Decision Trees and Random Forests

Laura Cline

23/10/2021

```r
rm(list = ls(all=TRUE))
```

Fitting Classification Trees

The `tree` library is used to construct classification and regression trees.

```r
#install.packages("tree")
#install.packages("gbm")
library(tree)
library(tidyverse)
```

```
## Registered S3 method overwritten by 'cli':
##   method     from
##   print.tree tree

## -- Attaching packages --------------------------------------- tidyverse 1.3.0 --

## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
library(stringr)
library(MASS)
```

```
##
## Attaching package: 'MASS'

## The following object is masked from 'package:dplyr':
##
##     select
```

```r
library(ISLR)
library(randomForest)
```

```
## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:dplyr':
```

```
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
library(gbm)
```

```
## Loaded gbm 2.1.8
```

We first use classification trees to analyze the `Carseats` dataset. In this data, `Sales` is a continuous variable, and so we begin by recoding it as a binary variable. We use the `ifelse()` function to create a variable, called `High`, which takes on a value of `Yes` if the `Sales` variable exceeds 8, and takes on a value of `No` otherwise.

```r
data <- Carseats %>%
  mutate(high = factor(if_else(Sales > 8, 1, 0)))
```

Finally, we use the `data.frame()` function to merge `High` with the rest of the `Carseats` data.

```r
names(data) = str_to_lower(names(data))
```

We now use the `tree()` function to fit a classification tree in order to predict `High` using all variables but `Sales`. The syntax of the `tree()` function is quite similar to that of the `lm()` function.

```r
# Set up initial tree
tree = tree(high ~ . -sales, data)
```

The `summary()` function lists the variables that are used as internal nodes in the tree, the number of terminal nodes, and the training error rate.

```r
summary(tree)
```

```
##
## Classification tree:
## tree(formula = high ~ . - sales, data = data)
## Variables actually used in tree construction:
## [1] "shelveloc"   "price"       "income"      "compprice"   "population"
## [6] "advertising" "age"         "us"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

We see that the training error rate is 9%. For classification trees, the deviance reported in the output of `summary()` is given by:

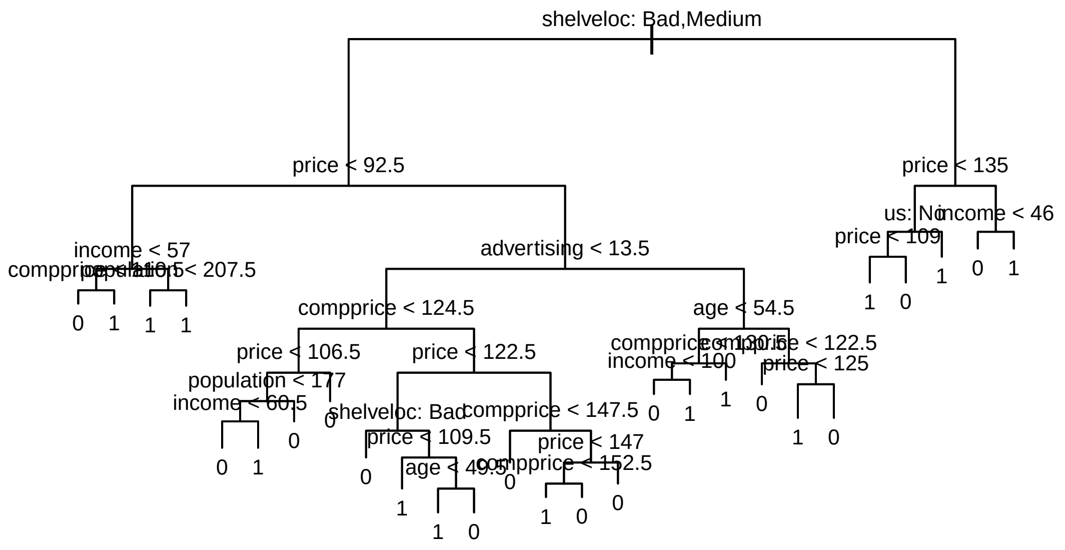
$-2\sum_m \sum_k n_{mk} log\hat{p}_{mk}$,

where n_{mk} is the number of observations in the mth terminal node that belong to the kth class. A small deviance indicates a tree that provides a good fit to the training data. The *residual mean deviance* reported is simply the deviance divided by $n - |T_0|$, which in this case is 400 - 27 = 373.

One of the most attractive properties of trees is that they can be graphically displayed. We use the `plot()` function to display the tree structure, and the `text()` function to display the node labels. The argument `pretty = 0` instructs `R` to include the category names for any qualitative predictors, rather than simply displaying a letter for each category.

We know tha trees can easily be visually interpreted, so let's check out what our test case looks like. Be wary of doing any plotting because the tree can easily grow out of control and make the graph impossible to read. We see that in the following plot of our current results:

```
plot(tree)
text(tree, pretty = 0, cex = 0.65, digits = 1)
```



The most important indicator of `Sales` appears to be shelving location since the first branch differentiates 'Good' locations from 'Bad' and 'Medium' locations.

If we just type the name of the tree object, 'R' prints output corresponding to each branch of the tree. `R` displays the split criterion (e.g., `Price < 92.5`), the number of observations in that branch, the deviance, the overall prediction of that branch (`Yes` or `No`), and the fraction of observations in that branch that take on the values of `Yes` and `No`. Branches that lead to terminal nodes are indicates using asterisks.

```
#tree
```

In order to properly evaluate the performance of a classification tree on this data, we must estimate the test error rather than simply computing the training error. We split the observations into a training set and a test set, build the tree using the training set, and evaluate its performance on the test data. The `predict()` function can be used for this purpose. In the case of a classification tree, the argument `type="class"` instructs `R` to return the actual class prediction. This approach leads to correct predictions for around 77% of the locations in the test dataset.

In order to evaluate a classification tree, we need to use training and testing sets. Let's now repeat what we did above, this time including the calculation for the test error rate.

