IoT attack detection

Yulia Zamyatina

April, 2021

Contents

Introduction

The **detection_of_IoT_botnet_attacks_N_BaIoT** data set[1] contains a traffic data from 9 commercial IoT (*Internet of Things*) devices authentically infected by Mirai and BASHLITE (Gafgyt).

The data set has 115 attributes (parameters), below is the description of their headers:

1. It has 5 time-frames: L5 (1 min), L3 (10 sec), L1 (1.5 sec), L0.1 (500 ms) and L0.01 (100 ms).
2. The statistics extracted from each traffic stream for each time-frame:
    - *weight*: the weight of the stream (can be viewed as the number of items observed in recent history)
    - *mean*
    - *std (variance)*
    - *radius*: the root squared sum of the two streams' variances
    - *magnitude*: the root squared sum of the two streams' means
    - *covariance*: an approximated covariance between two streams
    - *pcc*: an approximated correlation coefficient between two streams

[1] https://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT

3. It has following stream aggregations:
    - *MI*: ("Source MAC-IP" in N-BaIoT paper) Stats summarizing the recent traffic from this packet's host (IP + MAC)
    - *H*: ("Source IP" in N-BaIoT paper) Stats summarizing the recent traffic from this packet's host (IP)
    - *HH*: ("Channel" in N-BaIoT paper) Stats summarizing the recent traffic going from this packet's host (IP) to the packet's destination host.
    - *HH_jit*: ("Channel jitter" in N-BaIoT paper) Stats summarizing the jitter of the traffic going from this packet's host (IP) to the packet's destination host.
    - *HpHp*: ("Socket" in N-BaIoT paper) Stats summarizing the recent traffic going from this packet's host+port (IP) to the packet's destination host+port. Example 192.168.4.2:1242 -> 192.168.4.12:80

Thus, the column *'MI_dir_L5_weight'* in the data set shows the weight of the recent traffic from the packet's host for L5 time-frame.

I've added extra *'botnet'* column, where I keep information about the attacks from the different botnets and benign traffic. I've used "ga_" prefix for Gafgyt attacks, and "ma_" prefix for Mirai attacks. List of attacks executed and tested can be found in "N-BaIoT: Network-based Detectionof IoT Botnet Attacks Using Deep Autoencoders"[2] article.

Data set structure can be found in **"Appendix. Data set structure"**.

The team that collected this data set used 2/3 of their benign traffic (their train set) to train their deep autoencoder. Then they used remaining 1/3 of benign traffic and all the malicious data (their test set) to detect anomalies with deep autoencoder. The detection of the cyberattacks launched from each of the above IoT devices concluded with 100% TPR.

In HarvardX PH125.9x Data Science course we learned several algorithms that can be used for the classification, such as **KNN**, **rpart** or **randomForest**.

My aim in this project is to check how well all these algorithms can detect anomalies (classify benign traffic or attack) in the data set, how accurate they can perform the classification. I have no task to compare algorithms between each other.

The source data set consists of *.csv and *.rar files, each representing the benign traffic or the attack, that are divided into folders with devices names[3]. Because R has no package that can unpack *.rar files for its own, so all *.rar archives have to be manually unpacked with command line or third-party applications, depending on OS. Therefore, I had to prepare the data set for this project that can be easily downloaded.

The whole data set size was more that 1TB. Nowadays it's not a problem to find a hosting to share this huge data set, but I thought that everyone who will check this project won't be happy to download it. Because the team, that collected this data set, trained and tested their autoencoder for each device separately, I decided that I can use the data only from one device for my project. I've used the data only from *Danmini doorbell* device, but the data for other devices can be downloaded from the source and prepared for the classification using provided *download-data.R* script.

The data set for *Danmini doorbell* has the size of 970MB in *data.csv* file and only 200MB in *data.zip* archive. I used the third-party file hosting to share it, because the archive file size exceeds GitHub limits of file size that can be stored with it.

The archive file is downloading during the first time of using project's scripts and saving as *data.csv* file in a local project data folder. For the next time, saved *data.csv* file is used.

[2] https://arxiv.org/pdf/1805.03409.pdf, p.5
[3] https://archive.ics.uci.edu/ml/machine-learning-databases/00442

Dataset analisys

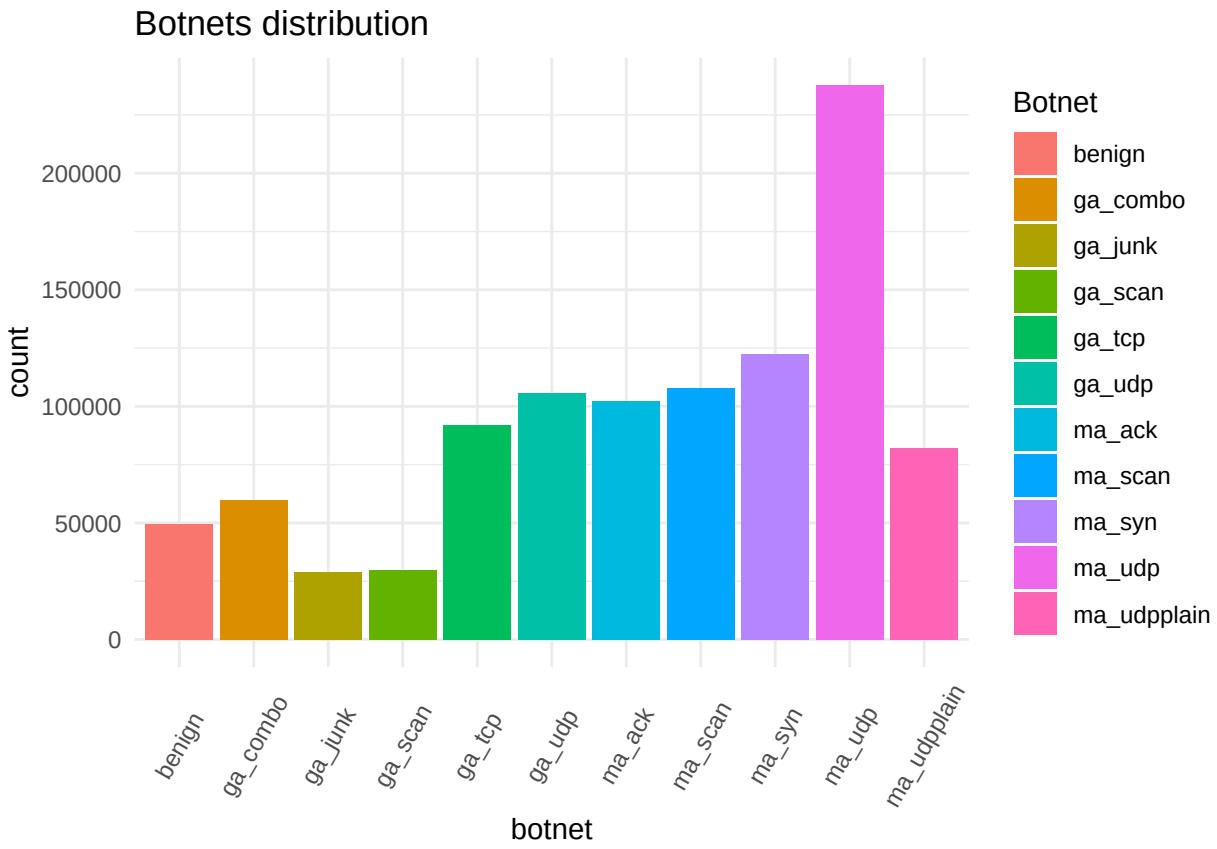
```
# Data frame dimension
dim( df )
```

```
## [1] 1018298     116
```

