.mycompany.com/us/en_us/

1393 http://www.mycompany.com/us/en_us/?cp=USNS_KW_0611081618

1379 http://m.mycompany.com/us/en_us/?ref=http%3A%2F%2Fwww.mycompany.
com%2F

1230 http://www.mycompany.com/us/en_us/c/running
```

I used the `scala.sys.process` package to call familiar Unix commands from Scala REPL. From the output, we can immediately see the customers of our Webshop are mostly interested in men's shoes and running, and that most visitors are using the referral code, **KW_0611081618**.

One may wonder when we start using complex Scala types and algorithms. Just wait, a lot of highly optimized tools were created before Scala and are much more efficient for explorative data analysis. In the initial stage, the biggest bottleneck is usually just the disk I/O and slow interactivity. Later, we will discuss more iterative algorithms, which are usually more memory intensive. Also note that the UNIX pipeline operations can be implicitly parallelized on modern multi-core computer architectures, as they are in Spark (we will show it in the later chapters).

It has been shown that using compression, implicit or explicit, on input data files can actually save you the I/O time. This is particularly true for (most) modern semi-structured datasets with repetitive values and sparse content. Decompression can also be implicitly parallelized on modern fast multi-core computer architectures, removing the computational bottleneck, except, maybe in cases where compression is implemented implicitly in hardware (SSD, where we don't need to compress the files explicitly). We also recommend using directories rather than files as a paradigm for the dataset, where the insert operation is reduced to dropping the data file into a directory. This is how the datasets are presented in big data Hadoop tools such as Hive and Impala.

Summarization of a numeric field

Let's look at the numeric data, even though most of the columns in the dataset are either categorical or complex. The traditional way to summarize the numeric data is a five-number-summary, which is a representation of the median or mean, interquartile range, and minimum and maximum. I'll leave the computations of the median and interquartile ranges till the Spark DataFrame is introduced, as it makes these computations extremely easy; but we can compute mean, min, and max in Scala by just applying the corresponding operators:

```
scala> import scala.sys.process._

import scala.sys.process._

scala> val nums = ( "gzcat chapter01/data/clickstream/clickstream_sample.
tsv.gz"  #|  "cut -f 6" ).lineStream

nums: Stream[String] = Stream(0, ?)

scala> val m = nums.map(_.toDouble).min

m: Double = 0.0

scala> val m = nums.map(_.toDouble).sum/nums.size

m: Double = 3.6883642764024662

scala> val m = nums.map(_.toDouble).max

m: Double = 33.0
```

Grepping across multiple fields

Sometimes one needs to get an idea of how a certain value looks across multiple fields—most common are IP/MAC addresses, dates, and formatted messages. For examples, if I want to see all IP addresses mentioned throughout a file or a document, I need to replace the `cut` command in the previous example by `grep -o -E [1-9][0-9]{0,2}(?:\\.[1-9][0-9]{0,2}){3}`, where the `-o` option instructs `grep` to print only the matching parts—a more precise regex for the IP address should be `grep -o -E (?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)`, but is about 50% slower on my laptop and the original one works in most practical cases. I'll leave it as an excursive to run this command on the sample file provided with the book.

Basic, stratified, and consistent sampling

I've met quite a few data practitioners who scorn sampling. Ideally, if one can process the whole dataset, the model can only improve. In practice, the tradeoff is much more complex. First, one can build more complex models on a sampled set, particularly if the time complexity of the model building is non-linear—and in most situations, if it is at least *N* log(N)*. A faster model building cycle allows you to iterate over models and converge on the best approach faster. In many situations, *time to action* is beating the potential improvements in the prediction accuracy due to a model built on complete dataset.

Sampling may be combined with appropriate filtering—in many practical situation, focusing on a subproblem at a time leads to better understanding of the whole problem domain. In many cases, this partitioning is at the foundation of the algorithm, like in decision trees, which are considered later. Often the nature of the problem requires you to focus on the subset of original data. For example, a cyber security analysis is often focused around a specific set of IPs rather than the whole network, as it allows to iterate over hypothesis faster. Including the set of all IPs in the network may complicate things initially if not throw the modeling off the right track.

When dealing with rare events, such as clickthroughs in ADTECH, sampling the positive and negative cases with different probabilities, which is also sometimes called oversampling, often leads to better predictions in short amount of time.

Fundamentally, sampling is equivalent to just throwing a coin—or calling a random number generator—for each data row. Thus it is very much like a stream filter operation, where the filtering is on an augmented column of random numbers. Let's consider the following example:

