CENG 790: Big Data Analytics, Fall 2020

Report of Assignment 3: Random Forest Classifier

Extract Features

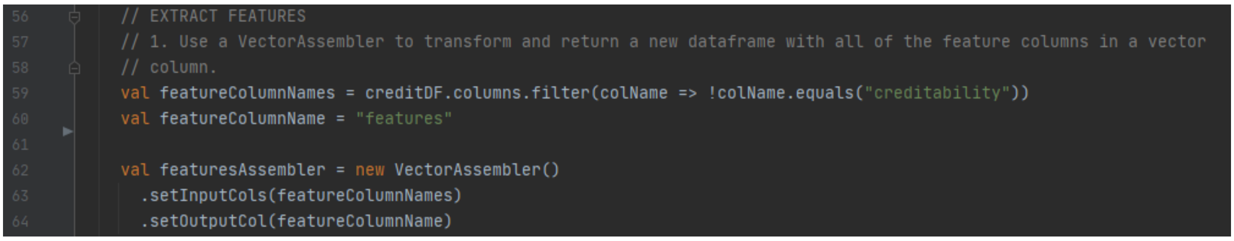
To build a classifier model, you first extract the features that most contribute to the classification. In the credit data set the data is labeled with two classes – 1 (**creditable**) and 0 (**not creditable**).

The features for each item consist of the fields shown below:

- Label → **creditable**: 0 or 1
- Features → {"**balance**", "**duration**", "**history**", "**purpose**", "**amount**", "**savings**", "**employment**", "**instPercent**", "**sexMarried**", "**guarantors**", "**residenceDuration**", "**assets**", "**age**", "**concCredit**", "**apartment**", "**credits**", "**occupation**", "**dependents**", "**hasPhone**", "**foreign**"}

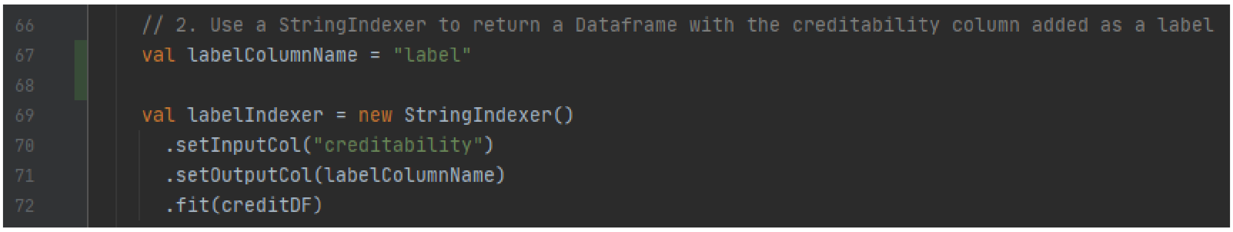
In order for the features to be used by a machine learning algorithm, the features are transformed and put into Feature Vectors, which are vectors of numbers representing the value for each feature.

1. Use a **VectorAssembler** to transform and return a new dataframe with all of the feature columns in a vector column.



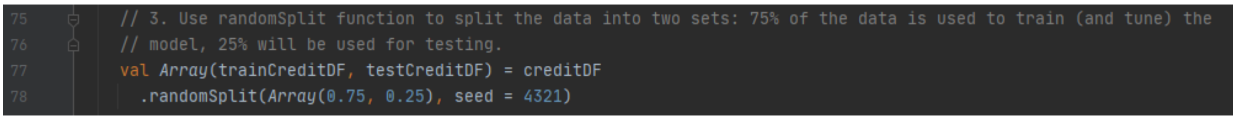
```scala
// EXTRACT FEATURES
// 1. Use a VectorAssembler to transform and return a new dataframe with all of the feature columns in a vector
// column.
val featureColumnNames = creditDF.columns.filter(colName => !colName.equals("creditability"))
val featureColumnName = "features"

val featuresAssembler = new VectorAssembler()
  .setInputCols(featureColumnNames)
  .setOutputCol(featureColumnName)
```

