Emre Doğan

Proposed to: Hande Alemdar

CENG790, Assignment 3

April 28, 2019

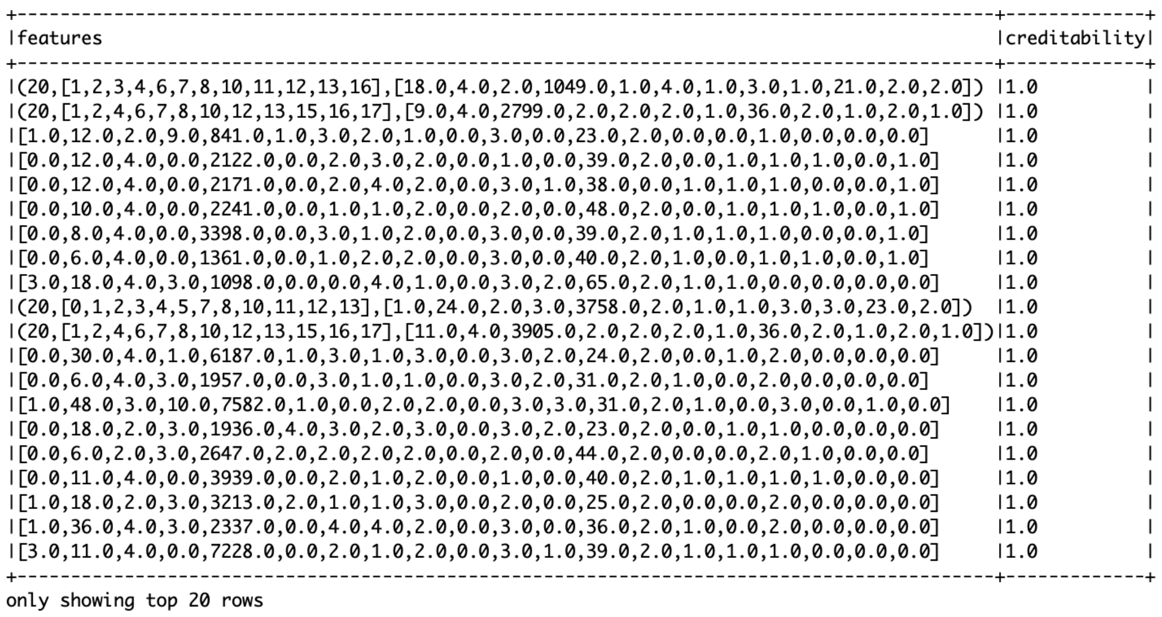
TECHNICAL REPORT

This is the technical report of 3rd Assignment of the course, CENG790 Big Data Analytics. In this assignment, the random forest classifier is used for the credit risk prediction task. There are 20 different attributes and one binary output label for the creditability.

1. The data is read from csv file and written to a dataframe. Then, I used `VectorAssembler` to combine all attributes into a single vector column. By `setInputCols()` method, the names of attributes are set. All these combined attributes are named as `‘features’` and applied to our dataframe, `creditDF`.

```
// question 1.
// transforming a given list of columns (features) into a single vector column.
val assembledFeatures = new VectorAssembler()
  .setInputCols(Array("balance", "duration", "history", "purpose",
  "amount", "savings", "employment", "instPercent", "sexMarried",
  "guarantors", "residenceDuration", "assets", "age", "concCredit",
  "apartment","credits", "occupation", "dependents", "hasPhone",
  "foreign"))
  .setOutputCol("features")
  .transform(creditDF)
```