```
import scala.util.Random
import util.Properties

val threshold = 0.05

val lines = scala.io.Source.fromFile("chapter01/data/iris/in.txt").
getLines
val newLines = lines.filter(_ =>
    Random.nextDouble() <= threshold
)

val w = new java.io.FileWriter(new java.io.File("out.txt"))
newLines.foreach { s =>
    w.write(s + Properties.lineSeparator)
}
w.close
```

This is all good, but it has the following disadvantages:

- The number of lines in the resulting file is not known beforehand—even though on average it should be 5% of the original file
- The results of the sampling is non-deterministic—it is hard to rerun this process for either testing or verification

To fix the first point, we'll need to pass a more complex object to the function, as we need to maintain the state during the original list traversal, which makes the original algorithm less functional and parallelizable (this will be discussed later):

```
import scala.reflect.ClassTag
import scala.util.Random
import util.Properties

def reservoirSample[T: ClassTag](input: Iterator[T],k: Int): Array[T]
= {
  val reservoir = new Array[T](k)
  // Put the first k elements in the reservoir.
  var i = 0
  while (i < k && input.hasNext) {
    val item = input.next()
    reservoir(i) = item
    i += 1
  }

  if (i < k) {
    // If input size < k, trim the array size
    reservoir.take(i)
  } else {
    // If input size > k, continue the sampling process.
    while (input.hasNext) {
      val item = input.next
      val replacementIndex = Random.nextInt(i)
      if (replacementIndex < k) {
        reservoir(replacementIndex) = item
      }
      i += 1
    }
    reservoir
  }
}

val numLines=15
```

```
val w = new java.io.FileWriter(new java.io.File("out.txt"))
val lines = io.Source.fromFile("chapter01/data/iris/in.txt").getLines
reservoirSample(lines, numLines).foreach { s =>
    w.write(s + scala.util.Properties.lineSeparator)
}
w.close
```

This will output `numLines` lines. Similarly to reservoir sampling, stratified sampling is guaranteed to provide the same ratios of input/output rows for all strata defined by levels of another attribute. We can achieve this by splitting the original dataset into *N* subsets corresponding to the levels, performing the reservoir sampling, and merging the results afterwards. However, MLlib library, which will be covered in *Chapter 3*, *Working with Spark and MLlib*, already has stratified sampling implementation:

```
val origLinesRdd = sc.textFile("file://...")
val keyedRdd = origLines.keyBy(r => r.split(",")(0))
val fractions = keyedRdd.countByKey.keys.map(r => (r, 0.1)).toMap
val sampledWithKey = keyedRdd.sampleByKeyExact(fractions)
val sampled = sampledWithKey.map(_._2).collect
```

The other bullet point is more subtle; sometimes we want a consistent subset of values across multiple datasets, either for reproducibility or to join with another sampled dataset. In general, if we sample two datasets, the results will contain random subsets of IDs which might have very little or no intersection. The cryptographic hashing functions come to the help here. The result of applying a hash function such as MD5 or SHA1 is a sequence of bits that is statistically uncorrelated, at least in theory. We will use the `MurmurHash` function, which is part of the `scala.util.hashing` package:

```
import scala.util.hashing.MurmurHash3._

val markLow = 0
val markHigh = 4096
val seed = 12345

def consistentFilter(s: String): Boolean = {
  val hash = stringHash(s.split(" ")(0), seed) >>> 16
  hash >= markLow && hash < markHigh
}

val w = new java.io.FileWriter(new java.io.File("out.txt"))
val lines = io.Source.fromFile("chapter01/data/iris/in.txt").getLines
lines.filter(consistentFilter).foreach { s =>
     w.write(s + Properties.lineSeparator)
}
w.close
```