Ensemble Methods: Intrusion Detection

Vamshidhar Pandrapagada

April 20, 2017

1.Overview

The goal of this project on model optimization using ensemble methods is:

1. To detect intrusion from a complex, unbalanced data set that has ~500 thousand observations, 42 features, 23 classes; with highly unequal distribution of observations across different classes.
2. To design and optimize an ensemble classifier using bagging, boosting, and random forest techniques
3. To validate and evaluate the performance metrics relevant to the problem.

2.Load the required libraries

Load the required R libraries

```
library(data.table)
library(dplyr)
library(ggplot2)
library(caret)
library(gridExtra)
library(C50)
library(adabag)
library(randomForest)
library(e1071)
```

3.Get the Data

The data set is in 4 parts

1. **intrusion.detection.csv** - This CSV file contains the actual data for us to examine and build a model which performs intrusion detection.
2. **feature.names** - names of each feature in intrusion.detection.csv. We will use this file to assign feature names to the data set once it is read.
3. **feature.names.types** - Indicates the type of the feature (Continuous vs Symbolic).
4. **attack_types** - type of the attack or intrusion. Based on the patterns in the feature set, our model will predict one of these attack types.

For our convenience,all these files will be placed in the data folder under the current working directory. The code snippet below does the following:

1. Set the working directory.
2. Create a relative path to read data sets under the working directory.
3. Read the data into intrusion_Details data frame and assign column names.

```
wd <- "D:/My Folder/Data Science and Big Data/Cisco Data Science Training/Level 1/Projects/ensemble-meth
setwd(wd)

relativeFilePath <- paste('./data/','intrusion.detection.csv', sep = "",collapse = "")
relativeFeaturePath <- paste('./data/','feature.names.types.txt', sep = "",collapse = "")
```

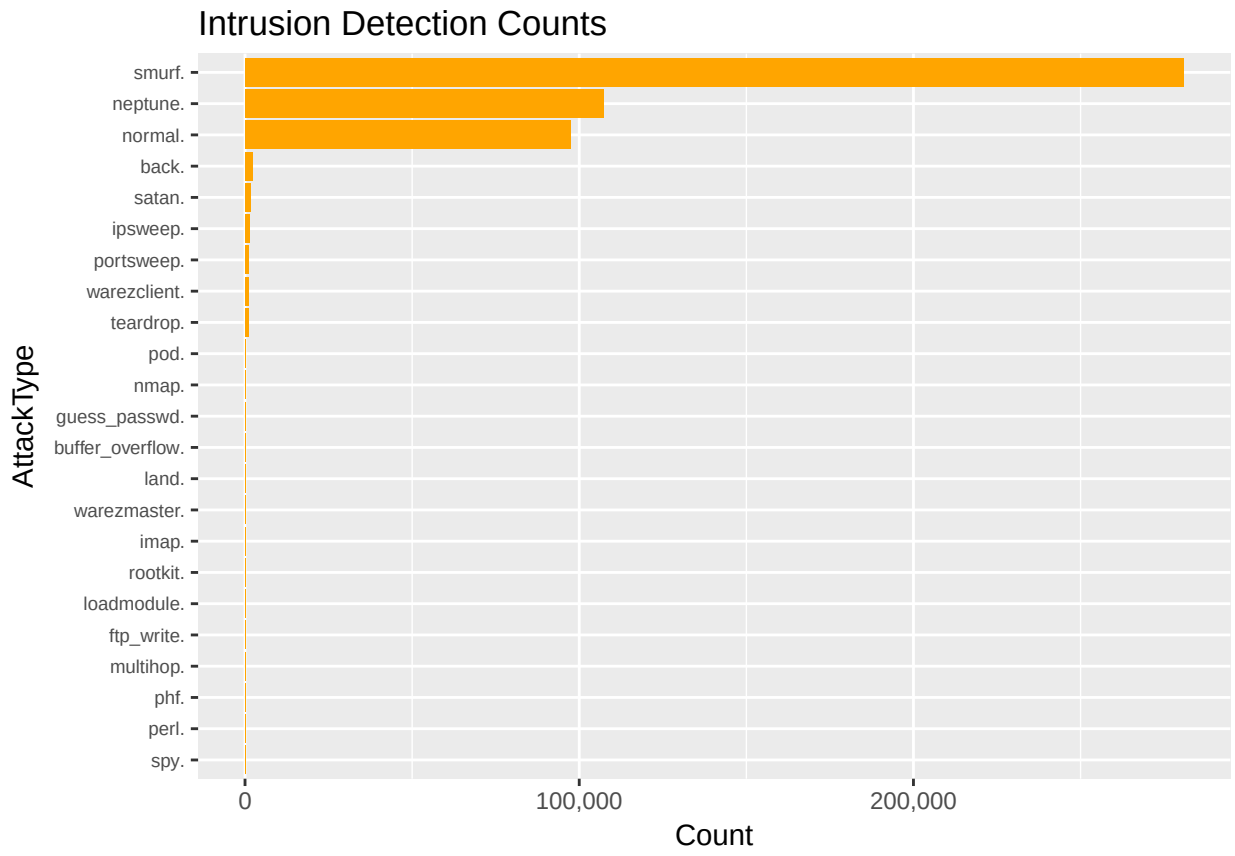
```
intrusion_Details <- read.csv(relativeFilePath, header=T, stringsAsFactors = F)
featureNames <- read.table(relativeFeaturePath, header=T, sep=',', stringsAsFactors=F)

#Assign column names to the data frame
colnames(intrusion_Details) <-c(as.vector(featureNames[ ,1]),"Type")
```

4.Exploratory Data Analysis

Now let's look at how our Classes are distributed over the entire data set. This gives us a general idea on how we split the data into Training, testing and validation data sets.
The idea here is, if any one of the classes in the training data is very rare, and when we draw bootstrap samples on it, we may encounter samples in which the rare class may never appear. When that happens , the model we build will not be able to make predictions when it encounters an observation with a value for the class it has never seen before.

