Appendix R10: Decision Trees

Bagging, Random Forest & Boosted Trees

J. Alberto Espinosa

3/19/2022

Table of Contents

Decision Trees

I covered basic regression and classification trees in the R appendix to Chapter 3. This appendix assumes that you have reviewed that material. In that appendix I illustrated how to fit regression and classification trees, and also how to control the size of the tree with the **mindev** parameter. However, this parameter only restricts the size of the tree, but does not let us identify the optimal tree size. As I discussed earlier, plain trees are generally not as accurate as other predictive models. However, there are more advanced methods that have proven to be quite accurate, sometimes outperforming other regression models.

In this appendix, I illustrate how to identify the optimal size of a tree using cross-validation, and also discuss three advanced tree modeling methods: **Bootstrap Aggregation (Bagging)**, **Random Forest** and **Boosted Trees**.

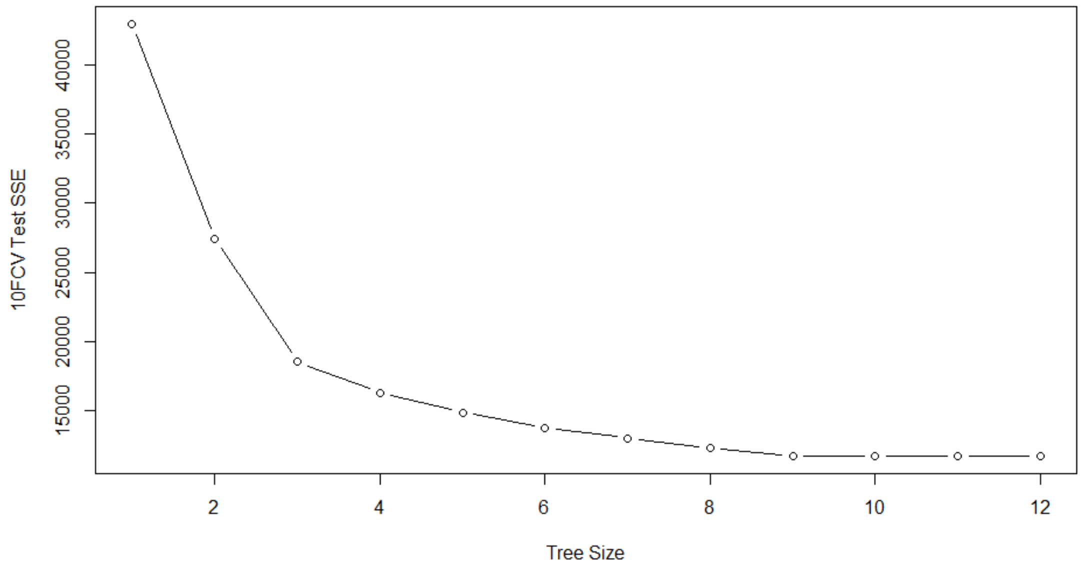
Optimal Regression Tree Size with Cross-Validation

Cross validation of tree results. Let's explore the CV for various pruned trees. I use the **cv.tree()** function from the **{tree}** library. I illustrate this example using the **Boston** housing data set from the **{MASS}** library.

```
library(MASS) # Contains the Boston data set
library(tree)

set.seed(1) # To get repeatable results

tree.boston <- tree(medv ~ ., Boston, mindev = 0.005) # Fit the tree
cv.boston <- cv.tree(tree.boston) # Extract CV data from it
```

Plot the tree size (number of terminal nodes) against deviance:

```
plot(cv.boston$size,
     cv.boston$dev,
     xlab = "Tree Size", ylab = "10FCV Test SSE",
     type = 'b')
```



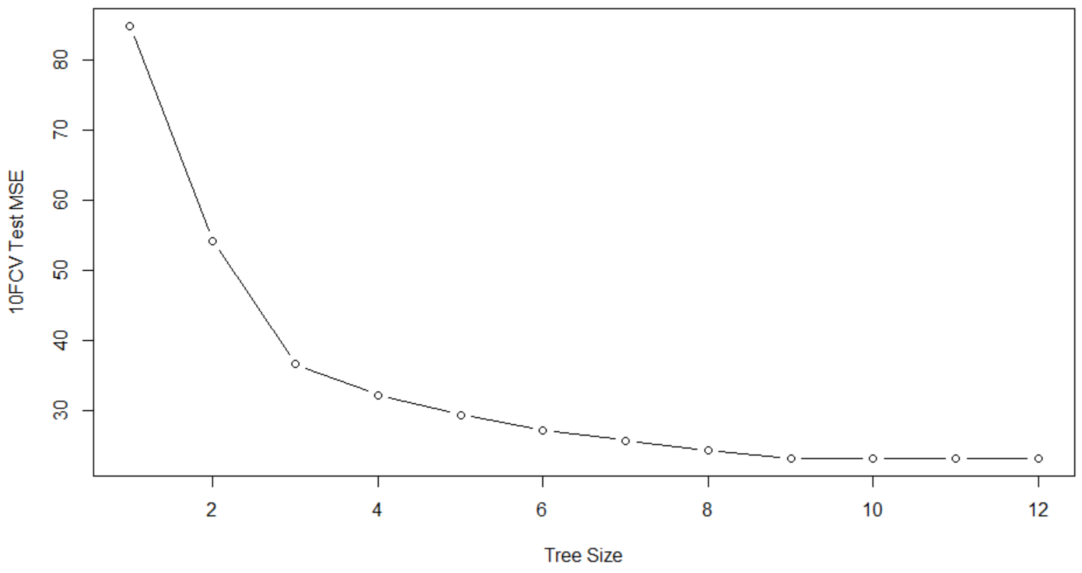
Notes:

- The **cv.tree()** function does **10FCV** and minimizes the deviance measured as the MSE in regression trees (or 2LL for classification trees). You can change the 10-Fold to other folds with the attribute **K = nn** (number of folds).

- `type='b'` is for **“both”**, points and lines
- The **$dev** value in the plot is the sum of squared errors (SSE), which we easily convert to MSE by dividing by the number of observations.

If you prefer to plot the MSE, you can do this:

```
plot(cv.boston$size,
     cv.boston$dev / nrow(Boston),
     xlab = "Tree Size", ylab = "10FCV Test MSE",
     type = 'b')
```



Let’s now list the tree size and their corresponding deviance:

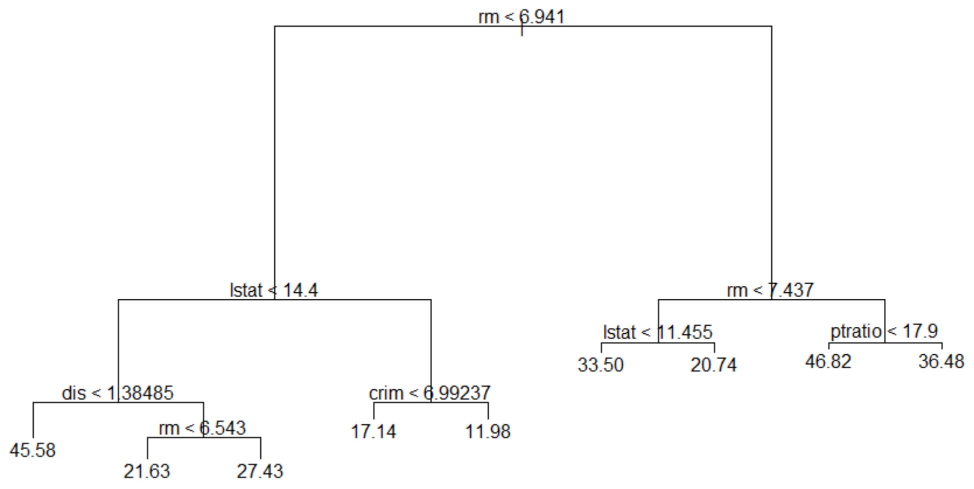
```
round(
  cbind("Size" = cv.boston$size,
        "10FCV Test MSE" = cv.boston$dev / nrow(Boston)),
  digits = 2)

##       Size 10FCV Test MSE
##  [1,]   12          23.21
##  [2,]   11          23.21
##  [3,]   10          23.21
##  [4,]    9          23.21
##  [5,]    8          24.32
##  [6,]    7          25.75
##  [7,]    6          27.19
##  [8,]    5          29.41
##  [9,]    4          32.26
## [10,]    3          36.62
## [11,]    2           54.22
## [12,]    1           84.84
```

Since the data is in 2 vectors, we can find the best tree size computationally:

```
# Find the smallest deviance
min.dev <- min(cv.boston$dev)

# Find the index of the tree with best CV deviance
best.ind <- which(cv.boston$dev == min.dev)

# Use the index to fint the tree size with best CV deviance
best.size <- cv.boston$size[best.ind]

# List all three values
cbind("Smallest Deviance" = min.dev,
      "Tree Number" = best.ind,
      "Best Tree Size" = best.size)

##      Smallest Deviance Tree Number Best Tree Size
## [1,]          11744.91           1             12
## [2,]          11744.91           2             11
## [3,]          11744.91           3             10
## [4,]          11744.91           4              9
```

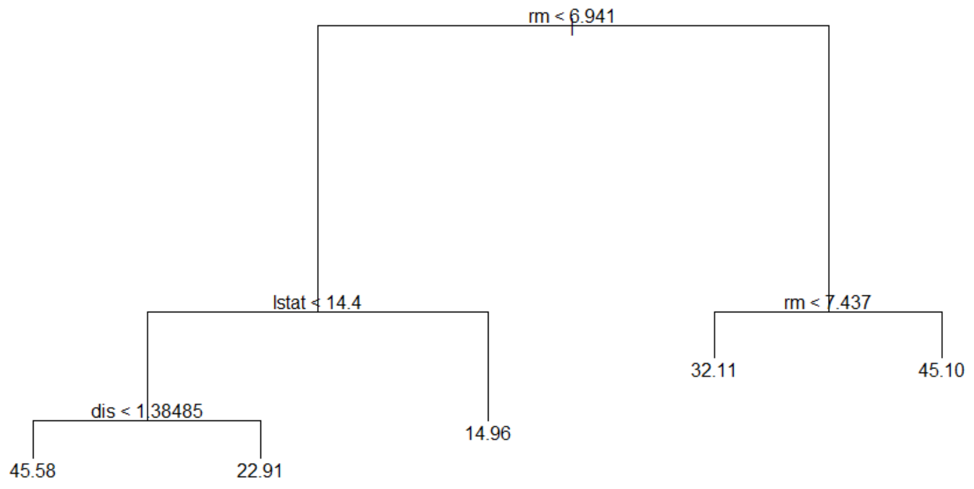
Notice that the lowest deviance is the one for the most complex tree with 9 branches. It is possible that the best tree that minimizes the CV deviance is larger than 9 branches, because mindev stopped the tree at 9 branches. There is no need to prune the tree because the best tree is the largest, but let's prune it anyway as an illustration. In this case **best.size = 9**, but this is how you would prune any tree to its optimal size, computationally

```
prune.boston <- prune.tree(tree.boston, best = best.size)

plot(prune.boston)
text(prune.boston, pretty = 0)
```



