# Ensamble Methods

*Ignasi Mañé, Antoni Company & Chiara Barbi*

*8 de Abril de 2019*

```
library(caret)
library(gplots)
library(rpart.plot)
library(randomForest)
library(mixOmics)
library(dplyr)
library(adabag)
library(ggplot2)
library(tidyr)
library(rsample)
```

```
data(srbct)
data <- as.data.frame(srbct$gene)
data$class <- srbct$class
```

## 1. Use Leave-One-Out cross-validation in order to determine the better combination of the random forest parameters ntree and mtry. To this end implement a two dimensional grid search approach. Explore LOOCV error for (ntree,mtry) values ranging in {5, 25, 100, 400}x {2, 10, 50}. Represent the LOOCV error values attained on the grid using a heat color map.

First, we create a list made off different functions to customize the behavior of the caret's train function. This configuration will allow the usage of ntree as a tunegrid parameter.

```
#method customization
customRF <- list(type = "Classification",
                 library = "randomForest",
                 loop = NULL)

customRF$parameters <- data.frame(parameter = c("mtry", "ntree"),
                                  class = rep("numeric", 2),
                                  label = c("mtry", "ntree"))

customRF$grid <- function(x, y, len = NULL, search = "grid") {}

customRF$fit <- function(x, y, wts, param, lev, last, weights,
                         classProbs, ...) {
```

```
  randomForest(x, y, mtry = param$mtry, ntree=param$ntree, ...)}

customRF$predict <- function(modelFit, newdata, preProc = NULL,
                             submodels = NULL)
   predict(modelFit, newdata)

customRF$prob <- function(modelFit, newdata, preProc = NULL,
                          submodels = NULL)
   predict(modelFit, newdata, type = "prob")

customRF$sort <- function(x) x[order(x[,1]),]
customRF$levels <- function(x) x$classes
```

We set the train control method to Leave One Out Cross Validation and define the combination of tunegrid parameters for which the model has to be trained. Then, after setting a seed to make the exercise reproducible, we train the random forest model.

```
# train model
ctrl.rf <- trainControl(method = "LOOCV")
tunegrid <- expand.grid(.mtry=c(2,10,50), .ntree=c(5,25,100,400))

set.seed(1234)
random.forest <- train(class~.,
                       data=data,
                       method=customRF,
                       metric="Accuracy",
                       trControl=ctrl.rf,
                       tuneGrid = tunegrid)
```

For each possible combination of the tune parameters, a model and its expected out sample error accuracy measured using LOOCV are obtained. The following table summarizes these results. Notice that the combination of parameters that minimizes the expected out sample error is try=50 and ntree=100.

```
sum <- spread(random.forest$results[,1:3], ntree, Accuracy,
              fill = NA, convert = FALSE)[,-1]
rownames(sum) <- c("2", "10", "50")
```