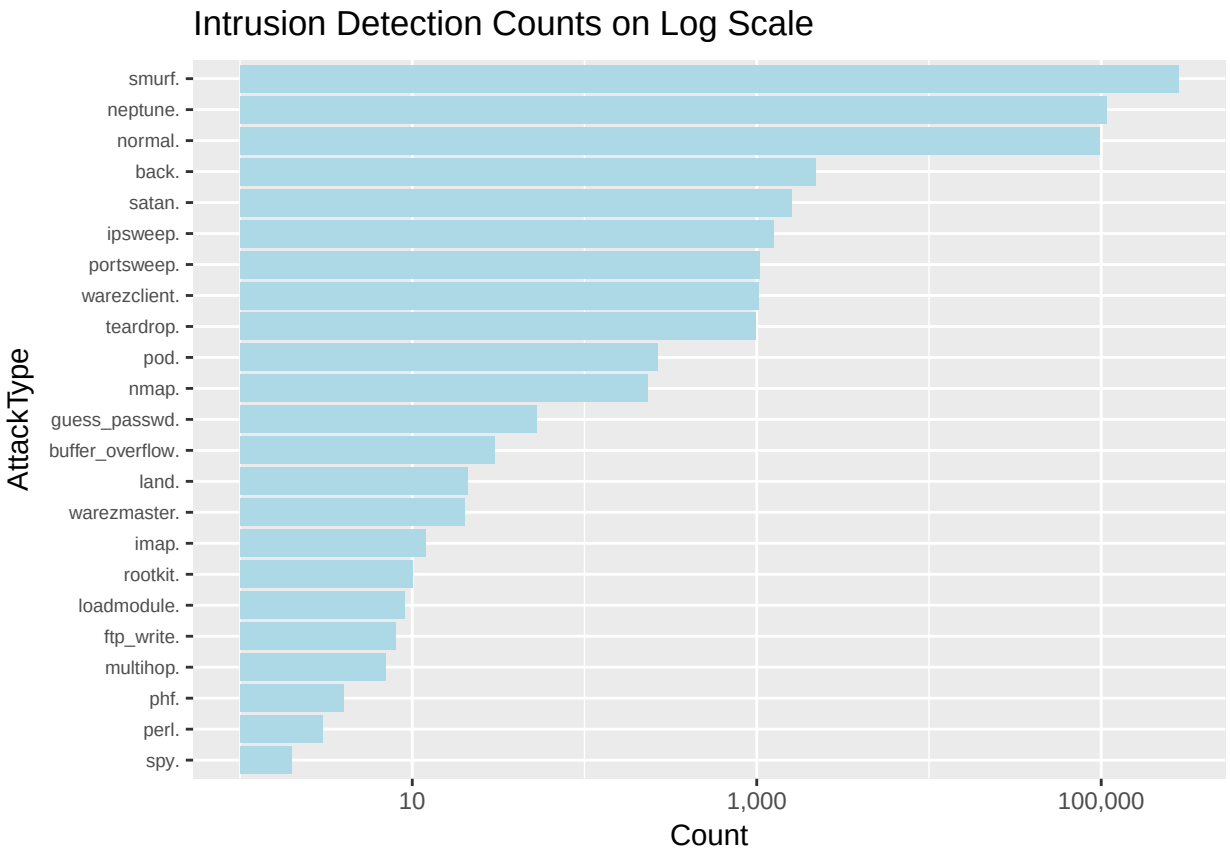
The following code gets the count of each Attack type in descending order and creates a bar plot.

```
#type_count <- intrusion_Details %>% group_by(Type) %>% summarize(counts = length(Type))

#Reorder data by top attack types
topAttacks <- as.data.frame(table(intrusion_Details$Type))
colnames(topAttacks) <- c("AttackType","Count")
topAttacks <-transform(topAttacks, AttackType = reorder(AttackType, Count))

g1<- ggplot(topAttacks) + geom_bar(aes(x= AttackType, y=Count),stat ="identity",fill="orange")+ coord_fl
     scale_y_continuous(labels=scales::comma) +
     ggtitle('Intrusion Detection Counts') +
     theme(axis.text.y=element_text(size = rel(0.8)))
g1
```



Visually, it is very evident that there are certain intrusion detection types which are very low in number. The data range is very wide and the mass of the distribution of classes is heavily concentrated on one side and it is very difficult to see the actual numbers of the classes which are very low. Plotting the graph on LOG scale will show some more details.

```
g2<- ggplot(topAttacks) + geom_bar(aes(x= AttackType, y=Count),stat ="identity",fill="lightblue")+ coor
     scale_y_log10(labels=scales::comma) +
     ggtitle('Intrusion Detection Counts on Log Scale') +
     theme(axis.text.y=element_text(size = rel(0.8)))
g2
```



For this exercise, lets filter the data set to retain the top 3 attack types and build ensemble models on top of the resultant data set.

The following code removes all the rows which are not in the top 3 attack types.

```
top3Attacks <- as.character(topAttacks[order(topAttacks$Count, decreasing = TRUE), ][1:3,]$AttackType)
intrusion_Details_top3 <- intrusion_Details[intrusion_Details$Type %in% top3Attacks,]

dim(intrusion_Details_top3)
```

```
## [1] 485268     42
```

Out of Total number of rows in 494020, top 3 attack types constitute 485268 rows.

Exclude un-necessary features from the data frame

As in every data set, it is sometimes needed to plug out features which do not add weight to the model. After initial analysis, we have decided to filter out the following features.

1. protocol_type
2. service
3. flag
4. is_host_login
5. num_outbound_cmds

```
excludeFeatures <- list('protocol_type','service','flag','is_host_login','num_outbound_cmds')
retainFeatures <- setdiff(colnames(intrusion_Details_top3), excludeFeatures)
intrusion_Details_top3 <- intrusion_Details_top3[,retainFeatures]
```

```
intrusion_Details_top3$Type <-as.factor(intrusion_Details_top3$Type)
dim(intrusion_Details_top3)
```