```
# Define our training/testing sets
set.seed(2)
train <- sample_n(data, 200)
test <- setdiff(data, train)
```

```
# Run the recursive partitioning algorithm
ttree <- tree(high ~. -sales, data=train)
```

```
# Make predictions and display the confusion matrix
test_predictions <- predict(ttree, test, type='class')
table(test_predictions, test$high)
```

```
##
## test_predictions   0   1
##                0 104  33
##                1  13  50
```

```r
(104 + 50) / 200
```

```
## [1] 0.77
```

Next, we consider whether pruning the tree might lead to improved results. The function `cv.tree()` performs cross-validation in order to determine the optimal level of tree complexity; cost complexity pruning is used in order to select a sequence of trees for consideration. We use the argument `FUN=prune.misclass` in order to indicate that we want the classification error rate to guide the cross-validation and pruning process, rather than the default for `cv.tree()` function, which is deviance. The `cv.tree()` function reports the number of terminal nodes of each tree considered (`size`) as well as the corresponding error rate and the value of the cost-complexity parameter used (`k`, which corresponds to α).

We now add another layer of complexity by pruning our results. Recall that unpruned trees are prone to overfitting the data, so our method will be to watch variation in the test error rates as we increase the penalty in the number of terminal nodes. To refresh your memory, we summarize **Algorithm 8.1** below:

Algorithm 8.1: Pruning Trees

1. Grow your original tree T_0 using your training data.
2. As a function of α (the penalty parameter), define the sequence of best subtrees.
3. Use K-fold cross-validation to find the α that minimizes the average mean squared prediction error of the kth fold of the training data.
4. Find the best subtree from Step 2 using the α found in the previous step.

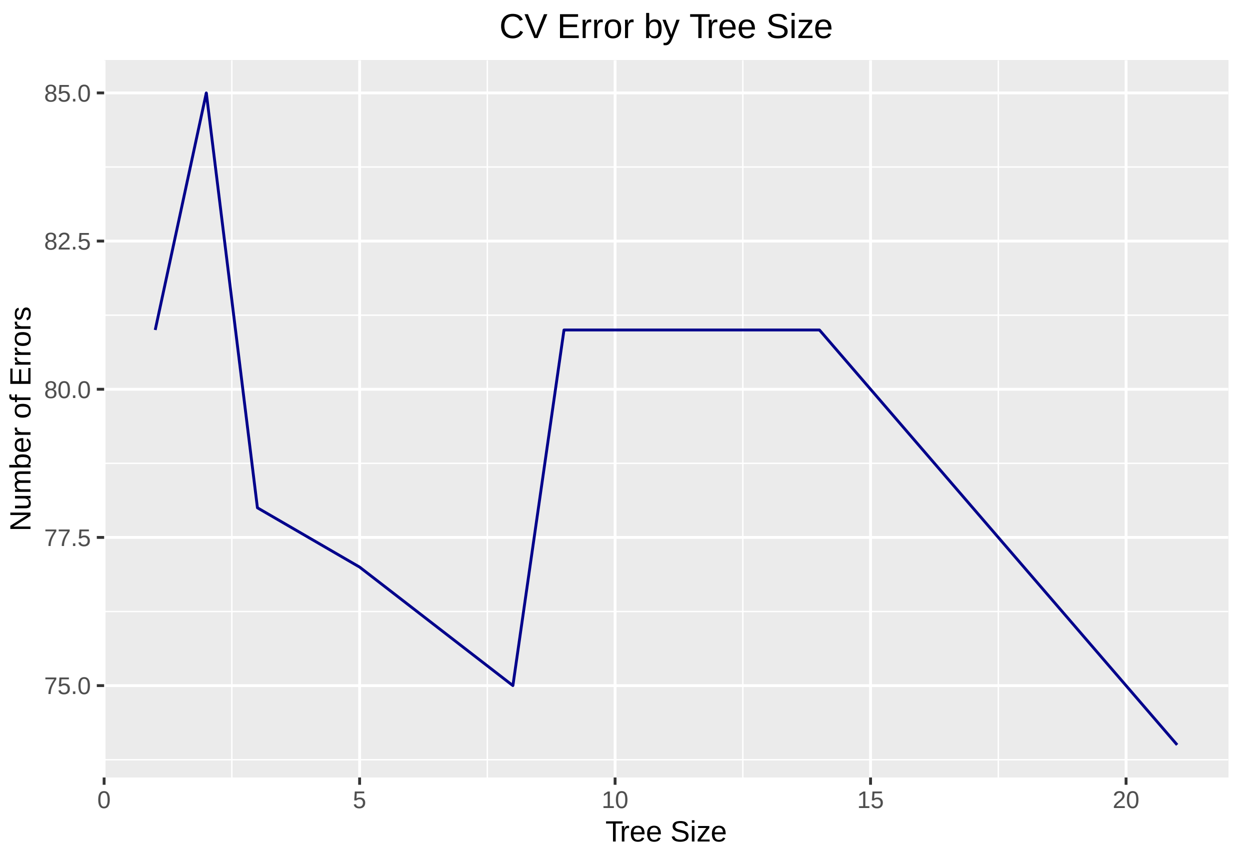
Luckily, `tree::cv.tree` will be doing most of the work for us. It will perform cross-validation required to determine the optimal tree size. It also allows us to choose the function by which the tree is pruned. In this case, pruning will be guided by the classification error rate.

```r
set.seed(3)
cv_tree <- cv.tree(ttree, FUN = prune.misclass)
cv_tree
```

```
## $size
## [1] 21 19 14  9  8  5  3  2  1
##
## $dev
## [1] 74 76 81 81 75 77 78 85 81
##
## $k
## [1] -Inf  0.0  1.0  1.4  2.0  3.0  4.0  9.0 18.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

Note that, despite the name, `dev` corresponds to the cross-validation error rate in this instance. The tree with 21 terminal nodes results in the lowest cross-validation error rate, with 74 cross validation errors. We plot the error rate as a function of both `size` and `k`.

```r
library(ggplot2)

ggplot(data = data.frame(cv_tree$size, cv_tree$dev),
  aes(x = cv_tree$size, y = cv_tree$dev)) +
  geom_line(color = "darkblue") +
  labs(x = "Tree Size", y = "Number of Errors", title = "CV Error by Tree Size") +
  theme(plot.title = element_text(hjust = .5))
```



We now apply the `prune.misclass()` function in order to prune the tree to obtain the nine-node tree.