Just to illustrate, let's prune to just 5 terminal nodes (or regions, or leaves):

```
prune.boston <- prune.tree(tree.boston, best = 5)

plot(prune.boston)
text(prune.boston, pretty = 0)
```



If instead of using **cv.tree** to do cross-validation, you prefer to do sub-sample splitting manually, with training and test data (illustrated with the Boston data set):

```r
set.seed(1) # To get replicable results

trsize <- 0.7
train <- sample(1:nrow(Boston), trsize * nrow(Boston)) # Train index

Boston.train <- Boston[train, ] # Train sub-sample
Boston.test <- Boston[-train, ] # Test sub-sample
```

Now fit a regression tree on the train data

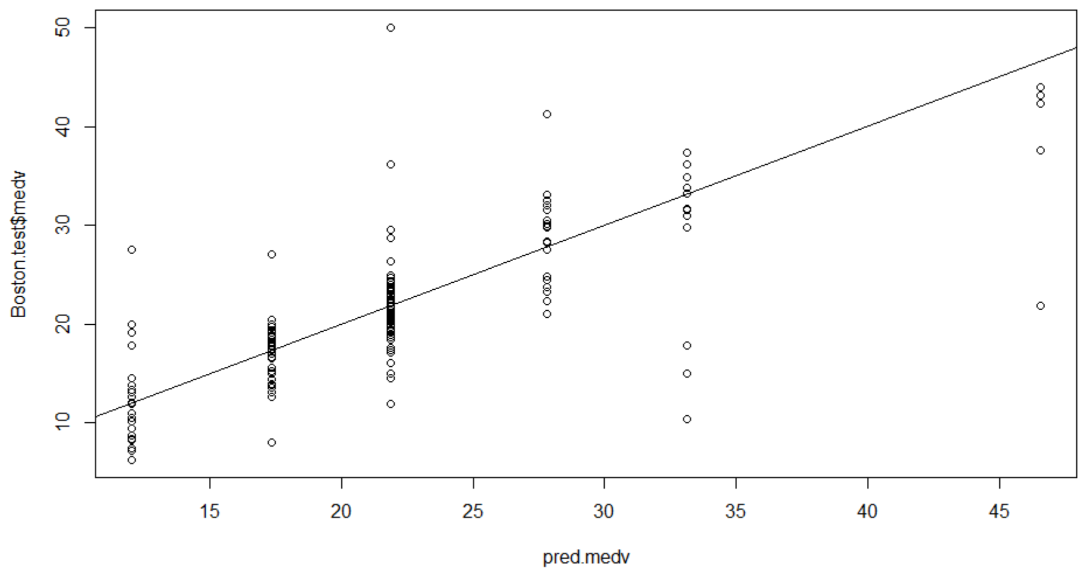
```r
tree.boston.tr <- tree(medv ~ ., Boston.train)
```

Then use the trained model to test it with the test subset

```r
pred.medv <- predict(tree.boston.tr, Boston.test)
```

Now, let's visualize predicted vs. actual values

```r
plot(pred.medv, Boston.test$medv) # Let's plot predicted vs. actual
abline(0, 1) # And draw a line
```



And then compute the test MSE

```r
mse.tree <- mean( (pred.medv - Boston.test$medv) ^ 2 ) # And calculate the MS
E
mse.tree

## [1] 36.2319
```

The disadvantage of finding the CV Test MSE this way is that you need to fit multiple trees with a loop or else to identify the best tree. The **cv.tree()** approach is more efficient.

Optimal Classification Tree Size with Cross-Validation

Let's fit a classification tree with the **diamonds** data set in the **{ggplot2}** library. But instead of predicting price, let's divide the diamonds into expensive ones if their price is over the median price, and affordable (i.e., not expensive) otherwise.

```
library(ggplot2)

median.price <- median(diamonds$price)
diamonds$expensive <- factor(
      ifelse(diamonds$price <= median.price, "No", "Yes"))

table(diamonds$expensive) # Roughly evenly divided

##
##    No   Yes
## 26985 26955
```

Let's now fit a classification tree to predict what makes diamonds expensive, using a few predictors and then extract the CV data. Let's use a small mindev to get a larger tree.

Note: `FUN = prune.misclass` uses misclassification error for cross-validation and prunning. Othewise, the default is deviance.

```
diamonds.exp <- tree(expensive ~ carat + cut + color + clarity,
                     data = diamonds,
                     mindev = 0.001)

summary(diamonds.exp)

##
## Classification tree:
## tree(formula = expensive ~ carat + cut + color + clarity, data = diamonds,
##     mindev = 0.001)
## Number of terminal nodes:  28
## Residual mean deviance:  0.134 = 7224 / 53910
## Misclassification error rate: 0.02842 = 1533 / 53940

cv.diamonds <- cv.tree(diamonds.exp, FUN = prune.misclass)
cv.diamonds # Inspect the basic cv.tree() object results

## $size
##  [1] 28 18 17 16 13 11  9  8  7  5  2  1
##
## $dev
##  [1]  2768  2768  2768  2846  2856  2856  2856  2856  2856  2856  2856 271
14
##
## $k
##  [1]        -Inf     0.00000    20.00000    26.00000    40.66667    53.500
00
```

```
##  [7]    55.50000    59.00000    91.00000    95.00000   199.00000 24099.000
00
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

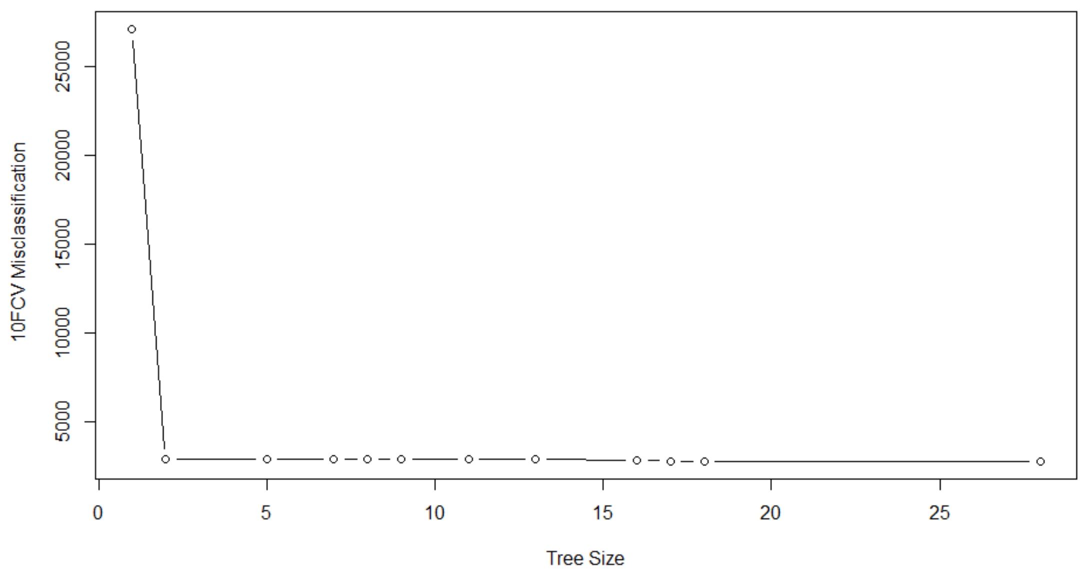
size ∗∗ *and* ∗∗**dev** report the tree size and the corresponding CV miss-classification rate in this case.

```
cbind("Tree Size" = cv.diamonds$size, "Misclass" = cv.diamonds$dev)
```

```
##       Tree Size Misclass
##  [1,]        28     2768
##  [2,]        18     2768
##  [3,]        17     2768
##  [4,]        16     2846
##  [5,]        13     2856
##  [6,]        11     2856
##  [7,]         9     2856
##  [8,]         8     2856
##  [9,]         7     2856
## [10,]         5     2856
## [11,]         2     2856
## [12,]         1    27114
```

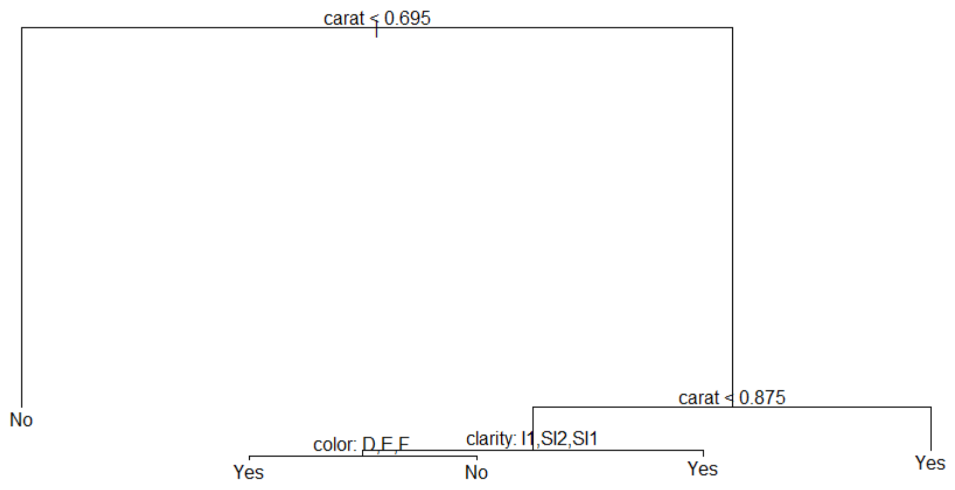
Let's inspect the trees visually. Let's plot CV for each tree size, misclassification in this case.