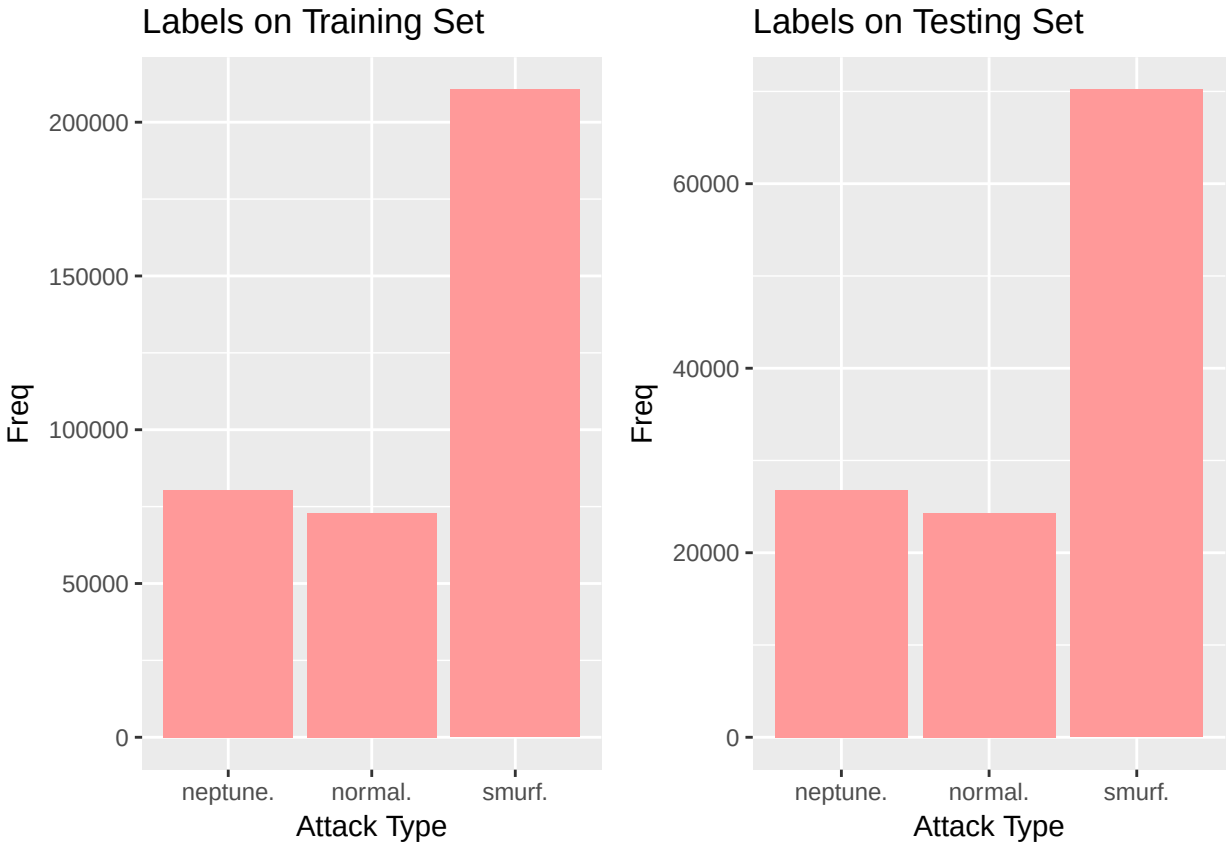
```
## [1] 485268     37
```

Now that our data frame is ready, lets starting building the models

5.Split the data into training and testing sets using functions in the caret package

The following code splits the data into training and testing sets. As a usual norm, we reserve 25% of the data for testing and use the remaining 75% for training our models.

```
indexes <- createDataPartition(intrusion_Details_top3$Type, p=0.75, list=FALSE)
training <- intrusion_Details_top3[indexes, ]
testing <- intrusion_Details_top3 [-indexes, ]
plot_table <-as.data.frame(table(training$Type))

g1<- ggplot(plot_table, aes(x=Var1, y=Freq)) + geom_bar(stat="identity", fill="#FF9999") + ggtitle("Labe
plot_table <-as.data.frame(table(testing$Type))
g2<- ggplot(plot_table, aes(x=Var1, y=Freq)) + geom_bar(stat="identity", fill="#FF9999") + ggtitle("Labe
grid.arrange(g1,g2, ncol=2)
```



The class label distribution is uniform in training and testing sets. This is a good sign and an advantage of splitting the data using createDataPartition function from caret package.

6.Ensemble Model 1: Bagging

Bagging also called as **Bootstrap Aggregation** is a procedure which draws bootstrap samples (with replacement) from the data set and models are built by subsetting the data set based on these samples. From the original data set, the observations that were not chosen in a bootstrap sample for a particular iteration of the procedure are referred to as Out of bag (OOB) samples. These observations are not used in the training of the model at that iteration. We can rely on OOB samples to record the accuracy of the model.

Final prediction is made by using majority voting for classification models and average value for regression models.

Though bagging is frequently explained in context with decision trees, in general bagging can be used with any of the modeling techniques (e.g. Logistic regression, SVMs etc).

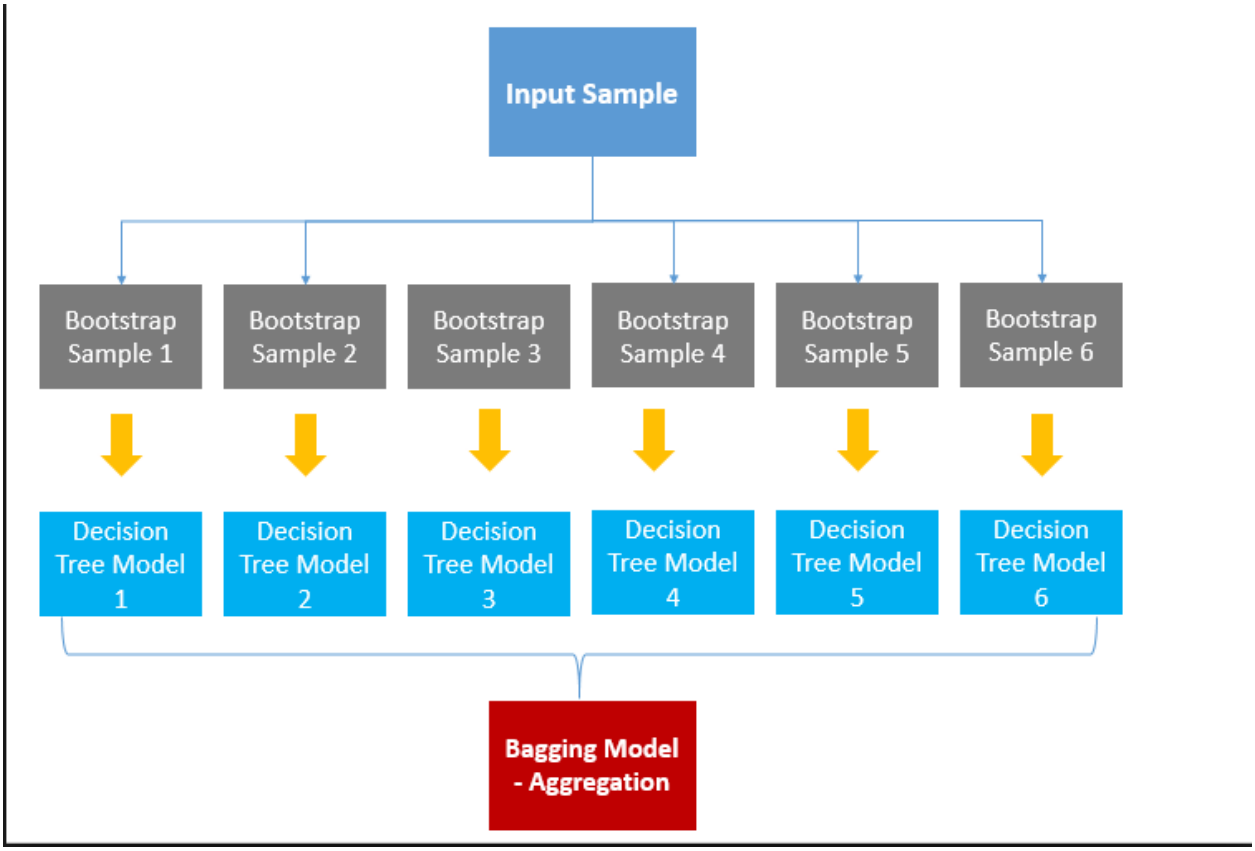


Figure 1: Image Source : http://dni-institute.in/blogs/bagging-algorithm-concepts-with-example/

Step 1: Set the seed and collect the bootstrap samples.