The data set for *Danmini doorbell* device consists of more than 1 million rows. It has 115 predictors, and the 'botnet' column (116^{th}) contains the outcome for the classification.

```
## # A tibble: 11 x 3
##    botnet           n   prop
##    <fct>        <int>  <dbl>
##  1 ma_udp      237665 0.233
##  2 ma_syn      122573 0.120
##  3 ma_scan     107685 0.106
##  4 ga_udp      105874 0.104
##  5 ma_ack      102195 0.100
##  6 ga_tcp       92141 0.0905
##  7 ma_udpplain  81982 0.0805
##  8 ga_combo     59718 0.0586
##  9 benign       49548 0.0487
## 10 ga_scan      29849 0.0293
## 11 ga_junk      29068 0.0285
```

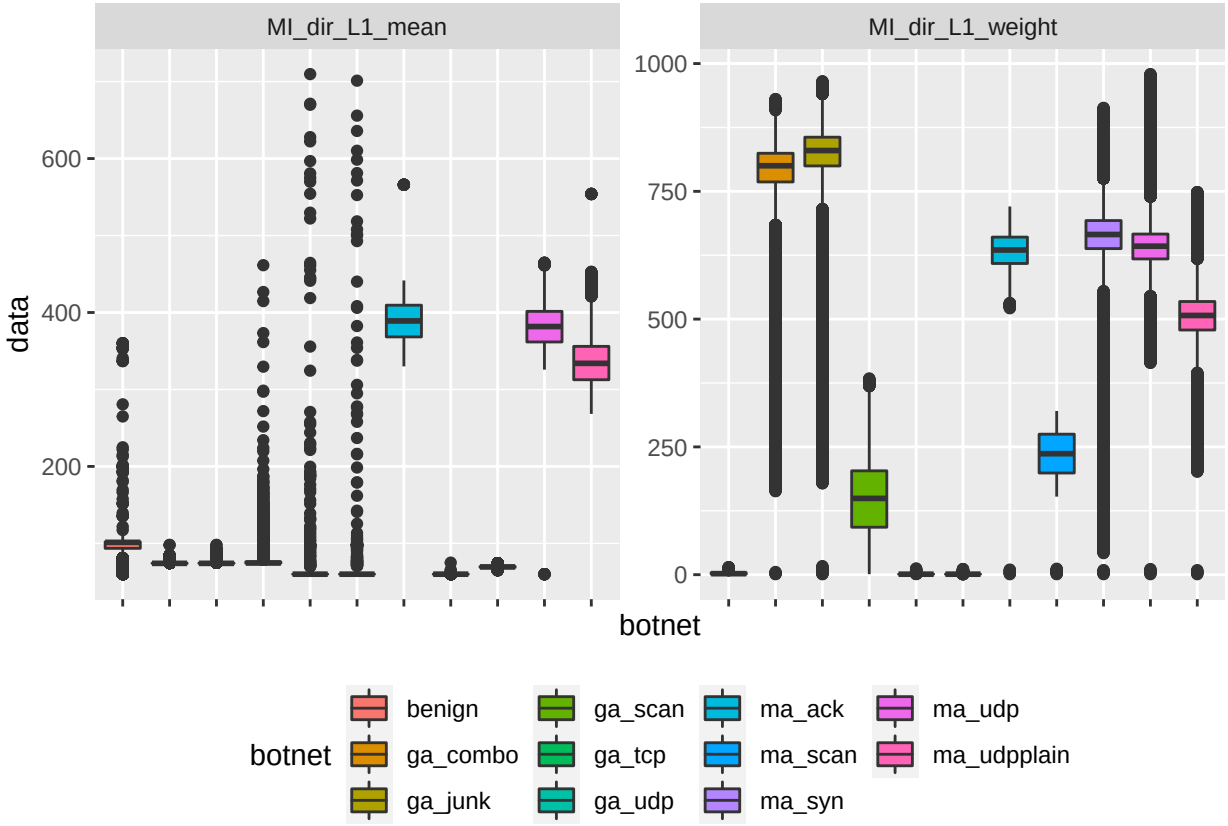
There are 11 botnets in the data set: 10 attacks plus benign traffic. Benign traffic is only 4.87% of all traffic.



The full data exploration will take a lot of pages, because the whole data set has 115 predictors. So I created *data_exploration.pdf* with full data exploration. The *data_exploration.R* script and *data_exploration.Rmd* Rmarkdown files also can be found on GitHub.