```
plot(cv.diamonds$size,
     cv.diamonds$dev,
     xlab = "Tree Size", ylab = " 10FCV Misclassification",
     type = "b") # type="b" plots "both", lines and dots
```



Missclassification is lowest at 28 nodes, but flattens very quickly at around 2, so let's prune to 5 nodes.

```
prune.diamonds <- prune.misclass(diamonds.exp, best = 5)

plot(prune.diamonds) # Plot the tree
text(prune.diamonds, pretty = 0) # With labels
```



Tree splitting based on Gini index of Purity

The default leaf splitting method is **miss-classification**. But you can split leaves using the **Gini** index of **impurity**. Notice that the misclassification error improves with impurity splitting. This is not always the case. If the data has high classification purity, then `split = "gini"` will work better. Otherwise the default based on miss-classification will work best. The best thing is to try both.

```
diamonds.exp.g <- tree(expensive ~ carat + cut + color + clarity,
                       data = diamonds,
                       mindev = 0.001,
                       split = "gini")

summary(diamonds.exp.g) # Notice the higher misclassification error

##
## Classification tree:
## tree(formula = expensive ~ carat + cut + color + clarity, data = diamonds,
##     split = "gini", mindev = 0.001)
## Number of terminal nodes:  582
## Residual mean deviance:  0.07269 = 3878 / 53360
## Misclassification error rate: 0.01674 = 903 / 53940
```

Cross Validation with the Confusion Matrix

The method above fits a tree model with the entire data, but then the cv.tree() function does 10-Fold Cross Validation to find the optimal tree size to prune to. This is good if you are only interested in overall predictive accuracy. But in most classification models (e.g., logistic, LDA, etc.), we are also interested in accuracy statistics like **sensitivity**, **specificity** and **false positives**. We can compute all the necessary **Confusion Matrix** statistics exactly as we did for classification models. We use random splitting CV to build the Confusion Matrix and compute the metrics. Let's illustrate an example with the **Heart.csv** data set.

Note: I use the file **Heart.csv**, which I pre-downloaded, but you can access the data set from the **{ISLR}** book authors and library developer's site as follows:

```
heart <- read.table("https://www.statlearning.com/s/Credit.csv",
                    sep = ",",
                    header = T,
                    stringsAsFactors = T)

heart <- read.table("Heart.csv",
                    sep = ",",
                    header = T,
                    stringsAsFactors = T)

heart$chd <- as.factor(heart$chd) # Convert chd from numeric to factor
```

Let's now extract the train and test subsets

```
set.seed(1)
```

```r
trsize <- 0.7
train <- sample(1:nrow(heart), trsize * nrow(heart))

heart.train <- heart[train, ] # Train subset
heart.test <- heart[-train, ] # Test subset
```

Train the tree with the train subset, and then make classification predictions using the trained model with the test subset. We use the **predict()** to predict **classifications** (0 or 1) using the parameter `type = "class"`. If the probability of an outcome is greater than 0.5 the predicted classification is 1, and 0 otherwise. Note that I'm naming the prediction object **heart.tree.pred.class.5**. It's a long name, but it helps to remember that it's tree prediction for the heart data set, with classification outcomes with a threshold of 0.5. The long name makes it easier to remember all this.

```r
heart.tree.train <- tree(chd ~ ., data = heart.train)

heart.tree.pred.class.5 <- predict(heart.tree.train,
                                   heart.test,
                                   type = "class")
```

If we want to use a different classification threshold, we need to make probability predictions and convert them to classifications with an `ifelse()` function, as I illustrate a bit later. If we omit the parameter `type = "class"`, the default prediction is the probability of the outcome being 1.

```r
heart.tree.pred.prob <- predict(heart.tree.train, heart.test)
```

Confusion Matrix

```r
conf.mat <- table("Predicted" = heart.tree.pred.class.5,
                  "Actual" = heart.test$chd)

conf.mat # Take a look

##            Actual
## Predicted  0  1
##          0 68 28
##          1 22 21
```

Confusion Matrix Metrics

```r
TruN <- conf.mat[1,1] # True negatives
TruP <- conf.mat[2,2] # True positives
FalN <- conf.mat[1,2] # False negatives
FalP <- conf.mat[2,1] # False positives
TotN <- TruN + FalP # Total actual negatives
TotP <- TruP + FalN # Total actual positives
TotNpr <- TruN + FalN # Total negative predictions
TotPpr <- TruP + FalP # Total positive predictions
Tot <- TotN + TotP # Total
```

```
# Do a quick check of the computations
cbind(TruN, TruP, FalN, FalP, TotN, TotP, TotNpr, TotPpr, Tot)

##      TruN TruP FalN FalP TotN TotP TotNpr TotPpr Tot
## [1,]   68   21   28   22   90   49     96     43 139
```

Let’s add totals and labels to the confusion matrix

```
conf.mat.totals <- cbind(conf.mat, c(TotNpr, TotPpr))
conf.mat.totals <- rbind(conf.mat.totals, c(TotN, TotP, Tot))

colnames(conf.mat.totals) <-
  rownames(conf.mat.totals) <-
     c("No","Yes", "Total")

conf.mat.totals

##       No Yes Total
## No    68  28    96
## Yes   22  21    43
## Total 90  49   139
```

Confusion Matrix Accuracy and Error Rates

```
Accuracy.Rate <- (TruN + TruP) / Tot
Error.Rate <- (FalN + FalP) / Tot
Sensitivity <- TruP / TotP
Specificity <- TruN / TotN
FalseP.Rate <- 1 - Specificity

# All together

heart.tree.rates.5 <- c(Accuracy.Rate, Error.Rate, Sensitivity,
                        Specificity, FalseP.Rate)

names(heart.tree.rates.5) <- c("Accuracy", "Error", "Sensitivity",
                               "Specificity", "False Pos")

print(heart.tree.rates.5, digits = 2)

##    Accuracy       Error Sensitivity Specificity   False Pos
##        0.64        0.36        0.43        0.76        0.24
```

If we want to change the **classification threshold** to, say **0.6**, you can compute the classifications as follows:

```
thresh <- 0.6 # Set the threshold
heart.tree.pred.class.6 <- ifelse(heart.tree.pred.prob[, 2] > thresh, 1, 0)

# Confusion matrix with this new threshold
```

```
conf.mat <- table("Predicted" = heart.tree.pred.class.6,
                 "Actual" = heart.test$chd)

conf.mat

##          Actual
## Predicted  0  1
##         0 73 33
##         1 17 16
```

The rest of the confusion matrix script is exactly the same as above:

```
TruN <- conf.mat[1,1] # True negatives
TruP <- conf.mat[2,2] # True positives
FalN <- conf.mat[1,2] # False negatives
FalP <- conf.mat[2,1] # False positives
TotN <- TruN + FalP # Total actual negatives
TotP <- TruP + FalN # Total actual positives
TotNpr <- TruN + FalN # Total negative predictions
TotPpr <- TruP + FalP # Total positive predictions
Tot <- TotN + TotP # Total

cbind(TruN, TruP, FalN, FalP, TotN, TotP, TotNpr, TotPpr, Tot)

##      TruN TruP FalN FalP TotN TotP TotNpr TotPpr Tot
## [1,]   73   16   33   17   90   49    106     33 139
```

Let’s add totals and labels to the confusion matrix

```
conf.mat.totals <- cbind(conf.mat, c(TotNpr, TotPpr))
conf.mat.totals <- rbind(conf.mat.totals, c(TotN, TotP, Tot))

colnames(conf.mat.totals) <-
  rownames(conf.mat.totals) <-
     c("No","Yes", "Total")

conf.mat.totals

##       No Yes Total
## No    73  33   106
## Yes   17  16    33
## Total 90  49   139
```

Confusion Matrix Accuracy and Error Rates

```
Accuracy.Rate <- (TruN + TruP) / Tot
Error.Rate <- (FalN + FalP) / Tot
Sensitivity <- TruP / TotP
Specificity <- TruN / TotN
FalseP.Rate <- 1 - Specificity
```

```
heart.tree.rates.6 <- c(Accuracy.Rate, Error.Rate, Sensitivity,
                         Specificity, FalseP.Rate)

names(heart.tree.rates.6) <- c("Accuracy", "Error", "Sensitivity",
                                "Specificity", "False Pos")

print(heart.tree.rates.6, digits = 2)