It is always a good idea to build odd number of models with bagging so that there is no tie during the majority voting process.

```
noModels <-11
seeds <- 125678: (125678 +noModels - 1)  #125678 is any random seed
nTrain <- nrow(training)
bagsize <- nTrain
bootSamples <- sapply (seeds, function(x){
                                        set.seed(x)
```

```
                                          sample(nTrain, bagsize, replace = T)
                                        }

                      )
dim(bootSamples)
```

```
## [1] 363952     11
```

Step 2: Build the models based on bootstrap samples.

In this exercise, Decision tree implementation **C5.0** method from **C50 Package** is used.
C5.0 method , instead of giving a probability while predicting, it preserves the actual class name as predicted value.

```
train_dtree <- function(samples){
  subsetted <- training[samples,]
  dtrees <- C5.0(Type ~., data = subsetted)
  return(dtrees)
}
trees <- apply(bootSamples, 2, train_dtree)
```

Step 3: Collect the Out of the Bag samples.

Out of the Bag samples are the rows which were not selected at a particular iteration. We will use these OOB samples to predict the out-of-bag accuracy of our model. To be able to identify the row number of the observation from the training set, lets add a new column ID (which is the original row index) to the OOB subsetted data frame.

```
out_of_bag <- function(samples)
{
  oob_indices <-setdiff(1:nrow(training), unique(samples))
  oob_subsetted <- training[oob_indices,]
  oob_subsetted$ID <-oob_indices
  return(oob_subsetted)
}
oobags<-apply(bootSamples,2,out_of_bag)
dim(oobags[[3]])
```

```
## [1] 134134     38
```

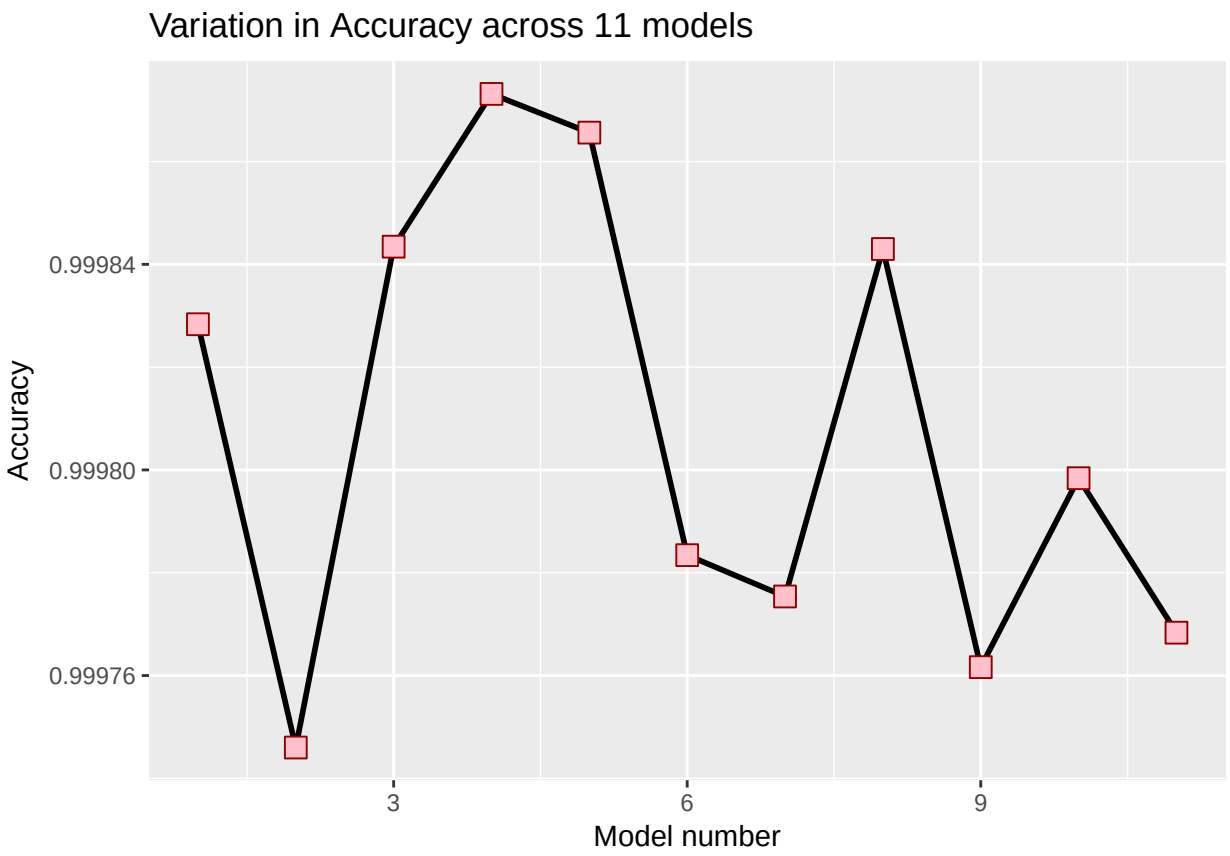
You can see that, in iteration 1, 133992 rows were not selected are are called OOB samples.

Step 4: Build a function for Predictions.

We now have list of models, list of data frames with OOB observations. From these two, let's create a list of predictions. Plot the variations in Out of the bag accuracy across different models.

```
cmatrix <- data.frame()
predictions <- function(model, data, model_number)
{
  colname <- paste("PREDICTIONS",model_number)
  data[colname] <- predict(model, data)
  cm <- confusionMatrix(predict(model, data), data$Type  )
  cmatrix <<- rbind(cmatrix, c(model_number, cm$overall[1]))
  return (data[,c ("ID", colname),drop = FALSE])
```

```
}

OOB_predictions <- mapply(predictions, trees, oobags, 1:noModels, SIMPLIFY = F)
colnames(cmatrix) <- c("Model_number","Accuracy")
ggplot(cmatrix, aes(x = Model_number, y= Accuracy)) + ggtitle("Variation in Accuracy across 11 models")
  xlab("Model number") + geom_line(size=1) + geom_point(size=4, shape=22, colour="darkred", fill="pink")
```



Step 5: Merge All Predictions into a single Data frame and get the predition using max VOTE.

