Machine Learning with PySpark

With Natural Language Processing and Recommender Systems

Pramod Singh

Apress

Machine Learning with PySpark

Pramod Singh
Bangalore, Karnataka, India

ISBN-13 (pbk): 978-1-4842-4130-1 ISBN-13 (electronic): 978-1-4842-4131-8
https://doi.org/10.1007/978-1-4842-4131-8

Library of Congress Control Number: 2018966519

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4130-1. For more detailed information, please visit http://www.apress.com/source-code.

Contents

Introduction

Before even starting to write this book, I asked myself a question: Is there a need for another book on Machine Learning? I mean that there are so many books written on this subject already that this might end up as just another book on the shelf. To find the answer, I spent a lot of time thinking and after a while, a few patterns started to emerge. The books that have been written on Machine Learning were too detailed and lacked a high-level overview. Most of these would start really easy but after a couple of chapters, it felt overwhelming to continue as the content became too deep. As a result, readers would give up without getting enough out of the book. That's why I wanted to write this book, which demonstrates the different ways of using Machine Learning without getting too deep, yet capturing the complete methodology to build an ML model from scratch. The next obvious question was this: Why Machine Learning using PySpark? The answer to this question did not take too long since I am a practicing Data Scientist and well aware of the challenges faced by people dealing with data. Most of the packages or modules are often limited as they process data on a single machine. Moving from a development to production environment becomes a nightmare if ML models are not meant to handle Big Data, and finally the processing of data itself needs to be fast and scalable. For all these reasons, it made complete sense to write this book on Machine Learning using PySpark to understand the process of using Machine Learning from a Big Data standpoint.

Now we come to the core of the book *Machine Learning with PySpark*. This book is divided into three different sections. The first section gives the introduction to Machine Learning and Spark, the second section talks about Machine Learning in detail using Big Data, and finally the third part

showcases Recommender Systems and NLP using PySpark. This book might also be relevant for Data Analysts and Data Engineers as it covers steps of Big Data processing using PySpark as well. The readers who want to make a transition to Data Science and the Machine Learning field would also find this book easier to start with and can gradually take up more complicated stuff later. The case studies and examples given in the book make it really easy to follow along and understand the fundamental concepts. Moreover, there are very few books available on PySpark out there, and this book would certainly add some value to the knowledge of the readers. The strength of this book lies in explaining the Machine Learning algorithms in the most simplistic ways and uses a practical approach toward building them using PySpark.

I have put in my entire experience and learning into this book and feel it is precisely relevant to what businesses are seeking out there to solve real challenges. I hope you have some useful takeaways from this book.

Evolution of Data

Before understanding Spark, it is imperative to understand the reason behind this deluge of data that we are witnessing around us today. In the early days, data was generated or accumulated by workers, so only the employees of companies entered the data into systems and the data points were very limited, capturing only a few fields. Then came the internet, and information was made easily accessible to everyone using it. Now, users had the power to enter and generate their own data. This was a massive shift as the number of internet users grew exponentially, and the data created by these users grew at even a higher rate. For example: login/sign-up forms allow users to fill in their own details, uploading photos and videos on various social platforms. This resulted in huge data generation and the need for a fast and scalable framework to process this amount of data.

Data Generation

This data generation has now gone to the next level as machines are generating and accumulating data as shown in Figure 1-1. Every device around us is capturing data such as cars, buildings, mobiles, watches, flight engines. They are embedded with multiple monitoring sensors and recording data every second. This data is even higher in magnitude then the user-generated data.

Figure 1-1. *Data Evolution*

Earlier, when the data was still at enterprise level, a relational database was good enough to handle the needs of the system, but as the size of data increased exponentially over the past couple of decades, a tectonic shift happened to handle the big data and it was the birth of Spark. Traditionally, we used to take the data and bring it to the processer to process it, but now it's so much data that it overwhelms the processor. Now we are bringing multiple processors to the data. This is known as parallel processing as data is being processed at a number of places at the same time.

Let's look at an example to understand parallel processing. Assume that on a particular freeway, there is only a single toll booth and every vehicle has to get in a single row in order to pass through the toll booth as shown in Figure 1-2. If, on average, it takes 1 minute for each vehicle to pass through the toll gate, for eight vehicles, it would take a total of 8 minutes. For 100 vehicles, it would take 100 minutes.

Figure 1-2. *Single Thread Processing*

But imagine if instead of a single toll booth, there are eight toll booths on the same freeway and vehicles can use anyone of them to pass through. It would take only 1 minute in total for all of the eight vehicles to pass through the toll booth because there is no dependency now as shown in Figure 1-3. We have parallelized the operations.

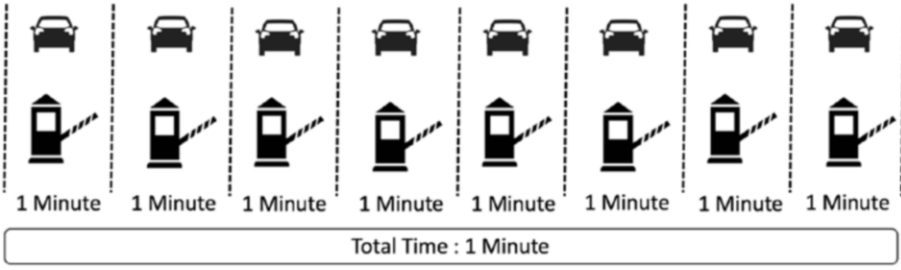


Figure 1-3. *Parallel Processing*

Parallel or Distributed computing works on a similar principle, as it parallelizes the tasks and accumulates the final results at the end. Spark is a framework to handle massive datasets with parallel processing at high speed and is a robust mechanism.

Spark

Apache Spark started as a research project at the UC Berkeley AMPLab in 2009 and was open sourced in early 2010 as shown in Figure 1-4. Since then, there has been no looking back. In 2016, Spark released TensorFrames for Deep Learning.

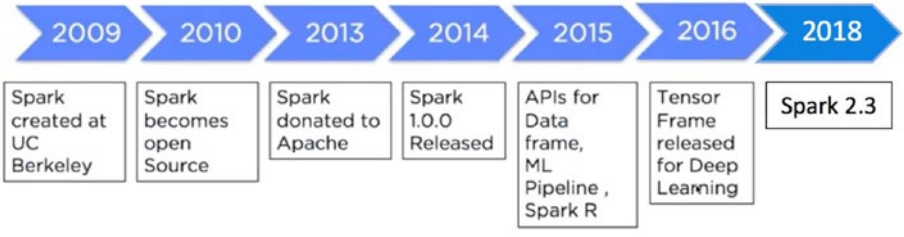


Figure 1-4. *Spark Evolution*

Under the hood, Spark uses a different data structure known as RDD (Resilient Distributed Dataset). It is resilient in a sense that they have an ability to re-create any point of time during the execution process. So RDD creates a new RDD using the last one and always has the ability to reconstruct in case of any error. They are also immutable as original RDDs remain unaltered. As Spark is a distributed framework, it works on master and worker node settings as shown in Figure 1-5. The code to execute any of the activities is first written on Spark Driver, and that is shared across worker nodes where the data actually resides. Each worker node contains Executors that will actually execute the code. Cluster Manager keeps a check on the availability of various worker nodes for the next task allocation.

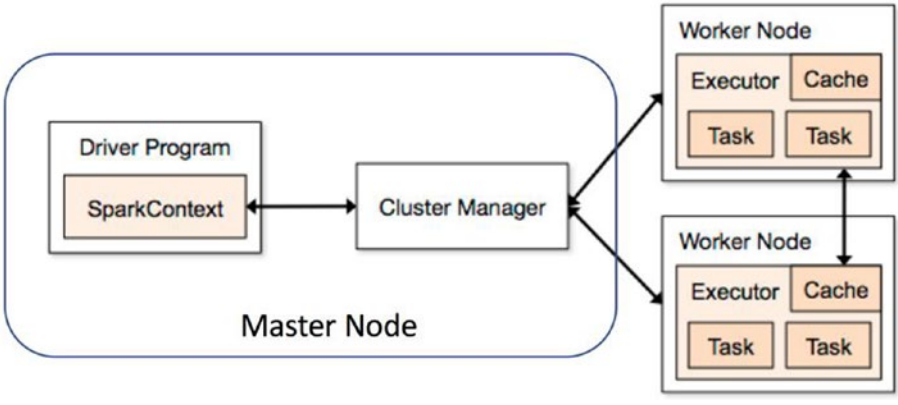


Figure 1-5. *Spark Functioning*

The prime reason that Spark is hugely popular is due to the fact that it's very easy to use it for data processing, Machine Learning, and streaming data; and it's comparatively very fast since it does all in-memory computations. Since Spark is a generic data processing engine, it can easily be used with various data sources such as HBase, Cassandra, Amazon S3, HDFS, etc. Spark provides the users four language options to use on it: Java, Python, Scala, and R.

Spark Core

Spark Core is the most fundamental building block of Spark as shown in Figure 1-6. It is the backbone of Spark's supreme functionality features. Spark Core enables the in-memory computations that drive the parallel and distributed processing of data. All the features of Spark are built on top of Spark Core. Spark Core is responsible for managing tasks, I/O operations, fault tolerance, and memory management, etc.

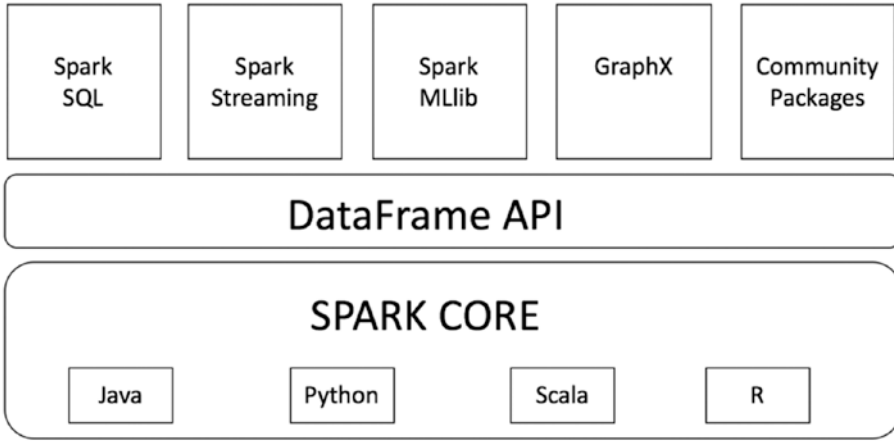


Figure 1-6. *Spark Architecture*

Spark Components

Let's look at the components.

Spark SQL

This component mainly deals with structured data processing. The key idea is to fetch more information about the structure of the data to perform additional optimization. It can be considered a distributed SQL query engine.

Spark Streaming

This component deals with processing the real-time streaming data in a scalable and fault tolerant manner. It uses micro batching to read and process incoming streams of data. It creates micro batches of streaming data, executes batch processing, and passes it to some file storage or live dashboard. Spark Streaming can ingest the data from multiple sources like Kafka and Flume.

Spark MLlib

This component is used for building Machine Learning Models on Big Data in a distributed manner. The traditional technique of building ML models using Python's scikit learn library faces lot of challenges when data size is huge whereas MLlib is designed in a way that offers feature engineering and machine learning at scale. MLlib has most of the algorithms implemented for classification, regression, clustering, recommendation system, and natural language processing.

Spark GraphX/Graphframe

This component excels in graph analytics and graph parallel execution. Graph frames can be used to understand the underlying relationships and visualize the insights from data.

Setting Up Environment

This section of the chapter covers setting up a Spark Environment on the system. Based on the operating system, we can choose the option to install Spark on the system.

Windows

Files to Download:

1. Anaconda (Python 3.x)
2. Java (in case not installed)
3. Apache Spark latest version
4. Winutils.exe

Anaconda Installation

Download the Anaconda distribution from the link https://www.anaconda.com/download/#windows and install it on your system. One thing to be careful about while installing it is to enable the option of adding Anaconda to the path environment variable so that Windows can find relevant files while starting Python.

Once Anaconda is installed, we can use a command prompt and check if Python is working fine on the system. You may also want to check if Jupyter notebook is also opening up by trying the command below:

```
[In]: Jupyter notebook
```

Java Installation

Visit the https://www.java.com/en/download/link and download Java (latest version) and install Java.

Spark Installation

Create a folder named spark at the location of your choice. Let's say we decide to create a folder named spark in D:/ drive. Go to https://spark.apache.org/downloads.html and select the Spark release version that you want to install on your machine. Choose the package type option of

"Pre-built for Apache Hadoop 2.7 and later." Go ahead and download the .tgz file to the spark folder that we created earlier and extract all the files. You will also observe that there is a folder named bin in the unzipped files.

The next step is to download winutils.exe and for that you need to go to the link https://github.com/steveloughran/winutils/blob/master/hadoop-2.7.1/bin/winutils.exe and download the .exe file and save it to the bin folder of the unzipped spark folder (D:/spark/spark_unzipped/bin).

Now that we have downloaded all the required files, the next step is adding environment variables in order to use pyspark.

Go to the start button of Windows and search for "Edit environment variables for your account." Let's go ahead and create a new environment variable for winutils and assign the path for the same. Click on new and create a new variable with the name HADOOP_HOME and pass the path of the folder (D:/spark/spark_unzipped) in the variable value placeholder.

We repeat the same process for the spark variable and create a new variable with name SPARK_HOME and pass the path of spark folder (D:/spark/spark_unzipped) in the variable value placeholder.

Let's add a couple of more variables to use Jupyter notebook. Create a new variable with the name PYSPARK_DRIVER_PYTHON and pass Jupyter in the variable value placeholder. Create another variable named PYSPARK_DRIVER_PYTHON_OPTS and pass the notebook in the value field.

In the same window, look for the Path or PATH variable, click edit, and add D:/spark/spark_unzipped/bin to it. In Windows 7 you need to separate the values in Path with a semicolon between the values.

We need to add Java as well to the environment variable. So, create another variable JAVA_HOME and pass the path of the folder where Java is installed.

We can open the cmd window and run Jupyter notebook.

```
[In]: Import findspark
[In]: findspark.init()
```

```
[In]:import pyspark
[In]:from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.getOrCreate()
```

IOS

Assuming we have Anaconda and Java installed on our Mac already, we can download the latest version of Spark and save it to the home directory. We can open the terminal and go to the home directory using

```
[In]:  cd ~
```

Copy the downloaded spark zipped file to the home directory and unzip the file contents.

```
[In]: mv /users/username/Downloads/ spark-2.3.0-bin-hadoop2.7
/users/username
```

```
[In]: tar -zxvf spark-2.3.0-bin-hadoop2.7.tgz
```

Validate if you have a .bash_profile.

```
[In]: ls -a
```

Next, we will edit the .bash_profile so that we can open a Spark notebook in any directory.

```
[In]: nano .bash_profile
```

Paste the items below in the bash profile.

```
export SPARK_PATH=~/spark-2.3.0-bin-hadoop2.7
export PYSPARK_DRIVER_PYTHON="jupyter"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"

alias notebook='$SPARK_PATH/bin/pyspark --master local[2]'
```

```
[In]: source  .bash_profile
```

Now try opening Jupyter notebook in a terminal and import Pyspark to use it.

Docker

We can directly use PySpark with Docker using an image from the repository of Jupyter but that requires Docker installed on your system.

Databricks

Databricks also offers a community edition account that is free of cost and provides 6 GB clusters with PySpark.

Conclusion

In this chapter, we looked at Spark Architecture, various components, and different ways to set up the local environment in order to use Spark. In upcoming chapters, we will go deep into various aspects of Spark and build a Machine Learning model using the same.

Introduction to Machine Learning

When we are born, we are incapable of doing anything. We can't even hold our head straight at that time, but eventually we start learning. Initially we all fumble, make tons of mistakes, fall down, and bang our head many times but slowly learn to sit, walk, run, write, and speak. As a built-in mechanism, we don't require a lot of examples to learn about something. For example, just by seeing two to three houses along the roadside, we can easily learn to recognize a house. We can easily differentiate between a car and a bike just by seeing a few cars and bikes around. We can easily differentiate between a cat and a dog. Even though it seems very easy and intuitive to us as human beings, for machines it can be a herculean task.

Machine Learning is the mechanism through which we try to make machines learn without explicitly programming them to do so. In simple terms, we showcase the machine a lot of pictures of cats and dogs, just enough for the machine to learn the difference between the two and recognise the new picture correctly. The question here might be the following: What is the need of so many pictures to learn something as simple as the differntiating between cats and dogs? The challenge that the machines face is that they are able to learn the entire pattern or abstraction features just from a few images; they would need enough examples (different in some ways) to learn as many features

as possible to be able to make the right prediction whereas as humans we have this amazing ability to draw abstraction at different levels and easily recognize objects. This example might be specific to an image recognition case, but for other applications as well, machines would need a good amount of data to learn from.

Machine Learning is one of the most talked about topics in the last few years. More and more businesses want to adopt it to maintain the competitive edge; however, very few really have the right resources and the appropriate data to implement it. In this chapter, we will cover basic types of Machine Learning and how businesses can benefit from using Machine Learning.

There are tons of definitions of Machine Learning on the internet, although if I could try to put in in simple terms, it would look something like this:

1. Machine Learning is using statistical techniques and sometimes advanced algorithms to either make predictions or learn hidden patterns within the data and essentially replacing rule-based systems to make data-driven systems more powerful.

Let's go through this definition in detail. Machine Learning, as the name suggests, is making a machine learn, although there are many components that come into the picture when we talk about making a machine learn.

One component is data, which is the backbone for any model. Machine Learning thrives on relevant data. The more signals in the data, the better are the predictions. Machine Learning can be applied in different domains such as financial, retail, health care, and social media. The other part is the algorithm. Based on the nature of the problem we are trying to solve, we choose the algorithm accordingly. The last part consists of the hardware and software. The availability of open sourced, distributed computing frameworks like Spark and Tensorflow have made Machine Learning more accessible to everyone. The rule-based systems

came into the picture when the scenarios were limited and all the rules could be configured manually to handle the situations. Lately, this has changed, specifically the number of scenarios part. For example, the manner in which a fraud can happen has dramatically changed over the past few years, and hence creating manual rules for such conditions is practically impossible. Therefore, Machine Learning is being leveraged in such scenarios that learn from the data and adapts itself to the new data and makes a decision accordingly. This has proven to be of tremendous business value for everyone.

Let's see the different types of machine learning and its applications. We can categorize machine learning into four major categories:

1. Supervised Machine Learning
2. Unsupervised Machine Learning
3. Semi-supervised Machine Learning
4. Reinforcement Learning

Each of the above categories is used for a specific purpose and the data that is used also differs from each other. At the end of the day, machine learning is learning from data (historical or real time) and making decisions (offline or real time) based on the model training.

Supervised Machine Learning

This is the prime category of machine learning that drives a lot of applications and value for businesses. In Supervised Learning, we train our models on the labeled data. By labeled, it means having the correct answers or outcome for the data. Let's take an example to illustrate supervised learning. If there is a financial company that wants to filter customers based on their profiles before accepting their loan requests, the machine learning model would get trained on historical data, which

contains information regarding profiles of the past customer and the label column if a customer has defaulted on a loan or not. The sample data looks like that given in Table 2-1.

Table 2-1. *Customer Details*

Customer ID	Age	Gender	Salary	Number of Loans	Job Type	Loan Default
AL23	32	M	80K	1	Permanent	No
AX43	45	F	105K	2	Permanent	No
BG76	51	M	75K	3	Contract	Yes

In Supervised Learning, the model learns from the training data that also has a label/outcome/target column and uses this to make predictions on unseen data. In the above example, the columns such as Age, Gender, and Salary are known as attributes or features, whereas the last column (Loan Default) is known as the target or label that the model tries to predict for unseen data. One complete record with all these values is known as an observation. The model would require a sufficient amount of observations to get trained and then make predictions on similar kind of data. There needs to be at least one input feature/attribute for the model to get trained along with the output column in supervised learning. The reason that the machine is able to learn from the training data is because of the underlying assumption that some of these input features individually or in combination have an impact on the output column (Loan Default).

There are many applications that use supervised learning settings such as:

Case 1: If any particular customer would buy the product or not?

Case 2: If the visitor would click on the ad or not?

Case 3: If the person would default on the loan or not?

Case 4: What is the expected sale price of a given property?

Case 5: If the person has a malignant tumor or not?

Above are some of the applications of Supervised Learning, and there are many more. The methodology that is used sometimes varies based on the kind of output the model is trying to predict. If the target label is a categorical type, then its falls under the Classification category; and if the target feature is a numerical value, it would fall under the Regression category. Some of the supervised ML algorithms are the following:

1. Linear Regression
2. Logistic Regression
3. Support Vector Machines
4. Naïve Bayesian Classifier
5. Decision Trees
6. Ensembling Methods

Another property of Supervised Learning is that the model's performance can be evaluated. Based on the type of model (Classification/Regression/time series), the evaluation metric can be applied and performance results can be measured. This happens mainly by splitting the training data into two sets (Train Set and Validation Set) and training the model on a train set and testing its performance on a validation set since we already know the right label/outcome for the validation set. We can then make the changes in the Hyperparameters (covered in later chapters) or introduce new features using feature engineering to improve the performance of the model.

Unsupervised Machine Learning

In Unsupervised Learning, we train the models on similar sorts of data except for the fact that this dataset does not contain any label or outcome/target column. Essentially, we train the model on data without any right answers. In Unsupervised Learning, the machine tries to find hidden

patterns and useful signals in the data that can be later used for other applications. One of the uses is to find patterns within customer data and group the customers into different clusters that represent some of the properties. For example, let's look at some customer data in Table 2-2.

Table 2-2. *Customer Details*

Customer ID	Song Genre
AS12	Romantic
BX54	Hip Hop
BX54	Rock
AS12	Rock
CH87	Hip Hop
CH87	Classical
AS12	Rock

In the above data, we have customers and the kinds of music they prefer without any target or output column, simply the customers and their music preference data.

We can use unsupervised learning and group these customers into meaningful clusters to know more about their group preference and act accordingly. We might have to tweak the dataset into other form to actually apply the unsupervised learning. We simply take the value counts for each customer and it would look like that shown in Table 2-3.

Table 2-3. *Customer Details*

Customer ID	Romantic	Hip Hop	Rock	Classical
AS12	1	0	2	0
BX54	0	1	1	0
CH87	0	1	0	1

We can now form some useful groups of users and apply that information to recommend and formulate a strategy based on the clusters. The information we can certainly extract is which of the customers are similar in terms of preferences and can be targeted from a content standpoint.

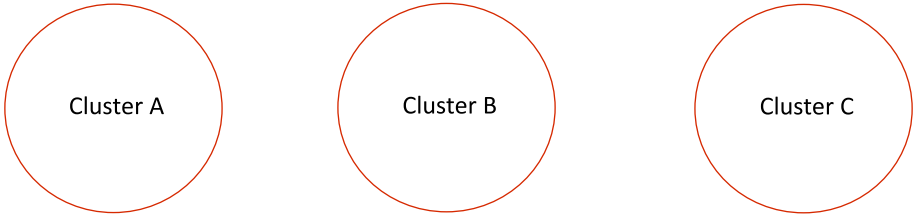


Figure 2-1. *Clusters post Unsupervised Learning*

Like what is shown in Figure 2-1, Cluster A can belong to customers who prefer only Rock and Cluster B can be of people preferring Romantic & Classical music, and the last cluster might be of Hip Hop and Rock lovers. One of the other uses of unsupervised learning is to find out if there is any unusual activity or anomaly detection. Unsupervised learning can help to determine the odd man out from the dataset. Most of the time, unsupervised learning can be very tricky as there are no clear groups or overlapping values between multiple groups, which doesn't give a clear picture of the clusters. For example, as shown in Figure 2-2, there are no clear groups in the data and unsupervised learning cannot help with forming real meaningful clusters of data points.

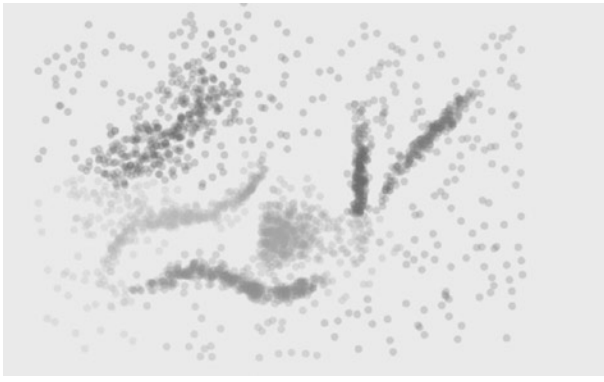


Figure 2-2. *Overlapping Clusters*

There are many applications that use unsupervised learning settings such as

Case 1: What are different groups within the total customer base?

Case 2: Is this transaction an anomaly or normal?

The algorithms used in unsupervised learning are

1. Clustering Algorithms (K-Means, Hierarchical)
2. Dimensionality Reduction Techniques
3. Topic Modeling
4. Association Rule Mining

The whole idea of Unsupervised learning is to discover and find out the patterns rather than making predictions. So, unsupervised learning is different from supervised in mainly two aspects.

1. There is no labeled training data and no predictions.
2. The performance of models in unsupervised learning cannot be evaluated as there are no labels or correct answers.

Semi-supervised Learning

As the name suggests, semi-supervised learning lies somewhere in between both supervised and unsupervised learning. In fact, it uses both of the techniques. This type of learning is mainly relevant in scenarios when we are dealing with a mixed sort of dataset, which contains both labeled and unlabeled data. Sometimes it's just unlabeled data completely, but we label some part of it manually. The semi-supervised learning can be used on this small portion of labeled data to train the model and then use it for labeling the other remaining part of data, which can then be used for other purposes. This is also known as Pseudo-labeling as it labels the unlabeled data. To quote a simple example, we have a lot of images of different brands from social media and most of it is unlabeled. Now using semi-supervised learning, we can label some of these images manually and then train our model on the labeled images. We then use the model predictions to label the remaining images to transform the unlabeled data to labeled data completely.

The next step in semi-supervised learning is to retrain the model on the entire labeled dataset. The advantage that it offers is that the model gets trained on a bigger dataset, which was not the case earlier, and is now more robust and better at predictions. The other advantage is that semi-supervised learning saves a lot of effort and time that could go to manually label the data. The flipside of doing all this is that it's difficult to get high performance of the pseudo-labeling as it uses a small part of the labeled data to make the predictions. However, it is still a better option rather than manually labeling the data, which can be very expensive and time consuming at the same time.

Reinforcement Learning

The is the fourth and last kind of learning and is a little different in terms of the data usage and its predictions. Reinforcement Learning is a big research area in itself, and this entire book can be written just on it. We will not go too deep into this as this book focuses more on building machine learning models using PySpark. The main difference between the other kinds of Learning and Reinforcement Learning is that we need data, mainly historical data to training the models whereas Reinforcement Learning works on a reward system. It is primarily decision making based on certain actions that the agent takes to change its state trying in order to maximize the rewards. Let's break this down to individual elements using a visualization.

- Autonomous Agent: This is the main character in this whole learning process who is responsible for taking action. If it is a game, the agent makes the moves to finish or reach the end goal.
- Actions: These are sets of possible steps that the agent can take in order to move forward in the task. Each action will have some effect on the state of the agent and can result in either a reward or penalty. For example, in a game of Tennis, actions might be to serve, return, move left or right, etc.

- Reward: This is the key to making progress in reinforcement learning. Rewards enable the agents to take actions based on if it's positive rewards or penalties. It is a feedback mechanism that differentiates it from traditional supervised and unsupervised learning techniques
- Environment: This is the territory in which the agent gets to play in. Environment decides whether the actions that the agent takes results in rewards or penalties.
- State: The position the agent is in at any given point of time defines the state of the agent. To move forward or reach the end goal, the agent has to keep changing states in a positive direction to maximize the rewards.

The unique thing about Reinforcement Learning is that there is a feedback mechanism that drives the next behavior of the agent based on maximizing the total discounted reward. Some of the prominent applications that use Reinforcement Learning are self-driving cars, optimization of energy consumption, and the gaming domain. However, it can be also used to build recommender systems as well.

Conclusion

In this chapter we briefly looked at different types of Machine Learning approaches and some of the applications. In upcoming chapters, we will look at Supervised and Unsupervised Learning in detail using PySpark.

Data Processing

This chapter tries to cover all the main steps to process and massage data using PySpark. Although the data size we consider in this section is relatively small, but steps to process large datasets using PySpark remains exactly the same. Data processing is a critical step required to perform Machine Learning as we need to clean, filter, merge, and transform our data to bring it to the desired form so that we are able to train Machine Learning models. We will make use of multiple PySpark functions to perform data processing.

Load and Read Data

Assuming the fact that we have Spark version 2.3 installed, we start with importing and creating the `SparkSession` object first in order to use Spark.

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('data_processing').
      getOrCreate()
[In]: df=spark.read.csv('sample_data.csv',inferSchema=True,
      header=True)
```

We need to ensure that the data file is in the same folder where we have opened PySpark, or we can specify the path of the folder where the data resides along with the data file name. We can read multiple datafile formats with PySpark. We just need to update the read format argument in

accordance with the file format (csv, JSON, parquet, table, text). For a tab-separated file, we need to pass an additional argument while reading the file to specify the separator (`sep='\t'`). Setting the argument `inferSchema` to true indicates that Spark in the background will infer the datatypes of the values in the dataset on its own.

The above command creates a spark dataframe with the values from our sample data file. We can consider this an Excel spreadsheet in tabular format with columns and headers. We can now perform multiple operations on this Spark dataframe.

```
[In]: df.columns
[Out]: ['ratings', 'age', 'experience', 'family', 'mobile']
```

We can print the columns name lists that are present in the dataframe using the "columns" method. As we can see, we have five columns in our dataframe. To validate the number of columns, we can simply use the `length` function of Python.

```
[In]: len(df.columns)
[Out]: 5
```

We can use the `count` method to get the total number of records in the dataframe:

```
[In]: df.count
[Out] : 33
```

We have a total of 33 records in our dataframe. It is always a good practice to print the shape of the dataframe before proceeding with preprocessing as it gives an indication of the total number of rows and columns. There isn't any direct function available in Spark to check the shape of data; instead we need to combine the count and length of columns to print the shape.

```
[In]: print((df.count),(len(df.columns))
[Out]: ( 33,5)
```

Another way of viewing the columns in the dataframe is the `printSchema` method of spark. It shows the datatypes of the columns along with the column names.

```
[In]:df.printSchema()
[Out]: root
 |-- ratings: integer (nullable = true)
 |-- age: integer (nullable = true)
 |-- experience: double (nullable = true)
 |-- family: double (nullable = true)
 |-- Mobile: string (nullable = true)
```

The `nullable` property indicates if the corresponding column can assume null values (true) or not (false). We can also change the datatype of the columns as per the requirement.

The next step is to have a sneak peek into the dataframe to view the content. We can use the Spark `show` method to view the top rows of the dataframe.

```
[In]: df.show(3)
[Out]:
```

```
+-------+---+----------+------+-------+
|ratings|age|experience|family| mobile|
+-------+---+----------+------+-------+
|      3| 32|       9.0|     3|   Vivo|
|      3| 27|      13.0|     3|  Apple|
|      4| 22|       2.5|     0|Samsung|
|      4| 37|      16.5|     4|  Apple|
|      5| 27|       9.0|     1|     MI|
+-------+---+----------+------+-------+
only showing top 5 rows
```

We can see only see five records and all of the five columns since we passed the value 5 in the `show` method. In order to view only certain columns, we have to use the `select` method. Let us view only two columns (age and mobile):

```
[In]: df.select('age','mobile').show(5)
[Out]:
```

```
+---+-------+
|age| mobile|
+---+-------+
| 32|   Vivo|
| 27|  Apple|
| 22|Samsung|
| 37|  Apple|
| 27|     MI|
+---+-------+
only showing top 5 rows
```

`Select` function returned only two columns and five records from the dataframe. We will keep using the `select` function further in this chapter. The next function to be used is `describe` for analyzing the dataframe. It returns the statistical measures for each column of the dataframe. We will again use show along with describe, since describe returns the results as another dataframe.

```
[In]: df.describe().show()
[Out]:
```

```
+-------+------------------+------------------+------------------+------------------+------+
|summary|           ratings|               age|        experience|            family|mobile|
+-------+------------------+------------------+------------------+------------------+------+
|  count|                33|                33|                33|                33|    33|
|   mean| 3.5757575757575757|30.484848484848484|10.303030303030303| 1.8181818181818181|  null|
| stddev| 1.1188806636071336|  6.18527087180309| 6.770731351213326| 1.8448330794164254|  null|
|    min|                 1|                22|               2.5|                 0| Apple|
|    max|                 5|                42|              23.0|                 5|  Vivo|
+-------+------------------+------------------+------------------+------------------+------+
```

For numerical columns, it returns the measure of the center and spread along with the count. For nonnumerical columns, it shows the count and the min and max values, which are based on alphabetic order of those fields and doesn't signify any real meaning.

Adding a New Column

We can add a new column in the dataframe using the `withColumn` function of spark. Let us add a new column (age after 10 years) to our dataframe by using the age column. We simply add 10 years to each value in the age column.

```
[In]: df.withColumn("age_after_10_yrs",(df["age"]+10)).
      show(10,False)
[Out]:
```

```
+-------+---+----------+------+-------+----------------+
|ratings|age|experience|family|mobile |age_after_10_yrs|
+-------+---+----------+------+-------+----------------+
|3      |32 |9.0       |3     |Vivo   |42              |
|3      |27 |13.0      |3     |Apple  |37              |
|4      |22 |2.5       |0     |Samsung|32              |
|4      |37 |16.5      |4     |Apple  |47              |
|5      |27 |9.0       |1     |MI     |37              |
|4      |27 |9.0       |0     |Oppo   |37              |
|5      |37 |23.0      |5     |Vivo   |47              |
|5      |37 |23.0      |5     |Samsung|47              |
|3      |22 |2.5       |0     |Apple  |32              |
|3      |27 |6.0       |0     |MI     |37              |
+-------+---+----------+------+-------+----------------+
only showing top 10 rows
```

As we can observe, we have a new column in the dataframe. The `show` function helps us to view the new column values, but in order to add the new column to the dataframe, we will need to assign this to an exisiting or new dataframe.

```
[In]: df= df.withColumn("age_after_10_yrs",(df["age"]+10))
```

This line of code ensures that the changes takes place and the dataframe now contains the new column (age after 10 yrs).

To change the datatype of the age column from integer to double, we can make use of the cast method in Spark. We need to import the DoubleType from pyspark.types:

```
[In]: from pyspark.sql.types import StringType,DoubleType
[In]: df.withColumn('age_double',df['age'].cast(DoubleType())).
      show(10,False)
[Out]:
```

```
+-------+---+----------+------+-------+----------+
|ratings|age|experience|family|mobile |age_double|
+-------+---+----------+------+-------+----------+
|3      |32 |9.0       |3     |Vivo   |32.0      |
|3      |27 |13.0      |3     |Apple  |27.0      |
|4      |22 |2.5       |0     |Samsung|22.0      |
|4      |37 |16.5      |4     |Apple  |37.0      |
|5      |27 |9.0       |1     |MI     |27.0      |
|4      |27 |9.0       |0     |Oppo   |27.0      |
|5      |37 |23.0      |5     |Vivo   |37.0      |
|5      |37 |23.0      |5     |Samsung|37.0      |
|3      |22 |2.5       |0     |Apple  |22.0      |
|3      |27 |6.0       |0     |MI     |27.0      |
+-------+---+----------+------+-------+----------+
only showing top 10 rows
```

So the above command creates a new column (age_double) that has converted values of age from integer to double type.

Filtering Data

Filtering records based on conditions is a common requirement when dealing with data. This helps in cleaning the data and keeping only relevant records. Filtering in PySpark is pretty straight-forward and can be done using the filter function.

Condition 1

This is the most basic type of filtering based on only one column of a dataframe. Let us say we want to fetch the records for only 'Vivo' mobile:

```
[In]: df.filter(df['mobile']=='Vivo').show()
[Out]:
```

```
+-------+---+----------+------+------+
|ratings|age|experience|family|mobile|
+-------+---+----------+------+------+
|      3| 32|       9.0|     3|  Vivo|
|      5| 37|      23.0|     5|  Vivo|
|      4| 37|       6.0|     0|  Vivo|
|      5| 37|      13.0|     1|  Vivo|
|      4| 37|       6.0|     0|  Vivo|
+-------+---+----------+------+------+
```

We have all records for which `Mobile` column has 'Vivo' values. We can further select only a few columns after filtering the records. For example, if we want to view the age and ratings for people who use 'Vivo' mobile, we can do that by using the `select` function after filtering records.