##    Accuracy       Error Sensitivity Specificity   False Pos
##        0.64        0.36        0.33        0.81        0.19
```

Both thresholds together

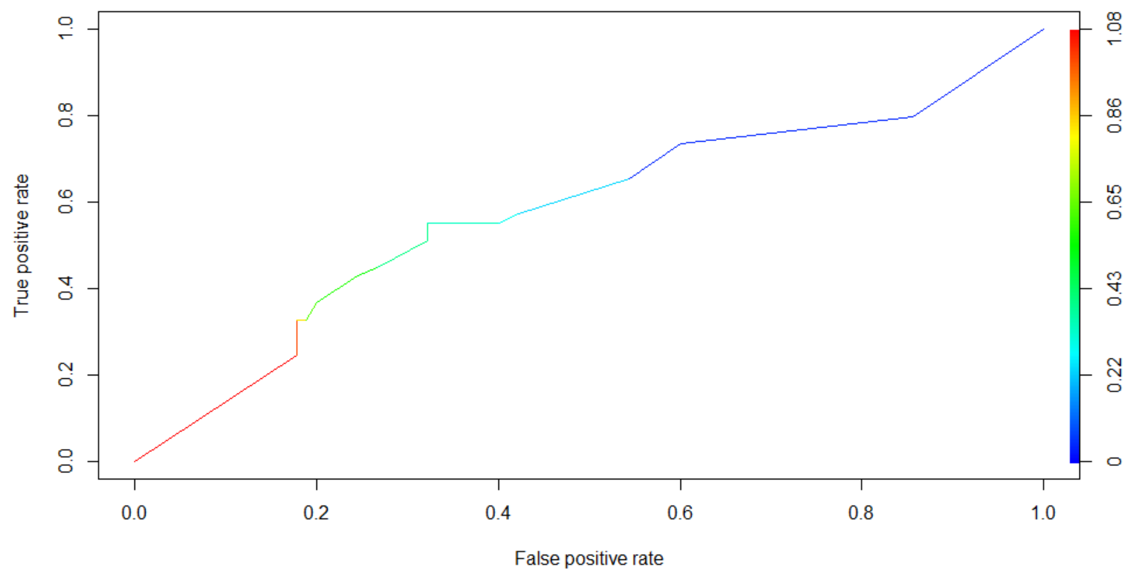
```
heart.tree.rates.both <-
    round(rbind(heart.tree.rates.5, heart.tree.rates.6),
          digits = 3)

rownames(heart.tree.rates.both) <- c("50% Threshold",
                                     "60% Threshold")

heart.tree.rates.both # Check it out

##               Accuracy Error Sensitivity Specificity False Pos
## 50% Threshold     0.64  0.36       0.429       0.756     0.244
## 60% Threshold     0.64  0.36       0.327       0.811     0.189
```

Based on the results above, can you figure out which model is better overall? When predicting positives? When predicting Negatives?

ROC Curves for Classification Trees

For ROC curves, we use the **prediction()** and **performance()** functions from the **{ROCR}** library, exactly as we did for other classification models. We use the **predicted probabilities** `heart.tree.pred.prob` that we computed above. This object is matrix with 2 columns. The first has the probability of being 0 and the second has the probability of being 1. We only need the second column, which we can extract with the [, 2] index.

```
library(ROCR)

pred <- prediction(heart.tree.pred.prob[, 2],
                   heart.test$chd)

perf <- performance(pred, "tpr", "fpr")

plot(perf, colorize = T)
```



```
auc <- performance(pred, "auc") # Compute the AUC

auc.name <- auc@y.name[[1]] # AUC label text
auc.value <- round(auc@y.values[[1]], digits = 3) # AUC value, rounded

paste(auc.name, "is", auc.value) # The model is not so great

## [1] "Area under the ROC curve is 0.577"
```

Bootstrap Aggregation (Bagging) Trees

Trees are notoriously inaccurate. Part of the issue is that the first split can totally change the results. So, it is not uncommon to have trees with both, bias and variance. But as it turns out, if you fit many trees, they tend to be less biased and more accurate on average. Bagging, Random Forest and Boosted trees take advantage of this property, rendering more accurate results.

Bagging stands for **Bootstrap Aggregation**. Bootstrap is the process of drawing multiple random samples of data to fit a model, but the samples are drawn with replacement. If the data set has n observations, the bootstrap sample matches that by drawing n observations. But because the sample is drawn with replacement, some observations will be repeated and some will be left out. The idea with bootstrapping is by drawin hundreds or thousands of bootstrap samples, in the end, all data points will be represented.

With **Bagging**, each bootstrapped tree includes **all predictors** in the model, fitted with a different bootstrap sample. The results are then aggregated, reducing both, bias and variance, yielding more accurate and stable results than single trees. The main **tuning**

parameters in Bagging is the size of the random sample, which is generally the same as the data set size, and the **number of trees** bootstrapped, fitted and aggregated.

As I discuss a bit later, **Bagging** is a special case of **Random Forest**. So, we use the **randomForest()** function from the **{randomForest}** library.

```
library(randomForest)
library(MASS) # Contains the Boston housing data set
```

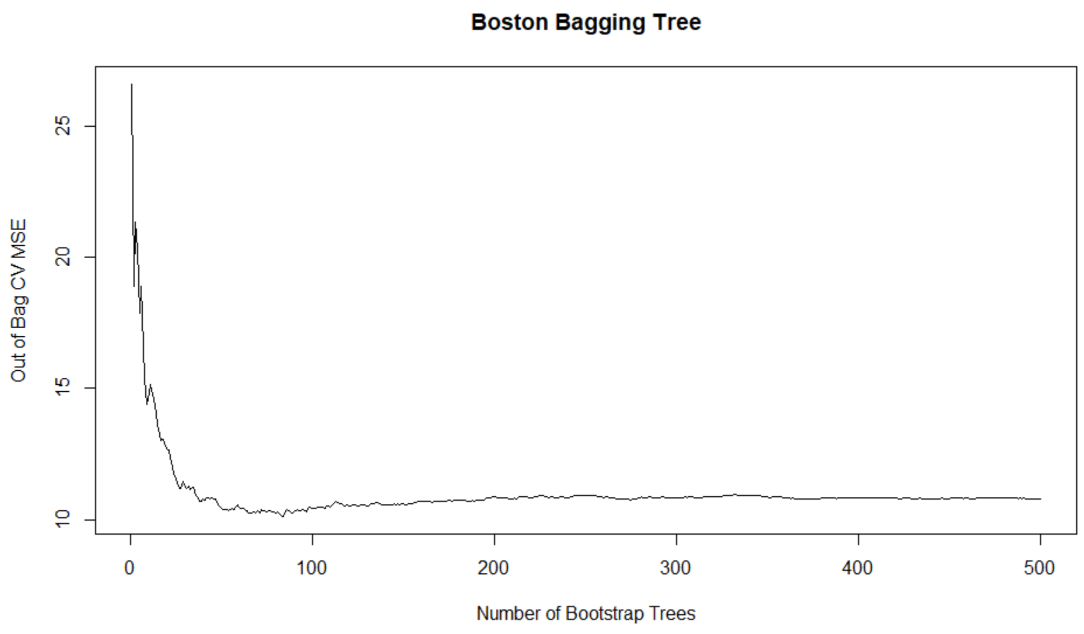
Let's now fit a Bagging tree model. Note that this is a regression tree model, but the same syntax applies to classification trees. A **Bagging** tree is the same as a **Random Forest** tree, but the bagging tree includes the same predictors in each bootsrapped tree. We will see later that this is not the case in more general Random Forest trees.

As you can see below, there are 13 predictors in the model, so we use `mtry = 13` to include all 13 predictors in each bootstrapped tree. The parameter `importance = T` instructs the randomForest() function to compute the **variable importance** graphs, which I discuss a bit later.

```
ncol(Boston.train) # 14 variables -> 1 outcome and 13 predictors

## [1] 14

bag.boston <- randomForest(medv ~ .,
                           data = Boston,
                           mtry = 13,
                           importance = T)

bag.boston # Take a quick look.

##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = T
)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 10.7787
##                     % Var explained: 87.23
```

The error is the out of bag MSE averaged over all bootstrapped trees trees. To see how the out of bag MSE changes as the number of bootstrapped trees increase see:

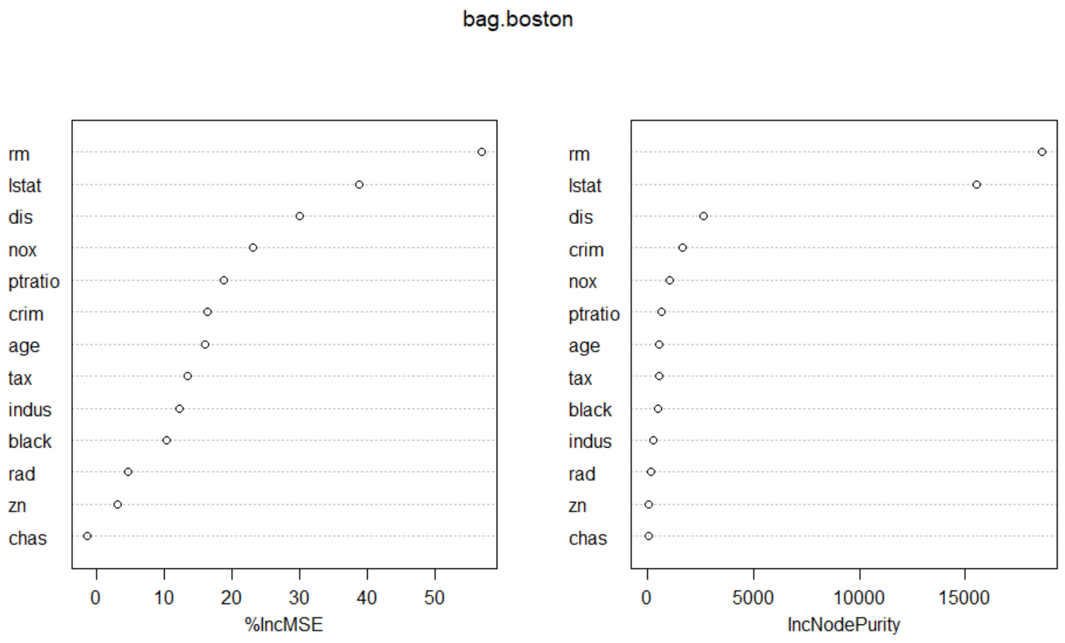
```
plot(bag.boston$mse,
     main = "Boston Bagging Tree",
     xlab = "Number of Bootstrap Trees",
     ylab = "Out of Bag CV MSE",
     type="l")
```



One shortcoming of trees is that there are no coefficients or p-values to ascertain which variables have stronger effects. But the **variable importance** computations help overcome this problem by displaying the importance of each variable to the tree model.

Variable Importance Plot

```
varImpPlot(bag.boston) # Variable importance plot
```



```
round(
      importance(bag.boston),
      digits = 2) # Variable importance values

