Lab 3: Data Science Using Azure Databricks

Technologies showcased: SparkR Notebook in Azure Databricks

Author: Sandeep Karunakaran

Date: 19/6/2018

Table of Contents

[Summary 3](#_Toc517184657)

[Pre-requisites 3](#_Toc517184658)

[Scenario 4](#_Toc517184659)

[Part 1 – Create A Databricks Workspace 4](#_Toc517184660)

[Part 2 – Create a Spark cluster in Databricks 7](#_Toc517184661)

[Part 3 – Creating SparkR DataFrames 8](#_Toc517184662)

[Part 4 – DataFrame Operations 10](#_Toc517184663)

[Part 5 – Machine Learning - Training a Linear Regression model using glm() 11](#_Toc517184664)

[Part 6 – Machine Learning - Training a Logistic Regression model using glm() 13](#_Toc517184665)

[Part 7 – Clean up resources 14](#_Toc517184666)

## Summary

In this lab you will learn the basics of SparkR Dataframes and operations and perform a Machine Learning exercise of producing and training a Linear Regression and Logistic Regression Model using SparkR notebook and libraries within Azure Databricks.

.

Steps -

* Create an Azure Databricks workspace – IGNORE if already completed in Lab 2
* Create a Spark cluster in Azure Databricks – IGNORE if already completed in Lab 2
* Creating SparkR Dataframes and Operations
* Train and Produce a Linear Regression Model using glm()
* Train and Produce a Logistic Regression Model using glm()

## Pre-requisites

* Azure Subscription with rights to use/deploy Azure services, and X of Azure credit
* Prereqs complete from lab 1
* Web browser (Edge/Chrome recommended)

## 

## Scenario

|  |  |  |
| --- | --- | --- |
| Part 1 – Create A Databricks Workspace | | |
| **Scenario** | | |
| First, we are going to create a Databricks Workspace, which allows you create and access Databricks resources | | |
| **Commentary / Notes** | **Click Steps & ‘Bits’** | **Screenshots** |
|  | In the Azure portal, select **Create a resource** > **Data + Analytics** > **Azure Databricks**. |  |
|  | Under **Azure Databricks Service**, provide the values to create a Databricks workspace. |  |
| The PowerShell file has descriptions on what each configuration variable represents. | Provide the following rules –   | Property | Description | | --- | --- | | **Workspace name** | Provide a name for your Databricks  workspace | | **Subscription** | From the drop-down, select your Azure  subscription. | | **Resource group** | Specify whether you want to create a new resource group or use an existing one. A resource group is a container that holds related resources for an Azure solution. For more information, see [Azure Resource Group overview](https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview). | | **Location** | Select **West Europe**. | | **Pricing Tier** | Choose **Standard** |   Select **Pin to dashboard** and then select **Create**.  The account creation takes a few minutes. During account creation, the portal displays the **Submitting deployment for Azure Databricks** tile on the right side. You may need to scroll right on your dashboard to see the tile. There is also a progress bar displayed near the top of the screen. You can watch either area for progress. |  |

|  |  |  |
| --- | --- | --- |
| Part 2 – Create a Spark cluster in Databricks | | |
| **Scenario** | | |
|  | | |
| **Commentary / Notes** | **Click Steps & ‘Bits’** | **Screenshots** |
|  | In the Azure portal, go to the Databricks workspace that you created, and then select Launch Workspace.  You are redirected to the Azure Databricks portal. From the portal, select Cluster. |  |
|  | In the **New cluster** page, provide the values to create a cluster. |  |
|  | Accept all other default values other than the following:   * Enter a name for the cluster. * For this article, create a cluster with **4.0** runtime. * Make sure you select the **Terminate after *\_\_* minutes of inactivity** checkbox. Provide a duration (in minutes) to terminate the cluster, if the cluster is not being used.   Select **Create cluster**. Once the cluster is running, you can attach notebooks to the cluster and run Spark jobs. |  |

