Scala for Machine Learning

Leverage Scala and Machine Learning to construct and study systems that can learn from data

Patrick R. Nicolas
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “Getting Started”
- A synopsis of the book’s content
- More information on Scala for Machine Learning

About the Author

Patrick R. Nicolas is a lead R&D engineer at Dell in Santa Clara, California. He has 25 years of experience in software engineering and building large-scale applications in C++, Java, and Scalas, and has held several managerial positions. His interests include real-time analytics, modeling, and optimization.

Special thanks to the Packt Publishing team: Mohammed Fahad for his patience and encouragement, Owen Roberts for the opportunity, and the reviewers for their guidance and dedication.
Scala for Machine Learning

Not a single day passes by that we do not hear about Big Data in the news media, technical conferences, and even coffee shops. The ever-increasing amount of data collected in process monitoring, research, or simple human behavior becomes valuable only if you extract knowledge from it. Machine learning is the essential tool to mine data for gold (knowledge).

This book covers the "what", "why", and "how" of machine learning:

- What are the objectives and the mathematical foundation of machine learning?
- Why is Scala the ideal programming language to implement machine learning algorithms?
- How can you apply machine learning to solve real-world problems?

Throughout this book, machine learning algorithms are described with diagrams, mathematical formulation, and documented snippets of Scala code, allowing you to understand these key concepts in your own unique way.

What This Book Covers

Chapter 1, Getting Started, introduces the basic concepts of statistical analysis, classification, regression, prediction, clustering, and optimization. This chapter covers the Scala languages features and libraries, followed by the implementation of a simple application.

Chapter 2, Hello World!, describes a typical workflow for classification, the concept of bias/variance trade-off, and validation using the Scala dependency injection applied to the technical analysis of financial markets.

Chapter 3, Data Preprocessing, covers time series analyses and leverages Scala to implement data preprocessing and smoothing techniques such as moving averages, discrete Fourier transform, and the Kalman recursive filter.

Chapter 4, Unsupervised Learning, focuses on the implementation of some of the most widely used clustering techniques, such as K-means, the expectation-maximization, and the principal component analysis as a dimension reduction method.

Chapter 5, Naïve Bayes Classifiers, introduces probabilistic graphical models, and then describes the implementation of the Naïve Bayes and the multivariate Bernoulli classifiers in the context of text mining.

Chapter 6, Regression and Regularization, covers a typical implementation of the linear and least squares regression, the ridge regression as a regularization technique, and finally, the logistic regression.
Chapter 7, Sequential Data Models, introduces the Markov processes followed by a full implementation of the hidden Markov model, and conditional random fields applied to pattern recognition in financial market data.

Chapter 8, Kernel Models and Support Vector Machines, covers the concept of kernel functions with implementation of support vector machine classification and regression, followed by the application of the one-class SVM to anomaly detection.

Chapter 9, Artificial Neural Networks, describes feed-forward neural networks followed by a full implementation of the multilayer perceptron classifier.

Chapter 10, Genetic Algorithms, covers the basics of evolutionary computing and the implementation of the different components of a multipurpose genetic algorithm.

Chapter 11, Reinforcement Learning, introduces the concept of reinforcement learning with an implementation of the Q-learning algorithm followed by a template to build a learning classifier system.

Chapter 12, Scalable Frameworks, covers some of the artifacts and frameworks to create scalable applications for machine learning such as Scala parallel collections, Akka, and the Apache Spark framework.

Appendix A, Basic Concepts, covers the Scala constructs used throughout the book, elements of linear algebra, and an introduction to investment and trading strategies.

Appendix B, References, provides a chapter-wise list of references for [source entry] in the respective chapters. This appendix is available as an online chapter at https://www.packtpub.com/sites/default/files/downloads/8742OS_AppendixB_References.pdf.

Short test applications using financial data illustrate the large variety of predictive, regression, and classification models.

The interdependencies between chapters are kept to a minimum. You can easily delve into any chapter once you complete Chapter 1, Getting Started, and Chapter 2, Hello World!
It is critical for any computer scientist to understand the different classes of machine learning algorithms and be able to select the ones that are relevant to the domain of their expertise and dataset. However, the application of these algorithms represents a small fraction of the overall effort needed to extract an accurate and performing model from input data. A common data mining workflow consists of the following sequential steps:

1. Loading the data.
2. Preprocessing, analyzing, and filtering the input data.
3. Discovering patterns, affinities, clusters, and classes.
4. Selecting the model features and the appropriate machine learning algorithm(s).
5. Refining and validating the model.
6. Improving the computational performance of the implementation.

As we will emphasize throughout this book, each stage of the process is critical to build the right model.

This first chapter introduces you to the taxonomy of machine learning algorithms, the tools and frameworks used in the book, and a simple application of logistic regression to get your feet wet.
Mathematical notation for the curious

Each chapter contains a small section dedicated to the formulation of the algorithms for those interested in the mathematical concepts behind the science and art of machine learning. These sections are optional and defined within a tip box. For example, the mathematical expression of the mean and the variance of a variable $X$ mentioned in a tip box will be as follows:

\[
\text{Mean value of a variable } X = \{x\} \text{ is defined as: } \\
E(X) = \frac{1}{n} \sum x_j
\]

\[
\text{The variance of a variable } X = \{x\} \text{ is defined as: } \\
Var(X) = \frac{\sum(E(X) - x_j)^2}{n - 1}
\]

Why machine learning?

The explosion in the number of digital devices generates an ever-increasing amount of data. The best analogy I can find to describe the need, desire, and urgency to extract knowledge from large datasets is the process of extracting a precious metal from a mine, and in some cases, extracting blood from a stone.

