Implementing Predictive Analytics with Hadoop in Azure HDInsight

Lab 2 – Building Machine Learning Models

Overview

In this lab, you will use Spark to create machine learning models.

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

If you already have an HDInsight Spark cluster running from the previous lab, you can skip this exercise. Otherwise, follow the instructions below to provision a 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

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, in the Hub Menu, click **New**. Then in the **Data + Analytics** menu, click **HDInsight**.
3. 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
    - **Cluster Operating System**: Linux
    - **Cluster Tier**: Standard
- **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*
- **Node Pricing Tiers:**
    - **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*

4. 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. Go and get some coffee!)

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.

View the HDInsight Cluster in the Azure Portal

1. In the Azure portal, browse to the Spark cluster you just created.
2. In the blade for your cluster, under **Quick Links**, click **Cluster Dashboards**.
3. In the **Cluster Dashboards** blade, note the dashboards that are available. These include a Jupyter Notebook that you will use later in this course.

Creating a Linear Regression Model

In this exercise, you will create a linear regression machine learning model that predicts the fuel efficiency of a car based on its features.

Upload Source Data to Azure Storage

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. Note the resource group and storage account name
7. Enter a `dir` (Windows) or `ls` (Mac OS X or Linux) command to view the contents of the **Lab02** folder where you extracted the lab files for this course (for example, `dir c:\HDPredictLabs\Lab02` or `ls HDPredictLabs/Lab02`), and verify that this folder contains a file named **autos.csv**. This file contains 358 observations of different cars, their attributes and the miles per gallon that they achieve.
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, 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 the following command on a single line to upload the autos.csv file to a blob named **autos.csv** in the container used by your HDInsight cluster. Replace ***autos_path*** with the local path to autos.csv (for example c:\HDPredictLabs\Lab02\autos.csv or HDPredictLabs/Lab02/autos.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 autos_path container autos.csv
    ```

12. Wait for the files to be uploaded.
13. Keep the command window open – you will use the Azure CLI again in a later exercise.

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, view the existing folders.
4. If you have not previously created a folder named **Labs** in this cluster, in the **New** drop-down menu, click **Folder**. This create a folder named **Untitled Folder**. Then select the checkbox for the **Untitled Folder** folder, and above the list of folders, click **Rename**. Then rename the directory to **Labs**.
5. Open the **Labs** folder, and verify that it contains no notebooks.
6. 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 create a new notebook named **Untitled**.
7. At the top of the new notebook, click **Untitled**, and rename notebook to **Linear Regression for Autos**.

Load the Autos Data into an RDD

1. In the empty cell at the top of the notebook, enter the following Scala or Python code to import the necessary namespaces:

   ### Scala
   ```
   import org.apache.spark.mllib.regression.LabeledPoint
   import org.apache.spark.mllib.regression.LinearRegressionModel
   import org.apache.spark.mllib.regression.LinearRegressionWithSGD
   import org.apache.spark.mllib.linalg.Vectors
   import org.apache.spark.mllib.feature.StandardScaler
   ```

   ### Python
   ```
   from pyspark.mllib.regression import LabeledPoint
   from pyspark.mllib.regression import LinearRegressionModel
   from pyspark.mllib.regression import LinearRegressionWithSGD
   from pyspark.mllib.linalg import Vectors
   from pyspark.mllib.feature import StandardScaler
   ```

2. With cursor still in the cell, on the toolbar click the **run cell, select below** button. It can take a short time to execute while the Spark context is created. As the code runs the ○ symbol next to **Spark** 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, in the new cell under the output from the first cell, enter the following code to read the **autos.csv** text file from Azure storage into an RDD named **input** and then turn the comma-separated line into a **LabeledPoint** which comprises of a label and set of features. This will then be persisted across the Spark cluster:

   ### Scala
   ```
   val input = sc.textFile("wasb:///autos.csv")

   val featurevector  = input.map { line =>
       val lineSplit = line.split(',')
       val featureArr = lineSplit.slice(1, 8).map(_.toDouble)
       val values = Vectors.dense(featureArr)
       LabeledPoint(lineSplit(0).toDouble, values)
   }.persist()
   ```

Python

```
def parsePoint(line):
    values = [float(x) for x in line.split(',')[0:7]]
    return LabeledPoint(values[0], values[1:])

input = sc.textFile("wasb:///autos.csv")

featurevector  = input.map(lambda line : parsePoint(line)).persist()
```

4. With cursor still in the second cell, on the toolbar click the **run cell, select below** button. When the cell contents have finished executing the `[*]` will be replace by a `[2]`

Scale the Data

1. In the new cell under the second cell, enter the following code to scale the features so that they use a similar independent scale.