Now that we know exactly how many terminal nodes we want, we prune our tree with `prune.misclass()` to obtain the optimal tree. Then check to see if this tree performs any better on the testing set than the base tree T_0 did.

```
pruned <- prune.misclass(ttree, best=21)
```

How well does this pruned tree perform on the test dataset? Once again, we apply the `predict()` function.

```
test_predictions <- predict(pruned, data=test, type='class')
table(test_predictions, test$high)
```

```
##
## test_predictions  0  1
##                0 79 49
##                1 38 34
```

```
(77 + 38)/200
```

```
## [1] 0.575
```

Now, 57% of the test observations are correctly classified, so in this case, the pruning process has produced a more interpretable tree, but the classification accuracy has decreased.

Fitting Regression Trees

Here we fit a regression tree to the `Boston` dataset. First we create a training set and fit the tree to the training data.

Not much changes in terms of code when we switch to regression trees, so this section will pretty much be a recap of the previous one, just using different data. We pull the `Boston` dataset from the `MASS` library for this section.

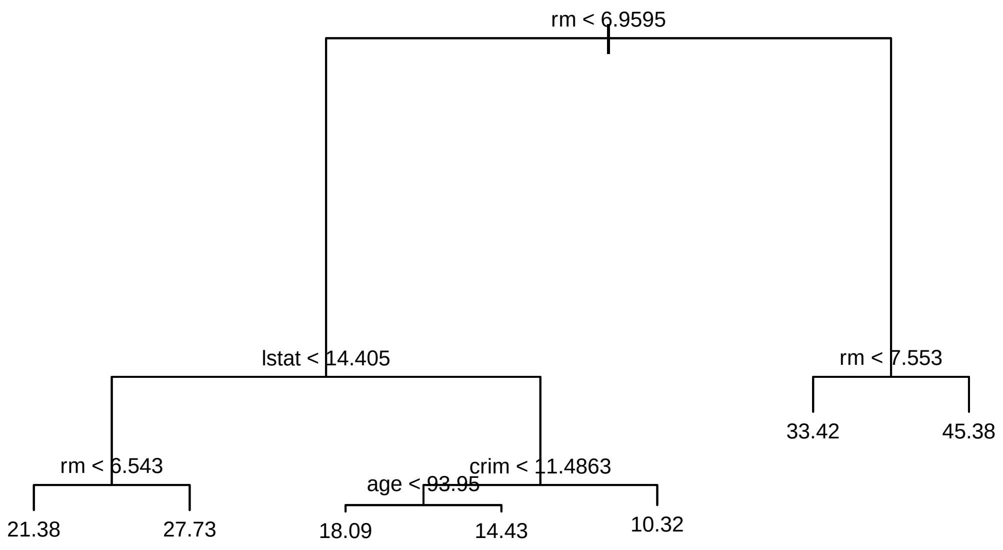
```
Boston <- MASS::Boston
set.seed(1)
train <- sample_frac(Boston, 0.5)
test <- setdiff(Boston, train)
```

```
tree_train <- tree(medv ~ ., data=train)
summary(tree_train)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = train)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"  "age"
## Number of terminal nodes:  7
## Residual mean deviance:  10.38 = 2555 / 246
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.1800  -1.7770  -0.1775   0.0000   1.9230  16.5800
```

Notice that the output of `summary()` indicates that only four of the variables have been used in constructing the tree. In the context of a regression tree, the deviance is simply the sum of squared errors for the tree. We now plot the tree.

```
plot(tree_train)
text(tree_train, pretty = 0, cex = 0.65)
```



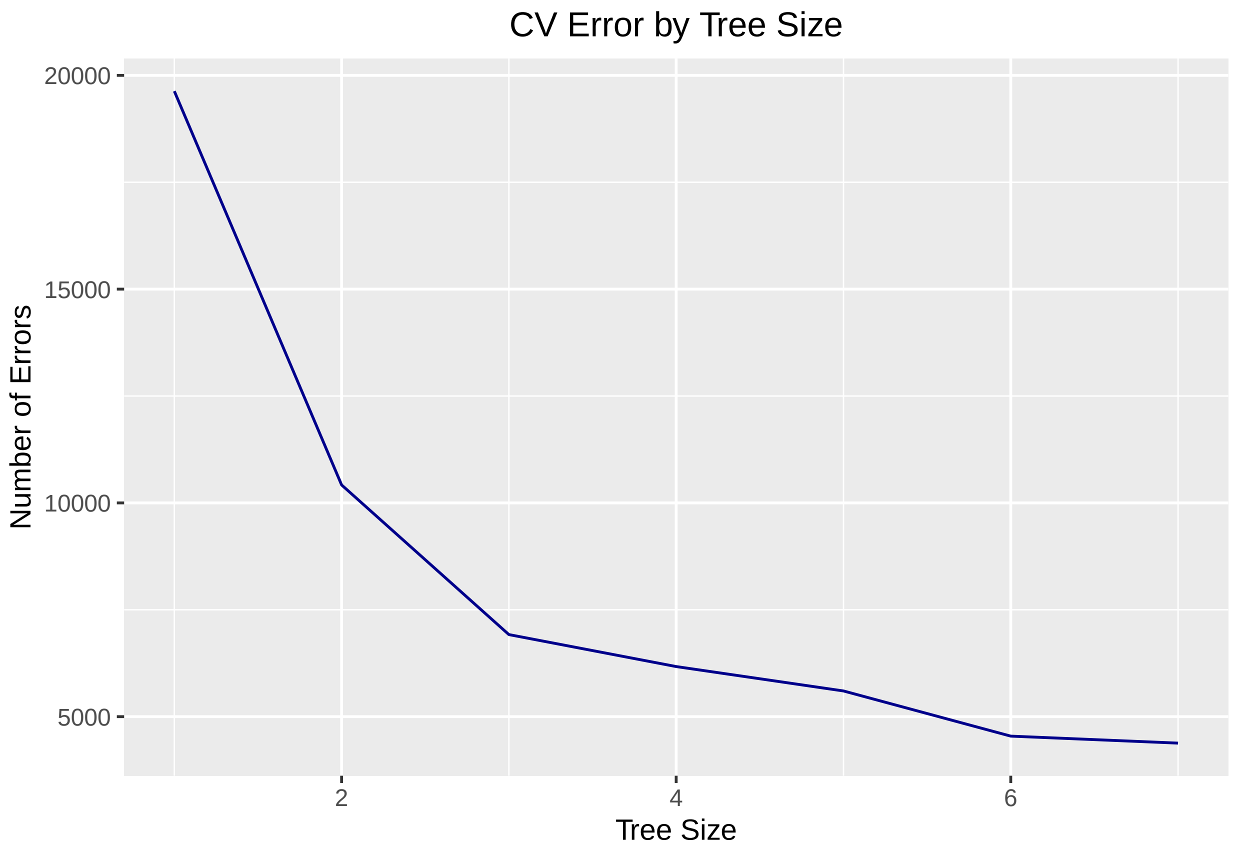
As you can see, `rm < 6.959` is the first partition in the tree. The variable means the average number of rooms per dwelling so lower values of `rm` means less average rooms on the left side of the tree, this suggests that houses with more rooms end up with much larger median house prices.