##         %IncMSE IncNodePurity
## crim      16.32       1656.10
## zn         3.06         38.40
## indus     12.18        247.43
## chas      -1.41         38.13
## nox       23.04       1040.60
## rm        56.95      18596.58
## age       16.04        573.93
## dis       30.03       2647.04
## rad        4.62        157.39
## tax       13.52        571.49
## ptratio   18.75        641.96
## black     10.38        496.13
## lstat     38.76      15542.23
```

The 2 values of importance reported above are:

- Mean increase (over all trees) in accuracy (% MSE explained) when the variable is added to the the model
- Mean increase (over all trees) in “purity” when the tree is split by that variable

Higher values are best for either

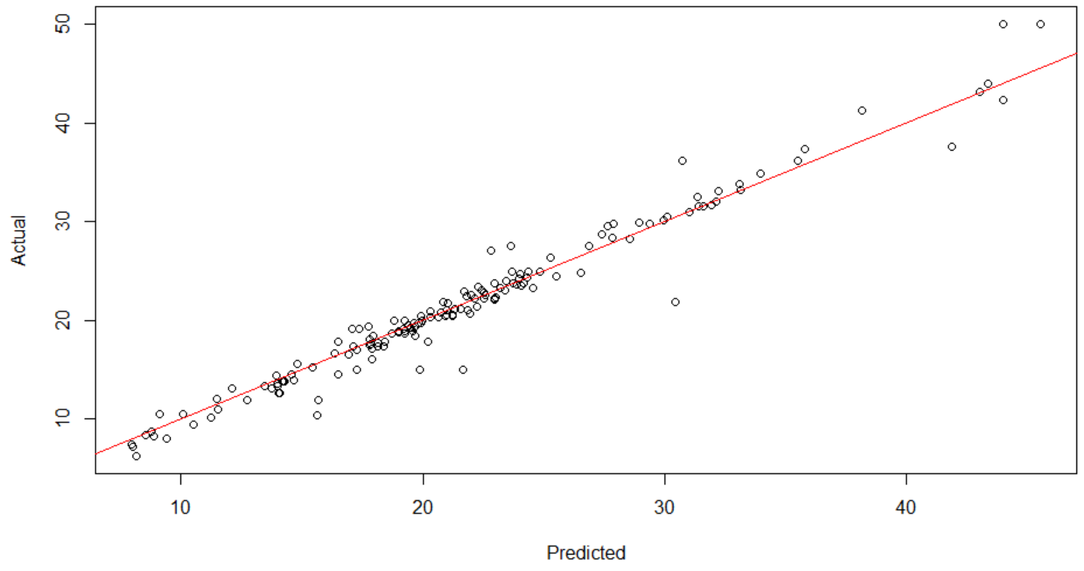
Bagging Tree Predictions and Cross-Validation

Let’s re-create the Boston training subset we created above for easy reference.

```
set.seed(1) # To get repeatable results

trsize <- 0.7
train <- sample(1:nrow(Boston), trsize * nrow(Boston))

Boston.train <- Boston[train, ] # Train sub-sample
Boston.test <- Boston[-train, ] # Test sub-sample
```

Let’s make predictions with the trained model and the test subset, and plot the predictions against the actual values, to evaluate the accuracy of our predictions.

```
bag.pred <- predict(bag.boston, newdata = Boston.test)

plot(bag.pred,
     Boston.test$medv,
     xlab = "Predicted",
     ylab = "Actual") # Plot pred vs. actual

abline(0, 1, col = "red") # 45 degree line (intercept = 0; slope = 1)
```



Now let’s calculate the CV Test MSE

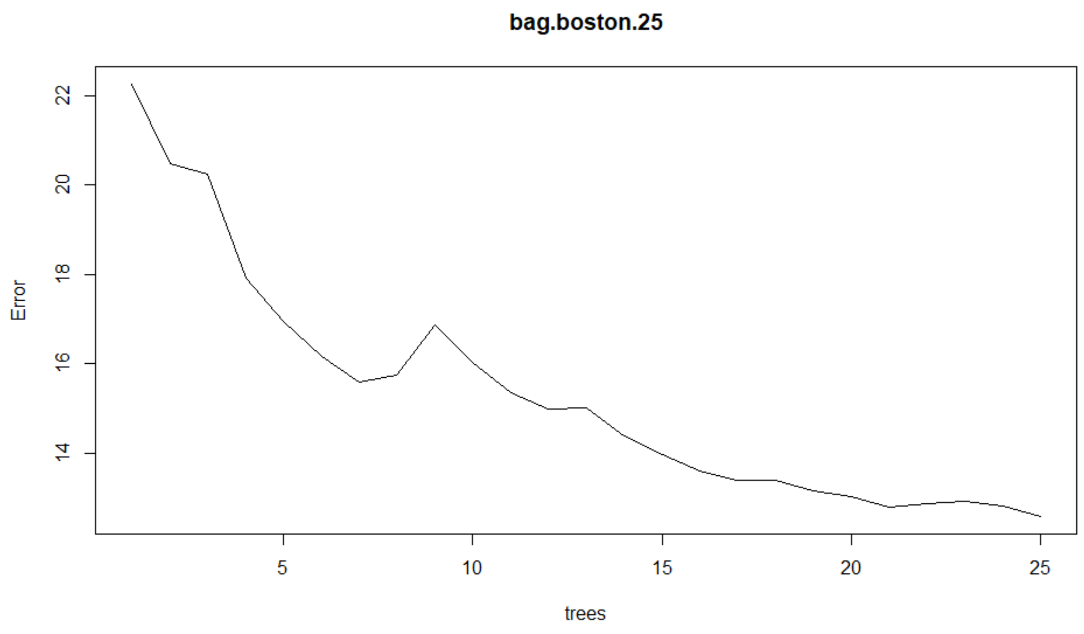
```
mse.bag <- mean((bag.pred - Boston.test$medv) ^ 2) # Get the mean squared err
or
mse.bag

## [1] 2.841752
```

Notice that the MSE is almost 1/2 of the regression tree MSE.

Technical Note: It wasn’t really necessary to compute the CV Test MSE with random splitting because the randomForest() function computes the out of bag Test MSE using bootstrapping. It is a helpful way to compare actual data against predictions.

Now, let’s change the number of bootstrapped trees, say `ntree = 25`.

```
bag.boston.25 <- randomForest(medv ~ .,
                              data = Boston.train,
                              mtry = 13,
                              ntree = 25)

bag.pred.25 <- predict(bag.boston.25, newdata = Boston.test)

plot(bag.boston.25)
```



```
mean( (bag.pred.25 - Boston.test$medv) ^ 2 )

## [1] 23.37267
```

Notice that the MSE is a bit higher than with 500 trees, although not by a lot.

Random Forest Trees

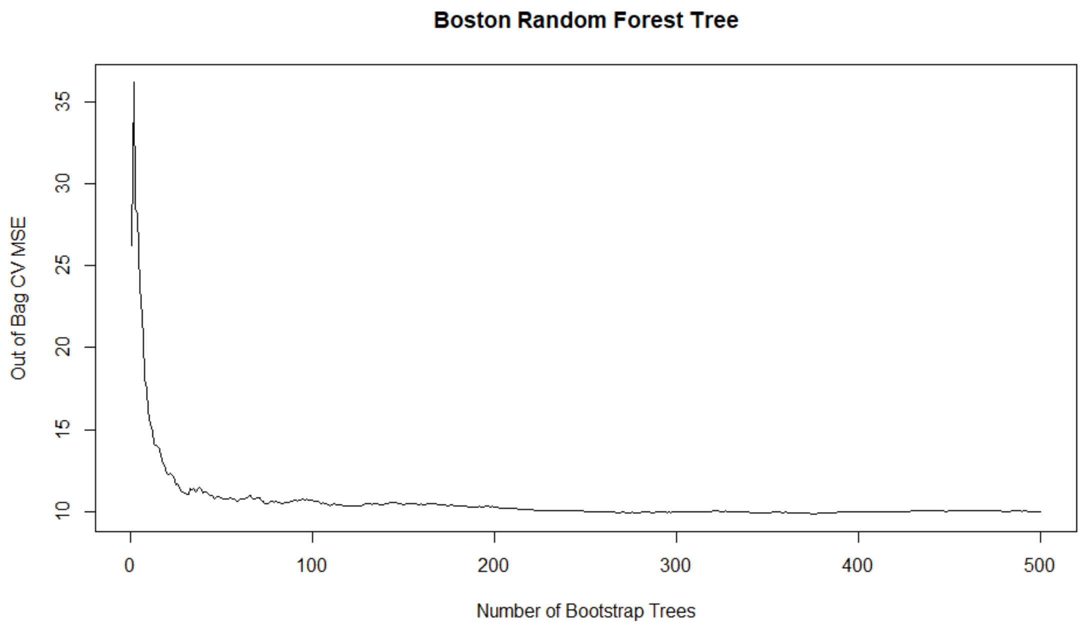
The main difference is that in Random Forest, the number of variables m used to fit the individual trees is a subset of all the available variables **p**, so **m <= p**. Bagging is a special case of Random Forest when m = p. Therefore, both methods use the same library and function **{randomForest}randomForest()**.

In Random Forest trees, we use a sample of **m** predictors from a set of **p** available predictors in the model, such that **m <= p**. The Random Forest method will select the first m predictors randomly, and then vary the chosen m predictors in each bootstrapped tree. This reduces the correlation between trees. The limitation of Bagging is that all trees are fitted with the same predictors, so results are likely to be somewhat similar or correlated. Random Forest with m < p corrects for this issue because every tree will be different. After bootstrapping hundreds or thousands of trees, all predictors will be included at one time or another.

We use the same **randomForest()** function we used for bagging, but we specify a smaller number of sampled predictors with the parameter `mtry` = than the number of available predictors. The default for mtry if omitted is p/3, that is one third of the number of predictors in the model.

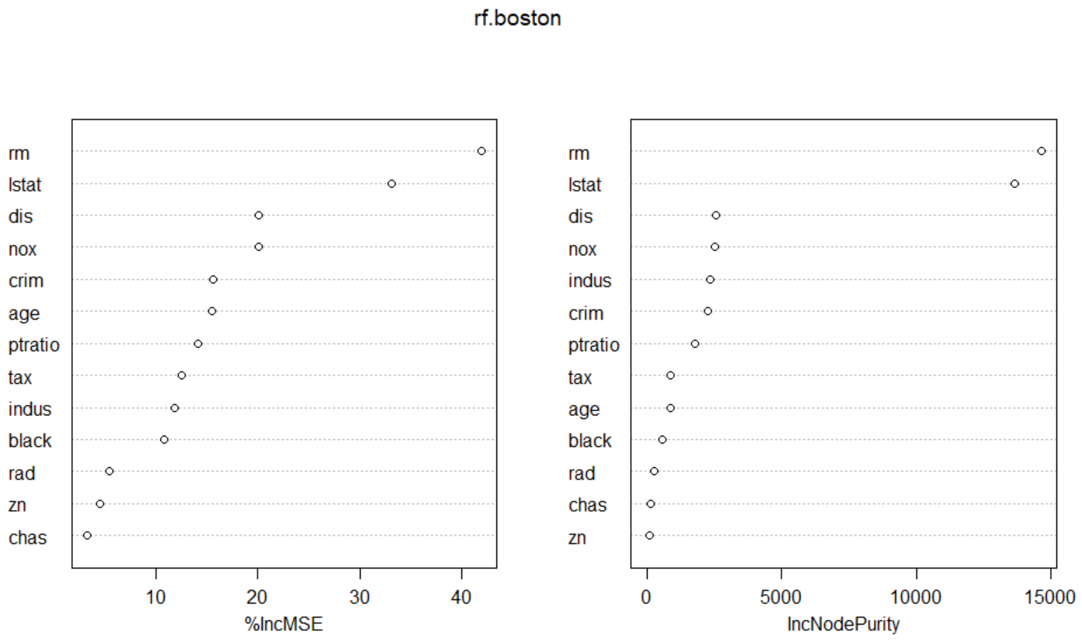
For the following illustration, we use the same function, train and test subsets we used for Bagging. I specify a Random Forest using trees with 6 random predictors each time. Everything else is the same as Bagging.

