CS176 Programming Assignment 2:

Network Intrusion Detection using Decision Trees

Isabelle B. Tingzon
BS Computer Science
2011-46060

1. INTRODUCTION

In this Programming Assignment (PA), I build a predictive decision tree model to detect computer network intrusion by classifying TCP records as either good or bad connections. I achieve this by using the C5.0 package in R to build and analyze decision trees, generate rulesets, and predict the labels (normal or attack) of a test dataset given the decision tree model of the training set.

One important concept in this assignment is *classification.* Classification is the task of assigning objects to a specific category. More specifically, it is the task of learning a target function *f* that maps each attribute *x* to some predefined category *y.* Classification has two model types, namely *descriptive model* and *predictive model.* In this PA, we are interested in the predictive model, which is a model that predicts the class labels of unknown records (in this case, the test set/TCP records). [1]

2. OBJECTIVES

The objectives of this Programming Assignment are as follows:

- To write R code that uses C5.0 algorithm to generate a decision tree model of the training set in `kddcup_data_10 percent.csv` containing TCP records, to analyze the generated decision tree, and to report the accuracy of its classification
- To write R code that generates rules for the training set in `kddcup_data_10 percent.csv` and to interpret these rules
- To predict the labels (normal or attack) of the test set containing unlabeled TCP connections in `kddcup_testingData_unlabeled_10 percent.csv` and report its accuracy
- To gain a deeper understanding of data mining through classification and to apply the predictive model to a real world application, namely network intrusion detection.

3. METHODOLOGY

In this section, I will discuss the R code implementation of predictive decision tree-based models using the C5.0 algorithm. I will also discuss rule generation for the training set. Lastly, I will discuss an R code implementation of predicting labels (normal or attack) of unlabeled TCP records.

3.1 C5.0 Algorithm

Ross Quinlan was known for developing tree-based models (e.g. ID3 and C4.5). Quinlan continually worked on classification tree and rule-based models, and in the 1980's created C5.0, an extension of C4.5. In this PA, I will be using the C5.0 package developed by Kuhn, Weston, and Coulter in R to build predictive decision trees. [2]

To install C5.0, I run the R system and load the C5.0 package using the command:

```
install.packages("C50")
```

. I can then load the C5.0 library in R using the command:

```
library(c50)
```

3.2 Data Pre-processing

Data pre-processing is an important step in data mining. This technique transforms data into an understandable format such that the final product of this stage is the training set. In this PA, the dataset was made available by DARPA and consists of millions of connection records from a military network environment. The attributes (e.g. duration, protocol_type, service, etc.) can be found in the file `kddcup_names.csv.` For fast computation, we consider merely a subset of the entire dataset.

The datasets available are .csv files. The training set can be found in the file named kddcup_data_10_percent.csv. Meanwhile the test set for which we will be predicting labels can be found in `kddcup_testingData_unlabeled_10_percent.csv.`

To load the datasets, I simply use `read.csv()` as follows:

```
trainingData <-
read.csv("kddcup_data_10_percent.csv",
header=FALSE)

testingData <-
read.csv("kddcup_testingData_unlabeled_10
_percent.csv", header=FALSE)

names <- read.csv("kddcup_names.csv",
sep=":", header=FALSE)
```

I then split `trainingData` into two sets: `trainingSet` and `testingSet`. I split it at an `80:20` ratio (80% of the `trainingData` is the `trainingSet` while the remaining 20% is the `testingSet`). I arrive at this split through a series of experimentations which I will discuss in the next section. For now, it suffices to know that the data is split based on the split that produces the highest accuracy for the `testingData`.

```
#Split data into training set and testing
set
80:20
trainingSet <- trainingData[1:395217,]
testingSet <-
trainingData[395218:494021,]
```

I assign column names to the data frames `trainingSet, testingSet,` and and `testingData` using the attributes listed in `kddcup_names.csv.` I call the last unlabeled column `attack_type` which consists of 23 levels (e.g. normal, back, buffer_overflow, etc.).

```
#assign column names to data files
colnames(trainingSet) <- names[,1]
colnames(trainingSet)[42] <-
"attack_type"
colnames(testingSet) <- names[,1]
colnames(testingSet)[42] <- "attack_type"
colnames(testingData) <- names[,1]
```

