Introduction to Data Science and Machine Learning Using Spark

When the facts change, I change my mind.

John Maynard Keynes, British economist

In This Chapter:

- Introduction to R and SparkR
- Statistical functions and predictive models with SparkR
- Machine learning and Spark using Spark MLlib
- Notebooks with Spark

Machine learning and data science are exciting areas of computer science. As more storage and computing capability become available at lower costs, we can harness the true power of machine learning to help make better decisions. Spark and the wider Big Data ecosystem are great enablers and accelerators of this capability.

Spark and R

R is a powerful programming language and software environment for statistical computing, visual analytics, and predictive modeling. For data analysts,

statisticians, mathematicians, and data scientists already using R, Spark provides a scalable runtime engine for R: *SparkR*. For developers and analysts new to R, this chapter provides an introduction and shows how R can seamlessly integrate with Spark.

Introduction to R

R is an open source language and runtime environment for statistical computing and graphics, based on a language called S originally developed at Bell Labs in the late 1970s. The R language is widely used among statisticians, data analysts, and data scientists as a popular alternative to SAS, IBM SPSS, and other similar commercial software packages.

Native R is primarily written in C and compiled into machine code for the targeted platform. Precompiled binary versions are available for various operating systems, including Linux, macOS, and Windows. R programs can run from the command line as batch scripts or through the interactive shell. In addition, there are several graphical user interfaces available for R, including desktop applications and web-based interfaces, discussed later in this chapter. R's graphical rendering capabilities combine its mathematical modeling strength with the capability to produce visual statistics and analytics, as shown in Figure 8.1.
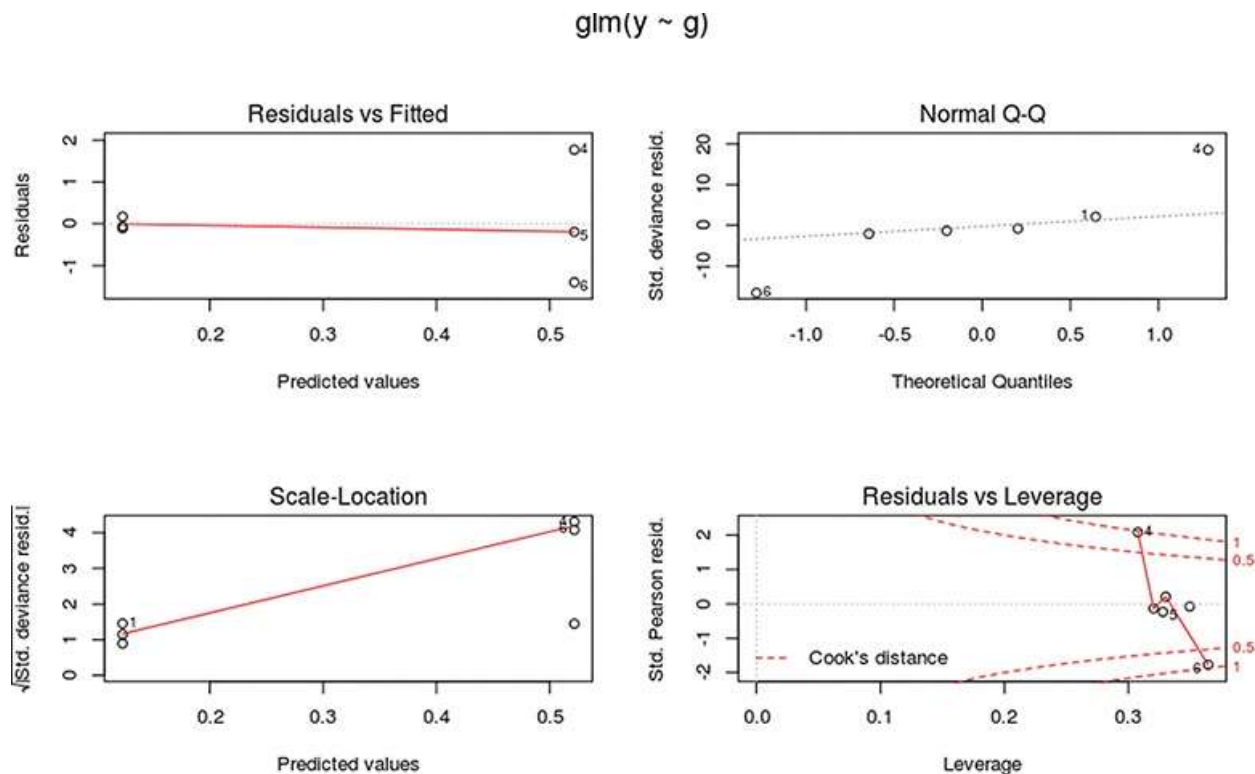
Figure 8.1 Visual statistics and analytics in R.

R is a case-sensitive, interpreted language. R code is generally easy to spot by its non-conventional assignment operator (<-), as in the following example:

```
y <- x + 2
```

The following sections look at some of the building blocks of the R programming language.

R Basic Datatypes

R has several basic datatypes used to represent data elements held within the data structures. The main R datatypes used to represent data elements are summarized in Table 8.1.

Table 8.1 **Primary R Datatypes**

Datatype	Description	Example
Logical	Boolean value	TRUE, FALSE
Numeric	Double-precision numeric value	3, 1.4, 1.1e+10

`Integer`	32-bit signed integer	`3L, 384L`
`Character`	String value of arbitrary length	`'spark', '123', 'A'`

Integers can cause some confusion, especially because the `L` notation declares them in R. R integer types are a subset of the `Numeric` type. At the time of this writing, an R integer is a 32-bit (or 4-byte) signed integer, in contrast to a long type in most programming languages, which is a 64-bit or 8-byte signed integer, often declared using the *n*`L` syntax. For conventional long numbers, you typically use the `Numeric` type in R, which is a double-precision number capable of storing much larger numbers.

Listing 8.1 shows the use of system functions to display the maximum values for `Integer` and `Numeric` (double) types in R.

Listing 8.1 **Max Values for R `Integer` and `Numeric` (Double) Types**

Click here to view code image

```
> .Machine$integer.max
[1] 2147483647
> .Machine$double.xmax
[1] 1.797693e+308
```

There are also more obscure types for complex numbers and raw byte arrays. However, they are beyond the scope of this book.

Data Structures in R

R's data model is based on the concept of vectors. A *vector* is a sequence of data elements of the same type. The members of a vector are called *components*. More complex structures are built on vectors, such as matrices, which are two-dimensional data structures with data elements of the same type, and arrays, which are multidimensional objects (with more than two dimensions).

Importantly, R has an additional data structure called a *data frame*. Data frames in R are conceptually similar to DataFrames in Spark SQL. In fact, the Spark SQL DataFrame was inspired by the data frame construct in R. R data frames are two-dimensional data structures where columns may be of different types, but all

the values within a column are of the same type. Basically, this is tantamount to a table object in a relational database.

Figure 8.2 shows a representation of the basic data structures in R with sample data.

Vector

3	8	4

ONE DIMENSIONAL ARRAY,
COMPONENTS OF THE SAME MODE (EG
NUMERIC, STRINGS, ETC)

TWO DIMENSIONAL ARRAY,
COMPONENTS OF THE SAME MODE

MULTI DIMENSIONAL ARRAY,
COMPONENTS OF THE SAME MODE

Matrix

3	8	4
1	9	0
6	4	2

TWO DIMENSIONAL ARRAY,
COLUMNS MAY HAVE
DIFFERENT MODES

3	8	4
1	9	0
6	4	2

Array

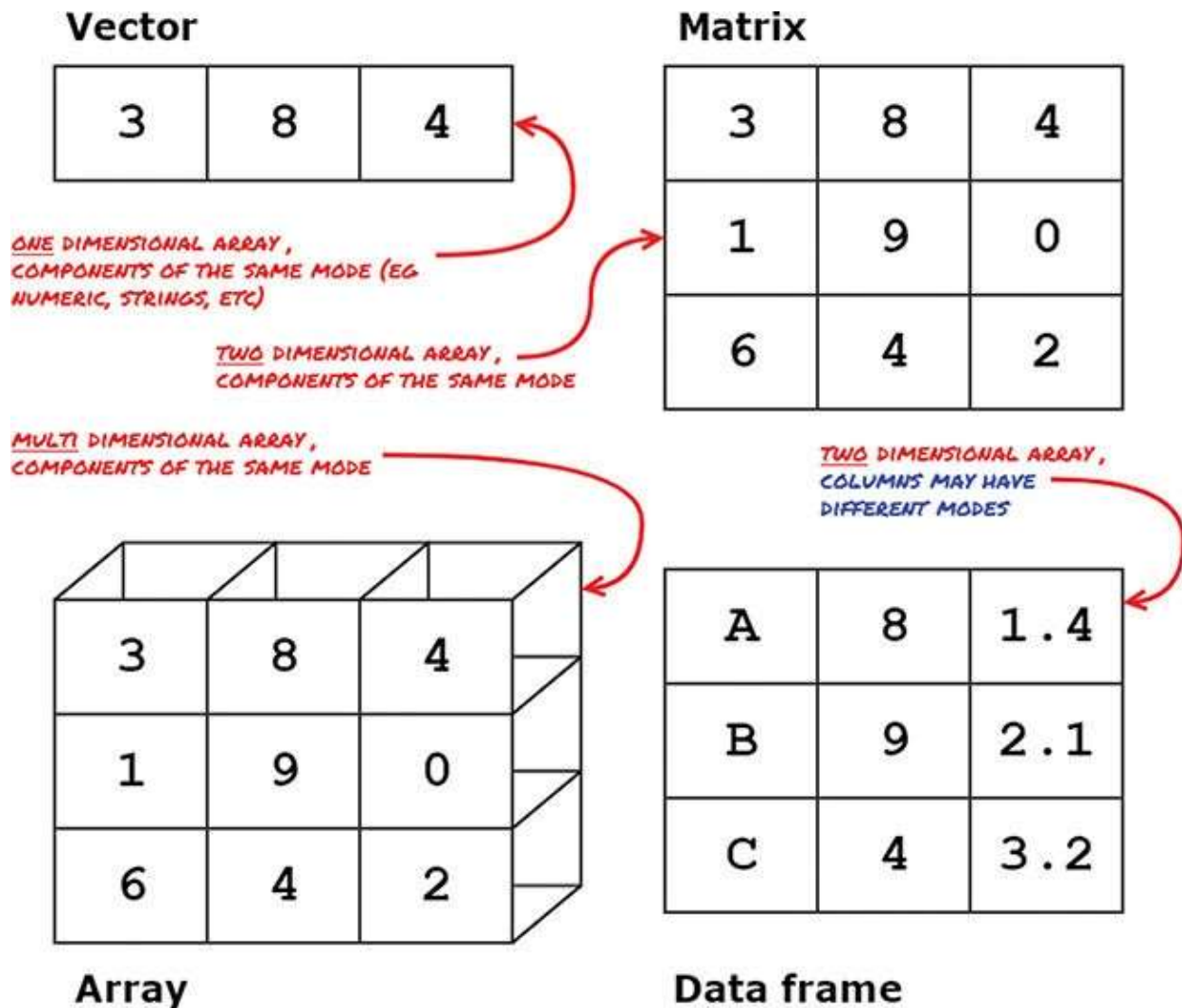| A	8	1.4
B	9	2.1
C	4	3.2

Data frame

Figure 8.2 Data structures in R.

R has no concept of scalar values, akin to primitive types available in most common programming languages. The equivalent of a scalar variable is represented as a vector with a length of 1 in R. Consider Listing 8.2. If you want to create a simple variable, var, with a scalar-like assignment equal to 1, var is created as a vector with one component.

Listing 8.2 **A Simple R Vector**

```
> var <- 1
> var
[1] 1
```

A multivalued vector is created using the combine, or c(), function, as demonstrated in Listing 8.3.