```
library(randomForest) # if not loaded already
library(MASS) # For Boston housing data set, if not loaded already



rf.boston <- randomForest(medv ~ .,
                          data = Boston,
                          mtry = 6,
                          importance = T)

rf.boston # Inspect the results

##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 6, importance = T)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 10.00386
##                     % Var explained: 88.15

plot(rf.boston$mse,
     main = "Boston Random Forest Tree",
     xlab = "Number of Bootstrap Trees",
     ylab = "Out of Bag CV MSE",
     type="l")
```



It seems like 50 or 60 bootstrapped trees may be sufficient. Let's look at the resulting variable importance.

```
varImpPlot(rf.boston) # We can also plot the results
```



```
round(
      importance(rf.boston),
      digits = 2) # To view the importance of each variable
```

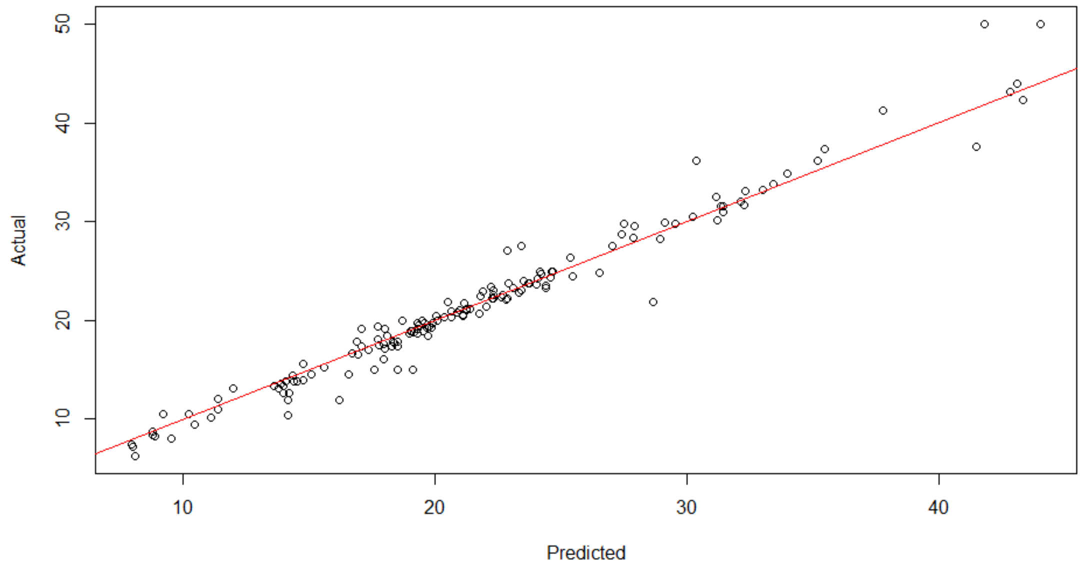
```
##         %IncMSE  IncNodePurity
## crim      15.59        2251.53
## zn         4.52          89.92
## indus     11.86        2342.04
## chas       3.23         114.20
## nox       20.09        2498.06
## rm        41.92       14641.60
## age       15.53         849.15
## dis       20.10        2554.26
## rad        5.41         239.92
## tax       12.55         859.32
## ptratio   14.13        1777.50
## black     10.75         571.88
## lstat     33.10       13641.56
```

The results show that house size (rm) and overall community wealth (lstat) and are the most important predictors.

Random Forest Tree Predictions and Cross-Validation

Let's do predictions with the trained model and the test subset:

```
set.seed(1) # To get replicable results

trsize <- 0.7
train <- sample(1:nrow(Boston), trsize * nrow(Boston))

Boston.train <- Boston[train, ]
Boston.test <- Boston[-train, ]

rf.pred <- predict(rf.boston, newdata = Boston.test)

plot(rf.pred,
     Boston.test$medv,
     xlab = "Predicted",
     ylab = "Actual") # Plot pred vs. actual

abline(0, 1, col = "red") # 45 degree line (intercept = 0; slope = 1)
```



Let’s now compute the CV Test MSE

```
mse.rf <- mean( (rf.pred - Boston.test $ medv) ^ 2 )
mse.rf

## [1] 2.695708
```

Note that the MSE for this model is even smaller than for the Bagged model

Boosted Trees

For the boosted tree illustration, we use the **gbm()** function from the Generalized Boosted Models **{gbm}** library. We also use the **Boston** housing data set in the **{MASS}** library.

```
library(gbm)
library(MASS)
```

Like Bagging and Random Forest, Boosting models fit several trees and aggregate the result. Unlike Bagging and Random Forest, Boosting does not fit several random bootstrapped trees, but it fits an initial tree with all the data, and then fits another one to explain and predict the residuals (errors), then again, etc.

Bagging and Random Forest are considered **“fast”** learning methods because the best model is generated in the first few samples and subsequent trees may or may not improve the MSE, whereas Boosting is considered to be a “slow” learning method because every new tree builds upon and improves upon the prior tree.

Boosted trees have a tuning parameter **lambda** (similar to shrinkage in Ridge and LASSO), which controls the speed of learning.

Aside: to understand this concept, imagine that you run an OLS regression or any other predictive model with certain predictors and you extract the residuals (i.e., errors). The residual values represent the portion of the outcome values that are not explained by the model. You can then build another regression model to explain (i.e., predict) those residuals. This new regression will explain some of the error variance, but will also yield new errors (smaller than the first ones, because some of the variance in the errors is already explained with the second model). Then you can fit a third regression model to explain the new residuals, and so on. You can then aggregate all the regression models, which on the aggregate, will have small residuals. Boosting applies this concept when generating trees.

When fitting boosted trees, we use the parameter:

- `distribution = "gaussian"` (i.e., normal distribution) for regression trees, and
- `distribution = "bernoulli"` for classification trees

Let's fit a model with 5000 trees, limiting the depth of each tree to 4 and using all available predictors

```
set.seed(1) # To get replicable results

boost.boston <- gbm(medv ~ ., data = Boston,
                    distribution = "gaussian", # Quantitative outcome
                    shrinkage = 0.01, # Lambda
                    cv.folds = 10, # 10FCV
                    n.trees = 5000, # Number of boosted trees
                    interaction.depth = 4) # 4 tree splits

boost.boston # Display the basic model data

## gbm(formula = medv ~ ., distribution = "gaussian", data = Boston,
##     n.trees = 5000, interaction.depth = 4, shrinkage = 0.01,
##     cv.folds = 10)
## A gradient boosted model with gaussian loss function.
## 5000 iterations were performed.
## The best cross-validation iteration was 3234.
## There were 13 predictors of which 13 had non-zero influence.
```

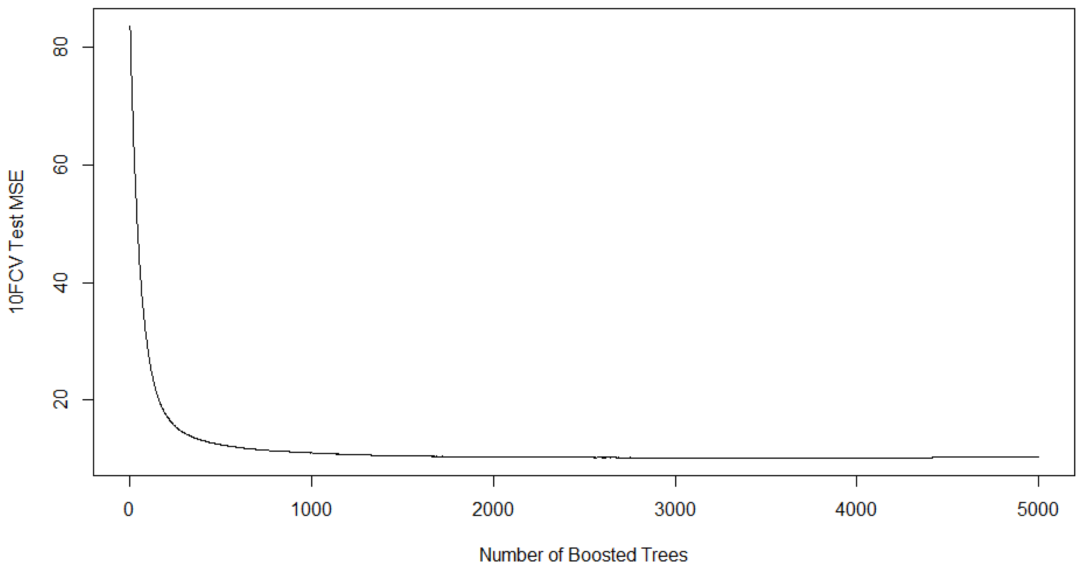
Let's find the number of boosted trees with smallest CV Test Error