Here I just mention my conclusion from this exploration.

My research has shown that I can use *weight*, *mean* and *covariance* columns only for the classification if I need to reduce the data set dimension. This is clearly visible on small time frames such as L1.



The plot above for *MI_dir_L1_weight* column shows that almost all types of the attacks, excluding *ga_tcp* and *ga_udp*, can be separated from benign traffic, as their IQR and medians have higher values that benign traffic has. *ga_tcp* and *ga_udp* attacks camouflaging as benign traffic very well on this plot.

But the plot for *MI_dir_L1_mean* column shows that IQR and median is higher for benign traffic comparing with these parameters for *ga_tcp* and *ga_udp* attacks. So this column data will help to separate *ga_tcp* and *ga_udp* attacks from the benign traffic.

Thus, as I wrote above, *weight* and *mean* columns can be used for the attacks classification. So far as all the attacks has outliers, the information these columns contain is not enough, so *covariance* column can be used also for the classification.

Once again, the full data exploration can be found on GitHub:

- *data_exploration.pdf*
- *data_exploration.R* script
- *data_exploration.Rmd* Rmarkdown

Methods

We have learned several methods for the data classification in HarvardX PH125.9x Data Science course, such as **KNN**, **rpart** and **randomForest**.

rpart and **randomForest** are easy to use algorithms because all you need to do is only to separate the data frame into two parts (train and test set) and allow function to do its work. That's why I decided to check how well **KNN** method can perform the attacks classification also. I used two approaches for **KNN** algorithm: PCA and *train()* function from *caret* package.

The team, that collected this data frame, used 2/3 of the benign traffic to train deep autoencoder. I can't use this approach, because all learned algorithms in the course should know all possible outcomes after training. Thus, I simply divide the data set into two equal parts (train and test sets). Anyway, my aim is only to check how accurate these algorithms are in the classification.

I have

```
## [1] '4.0.2'
```

R version, and it has

```
## [1] 8060
```

MB memory limit.

When I tried to perform some algorithms on the whole original data set, I got memory allocation error. I decided to use only part of the original data set for these algorithms.

KNN

KNN (with PCA)

I use 60% of the original data set and select only columns that contain *weight* parameter. Then I separate data set into train and test sets, make them equal dimension and convert into matrices. After that I perform PCA with train set.

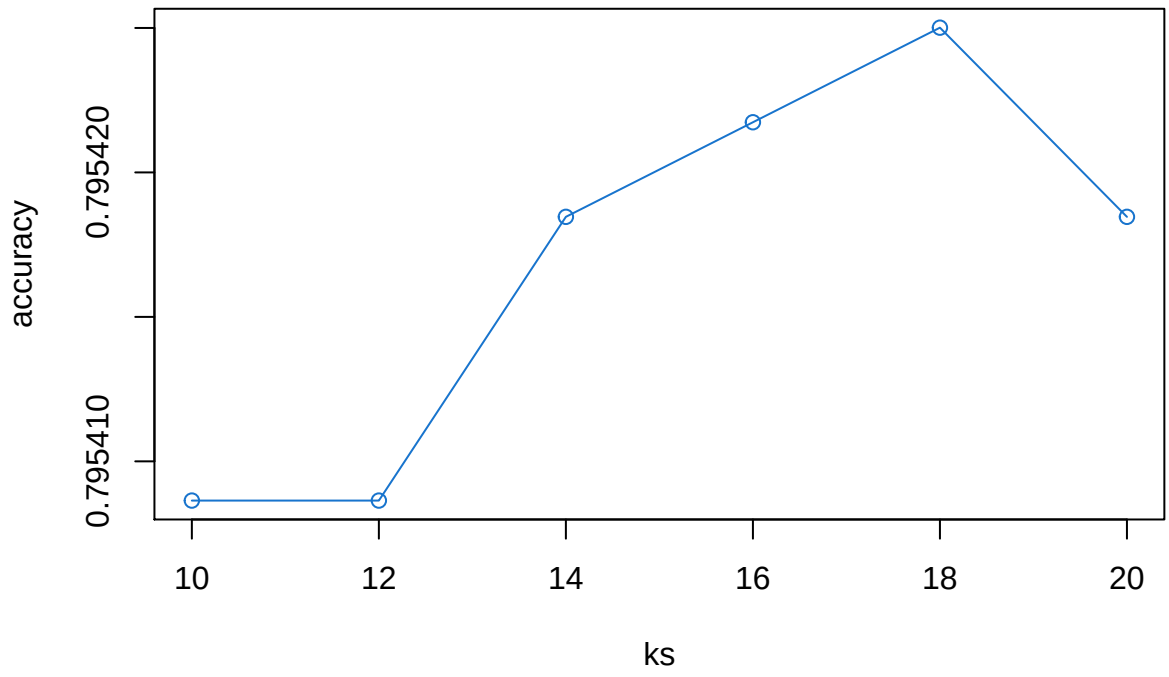
```
setSeed()

# 1. Check only columns with 'weight' attribute
pattern <- "(weight)"

# Select only columns that match the pattern. In this case - only 'weight' column
tmp <- df %>% select( botnet, grep( pattern = pattern, ignore.case = TRUE,
                                    x = colnames ) )

# As I face memory limit, I use only 'prop' of data set
prop <- 0.6
test_index <- createDataPartition( y = tmp$botnet, times = 1, p = prop,
                                   list = FALSE)
tmp <- tmp[ test_index, ]

# Create train and test sets
test_index <- createDataPartition( y = tmp$botnet, times = 1, p = 0.5,
```

```
                                    list = FALSE)
train_set <- tmp[ -test_index, ]
test_set <- tmp[ test_index, ]

# Number of rows in the train set
n_train <- nrow( train_set )

# Number of rows in the test set
n_test <- nrow( test_set )

# Make both sets equal dimension (with equal number of rows)
if( n_test > n_train ) {
  test_set <- test_set[ 1:n_train, ]
} else {
  train_set <- train_set[ 1:n_test, ]
}

# Convert train set as matrix
train_set <- train_set %>% data.matrix()

# Create levels for attacks classification
levels <- as.factor( train_set[ ,1 ] )

# Now remove 1st column, that represents 'botnet' column
train_set <- train_set[ ,-1 ]

# Convert test set as matrix
test_set <- test_set %>% data.matrix()

# Botnets outcome for the test set
bots <- as.factor( test_set[ ,1 ] )

# Now remove 1st column, that represents 'botnet' column
test_set <- test_set[ ,-1 ]

# Perform PCA
pca <- prcomp( train_set )
```