Listing 8.3 **Using the c() Function to Create an R Vector**

```
> vec <- c(1,2,3)
> vec
[1] 1 2 3
```

A two-dimensional matrix is created using the matrix command. An example of creating a 333 matrix using the c() function is shown in Listing 8.4. By default, elements fill in by column. However, you can specify byrow=TRUE to fill in a matrix row by row.

Listing 8.4 **Creating an R Matrix**

```
> mat = matrix(
+      c(1,2,3,4,5,6,7,8,9),
+      nrow=3,
+      ncol=3)
> mat
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Elements of a matrix are accessible using subscripts and brackets. For instance, x[i,] is a reference to the *i*th row of the matrix x; x[,j] is a reference to the *j*th column of a matrix x; and x[i,j] refers to the intersection of the *i*th row

and *j*th column. An example of this is shown in Listing 8.5.

Listing 8.5 **Accessing Data Elements in an R Matrix**

```
> mat[1,]
[1] 1 4 7
> mat[,1]
[1] 1 2 3
> mat[3,3]
[1] 9
```

Creating and Accessing R Data Frames

Arguably, the most important data structure in R is the data frame. Think of data frames in R as data tables, with rows and columns, where columns can be of mixed types. The important difference between data frames and other data structures in R is that data frames allow for column and row-specific operations such as projections and filtering. The R data frame is the primary data structure used to interact with SparkR, as discussed shortly.

You create data frames from column vectors by using the `data.frame` function, as shown in Listing 8.6.

Listing 8.6 **Creating an R Data Frame from Column Vectors**

```
> col1 = c("A", "B", "C")
> col2 = c(8,9,4)
> col3 = c(1.4,2.1,3.2)
> df = data.frame(col1,col2,col3)
> df    col1 col2 col3
1    A    8   1.4
2    B    9   2.1
3    C    4   3.2
```

You can also create R data frames from external sources by using the `read` command; `read` supports different sources, summarized in Table 8.2.

Table 8.2 **Functions to Create an R Data Frame from an External Source**

Function	Description
`read.table()`	Reads a new line–terminated file with fields delimited by whitespace in table format and creates a data frame.
`read.csv()`	Same as `read.table()` using commas (,) as field separators.
`read.fwf()`	Reads a table of fixed-width formatted data, a common extract format for many mainframe and other legacy systems.

There are several other SparkR-specific methods for creating distributed data frames in SparkR from external sources, as discussed shortly.

In R, several methods can be used to inspect and access data from within a data frame. Some of these are demonstrated in Listing 8.7, using the sample data frame created in Listing 8.6.

Listing 8.7 **Accessing and Inspecting Data in R Data Frames**

Click here to view code image

```
> # get element in row 1, col 2
> df[1,2] [1] 8
> # get number of cols in the dataframe
> ncol(df) [1] 3
> # get number of rows in the dataframe
> nrow(df) [1] 3
> # display first row from the dataframe
> head(df, 1)
  col1 col2 col3
1    A    8  1.4
```

R Functions and Packages

Most R programs involve manipulating data elements or data structures using functions. R, like most other languages, includes many common built-in functions. Table 8.3 provides a sampling of the available built-in functions.

Table 8.3 **Sample Built-in R Functions**

Category	Examples of Functions
Numeric	`abs()`, `sqrt()`, `ceiling()`, `floor()`, `log()`, `exp()`
Character	`substr()`, `grep()`, `strsplit()`, `toupper()`
Statistical	`mean()`, `sd()`, `median()`, `quantile()`, `sum()`, `min()`
Probability	`dnorm()`, `pnorm()`, `qnorm()`, `dpois()`, `ppois()`

The true power of R, however, is in libraries and packages written for R. *Packages* are collections of R functions, data, and compiled code in a well-defined and well-described format. The directory on the system where the packages reside is the *library*.

R ships with a standard set of packages, including several sample datasets, which we will look at shortly. You can also obtain custom R packages from a publicly available collection of packages from an R user community called *CRAN*. Find more information about the R packages available from CRAN at https://cran.r-project.org/.

If you cannot find a built-in function, an included package, or a CRAN package to do what you need, you can author your own packages.

You install by using the `R CMD INSTALL <package>` command on the system running the R program. After a package is installed, you can load into the current R session by using the `library(<package>)` command.

You can use the `library()` function with no arguments to view all the packages loaded and available in the current R session, as shown in Listing 8.8.

Listing 8.8 **Listing R Packages Installed and Available in an R Session**

Click here to view code image

```
> library()
Packages in library '/opt/spark/R/lib':

SparkR                  R Frontend for Apache Spark
```

```
Packages in library '/usr/lib/R/library':

base                    The R Base Package
boot                    Bootstrap Functions (Originally by Angelo Canty
                        for S)
class                   Functions for Classification
cluster                 "Finding Groups in Data": Cluster Analysis
                        Extended Rousseeuw et al.
codetools               Code Analysis Tools for R
compiler                The R Compiler Package
datasets                The R Datasets Package
foreign                 Read Data Stored by Minitab, S, SAS, SPSS,
                        Stata, Systat, Weka, dBase, ...
graphics                The R Graphics Package
...
```

As you can see in Listing 8.8, SparkR itself is an R package, as discussed in the next section.

Using Spark with R

The SparkR package for R provides an interface to access Spark from R, including the implementation of distributed data frames and large-scale statistical analysis, probability, and predictive modeling operations. SparkR comes with the Spark release. The package library is available in `$SPARK_HOME/R/lib/SparkR`. SparkR provides an R programming environment that enables R programmers to use Spark as a processing engine. Specific documentation about the SparkR API is available at https://spark.apache.org/docs/latest/api/R/index.html.

Accessing SparkR

Using the `sparkR` shell is the easiest way to get started with Spark and R. The command to launch the `sparkR` shell is `sparkR`, which is available in the `bin` directory of your Spark installation (the same directory as the other interactive shells, including `pyspark`, `spark-sql`, and `beeline`); `sparkR` starts an R session using the SparkR package with the Spark environment defaults for the specific system, such as `spark.master` and `spark.driver.memory`. Figure 8.3 shows an example of the `sparkR` shell.

Figure 8.3 The `sparkR` shell.

Notice that, as with `pyspark`, a `SparkSession` object named `spark` is created automatically. Likewise, a `SparkContext` is available as `sc`. The `SparkContext` and `SparkSession` objects are required as entry points to connect your R program to a Spark cluster and to be able to use data frames.

You can also use the `sparkR` command to run R programs in batch mode, using `spark-submit`, which recognizes an R program by its file extension (`.R`). Given an R program named `helloworld.R`, Figure 8.4 demonstrates how to run the program in batch mode, using `spark-submit`.

Figure 8.4 Running R programs in batch mode by using `sparkR`.

Creating Data Frames in SparkR

You can create SparkR data frames in a number of ways.

You can easily convert native R data frames into distributed data frames in SparkR. To demonstrate this, you can use the built-in R dataset `mtcars`, which consists of data extracted from the 1974 issues of the American magazine *Motor Trend,* including fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–1974 models).

> ### The R `datasets` Package
>
> One of the packages included with R is the `datasets` package. This package includes more than 100 diverse datasets from worldwide contributors, ranging from airline passenger numbers to air quality measurements to road casualties and violent crime rates. The `datasets` package also includes the famous Edgar Anderson's Iris Data dataset, which provides the measurements of sepal and petal length and width for 50 flowers from three species of irises—the "Hello, World" of data mining. You can view a complete list of the sample R datasets available in the `datasets` package by entering the following in your R or `sparkR` interactive shell:
>
> ```
> > library(help = "datasets")
> ```

The `mtcars` sample dataset is an R data frame with 32 observations on 11

variables. In Listing 8.9, using a `sparkR` session, the `mtcars` sample dataset is loaded into an R data frame named `df`, and then the `nrow()`, `ncol()`, and `head()` functions inspect the data frame.

Listing 8.9 `mtcars` Data Frame in R

Click here to view code image

```
> r_df <- mtcars
> nrow(r_df)
[1] 32
> ncol(r_df)
[1] 11
> head(r_df, 2)
              mpg cyl disp  hp  drat    wt  qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4
```

Note that because R is a scientific and modeling language, the data terminology used to refer to elements and constructs has an experimental science and mathematical modeling context. For instance, in the sample `mtcars` datasets, rows are *observations*, and fields within rows representing columns are *variables*.

The R data frame, `r_df`, created in Listing 8.9 can help create a SparkR data frame using the `createDataFrame()` SparkR API method, as demonstrated in Listing 8.10.

Listing 8.10 **Creating a SparkR Data Frame from an R Data Frame**

Click here to view code image

```
> spark_df <- createDataFrame(r_df)
> spark_df
SparkDataFrame[mpg:double, cyl:double, disp:double, hp:double,
drat:double, wt:double, qsec:double, vs:double, am:double, gear:double,
carb:double]
```

Another common requirement is to create SparkR data frames from comma-

separated value (CSV) files. The simplest method for loading a SparkR data frame from a CSV file is to use the SparkR `read.df()` method, as shown in Listing 8.11.

Listing 8.11 **Creating a SparkR Data Frame from a CSV File**

Click here to view code image

```
> csvPath <-
'file:///usr/lib/spark/examples/src/main/resources/people.txt'
> df <- read.df(csvPath, 'csv', header = 'false', inferSchema = 'true')
> head(df)
      _c0 _c1
1 Michael  29
2    Andy  30
3  Justin  19
```

The approach shown in Listing 8.11 results in an inferred schema for the resultant data frame. You can also explicitly define the schema for data in a CSV file by creating a schema object and supplying it to the `schema` argument in the `read.df()` method, as demonstrated in Listing 8.12.

Listing 8.12 **Defining the Schema for a SparkR Data Frame**

Click here to view code image

```
> csvPath <-
'file:///usr/lib/spark/examples/src/main/resources/people.txt'
> people_schema <- structType(structField("Name", "string"),
+ structField("age", "double"))
> df <- read.df(csvPath, 'csv', header = 'false', schema =
people_schema)
> head(df)
     Name age
1 Michael  29
2    Andy  30
3  Justin  19
```

There are also purpose-built functions in the SparkR API to create SparkR data frames from other common Spark SQL external data sources, such as

`read.parquet()` and `read.json()`.

You can also create SparkR data frames from Hive tables. The `sparkR.session()` function creates a connection to the configured Hive metastore; once this connection is available within a `sparkR` session, the `sql()` function in R can populate a SparkR data frame with the results of a Hive query. The `sql()` function can also be used to execute any Spark SQL statement, such as querying views and tables directly. Listing 8.13 shows an example of creating a SparkR data frame from a table in Hive.

Listing 8.13 **Creating a SparkR Data Frame from a Hive Table**

Click here to view code image

```
> sparkR.session()
> results <- sql("FROM stations SELECT station_id, lat, long")
  station_id       lat        long
1          2 37.32973 -121.9018
2          3 37.33070 -121.8890
3          4 37.33399 -121.8949
4          5 37.33141 -121.8932
5          6 37.33672 -121.8941
6          7 37.33380 -121.8869
```

After creating a SparkR data frame, you can reference columns by using the `<dataframe>$<column_name>` syntax. An example of this is shown in Listing 8.14.

Listing 8.14 **Accessing Columns in a SparkR Data Frame**

Click here to view code image