```
[In]: df.filter(df['mobile']=='Vivo').select('age','ratings',
      'mobile').show()
[Out]:
```

```
+---+-------+------+
|age|ratings|mobile|
+---+-------+------+
| 32|      3|  Vivo|
| 37|      5|  Vivo|
| 37|      4|  Vivo|
| 37|      5|  Vivo|
| 37|      4|  Vivo|
+---+-------+------+
```

Condition 2

This involves multiple columns-based filtering and returns records only if all conditions are met. This can be done in multiple ways. Let us say, we want to filter only 'Vivo' users and only those with experience of more than 10 years.

```
[In]: df.filter(df['mobile']=='Vivo').filter(df['experience']
      >10).show()
[Out]:
```

```
+-------+---+----------+------+------+
|ratings|age|experience|family|Mobile|
+-------+---+----------+------+------+
|      5| 37|      23.0|   5.5|  Vivo|
|      5| 37|      13.0|   1.0|  Vivo|
+-------+---+----------+------+------+
```

We use more than one filter function in order to apply those conditions on individual columns. There is another approach to achieve the same results as mentioned below.

```
[In]: df.filter((df['mobile']=='Vivo')&(df['experience'] >10)).
      show()
[Out]:
```

```
+-------+---+----------+------+------+
|ratings|age|experience|family|Mobile|
+-------+---+----------+------+------+
|      5| 37|      23.0|   5.5|  Vivo|
|      5| 37|      13.0|   1.0|  Vivo|
+-------+---+----------+------+------+
```

Distinct Values in Column

If we want to view the distinct values for any dataframe column, we can use the `distinct` method. Let us view the distinct values for the `mobile` column in the dataframe.

```
[In]: df.select('mobile').distinct().show()
[Out]:
```

```
+-------+
| mobile|
+-------+
|     MI|
|   Oppo|
|Samsung|
|   Vivo|
|  Apple|
+-------+
```

For getting the count of distinct values in the column, we can simply use `count` along with the `distinct` function.

```
[In]: df.select('mobile').distinct().count()
[Out]: 5
```

Grouping Data

`Grouping is a` very useful way to understand various aspects of the dataset. It helps to group the data based on columns values and extract insights. It can be used with multiple other functions as well. Let us see an example of the `groupBy` method using the dataframe.

```
[In]: df.groupBy('mobile').count().show(5,False)
[Out]:
```

```
+-------+-----+
|mobile |count|
+-------+-----+
|MI     |8    |
|Oppo   |7    |
|Samsung|6    |
|Vivo   |5    |
|Apple  |7    |
+-------+-----+
```

Here, we are grouping all the records based on the categorical values in the `mobile` column and counting the number of records for each category using the `count` method. We can further refine these results by making use of the `orderBy` method to sort them in a defined order.

```
[In]: df.groupBy('mobile').count().orderBy('count',ascending=
      False).show(5,False)
[Out]:
```

```
+-------+-----+
|mobile |count|
+-------+-----+
|MI     |8    |
|Oppo   |7    |
|Apple  |7    |
|Samsung|6    |
|Vivo   |5    |
+-------+-----+
```

Now, the count of the `mobiles` are sorted in descending order based on each category.

We can also apply the `groupBy` method to calculate statistical measures such as mean value, sum, min, or max value for each category. So let's see what is the mean value of the rest of the columns.

```
[In]: df.groupBy('mobile').mean().show(5,False)
[Out]:
```

```
+-------+------------------+------------------+------------------+------------------+
|mobile |avg(ratings)      |avg(age)          |avg(experience)   |avg(family)       |
+-------+------------------+------------------+------------------+------------------+
|MI     |3.5               |30.125            |10.1875           |1.375             |
|Oppo   |2.857142857142857 |28.428571428571427|10.357142857142858|1.4285714285714286|
|Samsung|4.166666666666667 |28.666666666666668|8.666666666666666 |1.8333333333333333|
|Vivo   |4.2               |36.0              |11.4              |1.8               |
|Apple  |3.4285714285714284|30.571428571428573|11.0              |2.7142857142857144|
+-------+------------------+------------------+------------------+------------------+
```

The mean method gives us the average of age, ratings, experience, and family size columns for each mobile brand. We can get the aggregated sum as well for each mobile brand by using the sum method along with groupBy.

```
[In]: df.groupBy('mobile').sum().show(5,False)
[Out]:
```

```
+-------+------------+--------+---------------+-----------+
|mobile |sum(ratings)|sum(age)|sum(experience)|sum(family)|
+-------+------------+--------+---------------+-----------+
|MI     |28          |241     |81.5           |11         |
|Oppo   |20          |199     |72.5           |10         |
|Samsung|25          |172     |52.0           |11         |
|Vivo   |21          |180     |57.0           |9          |
|Apple  |24          |214     |77.0           |19         |
+-------+------------+--------+---------------+-----------+
```

Let us now view the min and max values of users data for every mobile brand.

```
[In]: df.groupBy('mobile').max().show(5,False)
[Out]:
```

```
+-------+------------+--------+---------------+-----------+
|mobile |max(ratings)|max(age)|max(experience)|max(family)|
+-------+------------+--------+---------------+-----------+
|MI     |5           |42      |23.0           |5          |
|Oppo   |4           |42      |23.0           |2          |
|Samsung|5           |37      |23.0           |5          |
|Vivo   |5           |37      |23.0           |5          |
|Apple  |4           |37      |16.5           |5          |
+-------+------------+--------+---------------+-----------+
```

```
[In]:df.groupBy('mobile').min().show(5,False)
[Out]:
```

```
+-------+------------+--------+---------------+-----------+
|mobile |min(ratings)|min(age)|min(experience)|min(family)|
+-------+------------+--------+---------------+-----------+
|MI     |1           |27      |2.5            |0          |
|Oppo   |2           |22      |6.0            |0          |
|Samsung|2           |22      |2.5            |0          |
|Vivo   |3           |32      |6.0            |0          |
|Apple  |3           |22      |2.5            |0          |
+-------+------------+--------+---------------+-----------+
```

Aggregations

We can use the agg function as well to achieve the similar kinds of results as above. Let's use the agg function in PySpark for simply taking the sum of total experience for each mobile brand.

```
[In]: df.groupBy('mobile').agg({'experience':'sum'}).
      show(5,False)
[Out]:
```

```
+-------+---------------+
|mobile |sum(experience)|
+-------+---------------+
|MI     |81.5           |
|Oppo   |72.5           |
|Samsung|52.0           |
|Vivo   |57.0           |
|Apple  |77.0           |
+-------+---------------+
```

So here we simply use the agg function and pass the column name (experience) for which we want the aggregation to be done.

User-Defined Functions (UDFs)

UDFs are widely used in data processing to apply certain transformations to the dataframe. There are two types of UDFs available in PySpark: Conventional UDF and Pandas UDF. Pandas UDF are much more powerful in terms of speed and processing time. We will see how to use both types of UDFs in PySpark. First, we have to import `udf` from PySpark functions.

```
[In]: from pyspark.sql.functions import udf
```

Now we can apply basic UDF either by using a lambda or typical Python function.

Traditional Python Function

We create a simple Python function, which returns the category of price range based on the mobile brand:

```
[In]:
def price_range(brand):
    if brand in ['Samsung','Apple']:
        return 'High Price'
    elif brand =='MI':
        return 'Mid Price'
    else:
        return 'Low Price'
```

In the next step, we create a UDF (`brand_udf`) that uses this function and also captures its datatype to apply this tranformation on the mobile column of the dataframe.

```
[In]: brand_udf=udf(price_range,StringType())
```

In the final step, we apply the udf(brand_udf) to the mobile column of dataframe and create a new colum (price_range) with new values.

```
[In]: df.withColumn('price_range',brand_udf(df['mobile'])).
      show(10,False)
[Out]:
```

```
+-------+---+----------+------+-------+-----------+
|ratings|age|experience|family|mobile |price_range|
+-------+---+----------+------+-------+-----------+
|3      |32 |9.0       |3     |Vivo   |Low Price  |
|3      |27 |13.0      |3     |Apple  |High Price |
|4      |22 |2.5       |0     |Samsung|High Price |
|4      |37 |16.5      |4     |Apple  |High Price |
|5      |27 |9.0       |1     |MI     |Mid Price  |
|4      |27 |9.0       |0     |Oppo   |Low Price  |
|5      |37 |23.0      |5     |Vivo   |Low Price  |
|5      |37 |23.0      |5     |Samsung|High Price |
|3      |22 |2.5       |0     |Apple  |High Price |
|3      |27 |6.0       |0     |MI     |Mid Price  |
+-------+---+----------+------+-------+-----------+
only showing top 10 rows
```

Using Lambda Function

Instead of defining a traditional Python function, we can make use of the lambda function and create a UDF in a single line of code as shown below. We categorize the age columns into two groups (young, senior) based on the age of the user.

```
[In]: age_udf = udf(lambda age: "young" if age <= 30 else
      "senior", StringType())
[In]: df.withColumn("age_group", age_udf(df.age)).
      show(10,False)
[Out]:
```

```
+-------+---+----------+------+-------+---------+
|ratings|age|experience|family|mobile |age_group|
+-------+---+----------+------+-------+---------+
|3      |32 |9.0       |3     |Vivo   |senior   |
|3      |27 |13.0      |3     |Apple  |young    |
|4      |22 |2.5       |0     |Samsung|young    |
|4      |37 |16.5      |4     |Apple  |senior   |
|5      |27 |9.0       |1     |MI     |young    |
|4      |27 |9.0       |0     |Oppo   |young    |
|5      |37 |23.0      |5     |Vivo   |senior   |
|5      |37 |23.0      |5     |Samsung|senior   |
|3      |22 |2.5       |0     |Apple  |young    |
|3      |27 |6.0       |0     |MI     |young    |
+-------+---+----------+------+-------+---------+
only showing top 10 rows
```

Pandas UDF (Vectorized UDF)

Like mentioned earlier, Pandas UDFs are way faster and efficient compared to their peers. There are two types of Pandas UDFs:

- Scalar
- GroupedMap

Using Pandas UDF is quite similar to using the basic UDfs. We have to first import `pandas_udf` from PySpark functions and apply it on any particular column to be tranformed.

```
[In]: from pyspark.sql.functions import pandas_udf
```

In this example, we define a Python function that calculates the number of years left in a user's life assuming a life expectancy of 100 years. It is a very simple calculation: we subtract the age of the user from 100 using a Python function.

```
[In]:
def remaining_yrs(age):
    yrs_left=(100-age)
   return yrs_left

[In]: length_udf = pandas_udf(remaining_yrs, IntegerType())
```

Once we create the Pandas UDF (length_udf) using the Python function (remaining_yrs), we can apply it to the age column and create a new column yrs_left.

```
[In]:df.withColumn("yrs_left", length_udf(df['age'])).
     show(10,False)
[Out]:
```

ratings	age	experience	family	mobile	yrs_left
3	32	9.0	3	Vivo	68
3	27	13.0	3	Apple	73
4	22	2.5	0	Samsung	78
4	37	16.5	4	Apple	63
5	27	9.0	1	MI	73
4	27	9.0	0	Oppo	73
5	37	23.0	5	Vivo	63
5	37	23.0	5	Samsung	63
3	22	2.5	0	Apple	78
3	27	6.0	0	MI	73

only showing top 10 rows

Pandas UDF (Multiple Columns)

We might face a situation where we need multiple columns as input to create a new column. Hence, the below example showcases the method of applying a Pandas UDF on multiple columns of a dataframe. Here we will create a new column that is simply the product of the ratings and

experience columns. As usual, we define a Python function and calculate the product of the two columns.

```
[In]:
def prod(rating,exp):
    x=rating*exp
    return x
[In]: prod_udf = pandas_udf(prod, DoubleType())
```

After creating the Pandas UDF, we can apply it on both of the columns (`ratings`, `experience`) to form the new column (`product`).

```
[In]:df.withColumn("product",prod_udf(df['ratings'],
     df['experience'])).show(10,False)
[Out]:
```

```
+-------+---+----------+------+-------+-------+
|ratings|age|experience|family|mobile |product|
+-------+---+----------+------+-------+-------+
|3      |32 |9.0       |3     |Vivo   |27.0   |
|3      |27 |13.0      |3     |Apple  |39.0   |
|4      |22 |2.5       |0     |Samsung|10.0   |
|4      |37 |16.5      |4     |Apple  |66.0   |
|5      |27 |9.0       |1     |MI     |45.0   |
|4      |27 |9.0       |0     |Oppo   |36.0   |
|5      |37 |23.0      |5     |Vivo   |115.0  |
|5      |37 |23.0      |5     |Samsung|115.0  |
|3      |22 |2.5       |0     |Apple  |7.5    |
|3      |27 |6.0       |0     |MI     |18.0   |
+-------+---+----------+------+-------+-------+
only showing top 10 rows
```

Drop Duplicate Values

We can use the `dropDuplicates` method in order to remove the duplicate records from the dataframe. The total number of records in this dataframe are 33, but it also contains 7 duplicate records, which can easily be confirmed by droping those duplicate records as we are left with only 26 rows.

```
[In]: df.count()
[Out]: 33
[In]: df=df.dropDuplicates()
[In]: df.count()
[Out]: 26
```

Delete Column

We can make use of the `drop` function to remove any of the columns from the dataframe. If we want to remove the `mobile` column from the dataframe, we can pass it as an argument inside the `drop` function.

```
[In]: df_new=df.drop('mobile')
[In]: df_new.show()
[Out]:
```

```
+-------+---+----------+------+
|ratings|age|experience|family|
+-------+---+----------+------+
|      3| 32|       9.0|     3|
|      3| 27|      13.0|     3|
|      4| 22|       2.5|     0|
|      4| 37|      16.5|     4|
|      5| 27|       9.0|     1|
|      4| 27|       9.0|     0|
|      5| 37|      23.0|     5|
|      5| 37|      23.0|     5|
|      3| 22|       2.5|     0|
|      3| 27|       6.0|     0|
+-------+---+----------+------+
only showing top 10 rows
```

Writing Data

Once we have the processing steps completed, we can write the clean dataframe to the desired location (local/cloud) in the required format.

CSV

If we want to save it back in the original csv format as single file, we can use the `coalesce` function in spark.

```
[In]: pwd
[Out]: ' /home/jovyan/work '
[In]: write_uri= '  /home/jovyan/work/df_csv '
[In]: df.coalesce(1).write.format("csv").
option("header","true").save(write_uri)
```

Parquet

If the dataset is huge and involves a lot of columns, we can choose to compress it and convert it into a parquet file format. It reduces the overall size of the data and optimizes the performance while processing data because it works on subsets of required columns instead of the entire data. We can convert and save the dataframe into the parquet format easily by mentioning the format as `parquet` as shown below".

```
[In]: parquet_uri='/home/jovyan/work/df_parquet'
[In]: df.write.format('parquet').save(parquet_uri)
```

Note The complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.3 and higher versions.

Conclusion

In this chapter, we got familiar with a few of the functions and techniques to handle and tranform the data using PySpark. There are many more methods that can be explored further to preprocess the data using PySpark, but the fundamental steps to clean and prepare the data for Machine Learning have been covered in this chapter.

Linear Regression

As we talked about Machine Learning in the previous chapter, it's a very vast field and there are multiple algorithms that fall under various categories, but Linear Regression is one of the most fundamental machine learning algorithms. This chapter focuses on building a Linear Regression model with PySpark and dives deep into the workings of an LR model. It will cover various assumptions to be considered before using LR along with different evaluation metrics. But before even jumping into trying to understand Linear Regression, we must understand the types of variables.

Variables

Variables capture data information in different forms. There are mainly two categories of variables that are used widely as depicted in Figure 4-1.

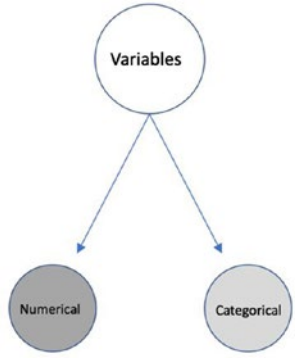


Figure 4-1. *Types of Variables*

We can even further break down these variables into sub-categories, but we will stick to these two types throughout this book.

Numerical variables are those kinds of values that are quantitative in nature such as numbers (integer/floats). For example, salary records, exam scores, age or height of a person, and stock prices all fall under the category of Numerical variables.

Categorical variables, on the other hand, are qualitative in nature and mainly represent categories of data being measured. For example, colors, outcome (Yes/No), Ratings (Good/Poor/Avg).

For building any sort of machine learning model we need to have input and output variables. Input variables are those values that are used to build and train the machine learning model to predict the output or target variable. Let's take a simple example. Suppose we want to predict the salary of a person given the age of the person using machine learning. In this case, the salary is our output/target/dependent variable as it depends on age, which is known as the input or independent variable. Now the output variable can be categorical or numerical in nature and depending on its type, machine learning models are chosen.

Now coming back to Linear Regression, it is primarily used in the cases of when we are trying to predict numerical output variable. Linear Regression is used to predict a line that fits the input data, points the best possible way, and can help in predictions for unseen data, but the point to notice here is how can a model learn just from "age" and predict the salary amount for a given person? For sure, there needs to be some sort of relationship between these two variables (salary and age). There are two major types of variable relationships:

- Linear
- Nonlinear

The notion of a linear relationship between any two variables suggests that both are proportional to each other in some ways. The correlation between any two variables gives us an indication on how strong or weak is the linear relationship between them. The correlation coefficient can range from -1 to + 1. Negative correlation means that by increasing one of the variables, the other variable decreases. For example, power and mileage of a vehicle can be negatively correlated because as we increase power, the mileage of a vehicle comes down. On the other hand, salary and years of work experience are an example of positively correlated variables. Non-linear relationships are comparatively complex in nature and hence require an extra amount of details to predict the target variables. For example, a self-driving car, the relationship between input variables such as terrain, signal system, and pedestrian to the speed of the car are nonlinear.

Note The next section includes theory behind Linear Regression and might be redundant for many readers. Please feel free to skip the section if this is the case.

Theory

Now that we understand the basics of variables and the relationships between them, let's build on the example of age and salary to understand Linear Regression in depth.

The overall objective of Linear Regression is to predict a straight line through the data, so that the vertical distance of each of these points is minimal from that line. So, in this case, we will predict the salary of a person given an age. Let's assume we have records of four people that capture age and their respective salaries as shown in Table 4-1.

Table 4-1. *Example Dataset*

Sr. No	Age	Salary ('0000 $)
1	20	5
2	30	10
3	40	15
4	50	22

We have an input variable (age) at our disposal to make use of in order to predict the salary (which we will do at a later stage of this book), but let's take a step back. Let's assume that all we have with us at the start is just the salary values of these four people. The salary is plotted for each person in the Figure 4-2.

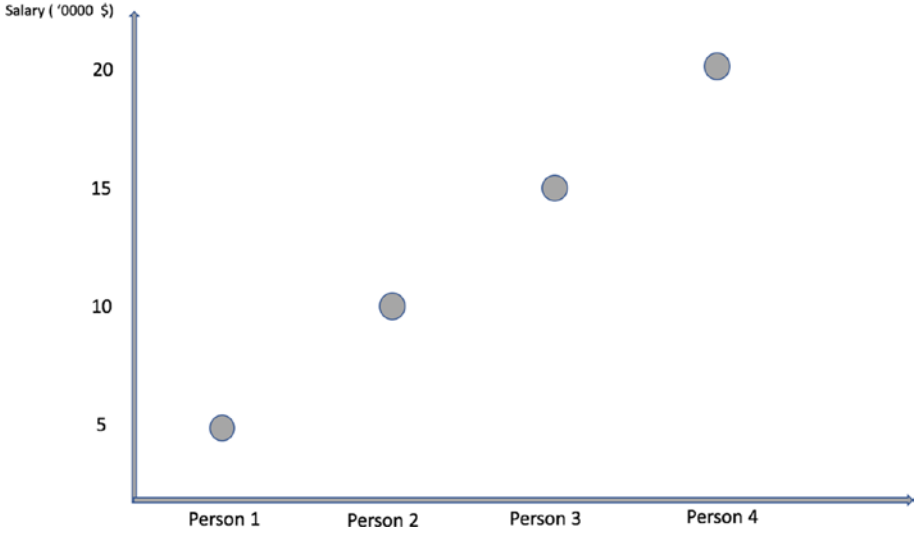


Figure 4-2. *Scatter plot of Salary*

Now, we if we were to predict the salary of the fifth person (new person) based on the salaries of these earlier people, the best possible way to predict that is to take an average/mean of existing salary values. That would be the best prediction given this information. It is like building a Machine Learning model but without any input data (since we are using the output data as input data).

Let's go ahead and calculate the average salary for these given salary values.

$$\text{Avg. Salary} = \frac{(5+10+15+22)}{4} = 13$$

So, the best prediction of the salary value for the next person is 13. Figure 4-3 showcases the salary values for each person along with the mean value (the best fit line in the case of using only one variable).

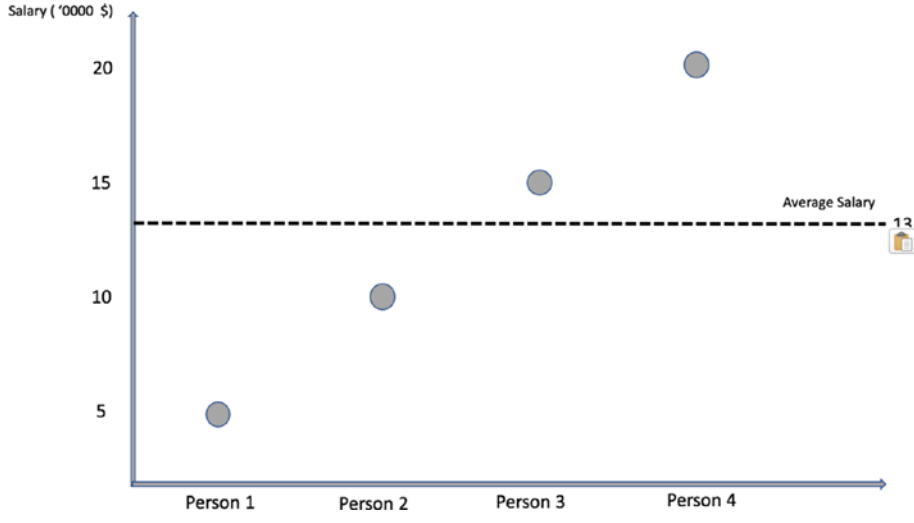


Figure 4-3. *Best Fit Line plot*

The line for the mean value as shown in Figure 4-3 is possibly the best fit line in this scenario for these data points because we are not using any other variable apart from salary itself. If we take a look closely, none of the earlier salary values lies on this best fit line; there seems to be some amount of separation from the mean salary value as shown in Figure 4-4. These are also known as errors. If we go ahead and calculate the total sum of this distance and add them up, it becomes 0, which makes sense since it's the mean value of all the data points. So, instead of simply adding them, we square each error and then add them up.

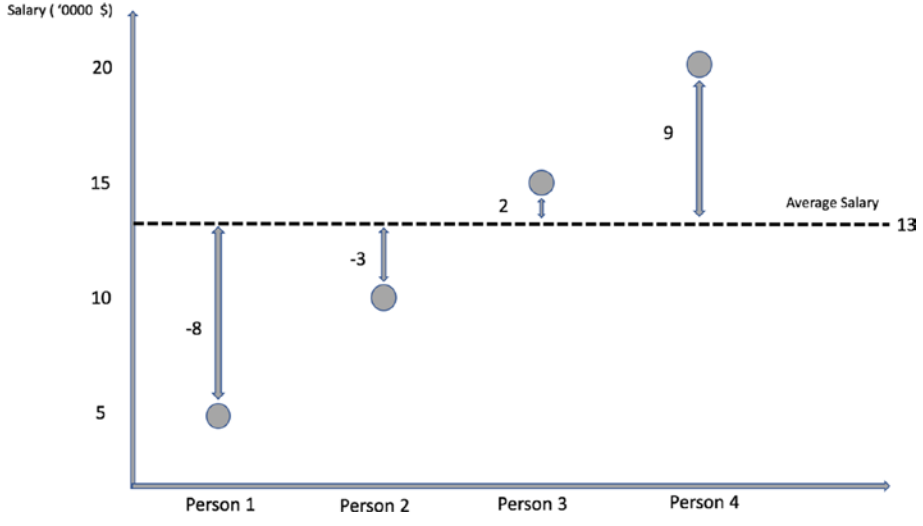


Figure 4-4. *Residuals Plot*

Sum of Squared errors = 64 + 9 + 4 + 81 = 158.

So, adding up the squared residuals, it gives us a total value of 158, which is known as the sum of squared errors (SSE).

Note We have not used any input variable so far to calculate the SSE.

Let us park this score for now and include the input variable (age of the person) as well to predict the salary of the person. Let's start with visualizing the relationship between Age and Salary of the person as shown in Figure 4-5.

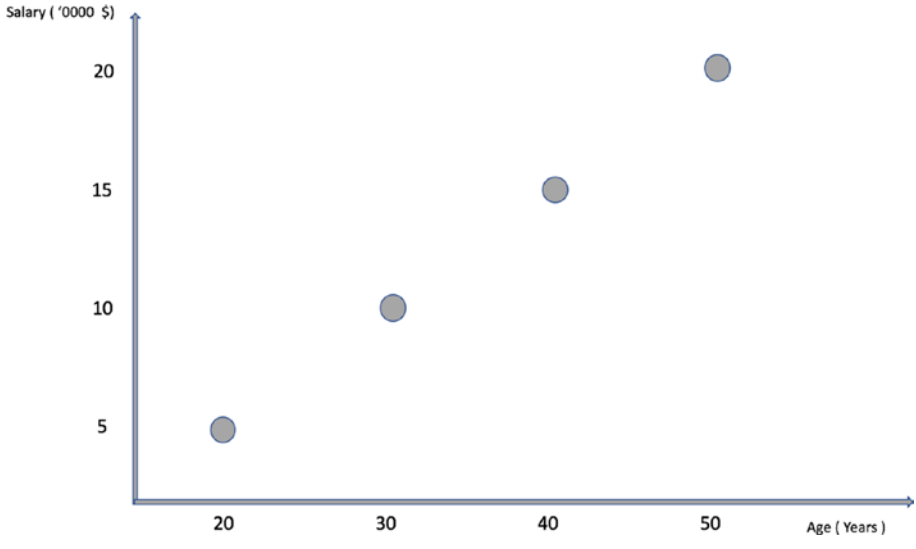


Figure 4-5. *Correlation plot between Salary and Age*

As we can observe, there seems to be a clear positive correlation between years of work experience and salary value, and it is a good thing for us because it indicates that the model would be able to predict the target value (salary) with a good amount of accuracy due to a strong linear relationship between input(age) and output(salary). As mentioned earlier, the overall aim of Linear Regression is to come up with a straight line that fits the data point in such a way that the squared difference between the actual target value and predicted value is minimized. Since it is a straight line, we know in linear algebra the equation of a straight line is

$y = mx + c$ and the same is shown in Figure 4-6.

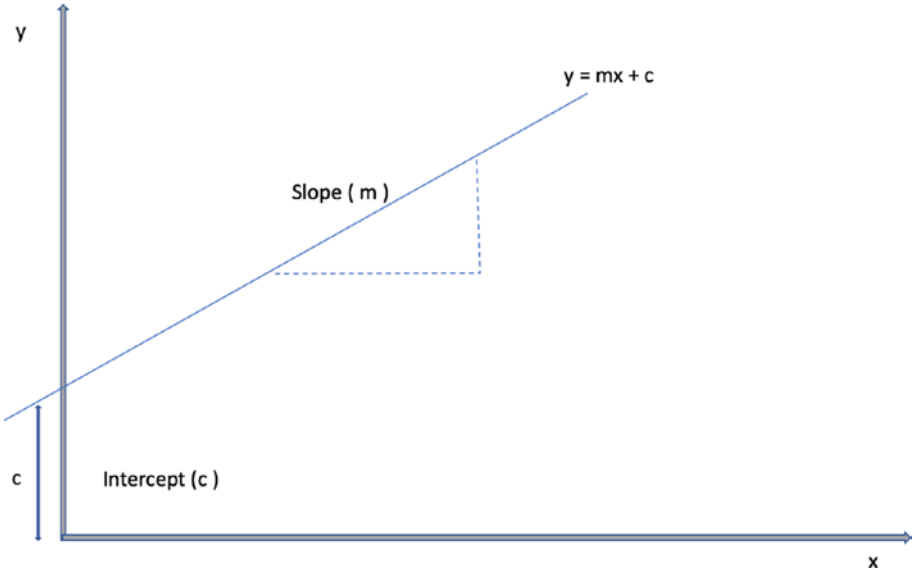


Figure 4-6. *Straight Line plot*

where,

m = slope of the line ($\frac{x_2 - x_1}{y_2 - y_1}$)

x = value at x-axis

y= value at y-axis

c = intercept (value of y at x = 0)

Since Linear Regression is also finding out the straight line, the Linear Regression equation becomes

$$\boldsymbol{y = B_0 + B_1 * x}$$

(since we are using only 1 input variable, i.e., Age)

where:

y= salary (prediction)

B0=intercept (value of Salary when Age is 0)

B1= slope or coefficient of Salary

x= Age

Now, you may ask, if there can be multiple lines that can be drawn through the data points (as shown in Figure 4-7) and how to figure out which is the best fit line.

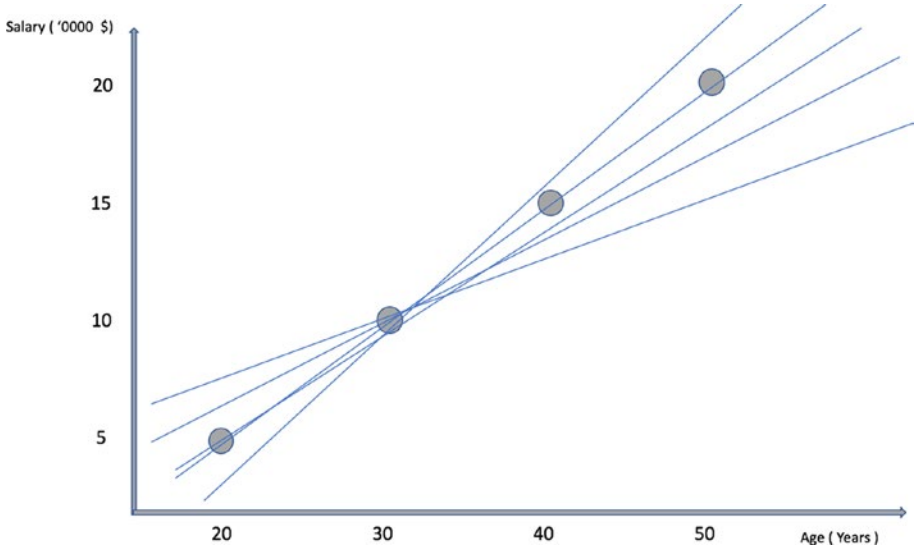


Figure 4-7. *Possible Straight lines through data*

The first criteria to find out the best fit line is that it should pass through the centroid of the data points as shown in Figure 4-8. In our case, the centroid values are

$$\text{mean (Age)} = \frac{(20+30+40+50)}{4} = 35$$

$$\text{mean (Salary)} = \frac{(5+10+15+22)}{4} = 13$$

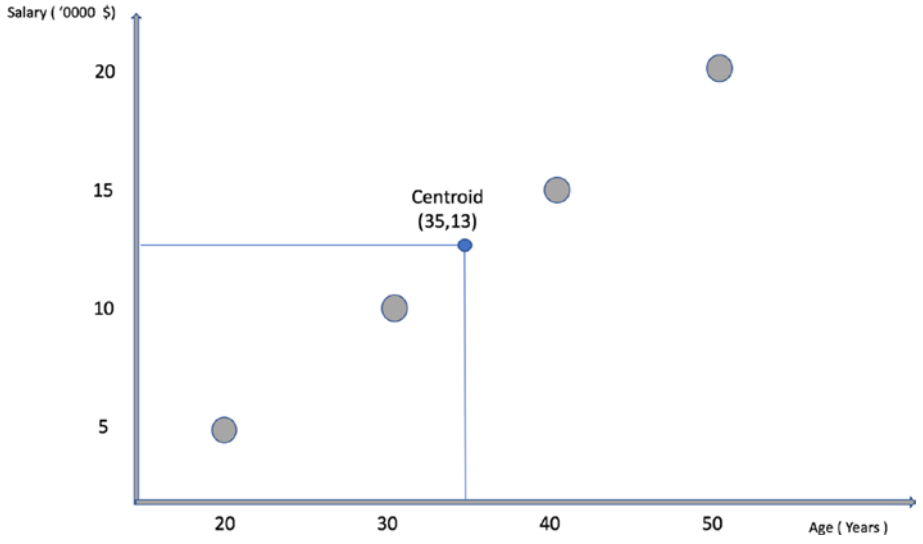


Figure 4-8. *Centroids of Data*

The second criteria are that it should be able to minimize the sum of squared errors. We know our regression line equation is equal to

$$\boldsymbol{y = B_0 + B_1 * x}$$

Now the objective of using Linear Regression is to come up with the most optimal values of the Intercept (B_0) and Coefficient (B_1) so that the residuals/errors are minimized to the maximum extent.

We can easily find out values of B_0 & B_1 for our dataset by using the below formulas.

$$B_1 = \frac{\sum(x_i - x_{mean}) * (y_i - y_{mean})}{\sum(x_i - x_{mean})^2}$$

$$B_0 = y_{mean} - B_1 * (x_{mean})$$

Table 4-2 showcases the calculation of slope and intercept for Linear Regression using input data.

Table 4-2. *Calculation of Slope and Intercept*

Age	Salary	Age variance (diff from mean)	Salary variance (diff. from mean)	Covariance (Product)	Age Variance (squared)
20	5	-15	-8	120	225
30	10	-5	-3	15	25
40	15	5	2	10	25
50	22	15	9	135	225

Mean (Age) = 35
Mean (Salary) =13

The covariance between any two variables (age and salary) is defined as the product of the distances of each variable (age and salary) from their mean. In short, the product of the variance of age and salary is known as covariance. Now that we have the covariance product and Age variance squared values, we can go ahead and calculate the values of slope and intercept of the Linear Regression line:

$$B_1 = \frac{\sum(Covariance)}{\sum(Age\ Variance\ Squared)}$$

$$= \frac{280}{500}$$

$$=0.56$$

$$B_0 = 13 - (0.56 * 35)$$

$$= -6.6$$

Our final Linear Regression equation becomes

$$\boldsymbol{y = B_0 + B_1 * x}$$

Salary = -6.6 + (0.56 * Age)

We can now predict any of the salary values using this equation at any given age. For example, the model would have predicted the salary of the first person to be something like this:

Salary (1st person) = -6.6 + (0.56*20)

= 4.6 ($ '0000)

Interpretation

Slope (B_1= 0.56) here means for an increase in 1 year of Age of the person, the salary also increases by an amount of $5,600.

Intercept does not always make sense in terms of deriving meaning out of its value. Like in this example, the value of negative 6.6 suggests that if the person is not yet born (Age =0), the salary of that person would be negative $66,000.

Figure 4-9 shows the final regression line for our dataset.

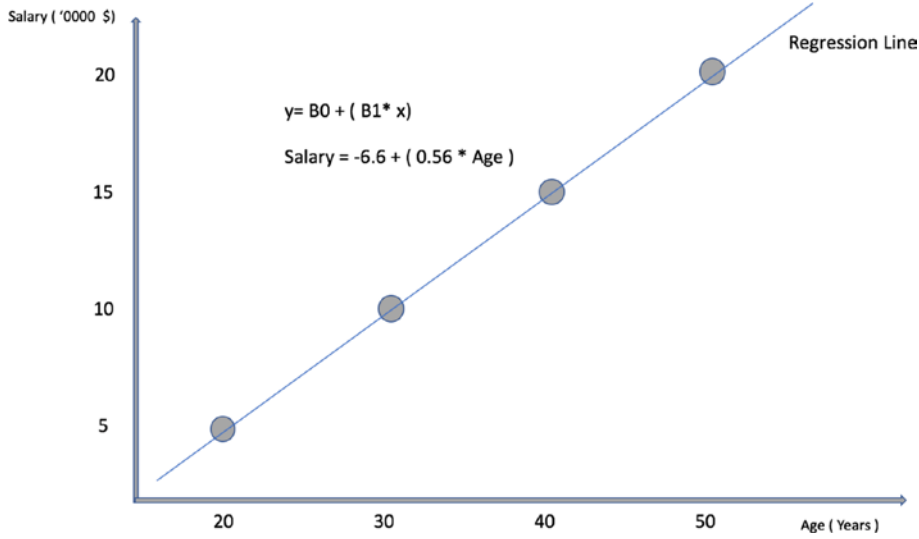


Figure 4-9. *Regression Line*

Let's predict the salary for all four records in our data using the regression equation, and compare the difference from actual salaries as shown in Table 4-3.

Table 4-3. *Difference Between Predictions and Actual Values*

Age	Salary	Predicted Salary	Difference /Error
20	5	4.6	-0.4
30	10	10.2	0.2
40	15	15.8	0.8
50	22	21.4	-0.6

In a nutshell, Linear Regression comes up with the most optimal values for the Intercept (B_0) and coefficients (B_1, B_2) so that the difference (error) between the predicted values and the target variables is minimum.

But the question remains: Is it a good fit?

Evaluation

There are multiple ways to evaluate the goodness of fit of the Regression line, but one of the ways is by using the coefficient of determination (r^{square}) value. Remember we had calculated the sum of squared errors when we had only used the output variable itself and its value was 158. Now let us recalculate the SSE for this model, which we have built using an input variable. Table 4-4 shows the calculation for the new SSE after using Linear Regression.

Table 4-4. *Reduction in SSE After Using Linear Regression*

Age	Salary	Predicted Salary	Difference /Error	Squared Error	old SSE
20	5	4.6	-0.4	0.16	64
30	10	10.2	0.2	0.04	9
40	15	15.8	0.8	0.64	4
50	22	21.4	-0.6	0.36	81

As we can observe, the total sum of the squared difference has reduced significantly from 158 to only 1.2, which has happened because of using Linear Regression. The variance in the target variable (Salary) can be explained with help of regression (due to usage of input variable – Age). So, OLS works toward reducing the overall sum of squared errors. The total sum of squared errors is a combination of two types:

TSS (Total Sum of Squared Errors) = SSE (Sum of squared errors) + SSR (Residual Sum of squared errors)

The total sum of squares is the sum of the squared difference between the actual and the mean values and is always fixed. This was equal to 158 in our example.

The SSE is the squared difference from the actual to predicted values of the target variable, which reduced to 1.2 after using Linear Regression.

SSR is the sum of squares explained by regression and can be calculated by (TSS – SSE).

$$SSR = 158 - 1.2 = 156.8$$

$$r^{square}\text{ (Coefficient of determination)} = \frac{SSR}{TSS} == \frac{156.8}{158} = 0.99$$

This percentage indicates that our Linear Regression model can predict with 99 % accuracy in terms of predicting the salary amount given the age of the person. The other 1% can be attributed toward errors that cannot be explained by the model. Our Linear Regression line fits the model really well, but it can also be a case of overfitting. Overfitting occurs when your model predicts with high accuracy on training data, but its performance drops on the unseen/test data. The technique to address the issues of overfitting is known as regularization, and there are different types of regularization techniques. In terms of Linear Regression, one can use Ridge, Lasso, or Elasticnet Regularization techniques to handle overfitting.

Ridge Regression is also known as L2 regularization and focuses on restricting the coefficient values of input features close to zero whereas Lasso regression (L1) makes some of the coefficients zero in order to improve generalization of the model. Elasticnet is a combination of both techniques.

At the end of the day, Regression is a still a parametric-driven approach and assumes few underlying patterns about distributions of input data points. If the input data does not affiliate to those assumptions, the Linear Regression model does not perform well. Hence it is important to go over these assumptions very quickly in order to know them before using the Linear Regression model.

Assumptions:

- There must be a linear relationship between the input variable and output variable.
- The independent variables (input features) should not be correlated to each other (also known as multicollinearity).
- There must be no correlation between the residuals/ error values.

- There must be a linear relationship between the residuals and the output variable.
- The residuals/error values must be normally distributed.

Code

This section of the chapter focuses on building a Linear Regression Model from scratch using PySpark and Jupyter Notebook.

Although we saw a simple example of only one input variable to understand Linear Regression, this is seldom the case. The majority of the time, the dataset would contain multiple variables and hence building a multivariable Regression model makes more sense in such a situation. The Linear Regression equation then looks something like this:

$$y = B_0 + B_1 * X_1 + B_2 * X_2 + B_3 * X_3 + \ldots$$

Note The complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.3 and higher versions.

Let's build a Linear Regression model using Spark's MLlib library and predict the target variable using the input features.

Data Info

The dataset that we are going to use for this example is a dummy dataset and contains a total of 1,232 rows and 6 columns. We have to use 5 input variables to predict the target variable using the Linear Regression model.

Step 1: Create the SparkSession Object

We start the Jupyter Notebook and import SparkSession and create a new SparkSession object to use Spark:

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('lin_reg').getOrCreate()
```