The tree predicts a median house price of $25,380 for larger homes with more than seven rooms.

Now we use the `cv.tree()` function to see whether pruning the tree will improve performance.

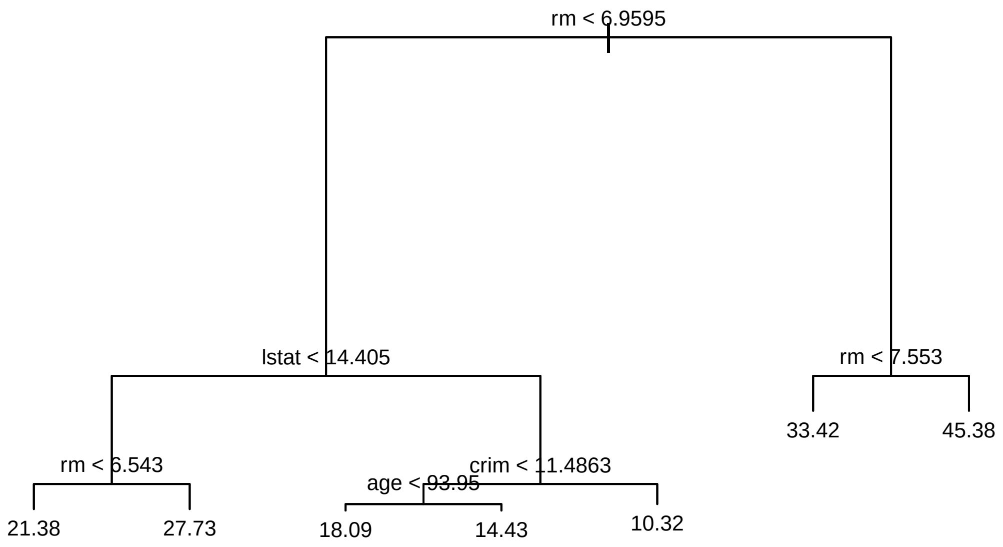
```r
cv_tree <- cv.tree(tree_train)
```

```r
# Get an idea of change in error by changing tree size
ggplot(data = data.frame(cv_tree$size, cv_tree$dev), aes(x = cv_tree$size, y = cv_tree$dev)) +
  geom_line(color = "darkblue") +
  labs(x = "Tree Size", y = "Number of Errors", title = "CV Error by Tree Size") +
  theme(plot.title = element_text(hjust = 0.5))
```



In this case, the most complex tree is selected by cross-validation. However, if we wish to prune the tree, we could do so as follows using the `prune.tree()` function:

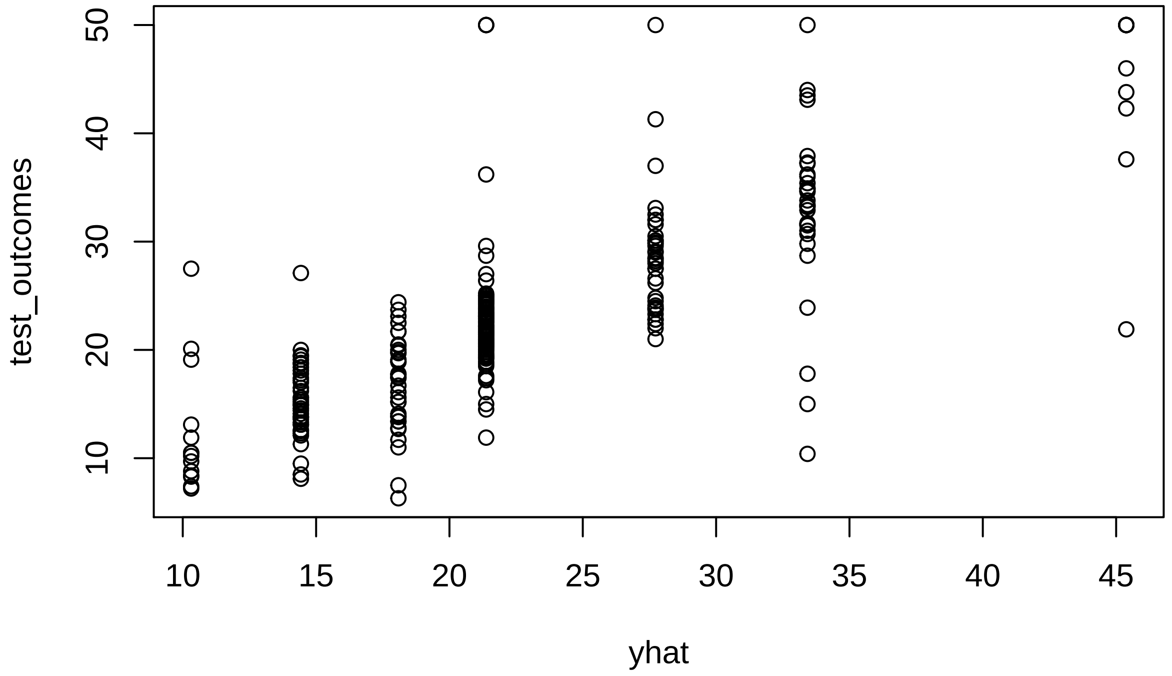
```r
prune.boston = prune.tree(tree_train, best= 7)
plot(prune.boston)
text(prune.boston, pretty = 0, cex = 0.65)
```