Next, I perform cross-validation to check how many principal components give maximum accuracy in the attack detection (classification). My previous research have shown that I can use from 10 to 20 principal components with step = 2 for cross-validation.

```
# Please, pay attention that this algorith is slow.

# Make cross-validation and calculate classification accuracy
ks <- seq( 10, 20, 2 )

# Prepare test set: remove all column means
col_means <- colMeans( test_set )
x_test <- sweep( test_set, 2, col_means ) %*% pca$rotation

# Please, take a note that this takes a time
accuracy <- sapply( ks, function( k ) {
```

```
  # Prepare train set
  x_train <- pca$x[ , 1:k ]

  # Train the algorithm
  fit <- knn3( x_train, levels, use.all = FALSE )

  # Prepare test set
  x_test_k <- x_test[ , 1:k ]

  # Predict
  y_hat <- predict( fit, x_test_k, type = "class" )
  cm <- confusionMatrix( y_hat, bots )
  return( cm$overall[ "Accuracy" ] )
})

# Plot k ~ accuracy
plot( ks, accuracy, type = "o", col = "dodgerblue3" )
```




```
## [1] "K =  18"
```

```
## [1] "Max accuracy =  0.795425009902222"
```

This method doesn't give significant result as its classification accuracy is only 79.54%. From other hand, I used only one *weight* column as a predictor.

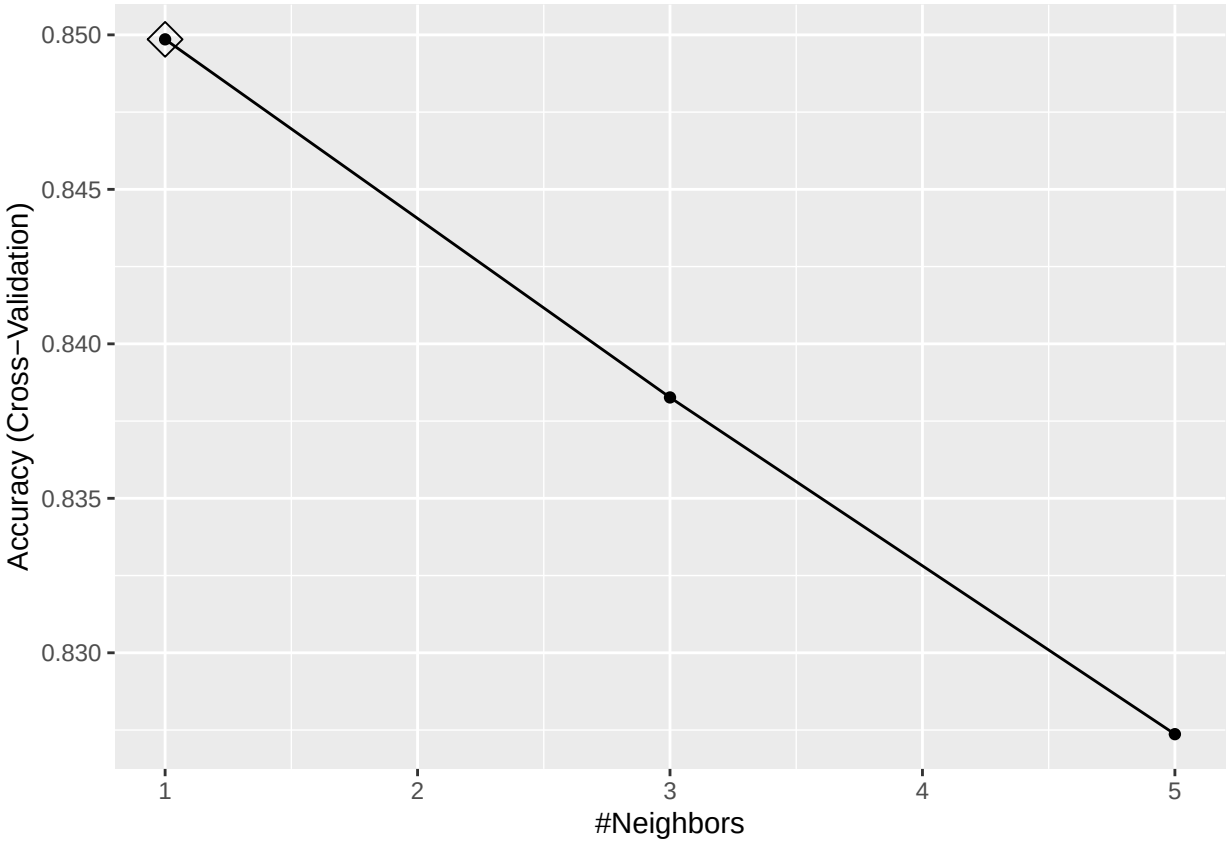
I decided not to perform the classification using *weight*, *mean* and *covariance* columns with this algorithm, because even with only selected *weight* column, it works really slow (about 2hrs with my memory limit).

After that, I decided to use *train()* function from *caret* package to check its **KNN** method.