Step 2: Read the Dataset

We then load and read the dataset within Spark using Dataframe. We have to make sure we have opened the PySpark from the same directory folder where the dataset is available or else we have to mention the directory path of the data folder:

```
[In]: df=spark.read.csv('Linear_regression_dataset.csv',
      inferSchema=True,header=True)
```

Step 3: Exploratory Data Analysis

In this section, we drill deeper into the dataset by viewing the dataset, validating the shape of the dataset, various statistical measures, and correlations among input and output variables. We start with checking the shape of the dataset.

```
[In]:print((df.count(), len(df.columns)))
[Out]: (1232, 6)
```

The above output confirms the size of our dataset, and we can validate the datatypes of the input values to check if we need to do change/cast any columns datatypes. In this example, all columns contain Integer or double values.

```
[In]: df.printSchema()
[Out]:
```

```
root
 |-- var_1: integer (nullable = true)
 |-- var_2: integer (nullable = true)
 |-- var_3: integer (nullable = true)
 |-- var_4: double (nullable = true)
 |-- var_5: double (nullable = true)
 |-- output: double (nullable = true)
```

There is a total of six columns out of which five are input columns (`var_1` to `var_5`) and target column (output). We can now use `describe` function to go over statistical measures of the dataset.

```
[In]: df.describe().show(3,False)
[Out]:
```

```
+-------+-----------------+-----------------+-----------------+--------------------+--------------------+-----------
--------+
|summary|var_1            |var_2            |var_3            |var_4               |var_5               |output
       |
+-------+-----------------+-----------------+-----------------+--------------------+--------------------+-----------
--------+
|count  |1232             |1232             |1232             |1232                |1232                |1232
       |
|mean   |715.0819805194806|715.0819805194806|80.90422077922078|0.3263311688311693  |0.25927272727272715 |0.397341720
77922014|
|stddev |91.5342940441652 |93.07993263118064|11.458139049993724|0.015012772334166148|0.012907228928000298|0.033266898
62173776|
|min    |463              |472              |40               |0.277               |0.214               |0.301
       |
|max    |1009             |1103             |116              |0.373               |0.294               |0.491
       |
+-------+-----------------+-----------------+-----------------+--------------------+--------------------+-----------
--------+
```

This allows us to get a sense of distribution, measure of center, and spread for our dataset columns. We then take a sneak peek into the dataset using the head function and pass the number of rows that we want to view.

```
[In]: df.head(3)
[Out]:
```

```
[Row(var_1=734, var_2=688, var_3=81, var_4=0.328, var_5=0.259, output=0.418),
 Row(var_1=700, var_2=600, var_3=94, var_4=0.32, var_5=0.247, output=0.389),
 Row(var_1=712, var_2=705, var_3=93, var_4=0.311, var_5=0.247, output=0.417)]
```

We can check the correlation between input variables and output variables using the corr function:

```
[In]: from pyspark.sql.functions import corr
[In]: df.select(corr('var_1','output')).show()
[Out] :
```

```
+-------------------+
|corr(var_1, output)|
+-------------------+
| 0.9187399607627283|
+-------------------+
```

var_1 seems to be most strongly correlated with the output column.

Step 4: Feature Engineering

This is the part where we create a single vector combining all input features by using Spark's VectorAssembler. It creates only a single feature that captures the input values for that row. So, instead of five input columns, it essentially merges all input columns into a single feature vector column.

```
[In]: from pyspark.ml.linalg import Vector
[In]: from pyspark.ml.feature import VectorAssembler
```

One can select the number of columns that would be used as input features and can pass only those columns through the VectorAssembler. In our case, we will pass all the five input columns to create a single feature vector column.

```
[In]: df.columns
[Out]: ['var_1', 'var_2', 'var_3', 'var_4', 'var_5', 'output']

[In]: vec_assmebler=VectorAssembler(inputCols=['var_1',
      'var_2', 'var_3', 'var_4', 'var_5'],outputCol='features')
[In]: features_df=vec_assmebler.transform(df)

[In]: features_df.printSchema()
[Out]:
```

```
root
 |-- var_1: integer (nullable = true)
 |-- var_2: integer (nullable = true)
 |-- var_3: integer (nullable = true)
 |-- var_4: double (nullable = true)
 |-- var_5: double (nullable = true)
 |-- output: double (nullable = true)
 |-- features: vector (nullable = true)
```

As, we can see, we have an additional column ('features') that contains the single dense vector for all of the inputs.

```
[In]: features_df.select('features').show(5,False)
[Out]:
```

```
+-----------------------------+
|features                     |
+-----------------------------+
|[734.0,688.0,81.0,0.328,0.259]|
|[700.0,600.0,94.0,0.32,0.247] |
|[712.0,705.0,93.0,0.311,0.247]|
|[734.0,806.0,69.0,0.315,0.26] |
|[613.0,759.0,61.0,0.302,0.24] |
+-----------------------------+
only showing top 5 rows
```

We take the subset of the dataframe and select only the features column and the output column to build the Linear Regression model.

```
[In]: model_df=features_df.select('features','output')
```

```
[In]: model_df.show(5,False)
[Out]:
```

```
+-----------------------------+------+
|features                     |output|
+-----------------------------+------+
|[734.0,688.0,81.0,0.328,0.259]|0.418 |
|[700.0,600.0,94.0,0.32,0.247] |0.389 |
|[712.0,705.0,93.0,0.311,0.247]|0.417 |
|[734.0,806.0,69.0,0.315,0.26] |0.415 |
|[613.0,759.0,61.0,0.302,0.24] |0.378 |
+-----------------------------+------+
only showing top 5 rows
```

```
[In]: print((model_df.count(), len(model_df.columns)))
[Out]: (1232, 2)
```

Step 5: Splitting the Dataset

We have to split the dataset into a training and test dataset in order to train and evaluate the performance of the Linear Regression model built. We split it into a 70/30 ratio and train our model on 70% of the dataset. We can print the shape of train and test data to validate the size.

```
[In]: train_df,test_df=model_df.randomSplit([0.7,0.3])

[In]: print((train_df.count(), len(train_df.columns)))
[Out]: (882, 2)

[In]: print((test_df.count(), len(test_df.columns)))
[Out]: (350, 2)
```

Step 6: Build and Train Linear Regression Model

In this part, we build and train the Linear Regression model using features of the input and output columns. We can fetch the coefficients (B1, B2, B3, B4, B5) and intercept (B0) values of the model as well. We can also evaluate the performance of model on training data as well using r2. This model gives a very good accuracy (86%) on training datasets.

```
[In]: from pyspark.ml.regression import LinearRegression

[In]: lin_Reg=LinearRegression(labelCol='output')

[In]: lr_model=lin_Reg.fit(train_df)

[In]: print(lr_model.coefficients)
[Out]: [0.000345569740987,6.07805293067e-05,0.000269273376209,-
       0.713663600176,0.432967466411]
```

```
[In]: print(lr_model.intercept)
[Out]: 0.20596014754214345

[In]: training_predictions=lr_model.evaluate(train_df)

[In]: print(training_predictions.r2)
[Out]: 0.8656062610679494
```

Step 7: Evaluate Linear Regression Model on Test Data

The final part of this entire exercise is to check the performance of the model on unseen or test data. We use the evaluate function to make predictions for the test data and can use r2 to check the accuracy of the model on test data. The performance seems to be almost similar to that of training.

```
[In]: test_predictions=lr_model.evaluate(test_df)

[In]: print(test_results.r2)
[Out]: 0.8716898064262081

[In]: print(test_results.meanSquaredError)
[Out]: 0.00014705472365990883
```

Conclusion

In this chapter, we went over the process of building a Linear Regression model using PySpark and also explained the process behind finding the most optimal coefficients and intercept values.

Logistic Regression

This chapter focuses on building a Logistic Regression Model with PySpark along with understanding the ideas behind logistic regression. Logistic regression is used for classification problems. We have already seen classification details in earlier chapters. Although it is used for classification, it's still called logistic regression. This is due to the fact that under the hood, linear regression equations still operate to find the relationship between input variables and the target variables. The main distinction between linear and logistic regression is that we use some sort of nonlinear function to convert the output of the latter into the probability to restrict it between 0 and 1. For example, we can use logistic regression to predict if a user would buy the product or not. In this case, the model would return a buying probability for each user. Logistic regression is used widely in many business applications.

Probability

To understand logistic regression, we will have to go over the concept of Probability first. It is defined as the chances of occurrence of a desired event or interested outcomes upon all possible outcomes. Take, for an example, if we flip a coin, the chances of getting heads or tails are equal (50%) as shown in Figure 5-1.

Figure 5-1. *Probability of events*

If we roll a fair dice, then the probability of getting any of the numbers between (1 to 6) is equal 16.7%.

If we pick a ball from a bag that contains four green and one blue ball, the probability of picking a green ball is 80%.

Logistic regression is used to predict the probability of each target class. In case of binary classification (only two classes), it returns the probability associated with each class for every record. As mentioned, it uses linear regression behind the scenes in order to capture the relationship between input and output variables, yet we additionally use one more element (nonlinear function) to convert the output from continuous form into probability. Let's understand this with the help of an example. Let's consider that we have to use models to predict if some particular user would buy the product or not, and we are using only a single input variable that is time spent by the user on the website. The data for the same is given in Table 5-1.

Table 5-1. *Conversion Dataset*

Sr. No	Time Spent (mins)	Converted
1	1	No
2	2	No
3	5	No
4	15	Yes
5	17	Yes
6	18	Yes

Let us visualize this data in order to see the distinction between converted and non-converted users as shown in Figure 5-2.

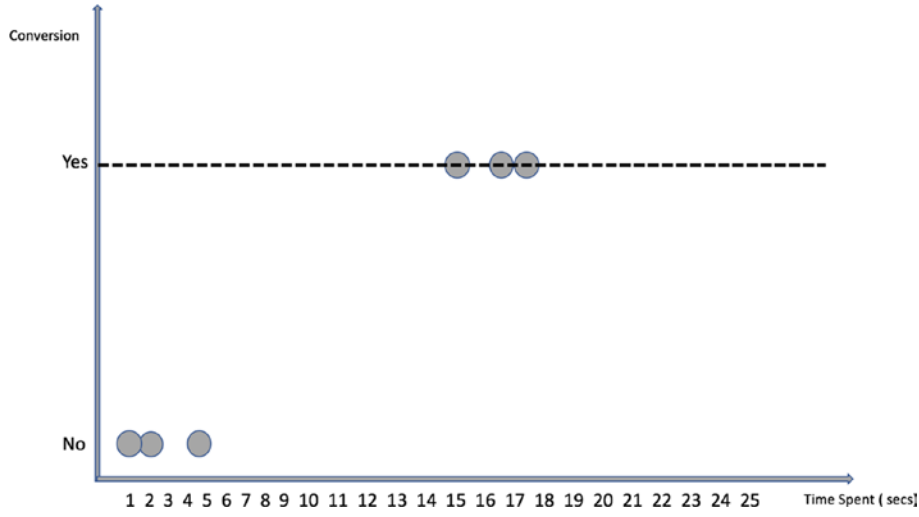


Figure 5-2. *Conversion Status vs. Time Spent*

Using Linear Regression

Let's try using linear regression instead of logistic regression to understand the reasons why logistic regression makes more sense in classification scenarios. In order to use linear regression, we will have to convert the target variable from the categorical into numeric form. So, let's reassign the values for the Converted column:

Yes = 1

No = 0

Now, our data looks something like this as given in Table 5-2.

Table 5-2. *Sample Data*

Sr. No	Time Spent (mins)	Converted
1	1	0
2	2	0
3	5	0
4	15	1
5	17	1
6	18	1

This process of converting a categorical variable to a numerical one is also critical, and we will go over this in detail in a later part of this chapter. For now, let's plot these data points to visualize and understand it better (Figure 5-3).

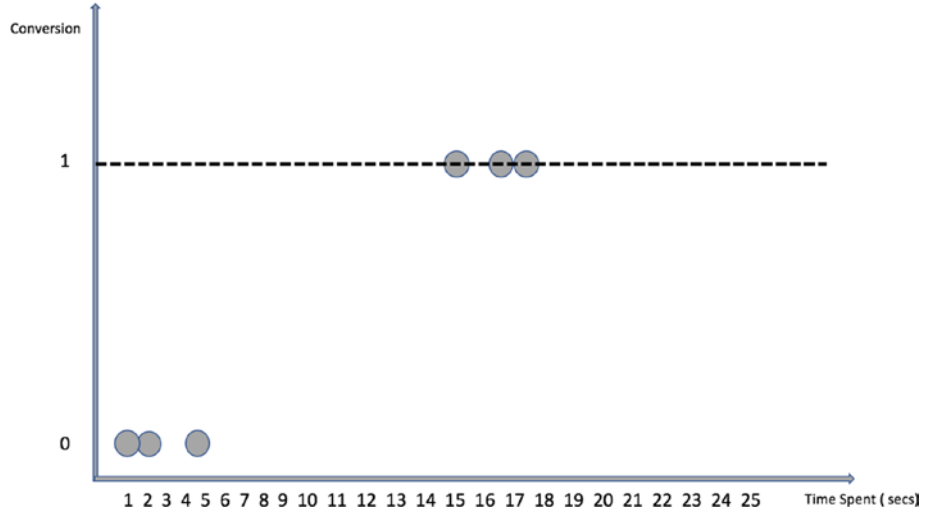


Figure 5-3. *Conversion Status (1 and 0) vs. Time spent*

As we can observe, there are only two values in our target column (1 and 0), and every point lies on either of these two values. Now, let's suppose we do linear regression on these data points and come up with a "Best fit line," which is shown in Figure 5-4.

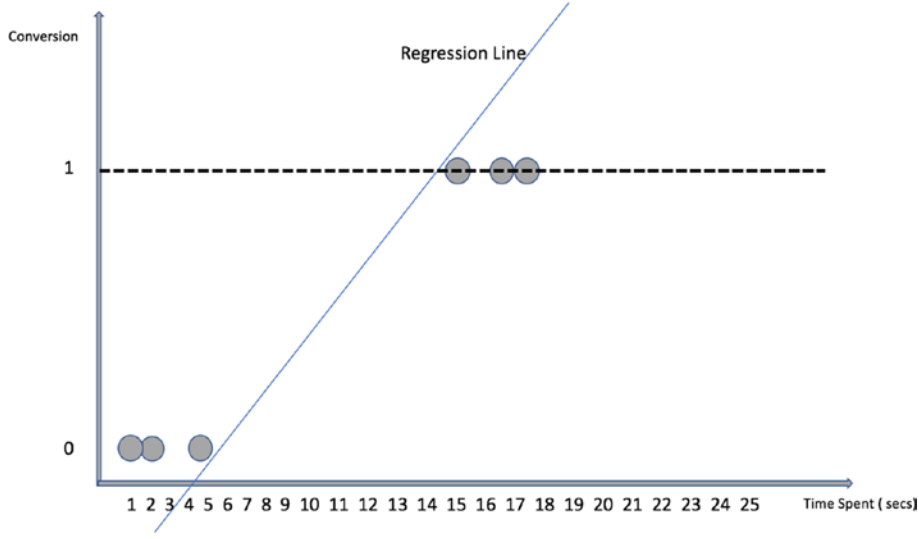


Figure 5-4. *Regression Line for users*

The regression equation for this line would be

$$y = B_0 + B_1 * x$$

$$y_{(1,0)} = B_0 + B_1 * Time\ Spent$$

All looks good so far in terms of coming up with a straight line to distinguish between 1 and 0 values. It seems like linear regression is also doing a good job of differentiating between converted and non-converted users, but there is a slight problem with this approach.

Take for an example, a new user spends 20 seconds on the website and we have to predict if this user will convert or not using the linear regression line. We use the above regression equation and try to predict the y value for 20 seconds time spent.

We can simply calculate the value of y by either calculating

$$y = B_0 + B_1 * (20)$$

or we can also simply draw a perpendicular line from the time spent axis onto the best fit line to predict the value of y. Clearly, the predicted value of y, which is 1.7, seems way above 1 as shown in Figure 5-5. This approach doesn't make any sense since we want to predict only between 0 and 1 values.

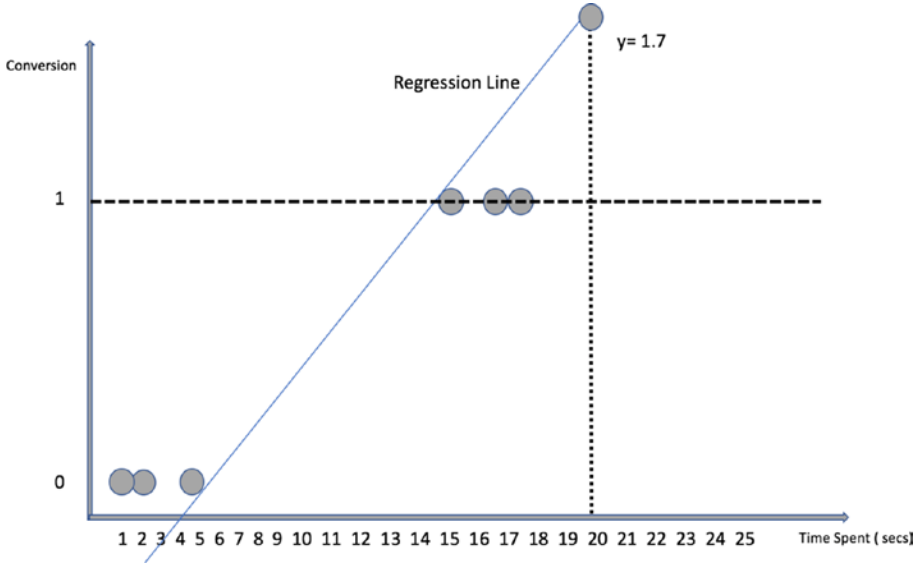


Figure 5-5. *Predictions using Regression Line*

So, if we use linear regression for classification cases, it creates a situation where the predicted output values can range from –infinity to + infinity. Hence, we need another approach that can tie these values between 0 and 1 only. The notion of values between 0 and 1 is not unfamiliar anymore as we have already seen probability. So, essentially logistic regression comes up with a decision boundary between positive and negative classes that are associated with a probability value.

Using Logit

To accomplish the objective of converting the output value in probability, we use something called Logit. Logit is a nonlinear function and does a nonlinear transformation of a linear equation to convert the output between 0 and 1. In logistic regression, that nonlinear function is the Sigmoid function, which looks like this:

$$\frac{1}{1+e^{-x}}$$

and it always produces values between 0 and 1 independent of the values of x.

So, going back to our earlier linear regression equation

$$y = B_0 + B_1 * Time\ Spent$$

we pass our output(y) through this nonlinear function(sigmoid) to change its values between 0 and 1.

$$\text{Probability} = \frac{1}{1+e^{-y}}$$

$$\text{Probability} = \frac{1}{1+e^{-(B_0+B_1*Time\ Spent)}}$$

Using the above equation, the predicted values gets limited between 0 and 1 and the output now looks as shown in Figure 5-6.

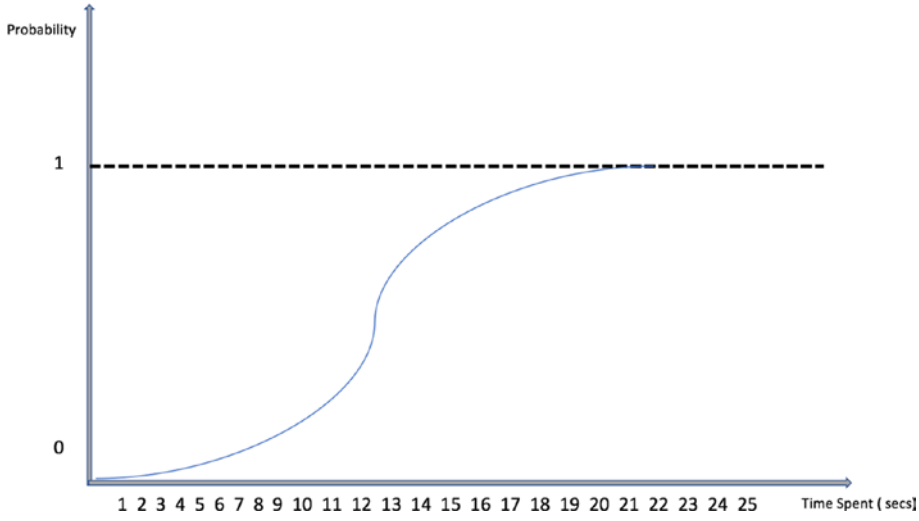


Figure 5-6. *Logistic Curve*

The advantage of using the nonlinear function is that irrespective of any value of input (time spent), the output would always be the probability of conversion. This curve is also known as a logistic curve. Logistic regression also assumes that there is a linear relationship between the input and the target variables, and hence the most optimal values of the intercept and coefficients are found out to capture this relationship.

Interpretation (Coefficients)

The coefficients of the input variables are found using a technique known as gradient descent, which looks for optimizing the loss function in such a way that the total error is minimized. We can look at the logistic regression equation and understand the interpretation of coefficients.

$$y = \frac{1}{1 + e^{-(B_0 + B_1 * x)}}$$

Let's say after calculating for the data points in our example, we get the coefficient value of time spent as 0.75.

In order to understand what this 0.75 means, we have to take the exponential value of this coefficient.

$$e^{0.75}=2.12$$

This 2.12 is known as an odd ratio, and it suggests that a per-unit increase in time spent on the website increases the odds of customer conversion by 112%.

Dummy Variables

So far, we have only dealt with continuous/numerical variables, but the presence of categorical variables in the dataset is inevitable. So, let's understand the approach to use the categorical values for modeling purposes. Since machine learning models only consume data in a numerical format, we have to adopt some technique to convert the categorical data in a numerical form. We have already seen one example above where we converted our target class (Yes/No) into numerical values (1 or 0). This is known as label encoding where we assign unique numerical values to each of the categories present in that particular column. There is another approach that works really well and is known as dummification or one hot encoding. Let's understand this with help of an example. Let's add one more column to our existing example data. Suppose we have one addition column that contains the search engine that the user used. So, our data looks something like this as shown in Table 5-3.

Table 5-3. *Categorical Dataset*

Sr. No	Time Spent (mins)	Search Engine	Converted
1	5	Google	0
2	2	Bing	0
3	10	Yahoo	1
4	15	Bing	1
5	1	Yahoo	0
6	12	Google	1

So, to consume the additional information provided in the Search Engine column, we have to convert this into a numerical format using dummification. As a result, we would get an additional number of dummy variables (columns), which would be equal to the number of distinct categories in the Search Engine column. The steps below explain the entire process of converting a categorical feature into a numerical one.

1. Find out the distinct number of categories in the categorical column. We have only three distinct categories as of now (Google, Bing, Yahoo).
2. Create new columns for each of the distinct categories and add value 1 in the category column for which it is present or else 0 as shown in Table 5-4.

Table 5-4. *One hot encoding*

Sr. No	Time Spent (mins)	Search Engine	SE_Google	SE_Bing	SE_Yahoo	Converted
1	1	Google	1	0	0	0
2	2	Bing	0	1	0	0
3	5	Yahoo	0	0	1	0
4	15	Bing	0	1	0	1
5	17	Yahoo	0	1	0	1
6	18	Google	1	0	0	1

3. Remove the original categories column. So, the dataset now contains five columns in total (excluding index) because we have three additional dummy variables as shown in Table 5-5.

Table 5-5. *Dummification*

Sr. No	Time Spent (mins)	SE_Google	SE_Bing	SE_Yahoo	Converted
1	1	1	0	0	0
2	2	0	1	0	0
3	5	0	0	1	0
4	15	0	1	0	1
5	17	0	1	0	1
6	18	1	0	0	1

The whole idea is to represent the same information in a different manner so that the Machine Learning model can learn from categorical values as well.

Model Evaluation

To measure the performance of the logistic regression model, we can use multiple metrics. The most obvious one is the accuracy parameter. Accuracy is the percentage of correct predictions made by the model. However, accuracy is not always the preferred approach. To understand the performance of the logistic model, we should use a confusion matrix. It consists of the value counts for the predictions vs. actual values.
A confusion matrix for the binary class looks like Table 5-6.

Table 5-6. *Confusion Matrix*

Actual/Prediction	Predicted Class (Yes)	Predicted Class (No)
Actual Class (Yes)	True Positives (TP)	False Negatives (FN)
Actual Class (No)	False Positives (FP)	True Negatives (TN)

Let us understand the individual values in the confusion matrix.

True Positives

These are the values that are of a positive class in actuality, and the model also correctly predicted them to be of the positive class.

- **Actual Class:** Positive (1)
- **ML Model Prediction Class**: Positive (1)

True Negatives

These are the values that are of a negative class in actuality, and the model also correctly predicted them to be of the negative class.

- **Actual Class:** Negative (0)
- **ML Model Prediction Class:** Negative (1)

False Positives

These are the values that are of the negative class in actuality, but the model incorrectly predicted them to be of the positive class.

- **Actual Class:** Negative (0)
- **ML Model Prediction Class:** Positive (1)

False Negatives

These are the values that are of the positive class in actuality, but the model incorrectly predicted them to be of the negative class.

- **Actual Class:** Positive (1)
- **ML Model Prediction Class:** Negative (1)

Accuracy

Accuracy is the sum of true positives and true negatives divided by the total number of records:

$$\frac{(TP + TN)}{Total\ number\ of\ Records}$$

But as said earlier, it is not always the preferred metric because of the target class imbalance. Most of the times, target class frequency is skewed (larger number of TN examples compared to TP examples). Take, for an example, the dataset for fraud detection contains 99 % of genuine transactions and only 1% fraudulent ones. Now, if our logistic regression model predicts all genuine transactions and no fraud cases, it still ends up with 99% accuracy. The whole point is to find out the performance in regard to the positive class; hence there are a couple of other evaluation metrics that we can use.

Recall

The Recall rate helps in evaluating the performance of the model from a positive class standpoint. It tells the percentage of actual positive cases that the model is able to predict correctly out of the total number of positive cases.

$$\frac{(TP)}{(TP + FN)}$$

It talks about the quality of the machine learning model when it comes to predicting a positive class. So out of total positive classes, how many was the model able to predict correctly? This metric is widely used as evaluation criteria for classification models.

Precision

Precision is about the number of actual positive cases out of all the positive cases predicted by the model:

$$\frac{(TP)}{(TP + FP)}$$

These can also be used as evaluation criteria.

F1 Score

$$\text{F1 Score} = 2 * \frac{(Precision * Recall)}{(Precision + Recall)}$$

Cut Off /Threshold Probability

Since we know the output of the logistic regression model is the probability score, it is very important to decide the cut off or threshold limit of the probability for prediction. By default, the probability threshold is set at 50%. It means that if the probability output of the model is below 50%, the model will predict it to be of a negative class (0), and if it is equal and above 50%, it would be assigned a positive class (1).

If the threshold limit is very low, then the model will predict a lot of positive classes and would have a high recall rate. On the contrary, if the threshold probability is very high then, the model might miss out on positive cases and the recall rate would be low, but the precision would be higher. In this case, the model will predict very few positive cases. Deciding a good threshold value is often challenging. A Receiver Operator Characteristic curve, or ROC curve, can help to decide which value of the threshold is best.

ROC Curve

The ROC is used to decide the threshold value for the model. It is the plot between recall (also known as sensitivity) and precision (specificity) as shown in Figure 5-7.

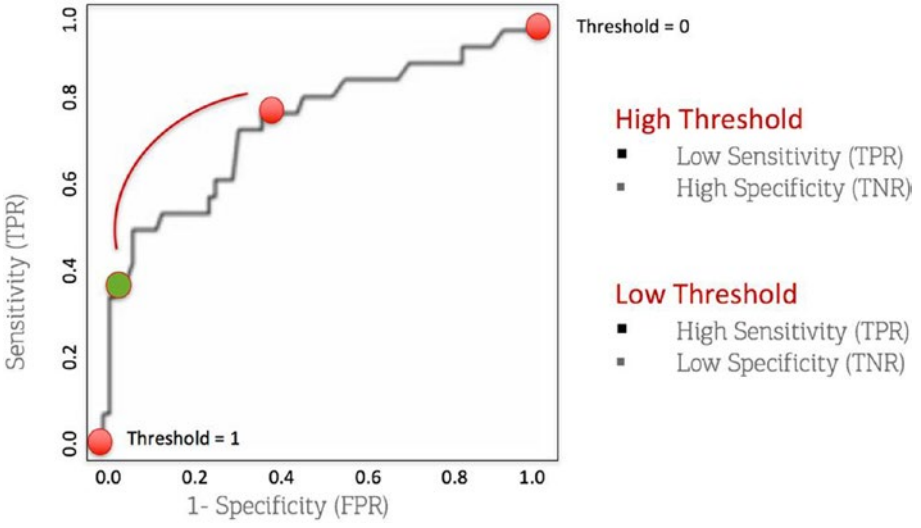


Figure 5-7. *ROC Curve*

One would like to pick a threshold that offers a balance between both recall and precision. So, now that we understand various components associated with Logistic Regression, we can go ahead and build a logistic regression model using PySpark.

Logistic Regression Code

This section of the chapter focuses on building a logistic regression model from scratch using PySpark and Jupyter Notebook.

Note A complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.3 and higher versions.

Let's build a logistic regression model using Spark's MLlib library and predict the target class label.

Data Info

The dataset that we are going to use for this example is a dummy dataset and contains a total of 20,000 rows and 6 columns. We have to use 5 input variables to predict the target class using the logistic regression model. This dataset contains information regarding online users of a retail sports merchandise website. The data captures the country of user, platform used, age, repeat visitor or first-time visitor, and the number of web pages viewed on the website. It also has the information if the customer ultimately bought the product or not (conversion status).

Step 1: Create the Spark Session Object

We start the Jupyter Notebook and import SparkSession and create a new SparkSession object to use Spark.

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('log_reg').
      getOrCreate()
```

Step 2: Read the Dataset

We then load and read the dataset within Spark using Dataframe. We have to make sure we have opened the PySpark from the same directory folder where the dataset is available or else we have to mention the directory path of the data folder.

```
[In]: df=spark.read.csv('Log_Reg_dataset.csv',inferSchema=True,
      header=True)
```

Step 3: Exploratory Data Analysis

In this section, we drill deeper into the dataset by viewing the dataset and validating the shape of the it and various statistical measures of the variables. We start with checking the shape of the dataset:

```
[In]:print((df.count(), len(df.columns)))
[Out]: (20000, 6)
```

So, the above output confirms the size of our dataset and we can then validate the datatypes of the input values to check if we need to change/cast any columns datatypes.

```
[In]: df.printSchema()
[Out]: root
 |-- Country: string (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Repeat_Visitor: integer (nullable = true)
 |-- Search_Engine: string (nullable = true)
 |-- Web_pages_viewed: integer (nullable = true)
 |-- Status: integer (nullable = true)
```

As we can see, there are two such columns (Country, Search_Engine), which are categorical in nature and hence need to be converted into numerical form. Let's have a look at the dataset using the show function in Spark.

```
[In]: df.show(5)
[Out]:
```

```
+---------+---+--------------+-------------+-----------------+------+
|  Country|Age|Repeat_Visitor|Search_Engine|Web_pages_viewed|Status|
+---------+---+--------------+-------------+-----------------+------+
|    India| 41|             1|        Yahoo|               21|     1|
|   Brazil| 28|             1|        Yahoo|                5|     0|
|   Brazil| 40|             0|       Google|                3|     0|
|Indonesia| 31|             1|         Bing|               15|     1|
| Malaysia| 32|             0|       Google|               15|     1|
+---------+---+--------------+-------------+-----------------+------+
only showing top 5 rows
```

We can now use the describe function to go over statistical measures of the dataset.

```
[In]: df.describe().show()
[Out]:
```

```
+-------+--------+------------------+--------------------+-------------+-----------------+--------------------+
|summary| Country|               Age|    Repeat_Visitor|Search_Engine| Web_pages_viewed|              Status|
+-------+--------+------------------+--------------------+-------------+-----------------+--------------------+
|  count|   20000|             20000|               20000|        20000|            20000|               20000|
|   mean|    null|          28.53955|              0.5029|         null|           9.5533|                 0.5|
| stddev|    null|7.888912950773227|0.500004090187782|         null|6.073903499824976|0.5000125004687693|
|    min|  Brazil|                17|                   0|         Bing|                1|                   0|
|    max|Malaysia|               111|                   1|        Yahoo|               29|                   1|
+-------+--------+------------------+--------------------+-------------+-----------------+--------------------+
```

We can observe that the average age of visitors is close to 28 years, and they view around 9 web pages during the website visit.

Let us explore individual columns to understand the data in deeper details. The groupBy function used along with counts returns the frequency of each of the categories in the data.