```
> head(filter(results, results$station_id > 10.0), 2)
  station_id       lat        long
1         11 37.33588 -121.8857
2         12 37.33281 -121.8839
```

SparkR and Predictive Analytics

Predictive analytics at scale is one of the key functional drivers of Big Data

platforms. Retailers want to better understand customers and predict their buying behavior and propensity, credit providers want to assess risk involved with products and applicants, utilities companies want to predict and preempt customer churn, and so on.

The primary cases for using SparkR, like R, are performing statistical analysis of data and building predictive models from observations and variables. SparkR provides the ability to do this at a much greater scale than R itself because it capitalizes on Spark's powerful distributed computing framework.

Introduction to Data Mining and Predictive Modeling

If you're a data scientist, feel free to skip the next few paragraphs. If you're not a data scientist, the next few paragraphs will give you a soft introduction to data science and how the processes and methods data scientists use can be extended to leverage Spark.

Data mining is the process of discovering patterns within data that can be combined to predict an outcome. The process of discovering the inputs to these predictions is called *predictive modeling*. Predictive modeling usually falls into one of two categories: supervised learning or unsupervised learning.

Supervised learning observations receive labels such as "spam," "notspam," and "defaulted." This label is then used when observing patterns in the associated data to determine the influence that these patterns have on the outcome (the label). You "teach" the system what a desirable (or undesirable) outcome looks like—hence the name *supervised*.

In contrast, *unsupervised learning* does not involve classified observations. Typically, unsupervised learning involves identifying similarity between observations or clustering instances, which can also facilitate identifying outliers or detecting anomalies. In either case, the process of building a model typically follows the workflow pictured in Figure 8.5.
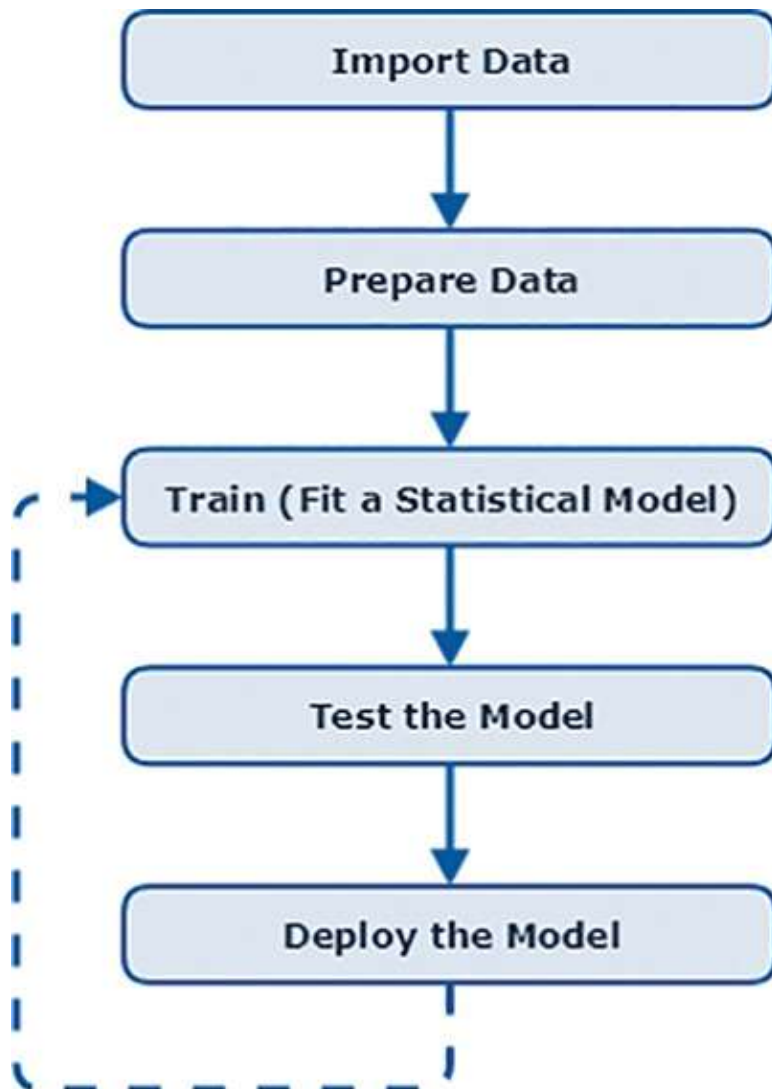
Figure 8.5 Steps involved in predictive modeling.

In this book, we have looked at how to import data and spent a considerable amount of time on the preparation and curation of data. The following process is what R is exceptionally good at:

- Fitting a statistical model to the data (or training the model)
- Testing the model against a known set of data not used in the training phase
- Deploying the model to predict outcomes for new data observations

Linear Regression

One of the simplest forms of a predictive model is the linear regression model.

Without going into the mathematics behind this type of model, a linear regression model assigns coefficients (weights) to variables and creates a generalized linear function, the result of which is a prediction.

After being trained, tested, and deployed, the regression function performs against new data (observations) to predict outcomes, given the known variables. The general linear model is defined as follows:

$$y_i = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p + \varepsilon$$

In this model, y_i is the response (or predicted outcome), b represents the coefficients or weight, and represents error.

R and SparkR include the function `glm()`, which creates a generalized linear model; `glm()` builds a model from observations in a data frame using an input formula in the following form:

$$y \sim x_1 \, 1 \, x_2 \, \ldots$$

Where y is the response, and x_1 and x_2 are continuous or categorical variables. Listing 8.15 shows the use of the `glm()` function in SparkR to create a generalized linear model to predict sepal length from the `iris` dataset. The `summary()` function can describe the model after it is built.

Listing 8.15 **Building a Generalized Linear Model with SparkR**

Click here to view code image

```
> # prepare data frame and build model
> iris_df <- createDataFrame(iris)
> training <- sample(iris_df, FALSE, 0.8)
> test <- sample(iris_df, FALSE, 0.2)
> model <- glm(Sepal_Length ~ Sepal_Width + Species, data = training,
family = "gaussian")
> summary(model)
Deviance Residuals:
(Note: These are approximate quantiles with relative error <= 0.01)
    Min        1Q    Median        3Q       Max
-1.31166  -0.25586  -0.05586   0.17351   1.40303

 Coefficients:
                 Estimate  Std. Error  t value    Pr(>|t|)
(Intercept)       2.08211     0.43376   4.8001  4.7693e-06
```

```
Sepal_Width          0.85317     0.12417   6.8708  3.3820e-10
Species_versicolor   1.47019     0.12693  11.5830  0.0000e+00
Species_virginica    1.99662     0.11553  17.2827  0.0000e+00
(Dispersion parameter for gaussian family taken to be 0.1969856)
Null deviance: 82.826  on 119  degrees of freedom
Residual deviance: 22.850  on 116  degrees of freedom
AIC: 151.5 Number of Fisher Scoring iterations: 1
```

After you've built your model in SparkR, you can apply it to new data to make predictions by using the `predict()` function (see Listing 8.16).

Listing 8.16 **Using a GLM to Make Predictions on New Data**

Click here to view code image

```
> # predict new data
> predictions <- predict(model, test)
> head(select(predictions, "Sepal_Length", "prediction"))
  Sepal_Length prediction
1          5.1   5.068201
2          4.9   4.641617
3          4.7   4.812251
4          4.8   4.641617
5          4.3   4.641617
6          4.8   4.982885
```

Using SparkR with RStudio

So far, you have interacted with SparkR by using the `sparkR` shell interface. Although this exposes all the key functions in R for data manipulation, preparation, analysis, and modeling, it lacks the rich visualization capabilities of a desktop or browser-based interface.

RStudio is an open source Integrated Development Environment (IDE) for R. RStudio is available as a desktop application, RStudio Desktop, and as a server-based application, RStudio Server. RStudio Server enables clients to connect and interact with an R environment using a web browser. Figure 8.6 shows the RStudio client interface.

RStudio provides the full set of capabilities available from the command line

interface, including built-in functions and packages, as well as the capability to create and export publication-quality visual analytic outputs.

RStudio is easily configurable for using SparkR as its runtime engine for execution.
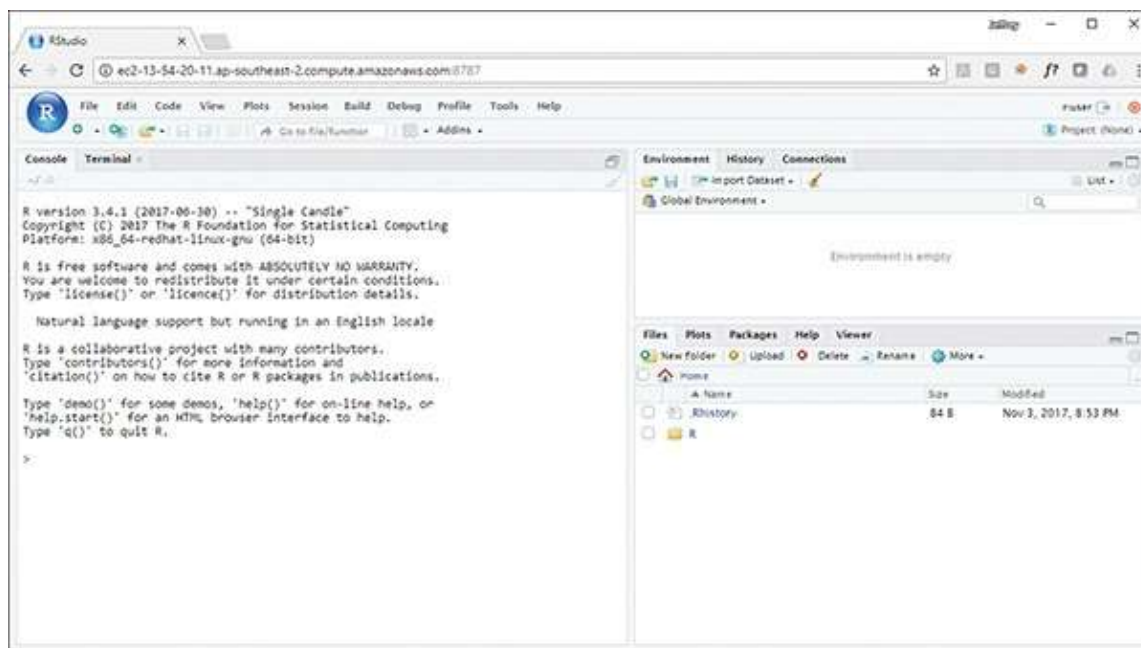

Figure 8.6 RStudio web interface.

Exercise: Using RStudio with SparkR

This exercise shows how to install RStudio alongside your Spark installation and configure RStudio to use SparkR as its processing engine. This example uses a Spark installation on a Red Hat/Centos system. RStudio is a compiled application with builds for various platforms. To obtain the specific build for your platform, go to www.rstudio.com/products/rstudio/download-server/ and follow these steps:

1. From your system, download and install your specific build of RStudio:

Click here to view code image

```
$ wget https://download2.rstudio.org/...x86_64.rpm
$ sudo yum install --nogpgcheck rstudio-server-rhel-....rpm
```

2. To ensure that RStudio is available on port 8787 of your server, go to http://<yourserver>:8787/.

3. Create a new R user:

```
$ sudo useradd -d /home/r-user -m r-user
$ sudo passwd r-user
```

R users require a home directory because R automatically saves the user's "workspace" to this directory. Note that you also need to create a home directory for the user in HDFS if you are running RStudio on a Hadoop cluster.

4. Log in to RStudio by using the `r-user` account created in step 3.

5. From the console window on the left side of the RStudio interface, at the R prompt, enter the following commands to load the `SparkR` package and initialize a SparkR session:

```
> Sys.setenv(SPARK_HOME = "/opt/spark")
> library(SparkR, lib.loc = c(file.path("/opt/spark/R/lib")))
> sparkR.session()
```

6. Test some simple visualizations using the built-in `iris` dataset by entering the following at the console prompt:

```
> hist(iris$Sepal.Length,xlim=c(4,8),col="blue",freq=FALSE)
> lines(density(iris$Sepal.Length))
```

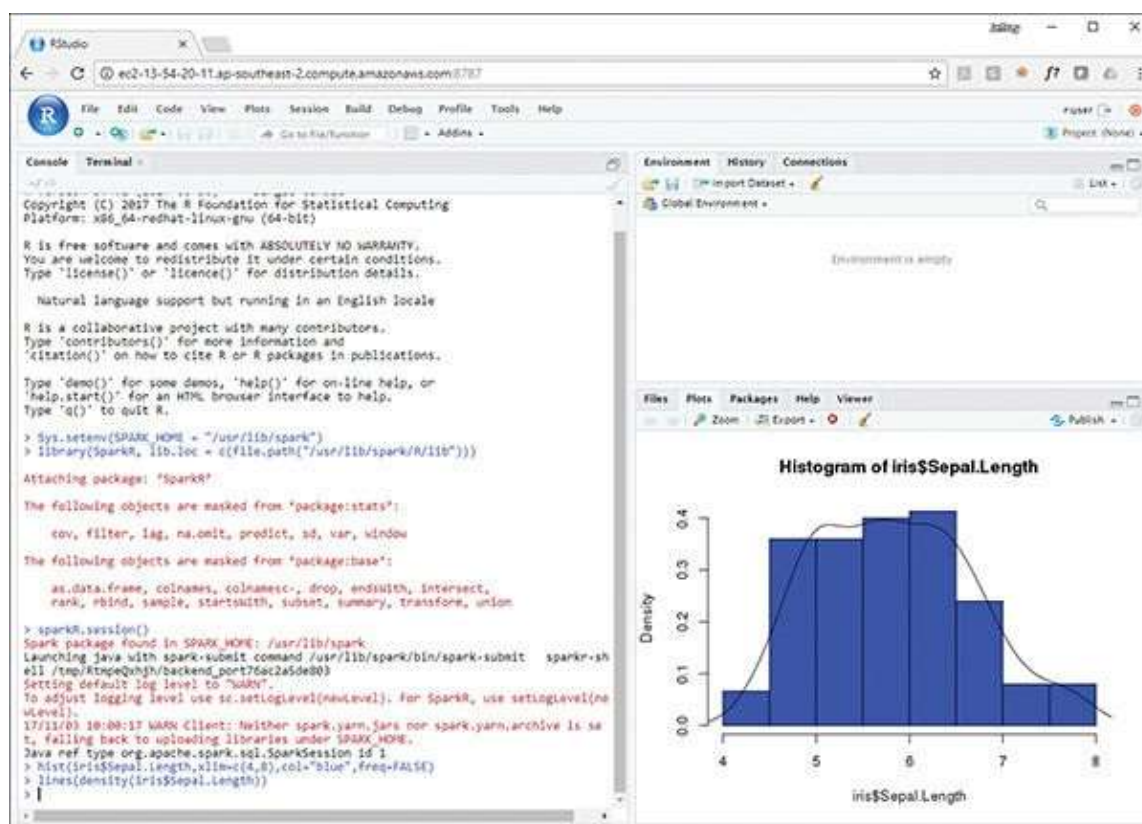In the Plots window, you should see the histogram shown in Figure 8.7.

Figure 8.7 Histogram

7. Try creating a SparkR data frame from one of the included R datasets. Recall that you can see information about available datasets by using the following command:

```
> library(help = "datasets")
```

Then use functions from the SparkR API to manipulate, analyze, or create and test a model from the data. Documentation for the SparkR API is available at https://spark.apache.org/docs/latest/api/R/index.html.

Machine Learning with Spark

Machine learning is the science of creating algorithms capable of learning based on the data provided to them. Common applications of machine learning are around every day, from recommendation engines to spam filters to fraud detection and much more. Machine learning is the process of automating data mining. Spark includes two purpose-built libraries, MLlib and ML, to make practical machine learning scalable, easy, and seamlessly integrated into Spark.

Machine Learning Primer

Machine learning is a specific discipline within the field of predictive analytics, which refers to programs that leverage the data they collect to influence the program's future behavior. In other words, the program "learns" from the data rather than relying on explicit instructions.

Machine learning is often associated with data at scale. As more data is observed in the learning process, the higher the accuracy of the model, or the better it is at making predictions.

You can see practical examples of machine learning in everyday life, including recommendation engines in ecommerce websites, optical character recognition, facial recognition, spam filtering, fraud detection, and so on.

Three primary techniques are used in machine learning:

- Classification
- Collaborative filtering
- Clustering

The following sections take a high-level look at each of these techniques.

Classification

Classification is a supervised learning technique that takes a set of data with known labels and learns how to label new data based on that information. Consider a spam filter on an email server that determines whether an incoming message should be classified as "spam" or "not spam." The classification algorithm trains itself by observing user behavior to discover what's classified as spam. Learning from this observed behavior, the algorithm classifies new mail accordingly. The classification process for this example is pictured in Figure 8.8.
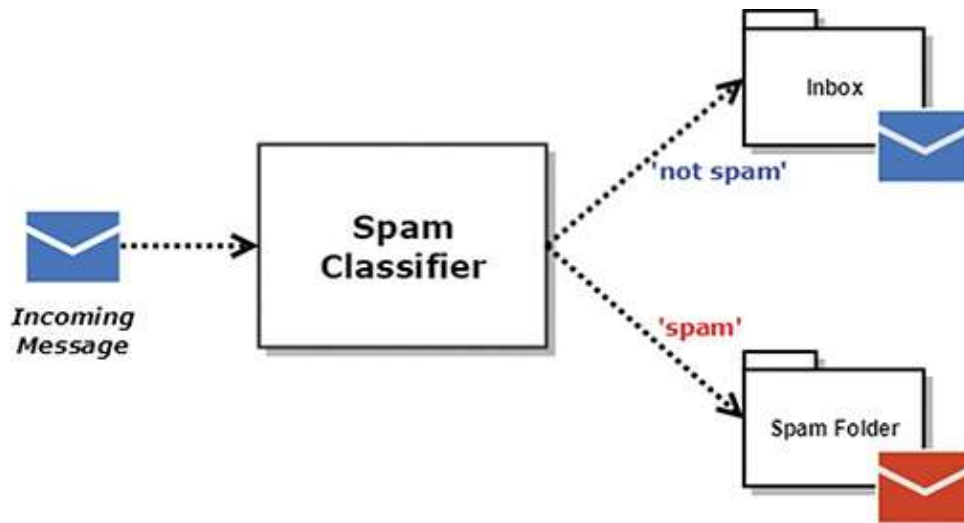
Figure 8.8 Classification of incoming email messages.

Classification techniques appear in a wide variety of applications across various domains, ranging from oncology, where a classifier may be trained to distinguish benign tumors from malignant tumors, to credit risk analysis, where a classifier may be trained to identify a customer at risk of defaulting on a credit product.

Collaborative Filtering

Collaborative filtering is a technique for making recommendations. It is commonly denoted by the "You might also like…" or similar sidebars or callouts on shopping websites. The algorithm processes large numbers of data observations to find entities with similar traits or characteristics and then makes recommendations or suggestions to newly observed entities based on the previous observations.

Collaborative filtering, unlike classification, is an unsupervised learning technique. And unlike with supervised learning, unsupervised learning algorithms can derive patterns in data without supplied labels.

Collaborative filtering is domain agnostic. It can be used in a wide variety of cases, from online retailing to streaming music and video services to travel sites to online gaming and more. Figure 8.9 depicts the process of collaborative filtering for the purpose of generating recommendations.
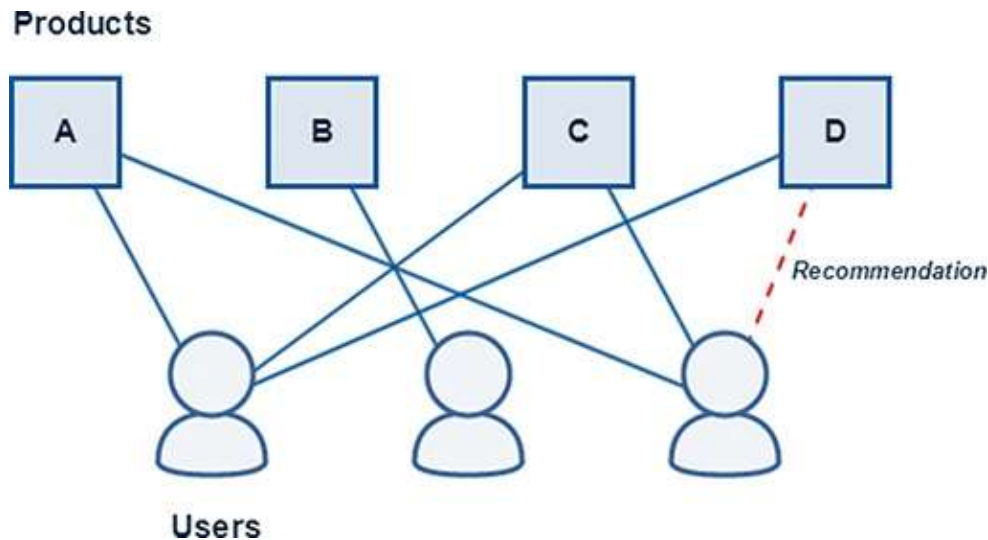
Products

Figure 8.9 Collaborative filtering.

Clustering

Clustering is the process of discovering structure within collections of data observations, especially where a formal structure is not obvious. Clustering algorithms discover, or "learn," what groupings naturally occur in the data provided to them.

Clustering is another example of an unsupervised learning technique often used for exploratory analysis. You can determine clusters in several ways, including by density, proximity, location, levels of connectivity, or size.

Some examples of clustering applications include the following:

- Market or customer segmentation

- Finding related news articles, tweets, or blog posts

- Image-recognition applications where clusters of pixels cohere into discernible objects

- Epidemiological studies, such as identifying "cancer clusters"

Figure 8.10 clearly shows three clusters when you look at the relationship between sepal length and sepal width in the iris dataset. The center of each cluster is the centroid. The centroid is a vector representing the mean of a variable for the observations in the cluster that are usable for approximating distances between clusters.
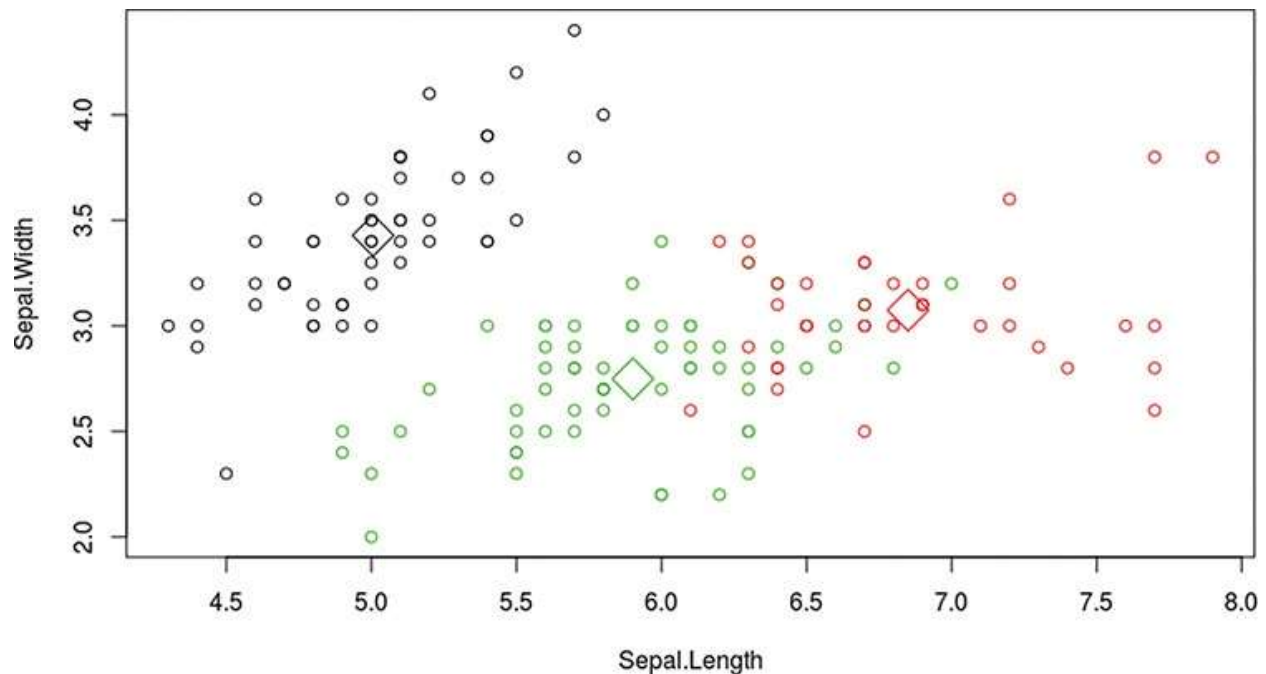
Figure 8.10 Clustering.

Features and Feature Extraction

In machine learning, a *feature* is a measurable attribute or characteristic of an observation. Variables for developing models are sourced from a pool of features. Examples of simple features for building a retail or financial services propensity or risk model are annual income, total amount spent in the past 12 months in a particular category, and a three-month moving-average credit card balance.

Often features don't present in the data itself; they derive from the data, historical data, or other available data sources. Moreover, features can be aggregated or summarized from the underlying data. Creating the set of features used by an algorithm in a machine learning program is the process of *feature extraction*. Selecting and extracting an appropriate set of features is as important as, if not more important than, algorithm selection or tuning.

Features often represent as numeric vectors. Sometimes it is necessary to represent text-based data as feature vectors. There are many established techniques for doing so, including TF-IDF (Term Frequency–Inverse Document Frequency). TF-IDF measures the significance of an element relevant to other elements within a set. This technique is common in text mining and search. For instance, you could assess how important the term "Spark" is in this book

compared to all the other books available on Amazon.com.

Machine Learning Using Spark MLlib

Spark MLlib is a Spark subproject that provides machine learning functions that can be used with RDDs. MLlib, like Spark Streaming and Spark SQL, is an integral component in the Spark program and has come with Spark since the 0.8 release.

Classification Using Spark MLlib

Common approaches or algorithms used for classification in machine learning include decision trees and naive Bayes. Both techniques learn from previous observations and make classification judgments based on probability.

Decision trees are an intuitive form of classification in which a decision process is represented as a tree. Nodes of the tree signify decisions that usually compare an attribute from the dataset with a constant or a label. Each decision node creates a fork in the structure until the end of the tree is reached and a classification prediction is made.

A simple example used to describe decision trees is a golf (or weather) dataset. This simple example is often cited in data mining textbooks as a sample dataset for generating a decision tree. This small dataset, shown in Table 8.4, contains 14 instances (or observations) and 5 primary attributes: outlook, temperature, humidity, windy, and play. The temperature and humidity attributes appear in nominal and numeric formats. The last attribute, play, is the class attribute, which can have a value of "Yes" or "No."

Table 8.4 **Golf/Weather Dataset**

Outlook	Numeric Temp	Nominal Temp	Numeric Humidity	Nominal Humidity	Windy	Play?
Overcast	83	Hot	86	High	False	Yes
Overcast	64	Cool	65	Normal	True	Yes
Overcast	72	Mild	90	High	True	Yes

Overcast	81	Hot	75	Normal	False	Yes
Rainy	70	Mild	96	High	False	Yes
Rainy	68	Cool	80	Normal	False	Yes
Rainy	65	Cool	70	Normal	True	No
Rainy	75	Mild	80	Normal	False	Yes
Rainy	71	Mild	91	High	True	No
Sunny	85	Hot	85	High	False	No
Sunny	80	Hot	90	High	True	No
Sunny	72	Mild	95	High	False	No
Sunny	69	Cool	70	Normal	False	Yes
Sunny	75	Mild	70	Normal	True	Yes

The weather dataset is also included in the *WEKA* (Waikato Environment for Knowledge Analysis) machine learning software package, a popular free software package developed at the University of Waikato, New Zealand. Although not directly related to Spark, this is a recommended package for those who wish to explore machine learning algorithms in more detail.

After using a machine learning decision tree classification algorithm against a set of input data, the model produced evaluates each attribute and progresses through the tree until a decision node is reached. Figure 8.11 shows the decision tree that results for the sample weather dataset using the nominal (or categorical) features.
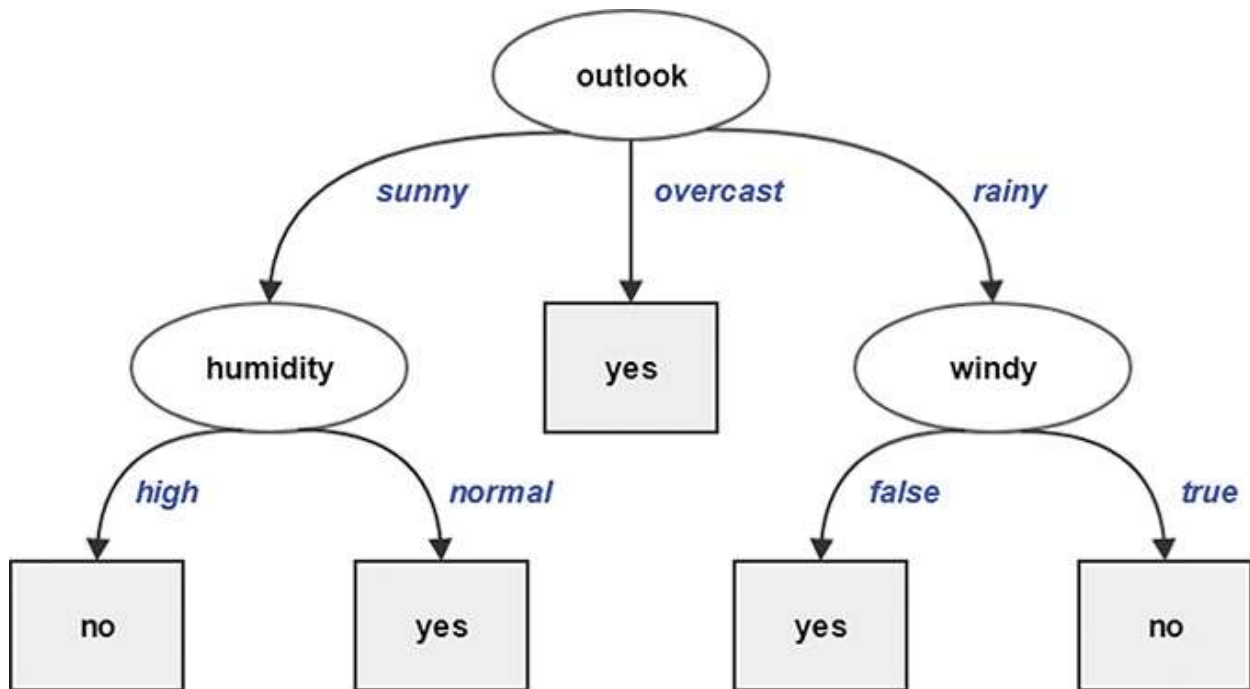
Figure 8.11 Decision tree for the weather dataset.

Spark MLlib supports decision trees for both continuous (numeric) and categorical features. The training process parallelizes instances from a training dataset and iterates over these instances to develop the resultant decision tree.

Splitting Data into Training and Test Datasets

In supervised machine learning model development, it's generally recommended that you split your input dataset into two subsets: a training dataset and a test dataset. The training dataset trains the model and usually comprises 60% or more of the overall input dataset. The test dataset comprises the remaining data from the input dataset and makes predictions to validate the accuracy of the trained model. Spark includes the `randomSplit()` function to split a dataset into multiple datasets for training and testing; `randomSplit()` accepts as input argument weights —that is, a list of weightings for the respective output datasets. Listing 8.17 shows an example of the `randomSplit()` function.

Listing 8.17 **Splitting Data into Training and Test Datasets**

Click here to view code image