KNN (with caret package)

I use all 115 columns as predictors and 2% of the original data set, because of memory limit. I perform 10-fold cross-validation to find better parameters for the algorithm.

```
# 2. Using train() function from caret package with "knn" method
setSeed()

prop <- 0.02
test_index <- createDataPartition( y = df$botnet, times = 1, p = prop,
                                   list = FALSE )
tmp <- df[ test_index, ]

# Create train and test sets
test_index <- createDataPartition( y = tmp$botnet, times = 1, p = 0.5,
                                   list = FALSE)
train_set <- tmp[ -test_index, ]
test_set <- tmp[ test_index, ]

# Use 10-fold cross-validation
control <- trainControl( method = "cv", number = 10, p = 0.9 )
fit_knn <- train( botnet ~ ., data = train_set, method = "knn",
                  tuneGrid = data.frame( k = seq( 1, 5, 2 ) ),
                  trControl = control )

ggplot( fit_knn, highlight = TRUE )
```



Its classification accuracy is:

```
y_hat <- predict( fit_knn, test_set, type = "raw" )
cm <- confusionMatrix( y_hat, test_set$botnet )
cm$overall[ "Accuracy" ]
```

```
##  Accuracy
## 0.8578581
```

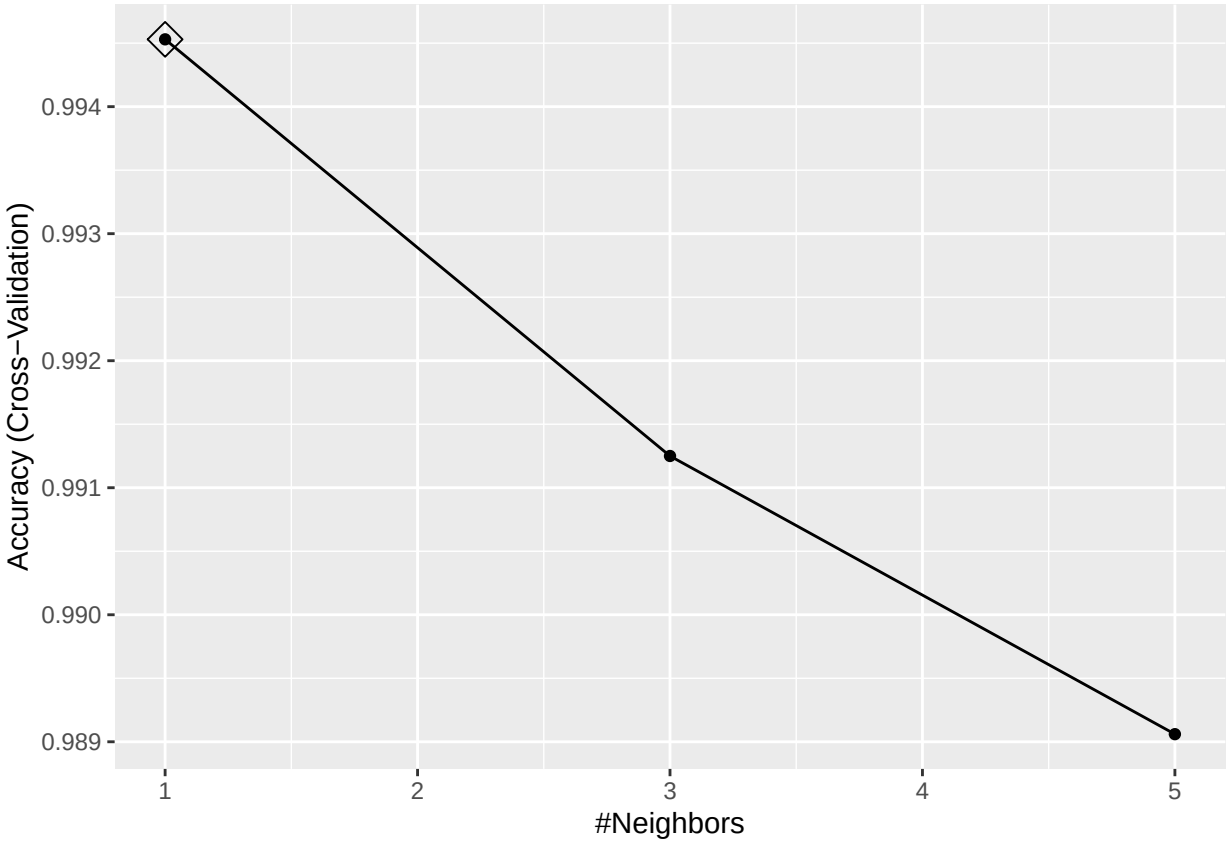
Sensitivity is the quantity, referred to as the true positive rate (TPR), while the *specificity* is also called the true negative rate (TNR). Both of them show how accurate the classification was made for each attack or benign traffic.

```
##                    Sensitivity Specificity
## Class: benign        0.9697581   0.9981426
## Class: ga_combo      0.9431438   0.9970800
## Class: ga_junk       0.9037801   0.9971706
## Class: ga_scan       0.9832776   0.9996966
## Class: ga_tcp        0.9989154   1.0000000
## Class: ga_udp        1.0000000   0.9998904
## Class: ma_ack        0.6369863   0.9567921
## Class: ma_scan       1.0000000   0.9998902
## Class: ma_syn        0.6843393   0.9587100
## Class: ma_udp        0.8081615   0.9427657
## Class: ma_udpplain   0.8158537   0.9833458
```

We have learned in the course, that k=1 may lead to overtraining, because each row in the data set was used to predict itself. In case of cyber attacks classification, I think, its reasonable to use k=1 for the higher accuracy. It's always better do not allow the attack to infect IoT device. The team that collected this data set used only benign traffic to train their autoencoder, and this spproach seams similar to use k=1 in **KNN** method.