Assembled features can be seen in Figure 1 below:



```
+-------------------------------------------------------------------------------------------+-------------+
|features                                                                                   |creditability|
+-------------------------------------------------------------------------------------------+-------------+
|(20,[1,2,3,4,6,7,8,10,11,12,13,16],[18.0,4.0,2.0,1049.0,1.0,4.0,1.0,3.0,1.0,21.0,2.0,2.0]) |1.0          |
|(20,[1,2,4,6,7,8,10,12,13,15,16,17],[9.0,4.0,2799.0,2.0,2.0,2.0,1.0,36.0,2.0,1.0,2.0,1.0]) |1.0          |
|[1.0,12.0,2.0,9.0,841.0,1.0,3.0,2.0,1.0,0.0,3.0,0.0,23.0,2.0,0.0,0.0,1.0,0.0,0.0,0.0]      |1.0          |
|[0.0,12.0,4.0,0.0,2122.0,0.0,2.0,3.0,2.0,0.0,1.0,0.0,39.0,2.0,0.0,1.0,1.0,1.0,0.0,1.0]     |1.0          |
|[0.0,12.0,4.0,0.0,2171.0,0.0,2.0,4.0,2.0,0.0,3.0,1.0,38.0,0.0,1.0,1.0,1.0,0.0,0.0,1.0]     |1.0          |
|[0.0,10.0,4.0,0.0,2241.0,0.0,1.0,1.0,2.0,0.0,2.0,0.0,48.0,2.0,0.0,1.0,1.0,1.0,0.0,1.0]     |1.0          |
|[0.0,8.0,4.0,0.0,3398.0,0.0,3.0,1.0,2.0,0.0,3.0,0.0,39.0,2.0,1.0,1.0,1.0,0.0,0.0,1.0]      |1.0          |
|[0.0,6.0,4.0,0.0,1361.0,0.0,1.0,2.0,2.0,0.0,3.0,0.0,40.0,2.0,1.0,0.0,1.0,1.0,0.0,1.0]      |1.0          |
|[3.0,18.0,4.0,3.0,1098.0,0.0,0.0,4.0,1.0,0.0,3.0,2.0,65.0,2.0,1.0,1.0,0.0,0.0,0.0,0.0]     |1.0          |
|(20,[0,1,2,3,4,5,7,8,10,11,12,13],[1.0,24.0,2.0,3.0,3758.0,2.0,1.0,1.0,3.0,3.0,23.0,2.0])  |1.0          |
|(20,[1,2,4,6,7,8,10,12,13,15,16,17],[11.0,4.0,3905.0,2.0,2.0,2.0,1.0,36.0,2.0,1.0,2.0,1.0])|1.0          |
|[0.0,30.0,4.0,1.0,6187.0,1.0,3.0,1.0,3.0,0.0,3.0,2.0,24.0,2.0,0.0,1.0,2.0,0.0,0.0,0.0]     |1.0          |
|[0.0,6.0,4.0,3.0,1957.0,0.0,3.0,1.0,1.0,0.0,3.0,2.0,31.0,2.0,1.0,0.0,2.0,0.0,0.0,0.0]      |1.0          |
|[1.0,48.0,3.0,10.0,7582.0,1.0,0.0,2.0,2.0,0.0,3.0,3.0,31.0,2.0,1.0,0.0,3.0,0.0,1.0,0.0]    |1.0          |
|[0.0,18.0,2.0,3.0,1936.0,4.0,3.0,2.0,3.0,0.0,3.0,2.0,23.0,2.0,0.0,1.0,1.0,0.0,0.0,0.0]     |1.0          |
|[0.0,6.0,2.0,3.0,2647.0,2.0,2.0,2.0,2.0,0.0,2.0,0.0,44.0,2.0,0.0,0.0,2.0,1.0,0.0,0.0]      |1.0          |
|[0.0,11.0,4.0,0.0,3939.0,0.0,2.0,1.0,2.0,0.0,1.0,0.0,40.0,2.0,1.0,1.0,1.0,1.0,0.0,0.0]     |1.0          |
|[1.0,18.0,2.0,3.0,3213.0,2.0,1.0,1.0,3.0,0.0,2.0,0.0,25.0,2.0,0.0,0.0,2.0,0.0,0.0,0.0]     |1.0          |
|[1.0,36.0,4.0,3.0,2337.0,0.0,4.0,4.0,2.0,0.0,3.0,0.0,36.0,2.0,1.0,0.0,2.0,0.0,0.0,0.0]     |1.0          |
|[3.0,11.0,4.0,0.0,7228.0,0.0,2.0,1.0,2.0,0.0,3.0,1.0,39.0,2.0,1.0,1.0,1.0,0.0,0.0,0.0]     |1.0          |
+-------------------------------------------------------------------------------------------+-------------+
only showing top 20 rows
```

Figure 1. Assembled Features.

2. `StringIndexer` method simply encodes the label column to a column of label indices. First, the indexer is created with appropriate input and output label names. The code snippet for this operation is given below:

```
  // question 2.
  // encoding a string column of labels to a column of label indices.
   val indexer = new StringIndexer()
     .setInputCol("creditability")
     .setOutputCol("label")

   val indexedData = indexer
     .fit(assembledFeatures)
     .transform(assembledFeatures)
   indexedData.show
```

3. By using `randomSplit` function, data is split so that we can get training and test data separately. Training data consists of 80% of all the data where test data is the remaining 20%. The following code snippet is used to complete the splitting task:

```
// question 3.
val splitWeights = Array(0.8, 0.2)
val Array(train_data, test_data)=indexedData.randomSplit(splitWeights)
```

4. After completing the data preparing process (feature assembling, indexing and train-test splitting), the random forest classifier is constructed with required parameters. Then, the model is trained with the training data created in the previous part.