```
[In]: df.groupBy('Country').count().show()
[Out]:
```

```
+---------+-----+
|  Country|count|
+---------+-----+
| Malaysia| 1218|
|    India| 4018|
|Indonesia|12178|
|   Brazil| 2586|
+---------+-----+
```

So, the maximum number of visitors are from Indonesia, followed by India:

```
[In]: df.groupBy('Search_Engine').count().show()
[Out]:
```

Search_Engine	count
Yahoo	9859
Bing	4360
Google	5781

The Yahoo search engine users are the highest in numbers.

```
[In]: df.groupBy('Status').count().show()
[Out]:
```

Status	count
1	10000
0	10000

We have an equal number of users who are converted and non-converted.

Let's use the groupBy function along with the mean to know more about the dataset.

```
[In]: df.groupBy('Country').mean().show()
[Out]:
```

Country	avg(Age)	avg(Repeat_Visitor)	avg(Web_pages_viewed)	avg(Status)
Malaysia	27.792282430213465	0.5730706075533661	11.192118226600986	0.6568144499178982
India	27.976854156296664	0.5433051269288203	10.727227476356397	0.6212045793927327
Indonesia	28.43159796354081	0.5207751683363442	9.985711939563148	0.5422893742814913
Brazil	30.274168600154677	0.322892498066512	4.921113689095128	0.038669760247486466

We have the highest conversion rate from Malaysia, followed by India. The average number of web page visits is highest in Malaysia and lowest in Brazil.

```
[In]: df.groupBy('Search_Engine').mean().show()
[Out]:
```

```
+-------------+-----------------+-------------------+--------------------+------------------+
|Search_Engine|         avg(Age)|avg(Repeat_Visitor)|avg(Web_pages_viewed)|       avg(Status)|
+-------------+-----------------+-------------------+--------------------+------------------+
|        Yahoo|28.569226087838523| 0.5094837204584644|    9.599655137437875|0.5071508266558474|
|         Bing| 28.68394495412844| 0.4720183486238532|    9.114908256880733|0.4559633027522936|
|       Google|28.380038055699707| 0.5149628092025601|    9.804878048780488|0.5210171250648676|
+-------------+-----------------+-------------------+--------------------+------------------+
```

We have the highest conversion rate from user visitors use the Google search engine.

```
[In]: df.groupBy(Status).mean().show()
[Out]:
```

```
+------+--------+-------------------+--------------------+-----------+
|Status|avg(Age)|avg(Repeat_Visitor)|avg(Web_pages_viewed)|avg(Status)|
+------+--------+-------------------+--------------------+-----------+
|     1| 26.5435|             0.7019|             14.5617|        1.0|
|     0| 30.5356|             0.3039|              4.5449|        0.0|
+------+--------+-------------------+--------------------+-----------+
```

We can clearly see there is a strong connection between the conversion status and the number of pages viewed along with repeat visits.

Step 4: Feature Engineering

This is the part where we convert the categorical variable into numerical form and create a single vector combining all the input features by using Spark's `VectorAssembler`.

```
[In]: from pyspark.ml.feature import StringIndexer
[In]: from pyspark.ml.feature import VectorAssembler
```

Since we are dealing with two categorical columns, we will have to convert the country and search engine columns into numerical form. The machine learning model cannot understand categorical values.

The first step is to label the column using `StringIndexer` into `numerical form`. It allocates unique values to each of the categories of the column. So, in the below example, all of the three values of search engine (Yahoo, Google, Bing) are assigned values (0.0,1.0,2.0). This is visible in the column named `search_engine_num`.

```
[In]: search_engine_indexer =StringIndexer(inputCol="Search_
      Engine", outputCol="Search_Engine_Num").fit(df)

[In]:df = search_engine_indexer.transform(df)

[In]: df.show(3,False)
[Out]:
```

```
+-------+---+--------------+-------------+----------------+------+-----------------+
|Country|Age|Repeat_Visitor|Search_Engine|Web_pages_viewed|Status|Search_Engine_Num|
+-------+---+--------------+-------------+----------------+------+-----------------+
|India  |41 |1             |Yahoo        |21              |1     |0.0              |
|Brazil |28 |1             |Yahoo        |5               |0     |0.0              |
|Brazil |40 |0             |Google       |3               |0     |1.0              |
+-------+---+--------------+-------------+----------------+------+-----------------+
only showing top 3 rows
```

```
[In]: df.groupBy('Search_Engine').count().orderBy('count',
      ascending=False).show(5,False)

[Out]:
```

```
+-------------+-----+
|Search_Engine|count|
+-------------+-----+
|Yahoo        |9859 |
|Google       |5781 |
|Bing         |4360 |
+-------------+-----+
```

```
[In]: df.groupBy('Search_Engine_Num').count().orderBy('count',
      ascending=False).show(5,False)
```

[Out]:

```
+-----------------+-----+
|Search_Engine_Num|count|
+-----------------+-----+
|0.0              |9859 |
|1.0              |5781 |
|2.0              |4360 |
+-----------------+-----+
```

The next step is to represent each of these values into the form of a one hot encoded vector. However, this vector is a little different in terms of representation as it captures the values and position of the values in the vector.

```
[In]: from pyspark.ml.feature import OneHotEncoder

[In]:search_engine_encoder=OneHotEncoder(inputCol="Search_
     Engine_Num", outputCol="Search_Engine_Vector")
 [In]: df = search_engine_encoder.transform(df)

[In]: df.show(3,False)
```

[Out]:

```
+-------+---+--------------+-------------+----------------+------+-----------------+-------------------+
|Country|Age|Repeat_Visitor|Search_Engine|Web_pages_viewed|Status|Search_Engine_Num|Search_Engine_Vector|
+-------+---+--------------+-------------+----------------+------+-----------------+-------------------+
|India  |41 |1             |Yahoo        |21              |1     |0.0              |(2,[0],[1.0])      |
|Brazil |28 |1             |Yahoo        |5               |0     |0.0              |(2,[0],[1.0])      |
|Brazil |40 |0             |Google       |3               |0     |1.0              |(2,[1],[1.0])      |
+-------+---+--------------+-------------+----------------+------+-----------------+-------------------+
only showing top 3 rows
```

```
[In]: df.groupBy('Search_Engine_Vector').count().orderBy('count',
      ascending=False).show(5,False)
```

[Out]:

```
+--------------------+-----+
|Search_Engine_Vector|count|
+--------------------+-----+
|(2,[0],[1.0])       |9859 |
|(2,[1],[1.0])       |5781 |
|(2,[],[])           |4360 |
+--------------------+-----+
```

The final feature that we would be using for building Logistic Regression is `Search_Engine_Vector`. Let's understand what these column values represent.

`(2,[0],[1.0]) represents a vector of length 2 , with 1 value :`

`Size of Vector – 2`
`Value contained in vector – 1.0`
`Position of 1.0 value in vector – 0th place`

This kind of representation allows the saving of computational space and hence a faster time to compute. The length of the vector is equal to one less than the total number of elements since each value can be easily represented with just the help of two columns. For example, if we need to represent Search Engine using one hot encoding, conventionally, we can do it as represented below.

Search Engine	Google	Yahoo	Bing
Google	1	0	0
Yahoo	0	1	0
Bing	0	0	1

Another way of representing the above information in an optimized way is just using two columns instead of three as shown below.

Search Engine	Google	Yahoo
Google	1	0
Yahoo	0	1
Bing	0	0

Let's repeat the same procedure for the other categorical column (Country).

```
[In]:country_indexer = StringIndexer(inputCol="Country",
     outputCol="Country_Num").fit(df)
[In]: df = country_indexer.transform(df)
[In]: df.groupBy('Country').count().orderBy('count',ascending=
      False).show(5,False)
[Out]:
```

```
+---------+-----+
|Country  |count|
+---------+-----+
|Indonesia|12178|
|India    |4018 |
|Brazil   |2586 |
|Malaysia |1218 |
+---------+-----+
```

```
[In]: df.groupBy('Country_Num').count().orderBy('count',
      ascending=False).show(5,False)
[Out]:
```

```
+-----------+-----+
|Country_Num|count|
+-----------+-----+
|0.0        |12178|
|1.0        |4018 |
|2.0        |2586 |
|3.0        |1218 |
+-----------+-----+
```

```
[In]: country_encoder = OneHotEncoder(inputCol="Country_Num",
      outputCol="Country_Vector")
[In]: df = country_encoder.transform(df)

[In]: df.select(['Country','Country_Num','Country_Vector']).
      show(3,False)
[Out]:
```

```
+-------+-----------+--------------+
|Country|country_Num|Country_Vector|
+-------+-----------+--------------+
|India  |1.0        |(3,[1],[1.0]) |
|Brazil |2.0        |(3,[2],[1.0]) |
|Brazil |2.0        |(3,[2],[1.0]) |
+-------+-----------+--------------+
only showing top 3 rows
```

```
[In]: df.groupBy('Country_Vector').count().orderBy('count',
      ascending=False).show(5,False)
[Out]:
```

```
+--------------+-----+
|Country_Vector|count|
+--------------+-----+
|(3,[0],[1.0]) |12178|
|(3,[1],[1.0]) |4018 |
|(3,[2],[1.0]) |2586 |
|(3,[],[])     |1218 |
+--------------+-----+
```

Now that we have converted both the categorical columns into numerical forms, we need to assemble all of the input columns into a single vector that would act as the input feature for the model.

So, we select the input columns that we need to use to create the single feature vector and name the output vector as features.

```
[In]: df_assembler = VectorAssembler(inputCols=['Search_Engine_
      Vector','Country_Vector','Age', 'Repeat_Visitor',
      'Web_pages_viewed'], outputCol="features")
[In}:df = df_assembler.transform(df)
```

```
[In]: df.printSchema()
[Out]:
root
 |-- Country: string (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Repeat_Visitor: integer (nullable = true)
 |-- Search_Engine: string (nullable = true)
 |-- Web_pages_viewed: integer (nullable = true)
 |-- Status: integer (nullable = true)
 |-- Search_Engine_Num: double (nullable = false)
 |-- Search_Engine_Vector: vector (nullable = true)
 |-- Country_Num: double (nullable = false)
 |-- Country_Vector: vector (nullable = true)
 |-- features: vector (nullable = true)
```

As we can see, now we have one extra column named features, which is nothing but a combination of all the input features represented as a single dense vector.

```
[In]: df.select(['features','Status']).show(10,False)
[Out]:
+----------------------------------+------+
|features                          |Status|
+----------------------------------+------+
|[1.0,0.0,0.0,1.0,0.0,41.0,1.0,21.0]|1     |
|[1.0,0.0,0.0,0.0,1.0,28.0,1.0,5.0] |0     |
|(8,[1,4,5,7],[1.0,1.0,40.0,3.0])   |0     |
|(8,[2,5,6,7],[1.0,31.0,1.0,15.0])  |1     |
|(8,[1,5,7],[1.0,32.0,15.0])        |1     |
|(8,[1,4,5,7],[1.0,1.0,32.0,3.0])   |0     |
|(8,[1,4,5,7],[1.0,1.0,32.0,6.0])   |0     |
|(8,[1,2,5,7],[1.0,1.0,27.0,9.0])   |0     |
|(8,[0,2,5,7],[1.0,1.0,32.0,2.0])   |0     |
|(8,[2,5,6,7],[1.0,31.0,1.0,16.0])  |1     |
+----------------------------------+------+
only showing top 10 rows
```

Let us select only features column as input and the Status column as output for training the logistic regression model.

```
[In]: model_df=df.select(['features','Status'])
```

Step 5: Splitting the Dataset

We have to split the dataset into a training and test dataset in order to train and evaluate the performance of the logistic regression model. We split it in a 75/25 ratio and train our model on 75% of the dataset. Another use of splitting the data is that we can use 75% of the data to apply cross-validation in order to come up with the best Hyperparameters. Cross-validation can be of a different type where one part of the training data is kept for training and the remaining part is used for validation purposes. K-fold cross-validation is primarily used to train the model with the best Hyperparameters.

We can print the shape of train and test data to validate the size.

```
[In]: training_df,test_df=model_df.randomSplit([0.75,0.25])
[In]: print(training_df.count())
[Out]: (14907)
```

```
[In]: training_df.groupBy('Status').count().show()
[Out]:
+------+-----+
|Status|count|
+------+-----+
|     1| 7417|
|     0| 7490|
+------+-----+
```

This ensures we have a balance set of the target class (`Status`) into the training and test set.

```
[In]:print(test_df.count())
[Out]: (5093)
```

```
[In]: test_df.groupBy('Status').count().show()
[Out]:
+------+-----+
|Status|count|
+------+-----+
|     1| 2583|
|     0| 2510|
+------+-----+
```

Step 6: Build and Train Logistic Regression Model

In this part, we build and train the logistic regression model using features as the input column and status as the output column.

```
[In]: from pyspark.ml.classification import LogisticRegression
```

```
[In]: log_reg=LogisticRegression(labelCol='Status').
fit(training_df)
```

Training Results

We can access the predictions made by the model using the evaluate function in Spark that executes all the steps in an optimized way. That gives another Dataframe that contains four columns in total, including prediction and probability. The `prediction` column signifies the class label that the model has predicted for the given row and `probability` column contains two probabilities (probability for negative class at 0th index and probability for positive class at 1st index).

```
[In]: train_results=log_reg.evaluate(training_df).predictions
```

```
[In]: train_results.filter(train_results['Status']==1).
filter(train_results['prediction']==1).select(['Status',
'prediction','probability']).show(10,False)
[Out]:
+------+----------+----------------------------------------+
|Status|prediction|probability                             |
+------+----------+----------------------------------------+
|1     |1.0       |[0.2978572628475072,0.7021427371524929] |
|1     |1.0       |[0.2978572628475072,0.7021427371524929] |
|1     |1.0       |[0.16704676975730415,0.8329532302426959]|
|1     |1.0       |[0.16704676975730415,0.8329532302426959]|
|1     |1.0       |[0.16704676975730415,0.8329532302426959]|
|1     |1.0       |[0.08659913656062515,0.9134008634393749]|
|1     |1.0       |[0.08659913656062515,0.9134008634393749]|
|1     |1.0       |[0.08659913656062515,0.9134008634393749]|
|1     |1.0       |[0.08659913656062515,0.9134008634393749]|
|1     |1.0       |[0.08659913656062515,0.9134008634393749]|
+------+----------+----------------------------------------+
```

So, in the above results, probability at the 0th index is for `Status = 0` and probability as 1st index is for `Status =1`.

Step 7: Evaluate Linear Regression Model on Test Data

The final part of the entire exercise is to check the performance of the model on unseen or test data. We again make use of the evaluate function to make predictions on the test.

We assign the predictions DataFrame to results and results DataFrame now contains five columns.

```
[In]:results=log_reg.evaluate(test_df).predictions
[In]: results.printSchema()
[Out]:
root
 |-- features: vector (nullable = true)
 |-- Status: integer (nullable = true)
 |-- rawPrediction: vector (nullable = true)
 |-- probability: vector (nullable = true)
 |-- prediction: double (nullable = false)
```

We can filter the columns that we want to see using the select keyword.

```
[In]: results.select(['Status','prediction']).show(10,False)
[Out]:
+------+----------+
|Status|prediction|
+------+----------+
|0     |0.0       |
|0     |0.0       |
|0     |0.0       |
|0     |0.0       |
```

```
|1     |0.0       |
|0     |0.0       |
|1     |1.0       |
|0     |1.0       |
|1     |1.0       |
|1     |1.0       |
+------+----------+
only showing top 10 rows
```

Since this is a classification problem, we will use a confusion matrix to gauge the performance of the model.

Confusion Matrix

We will manually create the variables for true positives, true negatives, false positives, and false negatives to understand them better rather than using the direct inbuilt function.

```
[In]:tp = results[(results.Status == 1) & (results.prediction
          == 1)].count()
[In]:tn = results[(results.Status == 0) & (results.prediction
          == 0)].count()
[In]:fp = results[(results.Status == 0) & (results.prediction
          == 1)].count()
[In]:fn = results[(results.Status == 1) & (results.prediction
          == 0)].count()
```

Accuracy

As discussed already in the chapter, accuracy is the most basic metric for evaluating any classifier; however, this is not the right indicator of the performance of the model due to dependency on the target class balance.

$$\frac{(TP+TN)}{(TP+TN+FP+FN)}$$

```
[In]: accuracy=float((true_postives+true_negatives) /(results.
      count()))
[In]:print(accuracy)
[Out]: 0.9374255065554231
```

The accuracy of the model that we have built is around 94%.

Recall

Recall rate shows how much of the positive class cases we are able to predict correctly out of the total positive class observations.

$$\frac{TP}{(TP+FN)}$$

```
[In]: recall = float(true_postives)/(true_postives + false_
               negatives)
[In]:print(recall)
[Out]: 0.937524870672503
```

The recall rate of the model is around 0.94.

Precision

$$\frac{TP}{(TP+FP)}$$

Precision rate talks about the number of true positives predicted correctly out of all the predicted positives observations:

```
[In]: precision = float(true_postives) / (true_postives +
false_positives)
```

```
[In]: print(precision)
[Out]: 0.9371519490851233
```

So, the recall rate and precision rate are also in the same range, which is due to the fact that our target class was well balanced.

Conclusion

In this chapter, we went over the process of understanding the building blocks of logistic regression, converting categorical columns into numerical features, and building a logistic regression model from scratch using PySpark.

Random Forests

This chapter focuses on building Random Forests (RF) with PySpark for classification. We will learn about various aspects of them and how the predictions take place; but before knowing more about random forests, we have to learn the building block of RF that is a decision tree (DT). A decision tree is also used for Classification/Regression. but in terms of accuracy, random forests beat DT classifiers due to various reasons that we will cover later in the chapter. Let's learn more about decision trees.

Decision Tree

A decision tree falls under the supervised category of machine learning and uses frequency tables for making the predictions. One advantage of a decision tree is that it can handle both categorical and numerical variables. As the name suggests, it operates in sort of a tree structure and forms these rules based on various splits to finally make predictions. The algorithm that is used in a decision tree is ID3 developed by J. R. Quinlan.

We can break down the decision tree in different components as shown in Figure 6-1.

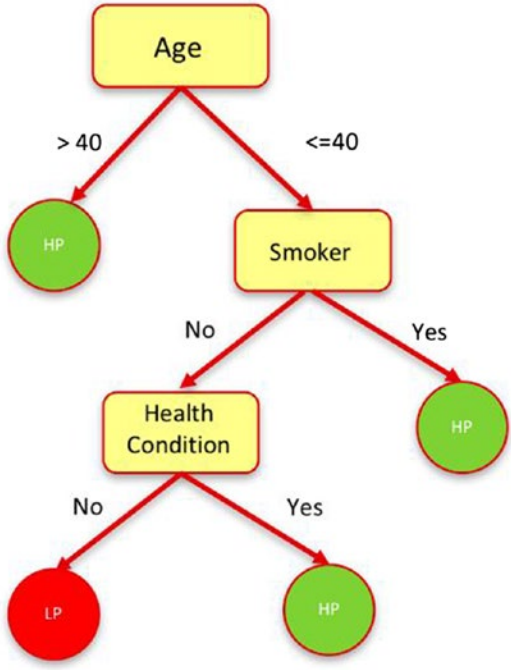


Figure 6-1. *Decision Tree*

The topmost split node from where the tree branches out is known as the root node; in the above example Age is the root node. The values represented in circles are known as leaf nodes or predictions. Let's take a sample dataset to understand how a decision tree actually works.

The data shown in Table 6-1 contains some sample data from people of different age groups and attributes. The final decision to be made based on these attributes is whether the insurance premium should be on the higher side or not. This is a typical classification case, and we will classify it using a decision tree. We have four input columns (Age Group, Smoker, Medical Condition, Salary Level).

Table 6-1. *Example Dataset*

Age Group	Smoker	Medical Condition	Salary Level	Insurance Premium
Old	Yes	Yes	High	High
Teenager	Yes	Yes	Medium	High
Young	Yes	Yes	Medium	Low
Old	No	Yes	High	High
Young	Yes	Yes	High	Low
Teenager	No	Yes	Low	High
Teenager	No	No	Low	Low
Old	No	No	Low	High
Teenager	No	Yes	Medium	High
Young	No	Yes	Low	High
Young	Yes	No	High	Low
Teenager	Yes	No	Medium	Low
Young	No	No	Medium	High
Old	Yes	No	Medium	High

Entropy

The decision tree makes subsets of this data in such a way that each of those subsets contains the same class values (homogenous); and to calculate homogeneity, we use something known as Entropy. This can also be calculated using couple of other metrics like the Gini Index and Classification error, but we will take up entropy to understand how decision trees work. The formula to calculate entropy is

$$-p \log_2 p - q \log_2 q$$

Figure 6-2 shows that entropy is equal to zero if the subset is completely pure; that means it belongs to only a single class, and it is equal to 1 if the subset is divided equally in two classes

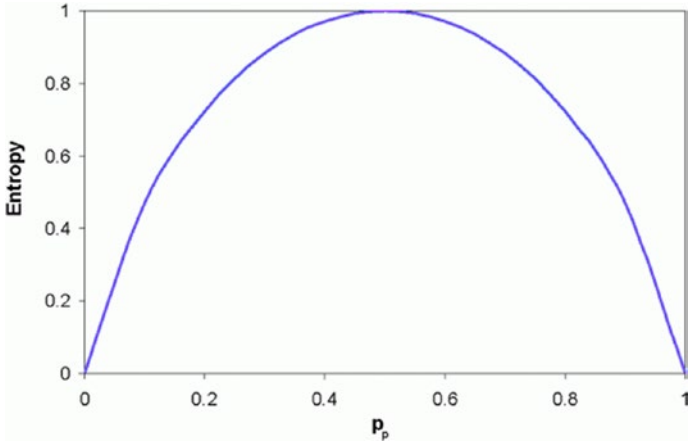


Figure 6-2. *Entropy*

If we want to calculate the entropy of our target variable (Insurance Premium), we have to first calculate the probability of each class and then use the above formula to calculate entropy.

Insurance Premium	
High (9)	Low (5)

The probability of a High category is equal to 9/14 =0.64
The probability of Low category is equal to 5/14 =0.36

$$\text{Entropy} = -p(High)log_2(p(High)) - p(Low)log_2(p(Low))$$

$$= -(0.64 * log_2(0.64)) - (0.36 * log_2(0.36))$$

$$= 0.94$$

In order to build the decision tree, we need to calculate two kinds of entropy:

1. Entropy of Target (Insurance Premium)
2. Entropy of Target with Attribute (ex. Insurance Premium – Smoker)

We have already seen the entropy of a target, so let's calculate a second entropy of the target with an input feature. Let's consider the Smoker feature, for example.

$$Entropy_{(Target,\ Feature)} = Probability_{Feature} * Entropy_{Categories}$$

Entropy Calculation		**Insurance Premium (Target)**	
Target – Insurance Premium			
Feature – Smoker		**High (9)**	**Low (5)**
Smoker (Feature)	Yes (7)	3	4
	No (7)	6	1

$$Entropy_{(Target,\ Smoker)} = P_{yes} * Entropy_{(3,4)} + P_{no} * Entropy_{(6,1)}$$

$$P_{yes} = \frac{7}{14} = 0.5$$

$$P_{no} = \frac{7}{14} = 0.5$$

$$Entropy_{(3,4)} = -\frac{3}{7} * log_2\left(\frac{3}{7}\right) - \left(\frac{4}{7}\right) * log_2\left(\frac{4}{7}\right)$$

$$= 0.99$$

$$Entropy_{(6,1)} = -\frac{6}{7} * log_2\left(\frac{6}{7}\right) - \left(\frac{1}{7}\right) * log_2\left(\frac{1}{7}\right)$$

$$= 0.59$$

$$Entropy_{(Target,\ Smoker)} = 0.55 * 0.99 + 0.5 * 0.59$$

$$=0.79$$

Similarly, we calculate the entropy for all the other attributes:

$$Entropy_{(Target,\ Age\ Group)} = 0.69$$

$$Entropy_{(Target,\ Medical\ Condition)} = 0.89$$

$$Entropy_{(Target,\ Salary\ Level)} = 0.91$$

Information Gain

The information gain (IG) is used to make the splits in decision trees. The attribute that offers the maximum information gain is used for splitting the subset. Information gain tells which is the most important feature out of all in terms of making predictions. In terms of entropy, IG is the change in entropy of the target before splitting and after splitting of a feature.

$$Information\ Gain = Entropy_{(Target)} - Entropy_{(Target,\ Feature)}$$

$$IG_{Smoker} = Entropy_{(Target)} - Entropy_{(Target,\ Smoker)}$$

$$= 0.94 - 0.79$$

$$= 0.15$$

$$IG_{Age\ Group} = Entropy_{(Target)} - Entropy_{(Target,\ Age\ Group)}$$

$$= 0.94 - 0.69$$

$$=0.25$$

$$IG_{Medical\ Condition} = Entropy_{(Target)} - Entropy_{(Target,\ Medical\ Condition)}$$

$$= 0.94 - 0.89$$

$$= 0.05$$

$$IG_{Salary\ Level} = Entropy_{(Target)} - Entropy_{(Target,\ Salary\ Level)}$$

$$= 0.94 - 0.91$$

$$= 0.03$$

As we can observe, the Age group attribute gives the maximum information gain; hence the decision tree's root node would be Age group and the first split happens on that attribute as shown in Figure 6-3.

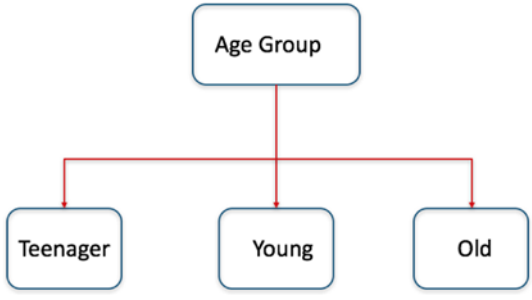


Figure 6-3. *Decision Tree Split*

The process of finding the next attribute that offers the largest information gain continues recursively and further splits are made in the decision tree. Finally, the decision tree might look something like that shown in Figure 6-4.

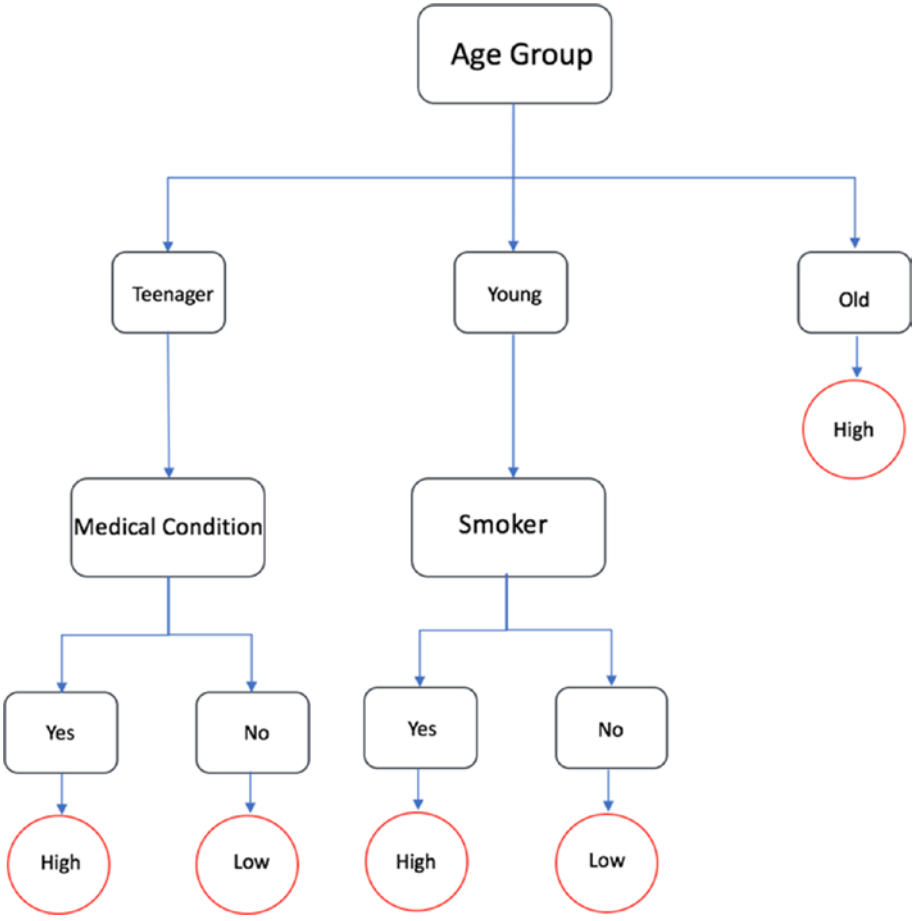


Figure 6-4. *Decision Tree Splits*

The advantage that a decision tree offers is that it can be easily transformed into set of rules by following the root node to any leaf node and hence can be easily used for classification. There are sets of Hyperparameters associated with decision trees that give more options to build trees in different manners. One of those is Max Depth, which allows us to decide the depth of a decision tree; the deeper the tree, the more slits the tree has and there are chances of overfitting.

Random Forests

Now that we know how a decision tree works, we can move on to a random forest. As the name suggests, random forests are made up of many trees: a lot of decision trees. They are quite popular and sometimes are the go-to method for supervised machine learning. Random forests can also be used for classification and regression. They combine votes from a lot of individual decision trees and then predict the class with majority votes or take the average in case of regression. This works really well because the weak learners eventually group together to make strong predictions. The importance lies in the way these decision trees are formed. The name "Random" is there for a reason in RF because the trees are formed with a random set of features and a random set of training examples. Now each decision tree being trained with a somewhat different set of data points tries to learn the relationship between input and output, which eventually gets combined with the predictions of other decision trees that used other sets of data points to get trained and hence random forests. If we take a similar example as above and create a random forest with five decision trees, it might look something like in Figure 6-5.

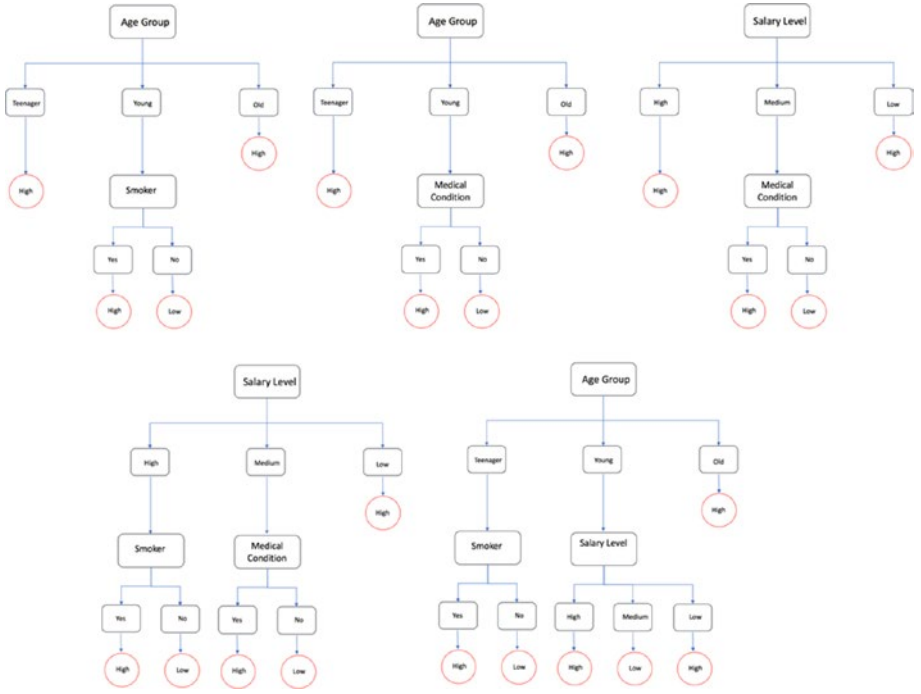


Figure 6-5. *Individual Decision Trees*

Now each of these decision trees has used a subset of data to get trained as well as a subset of features. This is also known as the "Bagging" technique – Bootstrap aggregating. Each tree sort of votes regarding the prediction, and the class with the maximum votes is the ultimate prediction by a random forest classifier as shown in Figure 6-6.

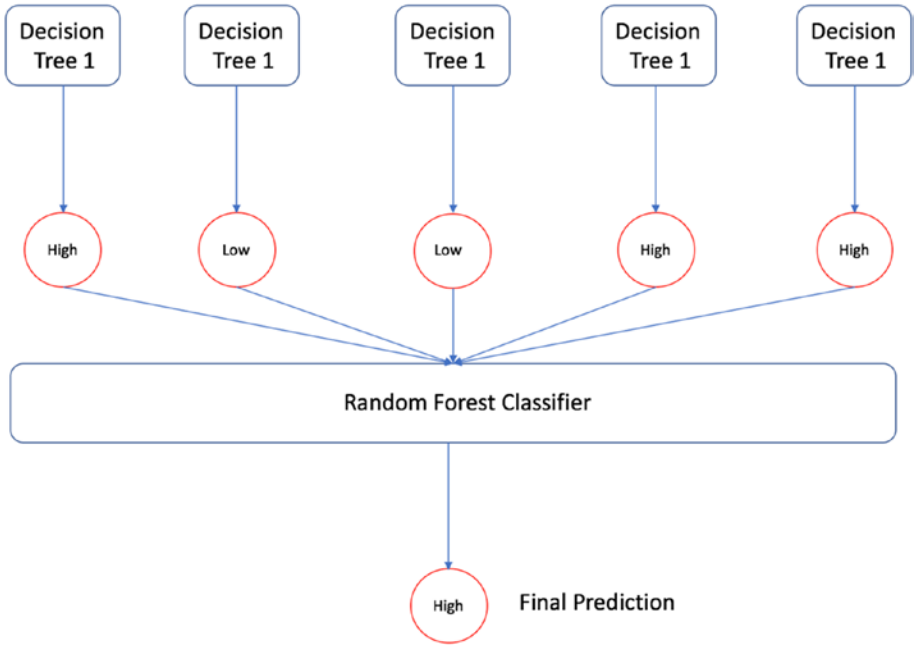


Figure 6-6. *Random Forest*

Some of the advantages that random forests offers are mentioned below:

- Feature Importance: A random forest can give the importance of each feature that has been used for training in terms of prediction power. This offers a great opportunity to select relevant features and drop the weaker ones. The total sum of all the features' importance is always equal to 1.
- Increased Accuracy: Since it collects the votes from individual decision trees, the prediction power of random forests is relatively higher compared to a single decision tree.
- Less Overfitting: The results of individual classifiers gets averaged or max voted and hence reduces the chances of overfitting.

One of the disadvantages of a random forest is that it is difficult to visualize compared to a decision tree and involves a little more on the computation side as it builds multiple individual classifiers,

Code

This section of the chapter focuses on building a Random Forest Classifier from scratch using PySpark and Jupyter Notebook.

Note A complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.0 and higher versions.

Let's build a random forest model using Spark's MLlib library and predict the target variable using the input features.

Data Info

The dataset that we are going to use for this example is an open source data set with a few thousand rows and six columns. We have to use five input variables to predict the target variable using the random forest model.

Step 1: Create the SparkSession Object

We start the Jupyter Notebook and import SparkSession and create a new SparkSession object to use Spark.

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('random_forest').
      getOrCreate()
```

Step 2: Read the Dataset

We then load and read the dataset within Spark using Dataframe. We have to make sure we have opened the PySpark from the same directory folder where the dataset is available or else we have to mention the directory path of the data folder.

```
[In]: df=spark.read.csv('affairs.csv',inferSchema=True,header=True)
```

Step 3: Exploratory Data Analysis

In this section, we drill deeper into the dataset by viewing the dataset and validating the shape of the dataset and various statistical measures of the variables. We start with checking the shape of the dataset.

```
[In]: print((df.count(), len(df.columns)))
[Out]: (6366, 6)
```

So, the above output confirms the size of our dataset and we can then validate the data types of the input values to check if we need to change/cast any columns data types.

```
[In]: df.printSchema()
```

```
[Out]: root
 |-- rate_marriage: integer (nullable = true)
 |-- age: double (nullable = true)
 |-- yrs_married: double (nullable = true)
 |-- children: double (nullable = true)
 |-- religious: integer (nullable = true)
 |-- affairs: integer (nullable = true)
```

As we can see there are no categorical columns which need to be converted into numerical form. Let's have a look at the dataset using show function in Spark:

```
[In]: df.show(5)
```

[Out]:

```
+-------------+----+-----------+--------+---------+-------+
|rate_marriage| age|yrs_married|children|religious|affairs|
+-------------+----+-----------+--------+---------+-------+
|            5|32.0|        6.0|     1.0|        3|      0|
|            4|22.0|        2.5|     0.0|        2|      0|
|            3|32.0|        9.0|     3.0|        3|      1|
|            3|27.0|       13.0|     3.0|        1|      1|
|            4|22.0|        2.5|     0.0|        1|      1|
+-------------+----+-----------+--------+---------+-------+
only showing top 5 rows
```

We can now use the describe function to go over statistical measures of the dataset.

```
[In]: df.describe().select('summary','rate_marriage','age',
      'yrs_married','children','religious').show()
```

[Out]:

```
+-------+------------------+------------------+------------------+------------------+------------------+
|summary|     rate_marriage|               age|      yrs_married|          children|         religious|
+-------+------------------+------------------+------------------+------------------+------------------+
|  count|              6366|              6366|              6366|              6366|              6366|
|   mean| 4.109644989004084|29.082862079798932| 9.00942507068803|1.3968740182218033|2.4261702796104303|
| stddev|0.9614295945655025| 6.847881883668817|7.280119972766412| 1.4334708285603441|0.8783688402641785|
|    min|                 1|              17.5|               0.5|               0.0|                 1|
|    max|                 5|              42.0|              23.0|               5.5|                 4|
+-------+------------------+------------------+------------------+------------------+------------------+
```

We can observe that the average age of people is close to 29 years, and they have been married for 9 years.