   Scala

   ```
   val scaler = new StandardScaler().fit(featurevector.map(x =>
   x.features))
   val scaledData = featurevector.map(x =>
   LabeledPoint(x.label,scaler.transform(Vectors.dense(x.features.toArray)
   )))
   ```

   Python

   ```
   labels = featurevector.map(lambda x: x.label)
   features = featurevector.map(lambda x: x.features)
   scaler = StandardScaler().fit(features)
   scaledData = scaler.transform(features)
   newfeatures = labels.zip(scaledData)
   allset = newfeatures.map(lambda line: LabeledPoint(line[0], line[1]))
   ```

2. Click the **run cell, select below** button to run the third cell.

Split the Data into Training and Test Data

1. In the fourth cell that appears at the bottom of the notebook, enter the following which will split the data into training and test data in the ratio of 70% to 30%:

   Scala

   ```
   val allData = scaledData.randomSplit(Array(0.7, 0.3), seed = 11L)
   val (training, test) = (allData(0), allData(1))
   ```

   Python

   ```
   training, test = allset.randomSplit(weights=[0.7, 0.3], seed=1)
   ```

2. Click the **run cell, select below** button to run the fourth cell.

Prepare and Train a Linear Regression Model

1. In the fifth cell enter the following code which will prepare the Linear Regression model:

   Scala

   ```
   val numIterations = 100
   val stepSize = 0.001
   val algorithm = new LinearRegressionWithSGD()
   algorithm.setIntercept(true)
   ```

```
algorithm.optimizer.setNumIterations(numIterations)
algorithm.optimizer.setStepSize(stepSize)
```

Python

```
numIterations = 100
stepSize = 0.001
```

2. Press **CTRL+ENTER** to run the fifth cell.
3. In the next cell that appear below enter the following code which will train the model with the training data we created earlier:

Scala

```
val model = algorithm.run(training)
```

Python

```
algorithm = LinearRegressionWithSGD.train(training,
iterations=numIterations, step=stepSize, intercept=True)
```

4. Click the **run cell, select below** button to run the contents of this cell.

Test the Model

1. In the new cell enter the following code to create a set of predictions over the miles per gallon of each vehicle. The code creates a tuple of Labels against predicted Labels and then displays the first twenty of them.

Scala

```
val valuesAndPreds = test.map { point =>
  val prediction = model.predict(point.features)
  (point.label, prediction)
}
valuesAndPreds.take(20)
```

Python

```
valuesAndPreds = test.map(lambda p: (p.label,
algorithm.predict(p.features)))

valuesAndPreds.take(20)
```

2. Click the **run cell, select below** button to run the contents of this cell.
3. You should be able to see an output below the Cell of twenty Labels and predicted Labels which match fairly closely to each other.
4. In the new cell below add the following code to calculate the Mean Squared Error (MSE) of the model:

Scala

```
val MSE = valuesAndPreds.map{case(v, p) => math.pow((v - p), 2)}.mean()
println("training Mean Squared Error = " + MSE)
```

Python

```
MSE = valuesAndPreds.map(lambda (v, p): (v - p)**2).reduce(lambda x, y:
x + y) / valuesAndPreds.count()

print("Mean Squared Error = " + str(MSE))
```

5. Click the **run cell, select below** button to run the contents of this cell and view the MSE.

6. On the toolbar, click the **Save** button.
7. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.
8. Verify that the **Linear Regression for Autos** notebook is listed in the **Labs** folder.

Creating a Pipeline

In this exercise, you will implement a pipeline to create and cross validate a model that performs sentiment analysis on tweets.

Upload the Source Data

1. Return to the command prompt window in which you previously used the Azure CLI to upload data to Azure storage.
2. Enter a `dir` (Windows) or `ls` (Mac OS X or Linux) command to view the contents of the **Lab02** folder where you extracted the lab files for this course, and verify that this folder contains files named **training-tweets.csv** and **test-tweets.csv**. These files contain tweets on which you will perform sentiment analysis.
3. Enter the following command on a single line to upload the training-tweets.csv file to a blob named **training-tweets.csv** in the container used by your HDInsight cluster. Replace ***tweets_path*** with the local path to training-tweets.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 tweets_path container training-tweets.csv
   ```

4. Enter a second `azure storage blob upload` command to upload test-tweets.csv to a blob named **test-tweets.csv**.

   ```
   azure storage blob upload tweets_path container test-tweets.csv
   ```

Create a New Notebook

1. Switch to the Jupyter tab in your web browser, and 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 create a new notebook named **Untitled**.
2. At the top of the new notebook, click **Untitled**, and rename notebook to **Sentiment Analysis of Tweets**.

Implement a Pipeline that Creates and Cross-Validates a Model

1. In the empty cell at the top of the notebook, enter the following code. This code imports the necessary namespaces:

   #### Scala

   ```
   import org.apache.spark.ml.Pipeline
   import org.apache.spark.ml.classification.LogisticRegression
   import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
   import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
   import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}
   import org.apache.spark.mllib.linalg.Vector
   import org.apache.spark.sql.Row
   ```

   #### Python

   ```
   from pyspark.ml import Pipeline
   from pyspark.ml.classification import LogisticRegression
   from pyspark.ml.evaluation import BinaryClassificationEvaluator
   ```