The results are summarized in the following table

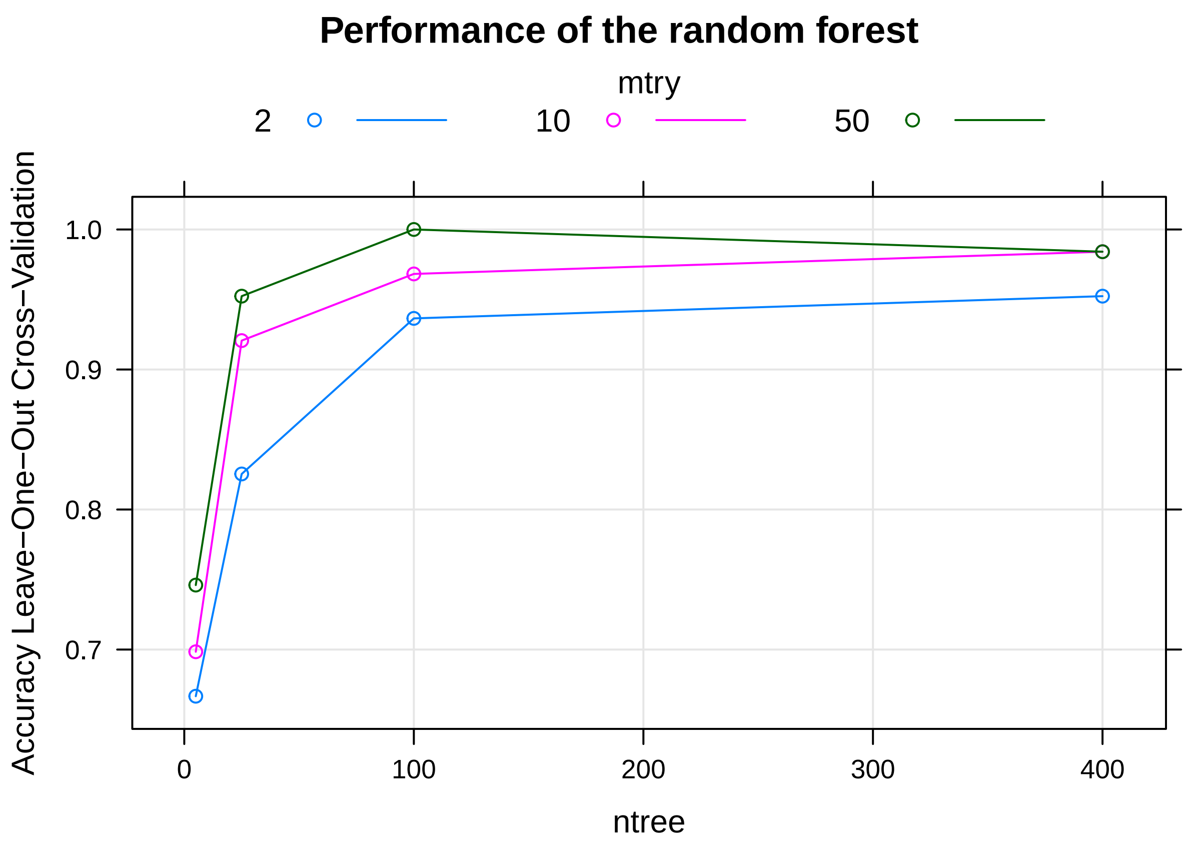
```
print(sum)
```

```
       5         25        100       400
```

2 0.6666667 0.8253968 0.9365079 0.952381 10 0.6984127 0.9206349 0.9682540 0.984127 50 0.7460317 0.9523810 1.0000000 0.984127

The following chart shows the same information visually represented

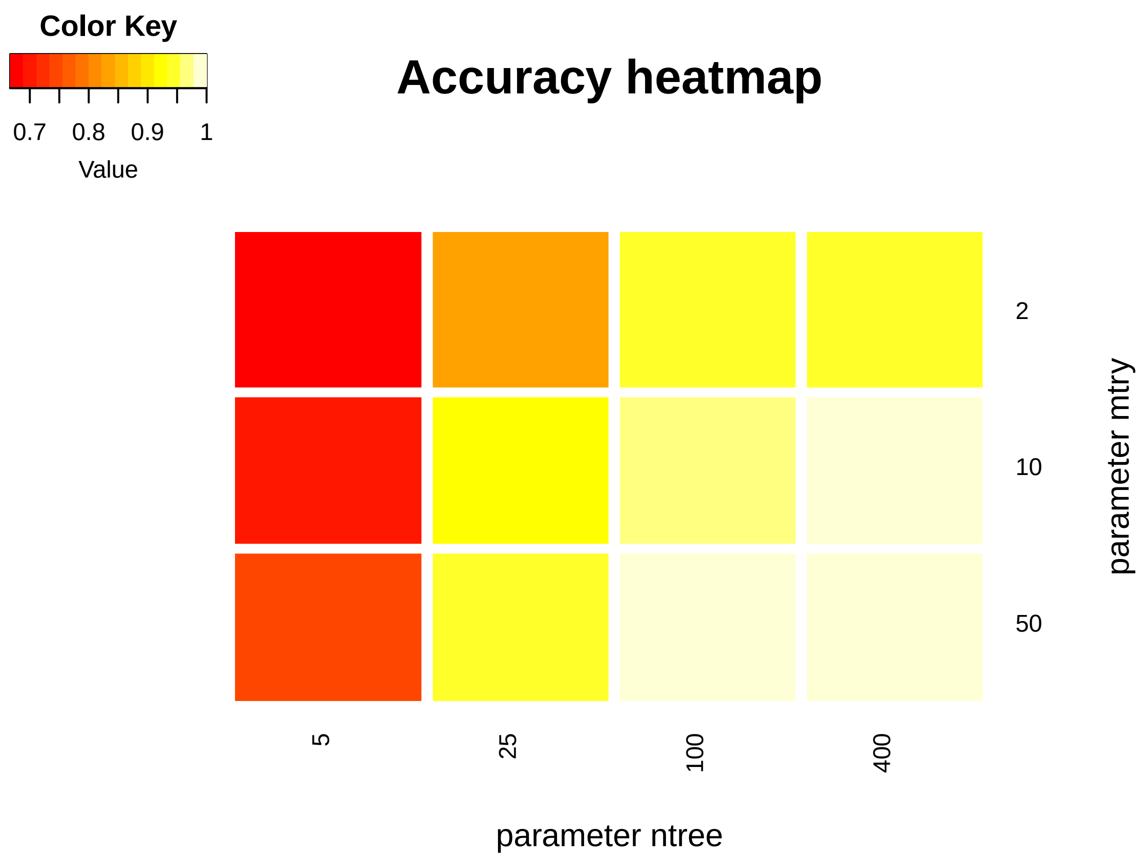
```
plot(random.forest, main="Performance of the random forest")
```



