EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

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
Ingeniería en Sistemas Computacionales

Materia y serie:
Datos Masivos BDD-1704TI9A

Unidad a evaluar: Unidad II

Nombre de la Tarea:
Práctica 3

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

Nombre del docente:
José Christian Romero Hernández

Practice 3

In order to use the necessary methods to perform the classification by Random Forest it is necessary first to import all the libraries that are going to be needed.

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.
{RandomForestClassificationModel, RandomForestClassifier}
import
org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature. {IndexToString, StringIndexer,
VectorIndexer}
```

Subsequently, the data set with which the training and test for the prediction will be carried out must be loaded, so the format and path are specified where the file to use is located.

```
val data = spark.read.format ("libsvm") .load
("sample_libsvm_data.txt")
```

The steps to perform a classification by Random Forest is practically the same as that used for Decision Tree, only the model to be used changes, which It is named after the type of classification to be carried out.

In the labelIndexer variable are the data that is tried to get with the respective classification method, and featureIndexer are the characteristics that will be used to get the data, all this using StringIndexer and VectorIndexer, the first one serves to translate the label column to a new column called indexedLabel, and the second one also takes the characteristics and creates a vector called indexedFeatures, the setMaxCategories method serves to establish a limit in which the data will be treated as categories.

```
val labelIndexer = new StringIndexer() .setInputCol ("label")
.setOutputCol ("indexedLabel")
.fit (data)
val featureIndexer = new VectorIndexer() .setInputCol ("features")
.setOutputCol ("indexedFeatures") .setMaxCategories (4) .fit
(data)
```

As in Decision Tree, the data is separated into 2 parts, the training part in which the 70% of the data and the test one, with the remaining 30%.

```
val Array(trainingData, testData) = data.randomSplit (Array(0.7,
0.3))
```

The difference between Decision Tree and Random Forest is found here, since RandomForestClassifier is used to generate the model, it is specified that the label column will be called indexedLabel and the one for features will be called indexedFeatures and finally the number of trees to use is set, which will be 10.

```
val rf = new RandomForestClassifier() .setLabelCol
("indexedLabel") .setFeaturesCol ("indexedFeatures") .setNumTrees
(10)
```

The labelConverter variable will store the values that once the entire process has been performed, the indexedLabel is translated back to label.

```
val labelConverter = new IndexToString() .setInputCol
("prediction") .setOutputCol ("predictedLabel") .setLabels
(labelIndexer.labels)
```

The pipeline is used to perform all the necessary processes for RandomForest, taking the variables created so far.

```
val pipeline = new Pipeline() .setStages (Array(labelIndexer,
featureIndexer, rf, labelConverter))
```

Once the pipeline is created, the data previously divided for training is passed as a parameter. This is stored in the model variable, it already contains the 10 trees, and I just need to evaluate it.

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

To carry out the evaluation, the transform method is used and the test data is passed to make the predictions, which will result in a dataset that will be stored in the predictions variable.

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

In order to visualize the results, the columns are selected and with the show method it is specified that the first 5 results are displayed.

```
predictions.select ("predictedLabel", "label", "features") .show
(5)
```

Result:

```
predictions: org.apache.spark.sql.DataFrame = [label: double, features: vector
... 6 more fields]
+ -------------- + ----- + -------------------- +
|predictedLabel|label|            features|
+ -------------- + ----- + -------------------- +
|           0.0|  0.0| (692, [98,99,100,1...|
|           1.0|  0.0| (692, [100,101,102...|
|           0.0|  0.0| (692, [122,123,124. ..|
|           0.0|  0.0| (692, [122,123,148...|
|           0.0|  0.0| (692, [124,125,126...|
+ ---------- ---- + ----- + -------------------- +
only showing top 5 rows
```

The following lines of code, as in the decision tree, are to obtain the accuracy and error rates, as well as to display the Random Forest model.

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

Result:

```
accuracy: Double = 0.875
Test Error = 0.125
```

```
val rfModel = model.stages (2) .asInstanceOf
[RandomForestClassificationModel]
println (s "Learned classification forest model: \ n $
{rfModel.toDebugString}")
```

Result:

```
Learned classification forest model:
 RandomForestClassificationModel (uid=rfc_d58c33307ff3) with 10 trees
  Tree 0 (weight 1.0):
    If (feature 552 <= 5.5)
     If (feature 356 <= 19.0)
      Predict: 0.0
     Else (feature 356 > 19.0)
      Predict: 1.0
    Else (feature 552 > 5.5)
```

```
   If (feature 323 <= 23.0)
    Predict: 1.0
   Else (feature 323 > 23.0)
    Predict: 0.0
Tree 1 (weight 1.0):
  If (feature 567 <= 8.0)
   If (feature 456 <= 31.5)
    Predict: 0.0
   Else (feature 456 > 31.5)
    Predict: 1.0
  Else (feature 567 > 8.0)
   If (feature 317 <= 8.0)
    Predict: 0.0
   Else (feature 317 > 8.0)
    Predict: 1.0
Tree 2 (weight 1.0):
  If (feature 385 <= 4.0)
   If (feature 317 <= 158.0)
    Predict: 0.0
   Else (feature 317 > 158.0)
    Predict: 1.0
  Else (feature 385 > 4.0)
   Predict: 1.0
Tree 3 (weight 1.0):
  If (feature 328 <= 24.0)
   If (feature 439 <= 28.0)
    Predict: 0.0
   Else (feature 439 > 28.0)
    Predict: 1.0
  Else (feature 328 > 24.0)
   Predict: 1.0
Tree 4 (weight 1.0):
  If (feature 429 <= 11.5)
   If (feature 358 <= 17.5)
    Predict: 0.0
   Else (feature 358 > 17.5)
    Predict: 1.0
  Else (feature 429 > 11.5)
   Predict: 1.0
Tree 5 (weight 1.0):
  If (feature 462 <= 63.0)
   If (feature 240 <= 253.5)
```

```
      Predict: 1.0
     Else (feature 240 > 253.5)
      If (feature 600 <= 5.5)
       Predict: 0.0
      Else (feature 600 > 5.5)
       Predict: 1.0
    Else (feature 462 > 63.0)
     Predict: 0.0
  Tree 6 (weight 1.0):
    If (feature 512 <= 8.0)
     If (feature 289 <= 28.5)
      Predict: 0.0
     Else (feature 289 > 28.5)
      Predict: 1.0
    Else (feature 512 > 8.0)
     Predict: 1.0
  Tree 7 (weight 1.0):
    If (feature 512 <= 8.0)
     If (feature 510 <= 6.5)
      Predict: 0.0
     Else (feature 510 > 6.5)
      Predict: 1.0
    Else (feature 512 > 8.0)
     Predict: 1.0
  Tree 8 (weight 1.0):
    If (feature 462 <= 63.0)
     If (feature 324 <= 253.5)
      Predict: 1.0
     Else (feature 324 > 253.5)
      Predict: 0.0
    Else (feature 462 > 63.0)
     Predict: 0.0
  Tree 9 (weight 1.0):
    If (feature 385 <= 4.0)
     If (feature 298 <= 224.5)
      Predict: 0.0
     Else (feature 298 > 224.5)
      Predict: 1.0
    Else (feature 385 > 4.0)
     Predict: 1.0
```