3.3 Data Processing

3.3.1 Decision Tree

I now process the data by calling `C5.0.default` or `C5.0( function`. According to the documentation of the `C5.0 Package`, `C5.0()` takes primarily 2 arguments: a data frame or matrix of predictors, x and a factor vector with 2 or more levels, y. In this case, x is the data frame, `trainingSet` excluding the `attack_type` column (or the 42nd column).

```
treeModel <- C5.0(attack_type ~ ., data =
trainingSet)
```

which is also equivalent to:

```
treeModel <- C5.0(x = trainingSet[,-42],
y = trainingSet$attack_type)
```

To increase accuracy of data, I employ a technique called *boosting.* Boosting aids to increase accuracy of the tree model by adding weak learners such that new learners pick up the slack of old learners [3]. The number of boostings can be increased by increasing the value of the `trials` parameter.

```
boostTreeModel <- C5.0(attack_type ~ .,
data = trainingSet, trials = 10)
```

3.3.2 Rule-based Model

According to Kuhn [2], rules are defined as *"if-then statements generated by a tree define a unique route to one terminal node for any sample. A rule is a set of if-then conditions that have been collapsed into independent conditions."*. Examples of rules are the following:

```
if X1 >= 5.0 and X2 >= 300.5 then Class = 5
if X1 >= 5.0 and X2 < 300.5 then Class = 5
if X1 < 5.0 then Class = 3
```

Rules can be easily generated by setting the `rules` parameter in `C5.0()` function to TRUE. The R code is as follows:

```
rules <- C5.0(attack_type ~ ., data =
trainingSet, rules = TRUE)
```

3.3.3 Prediction

Before predicting the labels of the test dataset, I first set the levels of the `testingData` service column to the `trainingSet` service column:

```
levels(testingData$service) <-
trainingSet$service
```

This is done because the number of levels of the service column of `testingData` is one value shorter than that of `trainingSet`.

To predict the labels of `testingData`, I simply call the `predict()` function. The arguments include the three model and the test set:

```
treeModelPred <- predict(treeModel,
testingData, type ="class"))
treeModelProbs <- predict(treeModel,
testingData, type ="prob")
```

To predict the labels for test data using the tree model with boosting, I invoke

```
boostTreeModelPred <-
predict(boostTreeModel, testingData)
```

4. Experimental Results

In this section I will present the experimental results of the R code implementation discussed in the previous section.

4.1 Decision Tree

To get the summary of `treeModel`, I invoke

```
summary(treeModel)
```

The complete result can be found in the `/experimental_results/decision_tree_results` in the folder attached. For the purpose of analysis, the results (some parts omitted) are as follows:

```
Call:
C5.0.formula(formula = attack_type ~ ., data =
trainingSet)

C5.0 [Release 2.07 GPL Edition]     Tue Apr 21
17:19:45 2015
-------------------------------

Class specified by attribute `outcome'

Read 395217 cases (42 attributes) from
undefined.data

Decision tree: [Decision tree is omitted due to
excess length. A full copy of the results can be
found in the attached file
/experimental_results/decision_tree_results]


Evaluation on training data (395217 cases):

          Decision Tree
        ----------------
        Size      Errors
```

```
   112   75( 0.0%)   <<


Class                Cases   False   False
                              Pos     Neg
 -----               -----   -----   -----
 back.                2103       0       6
 buffer_overflow.       17       0       1
 ftp_write.              8       1       3
 guess_passwd.          53       0       2
 imap.                  12       1       2
 ipsweep.             1118       4       0
 land.                  18       0      18
 loadmodule.             8       3       0
 multihop.               7       0       1
 neptune.            84645      36       0
 nmap.                 231       1       2
 normal.             78010      22      22
 perl.                   3       0       0
 phf.                    3       0       0
 pod.                  242       1       0
 portsweep.            859       0       3
 rootkit.                7       0       4
 satan.               1588       3       5
 smurf.             224364       1       0
 spy.                    2       1       0
 teardrop.             879       0       0
 warezclient.         1020       1       6
 warezmaster.           20       0       0


Attribute usage:

100.00% dst_host_serror_rate
 78.55% wrong_fragment
 78.27% srv_count
 41.04% dst_host_diff_srv_rate
 21.57% num_compromised
 21.03% count
 20.70% dst_host_srv_diff_host_rate
 20.40% num_failed_logins
 20.38% flag
 20.32% protocol_type
 20.23% dst_host_srv_serror_rate
 20.19% is_guest_login
 19.81% service
 19.77% dst_host_same_src_port_rate
 19.73% same_srv_rate
 18.65% num_shells
 18.63% hot
  3.19% src_bytes
  0.94% duration
  0.40% dst_bytes
  0.35% rerror_rate
  0.35% dst_host_srv_count
  0.18% num_root
  0.17% logged_in
  0.16% num_file_creations
  0.04% num_access_files
  0.03% dst_host_count
  0.03% dst_host_same_srv_rate
  0.00% diff_srv_rate
Time: 7.9 secs
```