Following the instructions of the assignment, the previous table is also represented using a heat map.

```
res <- as.matrix(sum)

heatmap.2(res, dendrogram = "none", trace = "none", density.info = "none",
          Rowv = FALSE, rowsep = c(1,2), colsep = c(1,2,3,4),
          xlab = "parameter ntree", ylab = "parameter mtry",
          main = "Accuracy heatmap",
          cexRow=0.9, cexCol=0.9)
```



Taking into account the results, its seems that a higher value of mtry increases the accuracy of the model. Similarly, it seems that the accuracy of the model increases for a larger number of trees until it gets stabilized at some value.

**2. Run random forests in 100 bootstrap samples of SRBCT data and setting ntree and mtry parameters equal to the optimal values according with the previous point. Save the 30 most-important variables you find at each bootstrap loop.**

Using the optimal value for the parameters obtained in the previous question we run a random forest in 100 bootstrap samples to get the 30 most important variables. We will also record the importance of the whole set of variables for each sample in an other list.

```r
set.seed(1234)
top30.output <- list()
all.output <- list()
resamples <- bootstraps(data, times = 100)

for (i in 1:length(resamples$splits)){
  rf <- randomForest(class~., data = resamples$splits[[i]],
                     mtry = 50, ntree = 100)
  impvar <- importance(rf) # get importance of each variable
  all.output[[resamples$id[i]]] <- as.vector(impvar)
  # save importance for all variables
  impvar <- sort(impvar, decreasing = TRUE, index.return = TRUE)
  #sort variables by importance
  names(impvar$x) <- paste("g", impvar$ix, sep = "")
  set <- data.frame(impvar$x[1:30])
  set$names <- rownames(set)
  top30.output[[resamples$id[i]]] <- set[,c(2,1)]
  # save 30 most important variables
}
```

**3. You should see different sets of variables being ranked as the 30 most important for each loop. Implement a procedure for visualizing the overall ranking of the variables and the stability of ranks across the loops.**

In the previous exercise we obtained 100 sets of the most important 30 variables, not necessarily equal, and its importance to predict the response.