Remember, we have created a new column ID on all OOB data frames, which stores the row number of the OOB sample from original training data set. These rows are the samples which were not selected during iterations. Let's use this ID column to merge all the training predictions.

The Following Code

1. Uses Reduce() function along with merge()function in order to merge all the OOB_predictions data frame.
2. Compute the predicted value using majority vote across 11 different models.
3. Calculate mean accuracy by comparing the prediction with actual Value.

```
final_oob_predictions <- Reduce(function(x,y) merge(x,y,by= "ID", all=T),OOB_predictions)
max_vote_pred <- apply(final_oob_predictions[,-1], 1, function(x) names(which.max(table(x))))
mean_accuracy <- mean(max_vote_pred == training$Type[as.numeric(final_oob_predictions$ID)],na.rm=TRUE)
mean_accuracy
```

```
## [1] 0.9998147
```

Mean OOB Accuracy obtained using bagging with 11 bootsrapped Bags is : 0.9998147 .
OOB Accuracy is a much better measure of performance on unseen data.

Step 6: Calculate Test Accuracy.

Now let's get the predicted values by calling predict function using trees and testing data set. Finally we calculate the mean accuracy by taking the MAX VOTE logic. Also print the classification table.

```
final_test_predictions <- sapply(trees , function(x) { predict(x, testing)})
test_max_vote_pred <- apply(final_test_predictions, 1, function(x) names(which.max(table(x))))
test_mean_accuracy <- mean(test_max_vote_pred == testing$Type,na.rm=TRUE)
test_mean_accuracy
```

```
## [1] 0.9998269
```

```
table(test_max_vote_pred, testing$Type)
```

```
##
## test_max_vote_pred neptune. normal. smurf.
##           neptune.    26795       1      0
##           normal.         5   24318     15
##           smurf.          0       0  70182
```

Mean Test Accuracy obtained using bagging with 11 bootsrapped Bags is : 0.9998269

7.Ensemble Model 2: Boosting

Boosting is another alternative to combine different models on the same data set and achieve greater performance. This model is well suited for weak learners.
Boosting starts with a weak classifier and gradually improves it by re-weighting the mis-classified sample. This is an iterative procedure to adaptively change distribution of training data by focusing more on previously mis-classified records.

- Records that are wrongly classified will have their weights increased.
- Records that are classified correctly will have their weights decreased.

The main difference between **Bagging** and **Boosting** is that bagging combines independent models and each model is trained using a different bootstrap sample. Whereas boosting performs an iterative process by giving more weights to mis-classified observations and reduce the error of preceding models by predicting them with successive models.

There are several types of Boosting techniques. But in this example, we will use **AdaBoost** which is short for **Adaptive Boosting**.
adabag package provides an implementation of **AdaBoost* algorithm.

Step 1: Use *boosting* method from *adabag* to perform boosting.

The following code

1. Uses boosting method from *adabag* package.
2. *mfinal* is the parameter used to specify the number of trees. We create 11 Trees here as well.
3. *coeflearn* If 'Freund', error function alpha=ln((1-err)/err) is used.
4. *boos* if TRUE (by default), a bootstrap sample of the training set is drawn using the weights for each observation on that iteration. If FALSE, every observation is used with its weights.
5. *control* is rpart control which uses a decision tree fit with a maximum depth of 3.

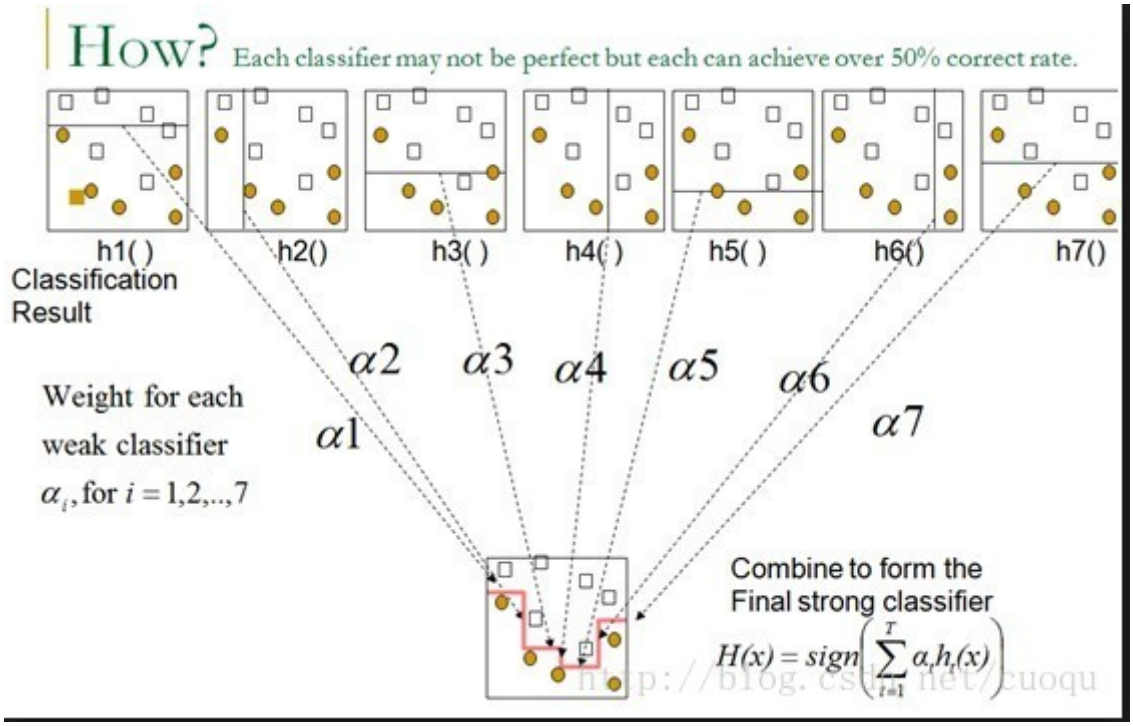


Figure 2: Image Source : http://www.kdnuggets.com/2014/09/advanced-data-analytics-business-leaders-explained.html

```
modFit <- boosting(Type ~.,data=training,mfinal=11, coeflearn="Freund", boos = FALSE,   control=rpart.c
```

Step 2: Use *predict.boosting* function to predict the results on testing set.