To get the summary of boostTreeModel, I invoke

```
summary(boostTreeModel)
```

Similarly, the complete result can be found in the /experimental_results/boost_decision_tree_results in the folder attached. For the purpose of analysis, the results (some parts omitted) are as follows:

```
Call:
C5.0.formula(formula = attack_type ~ ., data =
trainingSet, trials = 10)


C5.0 [Release 2.07 GPL Edition]      Tue Apr 21
17:37:11 2015
-------------------------------

Class specified by attribute `outcome'

Read 395217 cases (42 attributes) from
undefined.data

-----  Trial 0:  -----

Decision tree: [Decision tree is omitted due to
excess length. A full copy of the results can be
found in the attached file
/experimental_results/boost_decision_tree_result
s]

Evaluation on training data (395217 cases):

            Trial      Decision Tree
            -----    ----------------
                Size      Errors

            0         112   75( 0.0%)
            1          33 4523( 1.1%)
            2          45 2460( 0.6%)
            3          53 5403( 1.4%)
            4          64  712( 0.2%)
            5          63 1299( 0.3%)
            6          59 1133( 0.3%)
            7          64 4836( 1.2%)
            8          56 1503( 0.4%)
            9          71  976( 0.2%)
        boost             42( 0.0%)   <<


        Class                Cases   False   False
                                      Pos     Neg
        -----                -----   -----   -----
        back.                 2103       0       0
        buffer_overflow.        17       2       4
        ftp_write.               8       2       2
        guess_passwd.           53       0       0
        imap.                   12       0       0
        ipsweep.              1118       2       3
        land.                   18       0       0
        loadmodule.              8       0       5
```

```
    multihop.           7       0       2
    neptune.        84645       0       0
    nmap.             231       0       5
    normal.         78010      32       4
    perl.               3       0       0
    phf.                3       0       3
    pod.              242       0       3
    portsweep.        859       0       1
    rootkit.            7       0       3
    satan.           1588       0       3
    smurf.         224364       0       0
    spy.                2       0       1
    teardrop.         879       0       0
    warezclient.     1020       2       3
    warezmaster.       20       2       0


	Attribute usage:

	100.00%	wrong_fragment
	100.00%	dst_host_serror_rate
	 99.99%	src_bytes
	 99.72%	land
	 99.72%	srv_count
	 99.71%	flag
	 99.71%	num_failed_logins
	 99.71%	dst_host_srv_serror_rate
	 99.71%	num_compromised
	 99.71%	dst_host_same_srv_rate
	 99.71%	num_file_creations
	 99.71%	service
	 99.70%	hot
	 99.41%	num_shells
	 98.53%	dst_host_same_src_port_rate
	 79.05%	srv_serror_rate
	 78.37%	dst_host_srv_diff_host_rate
	 78.27%	count
	 78.23%	urgent
	 78.23%	root_shell
	 78.02%	is_guest_login
	 77.86%	duration
	 77.81%	dst_host_count
	 76.31%	rerror_rate
	 75.98%	num_access_files
	 75.95%	dst_host_srv_rerror_rate
	 42.96%	same_srv_rate
	 42.29%	dst_host_diff_srv_rate
	 42.18%	dst_host_rerror_rate
	 42.08%	dst_host_srv_count
	 23.22%	serror_rate
	 23.00%	srv_rerror_rate
	 22.62%	logged_in
	 20.54%	num_root
	 20.48%	protocol_type
	 20.40%	su_attempted
	 18.15%	diff_srv_rate
	  3.41%	dst_bytes
	  0.26%	srv_diff_host_rate


Time: 56.9 secs
```