To asses the question, we can compute the average importance of each variable in the whole set of iterations. However, since a variable might not be recorded in every set, we established a criteria that assigns a value of 0 importance if a variable is not within the top 30 most important variables in one iteration.

```
join <- data.frame(top30.output[[1]]) %>% rename(b1 = impvar.x.1.30.)

for (i in 2:length(top30.output)){
  a <- paste("b",i,sep = "")
  join <- full_join(join,top30.output[[i]],by="names") %>%
    rename(!!a := impvar.x.1.30.)

}

join[is.na(join)] <- 0
join <- join %>% mutate(Mean.Imp = rowMeans(as.data.frame(join[,-1]))) %>%
  select(names, Mean.Imp) %>% arrange(desc(Mean.Imp)) %>%
  top_n(30, Mean.Imp) %>% rename(Variable := names)
```

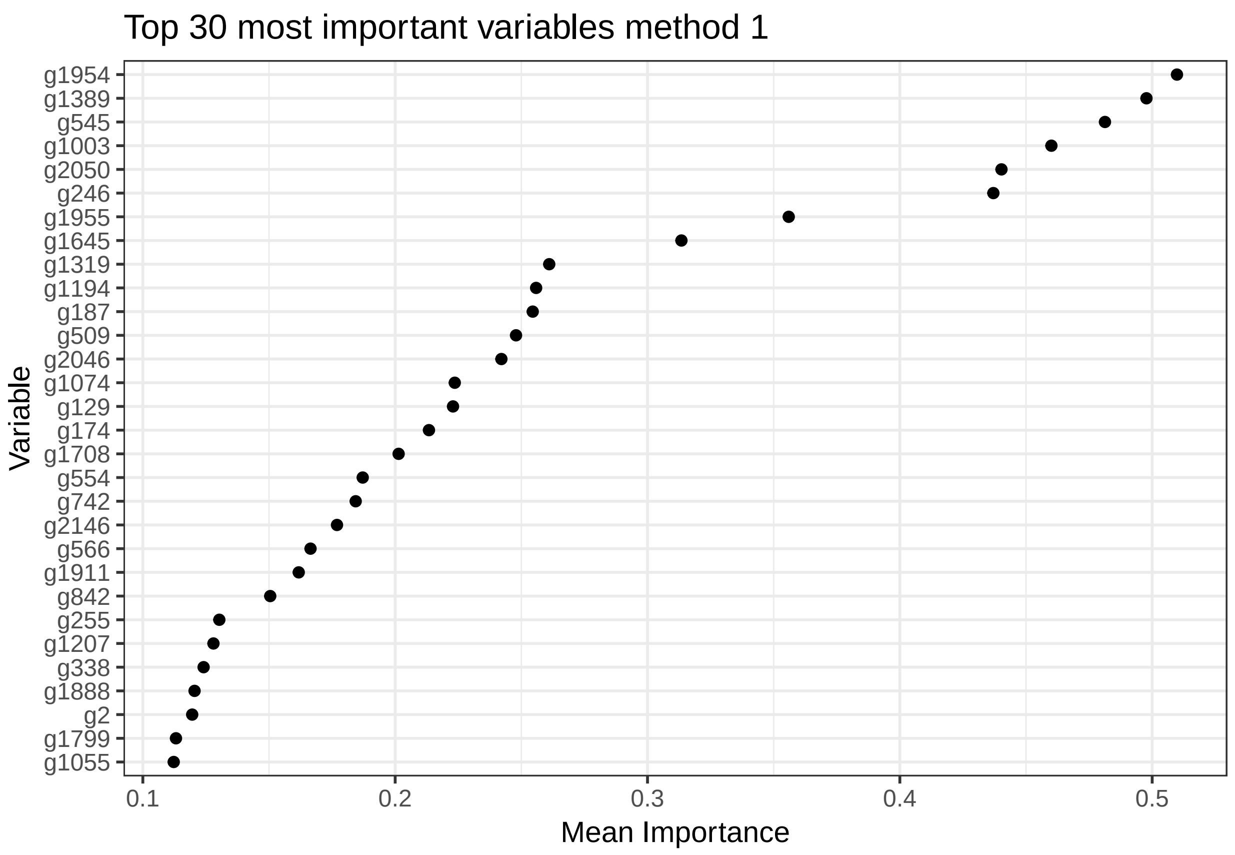
The results are summarized in the following table

```
print(join)
```

```
##    Variable  Mean.Imp
## 1     g1954 0.5098200
## 2     g1389 0.4977383
## 3      g545 0.4812905
## 4     g1003 0.4600549
## 5      g2050 0.4402703
## 6       g246 0.4370560
## 7      g1955 0.3559658
## 8      g1645 0.3134367
## 9      g1319 0.2610529
## 10     g1194 0.2558577
## 11      g187 0.2545132
## 12      g509 0.2478953
## 13     g2046 0.2420939
## 14     g1074 0.2236209
## 15      g129 0.2229189
## 16      g174 0.2133854
## 17     g1708 0.2013763
## 18      g554 0.1871328
## 19      g742 0.1843558
## 20     g2146 0.1769461
## 21      g566 0.1664480
## 22     g1911 0.1617865
## 23      g842 0.1504878
## 24      g255 0.1303071
## 25     g1207 0.1279804
## 26      g338 0.1240876
## 27     g1888 0.1205008
## 28        g2 0.1195644
## 29     g1799 0.1131320
## 30     g1055 0.1122399
```

