

Implementing Predictive Analytics with Spark in Azure HDInsight

Lab 2 – Exploring and Preprocessing Data

Overview

In this lab, you will provision an HDInsight Spark cluster. You will then use the Spark cluster to explore data interactively.

What You'll Need

To complete the labs, you will need the following:

- A web browser
- A Microsoft account
- A Microsoft Azure subscription
- A Windows, Linux, or Mac OS X computer
- The lab files for this course

Note: To set up the required environment for the lab, follow the instructions in the [Setup](#) document for this course. Specifically, you must have signed up for an Azure subscription and installed the Azure CLI tool.

Provisioning an HDInsight Spark Cluster

The first task you must perform is to provision an HDInsight Spark cluster.

Note: The Microsoft Azure portal is continually improved in response to customer feedback. The steps in this exercise reflect the user interface of the Microsoft Azure portal at the time of writing, but may not match the latest design of the portal exactly.

Provision an HDInsight Cluster

Note: If you already have a Spark HDInsight cluster running, you can skip this procedure

1. In a web browser, navigate to <http://portal.azure.com>, and if prompted, sign in using the Microsoft account that is associated with your Azure subscription.
2. In the Microsoft Azure portal, add a new HDInsight cluster with the following settings:
3. In the Microsoft Azure portal, in the Hub Menu, click **New**. Then in the **Data + Analytics** menu, click **HDInsight**.
4. In the **New HDInsight Cluster** blade, enter the following settings, and then click **Create**:
 - **Cluster Name:** Enter a unique name (and make a note of it!)

- **Subscription:** *Select your Azure subscription*
 - **Select Cluster Type:**
 - **Cluster Type:** Spark
 - **Operating System:** Linux
 - **Version:** *Select the latest available version of Spark*
 - **Cluster Tier:** Standard
 - **Applications:** *None*
 - **Credentials:**
 1. **Cluster Login Username:** *Enter a user name of your choice (and make a note of it!)*
 2. **Cluster Login Password:** *Enter and confirm a strong password (and make a note of it!)*
 - **SSH Username:** *Enter another user name of your choice (and make a note of it!)*
 - **SSH Authentication Type:** Password
 - **SSH Password:** *Enter and confirm a strong password (and make a note of it!)*
 - **Data Source:**
 - **Create a new storage account:** *Enter a unique name consisting of lower-case letters and numbers only (and make a note of it!)*
 - **Choose Default Container:** *Enter the cluster name you specified previously*
 - **Location:** *Select any available region*
 - **Pricing:**
 - **Number of Worker nodes:** 1
 - **Worker Nodes Pricing Tier:** *Use the default selection*
 - **Head Node Pricing Tier:** *Use the default selection*
 - **Optional Configuration:** *None*
 - **Resource Group:** *Create a new resource group with a unique name*
 - **Pin to dashboard:** *Not selected*
5. In the Azure portal, view **Notifications** to verify that deployment has started. Then wait for the cluster to be deployed (this can take a long time – often 30 minutes or more. Feel free to go for a coffee break!)

Note: As soon as an HDInsight cluster is running, the credit in your Azure subscription will start to be charged. The free-trial subscription includes a credit limit of approximately \$200 (or local equivalent) that you can spend over a period of 30 days, which is enough to complete the labs in this course as long as clusters are deleted when not in use. If you decide not to complete this lab, follow the instructions in the *Clean Up* procedure at the end of the lab to delete your cluster in order to avoid using your Azure credit unnecessarily.

Uploading Source Data

The azure command-line interface is a cross-platform tool that you can use to work with Azure services, including HDInsight. In this exercise, you will use the Azure CLI to upload a data file to the Azure blob store for processing with Spark.

Upload a File to Azure Storage

Note: You can use any Azure Storage client to upload the files to your Azure storage container. The instructions here assume you are using the Azure command line interface.

1. Open a new command line window.
2. Enter the following command to switch the Azure CLI to resource manager mode.

```
azure config mode arm
```

Note: If a command not found error is displayed, ensure that you have followed the instructions in the setup guide to install the Azure CLI.