4.2 Rule Generation

To get the summary of `rules`, I invoke

```
summary(rules)
```

which produces the following results:

```
Call:
C5.0.formula(formula = attack_type ~ ., data =
trainingSet, rules = TRUE)

C5.0 [Release 2.07 GPL Edition]     Tue  Apr  21
17:27:22 2015
-------------------------------

Class specified by attribute `outcome'

Read  395217  cases  (42  attributes)  from
undefined.data

Rules:

Rule 1: (2082, lift 187.8)
	src_bytes > 26408
	src_bytes <= 2500058
	hot > 0
	->  class back.  [1.000]

Rule 2: (90, lift 185.9)
	service = http
	flag = RSTR
	num_failed_logins <= 0
	dst_host_diff_srv_rate <= 0
	->  class back.  [0.989]

Rule 3: (11, lift 21459.7)
	service = telnet
	num_compromised > 0
	num_shells <= 0
	dst_host_same_src_port_rate > 0.37
	->  class buffer_overflow.  [0.923]

Rule 4: (52, lift 7318.8)
	num_failed_logins > 0
	dst_host_same_srv_rate > 0.65
	->  class guess_passwd.  [0.981]

Rule 5: (10, lift 30190.2)
	service = imap4
	dst_host_serror_rate <= 0.93
	->  class imap.  [0.917]

Rule 6: (9, lift 29940.7)
	num_failed_logins <= 0
	dst_host_same_srv_rate > 0.64
	dst_host_serror_rate <= 0.93
	dst_host_srv_serror_rate > 0.2
	->  class imap.  [0.909]

Rule 7: (1080, lift 353.2)
	service  in  {eco_i,  ftp,  gopher,  link,
mtp, name, private, remote_job,
	              rje, ssh, time}
	wrong_fragment <= 0
```

```
       dst_host_srv_diff_host_rate > 0.48
       ->  class ipsweep.  [0.999]

Rule 8: (92/1, lift 346.0)
       src_bytes <= 5
       dst_host_count <= 164
       dst_host_diff_srv_rate > 0.94
       ->  class ipsweep.  [0.979]

Rule 9: (84564/22, lift 4.7)
       flag in {RSTO, S0}
       count <= 327
       diff_srv_rate > 0.02
       dst_host_srv_serror_rate > 0.2
       ->  class neptune.  [1.000]

Rule 10: (84672/29, lift 4.7)
       flag in {RSTO, S0, S3}
       num_failed_logins <= 0
       count <= 327
       dst_host_srv_diff_host_rate <= 0.48
       dst_host_srv_serror_rate > 0.2
       ->  class neptune.  [1.000]

Rule 11: (101, lift 1694.3)
       protocol_type = icmp
       src_bytes <= 19
       dst_host_srv_diff_host_rate > 0.12
       dst_host_srv_diff_host_rate <= 0.48
       ->  class nmap.  [0.990]

Rule 12: (103, lift 1694.6)
       flag = SH
       dst_host_diff_srv_rate > 0.58
       dst_host_srv_serror_rate > 0.2
       ->  class nmap.  [0.990]

Rule 13: (21, lift 1636.5)
       service = private
       src_bytes > 177
       count > 1
       same_srv_rate > 0.94
       ->  class nmap.  [0.957]

Rule 14: (14048/3, lift 5.1)
       protocol_type = tcp
       src_bytes > 111
       src_bytes <= 219
       num_failed_logins <= 0
       is_guest_login <= 0
       dst_host_diff_srv_rate <= 0.94
       dst_host_srv_serror_rate <= 0.2
       ->  class normal.  [1.000]

Rule 15: (1417, lift 5.1)
       duration <= 4
       flag in {S1, S2, SF}
       src_bytes > 843
       dst_bytes <= 1
       logged_in > 0
       ->  class normal.  [0.999]

Rule 16: (1387, lift 5.1)
       protocol_type = tcp
       service = ftp_data
       flag = SF
       dst_bytes <= 1
       count > 3
       ->  class normal.  [0.999]

Rule 17: (1755/3, lift 5.1)
       duration > 13
       duration <= 2700
       flag in {RSTO, S2, SF}
       num_failed_logins <= 0
       num_compromised <= 0
       num_file_creations <= 0
       dst_host_srv_serror_rate <= 0.2
       ->  class normal.  [0.998]

Rule 18: (809/2, lift 5.0)
       protocol_type = icmp
       src_bytes > 19
       src_bytes <= 373
       ->  class normal.  [0.996]

Rule 19: (42/2, lift 4.7)
       service = telnet
       num_compromised > 0
       dst_host_same_src_port_rate <= 0.37
       ->  class normal.  [0.932]

Rule 20: (85232/7241, lift 4.6)
       wrong_fragment <= 0
       srv_count <= 325
       dst_host_serror_rate <= 0.93
       ->  class normal.  [0.915]

Rule 21: (3, lift 4.1)
       dst_host_diff_srv_rate > 0.47
       dst_host_serror_rate > 0.93
       dst_host_srv_serror_rate <= 0.63
       ->  class normal.  [0.800]

Rule 22: (239, lift 1626.4)
       protocol_type = icmp
       wrong_fragment > 0
       ->  class pod.  [0.996]

Rule 23: (612, lift 459.3)
       flag in {OTH, RSTOS0, RSTR}
       dst_bytes <= 1927
       dst_host_srv_count <= 91
       dst_host_diff_srv_rate > 0
       ->  class portsweep.  [0.998]

Rule 24: (243, lift 458.2)
       rerror_rate > 0.98
       same_srv_rate <= 0.94
       dst_host_same_src_port_rate > 0.01
       ->  class portsweep.  [0.996]

Rule 25: (216, lift 458.0)
       count <= 1
       dst_host_count > 164
       dst_host_diff_srv_rate > 0.94
```

```
        dst_host_serror_rate <= 0.93
        ->  class portsweep.  [0.995]

Rule 26: (25, lift 443.0)
        flag in {RSTO, S0}
        dst_host_count > 140
        dst_host_same_src_port_rate > 0.01
        dst_host_serror_rate <= 0.93
        ->  class portsweep.  [0.963]

Rule 27: (1312/1, lift 248.5)
        count > 327
        srv_count <= 325
        dst_host_same_src_port_rate <= 0.5