The following code

1. Uses *predict.boosting* method to predict the results in test set.
2. Prints the Confusion matrix and the error.

```
boostPred <- predict.boosting(modFit,newdata=testing)
boostPred$confusion
```

```
##                Observed Class
## Predicted Class neptune. normal. smurf.
##         neptune.    26800       1      0
##         normal.         0   24318      1
##         smurf.          0       0  70196
```

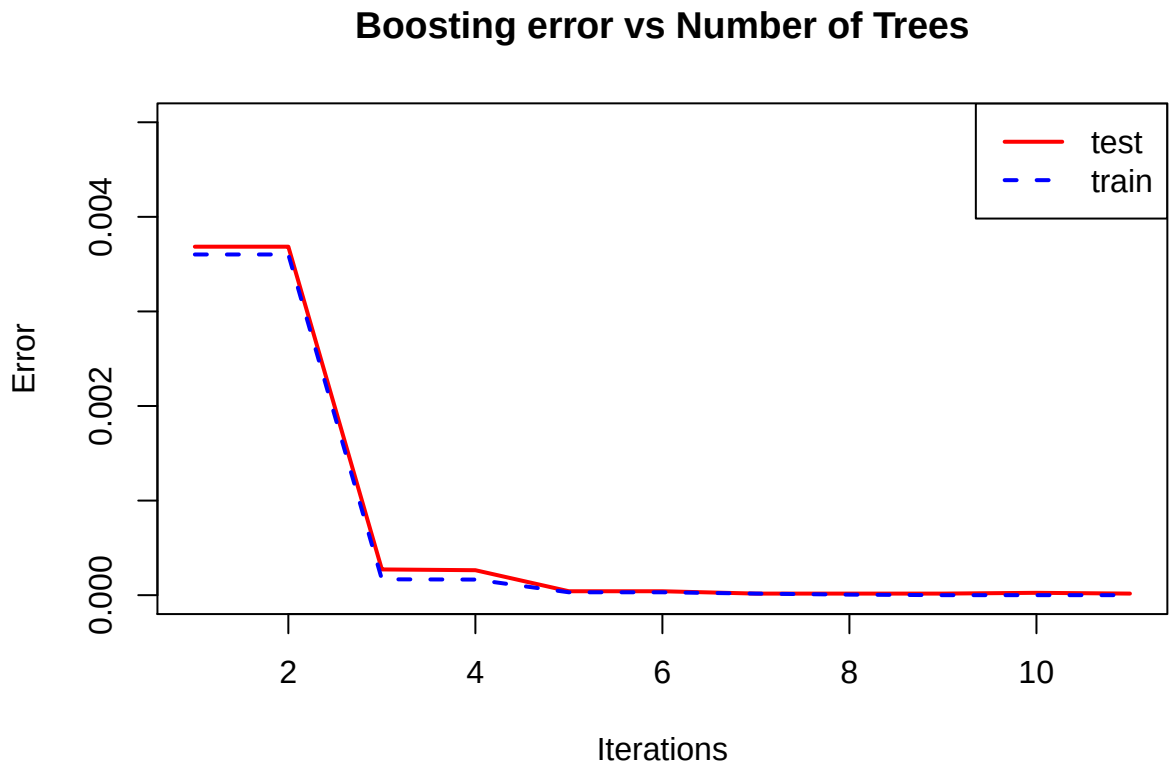
```
boostPred$error
```

```
## [1] 1.648587e-05
```

Step 3: Calculate error evolution of the ensemble method.

adabag package provides a method call *errorevol* which allows the user to estimate the errors in accordance to the number of iterations. Error evolution can be performed on both Training and Testing data sets.

```
trainEvol <- errorevol(modFit, training)
testEvol <- errorevol(modFit, testing)
plot(testEvol$error, type= "l", ylim = c(0,0.005), main ="Boosting error vs Number of Trees", xlab = "I
ylab ="Error", col="red", lwd =2)
lines(trainEvol$error, cex = 0.5, col ="blue", lty = 2, lwd = 2)
legend("topright", c("test", "train"), col =c("red","blue"), lty = 1:2, lwd = 2 )
```



Observe that the error rate starts to decrease on both testing and training data sets as we progress through the iterations.

8.Ensemble Model 3: Random Forest

Random Forest is a type of another ensemble learning method that grows multiple decision trees during the training process.
Random forests provide an improvement over bagged trees by de-correlating the trees. In bagging, trees are built using bootstrap samples with all the features included every time. But whereas in Random forest, each time a tree is built (also using bootstrapped samples), a random sample of m *predictors* are chosen as split candidates from the full set of p *predictors*. At each split, a new or fresh sample of m *predictors* are chosen. Typical norm to chose the number of predictors is the $m = \sqrt{p}$

Step 1: Use Random forest to fit the model.

The following code:

1. Uses randomForest method from randomForest package.

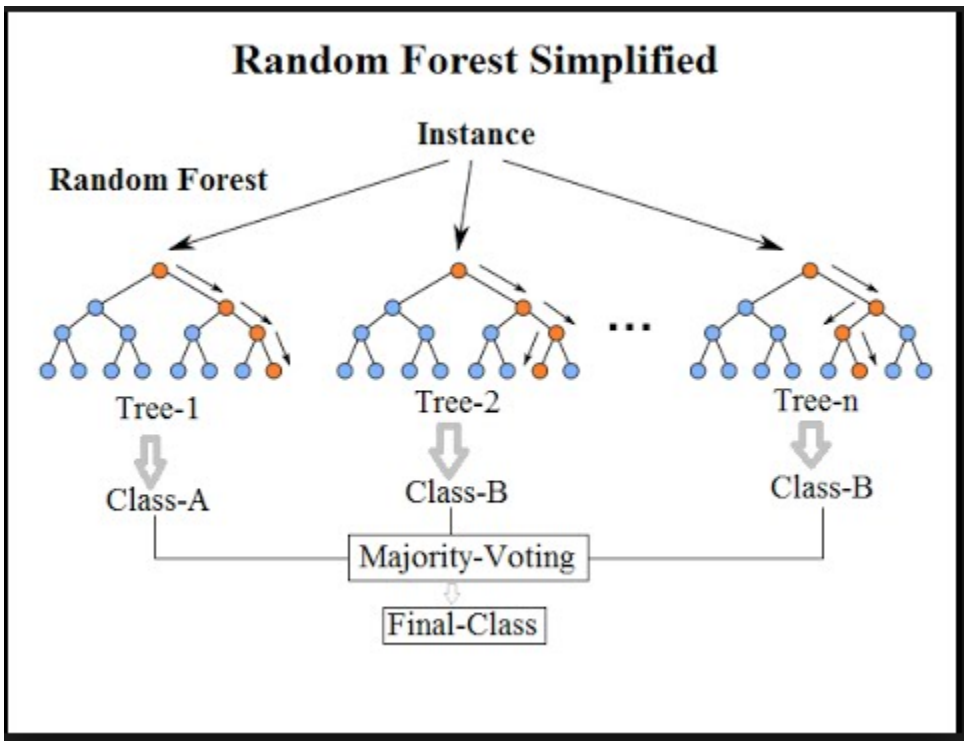


Figure 3:

2. *ntree* is the parameter to specify the number of trees we build in the random Forest.Lets use 11 trees here as well.
3. *mtry* is the number of features sampled for use at each node for splitting.
4. We have 37 features in our data set. Square root of 37 is ~ 6. So let's fit the model using *mtry* value as 6.

```
rfFit <- randomForest(Type ~. , data = training, ntree = 11, mtry =6, importance = T)
```

Step 2: Predict, Calculate Test Accuracy and Variable importance.