In keeping with cross-validation results, we use the unpruned tree to make predictions on the test set.

```
# Predict, plot and calculate MSE
yhat <- predict(tree_train, newdata=test)
test_outcomes <- test$medv
```

```
plot(yhat, test_outcomes)
```



```
mean((yhat - test_outcomes)^2)
```

```
## [1] 35.28688
```

In other words, the test set MSE associated with the regression tree is 35.29. The square root of the MSE is therefore around 5.94, indicating that this model leads to test predictions that are within $5,940 of the true median home value for the suburb.

Bagging and Random Forests

Here we apply bagging and random forests to the `Boston` data, using the `randomForest` package in `R`. The exact results obtained in this section may depend on the version of R and the version of the `randomForest` package installed on your computer. Recall that bagging is simply a special case of a random forest with $m = p$. Therefore, the `randomForest()` function can be used to perform both random forests and bagging.

We'll be using the same data from the previous section and the `randomForest` package to help us accomplish some simple examples. We begin with a bagging example, where all predictors are used in each split. We perform bagging as follows:

```
set.seed(1)
train <- sample_frac(Boston, 0.5)
test <- setdiff(Boston, train)
```

```
# Set up the randomForest for the bagging case (all vars included)
bag <- randomForest(medv ~ ., data=train, mtry = 13, importance = TRUE)
bag
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = train, mtry = 13, importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.33119
##                     % Var explained: 85.26
```

The argument mtry = 14 indicates that all 13 predictors should be considered for each split of the tree - in other words, that bagging should be done. How does this bagged model perform on the test set?

```
# Calculate MSE of the testing set for the bagged regression tree
yhat <- predict(bag, test)
mean((yhat - test$medv)^2)
```

```
## [1] 23.4579
```

The test set MSE associated with the bagged regression tree is 23.46, almost half that obtained using an optimally-pruned single tree. We could change the number of trees grown by `randomForest()` using the `ntree` argument.

```
bag.boston <- randomForest(medv ~ ., data=train, mtry = 13, ntree = 25)
yhat <- predict(bag.boston, test)
mean((yhat - test$medv)^2)
```

```
## [1] 22.99145
```

Growing a random forest proceeds in exactly the same way, except that we use a smaller value of the `mtry` argument. By default, `randomForest()` uses $p/3$ variables when building a random forest of regression trees, and $\sqrt{p}$ variables when building a random forest of classification trees. Here we use `mtry = 6`.

Compare the MSE of the bagged random forest to the optimally pruned single tree found earlier - it's much lower. We manually change the amount of variables at each split in the above bagging example, but we might achieve better results using a general random forest. By default, `randomForest` uses $p/3$ variables when building a forest of regression trees and $\sqrt{p}$ for classification trees. In the following example, we will use `mtry = 6` ($m \approx p/2$).

```
forest = randomForest(medv ~ ., data=train, mtry = 6, importance = TRUE)
```

```
yhat <- predict(forest, test)
mean((yhat - test$medv)^2)
```

```
## [1] 20.16422
```

We find that this approach worked - our MSE is now reduced to 19.88, lower than the previous two methods we tried.

Using the `importance()` function, we can view the importance of each variable.

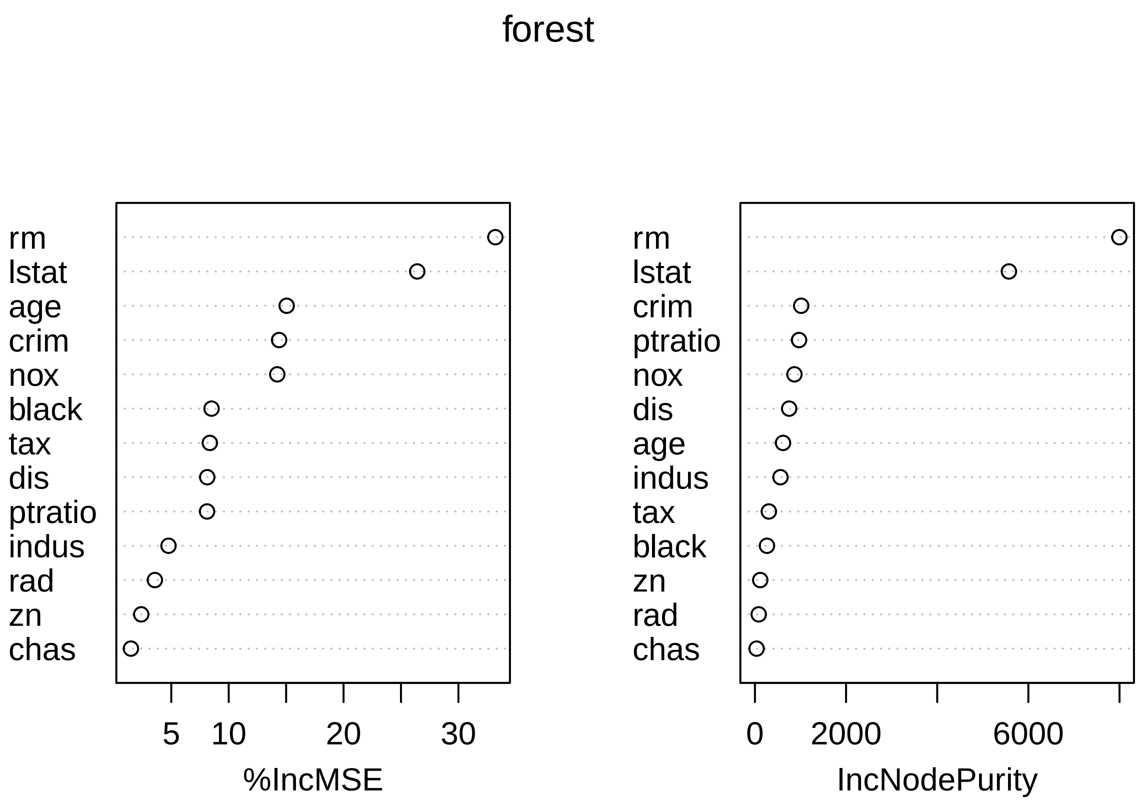
```
importance(forest)
```

```
##            %IncMSE IncNodePurity
## crim     14.389520    1015.38819
## zn        2.383675     116.75818
## indus     4.759755     560.00649
## chas      1.485182      36.49459
## nox      14.227996     866.91472
## rm       33.219106    7997.53209
## age      15.046634     616.54222
## dis       8.131485     751.23862
## rad       3.571826      83.08033
## tax       8.356359     305.72575
## ptratio   8.115858     967.88590
## black     8.507384     264.38182
## lstat    26.418993    5573.29309
```