        ->  class satan.  [0.998]

Rule 28: (1341/2, lift 248.3)
        flag in {REJ, SF}
        src_bytes <= 6
        dst_bytes <= 106
        rerror_rate <= 0.98
        same_srv_rate <= 0.94
        dst_host_srv_serror_rate <= 0.2
        ->  class satan.  [0.998]

Rule 29: (1191/2, lift 248.3)
        flag in {REJ, RSTO, SF}
        logged_in <= 0
        rerror_rate > 0.26
        rerror_rate <= 0.98
        same_srv_rate <= 0.94
        dst_host_srv_serror_rate <= 0.2
        ->  class satan.  [0.997]

Rule 30: (101, lift 246.5)
        service in {other, private}
        flag = SF
        src_bytes <= 52
        wrong_fragment <= 0
        dst_host_same_src_port_rate <= 0.99
        ->  class satan.  [0.990]

Rule 31: (58, lift 244.7)
        service = private
        flag = SF
        dst_host_count > 138
        dst_host_same_src_port_rate > 0.99
        ->  class satan.  [0.983]

Rule 32: (48, lift 243.9)
        rerror_rate > 0.98
        same_srv_rate <= 0.94
        dst_host_same_src_port_rate <= 0.01
        ->  class satan.  [0.980]

Rule 33: (224364, lift 1.8)
        protocol_type = icmp
        src_bytes > 798
        wrong_fragment <= 0
        ->  class smurf.  [1.000]

Rule 34: (879, lift 449.1)
        protocol_type = udp
        wrong_fragment > 0
        ->  class teardrop.  [0.999]

Rule 35: (548, lift 386.8)
        service = ftp_data
        src_bytes > 326
        src_bytes <= 353
        ->  class warezclient.  [0.998]

Rule 36: (275, lift 386.1)
        duration <= 13
        service = ftp
        flag = SF
        num_file_creations <= 0
        is_guest_login > 0
        ->  class warezclient.  [0.996]

Rule 37: (656/6, lift 383.3)
        dst_bytes <= 1
        logged_in > 0
        num_root <= 0
        count <= 3
        dst_host_srv_count <= 70
        dst_host_same_src_port_rate > 0.99
        ->  class warezclient.  [0.989]

Rule 38: (58, lift 381.0)
        src_bytes > 2500058
        dst_host_srv_serror_rate <= 0.2
        ->  class warezclient.  [0.983]

Rule 39: (41, lift 378.5)
        duration > 1
        service = ftp_data
        dst_host_srv_serror_rate > 0
        dst_host_srv_serror_rate <= 0.2
        ->  class warezclient.  [0.977]

Rule 40: (32, lift 376.1)
        duration > 2700
        is_guest_login > 0
        ->  class warezclient.  [0.971]

Rule 41: (22, lift 371.3)
        protocol_type = tcp
        flag in {S3, SF}
        same_srv_rate > 0.94
        dst_host_diff_srv_rate > 0.1
        dst_host_diff_srv_rate <= 0.94
        dst_host_same_src_port_rate > 0.99
        ->  class warezclient.  [0.958]

Rule 42: (620/313, lift 191.9)
        is_guest_login > 0
        ->  class warezclient.  [0.495]

Rule 43: (16, lift 18663.0)
        duration > 1
        service = ftp_data
        flag = SF
        logged_in <= 0
        dst_host_same_src_port_rate > 0.99
        ->  class warezmaster.  [0.944]
```

```
Default class: smurf.


Evaluation on training data (395217 cases):

	    Rules
	  ----------------
	    No      Errors

	    43  147( 0.0%)   <<


	   Class              Cases   False   False
	                               Pos     Neg
	   -----              -----   -----   -----
	   back.               2103       0       6
	   buffer_overflow.      17       0       6
	   ftp_write.             8       0       8
	   guess_passwd.         53       0       1
	   imap.                 12       0       2
	   ipsweep.            1118       1      12
	   land.                 18       0      18
	   loadmodule.            8       0       8
	   multihop.              7       0       7
	   neptune.           84645      25       1
	   nmap.                231       0       6
	   normal.            78010      91      13
	   perl.                  3       0       3
	   phf.                   3       0       3
	   pod.                 242       0       3
	   portsweep.           859       0      24
	   rootkit.               7       0       7
	   satan.              1588       3      10
	   smurf.            224364      27       0
	   spy.                   2       0       2
	   teardrop.            879       0       0
	   warezclient.        1020       0       3
	   warezmaster.          20       0       4


	Attribute usage:

	  78.55% wrong_fragment
	  61.97% src_bytes
	  61.19% protocol_type
	  25.80% dst_host_srv_serror_rate
	  25.45% num_failed_logins
	  22.96% flag
	  22.33% count
	  21.72% dst_host_srv_diff_host_rate
	  21.57% dst_host_serror_rate
	  21.57% srv_count
	  21.40% diff_srv_rate
	   3.79% dst_host_diff_srv_rate
	   3.71% is_guest_login
	   1.13% dst_bytes
	   0.93% service
	   0.89% duration
	   0.83% logged_in
	   0.62% dst_host_same_src_port_rate
	   0.53% hot
	   0.51% num_file_creations
	   0.46% num_compromised
	   0.42% same_srv_rate
	   0.41% rerror_rate
	   0.32% dst_host_srv_count
	   0.17% num_root
	   0.10% dst_host_count
	   0.02% dst_host_same_srv_rate
	   0.00% num_shells

Time: 14.5 secs
```

