Network Anomaly Detection

By Tyler Bohrer

What you will need

1. A computer
2. Java
3. Weka, which we had already downloaded from the last lab, that lab can be found by going to this link https://drive.google.com/open?id=1g6mvy-nqxGq1PY2Q1Esnl2cttZ4sv9nJiDRR65oOfIc.
4. Csic data set, which can be found at this link http://users.aber.ac.uk/pds7/csic_dataset/csic2010http.html.

   We will be using the full dataset and the norm_test dataset

OverView:

This lab will cover how to make a program in java, using weka, that will be able to detect network anomalies from data that was collected and saved to the csic dataset.

Downloading and installing Java

To install java go to this link www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

and download the correct version of java for your operating system. You will need to add java to your environment path. To do this copy the path that leads to the bin folder in the java folder. Next type Path into the start menu and select edit environment variables for your account. Here make a new user variable name it java_home, and add the path that you had for the bin folder. Next you will need to download and install an IDE that will be able to run java. For this I used eclipse, but you can search around for any that you feel comfortable with.

Converting a .csv file to .arff

It is important to convert the .csv file format to .arff file format since that is the type that weka was made to use. To do this open weka and go to tools, then select arffViewer. The next step is to load the .csv files and simply save them as the .arff filetype. Now the files are in the correct format to be used for this program.

Using Weka in Java

In the last lab we saw how to use weka to solve some simple problems through the weka application. In this lab we will use java with weka to try to solve the dataset that you had downloaded at the beginning of this lab. Once you have your IDE installed and running we can start the program. The first thing to do is make a new project, and then make a java main file in it. The next thing to do is import a couple of functions from weka and java file reader to be used in this program.

```
import weka.core.Instances;
import java.io.BufferedReader;
import java.io.FileReader;
import weka.classifiers.Classifier;
import weka.classifiers.Evaluation;
import weka.classifiers.misc.InputMappedClassifier;
import weka.classifiers.trees.J48;
```

These will all be needed in order for this program to work properly.

Try-Catch Statement

Next make a try catch statement to catch any error that may come from a failed read etc. This is an important step that must be taken in order for this program to work correctly. The try-catch statement will go inside the main program which can be seen in this example.

```
public class Main {

   public static void main(String[] args) {
         // must use try to stop the program from crashing in the event of a failed read etc...
         try{

         }catch(Exception e){
               System.out.println("There was an error "+e);
         }
    }
}
```

The System.out.println statement in the catch block is used to determine what exactly the error was that caused the program to fail was. For now on the rest of the code will go inside the two { } that are after the try statement.

Getting the Data

The next step is to get the info from the .arff files that we just made. To do this write these two commands into the program.

```
BufferedReader traininfo = new BufferedReader(new
FileReader("C:/Users/Rick/Documents/Datasets/output_http_csic_2010_weka_with_duplications_RAW-RFC2616_escd_v02_full.csv.arff"));
BufferedReader testinfo = new BufferedReader(new
FileReader("C:/Users/Rick/Documents/Datasets/output_http_csic_2010_weka_with_duplications_RAW-RFC2616_escd_v02_norm_test.csv.arff"));
```

The text that is in orange is the paths that lead to the test and training data that this program will use. The text in blue is the variable name that is given to each one of the paths, so traininfo holds the data for the train data and testinfo contains the test data.

From here we can make two instances, one for the traininfo and one for the testinfo.

```
Instances trains = new Instances(traininfo);
Instances tests = new Instances(testinfo);
```

The instances trains will hold a new instance that has the data from the traininfo, and the variable tests will be the instance that holds the data from testinfo. This is important to do since we will now be able to set classifiers to them and to set the index of each. We can now close the traininfo and testinfo since we will no longer be using them in this program.

```
traininfo.close();
testinfo.close();
```

Now we must set the indexes of each of the new instances. To do this use this code.

```
trains.setClassIndex(trains.numAttributes()-1);
tests.setClassIndex(tests.numAttributes()-1);
```

The text in orange is what is used to set the index for these instances. The numAttributes is used to determine the number of attributes that the instance has, and the -1 is used since all instances in java start with 0 and not with 1, example an index of size 4 goes from 0 to 3.

Building the Classifier

The next part of the code will determine which classifier the program will use, and which instance will that classifier will use to train on.

```
Classifier j = new J48();
j.buildClassifier(trains);
```

Here I make a variable that is called j which is assigned to the j48 classifier. It is then used in the next line to build its training data by using the trains variable, which contains the data that holds the training data.

The next step is to allow the code to keep working even if the datasets are slightly different in how their attribute values are named.

```
InputMappedClassifier mapcls = new InputMappedClassifier();
mapcls.setModelHeader(trains);
```

Here we make a new mapped classifier that is called mapcls, and it is given the trains attributes on how to model its header. This is needed in the case that the headers for the trains data and test data do not match up perfectly. Now we can set the classifier for mapcls and tell it only to give us the important warnings, given in blue, and we set the classifier, given in orange.

```
mapcls.setClassifier(j);    // set it to j48
mapcls.setSuppressMappingReport(true);
```

The next step is to evaluate the data that we have been given, which can be done by using this code.

```
Evaluation eval = new Evaluation(trains);    // evaluate over the training set
eval.evaluateModel(mapcls, tests);
```

Here it is easy to see that trains is set as the training data, and it is being evaluated in the next line with eval.evaluateModel(mapcls, tetsts). Here mapcls is the classifier being used and tests is the dataset that we are going to test and see how well it determines if there was an anomaly on the network or not.