Let us explore individual columns to understand the data in deeper detail. The groupBy function used along with counts returns us the frequency of each of the categories in the data.

```
[In]: df.groupBy('affairs').count().show()
[Out]:
```

```
+-------+-----+
|affairs|count|
+-------+-----+
|      1| 2053|
|      0| 4313|
+-------+-----+
```

So, we have more than 33% of the people who are involved in some sort of extramarital affair out of a total number of people.

```
[In]: df.groupBy('rate_marriage').count().show()
[Out]:
```

```
+-------------+-----+
|rate_marriage|count|
+-------------+-----+
|            1|   99|
|            3|  993|
|            5| 2684|
|            4| 2242|
|            2|  348|
+-------------+-----+
```

The majority of the people rate their marriage very high (4 or 5), and the rest rate it on the lower side. Let's drill down a little bit further to understand if the marriage rating is related to the affair variable or not.

```
[In]: df.groupBy('rate_marriage','affairs').count().
      orderBy('rate_marriage','affairs','count',ascending=
      True).show()
[Out]:
```

```
+-------------+-------+-----+
|rate_marriage|affairs|count|
+-------------+-------+-----+
|            1|      0|   25|
|            1|      1|   74|
|            2|      0|  127|
|            2|      1|  221|
|            3|      0|  446|
|            3|      1|  547|
|            4|      0| 1518|
|            4|      1|  724|
|            5|      0| 2197|
|            5|      1|  487|
+-------------+-------+-----+
```

Clearly, the figures indicate a high percentage of people having affairs when rating their marriages low. This might prove to be a useful feature for the prediction. We will explore other variables as well in a similar manner.

```
[In]: df.groupBy('religious','affairs').count().orderBy('religious',
      'affairs','count',ascending=True).show()
[Out]:
```

```
+---------+-------+-----+
|religious|affairs|count|
+---------+-------+-----+
|        1|      0|  613|
|        1|      1|  408|
|        2|      0| 1448|
|        2|      1|  819|
|        3|      0| 1715|
|        3|      1|  707|
|        4|      0|  537|
|        4|      1|  119|
+---------+-------+-----+
```

We have a similar story from ratings on religious perspective as well as the number of people who have rated lower on religious features and a higher percentage of affair involvement.

```
[In]: df.groupBy('children','affairs').count().orderBy('children',
      'affairs','count',ascending=True).show()
[Out]:
```

children	affairs	count
0.0	0	1912
0.0	1	502
1.0	0	747
1.0	1	412
2.0	0	873
2.0	1	608
3.0	0	460
3.0	1	321
4.0	0	197
4.0	1	131
5.5	0	124
5.5	1	79

The above table does not clearly indicate any of the trends regarding the relation between the number of children and chances of being involved in an affair. Let us use the groupBy function along with the mean to know more about the dataset.

```
[In]: df.groupBy('affairs').mean().show()
[Out]:
```

affairs	avg(rate_marriage)	avg(age)	avg(yrs_married)	avg(children)	avg(religious)	avg(affairs)
1	3.647345382708234	30.537018996590355	11.152459814905017	1.7289332683877252	2.261568436434486	1.0
0	4.329700904242986	28.39067934152562	7.989334569904939	1.2388128912589844	2.5045212149316023	0.0

So, the people who have affairs rate their marriages low and a little on the higher side from an age standpoint. They have also been married for a higher number of years and are less religious.

Step 4: Feature Engineering

This is the part where we create a single vector combining all input features by using Spark's VectorAssembler.

```
[In]: from pyspark.ml.feature import VectorAssembler
```

We need to assemble all of the input columns into a single vector that would act as the input feature for the model. So,we select the input columns that we need to use to create the single feature vector and name the output vector as features.

```
[In]: df_assembler = VectorAssembler(inputCols=['ra
      te_marriage', 'age', 'yrs_married', 'children',
      'religious'], outputCol="features")
```

```
[In}:df = df_assembler.transform(df)
```

```
[In]: df.printSchema()
[Out]:
root
 |-- rate_marriage: integer (nullable = true)
 |-- age: double (nullable = true)
 |-- yrs_married: double (nullable = true)
 |-- children: double (nullable = true)
 |-- religious: integer (nullable = true)
 |-- affairs: integer (nullable = true)
 |-- features: vector (nullable = true)
```

As we can see, now we have one extra column named features, which is nothing but a combination of all the input features represented as a single dense vector.

```
[In]: df.select(['features','affairs']).show(10,False)
[Out]:
```

```
+----------------------+-------+
|features              |affairs|
+----------------------+-------+
|[5.0,32.0,6.0,1.0,3.0] |0      |
|[4.0,22.0,2.5,0.0,2.0] |0      |
|[3.0,32.0,9.0,3.0,3.0] |1      |
|[3.0,27.0,13.0,3.0,1.0]|1      |
|[4.0,22.0,2.5,0.0,1.0] |1      |
|[4.0,37.0,16.5,4.0,3.0]|1      |
|[5.0,27.0,9.0,1.0,1.0] |1      |
|[4.0,27.0,9.0,0.0,2.0] |1      |
|[5.0,37.0,23.0,5.5,2.0]|1      |
|[5.0,37.0,23.0,5.5,2.0]|1      |
+----------------------+-------+
only showing top 10 rows
```

Let us select only the features column as input and the affairs column as output for training the random forest model.

```
[In]: model_df=df.select(['features','affairs'])
```

Step 5: Splitting the Dataset

We have to split the dataset into training and test datasets in order to train and evaluate the performance of the random forest model. We split it into a 75/25 ratio and train our model on 75% of the dataset. We can print the shape of the train and test data to validate the size.

```
[In]: train_df,test_df=model_df.randomSplit([0.75,0.25])
[In]: print(train_df.count())
[Out]: 4775
```

```
[In]: train_df.groupBy('affairs').count().show()
[Out]:
```

```
+-------+-----+
|affairs|count|
+-------+-----+
|      1| 1560|
|      0| 3215|
+-------+-----+
```

This ensures we have balanced set values for the target class ('affairs') into the training and test sets.

```
[In]: test_df.groupBy('affairs').count().show()
```

```
[Out]:
+-------+-----+
|affairs|count|
+-------+-----+
|      1|  493|
|      0| 1098|
+-------+-----+
```

Step 6: Build and Train Random Forest Model

In this part, we build and train the random forest model using features such as input and Status as the output colum.

```
[In]: from pyspark.ml.classification import
      RandomForestClassifier
```

```
[In]: rf_classifier=RandomForestClassifier(labelCol='affairs',
      numTrees=50).fit(train_df)
```

There are many hyperparameters that can be set to tweak the performance of the model, but we are chosing the deafault ones here except for one that is the number of decision trees that we want to build.

Step 7: Evaluation on Test Data

Once we have trained our model on the training dataset, we can evaluate its performance on the test set.

```
[In]: rf_predictions=rf_classifier.transform(test_df)
```

```
[In]: rf_predictions.show()
```

[Out]:

features	affairs	rawPrediction	probability	prediction
[1.0,22.0,2.5,0.0...	1	[14.6041967294583...	[0.29208393458916...	1.0
[1.0,22.0,2.5,0.0...	1	[16.0932303205154...	[0.32186460641030...	1.0
[1.0,22.0,2.5,1.0...	0	[17.7239032353726...	[0.35447806470745...	1.0
[1.0,22.0,2.5,1.0...	0	[19.2192402721879...	[0.38438480544375...	1.0
[1.0,27.0,2.5,0.0...	0	[14.2152260900801...	[0.28430452180160...	1.0
[1.0,27.0,6.0,0.0...	0	[18.8525524550372...	[0.37705104910074...	1.0
[1.0,27.0,6.0,1.0...	1	[18.3786805465211...	[0.36757361093042...	1.0
[1.0,27.0,6.0,2.0...	1	[19.3152479691891...	[0.38630495938378...	1.0
[1.0,27.0,9.0,4.0...	0	[20.9219018279125...	[0.41843803655825...	1.0
[1.0,32.0,13.0,2....	1	[15.2094265653290...	[0.30418853130658...	1.0
[1.0,32.0,13.0,2....	1	[12.9702263358626...	[0.25940452671725...	1.0
[1.0,32.0,16.5,3....	1	[17.1442313409021...	[0.34288462681804...	1.0
[1.0,37.0,13.0,3....	1	[16.0227955310337...	[0.32045591062067...	1.0
[1.0,37.0,16.5,1....	1	[15.2566244058027...	[0.30513248811605...	1.0
[1.0,37.0,16.5,2....	1	[15.8784129457800...	[0.31756825891560...	1.0
[1.0,37.0,16.5,3....	1	[12.6530379071666...	[0.25306075814333...	1.0
[1.0,37.0,16.5,3....	1	[12.6530379071666...	[0.25306075814333...	1.0
[1.0,42.0,16.5,2....	1	[16.1127125117274...	[0.32225425023454...	1.0
[1.0,42.0,16.5,5....	1	[22.7022609214829...	[0.45404521842965...	1.0
[1.0,42.0,23.0,2....	1	[15.9138711184069...	[0.31827742236813...	1.0

The first column in the predictions table is that of input features of the test data. The second column is the actual label or output of the test data. The third column (rawPrediction) represents the measure of confidence for both possible outputs. The fourth column is that of conditional probability of each class label, and the final column is the prediction by the random forest classifier.We can apply a groupBy function on the prediction column to find out the number of predictions made for the positive and negative classes.

```
[In]: rf_predictions.groupBy('prediction').count().show()
[Out]:
+----------+-----+
|prediction|count|
+----------+-----+
|       0.0| 1257|
|       1.0|  334|
+----------+-----+
```

To evaluate these preditions, we will import the classificationEvaluators.

```
[In]: from pyspark.ml.evaluation import
      MulticlassClassificationEvaluator
[In]: from pyspark.ml.evaluation import
      BinaryClassificationEvaluator
```

Accuracy

```
[In]: rf_accuracy=MulticlassClassificationEvaluator(labelCol='a
      ffairs',metricName='accuracy').evaluate(rf_predictions)
[In]: print('The accuracy of RF on test data is {0:.0%}'.
      format(rf_accuracy))
[Out]: The accuracy of RF on test data is 73%
```

Precision

```
[In]: rf_precision=MulticlassClassificationEvaluator(labelCol=
      'affairs',metricName='weightedPrecision').evaluate(rf_
      predictions)
[In]: print('The precision rate on test data is {0:.0%}'.
      format(rf_precision))

[Out]: The precision rate on test data is 71%
```

AUC

```
[In]: rf_auc=BinaryClassificationEvaluator(labelCol='affairs').
      evaluate(rf_predictions)
[In]: print( rf_auc)
[Out]: 0.738
```

As mentioned in the earlier part, RF gives the importance of each feature in terms of predictive power, and it is very useful to figure out the critical variables that contribute the most to predictions.

```
[In]: rf_classifier.featureImportances
[Out]: (5,[0,1,2,3,4],[0.563965247822,0.0367408623003,
       0.243756511958,0.0657893200779,0.0897480578415])
```

We used five features and the importance can be found out using the feature importance function. To know which input feature is mapped to which index values, we can use metadata information.

```
[In]: df.schema["features"].metadata["ml_attr"]["attrs"]
[Out]:
  {'idx': 0, 'name': 'rate_marriage'},
  {'idx': 1, 'name': 'age'},
  {'idx': 2, 'name': 'yrs_married'},
  {'idx': 3, 'name': 'children'},
  {'idx': 4, 'name': 'religious'}}
```

So, rate_marriage is the most important feature from a prediction standpoint followed by yrs_married. The least significant variable seems to be Age.

Step 8: Saving the Model

Sometimes, after training the model, we just need to call the model for preditions, and hence it makes a lot of sense to persist the model object and reuse it for predictions. There are two parts to this.

1. Save the ML model
2. Load the ML model

```
[In]: from pyspark.ml.classification import
      RandomForestClassificationModel
[In]: rf_classifier.save("/home/jovyan/work/RF_model")
      This way we saved the model as object locally.The next
      step is to load the model again for predictions
[In]: rf=RandomForestClassificationModel.load("/home/jovyan/
      work/RF_model")
 [In]: new_preditions=rf.transform(new_df)
```

A new predictions table would contain the column with the model predictions

Conclusion

In this chapter, we went over the process of understanding the building blocks of Random Forests and creating an ML model in PySpark for classification along with evaluation metrics such as accuracy, precison, and auc. We also covered how to save the ML model object locally and reuse it for predictions.

Recommender Systems

A common trend that can be observed in brick and mortar stores, is that we have salespeople guiding and recommending us relevant products while shopping on the other hand, with online retail platforms, there are zillions of different products available, and we have to navigate ourselves to find the right product. The situation is that users have too many options and choices available, yet they don't like to invest a lot of time going through the entire catalogue of items. Hence, the role of Recommender Systems (RS) becomes critical for recommending relevant items and driving customer conversion.

Traditional physical stores use planograms to arrange the items in such a way that can increase the visibility of high-selling items and increase revenue whereas online retail stores need to keep it dynamic based on preferences of each individual customer rather than keeping it the same for everyone.

Recommender systems are mainly used for auto-suggesting the right content or product to the right users in a personalized manner to enhance the overall experience. Recommender systems are really powerful in terms of using huge amounts of data and learning to understand the preferences of specific users. Recommendations help users to easily navigate through millions of products or tons of content (articles/videos/movies) and show them the right item/information that they might like or buy. So, in simple

terms, RS help discover information on behalf of the users. Now, it depends on the users to decide if RS did a good job at recommendations or not, and they can choose to either select the product/content or discard and move on. Each of the decisions of users (Positive or Negative) helps to retrain the RS on the latest data to be able to give even better recommendations. In this chapter, we will go over how RS work and the different types of techniques used under the hood for making these recommendations. We will also build a recommender system using PySpark.

Recommendations

Recommender systems can be used for multiple purposes in the sense of recommending various things to users. For example, some of them might fall in the categories below:

1. Retail Products
2. Jobs
3. Connections/Friends
4. Movies/Music/Videos/Books/Articles
5. Ads

The "What to Recommend" part totally depends on the context in which RS are used and can help the business to increase revenues by providing the most likely items that users can buy or increasing the engagement by showcasing relevant content at the right time. RS take care of the critical aspect that the product or content that is being recommended should either be something which users might like but would not have discovered on their own. Along with that, RS also need an element of varied recommendations to keep it interesting enough. A few examples of heavy usage of RS by businesses today such as Amazon products, Facebook's friend suggestions, LinkedIn's "People you may know," Netflix's movie, YouTube's videos, Spotify's music, and Coursera's courses.

The impact of these recommendations is proving to be immense from a business standpoint, and hence more time is being spent in making these RS more efficient and relevant. Some of the immediate benefits that RS offer in retail settings are:

1. Increased Revenue
2. Positive Reviews and Ratings by Users
3. Increased Engagement

For the other verticals such as ads recommendations and other content recommendation, RS help immensely to help them find the right thing for users and hence increases adoption and subscriptions. Without RS, recommending online content to millions of users in a personalized manner or offering generic content to each user can be incredibly off target and lead to negative impacts on users.

Now that we know the usage and features of RS, we can take a look at different types of RS. There are mainly five types of RS that can be built:

1. Popularity Based RS
2. Content Based RS
3. Collaborative Filtering based RS
4. Hybrid RS
5. Association Rule Mining based RS

We will briefly go over each one of these except for the last item, that is, Association Rule Mining based RS as it's out of the scope of this book.

Popularity Based RS

This is the most basic and simplest RS that can be used to recommend products or content to the users. It recommends items/content based on bought/viewed/liked/downloaded by most of the users. While it is

easy and simple to implement, it doesn't produce relevant results as the recommendations stay the same for every user, but it sometimes outperforms some of the more sophisticated RS. The way this RS is implemented is by simply ranking the items on various parameters and recommending the top-ranked items in the list. As already mentioned, items or content can be ranked by the following:

1. No. of times downloaded
2. No. of times bought
3. No. of times viewed
4. Highest rated
5. No. of times shared
6. No. of times liked

This kind of RS directly recommends the best-selling or most watched/bought items to the customers and hence increases the chances of customer conversion. The limitation of this RS is that it is not hyper-personalized.

Content Based RS

This type of RS recommends similar items to the users that the user has liked in the past. So, the whole idea is to calculate a similarity score between any two items and recommended to the user based upon the profile of the user's interests. We start with creating item profiles for each of the items. Now these item profiles can be created in multiple ways, but the most common approach is to include information regarding the details or attributes of the item. For an example, the item profile of a Movie can have values on various attributes such as Horror, Art, Comedy, Action, Drama, and Commercial as shown below.

Movie ID	Horror	Art	Comedy	Action	Drama	Commercial
2310	0.01	0.3	0.8	0.0	0.5	0.9

Above is the example of an item profile, and each of the items would have a similar vector representing its attributes. Now, let's assume the user has watched 10 such movies and really liked them. So, for that particular user, we end up with the item matrix shown in Table 7-1.

Table 7-1. *Movie Data*

Movie ID	Horror	Art	Comedy	Action	Drama	Commercial
2310	0.01	0.3	0.8	0.0	0.5	0.9
2631	0.0	0.45	0.8	0.0	0.5	0.65
2444	0.2	0.0	0.8	0.0	0.5	0.7
2974	0.6	0.3	0.0	0.6	0.5	0.3
2151	0.9	0.2	0.0	0.7	0.5	0.9
2876	0.0	0.3	0.8	0.0	0.5	0.9
2345	0.0	0.3	0.8	0.0	0.5	0.9
2309	0.7	0.0	0.0	0.8	0.4	0.5
2366	0.1	0.15	0.8	0.0	0.5	0.6
2388	0.0	0.3	0.85	0.0	0.8	0.9

User Profile

The other component in content based RC is the User Profile that is created using item profiles that the user has liked or rated. Assuming that the user has liked the movie in Table 7-1, the user profile might look like a single vector, which is simply the mean of item vectors. The user profile might look something like that below.

User ID	Horror	Art	Comedy	Action	Drama	Commercial
1A92	0.251	0.23	0.565	0.21	0.52	0.725

This approach to create the user profile is one of the most baseline ones, and there are other sophisticated ways to create more enriched user profiles such as normalized values, weighted values, etc. The next step is to recommend the items (movies) that this user might like based on the earlier preferences. So, the similarity score between the user profile and item profile is calculated and ranked accordingly. The more the similarity score, the higher the chances of liking the movie by the user. There are a couple of ways by which the similarity score can be calculated.

Euclidean Distance

The user profile and item profile both are high-dimensional vectors and hence to calculate the similarity between the two, we need to calculate the distance between both vectors. The Euclidean distance can be easily calculated for an n-dimensional vector using the formula below:

$$d(x,y)=\sqrt{(x_1-y_n)^2+\ldots+(x_n-y_n)^2}$$

The higher the distance value, the less similar are the two vectors. Therefore, the distance between the user profile and all other items are calculated and ranked in decreasing order. The top few items are recommended to the user in this manner.

Cosine Similarity

Another way to calculate a similarity score between the user and item profile is cosine similarity. Instead of distance, it measures the angle between two vectors (user profile vector and item profile vector). The

smaller the angle between both vectors, the more similar they are to each other. The cosine similarity can be found out using the formula below:

$$sim(x,y)=cos(\theta)= x*y / |x|*|y|$$

Let's go over some of the pros and cons of Content based RS.

Advantages:

1. Content based RC works independently of other users' data and hence can be applied to an individual's historical data.
2. The rationale behind RC can be easily understood as the recommendations are based on the similarity score between the User Profile and Item Profile.
3. New and unknown items can also be recommended to users just based on historical interests and preferences of users.

Disadvantages:

1. Item profile can be biased and might not reflect exact attribute values and might lead to incorrect recommendations.
2. Recommendations entirely depend on the history of the user and can only recommend items that are like the historically watched/liked items and do not take into consideration the new interests or liking of the visitor.

Collaborative Filtering Based RS

CF based RS doesn't require the item attributes or description for recommendations; instead it works on user item interactions. These interactions can be measured in various ways such as ratings, item bought,

time spent, shared on another platform, etc. Before diving deep in CF, let's take a step back and reflect on how we make certain decisions on a day-to-day basis – decisions such as the following:

1. Which movie to watch
2. Which book to read
3. Which restaurant to go to
4. Which place to travel to

We ask our friends, right! We ask for recommendations from people who are similar to us in some ways and have same tastes and likings as ours. Our interests match in some areas and so we trust their recommendations. These people can be our family members, friends, colleagues, relatives, or community members. In real life, it's easy to know who are the people falling in this circle, but when it comes to online recommendations, the key task in collaborative filtering is to find the users who are most similar to you. Each user can be represented by a vector that contains the feedback value of a user item interaction. Let's understand the user item matrix first to understand the CF approach.

User Item Matrix

The user item matrix is exactly what the name suggests. In the rows, we have all the unique users; and along the columns, we have all the unique items. The values are filled with feedback or interaction scores to highlight the liking or disliking of the user for that product. A simple User Item matrix might look something like shown in Table 7-2.

Table 7-2. *User Item Matrix*

User ID	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
14SD	1	4			5	
26BB		3	3			1
24DG	1	4	1		5	2
59YU		2			5	
21HT	3	2	1	2	5	
68BC		1				5
26DF	1	4		3	3	
25TR	1	4			5	
33XF	5	5	5	1	5	5
73QS	1		3			1

As you can observe, the user item matrix is generally very sparse as there are millions of items, and each user doesn't interact with every item; so the matrix contains a lot of null values. The values in the matrix are generally feedback values deduced based upon the interaction of the user with that particular item. There are two types of feedback that can be considered in the UI matrix.

Explicit Feedback

This sort of feedback is generally when the user gives ratings to the item after the interaction and has been experiencing the item features. Ratings can be of multiple types.

1. Rating on 1–5 scale
2. Simple rating item on recommending to others (Yes or No or never)
3. Liked the Item (Yes or No)

The Explicit feedback data contains very limited amounts of data points as a very small percentage of users take out the time to give ratings even after buying or using the item. A perfect example can be of a movie, as very few users give the ratings even after watching it. Hence, building RS solely on explicit feedback data can put us in a tricky situation, although the data itself is less noisy but sometimes not enough to build RS.

Implicit Feedback

This kind of feedback is not direct and mostly inferred from the activities of the user on the online platform and is based on interactions with items. For example, if user has bought the item, added it to the cart, viewed, and spent a great deal of time on looking at the information about the item, this indicates that the user has a higher amount of interest in the item. Implicit feedback values are easy to collect, and plenty of data points are available for each user as they navigate their way through the online platform. The challenges with implicit feedback are that it contains a lot of noisy data and therefore doesn't add too much value in the recommendations.

Now that we understand the UI matrix and types of values that go into that matrix, we can see the different types of collaborative filtering (CF). There are mainly two kinds of CF:

1. Nearest Neighbors based CF
2. Latent Factor based CF

Nearest Neighbors Based CF

This CF works by finding out the k-nearest neighbors of users by finding the most similar users who also like or dislike the same items as the active user (for user we are trying to recommend). There are two steps involved in the nearest neighbor's collaborative filtering. The first step is to find k-nearest neighbors, and the second step is to predict the rating or likelihood of the active user liking a particular item. The k-nearest

neighbors can be found out using some of the earlier techniques we have discussed in the chapter. Metrics such as cosine similarity or Euclidean distance can help us to find most similar users to active users out of the total number of users based on the common items that both groups have liked or disliked. One of the other metrics that can also be used is Jaccard similarity. Let's look at an example to understand this metric – going back to the earlier user item matrix and taking just five users' data as shown in Table 7-3.

Table 7-3. *User Item Matrix*

User ID	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
14SD	1	4			5	
26BB		3	3			1
24DG	1	4	1		5	2
59YU		2			5	
26DF	1	4		3	3	

Let's say we have in total five users and we want to find the two nearest neighbors to the active user (14SD). The Jaccard similarity can be found out using

$$sim(x,y)=|Rx \cap Ry|/ | Rx \cup Ry|$$

So, this is the number of items that any two users have rated in common divided by the total number of items that both users have rated:

sim (user1, user2) = 1 / 5 = 0.2 since they have rated only Item 2 in common).

The similarity score for the rest of the four users with active users would then look something like that shown in Table 7-4.

Table 7-4. *User Similarity Score*

User ID	Similarity Score
14SD	1
26BB	0.2
24DG	0.6
59YU	0.677
26DF	0.75

So, according to Jaccard similarity the top two nearest neighbors are the fourth and fifth users. There is a major issue with this approach, though, because the Jaccard similarity doesn't consider the feedback value while calculating the similarity score and only considers the common items rated. So, there could be a possibility that users might have rated many items in common, but one might have rated them high and the other might have rated them low. The Jaccard similarity score still might end up with a high score for both users, which is counterintuitive. In the above example, it is clearly evident that the active user is most similar to the third user (24DG) as they have the exact same ratings for three common items whereas the third user doesn't even appear in the top two nearest neighbors. Hence, we can opt for other metrics to calculate the k-nearest neighbors.

Missing Values

The user item matrix would contain lot of missing values for the simple reason that there are lot of items and not every user interacts with each item. There are a couple of ways to deal with missing values in the UI matrix.

1. Replace the missing value with 0s.
2. Replace the missing values with average ratings of the user.

The more similar the ratings on common items, the nearer the neighbor is to the active user. There are, again, two categories of Nearest Neighbors based CF

1. User based CF
2. Item based CF

The only difference between both RS is that in user based we find k-nearest users, and in item based CF we find k-nearest items to be recommended to users. We will see how recommendations work in user based RS.

As the name suggests, in user based CF, the whole idea is to find the most similar user to the active user and recommend the items that the similar user has bought/rated highly to the active user, which he hasn't seen/bought/tried yet. The assumption that this kind of RS makes is that if two or more users have the same opinion about a bunch of items, then they are likely to have the same opinion about other items as well. Let's look at an example to understand the user based collaborative filtering: there are three users, out of which we want to recommend a new item to the active user. The rest of the two users are the top two nearest neighbors in terms of likes and dislikes of items with the active user as shown in Figure 7-1.

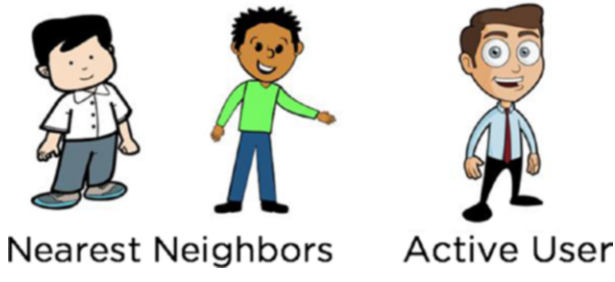


Figure 7-1. *Active User and Nearest Neighbors*

All three users have rated a particular camera brand very highly, and the first two users are the most similar users to the active user based on a similarity score as shown in Figure 7-2.

Figure 7-2. *All users like a item*

Now, the first two users have also rated another item (Xbox 360) very highly, which the third user is yet to interact with and has also not seen as shown in Figure 7-3. Using this information, we try to predict the rating that the active user would give to the new item (Xbox 360), which again is the weighted average of ratings of the nearest neighbors for that particular item (XBOX 360).

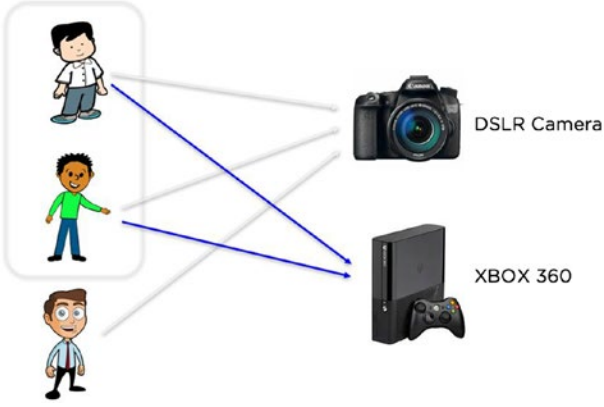


Figure 7-3. *Nearest Neighbors also like the other item*

The user based CF then recommends the other item (XBOX 360) to the active user since he is most likely to rate this item higher as the nearest neighbors have also rated this item highly as shown in Figure 7-4.

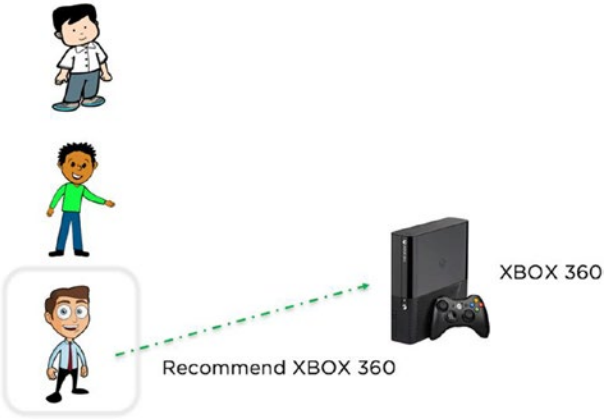


Figure 7-4. *Active User Recommendation*

Latent Factor Based CF

This kind of collaborative filtering also uses the user item matrix but instead of finding the nearest neighbors and predicting ratings, it tries to decompose the UI matrix into two latent factor matrices. The latent factors are derived values from original values. They are intrinsically related to the observed variables. These new matrices are much lower in terms of rank and contain latent factors. This is also known as matrix factorization. Let's take an example to understand the matrix factorization process. We can decompose an mxn size matrix 'A' of rank r into two smaller rank matrices X, Y such that the dot product of X and Y results in the original A matrix. If we have a matrix A shown in Table 7-5,

Table 7-5. *Latent Factor Calculation*

1	2	3	5
2	4	8	12
3	6	7	13

then we can write all the column values as linear combinations of the first and third columns (A1 and A3).

$$A1 = 1 * A1 + 0 * A3$$

$$A2 = 2 * A1 + 0 * A3$$

$$A3 = 0 * A1 + 1 * A3$$

$$A4 = 2 * A1 + 1 * A3$$

Now we can create the two small rank matrices in such a way that the product between those two would return the original matrix A.

X =

1	**3**
2	8
3	7

Y =

1	**2**	**0**	**2**
0	0	1	1

X contains columns values of A1 and A3 and Y contains the coefficients of linear combinations.

The dot product between X and Y results back into matrix 'A' (original matrix)

Considering the same user item matrix as shown in Table 7-2, we factorize or decompose it into two smaller rank matrices.

1. Users latent factor matrix
2. Items latent factor matrix

User ID	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
14SD	1	4			5	
26BB		3	3			1
24DG	1	4	1		5	2
59YU		2			5	
21HT	3	2	1	2	5	
68BC		1				5
26DF	1	4		3	3	
25TR	1	4			5	
33XF	5	5	5	1	5	5
73QS	1		3			1

	Item 1	Item 2	Item 3	Item 4	Item 5	Item n
ITF1	0.3	0.23			0.9	
ITF2		0.1	0.14			0.02
ITF3	0.25	0.8	0.09		0.9	0.33

Item latent Factor Matrix

User ID	USF1	USF2	USF3
14SD	0.02	0.97	
26BB		0.24	0.65
24DG	0.03	0.86	0.07
59YU		0.45	
21HT	0.65	0.38	0.05
68BC		0.03	
26DF	0.02	0.78	
25TR	0.01	0.84	
33XF	0.95	0.98	0.93
73QS	0.03		0.48

User latent Factor Matrix

0.23
0.1
0.8

The user latent factor matrix contains all the users mapped to these latent factors, and similarly the item latent factor matrix contains all items in columns mapped to each of the latent factors. The process of finding these latent factors is done using machine learning optimization

techniques such as Alternating Least squares. The user item matrix is decomposed into latent factor matrices in such a way that the user rating for any item is the product between a user's latent factor value and the item latent factor value. The main objective is to minimize the total sum of squared errors over the entire user item matrix ratings and predicted item ratings. For example, the predicted rating of the second user (26BB) for Item 2 would be

Rating (user2, item2) =

	0.24	0.65

There would be some amount of error on each of the predicted ratings, and hence the cost function becomes the overall sum of squared errors between predicted ratings and actual ratings. Training the recommendation model includes learning these latent factors in such a way that it minimizes the SSE for overall ratings. We can use the ALS method to find the lowest SSE. The way ALS works is that it fixes first the user latent factor values and tries to vary the item latent factor values such that the overall SSE reduces. In the next step, the item latent factor values are kept fixed, and user latent factor values are updated to further reduce the SSE. This keeps alternating between the user matrix and item matrix until there can be no more reduction in SSE.

Advantages:

1. Content information of the item is not required, and recommendations can be made based on valuable user item interactions.
2. Personalizing experience based on other users.

Limitations:

1. Cold Start Problem: If the user has no historical data of item interactions. then RC cannot predict the k-nearest neighbors for the new user and cannot make recommendations.
2. Missing values: Since the items are huge in number and very few users interact with all the items, some items are never rated by users and can't be recommended.
3. Cannot recommend new or unrated items: If the item is new and yet to be seen by the user, it can't be recommended to existing users until other users interact with it.
4. Poor Accuracy: It doesn't perform that well as many components keep changing such as interests of users, limited shelf life of items, and very few ratings of items.

Hybrid Recommender Systems

As the name suggests, the hybrid RS include inputs from multiple recommender systems, making it more powerful and relevant in terms of meaningful recommendations to the users. As we have seen, there are a few limitations in using individual RS, but in combination they overcome few of those and hence are able to recommend items or information that users find more useful and personalized. The hybrid RS can be built in specific ways to suit the requirement of the business. One of the approaches is to build individual RS and combine the recommendations from multiple RS output before recommending them to the user as shown in Figure 7-5.

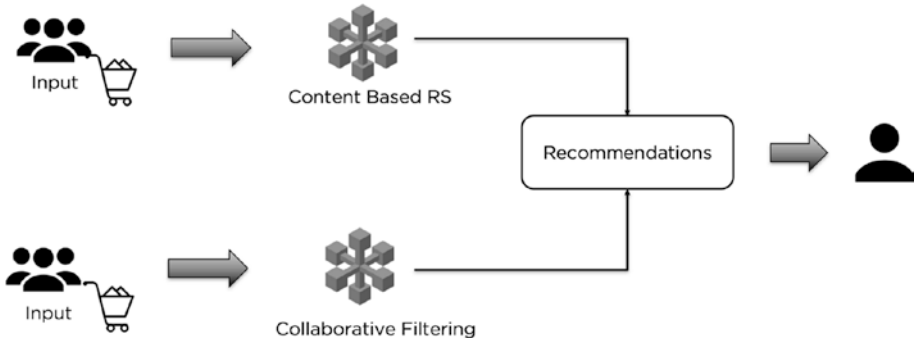


Figure 7-5. *Combining Recommendations*

The other approach is by leveraging content based recommender strengths and using them as input for collaborative filtering based recommendations to provide better recommendations to the users. This approach can also be reversed, and collaborative filtering can be used as input for content based recommendations as shown in Figure 7-6.