3. Enter the following command to log into your Azure subscription:

```
azure login
```

4. Follow the instructions that are displayed to browse to the Azure device login site and enter the authentication code provided. Then sign into your Azure subscription using your Microsoft account.
5. Enter the following command to view your Azure resources:

```
azure resource list
```

6. Verify that your HDInsight cluster and the related storage account are both listed. Note that the information provided includes the resource group name as well as the individual resource names.
7. Note the resource group and storage account name.
8. Enter the following command on a single line to determine the connection string for your Azure storage account, specifying the storage account and resource group names you noted earlier:

```
azure storage account connectionstring show account -g resource_group
```

9. Note the connection string (which starts "DefaultEndpointsProtocol=" and ends "=="), copying it to the clipboard if your command line tool supports it.
10. If you are working on a Windows client computer, enter the following command to set a system variable for the connection string:

```
SET AZURE_STORAGE_CONNECTION_STRING=your_connection_string
```

If you are using a Linux or Mac OS X client computer, enter the following command to set a system variable for the connection string (enter the connection string in quotation marks):

```
export AZURE_STORAGE_CONNECTION_STRING="your_connection_string"
```

11. Enter a `dir` (Windows) or `ls` (Mac OS X or Linux) command to view the contents of the **HDPredictLabs\Lab01** folder where you extracted the lab files for this course (for example, `dir c:\DAT202.3x\Lab02` or `ls DAT202.3x/Lab02`), and verify that this folder contains a file named **iris-multiclass.csv**. This file contains observations of different flowers, their attributes and types.
12. Enter the following command on a single line to upload the `iris-multiclass.csv` file to the container used by your HDInsight cluster. Replace *iris_path* with the local path to `iris-multiclass.csv` (for example `c:\DAT202.3x\Lab02\iris-multiclass.csv` or `HDPredictLabs/Lab02/iris-multiclass.csv`) and replace *container* with the name of the storage container used by your cluster (which should be the same as the cluster name):

```
azure storage blob upload iris_path container
```

13. Wait for the file to be uploaded.

Exploring Data

HDInsight provides Jupyter Notebook support for Spark. Jupyter supports both Python and Scala. In this exercise you will use Scala to explore data in the iris dataset.

Tip: You can copy and paste the code for this exercise from **Iris Exploration-Python.txt** or **Iris Exploration-Scala.txt** in the lab folder for this lab.

Create a Notebook

1. In your web browser, in the Azure Portal, view the blade for your cluster. Then, under **Quick Links**, click **Cluster Dashboards**; and in the **Cluster Dashboards** blade, click **Jupyter Notebook**.
2. If you are prompted, enter the HTTP user name and password you specified for your cluster when provisioning it (not the SSH user name).
3. In the Jupyter web page that opens in a new tab, on the **Files** tab, note that there are folders for **PySpark** and **Scala**.
4. If there is currently no folder named **Labs**, in the **New** drop-down menu, click **Folder**. This creates a folder named **Untitled Folder**. Then select the checkbox for **Untitled Folder**, above the list of folders, click **Rename**, and rename the directory to **Labs**.
5. Open the **Labs** folder, and then in the **New** drop-down list, click **Spark** if you prefer to program in Scala, or **PySpark** if you prefer to program in Python. This creates a new notebook named **Untitled**.
6. At the top of the new notebook, click **Untitled**, and rename notebook to **Iris Data Exploration**.

Explore Data in an RDD

RDDs are the basic building block of programming in Spark. In this task, you will explore the Iris data by loading it into an RDD.