The following chart shows the 30 most important variables according to this criteria

```
join %>%
  ggplot()+
  geom_point(aes(x = factor(Variable, levels = Variable[order(Mean.Imp)]),
                 y = Mean.Imp))+
  coord_flip()+
  ggtitle(label="Top 30 most important variables method 1")+
  xlab("Variable") + ylab("Mean Importance")+
  theme_bw()
```



The previous procedure is not accurate enough. Probably assigning a value of 0 importance to a variable that does not appear within the 30 most important penalizes in exes its true importance. Therefore, to asses the same question it would be better to compute the average importance of a variable taking into account the whole set of variables and its importance in random forest. Applying this criteria the following results are obtained.

```r
impor <- data.frame("Mean.Imp" = rowMeans(as.data.frame(all.output)))
impor["Variable"] <- paste("g", 1:(ncol(data)-1), sep = "")
impor <- impor %>% arrange(desc(Mean.Imp)) %>% top_n(30, Mean.Imp) %>%
  select(Variable,Mean.Imp)
```

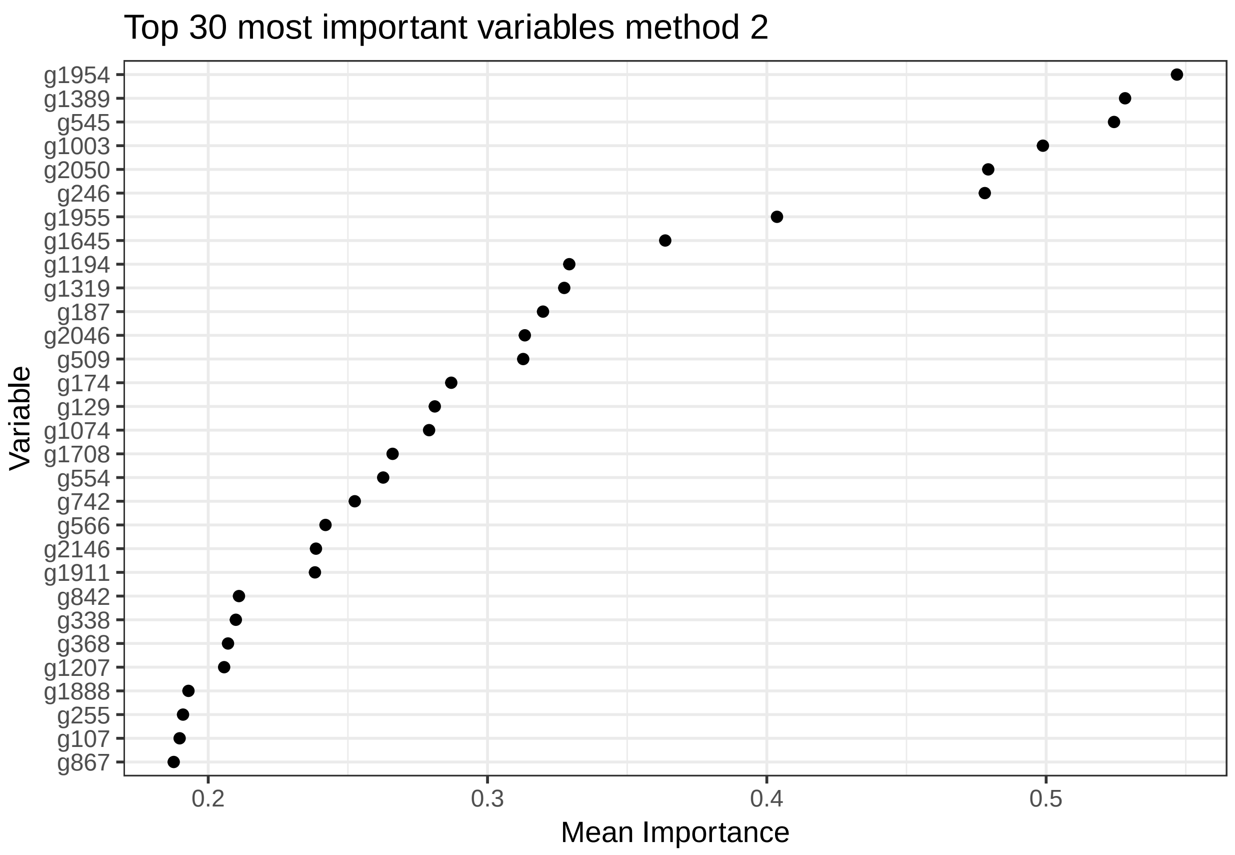
The results are summarized in the following table

```r
print(impor)
```

```
##    Variable  Mean.Imp
## 1     g1954 0.5468370
## 2     g1389 0.5282646
## 3      g545 0.5243111
## 4     g1003 0.4988438
## 5     g2050 0.4792720
## 6      g246 0.4780329
## 7     g1955 0.4036386
## 8      g1645 0.3636227
## 9      g1194 0.3292649
## 10     g1319 0.3274605
## 11      g187 0.3198668
## 12     g2046 0.3133520
## 13      g509 0.3127696
## 14      g174 0.2870099
## 15      g129 0.2811190
## 16     g1074 0.2790705
## 17     g1708 0.2660105
## 18      