```
data = sc.parallelize([1,2,3,4,5,6,7,8,9,10])
training, test = data.randomSplit([0.6, 0.4])
training.collect()
# returns: [1, 2, 5, 6, 9, 10]
test.collect()
# returns: [3, 4, 7, 8]
```

To construct an example of a decision tree classifier using the training dataset, you first need to create an RDD consisting of `LabeledPoint` (`pyspark.mllib.regression.LabeledPoint`) objects. A `LabeledPoint` object contains the label or class attribute for an instance, along with the associated instance attributes. Listing 8.18 shows an example of creating an RDD containing `LabeledPoint` objects. For brevity, this section shows how to use this RDD in some of the examples.

> ## NumPy and Pandas
>
> NumPy is a Python library used for scientific computing. Its special-purpose array objects are used by PySpark MLlib internally and, therefore, it is a required package if you are using MLlib with Python. NumPy is easily installed using `pip` (for example, `pip install numpy`). More information about NumPy is at http://www.numpy.org/. Pandas is another useful Python library; although not required for MLlib, Pandas is useful for structuring and analyzing data. More information about Pandas is at http://pandas.pydata.org/.

Listing 8.18 **Creating an RDD of `LabeledPoint` Objects**

Click here to view code image

```
from pyspark.mllib.regression import LabeledPoint
outlook = {"sunny": 0.0, "overcast": 1.0, "rainy": 2.0}
labeledpoints = [
    LabeledPoint(0.0,[outlook["sunny"],85,85,False]),
    LabeledPoint(0.0,[outlook["sunny"],80,90,True]),
    LabeledPoint(1.0,[outlook["overcast"],83,86,False]),
    LabeledPoint(1.0,[outlook["rainy"],70,96,False]),
    LabeledPoint(1.0,[outlook["rainy"],68,80,False]),
    LabeledPoint(0.0,[outlook["rainy"],65,70,True]),
    LabeledPoint(1.0,[outlook["overcast"],64,65,True]),
```

```
    LabeledPoint(0.0,[outlook["sunny"],72,95,False]),
    LabeledPoint(1.0,[outlook["sunny"],69,70,False]),
    LabeledPoint(1.0,[outlook["sunny"],75,80,False]),
    LabeledPoint(1.0,[outlook["sunny"],75,70,True]),
    LabeledPoint(1.0,[outlook["overcast"],72,90,True]),
    LabeledPoint(1.0,[outlook["overcast"],81,75,False]),
    LabeledPoint(0.0,[outlook["rainy"],71,91,True])
    ]
data = sc.parallelize(labeledpoints)
```

`LabeledPoint` object attributes must be `float` values or objects that can be converted to `float` values, such as `Boolean` or `int`. With a categorical feature (`outlook`), you need to create a dictionary or map to associate the `float` value used in the `LabeledPoint` with a categorical key.

> ## Input Data Formats for Machine Learning in Spark
>
> Spark's machine learning libraries support many input formats commonly used in classification or regression modeling. An example is the `libsvm` file format, a format from a library designed for support vector classification. Many other data structures from popular scientific and statistical packages, such as NumPy and SciPy, are supported in Spark's machine learning libraries as well.

Using the RDD containing `LabeledPoint` objects created in Listing 8.18, you can now train a decision tree model by using the `DecisionTree.trainClassifier()` function in the Spark `mllib` package, as shown in Listing 8.19.

Listing 8.19 **Training a Decision Tree Model with Spark MLlib**

Click here to view code image

```
from pyspark.mllib.tree import DecisionTree
model = DecisionTree.trainClassifier(data=data,
        numClasses=2,
        categoricalFeaturesInfo={0: 3})
print(model.toDebugString())
# returns:
# DecisionTreeModel classifier of depth 3 with 9 nodes
```