After the model is Fit,

1. Use *predict* function predict the values on Testing data set.
2. Print the confusion matrix table to print stats and gather accuracy.
3. Random Forest model allows us to examine the variable importance. This can be done using *importance* function

```
predictions <- predict (rfFit, testing)
confusionMatrix(predictions, testing$Type)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction neptune. normal. smurf.
##   neptune.    26799       1      0
##   normal.         1   24318      2
##   smurf.          0       0  70195
##
## Overall Statistics
```

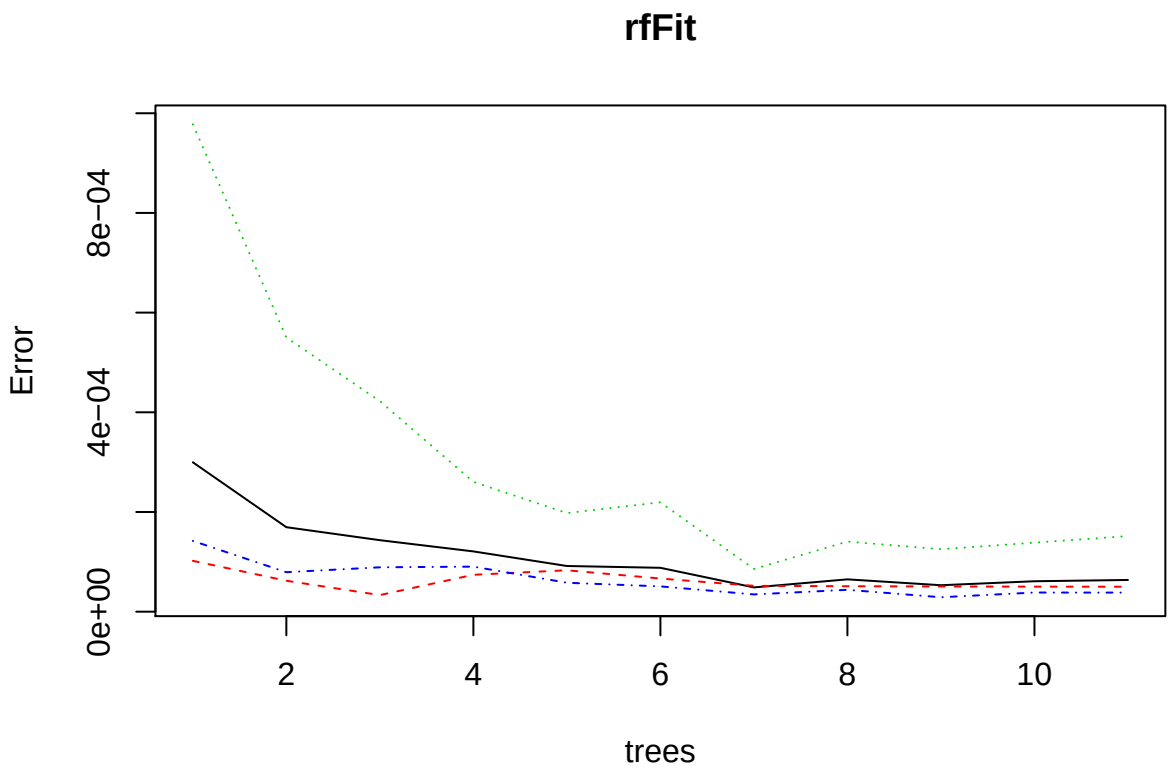
```
##
##                Accuracy : 1
##                  95% CI : (0.9999, 1)
##     No Information Rate : 0.5786
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9999
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: neptune. Class: normal. Class: smurf.
## Sensitivity                   1.0000         1.0000        1.0000
## Specificity                   1.0000         1.0000        1.0000
## Pos Pred Value                1.0000         0.9999        1.0000
## Neg Pred Value                1.0000         1.0000        1.0000
## Prevalence                    0.2209         0.2005        0.5786
## Detection Rate                0.2209         0.2005        0.5786
## Detection Prevalence          0.2209         0.2005        0.5786
## Balanced Accuracy             1.0000         1.0000        1.0000
```

Step 3: Plot Variable importance and mean square error of the forest object.