|  |
| --- |
| Part 3 – Creating SparkR DataFrames |

|  |  |
| --- | --- |
| **From a local R data.frame**  The simplest way to create a DataFrame is to convert a local R data.frame into a SparkDataFrame.  Specifically we can use createDataFrame and pass in the local R data.frame to create a SparkDataFrame. Like most other SparkR functions, createDataFrame syntax changed in Spark 2.0. You can see examples of this in the code snippet bellow.  Refer <http://spark.apache.org/docs/latest/api/R/createDataFrame.html> for more examples | **Copy**  *library(SparkR)*  *df <- createDataFrame(faithful)*  *# Displays the content of the DataFrame to stdout*  *head(df)* |
| **Using the data source API**  The general method for creating a DataFrame from a data source is read.df. This method takes the path for the file to load and the type of data source. SparkR supports reading CSV, JSON, text, and Parquet files natively. Through Spark Packages you can find data source connectors for popular file formats such as Avro.  SparkR automatically infers the schema from the CSV file. | Copy  *library(SparkR)*  *diamondsDF <- read.df("/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv",*  *source = "csv", header="true", inferSchema = "true")*  *head(diamondsDF)*  *Copy*  *printSchema(diamondsDF)*  *Copy*  *display(diamondsDF)* |
| **Adding a data source connector with Spark Packages**  As an example, we will use the spark-avro package to load an Avro file. The availability of the spark-avro package depends on your cluster’s image version. See Avro Files  First we take an existing data.frame, convert to a Spark DataFrame, and save it as an Avro file. | *require(SparkR)*  *irisDF <- createDataFrame(iris)*  *write.df(irisDF, source = "com.databricks.spark.avro", path = "dbfs:/tmp/iris.avro", mode = "overwrite")* |
| Now we use the spark-avro package again to read back the data. | *irisDF2 <- read.df(path = "/tmp/iris.avro", source = "com.databricks.spark.avro")*  *head(irisDF2)* |
| The data sources API can also be used to save DataFrames into multiple file formats. For example we can save the DataFrame from the previous example to a Parquet file using write.df | *write.df(irisDF2, path="dbfs:/tmp/iris.parquet", source="parquet", mode="overwrite")* |
| To verify that we saved a parquet file: | *%fs ls dbfs:/tmp/iris.parquet* |

|  |
| --- |
| Part 4 – DataFrame Operations |

|  |  |
| --- | --- |
| SparkDataFrames support a number of functions to do structured data processing. Here we include some basic examples and a complete list can be found in the API docs:  <https://spark.apache.org/docs/latest/api/R/> | *# Import SparkR package if this is a new notebook*  *require(SparkR)*  *# Create DataFrame*  *df <- createDataFrame(faithful)*  *df* |
| Selection of columns from dataframe | *# Select only the "eruptions" column*  *head(select(df, df$eruptions))*  *# You can also pass in column name as strings*  *head(select(df, "eruptions"))* |
| Filtering data from dataframe | *# Filter the DataFrame to only retain rows with wait times shorter than 50 mins*  *head(filter(df, df$waiting < 50))* |
| **Grouping and aggregation**  SparkDataFrames support a number of commonly used functions to aggregate data after grouping. For example we can count the number of times each waiting time appears in the faithful dataset. | *head(count(groupBy(df, df$waiting)))* |
| Aggregation and Sorting | *# We can also sort the output from the aggregation to get the most common waiting times*  *waiting\_counts <- count(groupBy(df, df$waiting))*  *head(arrange(waiting\_counts, desc(waiting\_counts$count)))* |
| **Column operations**  SparkR provides a number of functions that can be directly applied to columns for data processing and aggregation. The example below shows the use of basic arithmetic functions. | *# Convert waiting time from hours to seconds.*  *#Note that we can assign this to a new column in the same DataFrame*  *df$waiting\_secs <- df$waiting \* 6*  *head(df)* |

|  |
| --- |
| Part 5 – Machine Learning - Training a Linear Regression model using glm() |