The first column represents the mean decrease in accuracy of the prediction when the variable is removed from the model, and the second column is a measure of the total decrease in node purity resulting from splits over that variable (averaged over all of the trees).

Two measures of variable importance are reported. The former is based upon the mean decrease of accuracy in predictions on the out of bag samples when a given variable is excluded from the model. The latter is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees. In the case of regression trees, the node impurity is measured by the training RSS, and for classification trees by the deviance. Plots of these importance measures can be produced using the `varImpPlot()` function.

```
varImpPlot(forest)
```



The results indicate that across all of the trees considered in the random forest, the average number of rooms (`rm`) and the wealth level of the community (`lstat`) are by far the two most important variables.

Boosting

Here we use the `gbm` package, and within it the `gbm()` function, to fit boosted regression trees to the `Boston` dataset. we run `gbm()` with the option of `distribution="gaussian"` since this is a regression problem; if it were a binary classification problem, we would use `distribution=bernoulli"`. The argument `n.trees=5000` indicates that we want 5,000 trees, and the option `interaction.depth=4` limits the depth of each tree.

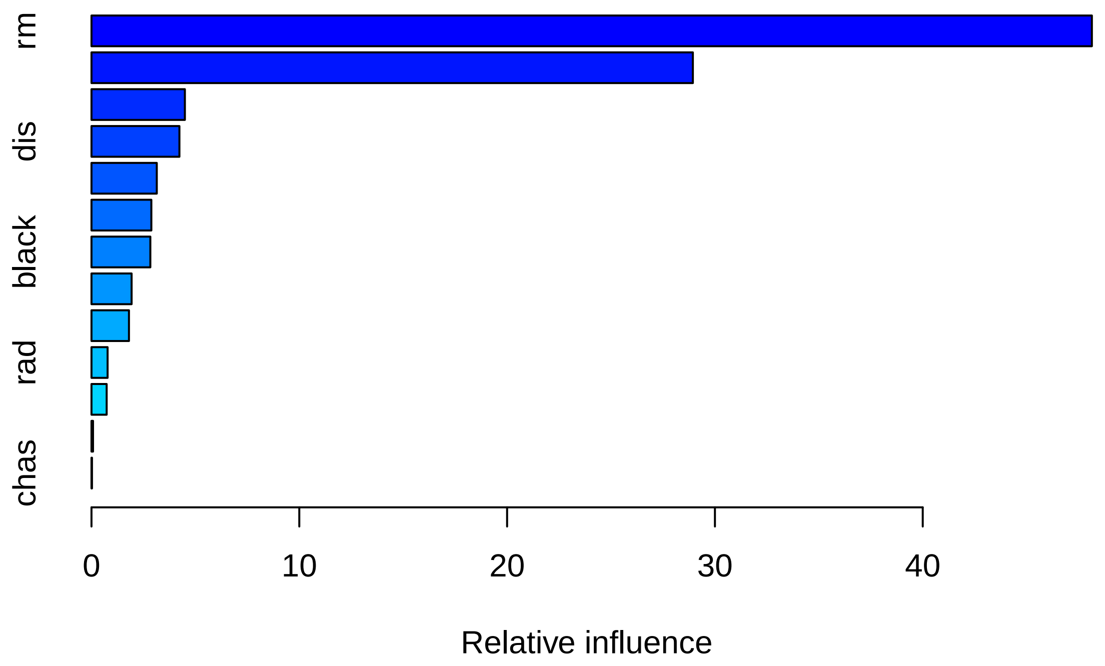
We'll be using the `gbm` package to help us fit boosted regression trees to the `Boston` dataset, which you should be familiar with by now.

```
set.seed(1)
train <- sample_frac(Boston, 0.5)
test <- setdiff(Boston, train)
```

```
# Regression => distr = gaussian
boosted <- gbm(medv ~ ., train, distribution="gaussian", n.trees=5000, interaction.depth = 4)
```

The `summary()` function produces a relative influence ploy and also outputs the relative influence statistics.

```
# Summarize and produce a quick plot to highlight importance of variables
summary(boosted)
```

