**Tecnológico Nacional de México**
**Instituto Tecnológico de Tijuana**

**Subdirección Académica**
**Departamento de Sistemas y Computación**

**Semestre:**
Agosto – Diciembre 2021

**Carrera:**
Ingeniería en Tecnologías de la Información y Comunicaciones

**Materia y serie:**
Datos Masivos BDD-1704TI9A

**Unidad a evaluar:** Unidad II

**Nombre de la Tarea:**
Práctica Evaluatoria - Unidad 2

**Nombre del Alumno:**
Flores Gonzalez Luis Diego C16211486
Sifuentes Martinez Manuel Javier 17212934

**Nombre del docente:**
José Christian Romero Hernández

**Develop the following instructions in Spark with the Scala programming language , using only the documentation from the Spark and Google Machine Learning Mllib library.**

In the evaluation of Big Data Unit 2, different actions and operations will be carried out with a provided Dataset (iris). The Spark syntax will be used in order to achieve all the expected results.

**1. Load in a dataframe Iris.csv**

The CSV to be used is imported, specifying through the option function that header is true, this means that the first line is identified as the "iris" of each column and is saved in the constant df.

```
val iris_df = spark.read.format ("csv") .option ("header","true") .load
("iris.csv")
```

Prepare the necessary data cleaning to be processed by the algorithm, this through the definition of the data type of each column and a change in its name for a better handling of the data.

```
import org.apache.spark.sql.types._
val data = iris_df.withColumn ("sepal_length", $"sepal_length".cast
(DoubleType)). withColumn ("sepal_width", $"sepal_width".cast
(DoubleType)) .withColumn ("petal_length", $"petal_length".cast
(DoubleType)). withColumn ("petal_width", $"petal_width".cast
(DoubleType))
```

**2. What are the names of the columns?**

Once the variable that contains the dataset has been defined, it is possible to show the name of each column using the columns method, this shows us the name of the columns in an array.

```
scala> data.columns
res0: Array[String] = Array(sepal_length, sepal_width, petal_length,
petal_width, species)
```

**3. What is the schema like?**

The scheme allows you to see the type of data that each column of the CSV has, in the case of the iris the scheme is defined by 4 variables, these refer to characteristics of the iris in question, such as the length of cepal and the petal, in the same way with the width, and the last column shows the type of species.

```
scala> data.printSchema ()
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)
```

**4. Print the first 5 columns.**
Using the show function you can show the amount required between the parentheses, in this case 5, in this way showing the values of each of its columns.

```
scala> data.show (5)
+ ------------ + ----------- + ------------ + --- -------- + ------- +
| sepal_length | sepal_width | petal_length | petal_width | species |
+ ------------ + ----------- + ------------ + ----------- + ------- +
|          5.1|          3.5|           1.4|          0.2| setosa |
|          4.9|          3.0|           1.4|          0.2| setosa |
|          4.7|          3.2|           1.3|          0.2| setosa |
|          4.6|          3.1|           1.5|          0.2| setosa |
|          5.0|          3.6|           1.4|          0.2| setosa |
+ ------------ + ----------- + ------------ + ----------- + ------- +
```

**5. Use themethod describe () to learn more about the data in the DataFrame.**
With themethod **describe**(), it was possible to observe common statistical data that allow us to have a better understanding of each column.

```
scala> data.describe (). show ()
+ ------- + ------------------ + ----------- -------- +
------------------ + ------------------ + --- ------ +
| summary | sepal_length | sepal_width | petal_length | petal_width |
species |
+ ------- + ------------------ + ------------------- + -
---------------- + ------------------ + --------- +
| count |              150|                150|                 150|
150|       150|
| mean | 5.843333333333335|
3.0540000000000007|3.7586666666666693|1.1986666666666672|     null|
| stddev |0.8280661279778637|0.43359431136217375|
1.7644204199522262|0.7631607417008414|     null|
| min |              4.3|                2.0|                 1.0|
0.1| setosa |
| max |              7.9|                4.4|                 6.9|
2.5| virginica |
+ ------- + ------------------ + ------------------- + -
---------------- + ------------------ + --------- +
```

**6. Make the pertinent transformation for categorical data which will be our labels to be classified.**
To transform the data we first use the Vector Assembler function, Vector Indexer is a transformer that combines a certain list of columns into a single vector column. This new vector is added as a new column (features) to be used later for prediction.

```
import org.apache.spark.ml.feature.VectorAssembler
val assembler = new VectorAssembler() .setInputCols
(Array("sepal_length", "sepal_width", "petal_length", "petal_width")).
SetOutputCol ("features")
val features = assembler.transform (data)
features.show (5)

+ - ---------- + ----------- + ------------ + ----------- + - ----- +
----------------- +
| sepal_length | sepal_width | petal_length | petal_width | species |
features |
+ ------------ + ----------- + ------------ + ----------- + ------- +
----------------- +
|         5.1|         3.5|          1.4|         0.2| setosa |
[5.1,3.5,1.4,0.2] |
|         4.9|         3.0|          1.4|         0.2| setosa |
[4.9,3.0,1.4,0.2] |
|         4.7|         3.2|          1.3|         0.2| setosa |
[4.7,3.2,1.3,0.2] |
|         4.6|         3.1|          1.5|         0.2| setosa |
[4.6,3.1,1.5,0.2] |
|         5.0|         3.6|          1.4|         0.2| setosa |
[5.0,3.6,1.4,0.2] |
+ ------------ + ----------- + ------------ + ----------- + ------- +
----------------- +
```