Knowledge is quite often defined as a model that can be constantly updated or tweaked as new data comes into play. Models are obviously domain-specific ranging from credit risk assessment, face recognition, maximization of quality of service, classification of pathological symptoms of disease, optimization of computer networks, and security intrusion detection, to customers' online behavior and purchase history.

Machine learning problems are categorized as classification, prediction, optimization, and regression.

Classification

The purpose of classification is to extract knowledge from historical data. For instance, a classifier can be built to identify a disease from a set of symptoms. The scientist collects information regarding the body temperature (continuous variable), congestion (discrete variables HIGH, MEDIUM, and LOW), and the actual diagnostic (flu). This dataset is used to create a model such as IF temperature > 102 AND congestion = HIGH THEN patient has the flu (probability 0.72), which doctors can use in their diagnostic.
Prediction
Once the model is extracted and validated against the past data, it can be used to draw inference from the future data. A doctor collects symptoms from a patient, such as body temperature and nasal congestion, and anticipates the state of his/her health.

Optimization
Some global optimization problems are intractable using traditional linear and non-linear optimization methods. Machine learning techniques improve the chances that the optimization method converges toward a solution (intelligent search). You can imagine that fighting the spread of a new virus requires optimizing a process that may evolve over time as more symptoms and cases are uncovered.

Regression
Regression is a classification technique that is particularly suitable for a continuous model. Linear (least square), polynomial, and logistic regressions are among the most commonly used techniques to fit a parametric model, or function, \( y = f(x_j) \), to a dataset. Regression is sometimes regarded as a specialized case of classification for which the output variables are continuous instead of categorical.

Why Scala?
Like most functional languages, Scala provides developers and scientists with a toolbox to implement iterative computations that can be easily woven dynamically into a coherent dataflow. To some extent, Scala can be regarded as an extension of the popular MapReduce model for distributed computation of large amounts of data. Among the capabilities of the language, the following features are deemed essential to machine learning and statistical analysis.

Abstraction
Monoids and monads are important concepts in functional programming. Monads are derived from the category and group theory allowing developers to create a high-level abstraction as illustrated in Twitter's Algebird (https://github.com/twitter/algebird) or Google's Breeze Scala (https://github.com/dlwh/breeze) libraries.

A monoid defines a binary operation \( \otimes \) on a dataset \( T \) with the property of closure, identity operation, and associativity.
Getting Started

Let's consider the + operation is defined for a set T using the following monoidal representation:

```scala
trait Monoid[T] {
  def zero: T
  def op(a: T, b: T): c
}
```

Monoids are associative operations. For instance, if ts1, ts2, and ts3 are three time series, then the property ts1 + (ts2 + ts3) = (ts1 + ts2) + ts2 is true. The associativity of a monoid operator is critical in regards to parallelization of computational workflows.

Monads are structures that can be seen either as containers by programmers or as a generalization of Monoids. The collections bundled with the Scala standard library (list, map, and so on) are constructed as monads [1:1]. Monads provide the ability for those collections to perform the following functions:

1. Create the collection.
2. Transform the elements of the collection.
3. Flatten nested collections.

A common categorical representation of a monad in Scala is a trait, Monad, parameterized with a container type M:

```scala
trait Monad[M[_]] {
  def apply[T](a: T): M[T]
  def flatMap[T, U](m: M[T])(f: T=>M[U]): M[U]
}
```

Monads allow those collections or containers to be chained to generate a workflow. This property is applicable to any scientific computation [1:2].

**Scalability**

As seen previously, monoids and monads enable parallelization and chaining of data processing functions by leveraging the Scala higher-order methods. In terms of implementation, Actors are the core elements that make Scala scalable. Actors act as coroutines, managing the underlying threads pool. Actors communicate through passing asynchronous messages. A distributed computing Scala framework such as Akka and Spark extends the capabilities of the Scala standard library to support computation on very large datasets. Akka and Spark are described in detail in the last chapter of this book [1:3].
In a nutshell, a workflow is implemented as a sequence of activities or computational tasks. Those tasks consist of high-order Scala methods such as `flatMap`, `map`, `fold`, `reduce`, `collect`, `join`, or `filter` applied to a large collection of observations. Scala allows these observations to be partitioned by executing those tasks through a cluster of actors. Scala also supports message dispatching and routing of messages between local and remote actors. The engineers can decide to execute a workflow either locally or distributed across CPU cores and servers with no code or very little code changes.

In this diagram, a controller, that is, the master node, manages the sequence of tasks 1 to 4 similar to a scheduler. These tasks are actually executed over multiple worker nodes that are implemented by the Scala actors. The master node exchanges messages with the workers to manage the state of the execution of the workflow as well as its reliability. High availability of these tasks is implemented through a hierarchy of supervising actors.

**Configurability**

Scala supports **dependency injection** using a combination of abstract variables, self-referenced composition, and stackable traits. One of the most commonly used dependency injection patterns, the **cake pattern**, is used throughout this book to create dynamic computation workflows and plots.
Maintainability

Scala embeds Domain Specific Languages (DSL) natively. DSLs are syntactic layers built on top of Scala native libraries. DSLs allow software developers to abstract computation in terms that are easily understood by scientists. The most notorious application of DSLs is the definition of the emulation of the syntax used in the MATLAB program, which data scientists are familiar with.

Computation on demand

Lazy methods and values allow developers to execute functions and allocate computing resources on demand. The Spark framework relies on lazy variables and methods to chain Resilient Distributed Datasets (RDD).

Model categorization

A model can be predictive, descriptive, or adaptive.