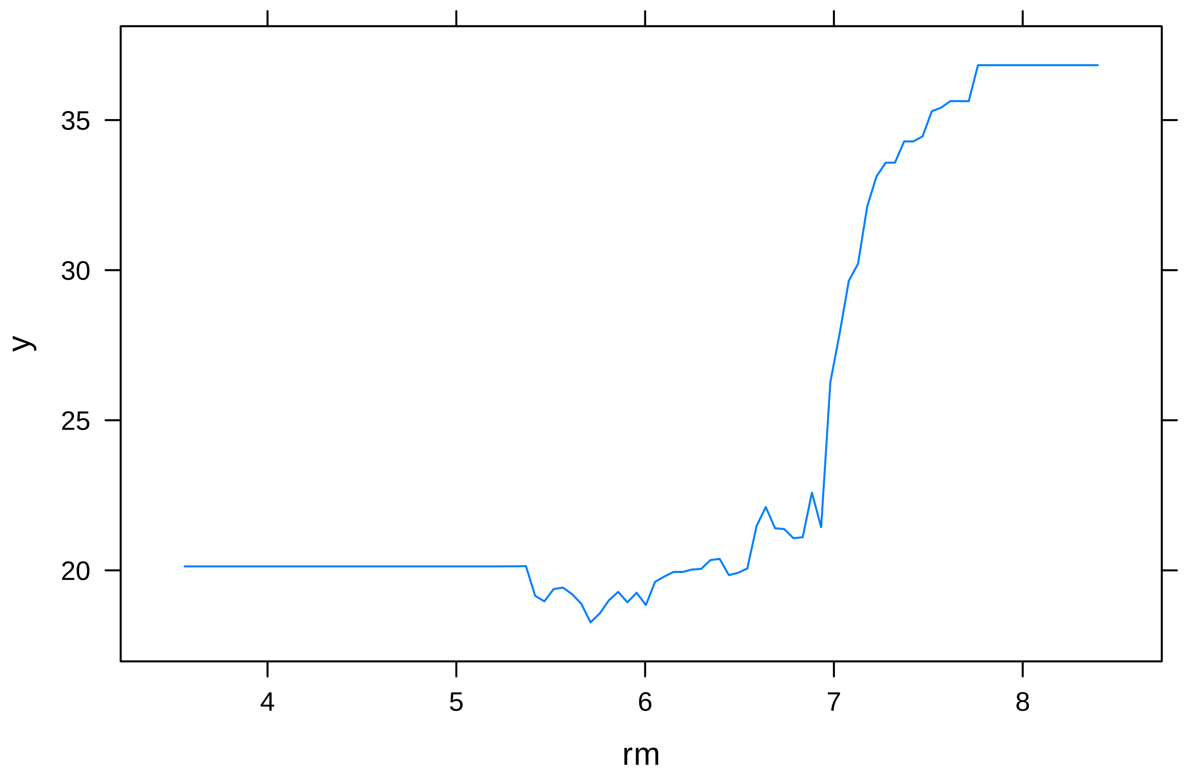


```
##               var     rel.inf
## rm             rm 48.13967682
## lstat       lstat 28.93851185
## crim         crim  4.49413146
## dis           dis  4.23182696
## age           age  3.14221169
## nox           nox  2.88094283
## black       black  2.83238772
## ptratio ptratio  1.93050932
## tax           tax  1.80427054
## rad           rad  0.77569461
## indus       indus  0.73110525
## zn             zn  0.07442923
## chas         chas  0.02430170
```

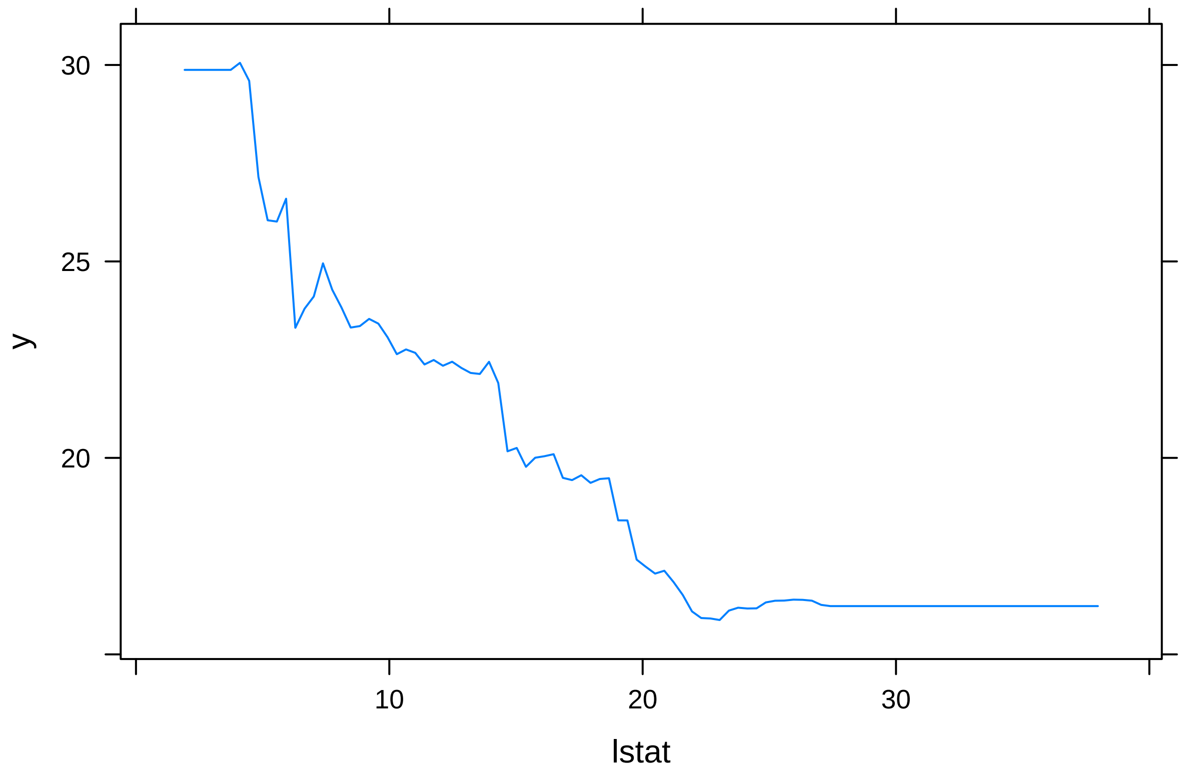
We see that `rm` and `lstat` are by far the most important variables. We can also produce *partial dependence plots* for these two variables. These plots illustrate the marginal effect of the selected variables on the response after *integrating* out the other variables. In this case, as we might expect, median house prices are increasing with `rm` and decreasing with `lstat`.