4.3 Predictiction

To predict the labelsl of the test data, I invoke

```
summary(treeModelPred)
```

and get the following results:

```
back.               1242
buffer_overflow.    0
ftp_write.          0
guess_passwd.       426
imap.               315
ipsweep.            3
land.               0
loadmodule.         6
multihop.           5
neptune.            17101
nmap.               195
normal.             82072
perl.               0
phf.                8
pod.                32913
portsweep.          132529
rootkit.            0
satan.              44152
smurf.              0
spy.                1
teardrop.           51
warezclient.        9
warezmaster.        1
```

Invoking

```
summary(boostTreeModelPred),
```

I get the following results:

```
back.               2053
buffer_overflow.    1
ftp_write.          3
guess_passwd.       426
imap.               0
ipsweep.            0
land.               9
loadmodule.         1
multihop.           1
neptune.            17484
nmap.               84
normal.             177438
perl.               3
phf.                0
pod.                96
portsweep.          413
```

```
rootkit.                2
satan.                  36382
smurf.                  76581
spy.                    0
teardrop.               51
warezclient.            0
warezmaster.            1
```

5. ANALYSIS AND DISCUSSION OF RESULTS

In this section, I will discuss and analyze the experimental results produced in the previous sections.

5.1 Decision Tree

Based on the results of the C5.0 decision tree (see `decision-tree-model.txt` for full description of the decision tree), I created a graphic visualization of the decision tree. Below is the illustration of the first five levels of the decision tree. The *size* or total number of nodes in the generated decision tree is 112.