Predictive models discover patterns in historical data and extract fundamental trends and relationships between factors. They are used to predict and classify future events or observations. Predictive analytics is used in a variety of fields such as marketing, insurance, and pharmaceuticals. Predictive models are created through supervised learning using a preselected training set.

Descriptive models attempt to find unusual patterns or affinities in data by grouping observations into clusters with similar properties. These models define the first level in knowledge discovery. They are generated through unsupervised learning.

A third category of models, known as adaptive modeling, is generated through reinforcement learning. Reinforcement learning consists of one or several decision-making agents that recommend and possibly execute actions in the attempt of solving a problem, optimizing an objective function, or resolving constraints.
Taxonomy of machine learning algorithms

The purpose of machine learning is to teach computers to execute tasks without human intervention. An increasing number of applications such as genomics, social networking, advertising, or risk analysis generate a very large amount of data that can be analyzed or mined to extract knowledge or provide insight into a process, a customer, or an organization. Ultimately, machine learning algorithms consist of identifying and validating models to optimize a performance criterion using historical, present, and future data [1:4].

Data mining is the process of extracting or identifying patterns in a dataset.

Unsupervised learning

The goal of unsupervised learning is to discover patterns of regularities and irregularities in a set of observations. The process known as density estimation in statistics is broken down into two categories: discovery of data clusters and discovery of latent factors. The methodology consists of processing input data to understand patterns similar to the natural learning process in infants or animals. Unsupervised learning does not require labeled data, and therefore, is easy to implement and execute because no expertise is needed to validate an output. However, it is possible to label the output of a clustering algorithm and use it for future classification.

Clustering

The purpose of data clustering is to partition a collection of data into a number of clusters or data segments. Practically, a clustering algorithm is used to organize observations into clusters by minimizing the observations within a cluster and maximizing the observations between clusters. A clustering algorithm consists of the following steps:

1. Creating a model by making an assumption on the input data.
2. Selecting the objective function or goal of the clustering.
3. Evaluating one or more algorithms to optimize the objective function.

Data clustering is also known as data segmentation or data partitioning.
Dimension reduction

Dimension reduction techniques aim at finding the smallest but most relevant set of features that models dataset reliability. There are many reasons for reducing the number of features or parameters in a model, from avoiding overfitting to reducing computation costs.

There are many ways to classify the different techniques used to extract knowledge from data using unsupervised learning. The following taxonomy breaks down these techniques according to their purpose, although the list is far from being exhaustive, as shown in the following diagram:

Supervised learning

The best analogy for supervised learning is function approximation or curve fitting. In its simplest form, supervised learning attempts to extract a relation or function \( f: x \rightarrow y \) from a training set \( \{x, y\} \). Supervised learning is far more accurate and reliable than any other learning strategy. However, a domain expert may be required to label (tag) data as a training set for certain types of problems.

Supervised machine learning algorithms can be broken into two categories:

- Generative models
- Discriminative models

Generative models

In order to simplify the description of statistics formulas, we adopt the following simplification: the probability of an event \( X \) is the same as the probability of the discrete random variable \( X \) to have a value \( x \), \( p(X) = p(X=x) \). The notation of joint probability (resp. conditional probability) becomes \( p(X, Y) = p(X=x, Y=y) \) (resp. \( p(X \mid Y)=p(X=x \mid Y=y) \)).
Generative models attempt to fit a joint probability distribution, $p(X,Y)$, of two events (or random variables), $X$ and $Y$, representing two sets of observed and hidden (latent) variables $x$ and $y$. Discriminative models learn the conditional probability $p(Y|X)$ of an event or random variable $Y$ of hidden variables $y$, given an event or random variable $X$ of observed variables $x$. Generative models are commonly introduced through the Bayes’ rule. The conditional probability of an event $Y$, given an event $X$, is computed as the product of the conditional probability of the event $X$, given the event $Y$, and the probability of the event $X$ normalized by the probability of event $Y$ [1:5].

Join probability (if $X$ and $Y$ are independent):

$$p(X,Y) = p(X \cap Y) = p(X) \cdot p(Y)$$

Conditional probability:

$$p(Y|X) = \frac{P(Y,X)}{P(X)}$$

The Bayes’ rule:

$$P(Y|X) = P(X|Y) \cdot P(X)/P(Y)$$

The Bayes’ rule is the foundation of the Naïve Bayes classifier, which is the topic of Chapter 5, Naïve Bayes Classifiers.

**Discriminative models**

Contrary to generative models, discriminative models compute the conditional probability $p(Y|X)$ directly, using the same algorithm for training and classification.

Generative and discriminative models have their respective advantages and drawbacks. Novice data scientists learn to match the appropriate algorithm to each problem through experimentation. Here is a brief guideline describing which type of models makes sense according to the objective or criteria of the project:

<table>
<thead>
<tr>
<th>Objective</th>
<th>Generative models</th>
<th>Discriminative models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>Highly dependent on the training set.</td>
<td>Probability estimates tend to be more accurate.</td>
</tr>
<tr>
<td>Modeling requirements</td>
<td>There is a need to model both observed and hidden variables, which requires a significant amount of training.</td>
<td>The quality of the training set does not have to be as rigorous as for generative models.</td>
</tr>
</tbody>
</table>
Getting Started

<table>
<thead>
<tr>
<th>Objective</th>
<th>Generative models</th>
<th>Discriminative models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation cost</td>
<td>This is usually low. For example, any graphical method derived from the Bayes' rule has low overhead.</td>
<td>Most algorithms rely on optimization of a convex that introduces significant performance overhead.</td>
</tr>
<tr>
<td>Constraints</td>
<td>These models assume some degree of independence among the model features.</td>
<td>Most discriminative algorithms accommodate dependencies between features.</td>
</tr>
</tbody>
</table>