2. Use a **StringIndexer** to return a Dataframe with the creditability column added as a label.



```scala
// 2. Use a StringIndexer to return a Dataframe with the creditability column added as a label
val labelColumnName = "label"

val labelIndexer = new StringIndexer()
  .setInputCol("creditability")
  .setOutputCol(labelColumnName)
  .fit(creditDF)
```

3. Use **randomSplit** function to split the data into two sets: 75% of the data is used to train (and tune) the model, 25% will be used for testing.



```scala
// 3. Use randomSplit function to split the data into two sets: 75% of the data is used to train (and tune) the
// model, 25% will be used for testing.
val Array(trainCreditDF, testCreditDF) = creditDF
  .randomSplit(Array(0.75, 0.25), seed = 4321)
```

Train the model and optimize hyperparameters

The model is trained by making associations between the input features and the labeled output associated with those features. In order to find the best model, we search for the optimal combinations of the classifier parameters.

You will optimize the model using a pipeline. A pipeline provides a simple way to try out different combinations of parameters, using a process called grid search, where you set up the parameters to test, and MLLib will test all the combinations. **Pipelines** make it easy to tune an entire model building workflow at once, rather than tuning each element in the Pipeline separately.

4. Next, we train a RandomForestClassifier with the parameters:

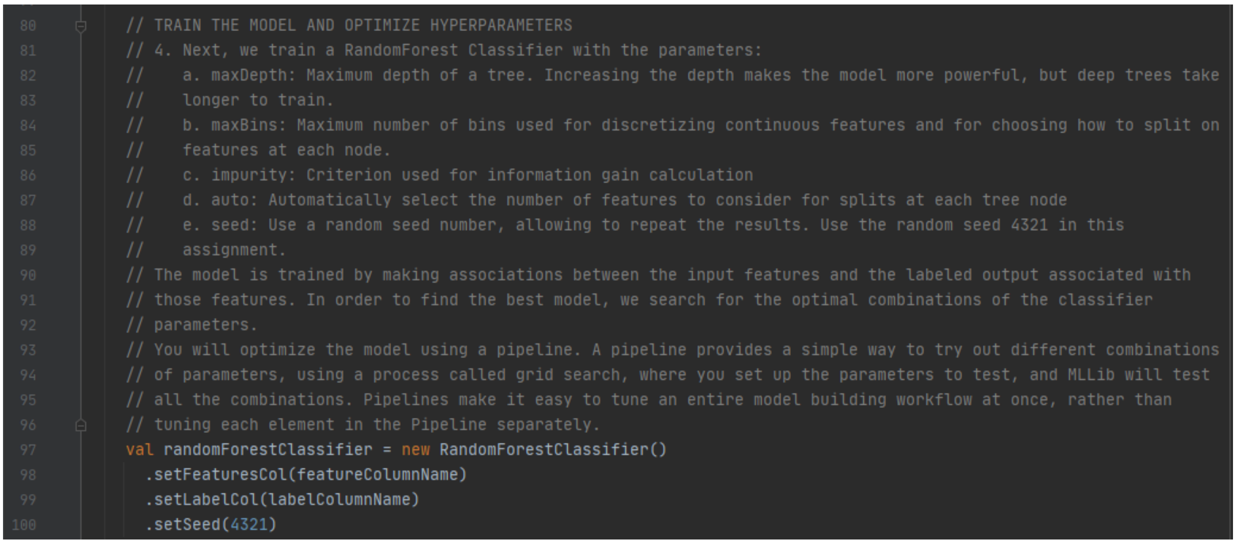
a. **maxDepth**: Maximum depth of a tree. Increasing the depth makes the model more powerful, but deep trees take longer to train.

b. **maxBins**: Maximum number of bins used for discretizing continuous features and for choosing how to split on features at each node.