```
from pyspark.ml.feature import HashingTF, Tokenizer
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.mllib.linalg import Vector
from pyspark.sql import Row
from pyspark.sql.types import *
import numpy as np
```

2. With cursor still in the cell, on the toolbar click the **run cell, select below** button.
3. When the code has finished running, and the Spark context has been created, in the new cell under the output from the first cell, enter the following code to read the **training-tweets.csv** and **test-tweets.csv** text file from Azure storage into RDD's labeled training and test. These are then converted into Spark DataFrames so that we can use them with ML Pipelines:

Scala

```
case class Tweets(id: Int, label: Double, source: String, text: String)
val training = sc.textFile("wasb:///training-
tweets.csv").zipWithIndex().filter(_._2 > 0).map(line =>
line._1.split(",")).map(tw => Tweets(tw(0).toInt, tw(1).toDouble,
tw(2), tw(3))).toDF()
val test = sc.textFile("wasb:///test-
tweets.csv").zipWithIndex().filter(_._2 > 0).map(line =>
line._1.split(",")).map(tw => Tweets(tw(0).toInt, tw(1).toDouble,
tw(2), tw(3))).toDF()
```

Python

```
trainingrdd = sc.textFile("wasb:///training-
tweets.csv").zipWithIndex().filter(lambda line: line[1] > 0).map(lambda
line: line[0].split(","))
testrdd = sc.textFile("wasb:///test-
tweets.csv").zipWithIndex().filter(lambda line: line[1] > 0).map(lambda
line: line[0].split(","))

fields = [StructField("id", StringType(), True), StructField("label",
StringType(), True), StructField("source", StringType(), True),
StructField("text", StringType(), True)]
schema = StructType(fields)

training = sqlContext.createDataFrame(trainingrdd, schema)
test = sqlContext.createDataFrame(testrdd, schema)
```

4. With cursor still in the second cell, on the toolbar click the **run cell, select below** button and wait for the code to finish running.
5. In the new empty cell, enter the following code to create an ML Pipeline. The code takes the tweet text from the input column "text" and outputs as a token to a new column called "words" using this as input to a "Hashing Term Frequency" (**HashingTF**) evaluator which outputs the hashed terms as a set of features. Logistic Regression will then be applied to calculate the sentiment of the tweet(s). All of these estimators are then set as stages in a single **Pipeline**:

Scala

```
val tokenizer = new
Tokenizer().setInputCol("text").setOutputCol("words")
val hashingTF = new
HashingTF().setInputCol(tokenizer.getOutputCol).setOutputCol("features"
)
```

```
val lr = new LogisticRegression().setMaxIter(10)
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF,
lr))
```

Python

```
training = training.withColumn("label",
training.label.cast(DoubleType()))
test = training.withColumn("label", training.label.cast(DoubleType()))
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

6. Click the **run cell, select below** button to run the third cell.
7. In the fourth cell that appears below, enter the following which will build a set of parameters for our model. The **HashingTF** evaluator needs a determination of the number of features so you will be parameterizing this using 3 different values and two different values for the **LogisiticRegresion**. This generates a grid of 3x2 grid of permutations that will be used with the cross validation we will perform in the next steps:

   Scala

   ```
   val paramGrid = new ParamGridBuilder()
   paramGrid.addGrid(hashingTF.numFeatures, Array(10, 20, 100))
   paramGrid.addGrid(lr.regParam, Array(0.1, 0.01))
   val grid = paramGrid.build()
   ```

   Python

   ```
   paramGrid = ParamGridBuilder()
   paramGrid.addGrid(hashingTF.numFeatures, [10, 20, 100])
   paramGrid.addGrid(lr.regParam, [0.1, 0.01])
   grid = paramGrid.build()
   ```

8. Click the **run cell, select below** button to run the fourth cell. You should now see all of the different permutations in the output that the ParamGridBuilder has made and will apply to the CrossValidator that will be created.
9. In the fifth cell enter the following code which will be build a **CrossValidator** to split the data based on a number of "k" folds. We set the **Pipeline Estimator** followed by an **Evaluator**, which in this case is **BinaryClassificationMetrics** as this is used to evaluate your **LogisticRegression** algorithm. Then we apply the ParamGrid we built in the previous cell to ensure that we cover the parameter permutation across the cross validation:

   Scala

   ```
   val crossvalidator = new CrossValidator()
   crossvalidator.setEstimator(pipeline)
   crossvalidator.setEvaluator(new BinaryClassificationEvaluator)
   crossvalidator.setEstimatorParamMaps(grid)
   crossvalidator.setNumFolds(6)
   ```

   Python

   ```
   bce = BinaryClassificationEvaluator()
   crossvalidator = CrossValidator(estimator=pipeline,
   estimatorParamMaps=grid,evaluator=BinaryClassificationEvaluator(),numFo
   lds=6)
   ```

10. Click the **run cell, select below** button to run the fifth cell.

11. In the next cell that appears below, enter the following code which will train the model with the training data we created earlier:

    Scala
    ```
    val model = crossvalidator.fit(training)
    ```

    Python
    ```
    model = crossvalidator.fit(training)
    ```

12. Click the **run cell, select below** button to run the contents of this cell. This can take a while to run.
13. In the new cell enter the following code to transform the test data and retrieve the probability and predictions of each label, while calculating the percentage accuracy of the model:

    Scala
    ```
    val modelem = model.transform(test).select("id", "label", "text",
    "probability", "prediction")
    modelem.collect().foreach { case Row(id: Int, label : Double, text:
    String, probability: Vector, prediction: Double) =>
        println(s"($id, $text) --> prob=$probability,
    prediction=$prediction")
      }
     (modelem.filter("label = prediction").count().toDouble /
    modelem.count().toDouble) * 100D
    ```

    Python
    ```
    prediction = model.transform(test)
    selected = prediction.select("id", "label", "text", "probability",
    "prediction")
    for row in selected.collect():
        print(row)

     (float(selected.filter("label = prediction").count()) /
    float(selected.count())) * 100.0
    ```

14. Click the **run cell, select below** button to run the contents of this cell.
15. In addition to this you'll calculate the Area Under the ROC Curve with the following code, so that we can see this metric via the Evaluator for each of the six params you're using:

    Scala
    ```
    model.avgMetrics
    ```

    Python
    ```
    crossvalidator.getEvaluator().getMetricName()
    ```

16. Click the **run cell, select below** button to run the contents of this cell.

Note: The remainder of this lab uses Spark classes that are not yet supported in Python. You can only complete the rest of the procedures using Scala.

Add a Training Split to the Pipeline

1. In the empty cell at the bottom of the notebook, enter the following code to import an additional namespace:

Scala

```
import org.apache.spark.ml.tuning.TrainValidationSplit
```

2. Click the **run cell, select below** button to run the contents of this cell.
3. In the new empty cell, enter the following code to import the tweet data and split 90/10 for training/test data:

Scala

```
case class Tweets(id: Int, label: Double, source: String, text: String)

val tweets = sc.textFile("wasb:///training-
tweets.csv").zipWithIndex().filter(_._2 > 0).map(line =>
line._1.split(",")).map(tw => Tweets(tw(0).toInt, tw(1).toDouble,
tw(2), tw(3))).toDF()

val Array(training, test) = tweets.randomSplit(Array(0.9, 0.1), seed =
11L)
```

4. Click the **run cell, select below** button to run the contents of this cell.
5. In the next cell enter the following code to create a `TrainValidationSplit()` in order to split redefine the split of dataset:

Scala

```
val trainValidationSplit = new TrainValidationSplit()
trainValidationSplit.setEstimator(pipeline)
trainValidationSplit.setEvaluator(new BinaryClassificationEvaluator)
trainValidationSplit.setEstimatorParamMaps(grid)
trainValidationSplit.setTrainRatio(0.8)
```

6. Click the **run cell, select below** button to run the contents of this cell.
7. In the next cell enter the following code to train the new model:

Scala

```
val model = trainValidationSplit.fit(training)
```

Python

```
Not introduced into Spark yet
```

8. Click the **run cell, select below** button to run the contents of this cell.
9. In the next cell enter the following code to see the results of the model for the first twenty rows:

Scala

```
model.transform(test).select("features", "label", "prediction").show()
```

10. Press **CTRL+ENTER** to run the contents of this cell.
11. In the next cell enter the following code to see the Area under the ROC Curve for the model based on the predefined Parameter Grid:

Scala

```
model.validationMetrics
```

12. On the toolbar, click the **Save** button.
13. On the **File** menu click **Close and Halt**. If prompted, confirm that you want to close the tab.

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.

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.