We can further refine the taxonomy of supervised learning algorithms by segregating between sequential and random variables for generative models and breaking down discriminative methods as applied to continuous processes (regression) and discrete processes (classification):

---

**Reinforcement learning**

Reinforcement learning is not as well understood as supervised and unsupervised learning outside the realms of robotics or game strategy. However, since the 90s, genetic-algorithms-based classifiers have become increasingly popular to solve problems that require collaboration with a domain expert. For some types of applications, reinforcement learning algorithms output a set of recommended actions for the *adaptive* system to execute. In its simplest form, these algorithms compute or estimate the best course of action. Most complex systems based on reinforcement learning establish and update policies that can be vetoed by an expert. The foremost challenge developers of reinforcement learning systems face is that the recommended action or policy may depend on partially observable states and how to deal with uncertainty.
Genetic algorithms are not usually considered part of the reinforcement learning toolbox. However, advanced models such as learning classifier systems use genetic algorithms to classify and reward the rules and policies.

As with the two previous learning strategies, reinforcement learning models can be categorized as Markovian or evolutionary:

This is a brief overview of machine learning algorithms with a suggested taxonomy. There are almost as many ways to introduce machine learning as there are data and computer scientists. We encourage you to browse through the list of references at the end of the book and find the documentation appropriate to your level of interest and understanding.

**Tools and frameworks**

Before getting your hands dirty, you need to download and deploy a minimum set of tools and libraries so as not to reinvent the wheel. A few key components have to be installed in order to compile and run the source code described throughout the book. We focus on open source and commonly available libraries, although you are invited to experiment with equivalent tools of your choice. The learning curve for the frameworks described here is minimal.

**Java**

The code described in the book has been tested with **JDK 1.7.0.45** and **JDK 1.8.0.25** on **Windows x64** and **MacOS X x64**. You need to install the Java Development Kit if you have not already done so. Finally, the environment variables `JAVA_HOME`, `PATH`, and `CLASSPATH` have to be updated accordingly.
Scala

The code has been tested with Scala 2.10.4. We recommend using Scala version 2.10.3 or higher and SBT 0.13 or higher. Let's assume that Scala runtime (REPL) and libraries have been properly installed and environment variables `SCALA_HOME` and `PATH` have been updated. The description and installation instructions of the Scala plugin for Eclipse are available at http://scala-ide.org/docs/user/gettingstarted.html.

You can also download the Scala plugin for IntelliJ IDEA from the JetBrains website at http://confluence.jetbrains.com/display/SCA/.

The ubiquitous simple build tool (sbt) will be our primary building engine. The syntax of the build file `sbt/build.sbt` conforms to version 0.13, and is used to compile and assemble the source code presented throughout this book.

Apache Commons Math

Apache Commons Math is a Java library for numerical processing, algebra, statistics, and optimization [1:6].

Description

This is a lightweight library that provides developers with a foundation of small, ready-to-use Java classes that can be easily weaved into a machine learning problem. The examples used throughout the book require version 3.3 or higher.

The main components of Apache Commons Math are:

- Functions, differentiation, and integral and ordinary differential equations
- Statistics distribution
- Linear and nonlinear optimization
- Dense and Sparse vectors and matrices
- Curve fitting, correlation, and regression

For more information, visit http://commons.apache.org/proper/commons-math.

Licensing

We need Apache Public License 2.0; the terms are available at http://www.apache.org/licenses/LICENSE-2.0.
Installation

The installation and deployment of the Commons Math library are quite simple:

2. Download the latest .jar files in the Binaries section, commons-math3-3.3-bin.zip (for version 3.3, for instance).
3. Unzip and install the .jar files.
4. Add commons-math3-3.3.jar to classpath as follows:
   - For Mac OS X, use the command export CLASSPATH=$CLASSPATH:/Commons_Math_path/commons-math3-3.3.jar
   - For Windows, navigate to System property | Advanced system settings | Advanced | Environment variables..., then edit the entry of the CLASSPATH variable
5. Add the commons-math3-3.3.jar file to your IDE environment if needed (that is, for Eclipse, navigate to Project | Properties | Java Build Path | Libraries | Add External JARs).

You can also download commons-math3-3.3-src.zip from the Source section.

JFreeChart

JFreeChart is an open source chart and plotting Java library, widely used in the Java programmer community. It was originally created by David Gilbert [1:7].

Description

The library supports a variety of configurable plots and charts (scatter, dial, pie, area, bar, box and whisker, stacked, and 3D). We use JFreeChart to display the output of data processing and algorithms throughout the book, but you are encouraged to explore this great library on your own, as time permits.

Licensing

It is distributed under the terms of the GNU Lesser General Public License (LGPL), which permits its use in proprietary applications.
Getting Started

Installation
To install and deploy JFreeChart, perform the following steps:

3. Unzip and install the .jar file.
4. Add jfreechart-1.0.17.jar (for version 1.0.17) to classpath as follows:
   - For Mac OS, update the classpath by using export CLASSPATH=$CLASSPATH:/JFreeChart_path/jfreechart-1.0.17.jar
   - For Windows, go to System property | Advanced system settings | Advanced | Environment variables... and then edit the entry of the CLASSPATH variable
5. Add the jfreechart-1.0.17.jar file to your IDE environment, if needed.