```
#   If (feature 0 in {0.0,2.0})
#    If (feature 2 <= 80.0)
#     If (feature 1 <= 65.0)
#      Predict: 0.0
#     Else (feature 1 > 65.0)
#      Predict: 1.0
#    Else (feature 2 > 80.0)
#     If (feature 1 <= 70.0)
#      Predict: 1.0
#     Else (feature 1 > 70.0)
#      Predict: 0.0
#   Else (feature 0 not in {0.0,2.0})
#    Predict: 1.0
```

The `DecisionTree.trainClassifier()` function creates a model by training the data, a parallelized collection of `LabeledPoint` objects. The `numClasses` argument specifies how many discrete classes to predict; in this case, it is two because the example simply predicts a binary outcome of yes/no. The `categoricalFeaturesInfo` argument is a dictionary or map that specifies which features are categorical and how many categorical values each of those features can take. In this case, you need to direct the `trainClassifier()` method that the values representing the outlook category are discrete—for example, `"sunny"` or `"rainy"` or `"overcast"`. Any features not specified in the `categoricalFeaturesInfo` argument are treated as continuous.

When you have a model, what is next? Now you need a method to predict the class attribute from new data that does not include the class attribute. Spark MLlib provides the `predict()` function to do this. Listing 8.20 demonstrates the use of the `predict()` method.

Listing 8.20 **Using a Spark MLlib Decision Tree Model to Classify New Data**

Click here to view code image

```
model.predict([outlook["overcast"],85,85,True])
# returns: 1.0
```

As you can see in Listing 8.20, given the inputs `outlook="overcast"`,

`temperature=85`, `humidity=85`, and `windy=True`, the decision to play is `1.0`, or yes. This follows the logic from the decision tree you created.

Naive Bayes is another popular technique for classification in machine learning. Naive Bayes is based upon Bayes' theorem, which describes how the conditional probability of an outcome can be evaluated from the known probabilities of its causes. Bayes' theorem is modeled mathematically as shown here:

$P(A|B)=P(B|A)P(A)P(B)$

In this case, *A* and *B* are independent events; *P*(*A*) and *P*(*B*) are the probabilities of *A* and *B* without regard to each other; *P*(*A*|*B*) is the probability of observing event *A* given that *B* is true; and *P*(*B*|*A*) is the probability of observing event *B* given that *A* is true.

You implement naive Bayes classification by using Spark MLlib with the `NaiveBayes.train()` method from the `pyspark.mllib.classification.NaiveBayes` package.

`NaiveBayes.train()` takes an input RDD consisting of `LabeledPoint` objects, as in the decision tree example, and includes an optional smoothing parameter, `lambda_`. The output is a `NaiveBayesModel` (`pyspark.mllib.classification.NaiveBayesModel`) that can classify new data using the `predict()` method.

Listing 8.21 uses the weather dataset to create a model using the naive Bayes algorithm implementation in Spark MLlib and then uses this model to predict the class attribute of new data.

Listing 8.21 **Implementing a Naive Bayes Classifier Using Spark MLlib**

Click here to view code image

```
from pyspark.mllib.classification import NaiveBayes, NaiveBayesModel
model = NaiveBayes.train(data=data, lambda_=1.0)
model.predict([1.0,85,85,True])
# returns: 1.0
```

Collaborative Filtering Using Spark MLlib

Collaborative filtering is one of the most common applications of machine

learning in use in many different domains. Spark uses the *ALS* (or *Alternating Least Squares*) technique in its collaborative filtering or recommendation module. ALS is an algorithm for performing matrix factorization. *Matrix factorization* is the process of factorizing a matrix into a product of matrixes. A simple example is shown in Figure 8.12.
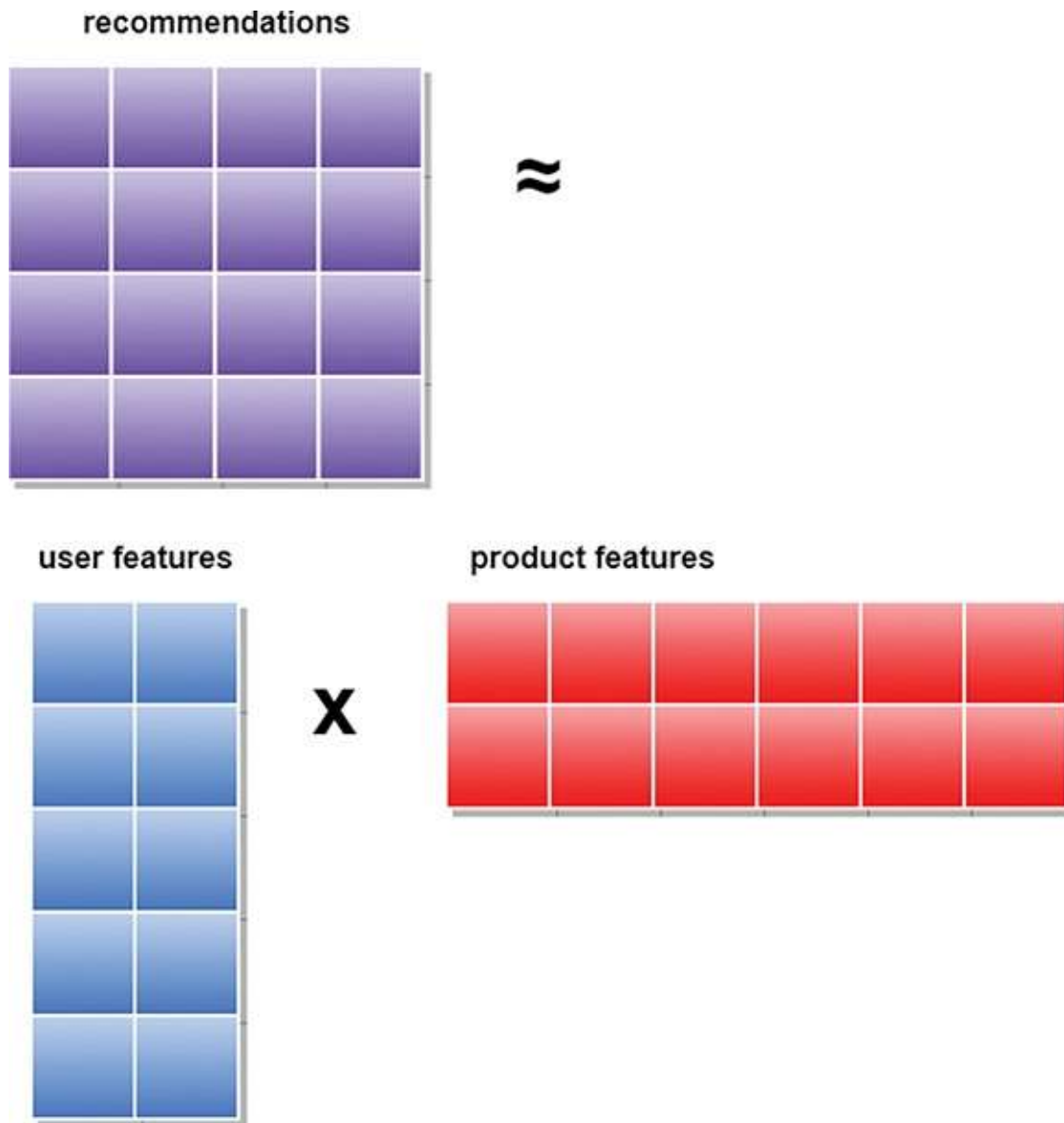

Figure 8.12 Matrix factorization.

A deep dive into matrix factorization and the ALS algorithm is beyond the scope of this book. However, ALS is the preferred implementation method for machine learning in Spark because it is a fully parallelizable algorithm.

The exercise that follows shows an implementation of a recommender using Spark MLlib and ALS.

Exercise: Implementing a Recommender Using Spark MLlib

This exercise uses a subset of the Movielens dataset, which originated at the University of Minnesota as a data exploration and recommendation project. The Movielens dataset captures movie ratings by user, along with user and movie attributes, and can be used for collaborative filtering exercises. The website for the Movielens project is https://movielens.org/. You can download the subset of data used for this exercise at https://s3.amazonaws.com/sparkusingpython/movielens/movielens.dat. This dataset contains 100,000 ratings by 943 users on 1,682 items, with each user having rated at least 20 movies. The ratings data (`movielens.dat`) is a tab-delimited, new line–terminated text file with this structure:

Click here to view code image

```
user id | item id | rating | timestamp
```

For the following exercise you need to be running Spark on a Hadoop cluster and must have the `movielens.dat` file saved to a directory named `/data/movielens`. Follow these steps:

1. Start a `pyspark` shell.

2. Import the required MLlib libraries:

Click here to view code image
```
from pyspark.mllib.recommendation \
import ALS, MatrixFactorizationModel, Rating
```

3. Load the Movielens dataset and create an RDD containing `Rating` objects:

Click here to view code image
```
data = sc.textFile("hdfs:///data/movielens")
ratings = data.map(lambda x: x.split(' ')) \
    .map(lambda x: Rating(int(x[0]), int(x[1]), float(x[2])))
```

`Rating` is a special tuple Spark uses and represents (`user`, `product`, `rating`). Note also that you filter the `timestamp` field because it is not necessary in this case.

4. Train a model using the ALS algorithm:

```
rank = 10
numIterations = 10
model = ALS.train(ratings, rank, numIterations)
```

Note that `rank` and `numIterations` are algorithm tuning parameters; `rank` is the number of latent factors in the model, and `numIterations` is the number of iterations to run.

5. Now you can test the model against the same dataset without the `rating` (use the model to predict this attribute). Then compare the results of the predictions with the actual ratings to determine the mean squared error, measuring the accuracy of the model:

```
testdata = ratings.map(lambda p: (p[0], p[1]))
predictions = model.predictAll(testdata) \
     .map(lambda r: ((r[0], r[1]), r[2]))
ratesAndPreds = ratings.map(lambda r: ((r[0], r[1]), r[2])) \
     .join(predictions)
MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2) \
     .mean()
print("Mean Squared Error = " + str(MSE))
# returns: Mean Squared Error = 0.482478475145
```

As discussed earlier in this chapter, a good practice is to divide your input dataset into two discrete sets, one for training and another for testing. This helps avoid overfitting your model.

6. To save the model for use with new recommendations, use the `model.save()` function, as shown here:

```
model.save(sc, "ratings_model")
```

This saves the model to a folder named `ratings_model` in your current user's home directory in HDFS.

7. To reload the model in a new session—for instance, to deploy the model against real-time data from a Spark DStream—use the `MatrixFactorizationModel.load()` function, as shown here:

```
from pyspark.mllib.recommendation \
import MatrixFactorizationModel
```

```
reloaded_model = MatrixFactorizationModel.load \
                    (sc, "ratings_model")
```

The complete source code for this exercise is in the `recommendation-engine` folder at https://github.com/sparktraining/spark_using_python.

Clustering Using Spark MLlib

As discussed earlier in this chapter, clustering algorithms discover groups or clusters of associated instances within a collection of data. A common approach to clustering is the *k-means* technique.

By definition, k-means clustering is an iterative algorithm used in machine learning and graph analysis. Consider a set of data in a plane—which could represent a variable and an independent variable on an x, y axis, for simplicity. The objective of the k-means algorithm is to find the center of each cluster (the centroid) presented in the data, as pictured in Figure 8.13.

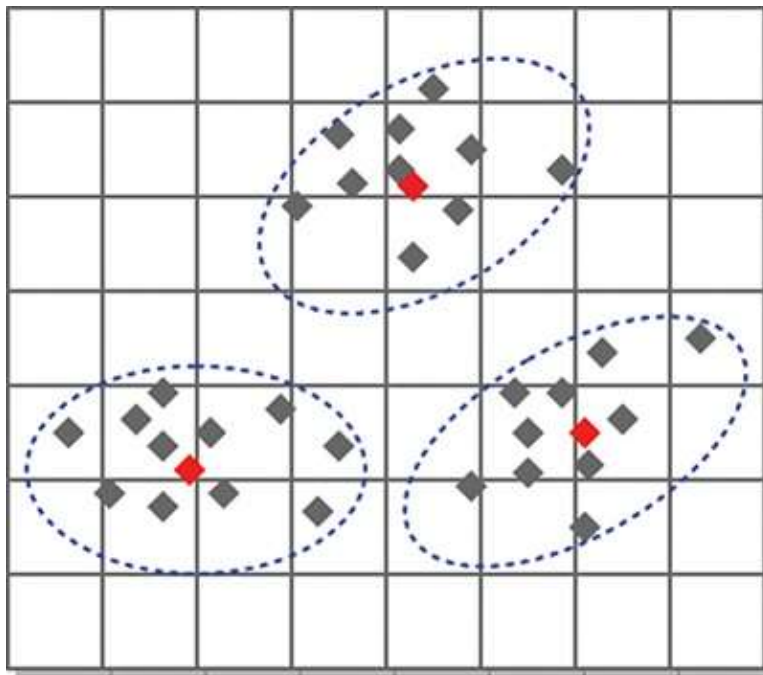

Figure 8.13 k-means clustering.

The k-means approach works as follows:

- Select *k* random points as starting center points (centroids).
- For each point, find the closest *k* and allocate the point to the cluster

associated with *k*.

- Calculate the mean (center) of each cluster by averaging all the points in that cluster.

- Iterate until no points reassign to new clusters.

As you can see, this is a brute-force, parallelizable, iterative routine, which makes it very well suited to Spark.

To implement k-means in Spark, use the `pyspark.mllib.clustering.KMeans` package. Listing 8.22 demonstrates how to train a k-means clustering machine learning model using the sample `kmeans_data` dataset provided as part of the Spark release.

Listing 8.22 **Training a k-Means Clustering Model Using Spark MLlib**

Click here to view code image

```
from pyspark.mllib.clustering import KMeans, KMeansModel
from numpy import array
from math import sqrt
# Load and parse the data
data = sc.textFile("file:///opt/spark/data/mllib/kmeans_data.txt")
parsedData = data.map(lambda line: array( \
              [float(x) for x in line.split(' ')]))
# Build the model (cluster the data)
clusters = KMeans.train(parsedData, 2, maxIterations=10,
    initializationMode="random")
```

Notice that this example uses the NumPy library mentioned earlier. When you have a k-means clustering model, you can evaluate the error rate within each cluster, as shown in Listing 8.23.

Listing 8.23 **Evaluating a k-Means Clustering Model**

Click here to view code image

```
# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(point):
    center = clusters.centers[clusters.predict(point)]
```

```
    return sqrt(sum([x**2 for x in (point - center)]))
WSSSE = parsedData.map(lambda point: error(point)) \
    .reduce(lambda x, y: x + y)
print("Within Set Sum of Squared Error = " + str(WSSSE))
# returns:
# Within Set Sum of Squared Error = 0.692820323028
```

As with the collaborative filtering and classification models, with a k-means model you typically need to persist the model so it can load into a new session to evaluate new data. Listing 8.24 demonstrates the use of the `save` and `load` functions to accomplish this.

Listing 8.24 **Saving and Reloading a Clustering Model**

Click here to view code image

```
# Save and load model
clusters.save(sc, "kmeans_model")
reloaded_model = KMeansModel.load(sc, "kmeans_model")
```

Machine Learning Using Spark ML

Spark ML extends the MLlib library and functions to Spark SQL DataFrames. Spark ML may be a more natural choice for machine learning if you use Spark SQL DataFrames for data processing.

Classification Using Spark ML

Spark ML supports various classification methods, including logistic regression, binomial logistic regression, multinomial logistic regression, decision trees, random forest, gradient-boosted tree, multilayer perceptron, linear support vector machine, one-versus-rest, and naive Bayes.

Just as Spark MLlib classification algorithms require an RDD of `LabeledPoint` objects, Spark ML algorithms require a DataFrame of `Row` objects, including labels and features. The `label` column specifies the classification for the observation, and the `features` column contains either a `SparseVector` or a `DenseVector` object. A `DenseVector` is used when each observation contains the same features, whereas a `SparseVector` is used

when features may vary from instance to instance—that is, some features may be `null` or not populated for certain instances. The main advantage of `SparseVector` is that it only stores features that have a value, which requires less space in datasets that contain `null` values.

Listing 8.25 demonstrates a Spark ML implementation of a decision tree classifier example using the golf/weather dataset from earlier in this chapter.

Listing 8.25 ==Decision Tree== Classifier Using Spark ML

Click here to view code image

```
from pyspark.ml.linalg import DenseVector
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.sql import Row # Prepare DataFrame of labeled observations
outlook = {"sunny": 0.0, "overcast": 1.0, "rainy": 2.0}
observations = [
Row(label=0, features=DenseVector([outlook["sunny"],85,85,False])),
Row(label=0, features=DenseVector([outlook["sunny"],80,90,True])),
Row(label=1, features=DenseVector([outlook["overcast"],83,86,False])),
Row(label=1, features=DenseVector([outlook["rainy"],70,96,False])),
Row(label=1, features=DenseVector([outlook["rainy"],68,80,False])),
Row(label=0, features=DenseVector([outlook["rainy"],65,70,True])),
Row(label=1, features=DenseVector([outlook["overcast"],64,65,True])),
Row(label=0, features=DenseVector([outlook["sunny"],72,95,False])),
Row(label=1, features=DenseVector([outlook["sunny"],69,70,False])),
Row(label=1, features=DenseVector([outlook["sunny"],75,80,False])),
Row(label=1, features=DenseVector([outlook["sunny"],75,70,True])),
Row(label=1, features=DenseVector([outlook["overcast"],72,90,True])),
Row(label=1, features=DenseVector([outlook["overcast"],81,75,False])),
Row(label=0, features=DenseVector([outlook["rainy"],71,91,True]))
]
rdd = sc.parallelize(observations) data = spark.createDataFrame(rdd) #
Split data into training and test sets
(trainingData, testData) = data.randomSplit([0.7, 0.3]) # Train decision
tree model
dt = DecisionTreeClassifier()
model = dt.fit(trainingData)
# returns:
# DecisionTreeClassificationModel
```

```
(uid=DecisionTreeClassifier_495f9e5bcc6aaffa81c5) of depth 4 with 13
nodes # Make predictions using the test  dataset
predictions = model.transform(testData)
predictions.show()
# returns:
# +------------------+-----+-------------+-----------+----------+
# |          features|label|rawPrediction|probability|prediction|
# +------------------+-----+-------------+-----------+----------+
# |[0.0,75.0,80.0,0.0]|    1|    [0.0,4.0]|  [0.0,1.0]|       1.0|
# +------------------+-----+-------------+-----------+----------+ #
Evaluate model accuracy
evaluator = MulticlassClassificationEvaluator(
    labelCol="label", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g " % (1.0 - accuracy))
# returns: Test Error = 0
```

Collaborative Filtering Using Spark ML

As with Spark MLlib, the Spark ML collaborative filtering implementation uses the ALS algorithm. Listing 8.26 demonstrates collaborative filtering using Spark ML.

Listing 8.26 **Collaborative Filtering Example Using Spark ML**

Click here to view code image

```
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.recommendation import ALS
from pyspark.sql import Row # load and prepare data, split data into
training and test  datasets
ratings_rdd = sc.textFile("/opt/spark/data/movielens") \
    .map(lambda x: x.split(' ')) \
    .map(lambda x: Row(userId=int(x[0]), movieId=int(x[1]),
            rating=float(x[2]), timestamp=int(x[3])))
ratings = spark.createDataFrame(ratings_rdd) (training, test) =
ratings.randomSplit([0.7, 0.3]) # train model
als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId",
ratingCol="rating",
        coldStartStrategy="drop")
```

```
model = als.fit(training) # evaluate model
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating",
    predictionCol="prediction")
rmse = evaluator.evaluate(predictions)
print("Root-mean-square error = " + str(rmse))
# returns: Root-mean-square error = 1.093931162606997 # movie
recommendations for each user
model.recommendForAllUsers(3).show(3)
# returns:
# +------+--------------------+
# |userId|     recommendations|
# +------+--------------------+
# |   471|[[1206,9.413772],...|
# |   463|[[1206,6.576718],...|
# |   833|[[853,5.8933687],...|
# +------+--------------------+ # user recommendations for each movie
model.recommendForAllItems(3).show(3)
# returns:
# +-------+--------------------+
# |movieId|     recommendations|
# +-------+--------------------+
# |   1580|[[475,1.8473656],...|
# |    471|[[628,5.776228], ...|
# |   1591|[[777,8.130051], ...|
# +-------+--------------------+
```

Clustering Using Spark ML

Clustering techniques supported in Spark ML include k-means, bisecting k-means, latent Dirichlet allocation (LDA), and Gaussian mixture model (GMM). Listing 8.27 demonstrates k-means clustering using Spark ML.

Listing 8.27 k-Means Clustering with Spark ML

Click here to view code image

```
from pyspark.ml.clustering import KMeans

# load data
```

```
dataset =
spark.read.format("libsvm").load("/opt/spark/data/mllib/sample_kmeans_da


# train a k-means model
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

# evaluate using Within Set Sum of Squared Errors
wssse = model.computeCost(dataset)
print("Within Set Sum of Squared Errors = " + str(wssse))
# returns:
# Within Set Sum of Squared Errors = 0.11999999999994547

# show results
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
# returns:
# [ 0.1  0.1  0.1]
# [ 9.1  9.1  9.1]
```

`libsvm` Format

LIBSVM (library for support vector machines) provides a format specification for files containing training data. The `libsvm` format allows for sparse data. The Spark `DataFrameReader` and `DataFrameWriter` include native support for the `libsvm` format, as shown in Listing 8.27. The `libsvm` format provides a useful way to store and process training data for machine learning algorithms using Spark ML.

ML Pipelines

Spark ML introduces support for machine learning *pipelines*. The *Scikit-learn* project (Python machine learning library) inspired the pipeline concept. Pipelines allow you to chain data preparation and feature extraction steps with models to encapsulate all of your workflow. Pipeline components include *transformers, estimators,* and *parameters*.

A `Transformer` object transforms a DataFrame into another DataFrame by implementing a `transform()` method. An `Estimator` object is an algorithm that can fit on a DataFrame to produce a model by implementing a `fit()` method. A `Pipeline` object chains multiple `Transformer` and `Estimator` objects together to encapsulate a Spark ML workflow. The Parameter API provides a uniform mechanism for `Transformer` and `Estimator` objects to specify parameters. Listing 8.28 demonstrates a Spark ML pipeline for classifying text.

Listing 8.28 **Spark ML Pipelines**

Click here to view code image

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer # Prepare training
documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"]) # Configure an ML pipeline, which consists
of 3 stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(),
outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr]) # Fit the
pipeline to training documents.
model = pipeline.fit(training) # Make predictions on test documents ...
```

Using Notebooks with Spark

Notebooks have become popular tools in the Spark development community. Notebooks provide the capability to combine different languages along with visualization, rich text, and markup and markdown facilities, making it easy to explore and visualize data in an interactive environment. Importantly, notebooks allow you to use data to tell a story and can encapsulate data preparation, model

training and testing, and visualization in a single, succinct document, making it easy to follow or reproduce your thought process.