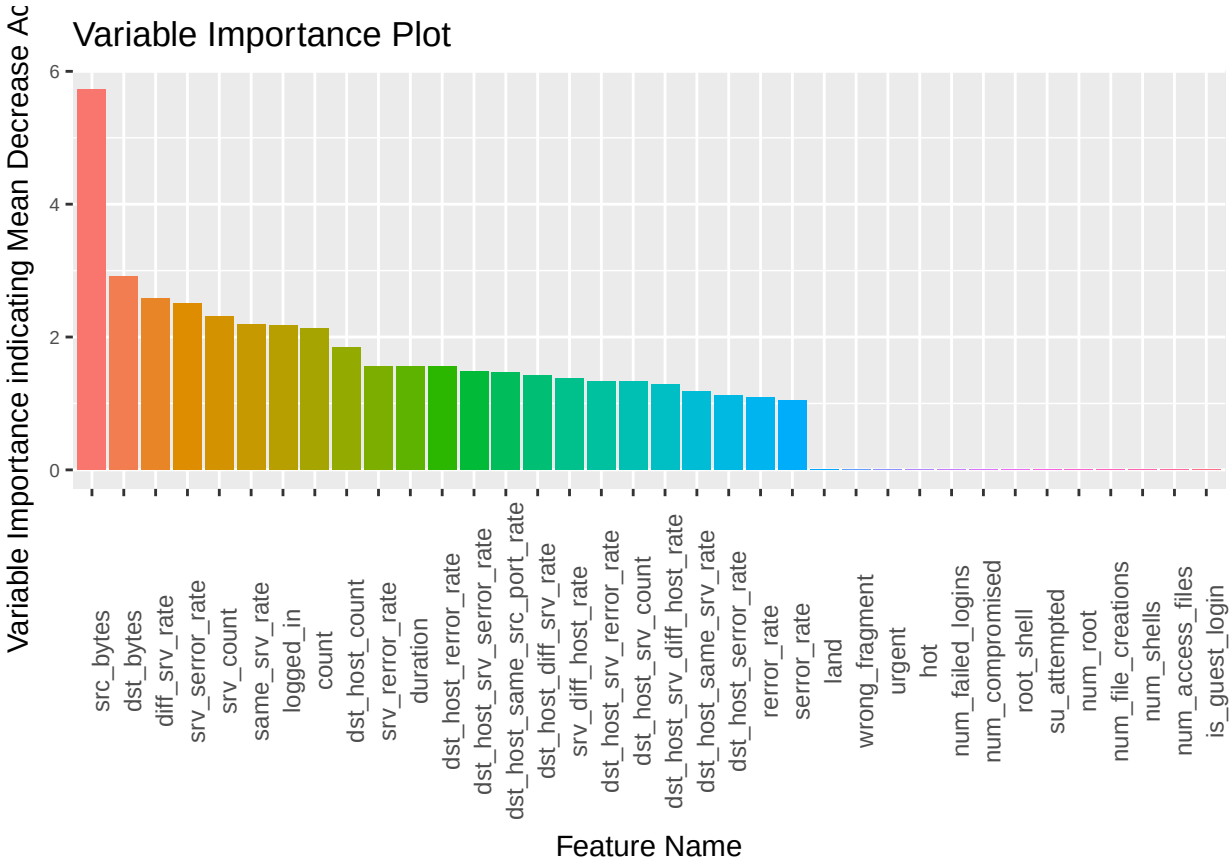
The following code:

1. Uses *plot* function to plot the mean square error of the forest object. This plot shows error variance for each tree that is fit inside the forest.
2. Uses *ggplot* function to obtain the plot of variable importance. Optionally, we can also use *varImpPlot* to plot a very basic version.

```
plot(rfFit)
```




```
importanceTable <-data.frame(feature = setdiff(colnames(testing), "Type"),
                             important_feature = as.vector(importance(rfFit)[,4]))

importanceTable <- arrange(importanceTable, desc(important_feature))
importanceTable$feature <- factor(importanceTable$feature,levels=importanceTable$feature)
implot <- ggplot(importanceTable, aes(x=feature, y=important_feature, fill = feature)) +
          geom_bar(stat = "identity") + ggtitle("Variable Importance Plot") +
          xlab("Feature Name") +
          ylab("Variable Importance indicating Mean Decrease Accuracy)") +
          scale_fill_discrete(name="Variable Name") +
          theme(axis.text.y=element_text(size = rel(0.8))) +
          theme(axis.text.x=element_text(angle = 90)) +
          theme(legend.position="none")


implot
```



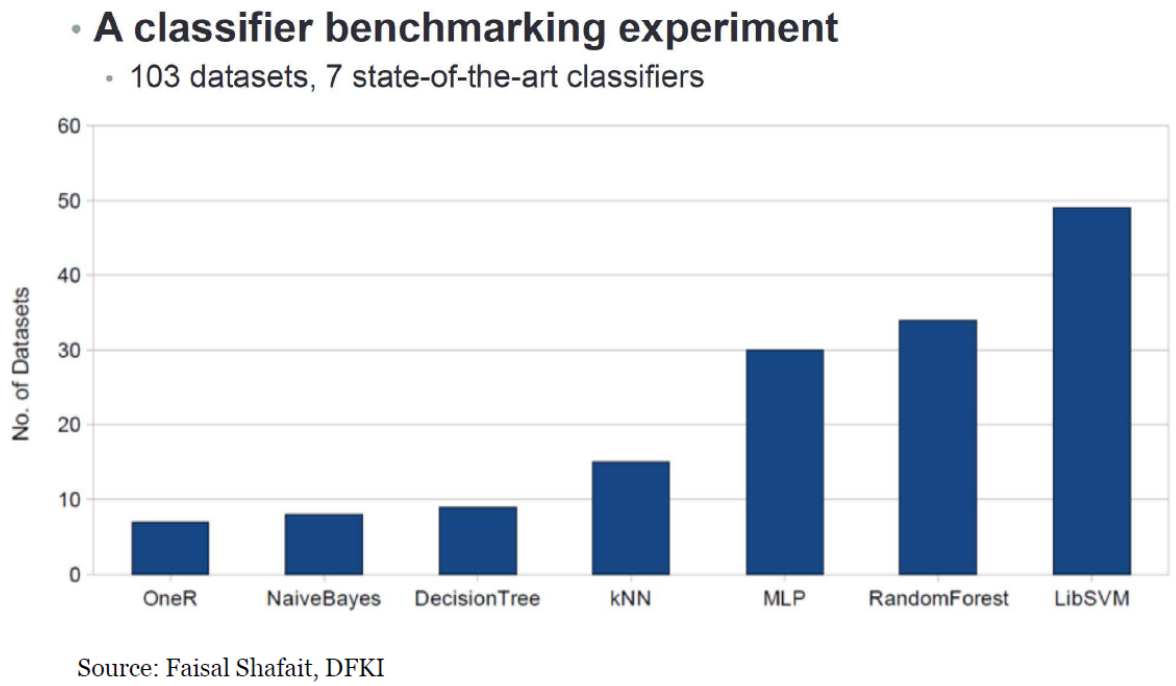
Variable importance plot shows that **src_bytes, dst_bytes and count** are the top 3 features which play a key role in determining the attack type. In this case for example, excluding **src_bytes** variable from the model decreases the overall accuracy by 30%.

9.Conclusion

For this particular data set, all the above 3 models performed fairly well. But by looking at the accuracy of each model, Random Forest won by a very low margin with accuracy almost equal to one without any False Positives or False Negatives.

Also in general, out of all the state of art classifiers, Random Forest is a winner majority of the times. This is a very powerful algorithm based on bagging decision trees, with an exception that a different feature sample is randomly selected during the construction of every new tree. By doing this, the model reduces the correlation between trees, captures significant localized variations in the output and improves the degree of variance reduction in the final result.

Random forest also scales well with a larger number of input features. Below is the snapshot of all the state of the art classifiers and bench marking experiment (image borrowed from lecture notes).



Source: Faisal Shafait, DFKI

10.References

1. Rui Miguel Forte (2015): Mastering Predictive Analytics with R.
2. Nina Zumel, John Mount (2014): Practical Data Science with R.
3. Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani (2015): An Introduction to Statistical Learning.