```
best.num.trees <- which.min(boost.boston$cv.error) # Which tree

min.10FCV.error <- round(min(boost.boston$cv.error), # Smallest CV Test Error
                         digits = 4)

# Display result

paste("Min 10FCV Test Error =", min.10FCV.error,
      "at", best.num.trees, "trees")
```

```
## [1] "Min 10FCV Test Error = 10.1746 at 3234 trees"
```

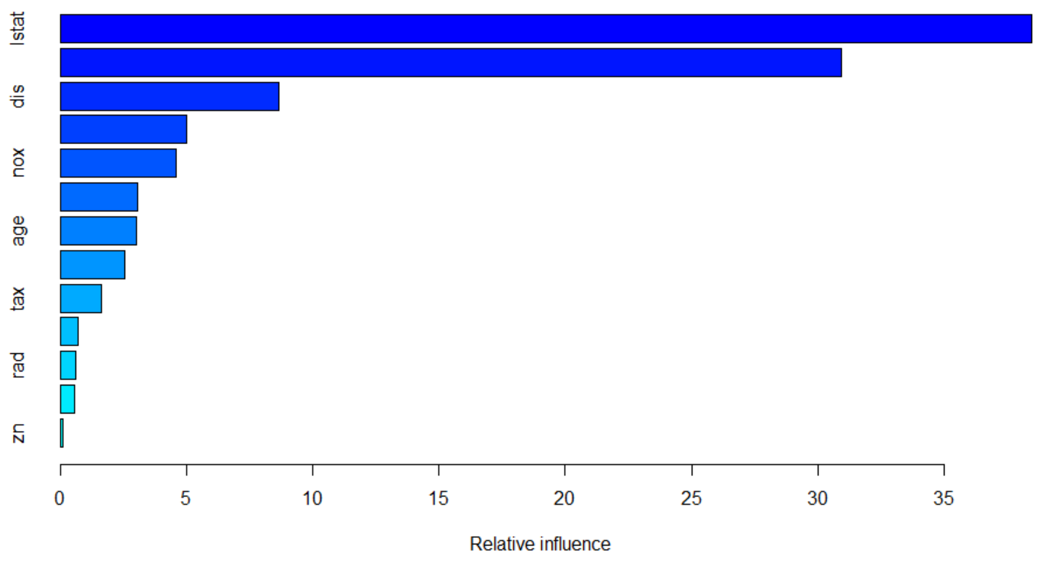
Let’s inspect how the number of trees affects the CV Test MSE

```
plot(boost.boston$cv.error,
     type = "l",
     xlab = "Number of Boosted Trees",
     ylab = "10FCV Test MSE")
```



Let’s plot the variable importance of the predictors. This graph shows the relative influence on the outcom of each predictor. The score is normalized, so that they add up to 100.

```
summary(boost.boston)
```



```
##             var     rel.inf
## lstat     lstat 38.46537753
## rm           rm 30.93608309
## dis         dis  8.66084763
## crim       crim  5.01123220
## nox         nox  4.61559339
## ptratio ptratio  3.09149607
## age         age  3.00897831
## black     black  2.57458355
## tax         tax  1.64718724
## indus     indus  0.73181333
## rad         rad  0.60145255
## chas       chas  0.56215998
## zn           zn  0.09319513
```

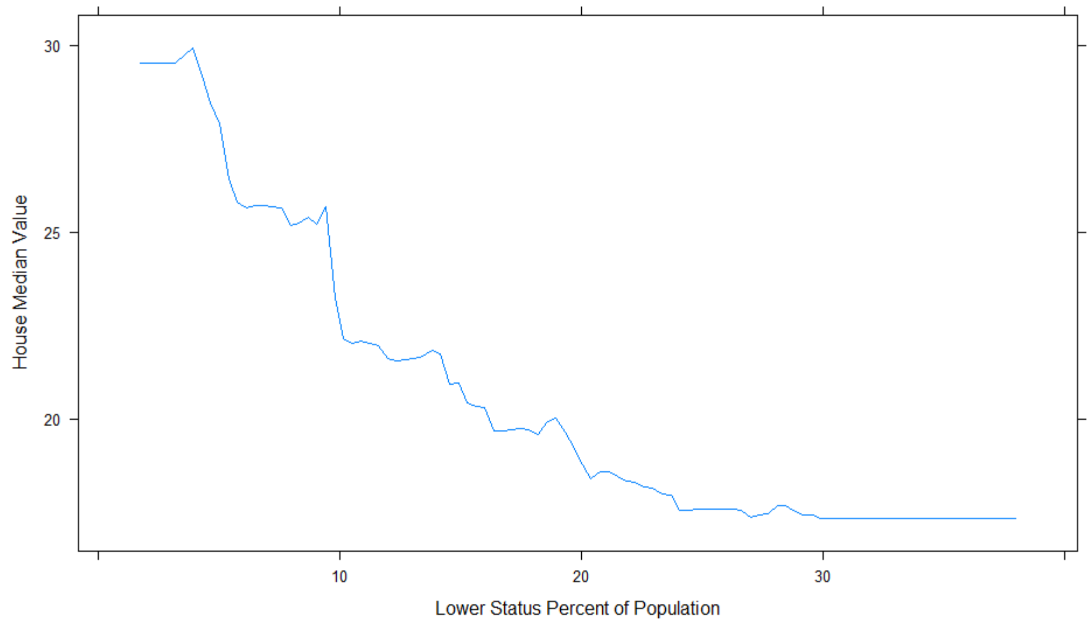
Note that **lstat** and **rm** are the most important variables, just like with Bagging and Random Forest.

Boosted Trees and Partial Dependencies

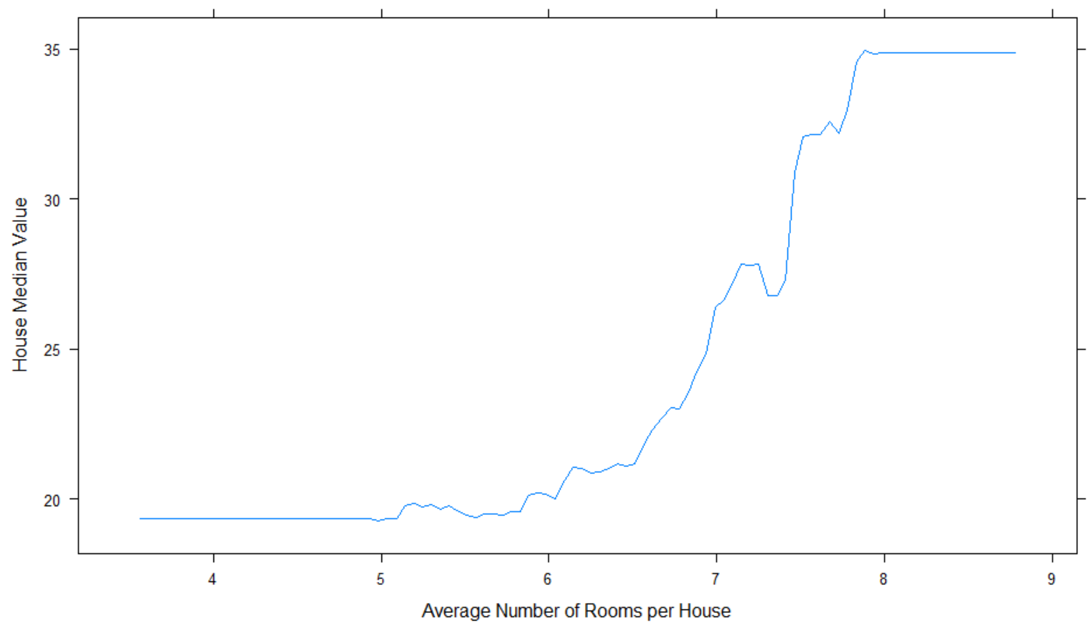
Partial dependencies show how a particular predictor affects the outcome variable, holding other predictors constant. It is important to note that this is NOT an effect. You can derive effects from the graph, but an effect is how much Y increases when X increases by 1. In contrast, in the partial dependencies graph, we can see what is the value of Y for a given value of X, holding everything else constant.

To see how predicted house median values in Boston **medv** vary with **lstat** and **rm**, we can use the **i** vector. The `boost.boston` object has an attribute property called **i**, which is a vector containing the variables used to build the model. To plot how the outcome variable is partially affected by a predictor use the parameter `i =` to specify the predictor of interest.

```r
plot(boost.boston,
     i = "lstat",
     ylab = "House Median Value",
     xlab = "Lower Status Percent of Population")
```



```r
plot(boost.boston,
     i = "rm",
     ylab = "House Median Value",
     xlab = "Average Number of Rooms per House")
```



Boosted Tree Predictions and Cross-Validation

Let’s compare results across models using the same random sample split

```
trsize <- 0.7
train <- sample(1:nrow(Boston),
                trsize * nrow(Boston)) # Train index

set.seed(1) # For replicable results

Boston.train <- Boston[train, ] # Train subset
Boston.test <- Boston[-train, ] # Test Subset

boost.boston.tr <- gbm(medv ~ .,
                       data = Boston.train,
                       distribution = "gaussian",
                       shrinkage = 0.01, # Lambda
                       cv.folds = 10, # 10FCV
                       n.trees = 5000, # Number of boosted trees
                       interaction.depth = 4) # 4 tree splits

boost.boston.tr # Basic output

## gbm(formula = medv ~ ., distribution = "gaussian", data = Boston.train,
##     n.trees = 5000, interaction.depth = 4, shrinkage = 0.01,
##     cv.folds = 10)
## A gradient boosted model with gaussian loss function.
## 5000 iterations were performed.
## The best cross-validation iteration was 4978.
## There were 13 predictors of which 13 had non-zero influence.
```

```
boost.pred <- predict(boost.boston.tr, # Predict with the trained model
                      newdata = Boston.test, # Test with the test subset data
                      n.trees = 5000)

mse.boost <- mean( (boost.pred - Boston.test$medv) ^ 2) # MSE
mse.boost

## [1] 11.77837
```

Let's compare results. Which method is better

```
round(cbind("MSE Plain Tree" = mse.tree,
            "MSE Bagging" = mse.bag,
            "MSE Random Forest" = mse.rf,
            "MSE Boosted Tree" = mse.boost),
      digits = 3)