My next step is to reduce the original data set dimension keeping only *weight*, *mean* and *covariance* columns for L1 and L0.01 time frames. This step will allow me to use 20% of the original data set with my memory limit.

```
# 3. Let's reduce data set dimension and increase data set size for classification
# 20% of data (because of memory limit), 24 predictors
setSeed()

prop <- 0.2
test_index <- createDataPartition( y = df$botnet, times = 1, p = prop,
                                   list = FALSE )
tmp <- df[ test_index, ]

pattern <- "L(0.0)?1_(weight|mean|covariance)"
tmp <- tmp %>% select( botnet, grep( pattern = pattern,
                                     ignore.case = TRUE, x = colnames ) )

# Separate to train and test sets
test_index <- createDataPartition( y = tmp$botnet, times = 1, p = 0.5,
                                   list = FALSE)
train_set <- tmp[ -test_index, ]
test_set <- tmp[ test_index, ]

# Use 10-fold cross-validation
control <- trainControl( method = "cv", number = 10, p = 0.9 )
fit_knn2 <- train( botnet ~ ., data = train_set, method = "knn",
                   tuneGrid = data.frame( k = seq(1, 5, 2 ) ),
                   trControl = control )

ggplot( fit_knn2, highlight = TRUE )
```




```
y_hat <- predict( fit_knn2, test_set, type = "raw" )
cm <- confusionMatrix( y_hat, test_set$botnet )
cm$overall[ "Accuracy" ]
```

```
##  Accuracy
## 0.9948446
```

The result shows that using only *weight*, *mean* and *covariance* columns for L1 and L0.01 time frames gives the accuracy of the classification (attack detection) about 99.48%!

Sensitivity and *specificity* show how accurate the classification was made for each attack or benign traffic:

```
##                     Sensitivity Specificity
## Class: benign         0.9917255   0.9993187
## Class: ga_combo       0.9783992   0.9987691
## Class: ga_junk        0.9580323   0.9988780
## Class: ga_scan        0.9845896   0.9996055
## Class: ga_tcp         0.9992404   0.9998596
## Class: ga_udp         0.9991500   0.9998794
## Class: ma_ack         0.9936399   0.9991814
## Class: ma_scan        0.9988857   0.9998243
## Class: ma_syn         0.9992658   0.9999107
## Class: ma_udp         0.9967181   0.9991418
## Class: ma_udpplain    0.9991462   0.9999893
```

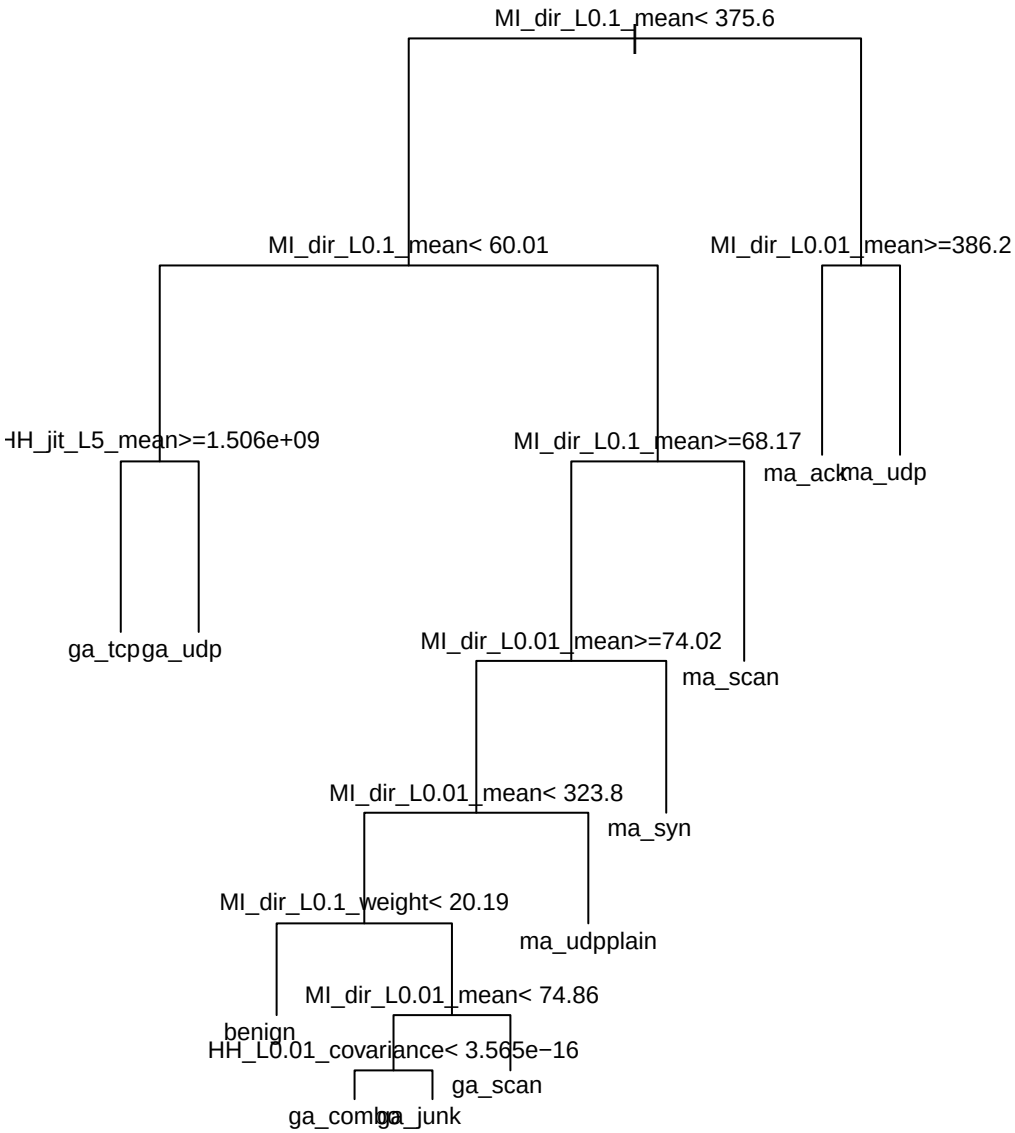
rpart

This method allows to use all 115 predictors and whole original data set even with my memory limit.

```
setSeed()

# Use all data set with all predictors
tmp <- df

# Create train and test sets
test_index <- createDataPartition( y = tmp$botnet, times = 1, p = 0.5, list = FALSE)
train_set <- tmp[ -test_index, ]
test_set <- tmp[ test_index, ]

fit_rpart <- rpart( botnet ~ ., data = train_set )
```

Visualize a decision tree:

rpart decision tree



According to the decision tree above, **rpart** mostly uses *mean* column and sometimes makes its decision using *weight* and *covariance* columns. The data exploration also has shown that using only *weight*, *mean* and *covariance* columns will be enough to detect attacks (make their classification).

```
y_hat <- predict( fit_rpart, test_set, type = "class" )
cm <- confusionMatrix( y_hat, as.factor(test_set$botnet ) )
cm$overall[ "Accuracy" ]
```

```
##  Accuracy
## 0.9865639
```

The accuracy of the attack detection (classification) using **rpart** methos is not so high and about 98.66%.