g554 0.2626445
## 19      g742 0.2524691
## 20      g566 0.2419869
## 21     g2146 0.2385749
## 22     g1911 0.2382068
## 23      g842 0.2110016
## 24      g338 0.2098879
## 25      g368 0.2070830
## 26     g1207 0.2056998
## 27     g1888 0.1929208
## 28      g255 0.1909757
## 29      g107 0.1897618
## 30      g867 0.1876436
```

The following chart shows the 30 most important variables according to this criteria

```
impor %>%
  ggplot()+
  geom_point(aes(x = factor(Variable, levels = Variable[order(Mean.Imp)]),
                 y = Mean.Imp))+
  coord_flip()+
  ggtitle(label="Top 30 most important variables method 2")+
  xlab("Variable") + ylab("Mean Importance")+
  theme_bw()
```



Notice that the results obtained through both methods are quite similar, actually almost all the variables and rank positions coincide.

**4. Apply the discrete AdaBoost algorithm (with an exponential loss function, and number of trees equal to 1000) in SRBCT data. Compare the LOOCV error attained by different ensemble classifiers based on trees of sizes: stumps, 4-node trees, 8-node trees, and 16- node trees.**

Since the AdaBoost algorithm is computationally very expensive we will only use the 30 most important variables of the previous question, cross validation with 5 folds instead of LOOCV and a maximum of 500 trees.

```
fil <- data.frame("Mean.Imp" = rowMeans(as.data.frame(all.output)))
fil["Variable"] <- paste("g", 1:(ncol(data)-1), sep = "")
fil30 <- fil %>% arrange(desc(Mean.Imp)) %>% top_n(30, Mean.Imp) %>%
  select(Variable,Mean.Imp)

data30 <- data[,c(fil30[,1],"class")] #data set with top 30 variables
```