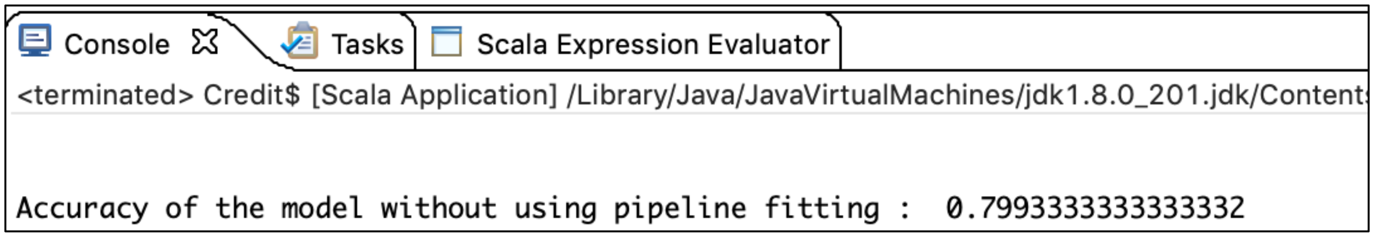
   Notice that this part does not include a pipeline structure. It simply constructs and trains a single random forest classifier with the given parameters.

   (`Chosen parameters: maxDepth=3,maxBins=25,impurity='gini',seed=1234`)

   The code snippet for 4th part is given below.

```
 // question 4.

Val Rfclassifier = new RandomForestClassifier()
.setMaxDepth(3)
.setMaxBins(25)
.setImpurity("gini")
.setFeatureSubsetStrategy("auto")
.setSeed(1234)

// the model creation without a pipeline; in other words, creating the random forest
classifier model.
Val model = Rfclassifier.fit(train_data)

// printing the model description.
Println("\n\nMODEL SUMMARY: \n")
println(model.toDebugString)

// predictions of RFClassifier are taken and model accuracy is resulted.
Val modelPredictions = model.transform(test_data)
val evaluator = new BinaryClassificationEvaluator().setLabelCol("label")
val modelAccuracy = evaluator.evaluate(modelPredictions)
println("\nAccuracy of the model without using pipeline fitting :  " + modelAccuracy)
```

The model created without the pipeline structure gives an accuracy result of 0.799. The related screenshot of accuracy is available in Figure 2.



```
<terminated> Credit$ [Scala Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Content


Accuracy of the model without using pipeline fitting :  0.7993333333333332
```

Figure 2. Accuracy of the Single Model when *(maxDepth=3, maxBins=25, impurity = 'gini', seed=1234)*.

5. In this part, I optimized the model by using pipeline structure. A pipeline provides a simple way to try out different combinations of parameters, using a process called grid search. `maxBins, maxDepth` and `impurity` are the parameters which take different values are set in order to have 12 different models and the best one is chosen. Then, the pipeline is constructed by the array of random forest classifiers. Lastly, the evaluator is constructed. In this case, the evaluator is a binary classification evaluator. It takes two different inputs: `predicted values` and `labels` of the original data. It is a metric to measure how well a trained & tuned model handles the unseen test data.

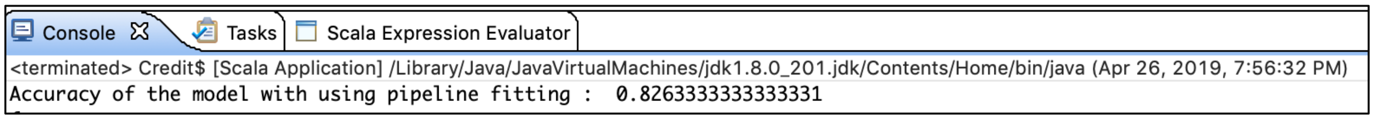
   After the parameter grid, pipeline and evaluator are prepared, `TrainValidationSplit` structure can be built. The training data originated at the beginning is split into training and validation data. The validation data provides an unbiased evaluation of a model fitted on the training dataset while tuning the hyperparameters of the model: `maxBins, maxDepth and impurity`.

   At the final stage, 0.8 x 0.75 = 0.6 ➔ 60% of original data is training data.
   0.8 x 0.25 = 0.2 ➔ 20% of original data is validation data.
   ➔ 20% of original data is test data.