|  |  |
| --- | --- |
| We will try to predict a diamond’s price from its features. We will do this by training a Linear Regression model using the training data. |  |
| Data extraction and reading | *# Read diamonds.csv dataset as SparkDataFrame*  *diamonds <- read.df("/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv",*  *source = "com.databricks.spark.csv", header="true", inferSchema = "true")*  *diamonds <- withColumnRenamed(diamonds, "", "rowID")* |
| Split the data for training and testing | *# Split data into Training set and Test set*  *trainingData <- sample(diamonds, FALSE, 0.7)*  *testData <- except(diamonds, trainingData)* |
| Excluding unwanted columns | *# Exclude rowIDs*  *trainingData <- trainingData[, -1]*  *testData <- testData[, -1]* |
| Print the counts of each dataset and view the contents of the training dataframe | *print(count(diamonds))*  *print(count(trainingData))*  *print(count(testData))*  *head(trainingData)* |
| Note that we have a mix of categorical features (for eg: cut - Ideal, Premium, Very Good...) and continuous features (for eg: depth, carat). Under the hood, SparkR automatically performs one-hot encoding of such features so that it does not have to be done manually.  Using glm  glm fits a Generalized Linear Model, similar to R’s glm().  Note: If you are planning to use a string column as your label, ensure that you are running Spark 1.6+ or glm might throw an error.  Syntax: - glm(formula, data, family...)  Parameters: - formula: Symbolic description of model to be fitted, for eg: ResponseVariable ~ Predictor1 + Predictor2. Supported operators: ‘~’, ‘+’, ‘-‘, and ‘.’ - data: Any SparkDataFrame - family: String, “gaussian” for Linear Regression, or “binomial” for Logistic Regression - lambda: Numeric, Regularization parameter - alpha: Numeric, Elastic-net mixing parameter  Output: - MLlib PipelineModel | *# Indicate family = "gaussian" to train a linear regression model*  *lrModel <- glm(price ~ ., data = trainingData, family = "gaussian")* |
| Print a summary of trained linear regression mode | *# Print a summary of trained linear regression model*  *summary(lrModel)* |
| We will use predict() on the test data to see how well our model works on new data.  Syntax for predict(): - predict(model, newData)  Parameters: - model: MLlib model - newData: SparkDataFrame, typically your test set  Output: - SparkDataFrame  Generate Predictions using the trained Linear Regression Model  View the Predictions against the price column | *# Generate predictions using the trained Linear Regression model*  *predictions <- predict(lrModel, newData = testData)*  *# View predictions against price column*  *display(select(predictions, "price", "prediction"))* |
| Evaluation of the Model | *errors <- select(predictions, predictions$price, predictions$prediction, alias(predictions$price - predictions$prediction, "error"))*  *display(errors)* |

|  |
| --- |
| Part 6 – Machine Learning - Training a Logistic Regression model using glm() |

|  |  |
| --- | --- |
| We can create a Logistic Regression on the same dataset. Let’s see if we can predict a diamond’s cut based on some of its features.  As of Spark 1.6, Logistic Regression in MLlib only supports binary classification. To test out the algorithm with our dataset in this example, we will subset our data such that we are able to work with only 2 labels. |  |
| Split Training and Test Data into separate dataframes. | *# Subset data to include rows where diamond cut = "Premium" or diamond cut = "Very Good"*  *trainingDataSub <- subset(trainingData, trainingData$cut %in% c("Premium", "Very Good"))*  *testDataSub <- subset(testData, testData$cut %in% c("Premium", "Very Good"))* |
| Set the family to Binomial for Logistic Regression  Print the Summary of the Model. | *# Indicate family = "binomial" to train a logistic regression model*  *logrModel <- glm(cut ~ price + color + clarity + depth, data = trainingDataSub, family = "binomial")*  *# Print summary of Logistic Regression model*  *# Note: This only works in Spark 1.6+*  *summary(logrModel)* |
| Predict from the trained model using the test data frame  View predictions against the “label” column. | *# Generate predictions using the trained Linear Regression model*  *predictionsLogR <- predict(logrModel, newData = testDataSub)*  *# View predictions against label column*  *display(select(predictionsLogR, "label", "prediction"))* |
| Evaluate the Model. | *# Evaluate Logistic Regression model*  *errorsLogR <- select(predictionsLogR, predictionsLogR$label, predictionsLogR$prediction, alias(abs(predictionsLogR$label - predictionsLogR$prediction), "error"))*  *display(errorsLogR)* |

|  |  |  |
| --- | --- | --- |
| Part 7 – Clean up resources | | |
| **Scenario** | | |
| After you have finished running the tutorial, you can terminate the cluster. | | |
| **Commentary / Notes** | **Click Steps & ‘Bits’** | **Screenshots** |
|  | From the Azure Databricks workspace, from the left pane, select **Clusters**. For the cluster you want to terminate, move the cursor over the ellipsis under **Actions** column, and select the **Terminate** icon. |  |
|  | If you do not manually terminate the cluster it will automatically stop, provided you selected the **Terminate after \_\_ minutes of inactivity** checkbox while creating the cluster. In such a case, the cluster automatically stops if it has been inactive for the specified time. |  |

**END OF LAB 3**

**IMPORTANT: AVOID INCURRING EXTRA CHARGES BY PAUSING YOUR SUBSCRIPTION RESOURCES**