Other libraries and frameworks
Libraries and tools that are specific to a single chapter are introduced along with the topic. Scalable frameworks are presented in the last chapter along with the instructions to download them. Libraries related to the conditional random fields and support vector machines are described in the respective chapters.

Why not use Scala algebra and numerical libraries
Libraries such as Breeze, ScalaNLP, and Algebird are great Scala frameworks for linear algebra, numerical analysis, and machine learning. They provide even the most seasoned Scala programmer with a high-quality layer of abstraction. However, this book is designed as a tutorial that allows developers to write algorithms from the ground up using simple common Java libraries [1:8].

Source code
The Scala programming language is used to implement and evaluate the machine learning techniques presented in this book. Only a subset of the source code used to implement the techniques are presented in the book. The formal implementation of these algorithms is available on the website of Packt Publishing (http://www.packtpub.com).
Context versus view bounds

Most Scala classes discussed in the book are parameterized with the type associated to the discrete/categorical value (Int) or continuous value (Double). Context bounds would require that any type used by the client code has Int or Double as upper bounds:

```scala
class MyClassInt[T <: Int]
class MyClassFloat[T <: Double]
```

Such a design introduces constraints on the client to inherit from simple types and to deal with covariance and contravariance for container types [1:9].

For this book, view bounds are used instead of context bounds only where they require an implicit conversion to the parameterized type to be defined:

```scala
class MyClassFloat[T <% Double]
implicit def T2Double(t : T): Double
```

Presentation

For the sake of readability of the implementation of algorithms, all nonessential code such as error checking, comments, exceptions, or imports are omitted. The following code elements are discarded in the code snippet presented in the book:

- Code comments
- Validation of class parameters and method arguments:
  ```scala
class BaumWelchEM(val lambda: HMMLambda ...) {
  require( lambda != null, "Lambda model is undefined")
  ```
- Exceptions and an exception handler:
  ```scala
  try { .. }
  catch {
    case e: ArrayIndexOutOfBoundsException =>println(e.toString)
  }
  ```
• Nonessential annotation:
  @inline def mean = ..

• Logging and debugging code:
  m_logger.debug( ...)

• Private and nonessential methods

Primitives and implicit

The algorithms presented in this book share the same primitive types, generic operators, and implicit conversions.

Primitive types

For the sake of readability of the code, the following primitive types will be used:

type XY = (Double, Double)
type XYTSeries = Array[(Double, Double)]
type DMatrix[T] = Array[Array[T]]
type DVector[T] = Array[T]
type DblMatrix = DMatrix[Double]
type DblVector = Array[Double]

The types have the behavior (methods) of their primitive counterpart (array). However, adding a new functionality to vectors, matrices, and time series requires classes of their own right. These classes will be introduced in the next chapter.

Type conversions

Implicit conversion is an important feature of the Scala programming language because it allows developers to specify a type conversion for an entire library in a single place. Here are a few of the implicit type conversions used throughout the book:

implicit def int2Double(n: Int): Double = n.toDouble
implicit def vectorT2Db1Vector[T <% Double](vt: DVector[T]): DblVector = vt.map( t => t.toDouble)
implicit def double2DblVector(x: Double): DblVector = Array[Double](x)
implicit def dblPair2DblLVector(x: (Double, Double)): DblVector = Array[Double](x._1, x._2)
implicit def dblPairs2Db1Rows(x: (Double, Double)): DblMatrix = Array[Array[Double]](Array[Double](x._1, x._2))
...
Library-specific conversion
The conversion between the primitive type listed here and types introduced in a particular library (such as Apache Commons Math) is declared in future chapters the first time those libraries are used.

Operators
Lastly, some operations are applied by multiple machine learning or preprocessing algorithms. They need to be defined implicitly. The operation on a pair of a vector of arbitrary type and vector of Double is defined as follows:

\[
\text{def Op}[T <\% \text{Double}](v: \text{DVector}[T], w: \text{DblVector}, op: (T, Double) => Double): \text{DblVector} = \vspace{1mm}
\]
\[
v.\text{zipWithIndex}.\text{map}(x => \text{op}(x._1, w(x._2))) 
\]

It is also convenient to define the following operators that are included in the Scala standard library:

\[
\text{implicit def }/(v: \text{DblVector}, n: \text{Int}):\text{DblVector} = v.\text{map}( x => x/n) 
\]
\[
\text{implicit def }/(m: \text{DblMatrix}, col: \text{Int}, z: \text{Double}): \text{DblMatrix} = \{ (0 until m(n).size).\text{foreach}(i => m(n)(i) /= z) \} 
\]

We won’t have to redefine the types, conversions, and operators from now on.

Immutability
It is usually a good idea to reduce the number of states of an object. Method invocation transitions an object from one state to another. The larger the number of methods or states, the more cumbersome the testing process becomes.

There is no point in creating a model that is not defined (trained). Therefore, making the training of a model as part of the constructor of the class it implements makes a lot of sense. Therefore, the only public methods of a machine learning algorithm are:

- Classification or prediction
- Validation
- Retrieval of model parameters (weights, latent variables, hidden states, and so on), if needed
Performance of Scala iterators
The evaluation of the performance of Scala high-order iterative methods is beyond the scope of this book. However, it is important to be aware of the trade-off of each method.

The for loop construct is to be avoided as a counting iterator except if it is used in conjunction with yield. It is designed to implement the for-comprehension monad (map-flatMap). The source code presented in this book uses the while and foreach constructs.

Scala reducer methods reduce and fold are also frequently used for their efficiency.

Let's kick the tires
This final section introduces the key elements of the training and classification workflow. A test case using a simple logistic regression is used to illustrate each step of the computational workflow.