1. In the empty cell at the top of the notebook, enter the following Scala or Python code to load the iris-multiclass.csv file you uploaded earlier into an RDD, and count the number of lines it contains:

Python

```
input = sc.textFile("wasb:///iris-multiclass.csv")
input.count()
```

Scala

```
val input = sc.textFile("wasb:///iris-multiclass.csv")
input.count()
```

2. With cursor still in the cell, on the toolbar click the **run cell, select below** button. As the code runs the  symbol at the top right of the page changes to a  symbol, and then returns to  when the code has finished running.
3. When the code has finished running (it can take a short time to execute initially while the Spark context is created), view the output, which should display the number of lines of text in the file.
4. In the new cell that has appeared, enter the following to view the first Iris flower in the dataset:

Python

```
input.first()
```

Scala

```
input.first()
```

5. Click the **run cell, select below** button to run the contents of this cell. You should be able to see a comma-separated list of doubles and a string value containing the features we'll be using and the label of the Iris flower
6. In the cell that appears below, enter the following code which will split the Iris rows by comma, create a map-reduce function to enable each of the different classes of Iris to be counted, and display the first three items:

Python

```
splitter = input.map(lambda line: line.split(','))
num = splitter.map(lambda line: (line[4], 1)).reduceByKey(lambda a, b:
a + b)
num.take(3)
```

Scala

```
val splitter = input.map(line => line.split(','))
val num = splitter.map(line => (line(4), 1)).reduceByKey((a, b) => a +
b)
num.take(3)
```

7. Click the **run cell, select below** button to run the contents of this cell.
8. Review the output, which shows three different species of iris and the count of each species in the source data:

Explore Data in a DataFrame

While RDDs are a good data structure for working with datasets, they can be tricky to work with. In this task, you'll load the iris data from the RDD into a dataframe and use it to explore the data.

1. In the empty cell at the bottom of the notebook, enter the following to split the dataset again by comma and take all of the feature values and use a case class called Iris to represent them. Then create a Spark DataFrame from the output:

Python

```
from pyspark.sql.types import *
schemaString = "SepalLength SepalWidth PetalLength PetalWidth Species"

fields = [StructField(field_name, StringType(), True) for field_name in
schemaString.split()]
schema = StructType(fields)

df = sqlContext.createDataFrame(splitter, schema)
df.groupBy("Species").count().show()
```

Scala

```
import sqlContext.implicits._
case class Iris(SepalLength : Double, SepalWidth : Double, PetalLength
: Double, PetalWidth : Double, Species : String)
val splitter = input.map(line => line.split(',')).map{ line =>
  Iris(line(0).toDouble, line(1).toDouble, line(2).toDouble,
line(3).toDouble, line(4))
}
val df = splitter.toDF()
```

```
df.groupBy("Species").count().show()
```

2. Click the **run cell, select below** button to run the contents of this cell. Note that the dataframe that is created contains **SepalLength**, **SepalWidth**, **PetalLength**, **PetalWidth**, and **Species** columns.
3. In the new cell below, enter the following code to create a table from the dataframe and query it using Spark SQL:

Python

```
df.registerTempTable("iris")
sqlContext.sql("SELECT Species, COUNT(Species) FROM iris GROUP BY Species").show()
```

Scala

```
df.registerTempTable("iris")
sqlContext.sql("SELECT Species, COUNT(Species) FROM iris GROUP BY Species").show()
```

4. Click the **run cell, select below** button to run the contents of the cell and verify that the species counts are returned.
5. On the **File** menu, click **Close and Halt**.

Quantizing and Normalizing Data

A common task when preparing to create machine learning models is to preprocess data by quantizing (or *binning*) numerical data into categorical ranges and normalizing data so that numerical values lie within similar scales.

Tip: You can copy and paste the code for this exercise from **Quantizing and Normalizing-Python.txt** or **Quantizing and Normalizing -Scala.txt** in the lab folder for this lab.

Create a New Notebook

You can use Scala code in a Jupyter notebook to preprocess data.

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark** if you prefer to program in Scala, or **PySpark** if you prefer to program in Python. This creates a new notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Quantizing and Normalizing**.

Quantize Data

In this task, you will create a dataframe from an array of sample rainfall data values, and then quantize them into categorical value ranges.