Displaying the results

Finally we can put in the last bit of code that will allow us to see how well the classifier does on this dataset.

```
System.out.print(eval.toSummaryString("\n\tInformation\n",false));
```

Here the eval variable is used to print out the result of the program. Below is a snippet of what my output looked like when running this with java.

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (Jan 5, 2017, 11:16:42 AM)
[InputMappedClassifier] Warning: incoming nominal attribute method does not have the same number of values as model attribute method
[InputMappedClassifier] Warning: incoming nominal attribute url does not have the same number of values as model attribute url
[InputMappedClassifier] Warning: incoming nominal attribute host does not have the same number of values as model attribute host
[InputMappedClassifier] Warning: incoming nominal attribute contentLength does not have the same number of values as model attribute contentLength
[InputMappedClassifier] Warning: incoming nominal attribute cookie does not have the same number of values as model attribute cookie
[InputMappedClassifier] Warning: incoming nominal attribute payload does not have the same number of values as model attribute payload
[InputMappedClassifier] Warning: incoming nominal attribute label does not have the same number of values as model attribute label

Results
============

Correctly Classified Instances       48497               46.6317 %
Incorrectly Classified Instances     55503               53.3683 %
Kappa statistic                          0
Mean absolute error                      0.3645
Root mean squared error                  0.499
Relative absolute error                 68.1506 %
Root relative squared error             93.2885 %
Total Number of Instances           104000
```

The next is the output of running the same training and test datasets in weka.

```
=== Evaluation on test set ===

Time taken to test model on supplied test set: 336.43 seconds

=== Summary ===

Correctly Classified Instances       48497               46.6317 %
Incorrectly Classified Instances     55503               53.3683 %
Kappa statistic                          0
Mean absolute error                      0.3645
Root mean squared error                  0.499
Relative absolute error                 68.1506 %
Root relative squared error             93.2885 %
Total Number of Instances           104000

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC      ROC Area  PRC Area  Class
                 0.000    0.534    0.000      0.000    0.000      0.000    ?         ?         anom
                 0.466    0.000    1.000      0.466    0.636      0.000    ?         1.000     norm
Weighted Avg.    0.466    0.000    1.000      0.466    0.636      0.000    0.000     1.000

=== Confusion Matrix ===

     a     b   <-- classified as
     0     0 |     a = anom
 55503 48497 |     b = norm
```

It is clear to see that they both got the same exact results, which is what should be expected.

The J48 did an alright job at classifying the dataset, however we can do better. At the import section at the top of the program type in this new classifier

```
import weka.classifiers.bayes.NaiveBayes;
```

Now all that we need to do to use this classifier instead of J48 is to comment out the old spot where we make the classifier and replace it with NaiveBayes, like so

```
Classifier j = new NaiveBayes();
```

Now run when we run the program it will use NaiveBayes as the classifier and we should get a much better prediction.

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_111\bin\javaw.exe (Jan 13, 2017, 5:54:15 PM)
[InputMappedClassifier] Warning: incoming nominal attribute method does not have the same number of values as model attribute method
[InputMappedClassifier] Warning: incoming nominal attribute url does not have the same number of values as model attribute url
[InputMappedClassifier] Warning: incoming nominal attribute host does not have the same number of values as model attribute host
[InputMappedClassifier] Warning: incoming nominal attribute contentLength does not have the same number of values as model attribute contentLength
[InputMappedClassifier] Warning: incoming nominal attribute cookie does not have the same number of values as model attribute cookie
[InputMappedClassifier] Warning: incoming nominal attribute payload does not have the same number of values as model attribute payload
[InputMappedClassifier] Warning: incoming nominal attribute label does not have the same number of values as model attribute label

Results
============

Correctly Classified Instances       76069               73.1433 %
Incorrectly Classified Instances     27931               26.8567 %
Kappa statistic                          0
Mean absolute error                      0.2766
Root mean squared error                  0.3814
Relative absolute error                 51.7077 %
Root relative squared error             71.3139 %
Total Number of Instances           104000
```

It is clear to see from the output that NaiveBayes was much better at predicting whether the data was a network anomaly than J48 was.

If you choose to use a different classifier for this dataset you may run into problems where there is not enough heap space to do the problem, which i ran into on several occasions using different classifiers.

Also if you run into any problems here is my code that you can use as a reference.

Conclusion

It is easy to see that it is possible to do what weka does in java, like solve network anomalies using machine learning, however this program, with a bit more code, could be made to automatically find network anomalies. This would allow the program to look at incoming data at real time and be able to detect whether there is a network anomaly going on at that moment.

If you wish to go over more in depth description on how this program works go to this link https://weka.wikispaces.com/Use+WEKA+in+your+Java+code, which was the source that I used to help me make this program.