Overview of computational workflows
In its simplest form, a computational workflow to perform runtime processing of a dataset is composed of the following stages:

1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Preprocessing data using filtering techniques, analysis of variance, and applying penalty and normalization functions whenever necessary.
4. Applying the model, either a set of clusters or classes to classify new data.
5. Assessing the quality of the model.

A similar sequence of tasks is used to extract a model from a training dataset:

1. Loading the dataset from files, databases, or any streaming devices.
2. Splitting the dataset for parallel data processing.
3. Applying filtering techniques, analysis of variance, and penalty and normalization functions to the raw dataset whenever necessary.
4. Selecting the training, testing, and validation set from the cleansed input data.
5. Extracting key features, establishing affinity between a similar group of observations using clustering techniques or supervised learning algorithms.
6. Reducing the number of features to a manageable set of attributes to avoid overfitting the training set.
7. Validating the model and tuning the model by iterating steps 5, 6, and 7 until the error meets criteria.
8. Storing the model into the file or database to be loaded for runtime processing of new observations.

Data clustering and data classification can be performed independent of each other or as part of a workflow that uses clustering techniques as a preprocessing stage of the training phase of a supervised learning algorithm. Data clustering does not require a model to be extracted from a training set, while classification can be performed only if a model has been built from the training set. The following image gives an overview of training and classification:

This diagram is an overview of a typical data mining processing pipeline. The first phase consists of extracting the model through clustering or training of a supervised learning algorithm. The model is then validated against test data, for which the source is the same as the training set but with different observations. Once the model is created and validated, it can be used to classify real-time data or predict future behavior. In reality, real-world workflows are more complex and require being dynamically configurable to allow experimentation of different models. Several alternative classifiers can be used to perform a regression and different filtering algorithms are applied against input data depending on the latent noise in the raw data.
Writing a simple workflow

This book relies on financial data to experiment with a different learning strategy. The objective of the exercise is to build a model that can discriminate between volatile and nonvolatile trading sessions. For this first example, we select a simplified version of the logistic regression as our classifier as we treat a stock-price-volume action as a continuous or pseudo-continuous process.

Logistic regression

Logistic regression is treated in depth in Chapter 6, Regression and Regularization. The model treated in this example is a simple binary classifier using logistic regression for two-dimensional observations.

The classification of trading sessions according to their volatility is as follows:

- Select a dataset
- Load the dataset
- Preprocess the dataset
- Display data
- Create the model through training
- Classify new data

Selecting a dataset

Throughout the book, we will rely on financial data to evaluate and discuss the merit of different data processing and machine learning methods. In this example, the data is extracted from Yahoo! Finances using the CSV format with the following fields:

- Date
- Price at open
- Highest price in session
- Lowest price in session
- Price at session close
- Volume
- Adjust price at session close

Let's create a simple program that loads the content of the file, executes some simple preprocessing functions, and creates a simple model. We selected the CSCO stock price between January 1, 2012 and December 1, 2013 as our data input.
Let's consider two variables, price and volume, as illustrated by the following screenshot. The top graph displays the variation of the price of Cisco stock over time and the bottom bar chart represents the daily trading volume on Cisco stock over time:

![Price-Volume action for the Cisco stock](image)

### Loading the dataset

The first step is loading the dataset from a local file. Typically, large datasets are loaded from a database or distributed filesystem such as **Hadoop Distributed File System (HDFS)**, as shown here:

```scala
def load(fileName: String): Option[XYTSeries] = {
  val src = Source.fromFile(fileName)
  val fields = src.getLines.map(_.split(CSV_DELIM)).toArray //1
  val cols = fields.drop(1) //2
  val data = transform(cols)
  src.close //3
  Some(data)
}
```

The `transform` method will be described in the next section.

The data file is extracted through an invocation of the `Source.fromFile` static method, and then the fields are extracted through a map (line 1). The header (first) row is removed with a call to `drop` (line 2).

#### Data extraction

The `Source.fromFile.getLines.map` invocation pipeline method returns an iterator, which needs to be converted into an array to store the information into memory.
The file has to be closed to avoid leaking of the file handle (line 3).

Code readability
A long pipeline of Scala high-order methods make the code and underlying code quite difficult to read. It is recommended to break down long chains of method calls. The following code is an example of a long chain of method calls:

```scala
val cols = Source.fromFile.getLines.map(_._split(CSV_DELIM).toArray.drop(1))
```

We can break down such method calls into several steps as follows:

```scala
val lines = Source.fromFile.getLines
val fields = lines.map(_._split(CSV_DELIM).toArray
val cols = fields.drop(1)
```

We strongly encourage you to consult the excellent guide *Effective Scala*, written by Marius Eriksen from Twitter. This is definitely a must read for any Scala developer [1:10].

Preprocessing the dataset
The next step is to normalize the data in the range [-0.5, 0.5] to be trained by the logistic binary classifier. It is time to introduce a non-sense statistics class.

Basic statistics
We select the computation of mean and standard deviation of the two time series as the first step of the preprocessing phase. The computation of these statistics can be implemented by the reduce methods `reduceLeft` and `foldLeft`:

```scala
val mean = price.reduceLeft(_ + _)/price.size
val s2 = price.foldLeft(0.0)((s,x) =>s+(x-mean)*(x-mean))
val stdDev = Math.sqrt(s2/(price.size-1) )
```

However, this implementation has one major drawback: the dataset (price in this example) has to be traversed for each method (mean, stdDev, min, max, and so on).