Continuing with the transformation, now the String Indexer library will be used, it will allow encoding a column of strings of tags in a tag index column in this case the "species" column. Converting category column to numeric for prediction.

```
import org.apache.spark.ml.feature.StringIndexer
val labelIndexer = new StringIndexer() .setInputCol ("species")
.setOutputCol ("indexedLabel") .fit (features)
println (s "Found labels: $ {labelIndexer.labels.mkString (" [",", ","]
")} ")

Found labels: [versicolor, virginica, setosa]
```

Now using VectorIndexer, this will help index categorical features in Vector data sets. You can automatically decide which characteristics are categorical and convert the original values to category indices.

```
import org.apache.spark.ml.feature.VectorIndexer
val featureIndexer = new VectorIndexer() .setInputCol ("features")
.setOutputCol ("indexedFeatures") .setMaxCategories (4) .fit (features)
```

Once the transformation is finished, the dataset selection is made for the model training and the test percentage, For this practice, 60% training and 40% tests will be used. Ending with the definition of the layers variable that will be used in the "MultilayerPerceptronClassifier" model.

```
val splits = features.randomSplit (Array(0.6, 0.4))
val trainingData = splits (0)
val testData = splits (1)

val layers = Array[Int] (4, 5, 5, 3)
```

**7. Build the model of classification and explain its architecture.**

First we import the "MultilayerPerceptronClassifier" classification library, in a new variable we make an instance of the model to be used with the setLayers options to select the variable that defines the number of layers, followed by setLabeCol where we select the new index created previously, like this selects the column with the vector of the values of each column, finalizing the definition of other options for a better execution of the model such as the speed, the maximum number of iterations and the size of the blocks to be analyzed.

To create the model, first a new variable is created that contains the conversion of the next prediction from index to string where it is going to be incorporated into a new column called “predictedLabel”.

For the creation of the model, it is necessary to implement a variable from the installation of a Pipeline, it allows mixing the workflows of the different variables that have been created, which are: labelIndexer, featureIndexer, trainer and labelConverter.

Finally, the model is created using the pipeline variable and the fit method with the training data.

```
import org.apache.spark.ml.classification.MultilayerPerceptronClassifier
val trainer = new MultilayerPerceptronClassifier() .setLayers (layers)
.setLabelCol ("indexedLabel") .setFeaturesCol ("indexedFeatures")
.setBlockSize (128) .setSeed (System.currentTimeMillis) .setMaxIter
(200)

import org.apache.spark.ml.feature.IndexToString
val labelConverter = new IndexToString() .setInputCol ("prediction")
.setOutputCol ("predictedLabel") .setLabels (labelIndexer.labels)

import org.apache.spark.ml.Pipeline
val pipeline = new Pipeline() .setStages (Array(labelIndexer,
featureIndexer, trainer, labelConverter))

val model = pipeline.fit (trainingData)
```

**8. Print the results of the model.**

Once the model is created, it is necessary to create a new variable for the prediction which is created by calling the "transform" method of the model. Making use in this case of the data for the test.

```
val predictions = model.transform (testData)
predictions.show (5)

+ ------------ + ----------- + --------- --- + ----------- + ------- +
----------------- + -------- ---- + ----------------- +
-------------------- + ------ ------------- + ---------- +
-------------- +
| sepal_length | sepal_width | petal_length | petal_width | species |
features | indexedLabel | indexedFeatures | rawPrediction | probability
| prediction | predictedLabel |
+ ------------ + ----------- + ------------ + ----------- + ------- +
----------------- + ------------ + ---------- ------- +
-------------------- + -------------------- + ---------- +
-------------- +
|         4.3|         3.0|          1.1|         0.1| setosa |
[4.3,3.0,1.1,0.1] |          2.0| [4.3,3.0,1.1,0.1] |
[-66.101884784536... | [8.44215869493861... |        2.0| setosa |
|         4.4|         2.9|          1.4|         0.2| setosa |
[4.4,2.9,1.4,0.2] |          2.0| [4.4,2.9,1.4,0.2] |
[-66.033126609199... | [1.03295317009589... |        2.0| setosa |
|         4.4|         3.0|          1.3|         0.2| setosa |
[4.4,3.0,1.3,0.2] |          2.0| [4.4,3.0,1.3,0.2] |
[-66.066245485006... | [9.37243165769060... |        2.0| setosa |
|         4.6|         3.6|          1.0|         0.2| setosa |
[4.6,3.6,1.0,0.2] |          2.0| [4.6,3.6,1.0,0.2] |
[-66.167489732401... | [6.96637809060764... |        2.0| setosa |
|         4.7|         3.2|          1.6|         0.2| setosa |
[4.7,3.2,1.6,0.2] |          2.0| [4.7,3.2,1.6,0.2] |
[-66.023075533948... | [1.06390712999899... |        2.0| setosa |
+ ------------ + ----------- + ------------ + ----------- + ------- +
----------------- + ------------ + ---------- ------- +
-------------------- + -------------------- + ---------- +
-------------- +
```

To check the efficiency of the model, use the "MulticlassClassificationEvaluator" library, which allows the creation of a variable that contain the accuracy that the prediction generates with its actual value of "indexedLabel" which contains its true categorical data. Once obtained we can show its efficiency, in this the model shows an accuracy of 0.9696 with an error of 0.030, which suggests that the model is efficient.

```
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
```

```
val evaluator = new MulticlassClassificationEvaluator() .setLabelCol
("indexedLabel") .setPredictionCol ("prediction") .setMetricName
("accuracy")
val accuracy = evaluator.evaluate (predictions)
println ("Test Error =" + (1.0 - accuracy))

accuracy: Double = 0.9696969696969697
Test Error = 0.030303030303030276
```

**YouTube Link**: https://www.youtube.com/watch?v=jgmMGNhPIRQ