Group project

Question 1

a) **Decision Tree**

- **Create an instance of spark decision tree class, Pass 2 args, 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 which we will later use to write our 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, which 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 set to string. The last column specifies the categorical type, whether or not the connection was anomalous or normal (hence the string datatype).**
- **We one hot Encode the categorical variables to integers so that they can be used in the decision tree can elaborate here why- one hot encoding using the string indexer.**

```
    // 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");
```

- **Split the data into test/train 70/30, format the train and test set to be LabeledPoint 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);
```

- **Create the decision tree model which uses the static method of the dt class to train a dt classifier on our training-split 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);
```

- **We then need to verify the test set accuracy of the DT model. Therefore we apply the model to the test set and compare the predicted results versus 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;
```

- **We then write outputs to a file so that we are able to read them.**

```
  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

b) see the readme.txt file

c)

Decision Tree	Train Accuracy	Test Accuracy	Run Time
1	92.47%	92.15%	18399ms
2	92.38%	92.38%	12213ms
3	92.43%	92.28%	14855ms
4	92.42%	92.28%	13237ms
5	92.42%	92.28%	12765ms
6	92.50%	92.13%	12810ms
7	93.11%	92.93%	12675ms
8	93.14%	92.75%	12226ms
9	92.40%	92.31%	12728ms
10	92.46%	92.20%	12641ms

Training:

Max	Min	Average	Standard Deviation
93.14%	92.38%	92.573%	0.293

Test:

Max	Min	Average	Standard Deviation
92.93%	92.13%	92.369%	0.263