One of the solutions is to create a class that computes the counters and the statistics on demand using, once again, the lazy values:

```scala
class Stats[T <% Double](private values: DVector[T]) {
  class _Stats(var minValue: Double, var maxValue: Double, var sum: Double, var sumSqr: Double)
  val stats = {
    val _stats = new _Stats(Double.MaxValue, Double.MinValue, 0.0, 0.0)
    
    val mean = values.reduceLeft(_ + _)/values.size
    val s2 = values.foldLeft(0.0)((s,x) =>s+(x-mean)*(x-mean))
    val stdDev = Math.sqrt(s2/(values.size-1) )
  }
}
```

```scala
class Stats[T <% Double](private values: DVector[T]) {
  class _Stats(var minValue: Double, var maxValue: Double, var sum: Double, var sumSqr: Double)
  val stats = {
    val _stats = new _Stats(Double.MaxValue, Double.MinValue, 0.0, 0.0)
    
    val mean = values.reduceLeft(_ + _)/values.size
    val s2 = values.foldLeft(0.0)((s,x) =>s+(x-mean)*(x-mean))
    val stdDev = Math.sqrt(s2/(values.size-1) )
  }
}
```
values.foreach(x => {
    if(x < _stats.minValue) x else _stats.minValue
    if(x > _stats.maxValue) x else _stats.maxValue
    _stats.sum + x
    _stats.sumSqr + x*x
})
_stats

lazy val mean = _stats.sum/values.size
lazy val variance = (_stats.sumSqr - mean*mean*values.size)/(values.size-1)
lazy val stdDev = if(variance < ZERO_EPS) ZERO_EPS else Math.sqrt(variance)
lazy val min = _stats.minValue
lazy val max = _stats.mazValue
}

We made the statistics object generic by using the view bounds T <% Double, which assumes a conversion from type T to Double. By defining the statistics as tuple counters (minimum value, maximum value, sum of values, and sum of square values) and folding these values into a statistics object, we limit the number of invocations of the foldLeft reducer method to 1, and therefore, avoid the recomputation of these statistics for the existing dataset each time new data is added.

The code illustrates the use and benefit of lazy values in Scala. The mean is computed only if and when needed.

**Normalization and Gauss distribution**

Statistics are usually used to normalize data into a probability value [0, 1] as required by most classification or clustering algorithms. It is logical to add the normalization method to the Stats class, as we have already extracted the min and max values:

```scala
def normalize: DblVector = {
    val range = max - min; values.map(x => (x - min)/range)
}
```

The same approach is used to compute the multivariate normal distribution:

```scala
def gauss: DblVector =
values.map(x => {
    val y = x - mean
    INV_SQRT_2PI/stdDev*Math.exp(-0.5*y*y/stdDev))
```
The price action chart has a very interesting characteristic. At a closer look, a sudden change in price and increase in volume occurs about every three months or so. Experienced investors will undoubtedly recognize that those price-volume patterns are related to the release of quarterly earnings of Cisco. Such regular but unpredictable patterns can be a source of concern or opportunity if risk can be managed. The strong reaction of the stock price to the release of corporate earnings may scare some long-term investors while enticing day traders.

The following graph visualizes the potential correlation between sudden price change (volatility) and heavy trading volume:

![Correlation price-volume action for the Cisco stock](image)

Let’s try to correlate the volatility of the stock price with volume. For the sake of this exercise, we define the volatility as the maximum variation of the stock price within each trading session: the relative difference between the highest price during the trading session and the lowest price during the session.

The YahooFinancials enumeration extracts historical stock prices and session volume from a CSV file. For example, the volatility is extracted from the CSV fields of each line in the CSV file as follows:

```scala
object YahooFinancials extends Enumeration {
  type YahooFinancials = Value
  val DATE, OPEN, HIGH, LOW, CLOSE, VOLUME, ADJ_CLOSE = Value
```
val volatility = (fs: Array[String]) => fs(HIGH.id).toDouble-fs(LOW.id).toDouble

...}

The transform method uses the YahooFinancials enumeration to generate the input data for the model:

def transform(cols: Array[Array[String]]): XYTSeries = {
    val volatility = Stats[Double](cols.map(YahooFinancials.
        volatility)).normalize
    val volume = Stats[Double](cols.map(YahooFinancials.volume)
        ).normalize
    volatility.zip(volume)
}

The volatility and volume data is normalized using the Stats.normalize method defined earlier.

Plotting data

Although charting is not the primary goal of this book, we thought that you will benefit from a brief introduction to JFreeChart. The skeleton code to generate a scatter plot is rather simple. The most relevant code is the transformation of the XYTSeries into graphical JFreeChart's XYSeries:

val xLegend = "Session Volatility"
val yLegend = "Session Volume"
def display(xy: XYTSeries, w: Int, h : Int): Unit = {
    val series = new XYSeries("CSCO 2012-2013 Stock")
    xy.foreach( x => series.add( x._1,x._2))
    val seriesCollection = new XYSeriesCollection
    seriesCollection.addSeries(series)
    ... // plot rendering code
    val chart = ChartFactory.createScatterPlot(xLegend, xLegend, yLegend, seriesCollection, PlotOrientation.VERTICAL, true, false, false)
    createFrame("Logistic Regression", chart)
}
Visualization

The JFreeChart library is introduced as a robust charting tool. The visualization of the results of a computation is beyond the scope of this book. The code related to plots and charts is omitted from the book in order to keep the code snippets concise and dedicated to machine learning. In a few occasions, output data is formatted as a CSV file to be simply imported into a spreadsheet.