Check *sensitivity* and *specificity* for each attack or benign traffic:

```
##                     Sensitivity Specificity
## Class: benign         0.9996771   0.9998926
## Class: ga_combo       0.9978566   0.9861337
## Class: ga_junk        0.5419017   0.9998767
## Class: ga_scan        0.9997990   0.9998685
## Class: ga_tcp         0.9988713   0.9999935
## Class: ga_udp         0.9990744   0.9999912
## Class: ma_ack         1.0000000   0.9999935
## Class: ma_scan        0.9999814   1.0000000
## Class: ma_syn         1.0000000   1.0000000
## Class: ma_udp         0.9999748   0.9999923
## Class: ma_udpplain    0.9999268   0.9999915
```

randomForest

The method builds a lot of trees for each forest, so the computation time will be considerable. I use only 10% of the original data set, because of memory limit.

```
setSeed()
prop <- 0.1
test_index <- createDataPartition( y = df$botnet, times = 1, p = prop, list = FALSE )
tmp <- df[ test_index, ]

# Separate to train and test sets
test_index <- createDataPartition( y = tmp$botnet, times = 1, p = 0.5, list = FALSE )
train_set <- tmp[ -test_index, ]
test_set <- tmp[ test_index, ]

fit_rm <- randomForest( botnet ~ . , data = train_set )
y_hat <- predict( fit_rm, test_set, type = "class")
cm <- confusionMatrix( y_hat, test_set$botnet )
cm$overall[ "Accuracy" ]
```

```
## Accuracy
## 0.999784
```

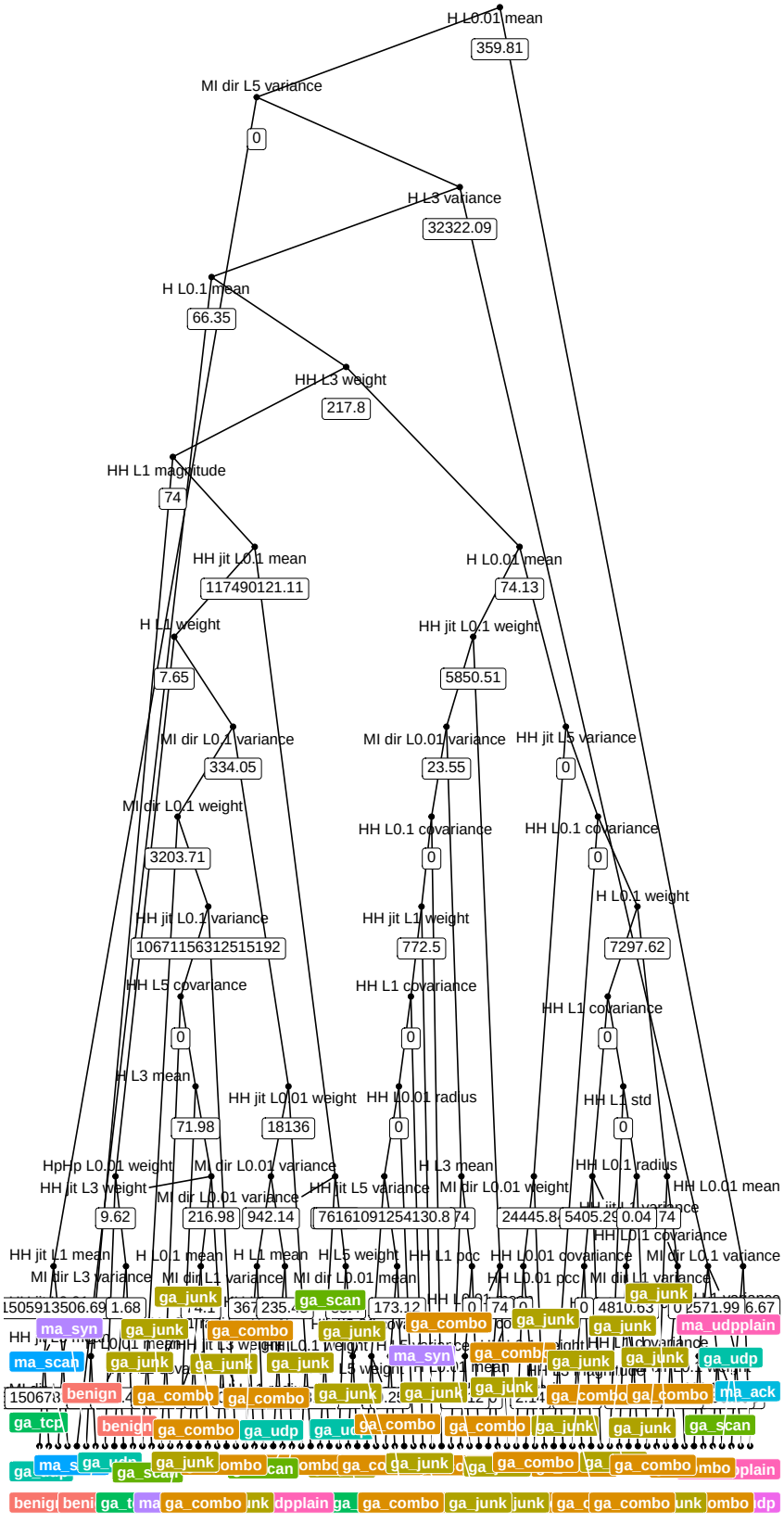
The accuracy of the classification (attack detection) is 99.98%! It's really significant result!

```
##                     Sensitivity Specificity
## Class: benign         1.0000000   0.9999174
## Class: ga_combo       1.0000000   0.9999374
## Class: ga_junk        0.9972490   1.0000000
## Class: ga_scan        1.0000000   1.0000000
## Class: ga_tcp         0.9995660   1.0000000
## Class: ga_udp         0.9994333   0.9999781
## Class: ma_ack         1.0000000   1.0000000
## Class: ma_scan        0.9998143   0.9999780
## Class: ma_syn         1.0000000   0.9999777
## Class: ma_udp         0.9999159   1.0000000
## Class: ma_udpplain    1.0000000   0.9999786
```

