Group project

Question 1

a) **Decision Tree**

- Create an instance of spark decision tree class which passes 2 args into it: input file and output dir

```
public static void main(String[] args) {
  new SparkDecisionTree().run(args[0], args[1]);
}
```

- Create a spark session and java spark context. This is used later to write the results to the Hadoop file system.

```
    SparkSession spark = SparkSession.builder()
      .appName(appName)
      .getOrCreate();
    context = new JavaSparkContext(spark.sparkContext());
```

- Read the data into a spark data frame using a custom schema. The schema is constructed using the following method:

```
StructType buildSchema(int l) {
  StructField[] f = new StructField[l];
  for(int i = 0; i < l-1; i++) {
    f[i] = DataTypes.createStructField("_c" + i, DataTypes.DoubleType, true);
  }
  f[l-1] = DataTypes.createStructField("_c" + (l-1), DataTypes.StringType, true);
  return new StructType(f);
}
```

- The first 41 columns of the kdd dataset are set to be of type Double and the last column is set to be a string. The last column specifies the categorical type, whether or not the connection was anomalous or normal (hence the string datatype).
- The categorical variables are one hot encoded. This is so they can be used by the decision tree.

```
    // Read and format the input dataset
    Dataset<Row> kddData = spark.read().schema(buildSchema(42)).csv(dataFile);
    Dataset<Row> indexed = new StringIndexer()
      .setInputCol("_c41")
      .setOutputCol("connection_type")
      .fit(kddData)
      .transform(kddData)
      .drop("_c41");
```

- The data is split with a 70/30, train/test split. The train and test set are formatted to be LabeledPoint. This is where each output class has a corresponding set of features.

```
// Split data 70% and 30%
Dataset<Row>[] split = indexed.randomSplit(new double[] {0.7, 0.3});
JavaRDD<Row> train = split[0].toJavaRDD();
JavaRDD<Row> test  = split[1].toJavaRDD();

JavaRDD<LabeledPoint> trainlp = getLabeledPoints(train);
JavaRDD<LabeledPoint> testlp  = getLabeledPoints(test);
```

- The Decision Tree model is created. This uses the static method of the decision tree class to train a decision tree classifier on the training dataset.

```
// Generate Model
Map<Integer, Integer> categoricalFeaturesInfo = new HashMap<>();
String impurity = "gini";
Integer maxDepth = 3;
Integer maxBins = 20;
final DecisionTreeModel model = DecisionTree.trainClassifier(trainlp, numClasses, categoricalFeaturesInfo, impurity, maxDepth, maxBins);
```

- The accuracy of the test set, from the Decision Tree model, is then verified. To do so we apply the trained model to the test set and compare the predicted results vs. the expected results.

```
JavaPairRDD<Double, Double> train_yHatToY = trainlp.mapToPair(p -> new Tuple2<>(model.predict(p.features()), p.label()));
double trainErr = train_yHatToY.filter(pl -> !pl._1().equals(pl._2())).count() / (double) trainlp.count();
double train_accuracy = 1 - trainErr;

JavaPairRDD<Double, Double> yHatToY = testlp.mapToPair(p -> new Tuple2<>(model.predict(p.features()), p.label()));
double testErr = yHatToY.filter(pl -> !pl._1().equals(pl._2())).count() / (double) testlp.count();
double accuracy = 1 - testErr;
```

- The outputs are then written to a .txt file so that the results are readable/interpretable.

```
  outputString += "Train Accuracy: " + train_accuracy + "\n";
  outputString += "Train Error:    " + trainErr + "\n";
  outputString += "Test Accuracy:  " + accuracy + "\n";
  outputString += "Test Error:     " + testErr + "\n";
  outputString += "Duration:       " + duration + "ms\n";
  outputString += "Learned classification tree model:\n" + model.toDebugString() + "\n";

  List<String> outputStrings = new ArrayList<String>();
  outputStrings.add(outputString);

  // Output results
  JavaRDD<String> output = context.parallelize(outputStrings);
  output.saveAsTextFile(outputDirectory);
}
```

Logistic Regression

- All program code runs within *main()* which takes three arguments – input directory (all files to process), output directory and random seed for training/test data split.

```
public static void main(String[] args) {
```