Using Jupyter (IPython) Notebooks with Spark

Jupyter, formally known as the IPython notebook, provides a web-based notebook experience that includes extensions for Ruby, R, and other languages. Jupyter notebook files are an open document format using JSON. Notebook files contain source code, text, markup, media content, metadata, and more. Notebook contents are stored in cells in a document. Figure 8.14 and Listing 8.29 show an example of a Jupyter notebook and an excerpt from the associated JSON document.
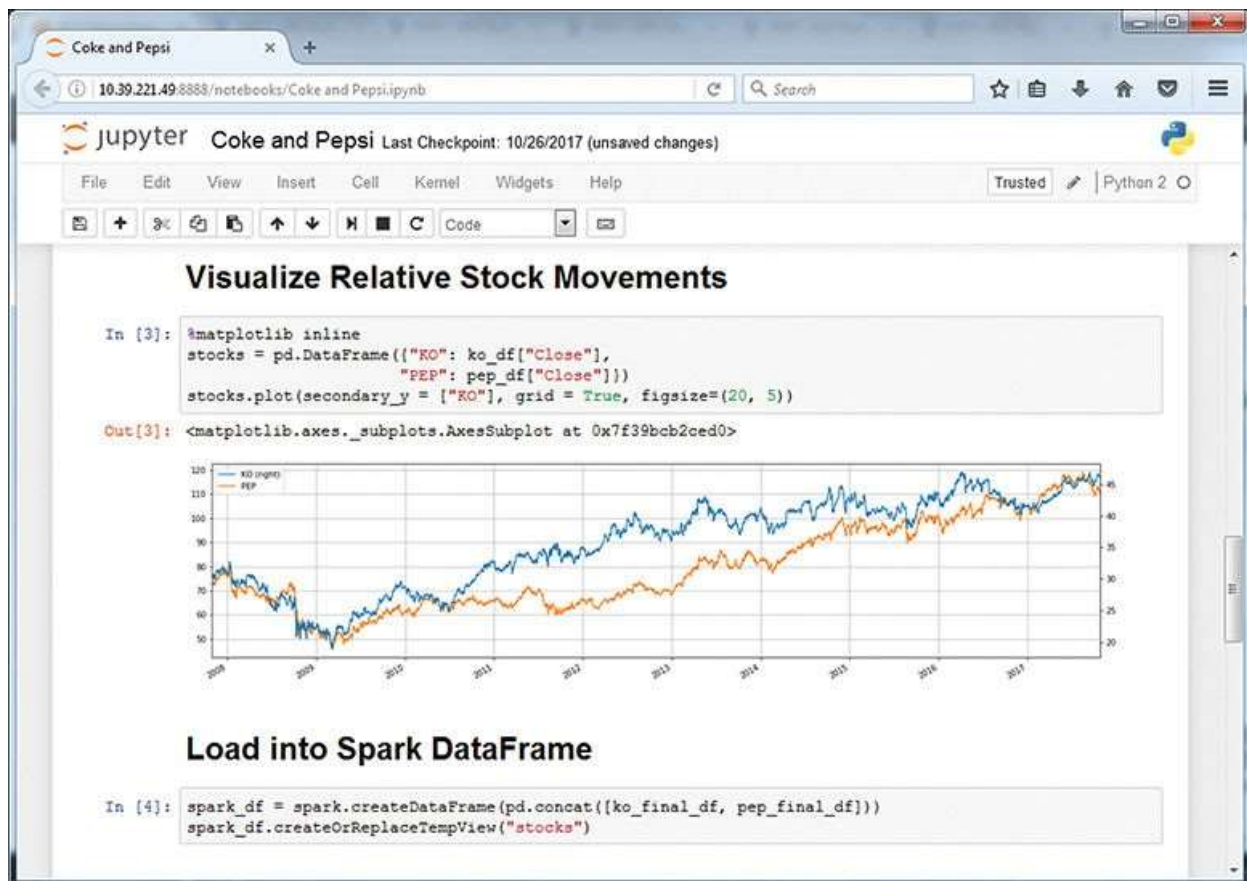

Figure 8.14 Jupyter notebook.

Listing 8.29 **Jupyter Notebook JSON Document**

Click here to view code image

```json
{"cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "# Calculate Pearson Coefficient"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {},
   "outputs": [],
   "source": [
    "import numpy as np\n",
    "from pyspark.mllib.stat import Statistics\n",
    "spark_df =
spark.read.parquet('hdfs://namenode:8020/data/closingprices/')\n",
    "seriesX =
spark_df.select('Close').where(\"Stock='KO'\").rdd.map(lambda x:
float(x.Close))\n", "seriesY =
spark_df.select('Close').where(\"Stock='PEP'\").rdd.map(lambda x:
float(x.Close))\n",
    "correlation = str(Statistics.corr(seriesX, seriesY,
method=\"pearson\"))\n",
    "printmd('# Pearson Correlation between KO and PEP is: <span
style=\"color:red\">' + correlation + '</span> ')"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "metadata": {
    "collapsed": true
   },
   "outputs": [],
   "source": []
  }
 ],
 "metadata": {
```

```
  "kernelspec": {
   "display_name": "Python 2",
   "language": "python",
   "name": "python2"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 2
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython2",
   "version": "2.7.13"
  }
 }}
```

Jupyter/IPython notebooks communicate with back-end systems using kernels. *Kernels* are processes that run interactive code in a particular programming language and return output to the user. Kernels also respond to tab completion and introspection requests. Kernels communicate with notebooks using the Interactive Computing Protocol, which is an open network protocol based on JSON data over ZMQ and WebSocket.

Kernels are currently available for Scala, Ruby, JavaScript, Erlang, Bash, Perl, PHP, PowerShell, Clojure, Go, Spark, and many other languages. Of course, there is a kernel to communicate with IPython (which was the basis of the Jupyter project). The IPython kernel is known as Kernel Zero, and it is the reference implementation for all other kernels.

Using Apache Zeppelin Notebooks with Spark

Apache Zeppelin is a web-based, multilanguage, interactive notebook application with native Spark integration. Zeppelin provides a query environment for Spark, and it provides data- visualization capabilities.

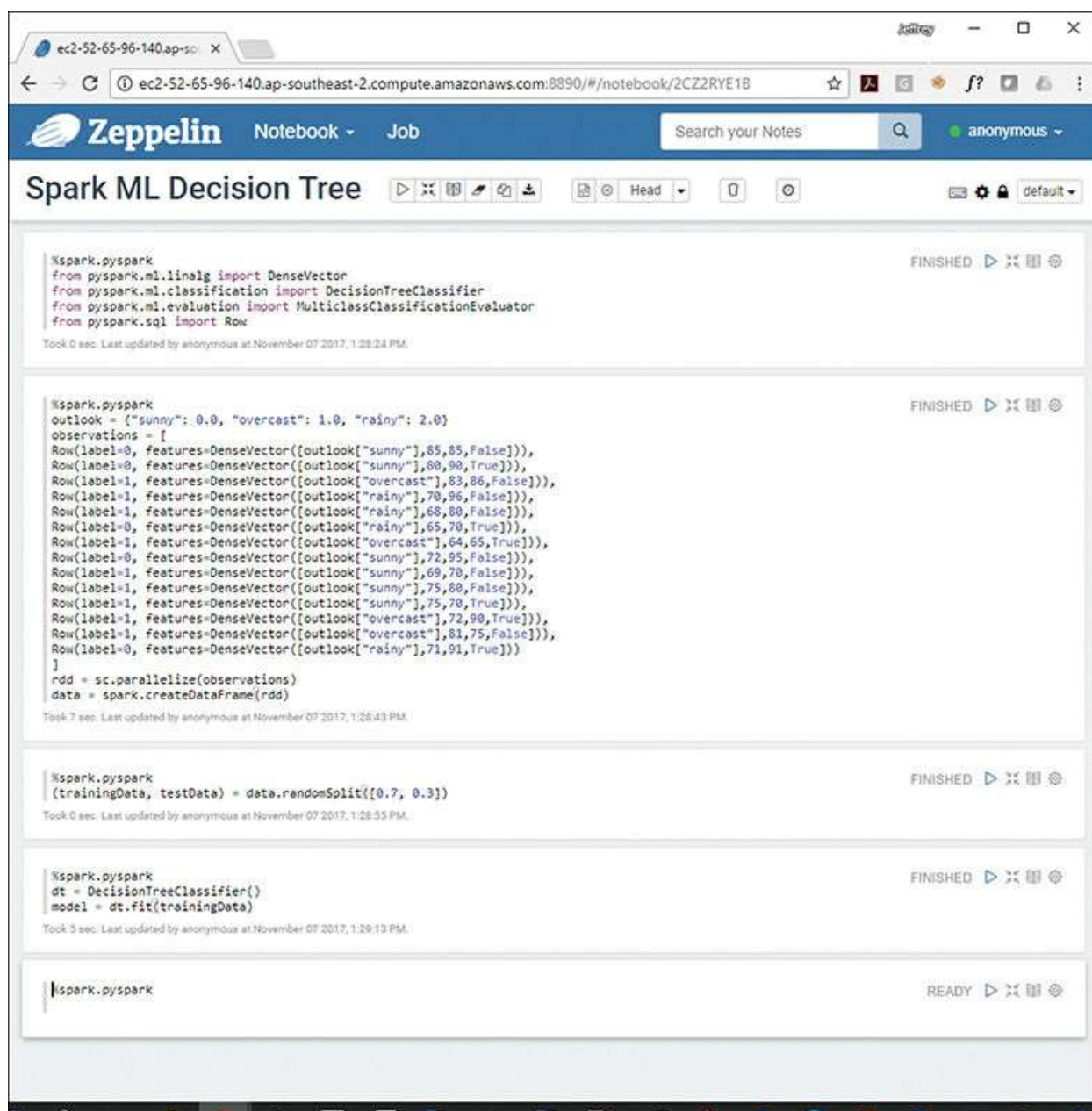Figure 8.15 shows a Zeppelin notebook running a PySpark program.

Figure 8.15 Zeppelin notebook.

Zeppelin Interpreters

Zeppelin interpreters are analogous to the Jupyter kernels just discussed. Interpreters allow Zeppelin to use various programming interfaces and runtimes. As an example, to use the PySpark language and runtime in Zeppelin, you need the `%spark.pyspark` interpreter. Other available interpreters include `%md` (Markdown), `%angular` (AngularJS), `%python`, `%sh` (Shell commands), `%spark.sql`, `%spark` (Spark using

the Scala API), and many others.

Summary

Predictive analytics and machine learning are core use cases for Spark. Spark provides seamless integration to R through the SparkR API, a package that provides access to Spark and distributed data frame operations from an R environment using the R programming language.

SparkR provides various methods for creating data frames, including loading data from external sources such as flat files in a local or distributed file system or from a Hive table. SparkR enables the use of R data frames for distributed operations with Spark, including statistical analysis and building, testing, and deployment of simple linear regression models. SparkR is accessible from an integrated REPL shell environment, `sparkR`, and through the graphical RStudio programming interface.

R continues to grow in popularity; it is finding its way from researchers at universities to business and data analysts in commercial and government organizations. As R becomes the standard for statistical analysis and modeling in many organizations, SparkR and the Spark distributed processing runtime become compelling features for analysis at scale.

Machine learning is a rapidly emerging area of computer science that enables systems and models to "learn" from observations and data. The three primary techniques used in machine learning are classification, collaborative filtering, and clustering. This chapter examines all of these approaches, using Spark's built-in machine learning libraries (MLlib and ML), including their specific applications and common uses and implementations.

MLlib is built on the Spark core RDD API, and ML is built on the DataFrame API. Both the MLlib and ML packages include many common machine learning algorithms and utilities to perform data preparation, feature extraction, model training, and testing. MLlib and ML are designed for succinct, user-friendly yet functionally rich, powerful, and scalable machine learning abstraction on top of Spark.

This chapter looks at using notebooks with Spark; notebooks are a popular development interface for researchers and data scientists. The IPython notebook Jupyter and Apache Zeppelin both enable you to combine multiple languages with visualizations, rich text, and markdown.

You have finished the first part of your journey of learning Spark using Python. I hope this book has helped you in building solid foundations as a Spark and Python practitioner. Thank you for your time, and I wish you all the best in your career!