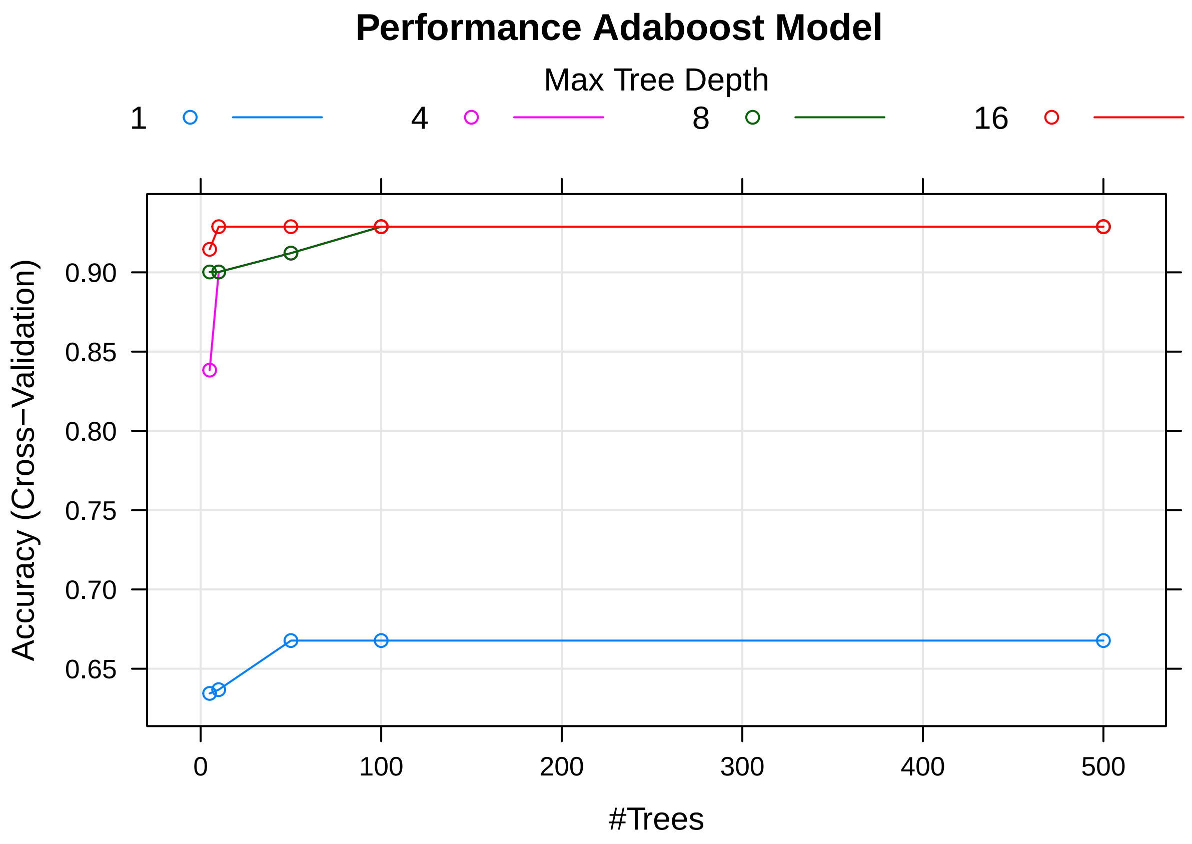
For each possible combination of the tunning parameters, a model and its expected out sample error accuracy measured using cross validation are obtained.

```r
# Adaboost
ctrl.ada <- trainControl(method = "cv", number=5, search = "grid")
tunegrid <- expand.grid(mfinal = c(5,10,50,100,500), maxdepth = c(1,4,8,16),
                        coeflearn = "Breiman")

ada <- train(class~.,
                data = data30,
                method ="AdaBoost.M1",
                metric="Accuracy",
                trControl=ctrl.ada,
                tuneGrid = tunegrid)
```

The following chart summarizes the results. Notice that the combination of parameters that minimizes the expected out sample error is maxdepth=4 and mfinal=100

```r
plot(ada, cex=0.8, main="Performance Adaboost Model")
```



Taking into account the results, its seems that a higher depth increases the accuracy of the model. However, at a certain point (8 in this example) the accuracy decreases. Therefore, we could state that a model with an excess of depth might over fit the data, so it is important to use methods such as cross validation to chose the optimal value for this parameter. Similarly, it seems that the accuracy of the model increases for larger numbers of trees until it gets stabilized at some value. So, at some point, more trees do not improve the performance of the model.