Here is an example of a plot using the `ScatterPlot.display` method:

```scala
val plot = new ScatterPlot(("CSCO 2012-2013", "Session High - Low", "Session Volume"), new BlackPlotTheme)
plot.display(volatility_vol.filter(_._1 < 0.5), 250, 340)
```

![Scatter plot of volatility and volume for the Cisco stock](image)

There is a level of correlation between session volume and session volatility. We can use this information to classify trading sessions by their volatility.

Creating a model (learning)

The objective of the training is to build a model that can discriminate between volatile and nonvolatile trading sessions. For the sake of the exercise, session volatility has been defined as session price high and session price low coupled with heavy trading volume, which constitute the two parameters of the model.
Logistic regression is commonly used in statistics inference. The following implementation of the binary logistic regression classifier exposes a single method, classify, to comply with our desire to reduce the complexity and life cycle of objects. The model parameters, weights, are computed during training when the LogBinRegression class/model is instantiated. As mentioned earlier, the sections of the code nonessential to the understanding of the algorithm are omitted:

```scala
class LogBinRegression(val labels: DVector[(XY, Double)], val maxIters: Int, val eta: Double, val eps: Double) {
  val dim = 3
  val weights = train

  def classify(xy: XY): Option[(Boolean, Double)] = {
    if(weights != None) {
      val likelihood = sigmoid(w(0) + xy._1*w(1) + xy._2*w(2))
      Some(likelihood > 0.5, likelihood)
    } else None
  }
}
```

The training method, train, consists of iterating through the computation of the weight using a simple descent gradient. The method computes the weights and returns an option, so the model is either trained and ready for runtime classification or nonexistent (None):

```scala
def train: Option[DblVector] = {
  val w = Array.fill(dim) { x => Random.nextDouble-1.0 }
  Range(0, maxIters).find(_ => {
    val deltaW = labels.foldLeft(Array.fill(dim)(0.0))((dw, lbl) => {
      val y = sigmoid(w(0) + w(1)*lbl._1._1 + w(2)*lbl._1._2)
      dw.map(dx => dx + (lbl._2 - y)*(lbl._1._1 + lbl._1._2))
    })
    val nextW = Array.fill(dim)(0.0)
    .zipWithIndex
    .map(nw => w(nw._2)+eta*deltaW(nw._2))
    val diff = Math.abs(nextW.sum - w.sum)
    nextW.copyToArray(w);
    diff < eps
  }) match {
    case Some(iters) => Some(w)
    case None => { … }
  }
}
```

```scala
def sigmoid(x: Double):Double = 1.0/(1.0 + Math.exp(-x))
```
The iteration is encapsulated in the Scala `find` method that exists if the algorithm converges \((\text{diff} < \epsilon)\). The model parameters, `weights`, are set to `None` if the maximum number of iterations is reached.

The training method, `train`, iterates across the set of observations by computing the gradient between the predicted and observed values. In our simplistic approach, the gradient is computed as a linear function of the sigmoid of the sum of the product of the weight and training observations. As for any optimization problem, the initialization of the solution vector, `weights`, is critical. We choose to initialize the weight with random values, although in practice, you would use a more deterministic approach to initialize the model parameters.

In order to train the model, we need to label data. The process consists of tagging every trading session as `volatile` and `non volatile` according to the observations (relative session volatility and session volume). The labeling process is usually quite cumbersome; therefore, let's generate the label automatically. A trading session is considered volatile if a volatility and volume are both greater than 60 percent of the maximum relative volatility and volume:

\[
\text{val labels} = \text{volatilityVol.zip(volatilityVol.map(x =>if( x._1>0.3 && x._2>0.3) 1.0 else 0.0))}
\]

**Automated labeling**

Although quite convenient, automated creation of training labels is not without risk because it may mislabel singular observations. This technique is used in this test for convenience but it is not recommended unless a domain expert reviews the labels manually.

The model is created (trained) by a simple instantiation of the logistic binary classifier:

\[
\text{val logit} = \text{new LogBinRegression(labels, 300, 0.00005, 0.02)}
\]

The training run is configured with a maximum of 300 iterations, a gradient slope of 0.00005, and convergence criteria of 0.02.

**Classify the data**

Finally, the model can be tested with a new fresh dataset, not related to the training set:

<table>
<thead>
<tr>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
<th>Adj Close</th>
</tr>
</thead>
</table>
It is just a matter of executing the classification method (exceptions, conditions on method arguments, and returned values are omitted):

```scala
val testData = load("resources/data/chap1/CSCO2.csv")
logit.classify(testData(0)) match {
  case Some(topCategory) => Display.show(topCategory)
  case None => { ... }
}
logit.classify(testData(1)) match {
  case Some(topCategory) => Display.show(topCategory)
  case None => { ... }
}
```

The result of the classification is (true, 0.516) for the first sample and (false, 0.1180) for the second sample.

**Validation**
The simple classification, in this test case, is provided for illustrating the runtime application of the model. It does not constitute a validation of the model by any stretch of imagination. The next chapter digs into validation metrics and methodology.

**Summary**
We hope you enjoyed this introduction to machine learning and how to leverage your existing skills in Scala programming to create a simple regression program to predict stock price/volume action. Here are the highlights of this introductory chapter:

- From monadic composition and high-order collection methods for parallelization to configurability to reusability patterns, Scala is the perfect fit to implement and leverage data mining and machine learning algorithms for large-scale projects
- There are many steps to create and apply a machine learning model
- The implementation of the logistic binary classifier presented as part of the test case is simple enough to encourage you to learn how to write and apply more advanced machine learning algorithms

To the delight of Scala programming aficionados, the next chapter will dig deeper into building a flexible workflow by leveraging traits and dependency injection.
Where to buy this book

You can buy Scala for Machine Learning from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.