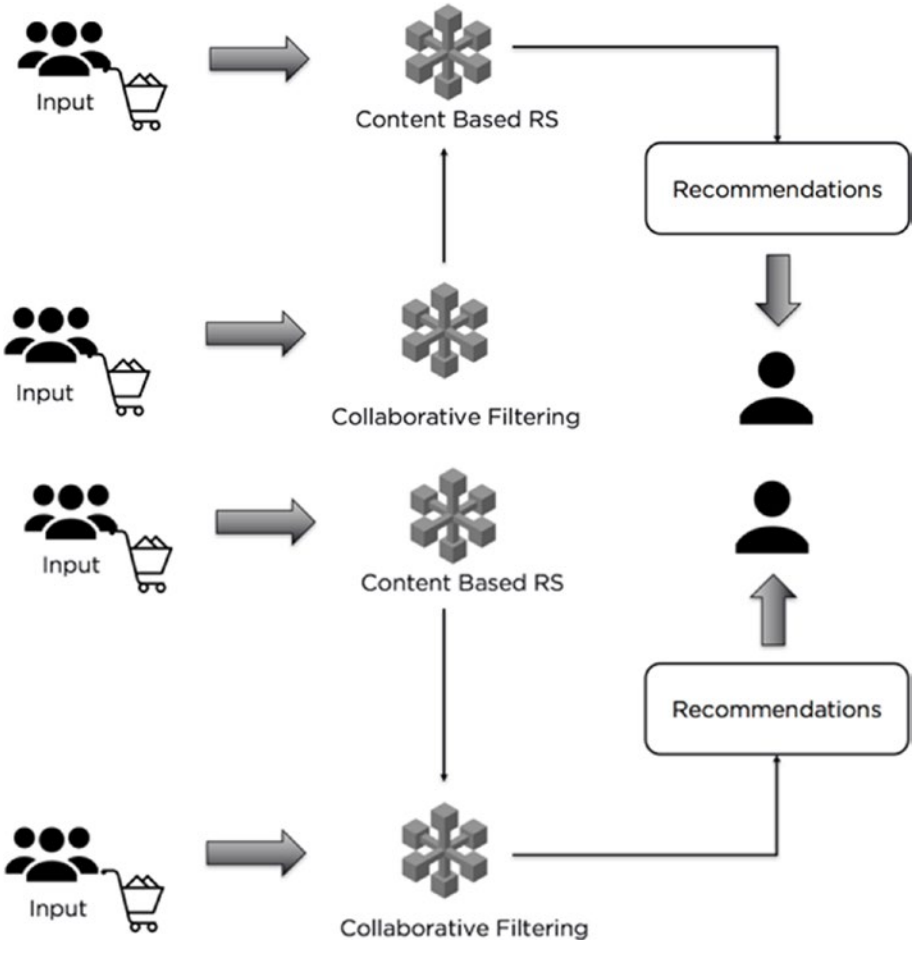


Figure 7-6. *Hybrid Recommendations*

Hybrid recommendations also include using other types of recommendations such as demographic based and knowledge based to enhance the performance of its recommendations. Hybrid RS have become integral parts of various businesses to help their users consume the right content, therefore deriving a lot of value.

Code

This section of the chapter focuses on building an RS from scratch using the ALS method in PySpark and Jupyter Notebook.

Note The complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.0 and higher versions.

Let's build a recommender model using Spark's MLlib library and predict the rating of an item for any given user.

Data Info

The dataset that we are going to use for this chapter is a subset from a famous open sourced movie lens dataset and contains a total of 0.1 million records with three columns (User_Id,title,rating). We will train our recommender model using 75% of the data and test it on the rest of the 25% user ratings.

Step 1: Create the SparkSession Object

We start the Jupyter Notebook and import SparkSession and create a new SparkSession object to use Spark:

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('lin_reg').
      getOrCreate()
```

Step 2: Read the Dataset

We then load and read the dataset within Spark using a dataframe. We have to make sure we have opened the PySpark from the same directory folder where the dataset is available or else we have to mention the directory path of the data folder.

```
[In]:
df=spark.read.csv('movie_ratings_df.csv',inferSchema=True,
header=True)
```

Step 3: Exploratory Data Analysis

In this section, we explore the dataset by viewing the dataset, validating the shape of the dataset, and getting a count of the number of movies rated and the number of movies that each user rated.

```
[In]: print((df.count(), len(df.columns)))
[Out]: (100000,3)
```

So, the above output confirms the size of our dataset and we can then validate the datatypes of the input values to check if we need to change/cast any columns' datatypes.

```
[In]: df.printSchema()
[Out]: root
 |-- userId: integer (nullable = true)
 |-- title: string (nullable = true)
 |-- rating: integer (nullable = true)
```

There is a total of three columns out of which two are numerical and the title is categorical. The critical thing with using PySpark for building RS is that we need to have user_id and item_id in numerical form. Hence, we will convert the movie title to numerical values later. We now view a few rows of the dataframe using the rand function to shuffle the records in random order.

```
[In]: df.orderBy(rand()).show(10,False)
[Out]:
```

```
+------+-----------------------------------------+------+
|userId|title                                    |rating|
+------+-----------------------------------------+------+
|13    |Liar Liar (1997)                         |2     |
|741   |Cape Fear (1991)                         |4     |
|916   |Return of the Jedi (1983)                |4     |
|698   |Birdcage, The (1996)                     |2     |
|682   |Primal Fear (1996)                       |3     |
|144   |Empire Strikes Back, The (1980)          |4     |
|887   |Willy Wonka and the Chocolate Factory (1971)|5     |
|389   |Before Sunrise (1995)                    |4     |
|370   |Dante's Peak (1997)                      |2     |
|138   |Truth About Cats & Dogs, The (1996)      |4     |
+------+-----------------------------------------+------+
only showing top 10 rows
```

```
[In]: df.groupBy('userId').count().orderBy('count',
      ascending=False).show(10,False)
[Out]:
```

```
+------+-----+
|userId|count|
+------+-----+
|405   |737  |
|655   |685  |
|13    |636  |
|450   |540  |
|276   |518  |
|416   |493  |
|537   |490  |
|303   |484  |
|234   |480  |
|393   |448  |
+------+-----+
only showing top 10 rows
```

```
[In]: df.groupBy('userId').count().orderBy('count',
      ascending=True).show(10,False)
[Out]:
```

```
+------+-----+
|userId|count|
+------+-----+
|732   |20   |
|631   |20   |
|636   |20   |
|926   |20   |
|93    |20   |
|300   |20   |
|572   |20   |
|596   |20   |
|685   |20   |
|34    |20   |
+------+-----+
only showing top 10 rows
```

The user with the highest number of records has rated 737 movies, and each user has rated at least 20 movies.

```
[In]: df.groupBy('title').count().orderBy('count',
      ascending=False).show(10,False)
[Out]:
```

```
+----------------------------+-----+
|title                       |count|
+----------------------------+-----+
|Star Wars (1977)            |583  |
|Contact (1997)              |509  |
|Fargo (1996)                |508  |
|Return of the Jedi (1983)   |507  |
|Liar Liar (1997)            |485  |
|English Patient, The (1996) |481  |
|Scream (1996)               |478  |
|Toy Story (1995)            |452  |
|Air Force One (1997)        |431  |
|Independence Day (ID4) (1996)|429  |
+----------------------------+-----+
only showing top 10 rows
```

The movie with highest number of ratings is *Star Wars* (1977) and has been rated 583 times, and each movie has been rated by at least by 1 user.

Step 4: Feature Engineering

We now convert the movie title column from categorical to numerical values using StringIndexer. We import the stringIndexer and Indextostring from the PySpark library.

```
[In]: from pyspark.sql.functions import *
[In]: from pyspark.ml.feature import StringIndexer,
      IndexToString
```

Next, we create the stringindexer object by mentioning the input column and output column. Then we fit the object on the dataframe and apply it on the movie title column to create new dataframe with numerical values.

```
[In]: stringIndexer = StringIndexer(inputCol="title",
      outputCol="title_new")

[In]: model = stringIndexer.fit(df)

[In]: indexed = model.transform(df)
```

Let's validate the numerical values of the title column by viewing few rows of the new dataframe (indexed).

```
[In]: indexed.show(10)
[Out]:
```

```
+------+--------------------+------+---------+
|userId|               title|rating|title_new|
+------+--------------------+------+---------+
|   932|    Cape Fear (1991)|     3|    161.0|
|   721|   Piano, The (1993)|     3|    173.0|
|   642|Low Down Dirty Sh...|     2|   1115.0|
|   798|That Darn Cat! (1...|     4|    686.0|
|   535|African Queen, Th...|     4|    199.0|
|   765|Stealing Beauty (...|     5|    521.0|
|   927|Poison Ivy II (1995)|     3|   1041.0|
|   544|    G.I. Jane (1997)|     3|    152.0|
|   788|Godfather: Part I...|     4|    108.0|
|   706|Birdcage, The (1996)|     4|     43.0|
+------+--------------------+------+---------+
only showing top 10 rows
```

As we can see, we now we have an additional column (title_new) with numerical values representing the movie titles. We have to repeat the same procedure in case the user_id is also a categorical type. Just to validate the movie counts, we rerun the groupBy on a new dataframe.

```
[In]: indexed.groupBy('title_new').count().orderBy('count',
      ascending=False).show(10,False)
[Out]:
```

```
+---------+-----+
|title_new|count|
+---------+-----+
|0.0      |583  |
|1.0      |509  |
|2.0      |508  |
|3.0      |507  |
|4.0      |485  |
|5.0      |481  |
|6.0      |478  |
|7.0      |452  |
|8.0      |431  |
|9.0      |429  |
+---------+-----+
only showing top 10 rows
```

Step 5: Splitting the Dataset

Now that we have prepared the data for building the recommender model, we can split the dataset into training and test sets. We split it into a 75 to 25 ratio to train the model and test its accuracy.

```
[In]: train,test=indexed.randomSplit([0.75,0.25])

[In]: train.count()
[Out]: 75104

[In]: test.count()
[Out]: 24876
```

Step 6: Build and Train Recommender Model

We import the ALS function from the PySpark ml library and build the model on the training dataset. There are multiple hyperparameters that can be tuned to improve the performance of the model. Two of the important ones are nonnegative ='True' doesn't create negative ratings in recommendations and coldStartStrategy='drop' to prevent any NaN ratings predictions.

```
[In]: from pyspark.ml.recommendation import ALS
[In]: rec=ALS(maxIter=10,regParam=0.01,userCol='userId',
      itemCol='title_new',ratingCol='rating',nonnegative=True,
      coldStartStrategy="drop")

[In]: rec_model=rec.fit(train)
```

Step 7: Predictions and Evaluation on Test Data

The final part of the entire exercise is to check the performance of the model on unseen or test data. We use the transform function to make predictions on the test data and RegressionEvaluate to check the RMSE value of the model on test data.

```
[In]: predicted_ratings=rec_model.transform(test)
```

```
[In]: predicted_ratings.printSchema()
root
 |-- userId: integer (nullable = true)
 |-- title: string (nullable = true)
 |-- rating: integer (nullable = true)
 |-- title_new: double (nullable = false)
 |-- prediction: float (nullable = false)
```

```
[In]: predicted_ratings.orderBy(rand()).show(10)
[Out]:
```

userId	title	rating	title_new	prediction
92	Tie Me Up! Tie Me...	4	766.0	3.1512196
222	Batman (1989)	3	116.0	3.503284
178	Beauty and the Be...	4	114.0	4.1487904
303	Jerry Maguire (1996)	5	15.0	4.348913
134	Flubber (1997)	2	579.0	2.5635276
295	Henry V (1989)	4	268.0	4.2598643
889	Adventures of Pri...	2	305.0	2.9040515
374	Men in Black (1997)	3	31.0	3.602631
559	Killing Fields, T...	4	276.0	4.55797
290	Star Trek: The Mo...	1	286.0	3.2992659

only showing top 10 rows

```
[xIn]: from pyspark.ml.evaluation import RegressionEvaluator
```

```
[In]: evaluator=RegressionEvaluator(metricName='rmse',
      predictionCol='prediction',labelCol='rating')

[In]: rmse=evaluator.evaluate(predictions)

[In] : print(rmse)

[Out]: 1.0293574739493354
```

The RMSE is not very high; we are making an error of one point in the actual rating and predicted rating. This can be improved further by tuning the model parameters and using the hybrid approach.

Step 8: Recommend Top Movies That Active User Might Like

After checking the performance of the model and tuning the hyperparameters, we can move ahead to recommend top movies to users that they have not seen and might like. The first step is to create a list of unique movies in the dataframe.

```
[In]: unique_movies=indexed.select('title_new').distinct()

[In]: unique_movies.count()

[Out]: 1664
```

So, we have in total 1,664 distinct movies in the dataframe.

```
[In]: a = unique_movies.alias('a')
```

We can select any user within the dataset for which we need to recommend other movies. In our case, we go ahead with userId = 85.

```
[In]: user_id=85
```

We will filter the movies that this active user has already rated or seen.

```
[In]: watched_movies=indexed.filter(indexed['userId'] ==
      user_id).select('title_new').distinct()

[In]: watched_movies.count()
[Out]: 287

[In]: b=watched_movies.alias('b')
```

So, there are total of 287 unique movies out of 1,664 movies that this active user has already rated. So, we would want to recommend movies from the remaining 1,377 items. We now combine both the tables to find the movies that we can recommend by filtering null values from the joined table.

```
[In]: total_movies = a.join(b, a.title_new == b.title_
      new,how='left')

[In]: total_movies.show(10,False)

[Out]:
```

```
+---------+---------+
|title_new|title_new|
+---------+---------+
|299.0    |null     |
|558.0    |null     |
|305.0    |305.0    |
|596.0    |null     |
|1051.0   |null     |
|934.0    |null     |
|496.0    |496.0    |
|769.0    |null     |
|692.0    |null     |
|720.0    |null     |
+---------+---------+
only showing top 10 rows
```

```
[In]: remaining_movies=total_movies.where(col("b.title_new").
      isNull()).select(a.title_new).distinct()

[In]: remaining_movies.count()

[Out]: 1377

[In]: remaining_movies=remaining_movies.withColumn("userId",lit
      (int(user_id)))
[In]: remaining_movies.show(10,False)

[Out]:
```

```
+---------+------+
|title_new|userId|
+---------+------+
|299.0    |85    |
|558.0    |85    |
|596.0    |85    |
|1051.0   |85    |
|934.0    |85    |
|769.0    |85    |
|692.0    |85    |
|720.0    |85    |
|576.0    |85    |
|810.0    |85    |
+---------+------+
only showing top 10 rows
```

Finally, we can now make the predictions on this remaining movie's dataset for the active user using the recommender model that we built earlier. We filter only a few top recommendations that have the highest predicted ratings.

```
[In]: recommendations=rec_model.transform(remaining_movies).
      orderBy('prediction',ascending=False)

[In]: recommendations.show(5,False)
[Out]:
```

```
+---------+------+----------+
|title_new|userId|prediction|
+---------+------+----------+
|1433.0   |85    |4.9689837 |
|1322.0   |85    |4.6927013 |
|1271.0   |85    |4.605163  |
|1470.0   |85    |4.5409293 |
|705.0    |85    |4.532007  |
+---------+------+----------+
```

So, movie titles 1433 and 1322 have the highest predicted rating for this active user (85). We can make it more intuitive by adding the movie title back to the recommendations. We use Indextostring function to create an additional column that returns the movie title.

```
[In]:
      movie_title = IndexToString(inputCol="title_new",
      outputCol="title",labels=model.labels)
[In]: final_recommendations=movie_title.
      transform(recommendations)

[In]: final_recommendations.show(10,False)

[Out]:
```

```
+---------+------+----------+----------------------------+
|title_new|userId|prediction|title                       |
+---------+------+----------+----------------------------+
|1433.0   |85    |4.9689837 |Boys, Les (1997)            |
|1322.0   |85    |4.6927013 |Faust (1994)                |
|1271.0   |85    |4.605163  |Whole Wide World, The (1996)|
|1470.0   |85    |4.5409293 |Some Mother's Son (1996)    |
|705.0    |85    |4.532007  |Laura (1944)                |
|303.0    |85    |4.5236835 |Close Shave, A (1995)       |
|1121.0   |85    |4.4936523 |Crooklyn (1994)             |
|1195.0   |85    |4.4636283 |Pather Panchali (1955)      |
|285.0    |85    |4.456875  |Wrong Trousers, The (1993)  |
|638.0    |85    |4.4495435 |Shall We Dance? (1996)      |
+---------+------+----------+----------------------------+
only showing top 10 rows
```

So, the recommendations for the userId (85) are *Boys, Les* (1997) and F*aust* (1994). This can be nicely wrapped in a single function that executes the above steps in sequence and generates recommendations for active users. The complete code is available on the GitHub repo with this function built in.

Conclusion

In this chapter, we went over various types of recommendation models along with the strengths and limitations of each. We then created a collaborative filtering based recommender system in PySpark using the ALS method to recommend movies to the users.

Clustering

In the previous chapters so far, we have seen supervised Machine Learning where the target variable or label is known to us, and we try to predict the output based on the input features. Unsupervised Learning is different in a sense that there is no labeled data, and we don't try to predict any output as such; instead we try to find interesting patterns and come up with groups within the data. The similar values are grouped together.

When we join a new school or college, we come across many new faces and everyone looks so different. We hardly know anyone in the institute, and there are no groups in place initially. Slowly and gradually, we start spending time with other people and the groups start to develop. We interact with a lot of different people and figure out how similar and dissimilar they are to us. A few months down the line, we are almost settled in our own groups of friends. The friends/members within the group have similar attributes/likings/tastes and hence stay together. Clustering is somewhat similar to this approach of forming groups based on sets of attributes that define the groups.

Starting with Clustering

We can apply clustering on any sort of data where we want to form groups of similar observations and use it for better decision making. In the early days, customer segmentation used to be done by a Rule-based approach, which was much of a manual effort and could only use a limited number

of variables. For example, if businesses wanted to do customer segmentation, they would consider up to 10 variables such as age, gender, salary, location, etc., and create rules-based segments that still gave reasonable performance; but in today's scenario that would become highly ineffective. One reason is data availability is in abundance, and the other is dynamic customer behavior. There are thousands of other variables that can be considered to come up with these machine learning driven segments that are more enriched and meaningful.

When we start clustering, each observation is different and doesn't belong to any group but is based on how similar are the attributes of each observation. We group them in such a way that each group contains the most similar records, and there is as much difference as possible between any two groups. So, how do we measure if two observations are similar or different?

There are multiple approaches to calculate the distance between any two observations. Primarily we represent that any observation is a form of vector that contains values of that observation(A) as shown below.

Age	Salary ($'0000)	Weight (Kgs)	Height(Ft.)
32	8	65	6

Now, suppose we want to calculate the distance of this observation/record from any other observation (B), which also contains similar attributes like those shown below.

Age	Salary ($'000)	Weight (Kgs)	Height(Ft.)
40	15	90	5

We can measure the distance using the Euclidean method, which is straightforward.

It is also known as Cartesian distance. We are trying to calculate the distance of a straight line between any two points; and if the distance between those is points is small, they are more likely to be similar, whereas if the distance is large, they are dissimilar to each other as shown in Figure 8-1.

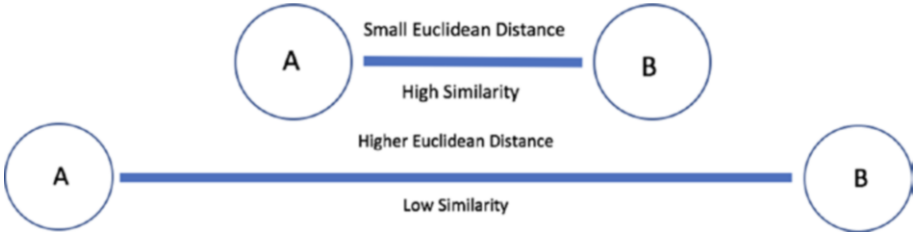


Figure 8-1. *Euclidean Distance based similarity*

The Euclidean distance between any two points can be calculated using the formula below:

$$Dist_{(A,B)} = \sqrt{(A1-B1)^2 + (A2-B2)^2 + (A3-B3)^2 + (A4-B4)^2}$$

$$Dist_{(A,B)} = \sqrt{(Age\ diff)^2 + (Salary\ diff)^2 + (Weight\ diff)^2 + (Height\ diff)^2}$$

$$Dist_{(A,B)} = \sqrt{(32-40)^2 + (8-15)^2 + (65-90)^2 + (6-5)^2}$$

$$Dist_{(A,B)} = \sqrt{(64+49+625+1)}$$

$$Dist_{(A,B)} = 27.18$$

Hence, the Euclidean distance between observations A and B is 27.18. The other techniques to calculate the distance between observations are the following:

1. Manhattan Distance
2. Mahalanobis Distance
3. Minkowski Distances

4. Chebyshev Distance
5. Cosine Distance

The aim of clustering is to have minimal intracluster distance and maximum intercluster difference. We can end up with different groups based on the distance approach that we have used to do clustering, and hence it's critical to be sure of opting for the right distance metric that aligns with the business problem. Before going into different clustering techniques, let quickly go over some of the applications of clustering.

Applications

Clustering is used in a variety of use cases these days ranging from customer segmentation to anomaly detection. Businesses widely use machine learning driven clustering for profiling customer and segmentation to create market strategies around these results. Clustering drives a lot of search engines' results by finding similar objects in one cluster and the dissimilar objects far from each other. It recommends the nearest similar result based on a search query

Clustering can be done in multiple ways based on the type of data and business requirement. The most used ones are the K-means and Hierarchical clustering.

K-Means

'K' stands for a number of clusters or groups that we want to form in the given dataset. This type of clustering involves deciding the number of clusters in advance. Before looking at how K-means clustering works, let's get familiar with a couple of terms first.

1. Centroid
2. Variance

Centroid refers to the center data point at the center of a cluster or a group. It is also the most representative point within the cluster as it's the utmost equidistant data point from the other points within the cluster. The centroid (represented by a cross) for three random clusters is shown in Figure 8-2.

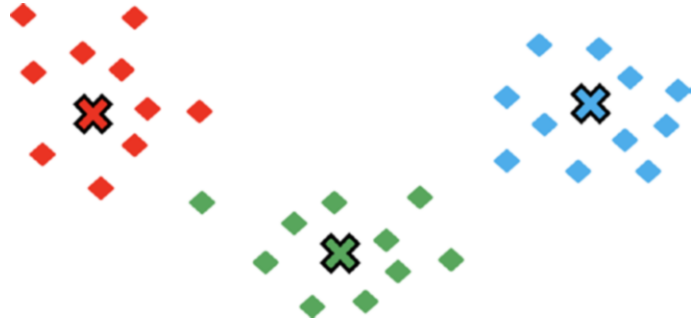


Figure 8-2. *Centroids of clusters*

Each cluster or group contains different number of data points that are nearest to the centroid of the cluster. Once the individual data points change clusters, the centroid value of the cluster also changes. The center position within a group is altered, resulting in a new centroid as shown in Figure 8-3.

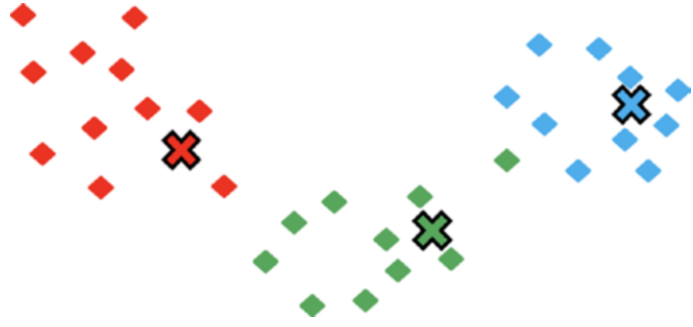


Figure 8-3. *New Centroids of new clusters*

The whole idea of clustering is to minimize intracluster distance, that is, the internal distance of data points from the centroid of cluster and maximize the intercluster distance, that is, between the centroid of two different clusters.

Variance is the total sum of intracluster distances between centroid and data points within that cluster as show in Figure 8-4. The variance keeps on decreasing with an increase in the number of clusters. The more clusters, the less the number of data points within each cluster and hence less variability.

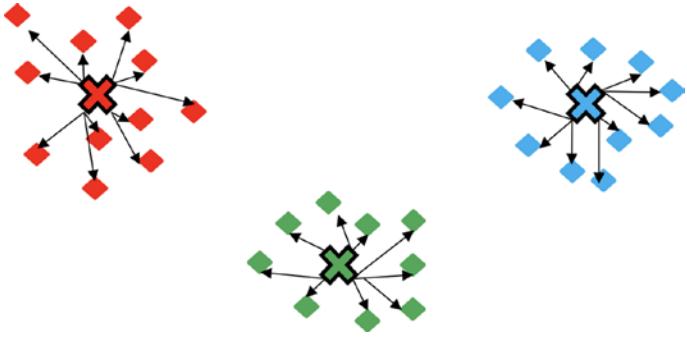


Figure 8-4. *Inracluster Distance*

K-means clustering is composed of four steps in total to form the internal groups within the dataset. We will consider a sample dataset to understand how a K-means clustering algorithm works. The dataset contains a few users with their age and weight values as shown in Table 8-1. Now we will use K-means clustering to come up with meaningful clusters and understand the algorithm.

Table 8-1. *Sample Dataset for K-Means*

User ID	Age	Weight
1	18	80
2	40	60
3	35	100
4	20	45
5	45	120
6	32	65
7	17	50
8	55	55
9	60	90
10	90	50

If we plot these users in a two-dimensional space, we can see that no point belongs to any group initially, and our intention is to find clusters (we can try with two or three) within this group of users such that each group contains similar users. Each user is represented by the age and weight as shown in Figure 8-5.

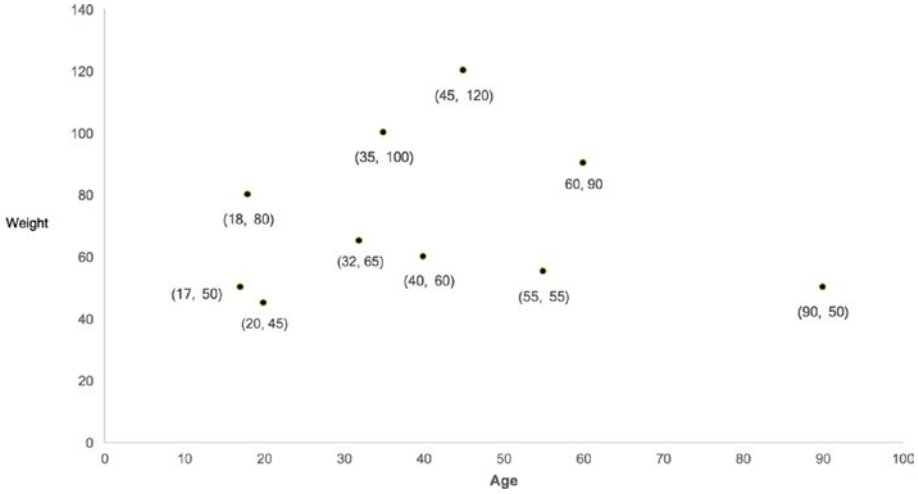


Figure 8-5. *Users before clustering*

Step I: Decide K

It starts with deciding the number of clusters(K) . Most of the time, we are not sure of the right number of groups at the start, but we can find the best number of clusters using a method called the Elbow method based on variability. For this example, let's start with K=2 to keep things simple. So, we are looking for two clusters within this sample data.

Step 2: Random Initialization of Centroids

The next step is to randomly consider any two of the points to be a centroid of new clusters. These can be chosen randomly, so we select user number 5 and user number 10 as the two centroids on the new clusters as shown in Table 8-2.

Table 8-2. *Sample Dataset for K-Means*

User ID	Age	Weight
1	18	80
2	40	60
3	35	100
4	20	45
5 (Centroid 1)	45	120
6	32	65
7	17	50
8	55	55
9	60	90
10 (centroid 2)	90	50

The centroids can be represented by weight and age values as shown in Figure 8-6.

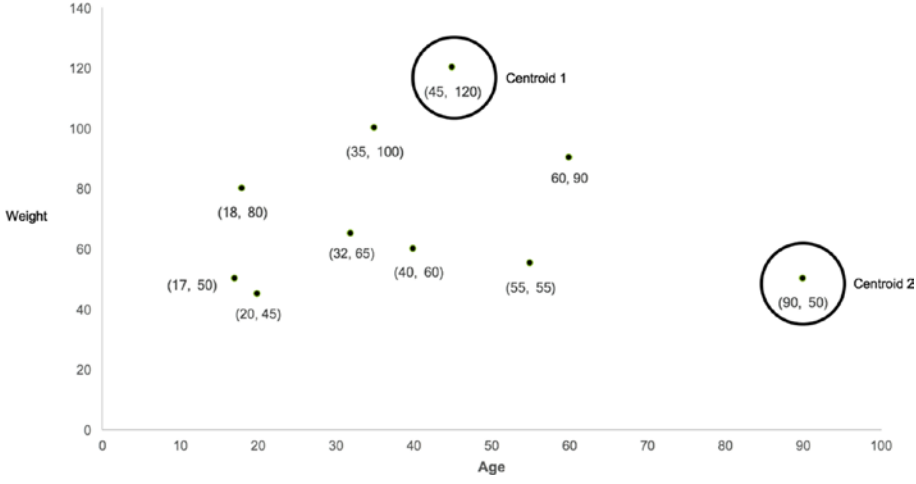


Figure 8-6. *Random Centroids of two clusters*

Step 3: Assigning Cluster Number to each Value

In this step, we calculate the distance of each point from the centroid. In this example, we calculate the Euclidean squared distance of each user from the two centroid points. Based on the distance value, we go ahead and decide which particular cluster the user belongs to (1 or 2). Whichever centroid the user is near to (less distance) would become part of that cluster. The Euclidean squared distance is calculated for each user shown in Table 8-3. The distance of user 5 and user 10 would be zero from respective centroids as they are the same points as centroids.

Table 8-3. *Cluster Assignment Based on Distance from Centroids*

User ID	Age	Weight	ED* from Centroid 1	ED* from Centroid 2	Cluster
1	18	80	48	78	1
2	40	60	60	51	2
3	35	100	22	74	1
4	20	45	79	70	2
5	45	120	0	83	1
6	32	65	57	60	1
7	17	50	75	73	2
8	55	55	66	35	2
9	60	90	34	50	1
10	90	50	83	0	2

*(*Euclidean Distance)*

So, as per the distance from centroids, we have allocated each user to either Cluster 1 or Cluster 2. Cluster 1 contains five users and Cluster 2 also contains five users. The initial clusters are shown in Figure 8-7.

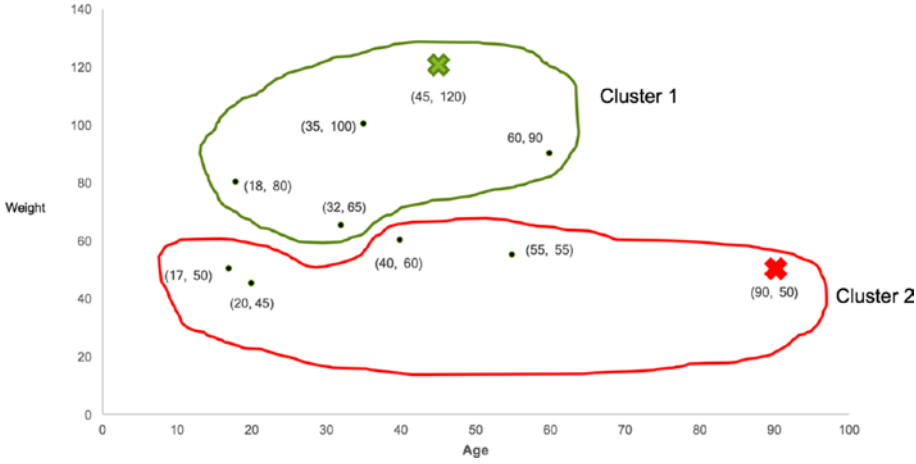


Figure 8-7. *Initial clusters and centroids*

As discussed earlier, the centroids of the clusters are bound to change after inclusion or exclusion of new data points in the cluster. As the earlier centroids (C1, C2) are no longer at the center of clusters, we calculate new centroids in the next step.

Step 4: Calculate New Centroids and Reassign Clusters

The final step in K-means clustering is to calculate the new centroids of clusters and reassign the clusters to each value based on the distance from new centroids. Let's calculate the new centroid of Cluster 1 and Cluster 2. To calculate the centroid of Cluster 1, we simply take the mean of age and weight for only those values that belong to Cluster 1 as shown in Table 8-4.

Table 8-4. *New Centroid Calculation of Cluster 1*

User ID	Age	Weight
1	18	80
3	35	100
5	45	120
6	32	65
9	60	90
Mean Value	**38**	**91**

The centroid calculation for Cluster 2 is also done in a similar manner and shown in Table 8-5.

Table 8-5. *New Centroid calculation of Cluster 2*

User ID	Age	Weight
2	40	60
4	20	45
7	17	50
8	55	55
10	90	50
Mean Value	**44.4**	**52**

Now we have new centroid values for each cluster represented by a cross as shown in Figure 8-8. The arrow signifies the movement of the centroid within the cluster.

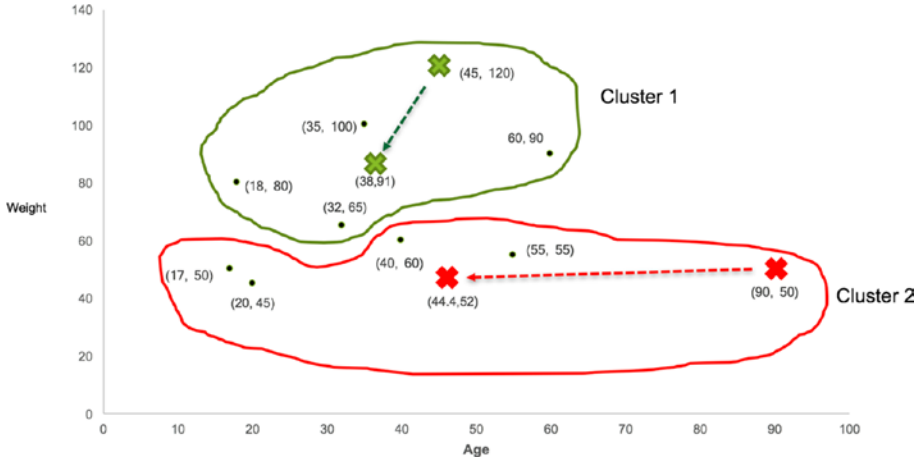


Figure 8-8. *New Centroids of both clusters*

With centroids of each cluster, we repeat step 3 of calculating the Euclidean squared distance of each user from new centroids and find out the nearest centroid. We then reassign the users to either Cluster 1 or Cluster 2 based on the distance from the centroid. In this case, only one value (User 6) changes its cluster from 1 to 2 as shown in Table 8-6.

Table 8-6. *Reallcoation of Clusters*

User ID	Age	Weight	ED* from Centroid 1	ED* from Centroid 2	Cluster
1	18	80	23	38	1
2	40	60	31	9	2
3	35	100	9	49	1
4	20	45	49	25	2
5	45	120	30	68	1

(*continued*)

Table 8-6. (*continued*)

User ID	Age	Weight	ED* from Centroid 1	ED* from Centroid 2	Cluster
6	32	65	27	18	2
7	17	50	46	27	2
8	55	55	40	11	2
9	60	90	22	41	1
10	90	50	66	46	2

Now, Cluster 1 is left with only four users and Cluster 2 contains six users based on the distance from each cluster's centroid as shown in Figure 8-9.

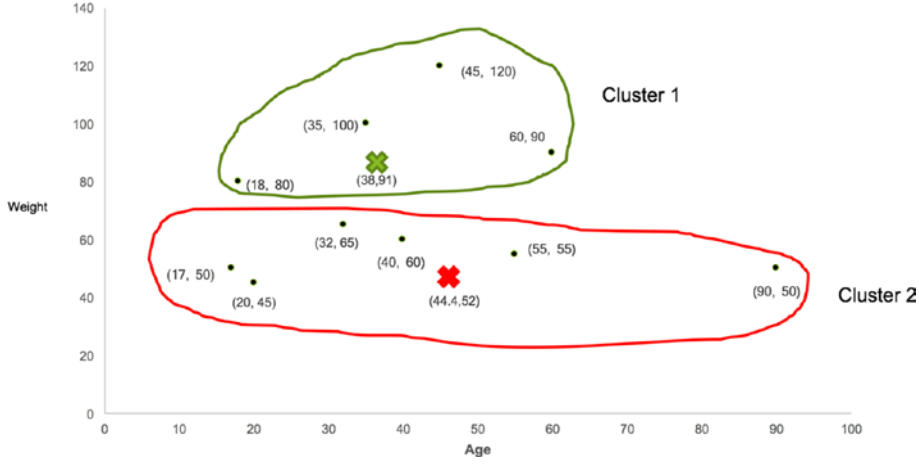


Figure 8-9. *Reallocation of clusters*

We keep repeating the above steps until there is no more change in cluster allocations. The centroids of new clusters are shown in Table 8-7.

Table 8-7. *Calculation of Centroids*

User ID	Age	Weight
1	18	80
3	35	100
5	45	120
9	60	90
Mean Value	39.5	97.5

User ID	Age	Weight
2	40	60
4	20	45
6	32	65
7	17	50
8	55	55
10	90	50
Mean Value	42.33	54.17

As we go through the steps, the centroid movements keep becoming small, and the values almost become part of that particular cluster as shown in Figure 8-10.

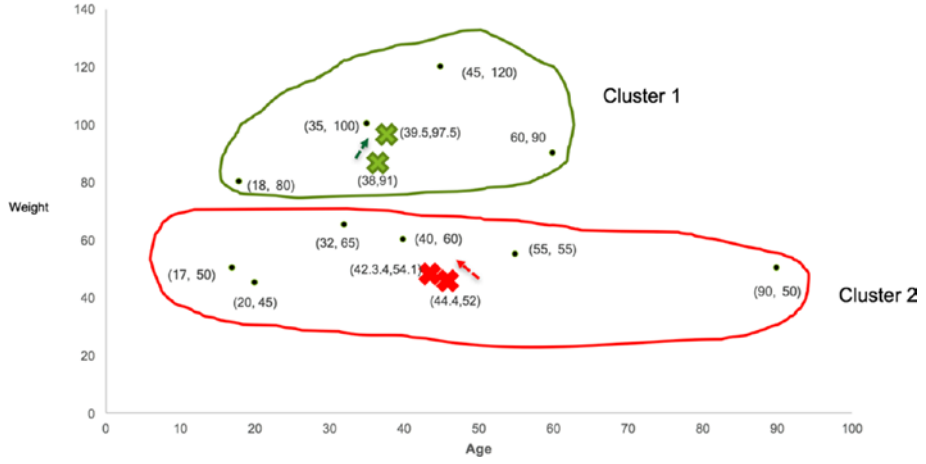


Figure 8-10. *Reallocation of clusters*

As we can observe, there is no more change in the points even after the change in centroids, which completes the K-means clustering. The results can vary as it's based on the first set of random centroids. To reproduce the results, we can set the starting points ourselves as well. The final clusters with values are shown in Figure 8-11.

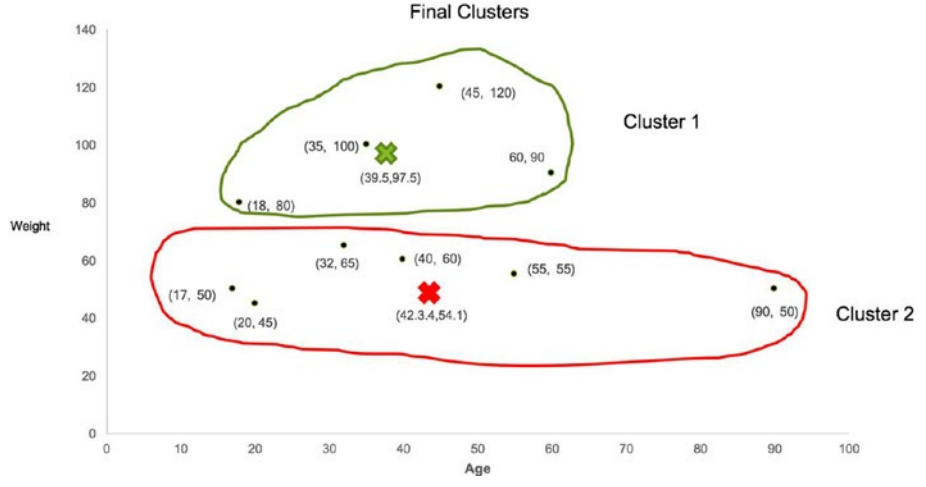


Figure 8-11. *Final clusters*

Cluster 1 contains the users that are average on the height attribute but seem to be very high on the weight variable whereas Cluster 2 seems to be grouping those users together who are taller than average but very conscious of their weight as shown in Figure 8-12.

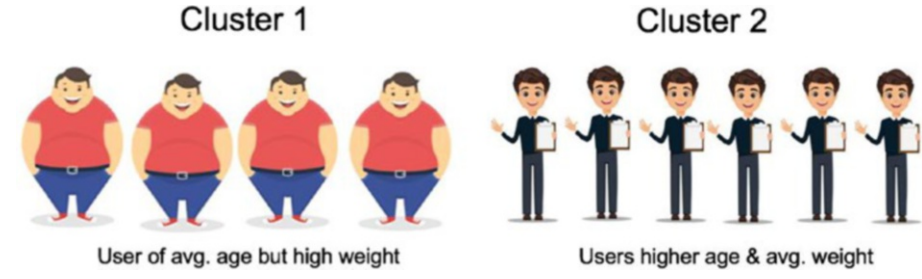


Figure 8-12. *Attributes of final clusters*

Deciding on Number of Clusters (K)

Selecting an optimal number of clusters is quite tricky most of the time as we need a deep understanding of the dataset and the context of the business problem. Additionally, there is no right or wrong answer when it comes to unsupervised learning. One approach might result in a different number of clusters compared to another approach. We have to try and figure out which approach works the best and if the clusters created are relevant enough for decision making. Each cluster can be represented with a few important attributes that signify or give information about that particular cluster. However, there is a method to pick the best possible number of clusters with a dataset. This method is known as the elbow method.

The elbow method helps us to measure the total variance in the data with a number of clusters. The higher the number of clusters, the less the variance would become. If we have an equal number of clusters to the number of records in a dataset, then the variability would be zero because the distance of each point from itself is zero. The variability or SSE (Sum of Squared Errors) along with 'K' values is shown in Figure 8-13.

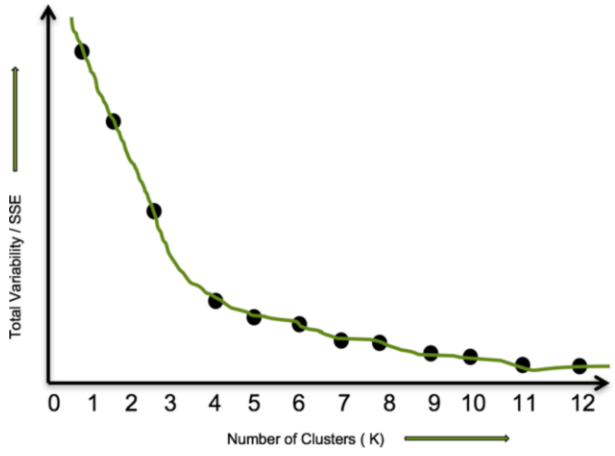


Figure 8-13. *Elbow Method*

As we can observe, there is sort of elbow formation between K values of 3 and 4. There is a sudden reduction in total variance (intra-cluster difference), and the variance sort of declines very slowly after that. In fact, it flattens after the K=9 value. So, the value of K =3 makes the most sense if we go with the elbow method as it captures the most variability with a lesser number of clusters.

Hierarchical Clustering

This is another type of unsupervised machine learning technique and is different from K-means in the sense that we don't have to know the number of clusters in advance. There are two types of Hierarchical clustering.

- Agglomerative Clustering (Bottom-Up Approach)
- Divisive Clustering (Top-Down Approach)

We'll discuss agglomerative clustering as it is the main type. This starts with the assumption that each data point is a separate cluster and gradually keeps combining the nearest values into the same clusters

until all the values become part of one cluster. This is a bottom-up approach that calculates the distance between each cluster and merges the two closest clusters into one. Let's understand the agglomerative clustering with help of visualization. Let's say we have seven data points initially (A1–A7), and they need to be grouped into clusters that contain similar values together using agglomerative clustering as shown in Figure 8-14.

Figure 8-14. *Each value as individual cluster*

At the initial stage (step 1), each point is treated as an individual cluster. In the next step, the distance between every point is calculated and the nearest points are combined together into a single cluster. In this example, A1 and A2, A5 and A6 are nearest to each other and hence form a single cluster as shown in Figure 8-15.

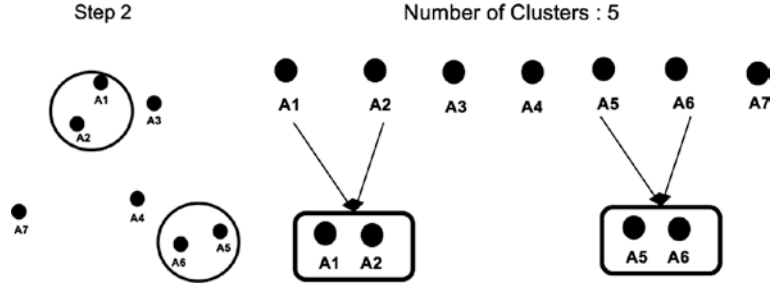


Figure 8-15. *Nearest clusters merged together*

Deciding the most optimal number of clusters while using Hierarchical clustering can be done in multiple ways. One way is to use the elbow method itself and the other option is by making use of something known as a Dendrogram. It is used to visualize the variability between clusters (Euclidean distance). In a Dendrogram, the height of the vertical lines represents the distance between points or clusters and data points listed along the bottom. Each point is plotted on the X-axis and the distance is represented on the Y-axis (length). It is the hierarchical representation of the data points. In this example, the Dendrogram at step 2 looks like the one shown in Figure 8-16.

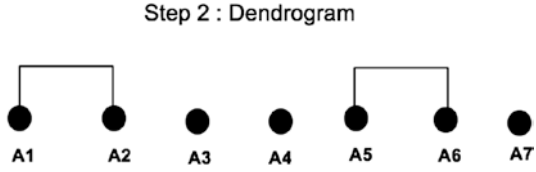


Figure 8-16. *Dendrogram*

In step 3, the exercise of calculating the distance between clusters is repeated and the nearest clusters are combined into a single cluster. This time A3 gets merged with (A1, A2) and A4 with (A5, A6) as shown in Figure 8-17.

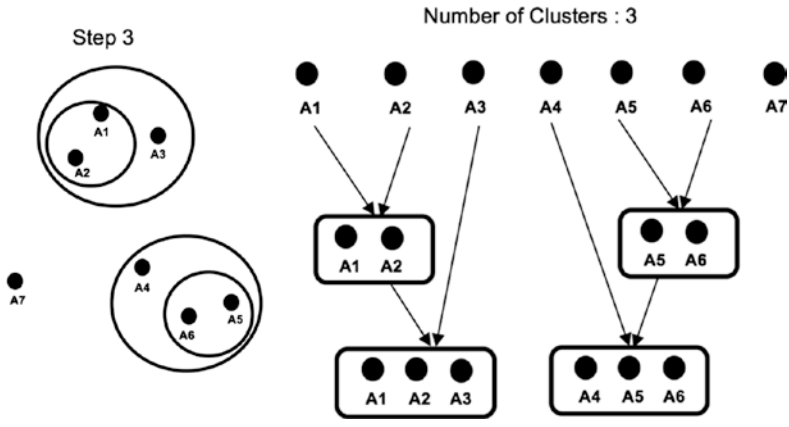


Figure 8-17. *Nearest clusters merged together*

The Dendrogram after step 3 is shown in Figure 8-18.

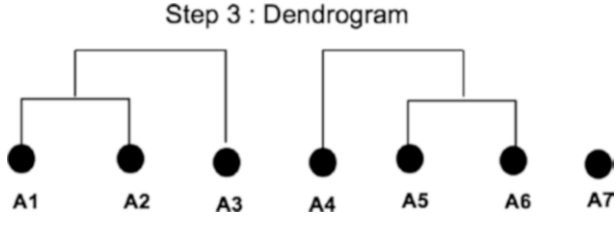


Figure 8-18. *Dendrogram post step 3*

In step 4, the distance between the only remaining point A7 gets calculated and found nearer to Cluster (A4, A5, A6). It is merged with the same cluster as shown in Figure 8-19.

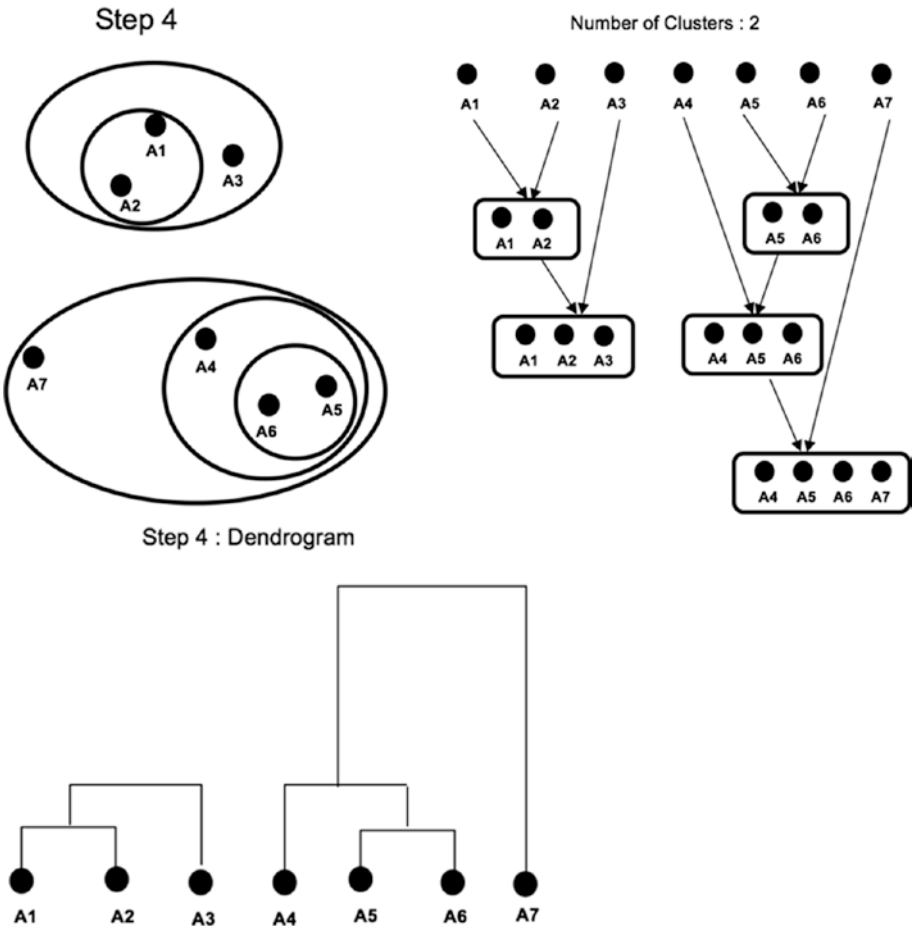


Figure 8-19. *Cluster formation*

At the last stage (step 5), all the points get combined into a single Cluster (A1, A2, A3, A4, A5, A6, A7) as shown in Figure 8-20.

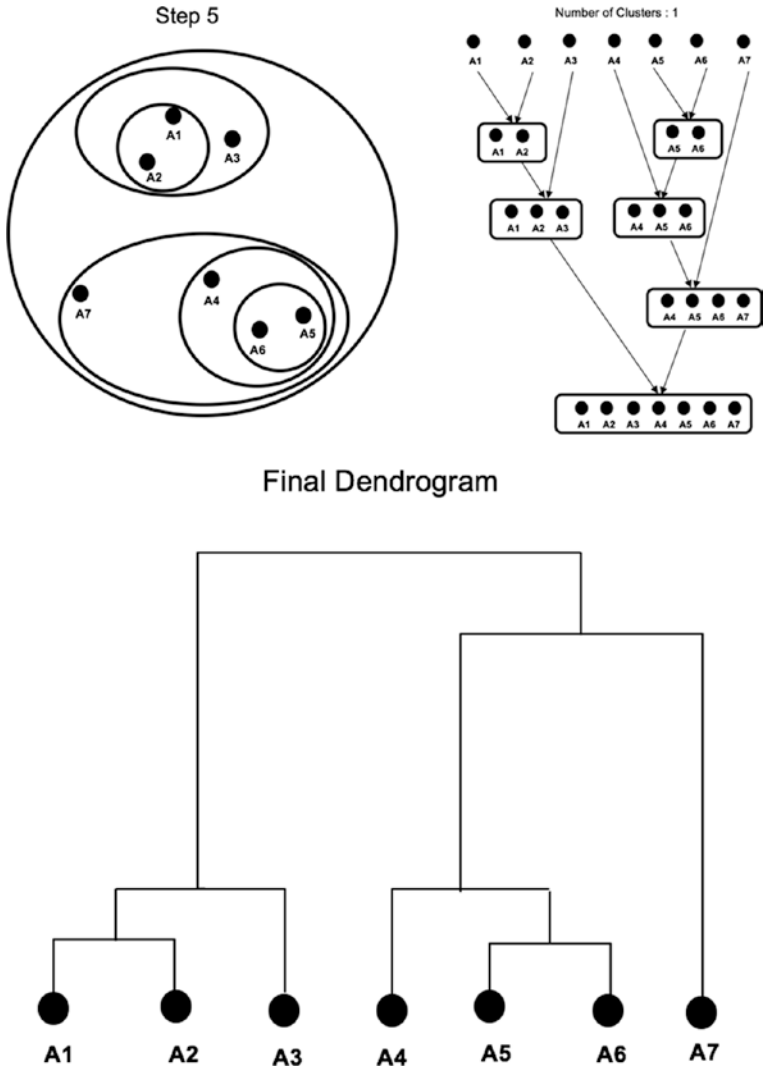


Figure 8-20. *Agglomerative clustering*

Sometimes it is difficult to identify the right number of clusters by the Dendrogram as it can become very complicated and difficult to interpret depending on the dataset being used to do clustering. The Hierarchical clustering doesn't work well on large datasets compared to K-means.

Clustering is also very sensitive to the scale of data points, so it's always advised to do data scaling before clustering. There are other types of clustering that can be used to group the similar data points together such as the following:

1. Gaussian Mixture Model Clustering
2. Fuzzy C-Means Clustering

But the above methods are beyond the scope of this book. We now jump into using a dataset for building clusters using K-means in PySpark.

Code

This section of the chapter covers K-Means clustering using PySpark and Jupyter Notebook.

Note The complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.0 and higher versions.

For this exercise, we consider the most standardized open sourced dataset out there – an IRIS dataset to capture the cluster number and compare supervised and unsupervised performance.

Data Info

The dataset that we are going to use for this chapter is the famous open sourced IRIS dataset and contains a total of 150 records with 5 columns (sepal length, sepal width, petal length, petal width, species). There are 50 records for each type of species. We will try to group these into clusters without using the species label information.

Step 1: Create the SparkSession Object

We start Jupyter Notebook and import SparkSession and create a new SparkSession object to use Spark:

```
[In]: from pyspark.sql import SparkSession
[In]: spark=SparkSession.builder.appName('K_means').
getOrCreate()
```

Step 2: Read the Dataset

We then load and read the dataset within Spark using a dataframe. We have to make sure we have opened PySpark from the same directory folder where the dataset is available or else we have to mention the directory path of the data folder.

```
[In]:
df=spark.read.csv('iris_dataset.csv',inferSchema=True,header=True)
```

Step 3: Exploratory Data Analysis

In this section, we explore the dataset by viewing it and validating its shape.

```
[In]:print((df.count(), len(df.columns)))
[Out]: (150,3)
```

So, the above output confirms the size of our dataset and we can then validate the datatypes of the input values to check if we need to change/cast any columns' datatypes.

```
[In]: df.printSchema()
[Out]: root
 |-- sepal_length: double (nullable = true)
```

```
 |-- sepal_width: double (nullable = true)
 |-- petal_length: double (nullable = true)
 |-- petal_width: double (nullable = true)
 |-- species: string (nullable = true)
```

There is a total of five columns out of which four are numerical and the label column is categorical.

```
[In]: from pyspark.sql.functions import rand

[In]: df.orderBy(rand()).show(10,False)
[Out]:
+------------+-----------+------------+-----------+----------+
|sepal_length|sepal_width|petal_length|petal_width|species   |
+------------+-----------+------------+-----------+----------+
|5.5         |2.6        |4.4         |1.2        |versicolor|
|4.5         |2.3        |1.3         |0.3        |setosa    |
|5.1         |3.7        |1.5         |0.4        |setosa    |
|7.7         |3.0        |6.1         |2.3        |virginica |
|5.5         |2.5        |4.0         |1.3        |versicolor|
|6.3         |2.3        |4.4         |1.3        |versicolor|
|6.2         |2.9        |4.3         |1.3        |versicolor|
|6.3         |2.5        |4.9         |1.5        |versicolor|
|4.7         |3.2        |1.3         |0.2        |setosa    |
|6.1         |2.8        |4.0         |1.3        |versicolor|
+------------+-----------+------------+-----------+----------+

[In]: df.groupBy('species').count().orderBy('count').
      show(10,False)
```

```
[Out]:
+----------+-----+
|species   |count|
+----------+-----+
|virginica |50   |
|setosa    |50   |
|versicolor|50   |
+----------+-----+
```

So, it confirms that there are an equal number of records for each species available in the dataset

Step 4: Feature Engineering

This is the part where we create a single vector combining all input features by using Spark's VectorAssembler. It creates only a single feature that captures the input values for that particular row. So, instead of four input columns (we are not considering a label column since it's an unsupervised machine learning technique), it essentially translates it into a single column with four input values in the form of a list.

```
[In]: from pyspark.ml.linalg import Vector
[In]: from pyspark.ml.feature import VectorAssembler

[In]: input_cols=['sepal_length', 'sepal_width', 'petal_
      length', 'petal_width']

[In]: vec_assembler = VectorAssembler(inputCols = input_cols,
      outputCol='features')

[In]: final_data = vec_assembler.transform(df)
```

Step 5: Build K-Means Clustering Model

The final data contains the input vector that can be used to run K-means clustering. Since we need to declare the value of 'K' in advance before using K-means, we can use elbow method to figure out the right value of 'K'. In order to use the elbow method, we run K-means clustering for different values of 'K'. First, we import K-means from the PySpark library and create an empty list that would capture the variability or SSE (within cluster distance) for each value of K.

```
[In]:from pyspark.ml.clustering import KMeans
[In]:errors=[]
[In]:
for k in range(2,10):
    kmeans = KMeans(featuresCol='features',k=k)
    model = kmeans.fit(final_data)
    intra_distance = model.computeCost(final_data)
    errors.append(intra_distance)
```

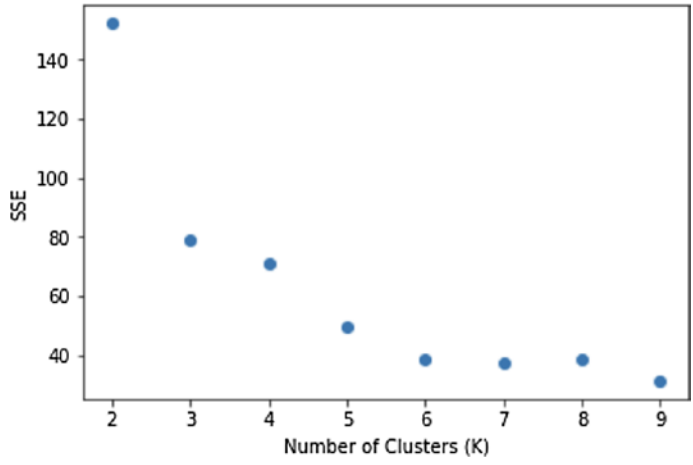
Note The 'K' should have a minimum value of 2 to be able to build clusters.

Now, we can plot the intracluster distance with the number of clusters using numpy and matplotlib.

```
[In]: import pandas as pd
[In]: import numpy as np
[In]: import matplotlib.pyplot as plt
[In]: cluster_number = range(2,10)
[In]: plt.xlabel('Number of Clusters (K)')
[In]: plt.ylabel('SSE')
```

```
[In]: plt.scatter(cluster_number,errors)
[In]: plt.show()
```

[Out]:



In this case, k=3 seems to be the best number of clusters as we can see a sort of elbow formation between three and four values. We build final clusters using k=3.

```
[In]: kmeans = KMeans(featuresCol='features',k=3)
[In]: model = kmeans.fit(final_data)
[In]: model.transform(final_data).groupBy('prediction').
      count().show()
[Out]:
+----------+-----+
|prediction|count|
+----------+-----+
|         1|   50|
|         2|   38|
|         0|   62|
+----------+-----+
```

K-Means clustering gives us three different clusters based on the IRIS data set. We certainly are making a few of the allocations wrong as only one category has 50 records in the group, and the rest of the categories are mixed up. We can use the transform function to assign the cluster number to the original dataset and use a groupBy function to validate the groupings.

```
[In]: predictions=model.transform(final_data)
```

```
[In]: predictions.groupBy('species','prediction').count().
show()
```

```
[Out]:
+----------+----------+-----+
|   species|prediction|count|
+----------+----------+-----+
| virginica|         2|   14|
|    setosa|         0|   50|
| virginica|         1|   36|
|versicolor|         1|    3|
|versicolor|         2|   47|
+----------+----------+-----+
```

As it can be observed, the setosa species is perfectly grouped along with versicolor, almost being captured in the same cluster, but verginica seems to fall within two different groups. K-means can produce different results every time as it chooses the starting point (centroid) randomly every time. Hence, the results that you might get in you K-means clustering might be totally different from these results unless we use a seed to reproduce the results. The seed ensures the split and the initial centroid values remain consistent throughout the analysis.

Step 6: Visualization of Clusters

In the final step, we can visualize the new clusters with the help of Python's matplotlib library. In order to do that, we convert our Spark dataframe into a Pandas dataframe first.

```
[In]: pandas_df = predictions.toPandas()
[In]: pandas_df.head()
```

	sepal_length	sepal_width	petal_length	petal_width	species	features	prediction
5	5.4	3.9	1.7	0.4	setosa	[5.4, 3.9, 1.7, 0.4]	0
95	5.7	3.0	4.2	1.2	versicolor	[5.7, 3.0, 4.2, 1.2]	2
132	6.4	2.8	5.6	2.2	virginica	[6.4, 2.8, 5.6, 2.2]	1
128	6.4	2.8	5.6	2.1	virginica	[6.4, 2.8, 5.6, 2.1]	1
23	5.1	3.3	1.7	0.5	setosa	[5.1, 3.3, 1.7, 0.5]	0

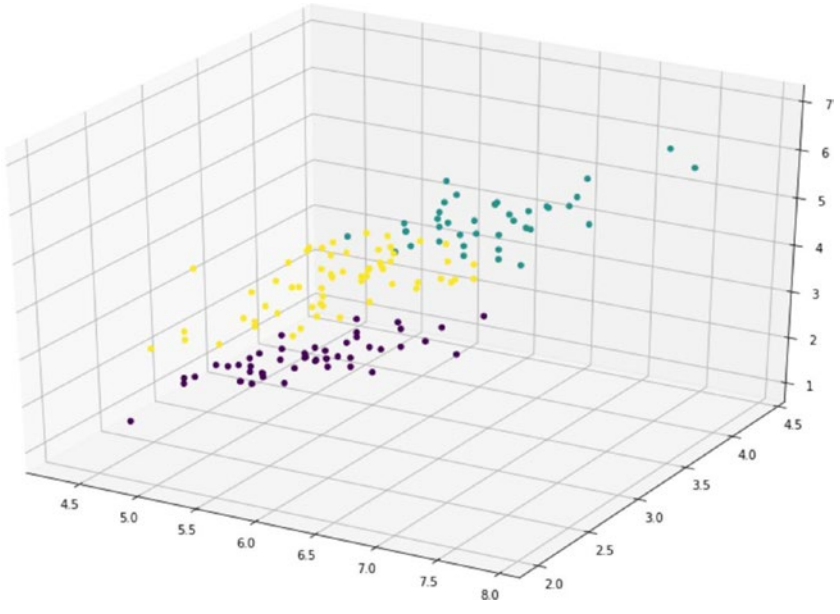
We import the required libraries to plot the third visualization and observe the clusters.

```
[In]: from mpl_toolkits.mplot3d import Axes3D

[In]: cluster_vis = plt.figure(figsize=(12,10)).
      gca(projection='3d')

[In]: cluster_vis.scatter(pandas_df.sepal_length, pandas_
      df.sepal_width, pandas_df.petal_length, c=pandas_
      df.prediction,depthshade=False)

[In]: plt.show()
```



Conclusion

In this chapter, we went over different types of unsupervised machine learning techniques and also built clusters using the K-means algorithms in PySpark. K-Means groups the data points using random centroid initialization whereas Hierarchical clustering focuses on merging entire datapoints into a single cluster. We also covered various techniques to decide the optimal number of clusters like the Elbow method and Dendrogram, which use variance optimization while grouping the data points.

Natural Language Processing

Introduction

This chapter uncovers some of the basic techniques to tackle text data using PySpark. Today's textual form of data is being generated at a lightning pace with multiple social media platforms offering users the options to share their opinions, suggestions, comments, etc. The area that focuses on making machines learn and understand the textual data in order to perform some useful tasks is known as Natural Language Processing (NLP). The text data could be structured or unstructured, and we have to apply multiple steps in order to make it analysis ready. NLP is already a huge contributor to multiple applications. There are many applications of NLP that are heavily used by businesses these days such as chatbot, speech recognition, language translation, recommender systems, spam detection, and sentiment analysis. This chapter demonstrates a series of steps in order to process text data and apply a Machine Learning Algorithm on it. It also showcases the sequence embeddings that can be used as an alternative to traditional input features for classification.

Steps Involved in NLP

There is no right way to do NLP analysis as one can explore multiple ways and take different approaches to handle text data. However, from a Machine Learning standpoint, there are five major steps that one should take to make the text data ready for analysis. The five major steps involved in NLP are:

1. Reading the corpus
2. Tokenization
3. Cleaning /Stopword removal
4. Stemming
5. Converting into Numerical Form

Before jumping into the steps to load and clean text data, let's get familiar with a term known as Corpus as this would keep appearing in the rest of the chapter.

Corpus

A corpus is known as the entire collection of text documents. For example, suppose we have thousands of emails in a collection that we need to process and analyze for our use. This group of emails is known as a corpus as it contains all the text documents. The next step in text processing is tokenization.

Tokenize

The method of dividing the given sentence or collection of words of a text document into separate /individual words is known as tokenization. It removes the unnecessary characters such as punctuation. For example, if we have a sentence such as:

Input: He really liked the London City. He is there for two more days.

Tokens:

He, really, liked, the, London, City, He, is, there, for, two, more, days

We end up with 13 tokens for the above input sentence.

Let us see how we can do tokenization using PySpark. The first step is to create a dataframe that has text data.

```
[In]: df=spark.createDataFrame([(1,'I really liked this movie'),
              (2,'I would recommend this movie to my friends'),
              (3,'movie was alright but acting was horrible'),
              (4,'I am never watching that movie ever again')],
              ['user_id','review'])
[In]: df.show(4,False)
[Out]:
+-------+----------------------------------------+
|user_id|review                                  |
+-------+----------------------------------------+
|1      |I really liked this movie               |
|2      |I would recommend this movie to my friends|
|3      |movie was alright but acting was horrible |
|4      |I am never watching that movie ever again |
+-------+----------------------------------------+
```

In this dataframe, we have four sentences for tokenization. The next step is to import Tokenizer from the Spark library. We have to then pass the input column and name the output column after tokenization. We use the transform function in order to apply tokenization to the review column.

```
[In]: from pyspark.ml.feature import Tokenizer
[In]: tokenization=Tokenizer(inputCol='review',outputCol='tokens')
[In]: tokenized_df=tokenization.transform(df)
[In]: tokenized_df.show(4,False)
[Out]:
```

```
+-------+----------------------------------------+----------------------------------------------+
|user_id|review                                  |tokens                                        |
+-------+----------------------------------------+----------------------------------------------+
|1      |I really liked this movie               |[i, really, liked, this, movie]               |
|2      |I would recommend this movie to my friends|[i, would, recommend, this, movie, to, my, friends]|
|3      |movie was alright but acting was horrible|[movie, was, alright, but, acting, was, horrible]|
|4      |I am never watching that movie ever again|[i, am, never, watching, that, movie, ever, again]|
+-------+----------------------------------------+----------------------------------------------+
```

We get a new column named tokens that contains the tokens for each sentence.

Stopwords Removal

As you can observe, the tokens column contains very common words such as 'this,' 'the,' 'to' , 'was,' 'that,' etc. These words are known as stopwords and they seem to add very little value to the analysis. If they are to be used in analysis, it increases the computation overhead without adding too much value or insight. Hence, it's always considered a good idea to drop these stopwords from the tokens. In PySpark, we use StopWordsRemover to remove the stopwords.

```
[In]: from pyspark.ml.feature import StopWordsRemover
[In]: stopword_removal=StopWordsRemover(inputCol='tokens',
      outputCol='refined_tokens')
      We then pass the tokens as the input column and name the
      output column as refined tokens.
[In]: refined_df=stopword_removal.transform(tokenized_df)
[In]: refined_df.select(['user_id','tokens','refined_tokens']).
      show(4,False)
[Out]:
```

```
+-------+----------------------------------------------+---------------------------------+
|user_id|tokens                                        |refined_tokens                   |
+-------+----------------------------------------------+---------------------------------+
|1      |[i, really, liked, this, movie]               |[really, liked, movie]           |
|2      |[i, would, recommend, this, movie, to, my, friends]|[recommend, movie, friends]  |
|3      |[movie, was, alright, but, acting, was, horrible]|[movie, alright, acting, horrible]|
|4      |[i, am, never, watching, that, movie, ever, again]|[never, watching, movie, ever]|
+-------+----------------------------------------------+---------------------------------+
```

As you can observe, the stopwords like 'I,' 'this,' 'was,' 'am,' 'but,' 'that' are removed from the tokens column.

Bag of Words

This is the methodology through which we can represent the text data into numerical form for it to be used by Machine Learning or any other analysis. Text data is generally unstructured and varies in its length. BOW (Bag of Words) allows us to convert the text form into a numerical vector form by considering the occurrence of the words in text documents. For example,

Doc 1: The best thing in life is to travel

Doc 2: Travel is the best medicine

Doc 3: One should travel more often

Vocabulary:

The list of unique words appearing in all the documents in known as a vocabulary. In the above example, we have 13 unique words that are part of the vocabulary. Each document can be represented by this vector of fixed size 13.

The	best	thing	in	life	is	to	travel	medicine	one	should	more	often

Another element is the representation of the word in the particular document using a Boolean value.

(1 or 0).

Doc 1:

The	best	thing	in	life	is	to	travel	medicine	one	should	more	often
1	1	1	1	1	1	1	1	0	0	0	0	0

Doc 2:

The	best	thing	in	life	is	to	travel	medicine	one	should	more	often
1	1	0	0	0	1	0	1	1	0	0	0	0

Doc 3:

The	best	thing	in	life	is	to	travel	medicine	one	should	more	often
0	0	0	0	0	0	0	1	0	1	1	1	1

The BOW does not consider the order of words in the document and the semantic meaning of the word and hence is the most baseline method to represent the text data into numerical form. There are other ways by which we can convert the textual data into numerical form, which are mentioned in the next section. We will use PySpark to go through each one of these methods.

Count Vectorizer

In BOW, we saw the representation of occurrence of words by simply 1 or 0 and did not consider the frequency of the words. The count vectorizer instead takes count of the word appearing in the particular document. We will use the same text documents that we created earlier while using tokenization. We first import the Count Vectorizer.

```
[In]: from pyspark.ml.feature import CountVectorizer
[In]: count_vec=CountVectorizer(inputCol='refined_tokens',
      outputCol='features')
[In]: cv_df=count_vec.fit(refined_df).transform(refined_df)
[In]: cv_df.select(['user_id','refined_tokens','features']).
      show(4,False)
[Out]:
```

```
+-------+---------------------------------+-------------------------------+
|user_id|refined_tokens                   |features                       |
+-------+---------------------------------+-------------------------------+
|1      |[really, liked, movie]           |(11,[0,4,9],[1.0,1.0,1.0])     |
|2      |[recommend, movie, friends]      |(11,[0,6,10],[1.0,1.0,1.0])    |
|3      |[movie, alright, acting, horrible]|(11,[0,2,3,5],[1.0,1.0,1.0,1.0])|
|4      |[never, watching, movie, ever]   |(11,[0,1,7,8],[1.0,1.0,1.0,1.0])|
+-------+---------------------------------+-------------------------------+
```

As we can observe, each sentence is represented as a dense vector. It shows that the vector length is 11 and the first sentence contains 3 values at the 0th, 4th, and 9th indexes.

To validate the vocabulary of the count vectorizer, we can simply use the vocabulary function.