c. impurity: Criterion used for information gain calculation

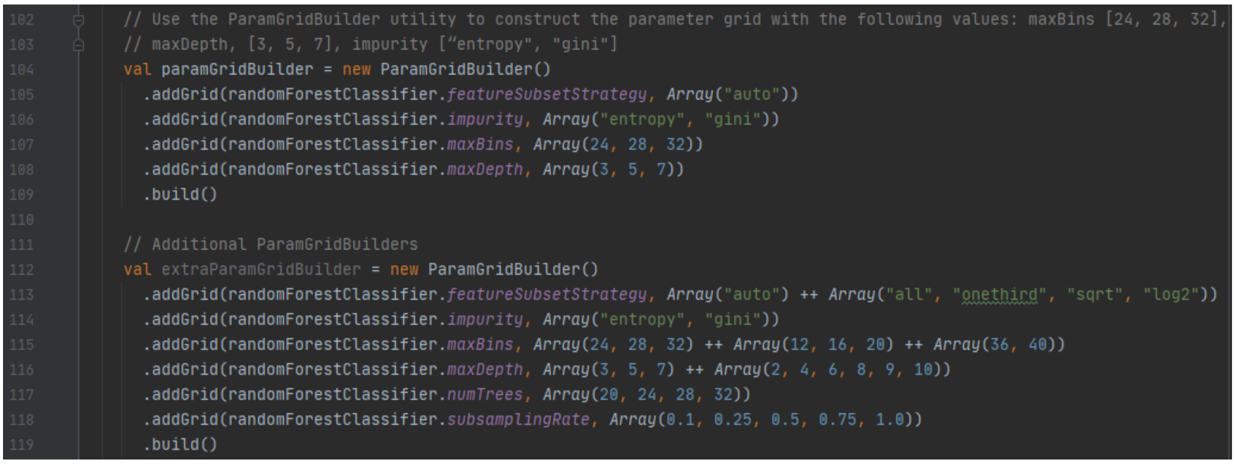
d. **auto**: Automatically select the number of features to consider for splits at each tree node

e. **seed**: Use a random seed number, allowing to repeat the results. Use the random seed 4321 in this assignment.



```
// TRAIN THE MODEL AND OPTIMIZE HYPERPARAMETERS
// 4. Next, we train a RandomForest Classifier with the parameters:
//    a. maxDepth: Maximum depth of a tree. Increasing the depth makes the model more powerful, but deep trees take
//    longer to train.
//    b. maxBins: Maximum number of bins used for discretizing continuous features and for choosing how to split on
//    features at each node.
//    c. impurity: Criterion used for information gain calculation
//    d. auto: Automatically select the number of features to consider for splits at each tree node
//    e. seed: Use a random seed number, allowing to repeat the results. Use the random seed 4321 in this
//    assignment.
// The model is trained by making associations between the input features and the labeled output associated with
// those features. In order to find the best model, we search for the optimal combinations of the classifier
// parameters.
// You will optimize the model using a pipeline. A pipeline provides a simple way to try out different combinations
// of parameters, using a process called grid search, where you set up the parameters to test, and MLLib will test
// all the combinations. Pipelines make it easy to tune an entire model building workflow at once, rather than
// tuning each element in the Pipeline separately.
val randomForestClassifier = new RandomForestClassifier()
  .setFeaturesCol(featureColumnName)
  .setLabelCol(labelColumnName)
  .setSeed(4321)
```

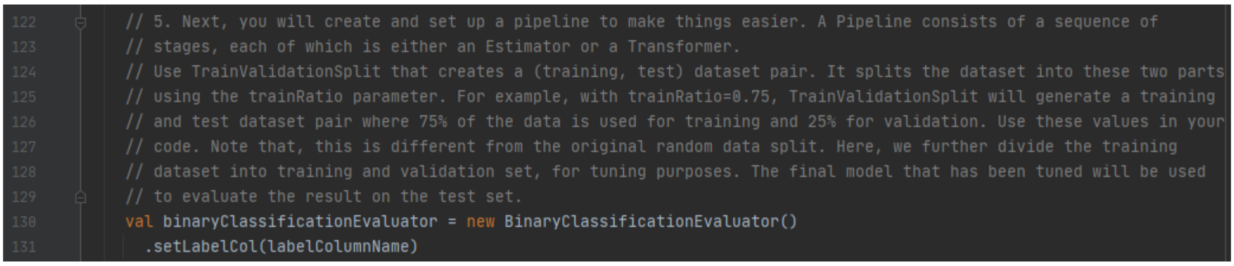
Use the **ParamGridBuilder** utility to construct the parameter grid with the following values: maxBins [24, 28, 32], maxDepth, [3, 5, 7], impurity [“entropy", "gini"]