1. In the new notebook, in the empty cell, enter the following code to create a set of data that represents rainfall measurements:

Python

```
from pyspark.sql.types import *
from pyspark.ml.feature import Bucketizer
splits = [-float("inf"), 8.0, 12.0, 15.0, float("inf")]
temp = [(1, 10.2), (2, 17.1), (3, 9.6), (4, 5.0), (5, 3.4)]
rainfall = sc.parallelize(temp)
```

Scala

```
import org.apache.spark.ml.feature.Bucketizer
import org.apache.spark.sql.DataFrame
val splits = Array(Double.NegativeInfinity, 8.0, 12.0, 15.0,
Double.PositiveInfinity)
val temp = Array((1, 10.2), (2, 17.1), (3, 9.6), (4, 5.0), (5, 3.4))
val rainfall = sc.parallelize(temp)
```

2. Click the **run cell, select below** button and wait for the code to finish executing (it can take a short time to execute initially while the Spark context is created). Then view the output, which should confirm the creation of an RDD named **rainfall**.
3. In the new empty cell, enter the following code to load the rainfall RDD into a dataframe:

Python

```
fields = [StructField("id", IntegerType(), True),
StructField("rainfall", DoubleType(), True)]
schema = StructType(fields)
```

```
df = sqlContext.createDataFrame(rainfall, schema)
```

Scala

```
val df = sqlContext.createDataFrame(rainfall).toDF("id", "rainfall")
```

4. Click the **run cell, select below** button to run the contents of the cell.
5. In the new cell, enter the following code to create a **Bucketizer** class using the **splits** array that you created earlier:

Python

```
bucketizer = Bucketizer(splits=splits, inputCol="rainfall",
outputCol="discrainfall")
```

Scala

```
val bucketizer = new Bucketizer()
bucketizer.setInputCol("rainfall")
bucketizer.setOutputCol("discrainfall")
bucketizer.setSplits(splits)
```

6. Click the **run cell, select below** button to run the contents of the cell.
7. In the new cell below enter the following code to transform the dataframe rainfall values into bucketed values and then show the results in a table:

Python

```
bucketedData = bucketizer.transform(df)
bucketedData.show()
```

Scala

```
val bucketedData = bucketizer.transform(df)
bucketedData.show()
```

8. Click the **run cell, select below** button, and view the discretized rainfall values that indicate the bins into which the original rainfall values have been assigned.

Normalize Data

In his task, you will create a set of vectors that contain widely disparate values, and then normalize those values.

1. In the empty cell at the bottom of the notebook, enter the following code to create a set of vectors and load them into an RDD:

Python

```
from pyspark.mllib.feature import Normalizer
from pyspark.mllib.linalg import Vectors

data = [[1, 34.0, 587.0], [5, 76.0, 1005.0], [3, 22.0, 867.0], [5, 19.0, 475.0], [2, 22.0, 666.0]]
input = sc.parallelize(data)
input.take(5)
```

Scala

```
import org.apache.spark.mllib.feature.Normalizer
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.mllib.linalg.Vectors

val data = Seq(Vectors.dense(1, 34D, 587D), Vectors.dense(5, 76D, 1005D), Vectors.dense(3, 22D, 867D), Vectors.dense(5, 19D, 475D), Vectors.dense(2, 22D, 666D))
val input = sc.parallelize(data)
input.take(5)
```

2. Click the **run cell, select below** button to run the contents of the cell, and note that the numeric values in each vector are significantly different in scale from one another.
3. Enter the following code to create a normalizer and transform the RDD:

Python

```
normalizer = Normalizer()
normalized = normalizer.transform(input)
normalized.take(5)
```

Scala

```
val normalizer = new Normalizer()
val normalized = normalizer.transform(input)
normalized.take(5)
```

4. Click the **run cell, select below** button to run the contents of the cell. It should be evident that the scale has changed and all of the values are now between zero and one.
5. On the **File** menu, click **Close and Halt**.

Using PCA to Reduce Features

When a feature vector contains multiple features, it can be useful to reduce the number of features using a technique called *principal component analysis*, or *PCA*.

