Assignment #3

COMP 424 Big Data

Qiangqiang Li (liqiangq@gmail.com)
Zhancheng Gan (ganzhan@myvuw.ac.nz)
Lei Yang (alu.yang.lei@gmail.com)

1 Group Work

The team members are: Qiangqiang Li (liqiangq@gmail.com) , Zhancheng Gan (ganzhan@myvuw.ac.nz), and Lei Yang (alu.yang.lei@gmail.com).

(a). Describe the two programs using UML class diagrams and/or pseudo-code;

Answer: The two programs are: Decision Tree and Logistic Regression.

(1). Decision Tree:

The decision tree program only use the main method:

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

After kdd.data loaded, it will be transformed to new libcsv for constructing a Spark session, convert to JavaRDD, and given the dataset using the spark of DataFrame.

```
SparkSession spark = SparkSession.builder().appName("Decision_Tree").master("local").getOrCreate();


Dataset<Row> kddCSV = spark.read().format("csv").load(data_path);
kddCSV.show();
JavaRDD<String> lines = spark.sparkContext().textFile(data_path, 0).toJavaRDD();
JavaRDD<LabeledPoint> linesRDD = lines.map(line -> {
                String[] tokens = line.split(",");
                double[] features = new double[tokens.length - 1];
                for (int i = 0; i < features.length; i++) {
                        features[i] = Double.parseDouble(tokens[i]);
                }
                DenseVector v = new DenseVector(features);
                if (tokens[features.length].equals("normal")) {
                        return new LabeledPoint(0.0, v);
                } else {
                        return new LabeledPoint(1.0, v);
                }
});


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

Then, the program initializes two functions for enumerating labels and enumerating features. The Max categories set as 4 since features with > 4 distinct values are treated as continuous.

```
StringIndexerModel labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel")
                .fit(data);


VectorIndexerModel featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures")
                .setMaxCategories(4)
                .fit(data);
```

The program splits the data into training and testing datatset at ratio: 0.7:0.3.

```
Dataset<Row>[] splits = data.randomSplit(new double[] { 0.7, 0.3 });
Dataset<Row> training_set = splits[0];
Dataset<Row> test_set = splits[1];
```

The program will then train a machine learning model with Decision Tree algorithm and covert the indexed labels back to original labels:

```
DecisionTreeClassifier dt = new DecisionTreeClassifier().setLabelCol("indexedLabel")
                .setFeaturesCol("indexedFeatures");

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

A pipeline will be used to chain the indexers and tree and then to train a pipeline model which also runs the indexers. Then, the model will be used to make predictions for training and testing datasets. Also, the test predictions will be displayed.

```
Pipeline pipeline = new Pipeline()
                .setStages(new PipelineStage[] { labelIndexer, featureIndexer, dt, labelConverter });
PipelineModel model = pipeline.fit(training_set);


Dataset<Row> train_predictions = model.transform(training_set);
Dataset<Row> test_predictions = model.transform(test_set);


test_predictions.select("predictedLabel", "label", "features").show();
```

Then, the program will evaluate the model's predictions and apply it to the both test and train prediction sets

```
MulticlassClassificationEvaluator evaluator = new MulticlassClassificationEvaluator()
                .setLabelCol("indexedLabel").setPredictionCol("prediction").setMetricName("accuracy");


double test_accuracy = evaluator.evaluate(test_predictions);
double training_accuracy = evaluator.evaluate(train_predictions);
```

Eventually, the metrics of accuracies and running time will be reported.

```
System.out.println("test error: " + (1.0 - test_accuracy));
System.out.println("test accuracy: " + test_accuracy);
System.out.println("train accuracy: " + training_accuracy);
DecisionTreeClassificationModel treeModel = (DecisionTreeClassificationModel) (model.stages()[2]);
System.out.println("desision tree model:\n" + treeModel.toDebugString());
long endTime = System.currentTimeMillis();
float running_time = (endTime - startTime)/1000;
System.out.println("running time：  " + running_time + "ms");
```

(2). Logistic Regression:

The same as Decision Tree program, the Logistic Regression program also only use the main method

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

Inherits the same structure with Decision Tree program, the Logistic Regression program will also construct the Spark session after loading the dataset. After that, split the dataset with 0.7:0.3 ratio to the training set and testing set.

```
SparkSession spark = SparkSession.builder().appName("Logistic_Regression").master("local").getOrCreate();


Dataset<Row> kddCSV = spark.read().format("csv").load(data_path);
kddCSV.show();
JavaRDD<String> lines = spark.sparkContext().textFile(data_path, 0).toJavaRDD();
JavaRDD<LabeledPoint> linesRDD = lines.map(line -> {
		String[] tokens = line.split(",");
		double[] features = new double[tokens.length - 1];
		for (int i = 0; i < features.length; i++) {
			features[i] = Double.parseDouble(tokens[i]);
		}
		DenseVector v = new DenseVector(features);
		if (tokens[features.length].equals("normal")) {
			return new LabeledPoint(0.0, v);
		} else {
			return new LabeledPoint(1.0, v);
		}
});
Dataset<Row> data = spark.createDataFrame(linesRDD, LabeledPoint.class);
data.show();

			Dataset<Row>[] splits = data.randomSplit(new double[] { 0.7, 0.3 });
			Dataset<Row> training_set = splits[0];
			Dataset<Row> test_set = splits[1];
```

