DATA ANALYTICS WITH R, EXCEL AND TABLAEU

ASSIGNMENT web scraping and text analysis 24.1

ANSWERS

By ASHISH S SHANBHAG

ashishshanbhag108@gmail.com

5. Problem Statement

1. Perform the below given activities:

a. Take a sample data set of your choice

b. Apply random forest, logistic regression using Spark R

Ans

Step-1: Loading required packages and APIs

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.util.HashMap;
import java.util.Map;
import scala.Tuple2;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
```

```
import org.apache.spark.mllib.evaluation.MulticlassMetrics;
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.tree.RandomForest;
import org.apache.spark.mllib.tree.model.RandomForestModel;
import org.apache.spark.mllib.util.MLUtils;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import com.example.SparkSession.UtilityForSparkSession;
import org.apache.spark.SparkConf;
```

Step-2: Creating a Spark session

```
static SparkSession spark = UtilityForSparkSession.mySession();
```

Here is the UtilityForSparkSession class:

```
public class UtilityForSparkSession {
    public static SparkSession mySession() {
        SparkSession spark =  SparkSession.builder()
.appName("HeartDiseasesPrediction")
                                        .master("local[*]")
                                        .config("spark.sql.warehouse.dir", "E:/Exp/")
                                       .getOrCreate();
        return spark;
    }
}
```

Note here the **Spark SQL warehouse** is set as "E:/Exp/" in Windows 7 environment. Please set the path accordingly based on your OS.

Step-3: Loading, parsing and creating data frame for exploratory view

```
String datapath = "input/YearPredictionMSD/YearPredictionMSD";
```

//String datapath = args[0]; // Make this line enable if you want to take input from commanline

Dataset<Row> df = spark.read().format("libsvm").option("header", "true").load(datapath); // Here the format is the "libsvm"

df.show(false);

The above line will produce a data frame like Figure 1.

Step-4: Preparing the RDD of LabelPoint for regression as follows:

JavaRDD<LabeledPoint> data = MLUtils.loadLibSVMFile(spark.sparkContext(), datapath).toJavaRDD(); // We use the MLUtils API to load the same data in LibSVM format

Step-5: Preparing the training and test set:

Split the data into training and test sets (89.98147% held out for training and the rest as testing)

JavaRDD<LabeledPoint>[] splits = data.randomSplit(new double[]{0.8998147, 0.1001853}); JavaRDD<LabeledPoint> trainingData = splits[0];

JavaRDD<LabeledPoint> testData = splits[1];

Step-6: Training the RF model

First, let's set the required parameters before training the RF model for regression.

Map<Integer, Integer> categoricalFeaturesInfo = new HashMap<>();

The above empty categoricalFeaturesInfo indicates that all features are continuous. That also means there are/is no categorical variables in the dataset we saw above.

Integer numTrees = 20;

String featureSubsetStrategy = "auto"; // You can set it as "onethird". However, it's sometimes wiser to let the algorithm choose the best for the dataset we have.

String impurity = "variance"; // Here the impurity is set as "vairance" for the regression related problem.

Integer maxDepth = 20;

Integer maxBins = 20;

Integer seed = 12345;

Note the above parameters are here set naively that means without applying the hyperparameters tuning. Readers are suggested to tune ML model before setting these values out. Now let's train the RandomForest model by utilizing the above parameters as follows:

```
final RandomForestModel model = RandomForest.trainRegressor(trainingData, categoricalFeaturesInfo, numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins, seed);
```

Step-7: Evaluating the model

First, let's calculate the Test Means Square error to weekly evaluate the model. Before that, we need to prepare the prediction and the label for evaluating the model on test instances and computing the test error as follows:

```
JavaPairRDD<Double, Double> predictionAndLabel =
    testData.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
      @Override
      public Tuple2<Double, Double> call(LabeledPoint p) {
        return new Tuple2<>(model.predict(p.features()), p.label());
      }
    });
```

The above code is self-explanatory as you saw in **Figure 1** that the dataset contains only the label and features (i.e., no categorical variables/features). The above code calculates the prediction for each pair of features and label out of the LabeledPoint we created in step 5. Now, let's compute the test mean squared error as follows.

```
Double testMSE =
    predictionAndLabel.map(new Function<Tuple2<Double, Double>, Double>() {
      @Override
      public Double call(Tuple2<Double, Double> pl) {
        Double diff = pl._1() - pl._2();
        return diff * diff;
      }
```

```
    }).reduce(new Function2<Double, Double, Double>() {
      @Override
      public Double call(Double a, Double b) {
        return a + b;
      }
    }) / testData.count();
  System.out.println("Test Mean Squared Error: " + testMSE);
```

Up to this point, the model has been evaluated using a weaker evaluation parameter (i.e., MSE). Now let's evaluate the model by calculating more robust evaluation metrics.

The below source code not only evaluates the model on test instances but also computes related performance measure statistics. However, here let's prepare the prediction and label and compute the prediction on the labels only.

```
JavaRDD<Tuple2<Object, Object>> predictionAndLabels = testData.map(
          new Function<LabeledPoint, Tuple2<Object, Object>>() {
            public Tuple2<Object, Object> call(LabeledPoint p) {
              Double prediction = model.predict(p.features());
              return new Tuple2<Object, Object>(prediction, p.label());
            }
          }
        );
```

Now let's get the evaluation metrics using the Multi metrics evaluator of Spark as follows:

```
  MulticlassMetrics metrics = new
MulticlassMetrics(predictionAndLabels.rdd());
  //System.out.println(metrics.confusionMatrix()); // You could print the
confusion metrics to get more insights.
  // System.out.println(metrics.confusionMatrix());
```

Now let's compute the related performance metrics like precision, recall, f_measure for the label that means column 0 as follows:

```
double precision = metrics.precision(metrics.labels()[0]);

double recall = metrics.recall(metrics.labels()[0]);

double f_measure = metrics.fMeasure();
```

Note the **F1-score** is the harmonic mean of precision and recall that used to evaluate the model performance more sophisticated way.

Now let's play around the prediction column. Let's check how did our RF model perform to predict a label, say the year 2001:

```
double query_label = 2001;
```

Now let's compute some metrics like true positive rate, false positive rate, weighted true positive rate and the weighted false positive rate as follows:

```
double TP = metrics.truePositiveRate(query_label);

double FP = metrics.falsePositiveRate(query_label);

double WTP = metrics.weightedTruePositiveRate();

double WFP =  metrics.weightedFalsePositiveRate();
```

Finally, let's print the above-mentioned metrics as follows:

```
System.out.println("Precision = " + precision);

System.out.println("Recall = " + recall);

System.out.println("F-measure = " + f_measure);

System.out.println("True Positive Rate = " + TP);

System.out.println("False Positive Rate = " + FP);

System.out.println("Weighted True Positive Rate = " + WTP);

System.out.println("Weighted False Positive Rate = " + WFP);
```

I got the following output for the above parameter settings:

```
Test Mean Squared Error: 0.48703966498866
Precision = 0.867559
Recall = 0.63254
F-measure = 0.73800738
True Positive Rate = 0.867559
False Positive Rate = 0.136587
Weighted True Positive Rate = 0.7936875921
Weighted False Positive Rate = 8.147273506679529E-
```

c. Predict for new dataset

Ans However, the current implementation of the **Logistic Regression** algorithm in Spark supports only binary classification, so you cannot solve this problem using the logistic one. Furthermore, you can try using the **Linear Regression**algorithm of course that can predict the outcome even if your dataset has up to 4096 classes with better accuracy.