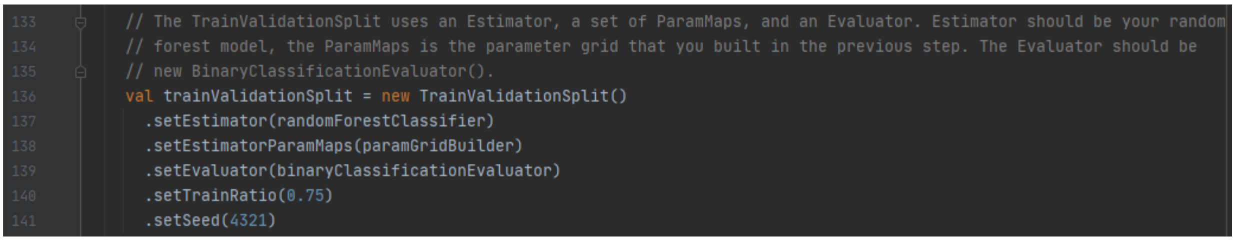
```
// Use the ParamGridBuilder utility to construct the parameter grid with the following values: maxBins [24, 28, 32],
// maxDepth, [3, 5, 7], impurity [“entropy", "gini"]
val paramGridBuilder = new ParamGridBuilder()
  .addGrid(randomForestClassifier.featureSubsetStrategy, Array("auto"))
  .addGrid(randomForestClassifier.impurity, Array("entropy", "gini"))
  .addGrid(randomForestClassifier.maxBins, Array(24, 28, 32))
  .addGrid(randomForestClassifier.maxDepth, Array(3, 5, 7))
  .build()

// Additional ParamGridBuilders
val extraParamGridBuilder = new ParamGridBuilder()
  .addGrid(randomForestClassifier.featureSubsetStrategy, Array("auto") ++ Array("all", "onethird", "sqrt", "log2"))
  .addGrid(randomForestClassifier.impurity, Array("entropy", "gini"))
  .addGrid(randomForestClassifier.maxBins, Array(24, 28, 32) ++ Array(12, 16, 20) ++ Array(36, 40))
  .addGrid(randomForestClassifier.maxDepth, Array(3, 5, 7) ++ Array(2, 4, 6, 8, 9, 10))
  .addGrid(randomForestClassifier.numTrees, Array(20, 24, 28, 32))
  .addGrid(randomForestClassifier.subsamplingRate, Array(0.1, 0.25, 0.5, 0.75, 1.0))
  .build()
```

Also, I tried other parameters such as **numTrees**, **subsamplingRate** and different values for **featureSubsetStrategy**, **maxBins**, **maxDepth** in the tuning phase. But none of them made a significant improvement to accuracy of the model over train and test data.

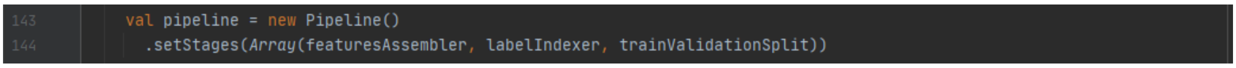
5. Next, you will create and set up a pipeline to make things easier. A Pipeline consists of a sequence of stages, each of which is either an Estimator or a Transformer.
   Use **TrainValidationSplit** that creates a (training, test) dataset pair. It splits the dataset into these two parts using the trainRatio parameter. For example, with trainRatio=0.75, TrainValidationSplit will generate a training and test dataset pair where 75% of the data is used for training and 25% for validation. Use these values in your code. **Note that, this is different from the original random data split.** Here, we further divide the training dataset into training and validation set, for tuning purposes. The final model that has been tuned will be used to evaluate the result on the test set.
   The TrainValidationSplit uses an Estimator, a set of ParamMaps, and an Evaluator. Estimatior should be your random forest model, the ParamMaps is the parameter grid that you built in the previous step. The Evaluator should be **new BinaryClassificationEvaluator().**