Then, the logistic regress instance is defined. The max iterations=10, the lamda=0.3, the alpha = 0.8. As discussed during the lectures, the value of lamda and alpha are the initial guess to start.

```
LogisticRegression lr = new LogisticRegression().setMaxIter(10)
		.setRegParam(0.3)
		.setElasticNetParam(0.8);
```

Next, the program is going to fit the training model and obtain the testing loss per iteration.

```
LogisticRegressionModel lrModel = lr.fit(training_set);
System.out.println("Coefficients: " + lrModel.coefficients() + " Intercept: " + lrModel.intercept());


BinaryLogisticRegressionTrainingSummary trainingSummary =
(BinaryLogisticRegressionTrainingSummary) lrModel.summary();


double[] objectiveHistory = trainingSummary.objectiveHistory();
for (double lossPerIteration : objectiveHistory) {
		System.out.println(lossPerIteration);
}
```

Display Receiver-Operating Characteristic for better analysing the data visually, the ROC curve as a dataframe and areaUnderROC is set.

```
Dataset<Row> roc = trainingSummary.roc();
roc.show();
roc.select("FPR").show();
System.out.println(trainingSummary.areaUnderROC());
```

Program applies the selected threshold which is corresponding to the maximum F-Measure for the test set, and then displays the prediction for test set,

```
Dataset<Row> fMeasure = trainingSummary.fMeasureByThreshold();
double maxFMeasure = fMeasure.select(functions.max("F-Measure")).head().getDouble(0);
double bestThreshold = fMeasure.where(fMeasure.col("F-
Measure").equalTo(maxFMeasure)).select("threshold").head()
                .getDouble(0);


lrModel.setThreshold(bestThreshold);


Dataset<Row> predictions = lrModel.transform(test_set);
predictions.show(5);
```

And then, the program will evaluate the model's predictions and apply it to the test. Eventually return the test error and the running time.

```
MulticlassClassificationEvaluator evaluator = new MulticlassClassificationEvaluator().setLabelCol("label")
                .setPredictionCol("prediction").setMetricName("accuracy");
double accuracy = evaluator.evaluate(predictions);
System.out.println("Test Error = " + (1.0 - accuracy));
long endTime = System.currentTimeMillis();
float running_time = (endTime - startTime)/1000;
System.out.println("running time：" + running_time + "ms");
```

(b). Describe how to install and run the two programs step by step in a Readme.txt file.
Answer:
The steps for applying both Decision Tree and Logistic Regression are the same. The contents of Readme.txt is listed below:

- **Step 1**: upload the two java programs: Logistic_Regression.java and Decision_Tree.java to the working folder;
- **Step 2**: ssh to co246a-3.ecs.vuw.ac.nz, and then run environment variables. We edited a HadoopSetup.csh to avoid inputting the commands manually each time. The contents of HadoopSetup.csh is:

```
◆ export HADOOP_VERSION=2.8.0
◆ export HADOOP_PREFIX=/local/Hadoop/hadoop-$HADOOP_VERSION
◆ export PATH=${PATH}:$HADOOP_PREFIX/bin
◆ export SPARK_HOME=/local/spark/spark-2.4.2-bin-hadoop2.7
◆ export PATH=${PATH}:$HADOOP_PREFIX/bin:$SPARK_HOME/bin
◆ export HADOOP_CONF_DIR=$HADOOP_PREFIX/etc/hadoop
◆ export YARN_CONF_DIR=$HADOOP_PREFIX/etc/Hadoop
◆ export
  LD_LIBRARY_PATH=$HADOOP_PREFIX/lib/native:$JAVA_HOME/jre/lib/amd64/s
  erver
```

The spark-hadoop package is already existed in the /local/spark/spark-2.4.2-hadoop2.7 in clusters;

- **Step 3**: upload the dataset: kdd.data to one of the Hadoop clusters under /user/<USERNAME>/input. In this case, co246a-3.ecs.vuw.ac.nz is the cluster we used. The command to transfer kdd.data is: $hdfs dfs -put kdd.data /user/<USERNAME>/input;
- **Step 4**: copy all the libs from /local/spark/spark-2.4.2-hadoop2.7/jars to the working folder;