```
[In]: count_vec.fit(refined_df).vocabulary
[Out]:
['movie',
 'horrible',
 'really',
 'alright',
 'liked',
 'friends',
 'recommend',
 'never',
 'ever',
 'acting',
 'watching']
```

So, the vocabulary size for the above sentences is 11 and if you look at the features carefully, they are similar to the input feature vector that we have been using for Machine Learning in PySpark. The drawback of using the Count Vectorizer method is that it doesn't consider the co-occurrences of words in other documents. In simple terms, the words appearing more often would have a larger impact on the feature vector. Hence, another approach to convert text data into numerical form is known as Term Frequency – inverse Document Frequency (TF-IDF).

TF-IDF

This method tries to normalize the frequency of word occurrence based on other documents. The whole idea is to give more weight to the word if appearing a high number of times in the same document but penalize if it is appearing a higher number of times in other documents as well. This indicates that a word is common across the corpus and is not as important as its frequency in the current document indicates.

Term Frequency: Score based on the frequency of word in current document.

Inverse Document Frequency: Scoring based on frequency of documents that contains the current word.

Now, we create features based on TF-IDF in PySpark using the same refined df dataframe.

```
[In]: from pyspark.ml.feature import HashingTF,IDF
[In]: hashing_vec=HashingTF(inputCol='refined_tokens',
      outputCol='tf_features')
[In]: hashing_df=hashing_vec.transform(refined_df)
[In]: hashing_df.select(['user_id','refined_tokens',
      'tf_features']).show(4,False)
[Out]:
```

```
+-------+-----------------------------------+-----------------------------------------------------------+
|user_id|refined_tokens                     |tf_features                                                |
+-------+-----------------------------------+-----------------------------------------------------------+
|1      |[really, liked, movie]             |(262144,[14,32675,155321],[1.0,1.0,1.0])                   |
|2      |[recommend, movie, friends]        |(262144,[129613,155321,222394],[1.0,1.0,1.0])              |
|3      |[movie, alright, acting, horrible]|(262144,[80824,155321,236263,240286],[1.0,1.0,1.0,1.0])|
|4      |[never, watching, movie, ever]     |(262144,[63139,155321,203802,245806],[1.0,1.0,1.0,1.0])|
+-------+-----------------------------------+-----------------------------------------------------------+
```

```
[In]: tf_idf_vec=IDF(inputCol='tf_features',outputCol='tf_idf_
      features')
[In]: tf_idf_df=tf_idf_vec.fit(hashing_df).transform(hashing_df)
[In]: tf_idf_df.select(['user_id','tf_idf_features']).show(4,False)
[Out]:
```

```
+-------+-----------------------------------------------------------------------------------------------------------------------------------------------+
|user_id|tf_idf_features                                                                                                                                |
+-------+-----------------------------------------------------------------------------------------------------------------------------------------------+
|1      |(262144,[14,32675,155321],[0.9162907318741551,0.9162907318741551,0.0])                                                                         |
|2      |(262144,[129613,155321,222394],[0.9162907318741551,0.0,0.9162907318741551])                                                                    |
|3      |(262144,[80824,155321,236263,240286],[0.9162907318741551,0.0,0.9162907318741551,0.9162907318741551])                                           |
|4      |(262144,[63139,155321,203802,245806],[0.9162907318741551,0.0,0.9162907318741551,0.9162907318741551])                                           |
+-------+-----------------------------------------------------------------------------------------------------------------------------------------------+
```

Text Classification Using Machine Learning

Now that we have a basic understanding of the steps involved in dealing with text processing and feature vectorization, we can build a text classification model and use it for predictions on text data. The dataset that we are going to use is the open source labeled Movie Lens reviews data, and we are going to predict the sentiment class of any given review (positive or negative). Let's start with reading the text data first and creating a Spark dataframe.

```
[In]: text_df=spark.read.csv('Movie_reviews.csv',inferSchema=
      True,header=True,sep=',')
[In]: text_df.printSchema()
[Out]:
root
 |-- Review: string (nullable = true)
 |-- Sentiment: string (nullable = true)
```

You can observe the Sentiment column in StringType, and we will need it to convert it into an Integer or float type going forward.

```
[In]: text_df.count()
[Out]: 7087
```

We have close to seven thousand records out of which some might not be labeled properly. Hence, we filter only those records that are labeled correctly.

```
[In]: text_df=text_df.filter(((text_df.Sentiment =='1') |
      (text_df.Sentiment =='0')))
[In]: text_df.count()
[Out]: 6990
```

Some of the records got filtered out and we are now left with 6,990 records for the analysis. The next step is to validate a number of reviews for each class.

```
[In]: text_df.groupBy('Sentiment').count().show()
[Out]:
+---------+-----+
|Sentiment|count|
+---------+-----+
|        0| 3081|
|        1| 3909|
+---------+-----+
```

We are dealing with a balanced dataset here as both classes have almost a similar number of reviews. Let us look at a few of the records in the dataset.

```
[In]: from pyspark.sql.functions import rand
[In]: text_df.orderBy(rand()).show(10,False)
[Out]:
```

```
+------------------------------------------------------------+---------+
|Review                                                      |Sentiment|
+------------------------------------------------------------+---------+
|Mission Impossible 3 was excellent.                         |1        |
|I wanted desperately to love'The Da Vinci Code as a film.|1        |
|love Harry Potter.                                          |1        |
|Oh, and Brokeback Mountain was a terrible movie.            |0        |
|Love luv lubb the Da Vinci Code!                            |1        |
|Brokeback Mountain was so awesome.                          |1        |
|""" DA VINCI CODE SUCKS."                                   |0        |
|"I liked the first "" Mission Impossible."                  |1        |
|I love Harry Potter.                                        |1        |
|i love your mission impossible move.                        |1        |
+------------------------------------------------------------+---------+
```

In the next step, we create a new label column as an Integer type and drop the original Sentiment column, which was a String type.

```
[In]: text_df=text_df.withColumn("Label", text_df.Sentiment.
      cast('float')).drop('Sentiment')
[In]: text_df.orderBy(rand()).show(10,False)
[Out]:
```

```
+------------------------------------------------------------------------+-----+
|Review                                                                  |Label|
+------------------------------------------------------------------------+-----+
|I hate Harry Potter.                                                    |0.0  |
|I am the only person in the world who thought Brokeback Mountain sucked.|0.0  |
|Not because I hate Harry Potter, but because I am the type of person tha|0.0  |
|Which is why i said silent hill turned into reality coz i was hella like|1.0  |
|The Da Vinci Code sucked big time.                                      |0.0  |
|The Da Vinci Code sucked big time.                                      |0.0  |
|Ok brokeback mountain is such a horrible movie.                         |0.0  |
|A / N: This is a gift for sivullinen who requested: " I'd love some NC  |1.0  |
|, she helped me bobbypin my insanely cool hat to my head, and she laughe|0.0  |
|I love the Da Vinci Code.                                               |1.0  |
+------------------------------------------------------------------------+-----+
```

We also include an additional column that captures the length of the review.

```
[In]: from pyspark.sql.functions import length
[In]: text_df=text_df.withColumn('length',length(text_df['Review']))
[In]: text_df.orderBy(rand()).show(10,False)
[Out]:
```

```
+------------------------------------------------------------------------+-----+------+
|Review                                                                  |Label|length|
+------------------------------------------------------------------------+-----+------+
|I have to say that I loved Brokeback Mountain.                          |1.0  |46    |
|The Da Vinci Code is awesome!!                                          |1.0  |30    |
|Oh, and Brokeback Mountain was a terrible movie.                        |0.0  |48    |
|man i loved brokeback mountain!                                         |1.0  |31    |
|Even though Brokeback Mountain is one of the most depressing movies, eve|0.0  |72    |
|da vinci code sucks...                                                  |0.0  |22    |
|Combining the opinion / review from Gary and Gin Zen, The Da Vinci Code |0.0  |71    |
|da vinci code sucks...                                                  |0.0  |22    |
|Finally feel up to making the long ass drive out to the Haunt tonight...|1.0  |72    |
|the last stand and Mission Impossible 3 both were awesome movies.       |1.0  |65    |
+------------------------------------------------------------------------+-----+------+
```

```
[In]: text_df.groupBy('Label').agg({'Length':'mean'}).show()
[Out]:
+-----+-----------------+
|Label|      avg(Length)|
+-----+-----------------+
|  1.0|47.61882834484523|
|  0.0|50.95845504706264|
+-----+-----------------+
```

There is no major difference between the average length of the positive and negative reviews. The next step is to start the tokenization process and remove stopwords.

```
[In]: tokenization=Tokenizer(inputCol='Review',outputCol='tokens')
[In]: tokenized_df=tokenization.transform(text_df)
[In]: stopword_removal=StopWordsRemover(inputCol='tokens',
      outputCol='refined_tokens')
[In]: refined_text_df=stopword_removal.transform(tokenized_df)
```

Since we are now dealing with tokens only instead of an entire review, it would make more sense to capture a number of tokens in each review rather than using the length of the review. We create another column (token count) that gives the number of tokens in each row.

```
[In]: from pyspark.sql.functions import udf
[In]: from pyspark.sql.types import IntegerType
[In]: from pyspark.sql.functions import *
[In]: len_udf = udf(lambda s: len(s), IntegerType())
[In]: refined_text_df = refined_text_df.withColumn("token_count",
      len_udf(col('refined_tokens')))
[In]: refined_text_df.orderBy(rand()).show(10)
[Out]:
```

```
+--------------------+-----+------+--------------------+--------------------+-----------+
|              Review|Label|length|              tokens|      refined_tokens|token_count|
+--------------------+-----+------+--------------------+--------------------+-----------+
|da vinci code was...|  1.0|    37|[da, vinci, code,...|[da, vinci, code,...|          5|
|Not because I hat...|  0.0|    72|[not, because, i,...|[hate, harry, pot...|          6|
|I love Harry Potter.|  1.0|    20|[i, love, harry, ...|[love, harry, pot...|          3|
|and I love Da Vin...|  1.0|    71|[and, i, love, da...|[love, da, vinci,...|          7|
|Da Vinci Code = U...|  0.0|    72|[da, vinci, code,...|[da, vinci, code,...|         15|
|Brokeback Mountai...|  1.0|    34|[brokeback, mount...|[brokeback, mount...|          3|
|I think I hate Ha...|  0.0|    72|[i, think, i, hat...|[think, hate, har...|          9|
|Harry Potter is b...|  1.0|    26|[harry, potter, i...|[harry, potter, b...|          3|
|The Da Vinci Code...|  1.0|    30|[the, da, vinci, ...|[da, vinci, code,...|          4|
|Combining the opi...|  0.0|    71|[combining, the, ...|[combining, opini...|         10|
+--------------------+-----+------+--------------------+--------------------+-----------+
```

Now that we have the refined tokens after stopword removal, we can use any of the above approaches to convert text into numerical features. In this case, we use a countvectorizer for feature vectorization for the Machine Learning Model.

```
[In]:count_vec=CountVectorizer(inputCol='refined_tokens',
     outputCol='features')
[In]: cv_text_df=count_vec.fit(refined_text_df).transform
      (refined_text_df)
[In]: cv_text_df.select(['refined_tokens','token_count','features',
      'Label']).show(10)
[Out]:
+--------------------+-----------+--------------------+-----+
|      refined_tokens|token_count|            features|Label|
+--------------------+-----------+--------------------+-----+
|[da, vinci, code,...|          5|(2302,[0,1,4,43,2...|  1.0|
|[first, clive, cu...|          9|(2302,[11,51,229,...|  1.0|
|[liked, da, vinci...|          5|(2302,[0,1,4,53,3...|  1.0|
|[liked, da, vinci...|          5|(2302,[0,1,4,53,3...|  1.0|
|[liked, da, vinci...|          8|(2302,[0,1,4,53,6...|  1.0|
|[even, exaggerati...|          6|(2302,[46,229,271...|  1.0|
|[loved, da, vinci...|          8|(2302,[0,1,22,30,...|  1.0|
|[thought, da, vin...|          7|(2302,[0,1,4,228,...|  1.0|
|[da, vinci, code,...|          6|(2302,[0,1,4,33,2...|  1.0|
|[thought, da, vin...|          7|(2302,[0,1,4,223,...|  1.0|
+--------------------+-----------+--------------------+-----+
```

```
[In]: model_text_df=cv_text_df.select(['features',
      'token_count','Label'])
```

Once we have the feature vector for each row, we can make use of VectorAssembler to create input features for the machine learning model.

```
[In]: from pyspark.ml.feature import VectorAssembler
[In]: df_assembler = VectorAssembler(inputCols=['features',
      'token_count'],outputCol='features_vec')
[In]: model_text_df = df_assembler.transform(model_text_df)
[In]: model_text_df.printSchema()

[Out]:
 root
 |-- features: vector (nullable = true)
 |-- token_count: integer (nullable = true)
 |-- Label: float (nullable = true)
 |-- features_vec: vector (nullable = true)
```

We can use any of the classification models on this data, but we proceed with training the Logistic Regression Model.

```
[In]: from pyspark.ml.classification import LogisticRegression
[In]: training_df,test_df=model_text_df.randomSplit([0.75,0.25])
```

To validate the presence of enough records for both classes in the train and test dataset, we can apply the groupBy function on the Label column.

```
[In]: training_df.groupBy('Label').count().show()
[Out]:
+-----+-----+
|Label|count|
+-----+-----+
|  1.0| 2979|
|  0.0| 2335|
+-----+-----+
```

```
[In]: test_df.groupBy('Label').count().show()
[Out]:
+-----+-----+
|Label|count|
+-----+-----+
|  1.0|  930|
|  0.0|  746|
+-----+-----+
[In]: log_reg=LogisticRegression(featuresCol='features_vec',
      labelCol='Label').fit(training_df)
```

After training the model, we evaluate the performance of the model on the test dataset.

```
[In]: results=log_reg.evaluate(test_df).predictions
[In]: results.show()
[Out]:
```

features	token_count	Label	features_vec	rawPrediction	probability	prediction
(2302,[0,1,4,5,64...	6	1.0	(2303,[0,1,4,5,64...	[-18.830941540939...	[6.63477190315568...	1.0
(2302,[0,1,4,5,89...	9	1.0	(2303,[0,1,4,5,89...	[-29.076874095578...	[2.35545075686894...	1.0
(2302,[0,1,4,5,30...	5	1.0	(2303,[0,1,4,5,30...	[-24.244556501413...	[2.95612568707732...	1.0
(2302,[0,1,4,5,44...	5	1.0	(2303,[0,1,4,5,44...	[-21.509618767712...	[4.55503020457943...	1.0
(2302,[0,1,4,5,82...	6	1.0	(2303,[0,1,4,5,82...	[-15.544123371433...	[1.77530475650336...	1.0
(2302,[0,1,4,11,1...	6	0.0	(2303,[0,1,4,11,1...	[19.7679077548244...	[0.99999999740039...	0.0
(2302,[0,1,4,11,4...	7	1.0	(2303,[0,1,4,11,4...	[-19.715182049118...	[2.74034489094736...	1.0
(2302,[0,1,4,12,1...	8	1.0	(2303,[0,1,4,12,1...	[-11.939533517377...	[6.52715073472097...	1.0
(2302,[0,1,4,12,1...	5	1.0	(2303,[0,1,4,12,1...	[-18.598733876142...	[8.36897949082172...	1.0
(2302,[0,1,4,12,3...	8	1.0	(2303,[0,1,4,12,3...	[-26.587158762213...	[2.84016558119887...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0
(2302,[0,1,4,12,3...	5	1.0	(2303,[0,1,4,12,3...	[-23.308368615675...	[7.53883308804207...	1.0

```
[In]: from pyspark.ml.evaluation import
      BinaryClassificationEvaluator

[In]: true_postives = results[(results.Label == 1) & (results.
      prediction == 1)].count()
```

```
[In]: true_negatives = results[(results.Label == 0) & (results.
      prediction == 0)].count()
[In]: false_positives = results[(results.Label == 0) &
      (results.prediction == 1)].count()
[In]: false_negatives = results[(results.Label == 1) &
      (results.prediction == 0)].count()
```

The performance of the model seems reasonably good, and it is able to differentiate between positive and negative reviews easily.

```
[In]: recall = float(true_postives)/(true_postives + false_
      negatives)
[In]:print(recall)
[Out]: 0.986021505376344

[In]: precision = float(true_postives) / (true_postives +
      false_positives)
[In]: print(precision)
[Out]: 0.9572025052192067

[In]: accuracy=float((true_postives+true_negatives) /(results.
      count()))
[In]: print(accuracy)
[Out]: 0.9677804295942721
```

Sequence Embeddings

Millions of people visit business websites every day, and each one of them takes a different set of steps in order to seek the right information/product. Yet most of them leave disappointed or dejected for some reason, and very few get to the right page within the website. In this kind of situation, it becomes difficult to find out if the potential customer actually got the information that he was looking for. Also, the individual journeys of these

viewers can't be compared to each other since every person has done a different set of activities. So, how can we know more about these journeys and compare these visitors to each other? Sequence Embedding is a powerful way that offers us the flexibility to not only compare any two distinct viewers' entire journeys in terms of similarity but also to predict the probability of their conversion. Sequence embeddings essentially help us to move away from using traditional features to make predictions and considers not only the order of the activities of a user but also the average time spent on each of the unique pages to translate into more robust features; and it also used in Supervised Machine Learning across multiple use cases (next possible action prediction, converted vs. non-converted, product classification). Using traditional machine learning models on the advanced features like sequence embeddings, we can achieve tremendous results in terms of prediction accuracy, but the real benefit lies in visualizing all these user journeys and observing how distinct these paths are from the ideal ones.

This part of the chapter will unfold the process creating sequence embeddings for each user's journey in PySpark.

Embeddings

So far, we have seen representation of text data into numerical form using techniques like count vectorizer, TF-IDF, and hashing vectorization. However, none of the above techniques consider semantic meanings of the words or the context in which words are present. Embeddings are unique in terms of capturing the context of the words and representing them in such a way that words with similar meanings are represented with similar sort of embeddings. There are two ways to calculate the embeddings.

1. Skip Gram
2. Continuous Bag of Words (CBOW)

Both methods give the embedding values that are nothing but weights of the hidden layer in a neural network. These embedding vectors can be of size 100 or more depending on the requirement. The word2vec gives the embedding values for each word where as doc2vec gives the embeddings for the entire sentence. Sequence Embeddings are similar to doc2vec and are the result of weighted means of the individual embedding of the word appearing in the sentence.

Let's take a sample dataset to illustrate how we can create sequence embeddings from an online retail journey of users.

```
[In]: spark=SparkSession.builder.appName('seq_embedding').
      getOrCreate()
[In]:
df = spark.read.csv('embedding_dataset.csv',header=True,
     inferSchema=True)
[In]: df.count()
[Out]: 1096955
```

The total number of records in the dataset is close to one million, and there are 0.1 million unique users. The time spent by each user on each of the web pages is also tracked along with the final status if the user bought the product or not.

```
[In]: df.printSchema()
[Out]:
root
 |-- user_id: string (nullable = true)
 |-- page: string (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- visit_number: integer (nullable = true)
 |-- time_spent: double (nullable = true)
 |-- converted: integer (nullable = true)
```

```
[In]: df.select('user_id').distinct().count()
[Out]: 104087
```

```
[In]: df.groupBy('page').count().orderBy('count',
      ascending=False).show(10,False)
[Out]:
+-------------+------+
|page         |count |
+-------------+------+
|product info |767131|
|homepage     |142456|
|added to cart|67087 |
|others       |39919 |
|offers       |32003 |
|buy          |24916 |
|reviews      |23443 |
+-------------+------+
```

```
[In]: df.select(['user_id','page','visit_number','time_spent',
      'converted']).show(10,False)
[Out]:
```

user_id	page	visit_number	time_spent	converted
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	homepage	0	0.16666667	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	product info	0	0.4	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	product info	0	0.31666666	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	product info	0	0.6333333	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	product info	0	0.15	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	homepage	1	0.8333333	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	product info	1	0.16666667	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	product info	2	0.16666667	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	buy	2	0.016666668	1
8057ed24427be18922f640b20b60997e7d070946b6c8f48117ae4d6dad0ebb23	added to cart	2	0.41666666	1

The whole idea of sequence embeddings is to translate the series of steps taken by the user during his or her online journey into a page sequence, which can be used for calculating embedding scores. The first step is to remove any of the consecutive duplicate pages during the journey of a user. We create an additional column that captures the

previous page of a user. Window is a function in spark that helps to apply certain logic specific to individual or group of rows in the dataset.

```
[In]:w = Window.partitionBy("user_id").orderBy('timestamp')
[In]: df = df.withColumn("previous_page", lag("page", 1,
      'started').over(w))
[In]: df.select('user_id','timestamp','previous_page','page').
      show(10,False)
[Out]:
```

user_id	timestamp	previous_page	page
004e96d0dc01f2541b7e5be735da6321b15f797ded220d5f6fb9d66910b5ce88	2017-04-10 20:23:09	started	product info
004e96d0dc01f2541b7e5be735da6321b15f797ded220d5f6fb9d66910b5ce88	2017-04-10 20:26:23	product info	product info
004e96d0dc01f2541b7e5be735da6321b15f797ded220d5f6fb9d66910b5ce88	2017-04-12 14:12:40	product info	product info
004e96d0dc01f2541b7e5be735da6321b15f797ded220d5f6fb9d66910b5ce88	2017-04-12 20:49:33	product info	product info
004e96d0dc01f2541b7e5be735da6321b15f797ded220d5f6fb9d66910b5ce88	2017-04-13 12:18:12	product info	product info
01158797281955155c5c6bbe7daaa368021adcc4eaf4b3794e1789b5ee412a34	2018-02-21 23:47:13	started	homepage
01158797281955155c5c6bbe7daaa368021adcc4eaf4b3794e1789b5ee412a34	2018-02-21 23:49:17	homepage	homepage
01158797281955155c5c6bbe7daaa368021adcc4eaf4b3794e1789b5ee412a34	2018-02-22 00:07:58	homepage	homepage
01158797281955155c5c6bbe7daaa368021adcc4eaf4b3794e1789b5ee412a34	2018-02-22 11:08:24	homepage	homepage
01158797281955155c5c6bbe7daaa368021adcc4eaf4b3794e1789b5ee412a34	2018-02-22 11:08:32	homepage	added to cart

```
[In]:
def indicator(page, prev_page):
    if page == prev_page:
        return 0
    else:
        return 1

[In]:page_udf = udf(indicator,IntegerType())
[In]: df = df.withColumn("indicator",page_udf(col('page'),
      col('previous_page'))) \
        .withColumn('indicator_cummulative',
         sum(col('indicator')).over(w))
```

Now, we create a function to check if the current page is similar to the previous page and indicate the same in a new column indicator. Indicator cumulative is the column to track the number of distinct pages during the user's journey.

```
[In]: df.select('previous_page','page','indicator',
      'indicator_cummulative').show(20,False)
[Out]:
```

```
+-------------+-------------+---------+---------------------+
|previous_page|page         |indicator|indicator_cummulative|
+-------------+-------------+---------+---------------------+
|started      |product info |1        |1                    |
|product info |product info |0        |1                    |
|product info |product info |0        |1                    |
|product info |product info |0        |1                    |
|product info |product info |0        |1                    |
|started      |homepage     |1        |1                    |
|homepage     |homepage     |0        |1                    |
|homepage     |homepage     |0        |1                    |
|homepage     |homepage     |0        |1                    |
|homepage     |added to cart|1        |2                    |
|added to cart|homepage     |1        |3                    |
|homepage     |added to cart|1        |4                    |
|added to cart|homepage     |1        |5                    |
|started      |homepage     |1        |1                    |
|homepage     |product info |1        |2                    |
|product info |product info |0        |2                    |
|product info |product info |0        |2                    |
|product info |product info |0        |2                    |
|product info |product info |0        |2                    |
|started      |product info |1        |1                    |
+-------------+-------------+---------+---------------------+
```

We keep creating new windows object to partition the data further in order to build the sequences for eadch user.

```
[In]: w2=Window.partitionBy(["user_id",'indicator_
      cummulative']).orderBy('timestamp')
[In]: df= df.withColumn('time_spent_cummulative',
      sum(col('time_spent')).over(w2))
[In]: df.select('timestamp','previous_page','page',
      'indicator','indicator_cummulative','time_spent',
      'time_spent_cummulative').show(20,False)
```

[Out]:

timestamp	previous_page	page	indicator	indicator_cummulative	time_spent	time_spent_cummulative
2017-04-10 20:23:09	started	product info	1	1	3.2333333	3.2333333
2017-04-10 20:26:23	product info	product info	0	1	0.08	3.3133333
2017-04-12 14:12:40	product info	product info	0	1	0.08	3.3933333
2017-04-12 20:49:33	product info	product info	0	1	0.08	3.4733333
2017-04-13 12:18:12	product info	product info	0	1	0.08	3.5533333000000002
2018-02-21 23:47:13	started	homepage	1	1	0.16666667	0.16666667
2018-02-21 23:49:17	homepage	homepage	0	1	0.06666667	0.23333334
2018-02-22 00:07:58	homepage	homepage	0	1	0.06666667	0.30000001
2018-02-22 11:08:24	homepage	homepage	0	1	0.13333334	0.43333334999999995
2018-02-22 11:08:32	homepage	added to cart	1	2	0.11666667	0.11666667
2018-02-22 11:10:08	added to cart	homepage	1	3	0.05	0.05
2018-02-22 11:10:11	homepage	added to cart	1	4	0.083333336	0.083333336
2018-02-22 12:31:58	added to cart	homepage	1	5	1.65	1.65
2017-12-09 21:35:03	started	homepage	1	1	0.25	0.25
2017-12-09 21:35:18	homepage	product info	1	2	0.1	0.1
2017-12-09 21:36:14	product info	product info	0	2	0.15	0.25
2017-12-09 21:36:23	product info	product info	0	2	0.33333334	0.58333334
2017-12-09 21:36:52	product info	product info	0	2	0.23333333	0.81666667
2017-12-09 21:42:31	product info	product info	0	2	0.21666667	1.03333334
2017-04-24 06:45:25	started	product info	1	1	0.15	0.15

In the next stage, we calculate the aggregated time spent on similar pages so that only a single record can be kept for representing consecutive pages.

```
[In]: w3 = Window.partitionBy(["user_id",'indicator_
      cummulative']).orderBy(col('timestamp').desc())

[In]: df = df.withColumn('final_page',first('page').over(w3))\
     .withColumn('final_time_spent',first('time_spent_
      cummulative').over(w3))

[In]: df.select(['time_spent_cummulative','indicator_cummulative',
      'page','final_page','final_time_spent']).show(10,False)
```

[Out]:

time_spent_cummulative	indicator_cummulative	page	final_page	final_time_spent
3.5533333000000002	1	product info	product info	3.5533333000000002
3.4733333	1	product info	product info	3.5533333000000002
3.3933333	1	product info	product info	3.5533333000000002
3.3133333	1	product info	product info	3.5533333000000002
3.2333333	1	product info	product info	3.5533333000000002
0.43333334999999995	1	homepage	homepage	0.43333334999999995
0.30000001	1	homepage	homepage	0.43333334999999995
0.23333334	1	homepage	homepage	0.43333334999999995
0.16666667	1	homepage	homepage	0.43333334999999995
0.11666667	2	added to cart	added to cart	0.11666667

```
[In]: aggregations=[]
[In]: aggregations.append(max(col('final_page')).alias('page_emb'))
[In]: aggregations.append(max(col('final_time_spent')).
      alias('time_spent_emb'))
[In]: aggregations.append(max(col('converted')).
      alias('converted_emb'))

[In]: df_embedding = df.select(['user_id','indicator_cummulative',
      'final_page','final_time_spent','converted']).groupBy
      (['user_id','indicator_cummulative']).agg(*aggregations)

[In]: w4 = Window.partitionBy(["user_id"]).orderBy('indicator_
      cummulative')
[In]: w5 = Window.partitionBy(["user_id"]).orderBy(col
      ('indicator_cummulative').desc())
```

Finally, we use a collect list to combine all the pages of a user's journey into a single list and for time spent as well. As a result, we end with the user journey in the form of a page list and time spent list.

```
[In]:df_embedding = df_embedding.withColumn('journey_page',
collect_list(col('page_emb')).over(w4))\
                          .withColumn('journey_time_temp',
collect_list(col('time_spent_emb')).over(w4)) \
                          .withColumn('journey_page_final',
first('journey_page').over(w5))\
                         .withColumn('journey_time_final',
first('journey_time_temp').over(w5)) \
                         .select(['user_id','journey_page_final',
'journey_time_final','converted_emb'])
```

We continue with only unique user journeys. Each user is represented by a single journey and time spent vector.

```
[In]: df_embedding = df_embedding.dropDuplicates()

[In]: df_embedding.count()
[Out]: 104087

[In]: df_embedding.select('user_id').distinct().count()
[Out]: 104087

[In]: df_embedding.select('user_id','journey_page_
      final','journey_time_final').show(10)
[Out]:
```

```
+--------------------+--------------------+--------------------+
|             user_id|  journey_page_final|  journey_time_final|
+--------------------+--------------------+--------------------+
|004e96d0dc01f2541...|      [product info]|[3.5533333000000002]|
|01158797281955155...|[homepage, added ...|[0.43333334999999...|
|020d29467c7810a85...|[homepage, produc...|  [0.25, 1.03333334]|
|032d6e8c20f41c18e...|      [product info]|       [2.329999975]|
|0377ebcf2ac8f8aef...|      [product info]|[1.0699999999999998]|
|03f484c3d0dc5afaf...|[homepage, produc...|[0.65000001000000...|
|040828e6773148d00...|      [product info]|       [1.430000006]|
|05bd9a73b6f61509e...|      [product info]|       [14.79999967]|
|068ea915e886eb11c...|[homepage, others...|[0.5, 0.5, 0.9799...|
|06a572f2d5e9d5c56...|      [product info]|              [0.56]|
+--------------------+--------------------+--------------------+
```

Now that we have the user journeys and time spent list, we convert this dataframe to a Pandas dataframe and build a word2vec model using these journey sequences. We have to install a gensim library first in order to use word2vec.We use the embedding size of 100 to keep it simple.

```
[In]: pd_df_emb0 = df_embedding.toPandas()
[In]: pd_df_embedding = pd_df_embedding.reset_index(drop=True)

[In]: !pip install gensim
[In]: from gensim.models import Word2Vec
[In]: EMBEDDING_SIZE = 100
```

```
[In]: model = Word2Vec(pd_df_embedding['journey_page_final'],
      size=EMBEDDING_SIZE)
[In]: print(model)
[Out]: Word2Vec(vocab=7, size=100, alpha=0.025)
```

As we can observe, the vocabulary size is 7 because we were dealing with 7 page categories only. Each of these pages category now can be represented with help of an embedding vector of size 100.

```
[In]: page_categories = list(model.wv.vocab)
[In]: print(page_categories)
[Out]:
['product info', 'homepage', 'added to cart', 'others',
'reviews', 'offers', 'buy']
```

```
[In]: print(model['reviews'])
[Out]:
[ 0.47489035  0.53834176 -0.47785276 -0.26488945 -0.2656599  -0.04322785
  0.24362227  0.26430744 -0.48902759 -0.31393662  0.41263634  0.78189737
  0.58100605 -0.4459599  -0.70117044 -0.63221812 -0.6908192  -0.6791628
 -0.02506013  0.21131983 -0.02721698 -0.20559131 -0.78862274 -0.55389541
 -0.1507041   0.7149269  -0.24301411  0.29431018 -0.52848756 -0.500494
  0.16006927 -0.10355954 -0.36789769 -0.01349463 -0.40723842  0.15346751
 -0.79262614 -0.67456675 -0.18617149  0.69221032  0.53981733  0.75779319
  0.0573662   0.85435468  0.78063792  0.57342744 -0.16319969  0.46502107
 -0.09518502  0.60525858  0.31979162 -0.26889852 -0.12189896  0.65022558
  0.07857032  0.06138223  0.15626955  0.23680885  0.33999926 -0.54703128
 -0.21992962  0.83436728 -0.34557605 -0.69831383  0.4595826  -0.49346444
 -0.14114673  0.37797749  0.70894194  0.55426389 -0.40428343 -0.67311144
 -0.46010655 -0.44518954  0.7340765  -0.04775194 -0.44416061  0.45019379
 -0.54332632 -0.48565596  0.093257   -0.5141685   0.24856164  0.39611688
 -0.15698397 -0.45113751 -0.15056689  0.75211751 -0.06628865  0.07008368
  0.46780539 -0.13114813 -0.61940897 -0.29163912 -0.3338908   0.40938324
  0.08697812  0.74824899  0.53244144 -0.20717621]
```

```
[In]: model['offers'].shape
[Out]: (100,)
```

To create the embedding matrix, we can use a model and pass the model vocabulary; it would result in a matrix of size (7,100.)

```
[In]: X = model[model.wv.vocab]
[In]: X.shape
[Out]: (7,100)
```

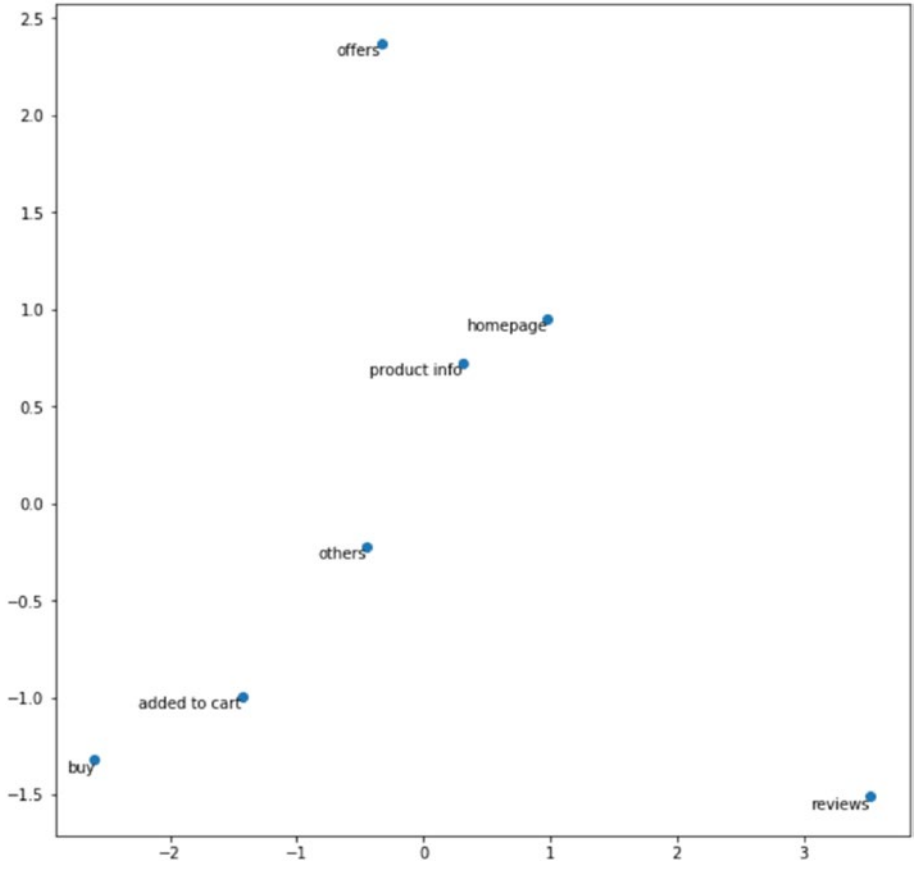
In order to better understand the relation between these page categories, we can use a dimensionality reduction technique (PCA) and plot these seven page embeddings on a two-dimensional space.

```
[In]: pca = PCA(n_components=2)
[In]: result = pca.fit_transform(X)

[In]: plt.figure(figsize=(10,10))
[In]: plt.scatter(result[:, 0], result[:, 1])

[In]: for i,page_category in enumerate(page_categories):
      plt.annotate(page_category,horizontalalignment='right',
      verticalalignment='top',xy=(result[i, 0], result[i, 1]))

[In]: plt.show()
```



As we can clearly see, the embeddings of buy and added to cart are near to each other in terms of similarity whereas homepage and product info are also closer to each other. Offers and reviews are totally separate when it comes to representation through embeddings. These individual embeddings can be combined and used for user journey comparison and classification using Machine Learning.

Note A complete dataset along with the code is available for reference on the GitHub repo of this book and executes best on Spark 2.3 and higher versions.

Conclusion

In this chapter, we became familiar with the steps to do text processing and create feature vectors for Machine Learning. We also went through the process of creating sequence embeddings from online user journey data for comparing various user journeys.

Index

E

F

G

H

I

J

K

L

M, N, O

P, Q

R

S

T

U

V, W, X, Y, Z