The code snippet for 5th part is given below:

```scala
// question 5.

/* We use a ParamGridBuilder to construct a grid of parameters to search over
 * this grid will have 3 x 3 x 2 = 12 parameter settings for trainValidationSplit to choose
from.
 * TrainValidationSplit will try all these combinations and choose best among them.
 */
val paramGrid = new ParamGridBuilder()
  .addGrid(RFclassifier.maxBins, Array(25,28, 31))
  .addGrid(RFclassifier.maxDepth, Array(4, 6, 8))
  .addGrid(RFclassifier.impurity, Array("entropy", "gini"))
  .build()

// The pipeline consists of a single stage, our constructed random forest classifier.
val pipeline = new Pipeline().setStages(Array(RFclassifier.setSeed(1234)))

// Evaluator for binary classification, which expects two input columns: rawPrediction and
label.
val evaluator = new BinaryClassificationEvaluator().setLabelCol("label")

/*  - TrainValidationSplit is used for hyperparameter tuning task, or model selection.
 *  - Estimator is the algorithm/pipeline to be tuned.
 *  - Evaluator is the metric to measure how well a fitted model does on held-out test data.
 *  - paramMaps corresponds to the changing parameters to find the best model.
 *  - trainValidationSplit is used for hyperparamter tuning. To be able to tune them, we
cannot
 *  use test data directly because we do not want our model to be fitted for specifically test
data.
 *  Instead, we want a more generalized model. So, our test data should not be used for
parameter tuning.
 *  A validation dataset is created (25% of train data = .8 * .25 = .2) from the 25% percent
of original data.
 *   - In summary, 60% data => training
 *                 20% data => validation
 *                 20% data => test
 */
val trainValidationSplit = new TrainValidationSplit()
  .setEstimator(pipeline)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(paramGrid)
  .setTrainRatio(0.75)

/* By using 'TrainValidationSplit' for model selection, the best resulting model
 * is returned when trained with training data and validated by validation data.
 * Then, we feed our test data to the model so that we can observe results on an unseen data!
 */
val pipelineFittedModel = trainValidationSplit.fit(train_data)
val pipelineFittedModelPredictions = pipelineFittedModel.transform(test_data)
val pipelineAccuracy = evaluator.evaluate(pipelineFittedModelPredictions)

println("Accuracy of the model with using pipeline fitting :  " +
evaluator.evaluate(pipelineFittedModelPredictions))
```

After the model is trained, the best model is chosen regarding the evaluation results for the validation data. Then, the test data is fed to the model in order to achieve the test accuracy. In the pipelined model, the best model with tuned hyperparameters gives the result of 0.8263 accuracy. The screenshot from the IDE can be seen in Figure 3.



```
<terminated> Credit$ [Scala Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_201.jdk/Contents/Home/bin/java (Apr 26, 2019, 7:56:32 PM)
Accuracy of the model with using pipeline fitting :  0.8263333333333331
```

Figure 3. The Accuracy of the Best Model Resulted from the Pipeline Structure.

6. The evaluation of the best fitted model of the pipeline structure is done in Part 5 with an accuracy of *0.8263*.

The parameters of the best model (with the accuracy 0.826) resulted from the pipeline structure is extracted by using `extractParamMap()` method. The code snippet to extract the best model is given below.

```
// Question 6.
/* Among 12 different models, the best model is extracted so that
 * we can print its hyperparameters.
 */
val bestModel = pipelineFittedModel.bestModel.
asInstanceOf[org.apache.spark.ml.PipelineModel]
.stages(0)
// finding the parameters of best resulting method.
val bestModelParameters = bestModel.extractParamMap()
println(bestModelParameters)
```

The extracted parameters from the best model are given below:

```
Hyperparameters of the Best Model:
{
      rfc_372ced9699eb-cacheNodeIds: false,
      rfc_372ced9699eb-checkpointInterval: 10,
      rfc_372ced9699eb-featureSubsetStrategy: auto,
      rfc_372ced9699eb-featuresCol: features,
      rfc_372ced9699eb-impurity: gini,
      rfc_372ced9699eb-labelCol: label,
      rfc_372ced9699eb-maxBins: 31,
      rfc_372ced9699eb-maxDepth: 6,
      rfc_372ced9699eb-maxMemoryInMB: 256,
      rfc_372ced9699eb-minInfoGain: 0.0,
      rfc_372ced9699eb-minInstancesPerNode: 1,
      rfc_372ced9699eb-numTrees: 20,
      rfc_372ced9699eb-predictionCol: prediction,
      rfc_372ced9699eb-probabilityCol: probability,
      rfc_372ced9699eb-rawPredictionCol: rawPrediction,
      rfc_372ced9699eb-seed: 1234,
      rfc_372ced9699eb-subsamplingRate: 1.0
}
```

The parameters giving the best result are as follows:

- maxBins = 31
- macDepth = 6
- impurity = “gini”