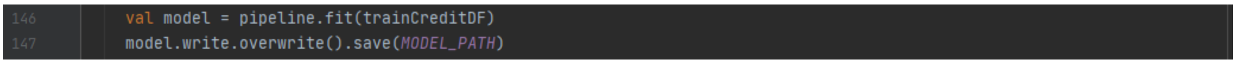
```
// 5. Next, you will create and set up a pipeline to make things easier. A Pipeline consists of a sequence of
// stages, each of which is either an Estimator or a Transformer.
// Use TrainValidationSplit that creates a (training, test) dataset pair. It splits the dataset into these two parts
// using the trainRatio parameter. For example, with trainRatio=0.75, TrainValidationSplit will generate a training
// and test dataset pair where 75% of the data is used for training and 25% for validation. Use these values in your
// code. Note that, this is different from the original random data split. Here, we further divide the training
// dataset into training and validation set, for tuning purposes. The final model that has been tuned will be used
// to evaluate the result on the test set.
val binaryClassificationEvaluator = new BinaryClassificationEvaluator()
  .setLabelCol(labelColumnName)
```



```
// The TrainValidationSplit uses an Estimator, a set of ParamMaps, and an Evaluator. Estimator should be your random
// forest model, the ParamMaps is the parameter grid that you built in the previous step. The Evaluator should be
// new BinaryClassificationEvaluator().
val trainValidationSplit = new TrainValidationSplit()
  .setEstimator(randomForestClassifier)
  .setEstimatorParamMaps(paramGridBuilder)
  .setEvaluator(binaryClassificationEvaluator)
  .setTrainRatio(0.75)
  .setSeed(4321)
```

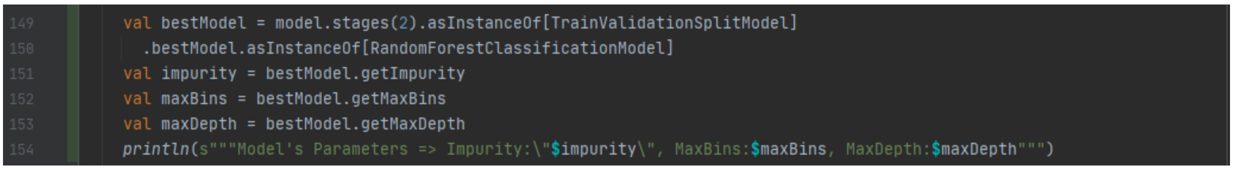


```
val pipeline = new Pipeline()
  .setStages(Array(featuresAssembler, labelIndexer, trainValidationSplit))
```

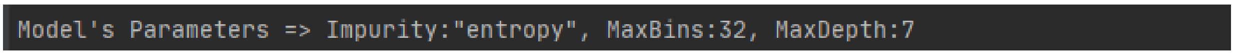


```
val model = pipeline.fit(trainCreditDF)
model.write.overwrite().save(MODEL_PATH)
```

What are the best combinations for the hyper parameters you optimized?



```
val bestModel = model.stages(2).asInstanceOf[TrainValidationSplitModel]
  .bestModel.asInstanceOf[RandomForestClassificationModel]
val impurity = bestModel.getImpurity
val maxBins = bestModel.getMaxBins
val maxDepth = bestModel.getMaxDepth
println(s"""Model's Parameters => Impurity:\"$impurity\", MaxBins:$maxBins, MaxDepth:$maxDepth""")
```