Tip: You can copy and paste the code for this exercise from **Feature Reduction-Python.txt** or **Feature Reduction-Scala.txt** in the lab folder for this lab.

Create a New Notebook

You can use Scala code in a Jupyter notebook to preprocess data.

1. In Jupyter, in the **Labs** folder, in the **New** drop-down list, click **Spark** if you prefer to program in Scala, or **PySpark** if you prefer to program in Python. This creates a new notebook named **Untitled**.

2. At the top of the new notebook, click **Untitled**, and rename notebook to **Feature Reduction**.

Use PCA to Reduce Features

In his task, you will return to the iris dataset and use PCA to reduce the number of features in the feature vectors.

1. In the new notebook, in the empty cell, enter the following code to load the iris-multiclass.csv file into an RDD, and then create an RDD containing arrays that contain four features:

Python

```
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.feature import PCA
input = sc.textFile("wasb:///iris-multiclass.csv")
splitter = input.map(lambda line: line.split(','))
vec = splitter.map(lambda line: Vectors.dense(float(line[0]),
float(line[1]), float(line[2]), float(line[3])))
vec.take(5)
```

Scala

```
import org.apache.spark.ml.feature.PCA
import org.apache.spark.mllib.linalg.Vectors
import sqlContext.implicits._
val input = sc.textFile("wasb:///iris-multiclass.csv")
val vec = input.map(line => line.split(",")).map(mapped =>
Vectors.dense(Array(mapped(0).toDouble, mapped(1).toDouble,
mapped(2).toDouble, mapped(3).toDouble)))
vec.take(5)
```

2. Click the **run cell, select below** button and wait for the code to finish executing (it can take a short time to execute initially while the Spark context is created). Note that each array in the RDD contains four values.
3. In the cell below enter the following code to use PCA to reduce the 4 dimensions from the Iris featureset to 2:

Python

```
model = PCA(2).fit(vec)
```

Scala

```
import sqlContext.implicits._
val data = vec.collect()
val df =
sqlContext.createDataFrame(data.map(Tuple1.apply)).toDF("features")
val pca = new
PCA().setInputCol("features").setOutputCol("outfeatures").setK(2)
val pcsModel = pca.fit(df)
```

4. Click the **run cell, select below** button to run the contents of the cell.
5. Enter the following code to use the model you've just created to transform the features and view them – you should be able to see two features per row instead of four:

Python

```
transformed = model.transform(vec)
transformed.take(5)
```

Scala

```
val pcaDF = pcsModel.transform(df)
```

```
val result = pcaDF.select("outfeatures")
result.take(5)
```

6. Click the **run cell, select below** button to run the contents of the cell.
7. If you are using Scala, enter and run the following code in the next cell to see how to do the same transformation using a **RowMatrix**:

Scala

```
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val matrix = new RowMatrix(vec)
val comps = matrix.computePrincipalComponents(2)
```

8. On the **File** menu, click **Close and Halt**.

Clean Up

If you intend to continue to the next lab immediately, you can leave your cluster running and use it to complete the next lab. Otherwise, you should delete your cluster and the associated storage account. This ensures that you avoid being charged for cluster resources when you are not using them. If you are using a trial Azure subscription that includes a limited free credit value, deleting the cluster maximizes your credit and helps to prevent using it all before the free trial period has ended.

Note: If you intend to proceed immediately to the next lab in this course, you should skip this procedure and use your existing cluster in the next lab. However, if you plan to wait before starting the next lab, you should delete your cluster to avoid incurring charges unnecessarily.

Delete the Resource Group for your HDInsight Cluster

1. If it is not already open in a tab in your web browser, browse to the new Azure portal at <https://portal.azure.com>.
2. In the Azure portal, view your **Resource groups** and select the resource group you created for your cluster. This resource group contains your cluster and the associated storage account.
3. In the blade for your resource group, click **Delete**. When prompted to confirm the deletion, enter the resource group name and click **Delete**.
4. Wait for a notification that your resource group has been deleted.