- **Step 5**: Go to the working folder, make two new class folder: dt_class and lr_class by command: #mkdir dt_class and #mkdir lr_class;
- **Step 6**: patch the java programs to jar patches: #javac -cp "libs/*" -d dt_class Decision_Tree.java and #javac -cp "libs/*" -d lr_class Logistic_Regress.java; and then, run: #jar cvf dt.jar dt_class/ . and #jar cvf lr.jar lr_class .
- **Step 7**: Submit the work to spark Hadoop jobs: #spark-submit --class "CLASSNAME" --master yarn --deploy-mode cluster dt.jar

(c) Report the training and test results (including the max, min, average accuracy and the standard deviation obtained from the 10 runs) and the running time of each program.
Answer:
The following table shows the max, min, average accuracy and running time

	Max training Acc	min training acc	average training acc	Max test Acc	min test ACC	Average test ACC	Average running time over 10 runs
Decision Tree	95.014%	94.632%	94.823%	95.006%	94.513%	94.760%	29 ms
Logistic Regression				83.228%	82.168%	82.698%	23 ms

Standard Deviation of ***decision tree*** program of training accuracy σ: 0.13619192340223
Standard Deviation of ***decision tree*** program of test accuracy σ: 0.16144872250966
Standard Deviation of ***logistic regression*** of test accuracy: σ: 0.35464981319606
The generated model is in Appendix A.

(d) Compare and discuss the results of the two models.
Answers:
Observably, the decision tree program has a more accurate result than logistic regression program about nearly 12% higher in the test dataset. And the average running time over 10 runs of decision tree program is also 6 Ms slower than logistic regression program. So in this case, the decision tree model may be a better model. But in a large dataset, time is important to the circumstance, the decision tree model is more significantly slower, and the decision tree model may be not a good model in other cases.
Secondly, it is seems that the test accuracy is lower than the training accuracy, this is expected as given enough time a model can perfectly fit its training set (overfitting).So the running times is 10 times, and the seed is random to create, it is to avoid overfitting.
Thirdly, it is clear that standard deviation of *decision tree* program is lower than Standard Deviation of *logistic regression* . Lower standard deviation, more closed to the expected value. So in this case, the decision tree model may be more closed to the mean of the data set.

Appendix A. Decision Tree Model for KDD

DecisionTreeClassificationModel (uid=dtc_9d090273b717) of depth 5 with 41 nodes

```
 If (feature 4 <= 29.5)
  If (feature 31 <= 244.5)
   If (feature 39 <= 0.995)
    Predict: 0.0
   Else (feature 39 > 0.995)
    If (feature 33 <= 0.795)
     Predict: 0.0
    Else (feature 33 > 0.795)
     If (feature 2 <= 17.5)
      Predict: 0.0
     Else (feature 2 > 17.5)
      Predict: 1.0
  Else (feature 31 > 244.5)
   If (feature 3 <= 8.5)
    Predict: 0.0
   Else (feature 3 > 8.5)
    If (feature 2 <= 22.0)
     If (feature 34 <= 0.855)
      Predict: 1.0
     Else (feature 34 > 0.855)
      Predict: 0.0
    Else (feature 2 > 22.0)
     Predict: 0.0
 Else (feature 4 > 29.5)
  If (feature 39 <= 0.005)
   If (feature 5 <= 0.5)
    If (feature 33 <= 0.795)
     If (feature 23 <= 42.5)
      Predict: 1.0
     Else (feature 23 > 42.5)
      Predict: 0.0
    Else (feature 33 > 0.795)
     Predict: 0.0
   Else (feature 5 > 0.5)
    If (feature 9 <= 0.5)
     Predict: 1.0
    Else (feature 9 > 0.5)
     If (feature 4 <= 209.5)
      Predict: 0.0
     Else (feature 4 > 209.5)
```

```
        Predict: 1.0
    Else (feature 39 > 0.005)
     If (feature 36 <= 0.005)
      If (feature 4 <= 13919.5)
       If (feature 5 <= 619.5)
        Predict: 0.0
       Else (feature 5 > 619.5)
        Predict: 1.0
      Else (feature 4 > 13919.5)
       If (feature 33 <= 0.795)
        Predict: 1.0
       Else (feature 33 > 0.795)
        Predict: 0.0
     Else (feature 36 > 0.005)
      If (feature 0 <= 3.5)
       Predict: 1.0
      Else (feature 0 > 3.5)
       If (feature 5 <= 129.5)
        Predict: 0.0
       Else (feature 5 > 129.5)
        Predict: 1.0
```

running time：24.0ms

LogisticRegressionModel:

Test Error = 0.17313000276014356
accuracy = 0.8268699972398564
regression model=LogisticRegressionModel: uid = logreg_b3253cc0cb16, numClasses = 2, numFeatures = 41
running time：26.0ms