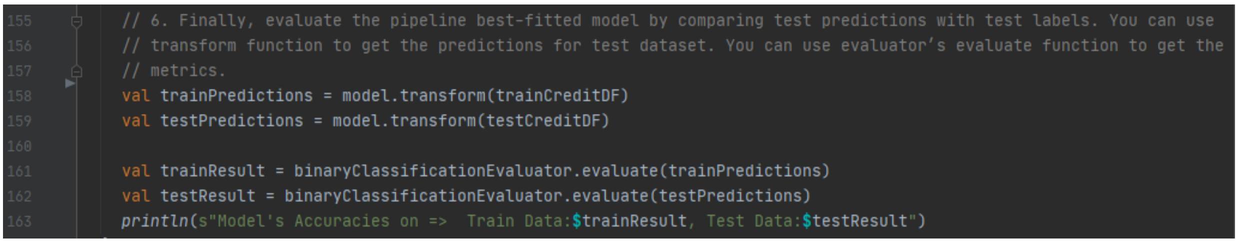


```
Model's Parameters => Impurity:"entropy", MaxBins:32, MaxDepth:7
```

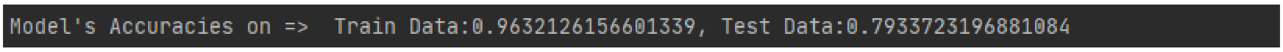
The combination for the best model:

- Impurity is **entropy**,
- MaxBins is **32**,
- MaxDepth is **7**.

6. Finally, evaluate the pipeline best-fitted model by comparing test predictions with test labels. You can use **transform** function to get the predictions for test dataset. You can use evaluator's **evaluate** function to get the metrics.
What are your accuracy values on train and test sets? Feel free to provide more stats and comment on your performance.



```
// 6. Finally, evaluate the pipeline best-fitted model by comparing test predictions with test labels. You can use
// transform function to get the predictions for test dataset. You can use evaluator's evaluate function to get the
// metrics.
val trainPredictions = model.transform(trainCreditDF)
val testPredictions = model.transform(testCreditDF)

val trainResult = binaryClassificationEvaluator.evaluate(trainPredictions)
val testResult = binaryClassificationEvaluator.evaluate(testPredictions)
println(s"Model's Accuracies on =>  Train Data:$trainResult, Test Data:$testResult")
```



```
Model's Accuracies on =>  Train Data:0.9632126156601339, Test Data:0.7933723196881084
```

The model's accuracy on train data is **0.9632126156601339**, and the accuracy on test data is **0.7933723196881084**. However, the accuracy on the test data didn't satisfy me as I had expected higher accuracy value. The reason for that might be the train and validation data size, which is 75% of all data (750 sample), isn't enough or the tuning parameters were not good enough. Therefore, I tried extending the parameter tuning phase in order to achieve better result:

- I extended the ParamGridBuilder with the following parameters:
  - **FeatureSubsetStrategy** -> auto, all, onethird, sqrt, log2
  - **Impurity** -> entropy, gini
  - **MaxBins** -> 24, 28, 32, 34, 36, 38, 40
  - **MaxDepth** -> 2, 3, 4, 5, 6, 7, 8, 9, 10
  - **NumTrees** -> 4, 8, 12, 16, 20, 24, 28, 32
  - **SubsamplingRate** -> 0.1, 0.25, 0.5, 0.75, 1.0
- Normally, there was 18 different combinations. With the extended parameters, there was more than 2500 different combinations and I was sure that the model would be better after the learning process. Yet, it didn't get any better! The best combination was:
  - FeatureSubsetStrategy is **log2**,
  - Impurity is **gini**,
  - MaxBins is **24**,
  - MaxDepth is **7**,
  - NumTrees is **32**,
  - SubsamplingRate is **0.75**.
- The new best model achieved approximately **0.964** accuracy on train data and approximately **0.787** accuracy, which is lower than the previous best model's accuracy, on test data.

I also tried some random parameters intuitively and achieved a better model in terms of accuracy on both train and test data. The model had the following parameters: FeatureSubsetStrategy **auto**, impurity **gini**, maxBins **12**, maxDepth **7**, numTrees **24**, subsamplingRate **1.0**. It achieved **0.9772576526382118** on train data and **0.7949155295646524** on test data.
In addition, I also tried changing randomSplit and trainValidationSplit ratios, but it didn't improve the model's accuracy neither. I even tried removing the seed with the thought of **seed 4321** was the unlucky one and running the training process multiple times in order to achieve a better result but that was not the case. I accepted the fact that it couldn't improve any further with the Random Forrest Classification algorithm.