- Create list and schema used to store any results throughout the program that need to be saved to the specified output directory for later evaluation.

```
List<Row> lrModelResults = new LinkedList<Row>();
StructType lrResultsSchema = new StructType(new StructField[]{
    new StructField( name: "Result Description", DataTypes.StringType, nullable: false, Metadata.empty()),
    new StructField( name: "Result Value", DataTypes.StringType, nullable: false, Metadata.empty())
});
```

- Create a spark session.

```
SparkSession spark = SparkSession.builder()
        .appName("SparkLogisticRegression")
        .getOrCreate();
```

- Read the KDD data from the specified input directory into lines (rows). Then map all lines to *LabeledPoint* objects, separating all data by comma, and into a double "features" vector, and class label (normal, anomaly – last feature in line), the latter being converted from string into a double - 0.0 and 0.1, respectively, so it can be handled by the logistic regression model correctly. Lastly, creating a data frame used for further processing.

```
JavaRDD<String> lines = spark.read().textFile(args[0]).toJavaRDD();

JavaRDD<LabeledPoint> linesRDD = lines.map(line -> {
    String[] tokens = line.split( regex: ","); // Comma separated
    double[] features = new double[tokens.length - 1];

    for (int i = 0; i < features.length; i++) {
        features[i] = Double.parseDouble(tokens[i]);
    }

    Vector vectorFeatures = new DenseVector(features);

    if (tokens[features.length].equals("normal")) { // normal class label
        return new LabeledPoint( label: 0.0, vectorFeatures);
    }
    else { // anomaly class label
        return new LabeledPoint( label: 1.0, vectorFeatures);
    }
});

Dataset<Row> kddData = spark.createDataFrame(linesRDD, LabeledPoint.class);
```

- Split the data into a 70/30 test/train split using external input (3rd argument) as the random seed.

```
Dataset<Row>[] randomSplit = kddData.randomSplit(new double[]{0.7, 0.3}, Convert.toInteger(args[2]));
Dataset<Row> trainingData = randomSplit[0];
Dataset<Row> testData = randomSplit[1];
```

- Check if the dataset is imbalanced in terms of the class label we are trying to predict. If it is, it's worth knowing, as may need a strategy to accommodate the interpreting of results. In this case, it's well balanced.

```
double normalCount = trainingData.select( col: "label").where("label = 0").count();
double anomalyCount = trainingData.select( col: "label").where("label = 1").count();
```

- Perform feature selection on the training data using the *ChiSqSelector* to see if we can have similar model accuracy results using less features (top ranked), thus cutting down on computational cost. In this case, the features were cut down to 30 from 41 to reach that balance point. Train/test data is transformed for new selected features.

```
ChiSqSelector featureSelector = new ChiSqSelector()
        .setNumTopFeatures(30) // Top 30 seems to reap similar results as with all 41.
        .setFeaturesCol("features")
        .setLabelCol("label")
        .setOutputCol("selectedFeatures");

ChiSqSelectorModel featureSelectorModel = featureSelector.fit(trainingData);

trainingData = featureSelectorModel.transform(trainingData);
testData = featureSelectorModel.transform(testData);
```

- Obtain the *LogisticRegressionModel* by fitting training data using the *LogisticRegression* classifier. To obtain an optimal threshold value for the model, all row F-measure values for the threshold are gathered and the maximum value is as the best threshold, which is updated for the model.

```
LogisticRegressionModel logisticRegressionModel = logisticRegression.fit(trainingData);

BinaryLogisticRegressionTrainingSummary trainingSummary = logisticRegressionModel.binarySummary();
Dataset<Row> fMeasure = trainingSummary.fMeasureByThreshold();

double maxFMeasure = fMeasure.select(functions.max( columnName: "F-Measure")).head().getDouble( i: 0);
double bestThreshold = fMeasure.where(fMeasure.col( colName: "F-Measure").equalTo(maxFMeasure))
        .select( col: "threshold")
        .head()
        .getDouble( i: 0);

logisticRegressionModel.setThreshold(bestThreshold);
```