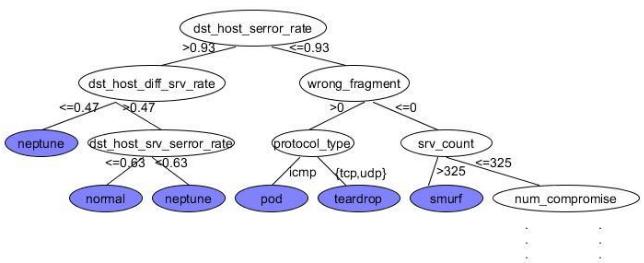

Figure 1.0 First 4 levels of the `treeModel`

To measure the accuracy of the decision tree model, we predict the `attack_type` of the `trainingSet` using the generated `treeModel`:

```
pred <- predict(treeModel, trainingSet)
```

To get the accuracy, we simply compute the following:

```
sum( pred == trainingSet$attack_type ) /
length( pred )
```

The resulting value of accuracy is:

`0.7912028`.

We can increase this accuracy through boosting. One way to see how boosting significantly increases accuracy is by considering the metrics for performance as illustrated in the Confusion Matrix in Figure 2.0.

	PREDICTED CLASS	
ACTUAL CLASS	Class=Yes	Class=No
Class=Yes	a	b
Class=No	c	d

a: TP (true positive)
b: FN (false negative)
c: FP (false positive)
d: TN (true negative)

CS176 (Data Mining)

Figure 2.0 Confusion Matrix [1]

We are interested in false positives and false negatives. False positives are predictions that indicate the presence of the condition when in actuality, there is none. False negatives erroneously reports no presence of the condition when in actuality it is present. In this PA, we want as much as possible to decrease the number of cases reported to be false positives and more importantly, *false negative*.

Consider the number of false positives per class before and after boosting. Without boosting, the decision tree reports a total of 72 false positives. After boosting, the total number of cases reported to be false positives decreases to 42.

Similarly, without boosting, the decision tree reports a total of 69 false negatives. After boosting, the total number of cases reported to be false negatives decreases to 41.

5.2 Rule Generation

We have defined *rules* in the previous sections as if-then statements that define a unique path to a terminal node or *class*. For our training set, we have generated a total of 49 rules.
To illustrate, consider Rule 21:

```
Rule 21: (3, lift 4.1)
dst_host_diff_srv_rate > 0.47
dst_host_serror_rate > 0.93
dst_host_srv_serror_rate <= 0.63
->  class normal.  [0.800]
```

This rule corresponds to the unique path from the root node of the decision tree in Figure 1.0 to the node labeled 'normal', wherein all conditions (i.e. `dst_host_diff_srv_rate > 0.47,dst_host_serror_rate>0.93,dst_host_srv_serror_rate <= 0.63`) must be satisfied. For every unique path to a terminal node, there is exactly one rule.

5.3 Prediction

To predict the labels (whether normal or some type of attack) of the TCP records of the test set, I used the decision tree generated using the training set and the predict() function as seen in the previous sections. That is, given the tree model and rules generated using the C5.0 algorithm, we can predict whether a TCP connection is normal or an attack.

6. CONCLUSION

In this Programming Assignment, I was able to build predictive decision tree model to detect computer network intrusion using the C5.0 algorithm in R. It was found that the C5.0 algorithm reported high accuracy of classification for the training set. Furthermore, this accuracy could be enhanced even more through boosting. I was also able to generate rules which are conditional statements correspond to a unique path to a terminal node in the decision tree. And lastly, I was able to use the decision tree generated using the the training set to predict the labels of unknown classes in the test set with high accuracy.

Through this Programming Assignment, I was able to demonstrate the real world application of data mining in network intrusion detection using classification, or more specifically, decision trees.

7. REFERENCES

[1] Naval, P. 2015. *Classification (Lecture 5)*. https://www.dropbox.com/s/jb1vqg4sdyd7ren/CS176-2S14-15%20Lec%205%20Classification%20and%20Prediction.pdf?dl=0. Last accessed April 14, 2015.

[2] Kuhn, M. *Classification using C5.0 UseR! 2013*. http://static1.squarespace.com/static/51156277e4b0b8b2ffe11c00/t/51e7e42ce4b0fd2e32684bca/1374151724529/user_C5.0.pdf. Last accessed April 13, 2015.

[3] Johnson, C. 2014. *Decision Trees in R using the C50 Package.* http://connor-johnson.com/2014/08/29/decision-trees-in-r-using-the-c50-package/. Last accessed April 14, 2015.