The *sensitivity* shows that the method has 100% TPR in some attack detection or close to 100% for other attacks. This result corralates with the result, that was obtained by the team that collected the original data set!

It will be interesting to visualize at least one decision tree from the 'forest'. The *randomForest* package has no tool for visualization, but the decision tree can be visualized using method described in *"Plotting trees from Random Forest models with ggraph"*[4] article.

[4] https://shiring.github.io/machine_learning/2017/03/16/rf_plot_ggraph


H L0.01 mean
359.81
MI dir L5 variance
0
H L3 variance
32322.09
H L0.1 mean
66.35
HH L3 weight
217.8
HH L1 magnitude
74
HH jit L0.1 mean
117490121.11
H L0.01 mean
74.13
H L1 weight
7.65
HH jit L0.1 weight
5850.51
MI dir L0.1 variance
334.05
MI dir L0.01 variance
23.55
HH jit L5 variance
0
MI dir L0.1 weight
3203.71
HH L0.1 covariance
0
HH L0.1 covariance
0
HH jit L0.1 variance
10671156312515192
HH jit L1 weight
772.5
H L0.1 weight
7297.62
HH L5 covariance
0
HH L1 covariance
0
HH L1 covariance
0
H L3 mean
71.98
HH jit L0.01 weight
18136
HH L0.01 radius
0
HH L1 std
0
HpHp L0.01 weight
HH jit L3 weight
9.62
MI dir L0.01 variance
MI dir L0.01 variance
216.98
HH jit L5 variance
942.14
H L3 mean
MI dir L0.01 weight
HH L0.1 radius
HH L0.01 mean
HH jit L1 mean
MI dir L3 variance
H L0.1 mean
H L1 mean
H L5 weight
HH L1 pcc
HH L0.01 covariance
MI dir L0.1 variance
ga_junk
ga_scan
ga_junk
ma_udpplain
ma_syn
ga_junk
ga_combo
ga_junk
ga_junk
ga_udp
ma_scan
ga_junk
ga_junk
ga_junk
ga_junk
benign
ga_combo
ga_combo
ga_junk
ga_junk
ga_combo
ga_combo
ma_ack
ga_tcp
benign
ga_combo
ga_udp
ga_combo
ga_junk
ga_junk
ga_scan

Result

I've checked how accurate **KNN**, **rpart** and **randomForest** methods can detect the cyber attacks. Here is the final result:

```
## # A tibble: 5 x 5
##   Method                       Predictors    Data_proportion Parameters Accuracy
##   <chr>                        <chr>                   <dbl> <chr>         <dbl>
## 1 KNN (with PCA)               25, (weight)             0.6  "k = 18"      0.795
## 2 KNN (with caret package)     115                      0.02 "k = 1"       0.858
## 3 KNN (with caret package, re~ 24, (L1, L0.~            0.2  "k = 1"       0.995
## 4 rpart                        115                      1    ""            0.987
## 5 randomForest                 115                      0.1  ""            1.00
```

As seen from the result table, *randomForest* method gives 100% accuracy in the attack classification. It also gives 100% TPR in detecting benign traffic and some attacks. But for other attacks it gives TPR less that 100% (but close). This means that it can't detect some attacks and may consider them as *benign traffic*. This is not good for security reason.

However, the result, I gained by this method, highly correlate with the result that got the team that collected this data set.

rpart method uses all 115 predictors and whole data set to make its decision tree, works fast, but its classification accuracy is only 98.7%.

**KNN* method on reduced data set gives the classification accuracy about 99.5%.

Conclusion

The team that collected the original data set, got 100% TPR in detecting cyber attacks. The methods we learned in HarvardX PH125.9x Data Science course are in no way inferior in the attack detection accuracy with some memory or time limits.

randomForest method gives 100% accuracy in attack classification and 100% TPR in *benign traffic* detection. It may consider some attacks as *benign traffic*, but after detecting *benign traffic* with 100% TPR, we may consider other traffic as the attacks.

The team used only *benign traffic* to train their deep autodecoder. I can't use this approach with methods we learned in the course, because they have to know all the possible outcomes after training. That's why I simply divided the data set into two parts: train and test sets.

I have memory limit of 8GB on my laptop with 4.0.2 R version. This limit allowed me to use 10% of the original data set to check the classification accuracy for *randomForest* method. However, I checked also this method on a computer with 4.0.4 R version and 32GB of memory limit. This memory limit allowed to use 60% of the original data set.

The aim that I set for my project, namely, to check with what accuracy the methods, we learned in the course, can detect attacks, has been achieved.

However, if I were further engaged in the cyber attack detection, then I would choose **randomForest** method and would take the following steps:

1. Set up higher memory limit
2. Check with varImp() function what columns are important for **randomForest** method to create its decision trees.

3. Reduce the data set dimension using variable importance.

All these steps will allow me to use all rows of the original data set. So far as **randomForest** method gives 100% accuracy in attack detection even on 10% of the original data set, all these steps above will not increase its accuracy. However, these steps can increase *sensitivity* in the attack detection.

Apendix

Data set structure

```
# Data frame structure
str( df )
```

```
## function (x, df1, df2, ncp, log = FALSE)
```