- Both training and test sets are now transformed using the created model to perform prediction estimations for the class label for each row. A *BinaryClassificationEvaluator* is then used to evaluate the model accuracy, which is in this case is ROC Curve, and in particular the area under the curve (AUC) as a percentage.

```
Dataset<Row> lrModelPredictionsTrain = logisticRegressionModel.transform(trainingData);
Dataset<Row> lrModelPredictionsTest = logisticRegressionModel.transform(testData);

BinaryClassificationEvaluator lrEvaluator = new BinaryClassificationEvaluator().setLabelCol("label").setRawPredictionCol("rawPrediction");
double lrModelAccuracyTrain = lrEvaluator.evaluate(lrModelPredictionsTrain);
double lrModelAccuracyTest = lrEvaluator.evaluate(lrModelPredictionsTest);

lrModelResults.add(RowFactory.create("Model test Error (AOC) for training data", Double.toString(lrModelAccuracyTrain)));
lrModelResults.add(RowFactory.create("Model test Error (AOC) for test data", Double.toString(lrModelAccuracyTest)));
```

- The model is then run a second time using cross-validation to tune the hyperparameters, in this case parameters of focus are elasticNet, fitIntercept, maxIterations, and regularisation (lambda) values. The model is then re-fit using CV with 3 folds to the training data to with the optimal values from the specified parameter grid.

```
ParamMap[] paramGrid = new ParamGridBuilder()
        .addGrid(logisticRegression.elasticNetParam(), new double[] {0.0, 0.5, 1.0})
        .addGrid(logisticRegression.fitIntercept())
        .addGrid(logisticRegression.maxIter(), new int[] {10, 100, 250})
        .addGrid(logisticRegression.regParam(), new double[] {0.01, 0.5, 2.0})
        .build();

CrossValidator cv = new CrossValidator()
        .setEstimator(logisticRegression)
        .setEvaluator(lrEvaluator)
        .setEstimatorParamMaps(paramGrid)
        .setNumFolds(3)  // Use 3+ in practice
        .setParallelism(2);  // Evaluate up to 2 parameter settings in parallel

CrossValidatorModel cvModel = cv.fit(trainingData);
```

- Using the new model, training and test sets are then transformed to make predictions, re-evaluated, AUC is written to the results buffer. All results gathered to the buffer are then written to the specified output directory within Hadoop file system (argument 2) as a CSV file for analysis.

```
Dataset<Row> cvModelPredictionsTrain = cvModel.transform(trainingData);
Dataset<Row> cvModelPredictionsTest = cvModel.transform(testData);

double cvModelAccuracyTrain = lrEvaluator.evaluate(cvModelPredictionsTrain);
double cvModelAccuracyTest = lrEvaluator.evaluate(cvModelPredictionsTest);

lrModelResults.add(RowFactory.create("CV tuned model test Error (AOC) for training data", Double.toString(cvModelAccuracyTrain)));
lrModelResults.add(RowFactory.create("CV tuned model test Error (AOC) for test data", Double.toString(cvModelAccuracyTest)));

Dataset<Row> lrResultsFile = spark.createDataFrame(lrModelResults, lrResultsSchema);
lrResultsFile.repartition(1).write().mode("append").format("csv").option("header", "true").save(args[1]);
```

b) see the readme.txt file

c)

Decision Tree	Train Accuracy	Test Accuracy	Run Time
1	98.74%	98.26%	10917ms
2	99.06%	98.59%	10754ms
3	98.99%	98.31%	10718ms
4	98.89%	98.16%	11378ms
5	98.86%	98.25%	10800ms
6	99.02%	98.34%	10896ms
7	98.96%	98.36%	11026ms
8	99.06%	98.20%	11096ms
9	99.03%	98.36%	10882ms
10	98.76%	98.34%	11022ms

Training:

Max	Min	Average	Standard Deviation
99.06%	98.74%	98.937%	0.119

Test:

Max	Min	Average	Standard Deviation
98.59%	98.16%	98.317%%	0.118

Logistic Regression	Train Accuracy	Test Accuracy	Run Time
1	96.20%	95.95%	191998 ms
2	95.15%	96.00%	148349ms
3	96.14%	96.20%	177472ms
4	96.08%	96.33%	165486ms
5	96.11%	96.33%	183509ms
6	96.19%	96.23%	204778ms
7	96.07%	96.20%	192505ms
8	96.09%	96.21%	150541ms
9	96.17%	96.04%	184233ms
10	96.21%	96.08%	184398ms

Training:

Max	Min	Average	Standard Deviation
96.21%	95.15%	96.041%	0.317

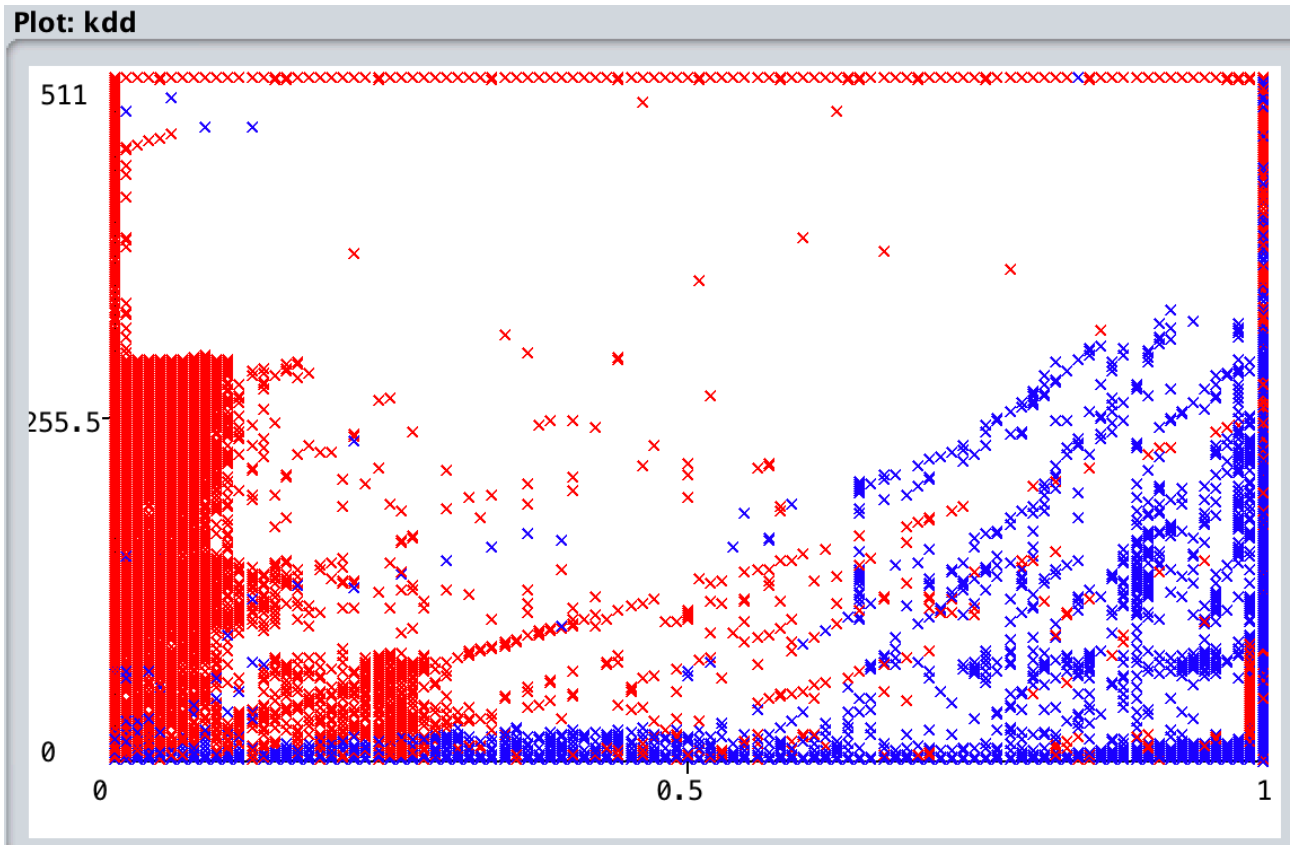
Test:

Max	Min	Average	Standard Deviation
96.33%	95.95%	96.157%	0.133

d)

Looking at all of the metrics, Max, Min, Average, Standard Deviation and Run Time, the decision tree outperforms the Logistic regression in every single one. The biggest difference can be noted in the run time . On average the run time for the decision tree was 10948.9 ms compared to 178326.9 ms from the Logistic Regression. This indicates that the Decision Tree is computationally cheaper and therefore far more efficient out of the two.

By visually exploring the data we can see that relationships between some features exhibit linearly separable characteristics. This is illustrated in the graph below:
This is why we speculate that the accuracy is so high in the decision tree algorithm as we are able to determine, using simple rules, which class the connection type belongs to - anomaly or normal connection.


Plot: kdd
511
255.5
0
0
0.5
1