##      MSE Plain Tree MSE Bagging MSE Random Forest MSE Boosted Tree
## [1,]         36.232       2.842             2.696           11.778
```

In this example, the boosted tree model is far superior to the other 3 models, and the plain tree is far inferior to the other 3 models. Bagging and Random Forest are somewhere in between.

Shrinkage

Boosting has a similar **shrinkage** effect, just like Ridge and LASSO regression. The difference is that Ridge and LASSO shrink the regression coefficients, whereas boosting shrinks earlier outcome predictions to weaken the model, to then be further strengthened as it learns from the errors in subsequent models. The shrinkage applies over each tree model, including the first one. A small lambda shrinks he prior tree model predictions more, thus making the early predictions less important for the final aggregated model (i.e., slow learning). Large lambdas give more weight to the initial trees, thus learning fast.

To vary the shrinkage factor lambda set the `shrinkage =` parameter. The default is **0.01**. The default for the interaction tree depth is 1. Let's lower the shrinkage parameter and retain the depth of 1 knot (i.e., a tree "stomp")

```
shrk <- 0.001 # Let's try a slow learning rate
dpth <- 1 # interaction tree depth

boost.boston.shrk <- gbm(medv ~ .,
                     data = Boston.train,
                     distribution = "gaussian",
                     n.trees = 5000,
                     interaction.depth = dpth,
                     shrinkage = shrk,
                     verbose = F)
```

Now let's do predictions with the test data

```r
medv.boost.pred <- predict(boost.boston.shrk,
                           newdata = Boston.test,
                           n.trees = 5000)

mean( (medv.boost.pred - Boston.test$medv) ^2)

## [1] 18.92473
```

Not much better

Boosted Classification Tree

Classification trees are fitted similarly to regression boosted trees, but with two important differences:

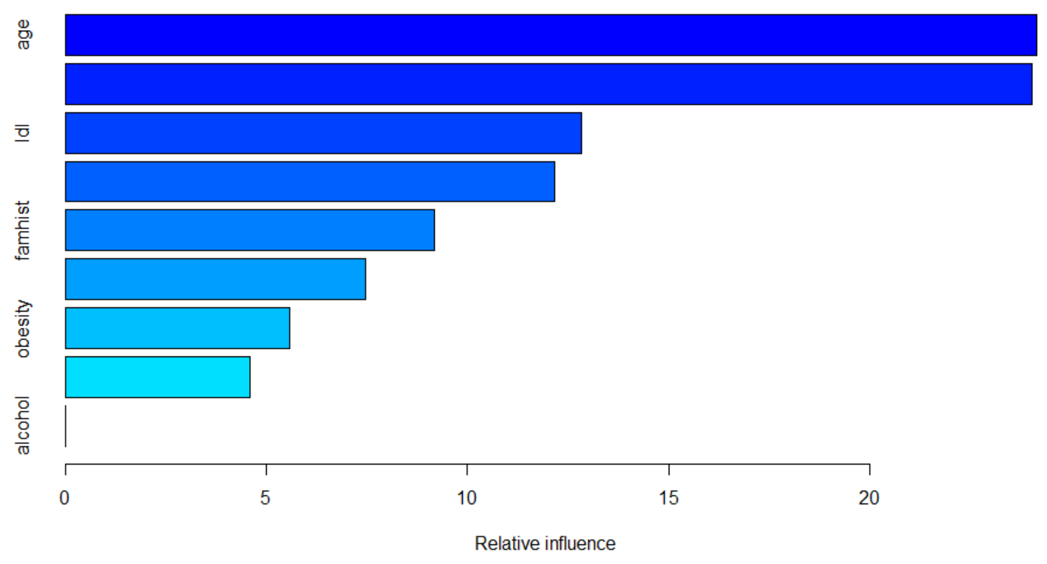
- Need to use the parameter `distribution = “bernoulli” instead of “gaussian”
- Need to convert the outcome from a factor variable to 0-1 numeric

For example, using the Heart.csv data set to predict **chd** (coronary heart disease). Let’s read the data again, for convenience

```r
heart <- read.table("Heart.csv",
                    sep = ",",
                    header = T,
                    stringsAsFactors = T)
```

We can now fit a classification boosted tree

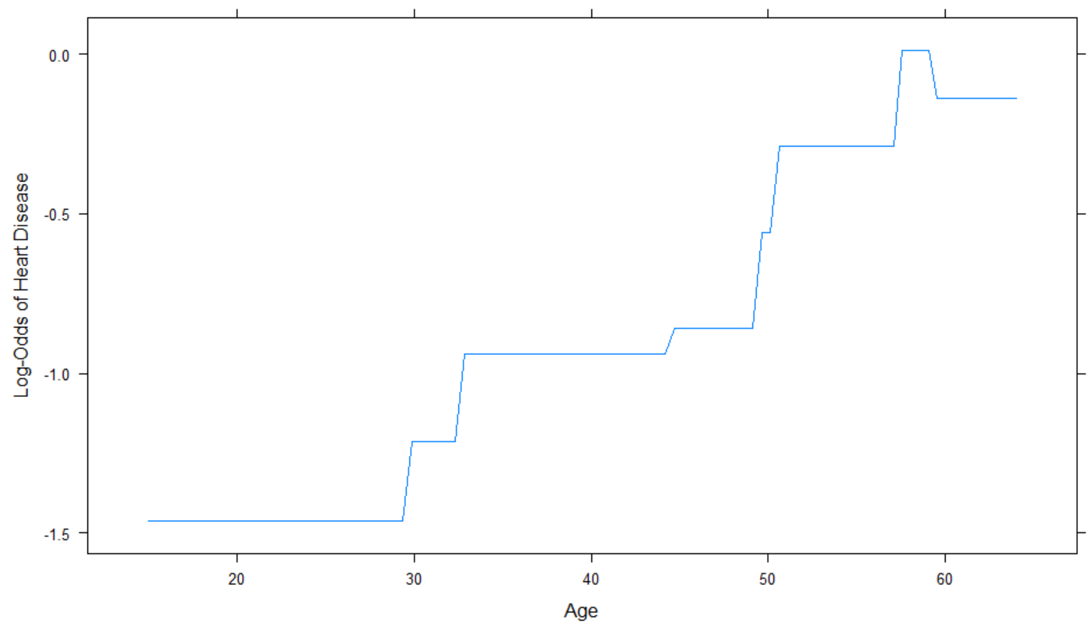
```r
boost.chd <- gbm(chd ~ . ,
                 data = heart,
                 distribution = "bernoulli")

summary(boost.chd)
```



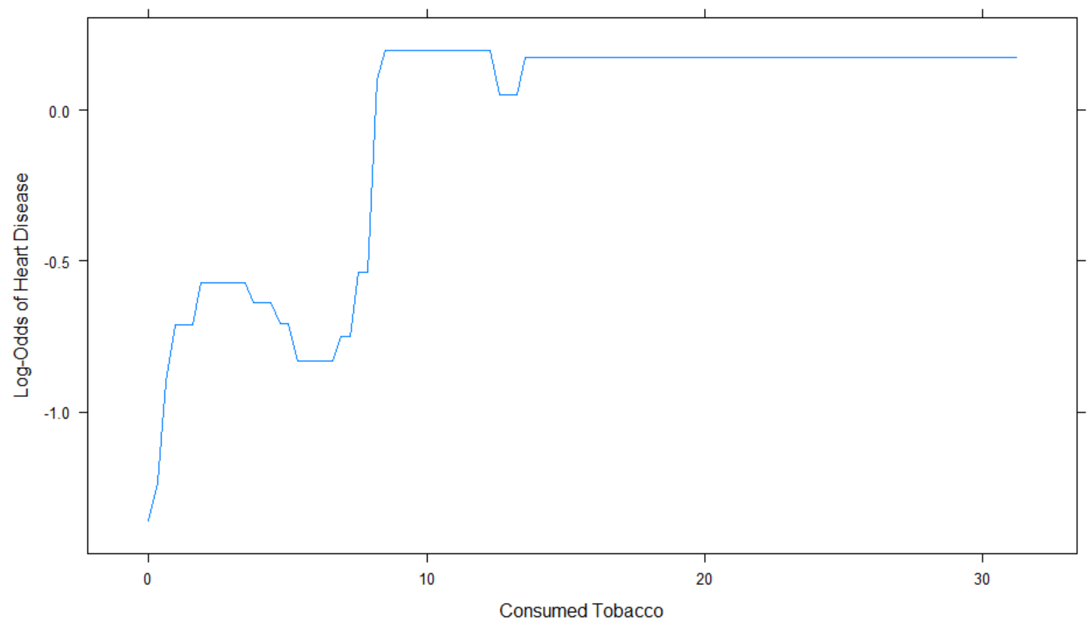
```
##                 var   rel.inf
## age             age 24.142084
## tobacco     tobacco 24.030552
## ldl             ldl 12.832443
## typea         typea 12.164290
## famhist     famhist  9.178191
## adiposity adiposity  7.469208
## obesity     obesity  5.591556
## sbp             sbp  4.591675
## alcohol     alcohol  0.000000
```

The predictors **age** and **tobacco** are the most important predictors of heart disease. Let's plot their partial correlation. The **Y axis** in the plot below depicts the **Log-Odds** of **chd = 1**. To convert to odds use `exp(Log-Odds)`.

```
plot(boost.chd,
     i = "age",
     ylab = "Log-Odds of Heart Disease",
     xlab = "Age")
```



```
plot(boost.chd,
     i = "tobacco",
     ylab = "Log-Odds of Heart Disease",
     xlab = "Consumed Tobacco")
```



For example, the Log-Odds of having coronary heart disease at the age of 60 is about - 0.2, holding everything else constant. The odds are exp(-.02) - 0.82. The probability is Odds / (Odds + 1) = 0.45 or 45%

```r
odds <- round(exp(-0.2), digits = 3)
paste("The odds of coronary heart disease are", odds)
```

```
## [1] "The odds of coronary heart disease are 0.819"
```

```r
prob <- round(odds / (1 + odds), digits = 3)
paste("The probability of coronary heart disease are", prob)
```

```
## [1] "The probability of coronary heart disease are 0.45"
```