Let's plot the marginal effect of these two variables, `lstat` and `rm`.

```
par(mfrow = c(1,2))
plot(boosted, i = 'rm')
```



```
plot(boosted, i = 'lstat')
```



Alright, this just confirmed what we should have already been expecting: median house values are decreasing with `lstat` and increasing in `rm`.

We now use the boosted model to predict `medv` on the test set.

Let's now test how well this boosted regression tree performs on the testing data.

```
yhat <- predict(boosted, newdata = test, n.trees=5000)
mean((yhat - test$medv)^2)
```

```
## [1] 19.37033
```

Not amazing, but not bad. The boosted model performed just about the same as random foresta and superior to that of the bagging model, but we might be able to squeeze out some extra performance by changing the shrinkage parameter λ.

The test MSE obtained is 19.37; similar to the test MSE for random forests and superior to that for bagging. If we want to, we can perform boosting with a different value of the shrinkage parameter λ. The default value is 0.01, but this is easily modified. Here we take $\lambda = 0.2$.

```
boosted <- gbm(medv ~ ., train, distribution="gaussian", n.trees=5000, interaction.depth = 4, shrinkage
yhat <- predict(boosted, newdata=test, n.trees=5000)
mean((yhat - test$medv)^2)
```

```
## [1] 18.68911
```

Changing the shrinkage parameter actually made a difference - we are now just slightly under what we got from our previous model where it was equal to 0.001.

Excercises

Question Three

Consider the Gini index, classification error and entropy in a simple classification setting with two classes. Create a single plot that displays each of these quantities as a function of $\hat{p}_{m1}$. The x-axis should display $\hat{p}_{m1}$, ranging from 0 to 1, and the y-axis should display the value of the Gini index, classification error and entropy.

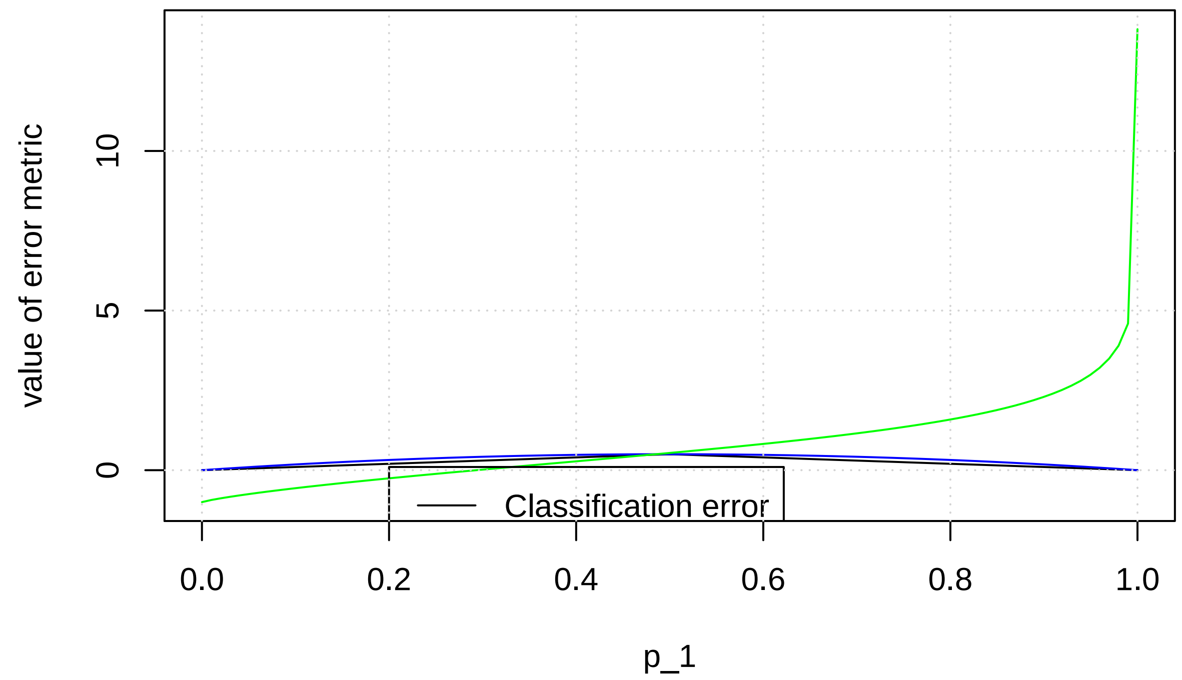
```
p1 <- seq(0+1e-06, 1-1e-6, length.out=100)
p2 = 1 - p1
```

```
# The miscalculation error rate:
E = 1 - apply(rbind(p1, p2), 2, max)
```

```
# The Gini index:
G = p1 * (1 - p1) + p2 * (1-p2)
```

```
# The cross entropy:
D = - (p1 * log(p1) + p2 + log(p2))
```

```
plot(p1, E, type='l', col='black', xlab = 'p_1', ylab='value of error metric', ylim=c(min(c(E,G,D)), ma
lines(p1, G, col='blue')
lines(p1, D, col='green')
legend(0.2, 0.1, c('Classification error', 'Gini index', 'Cross entropy'), col=c('black', 'blue', 'gree
grid()
```



Question Seven

In the lab, we applied random forests to the `Boston` data using `mtry=6` and using `ntree = 25` and `ntree = 500`. Create a plot displaying the test error resulting from random forests on this dataset for a more comprehensive range of values for `mtry` and `ntree`. Describe the results obtained.

```
set.seed(0)
```

```
n  = nrow(Boston)
p = ncol(Boston) - 1 # one column is the response we are trying to model, i.e., "medv"
```

```
train = sample(1:n, n/2)
test = (1:n)[-train]
```

```
ntree_to_test = seq(from=1, to=500, by=10)
```

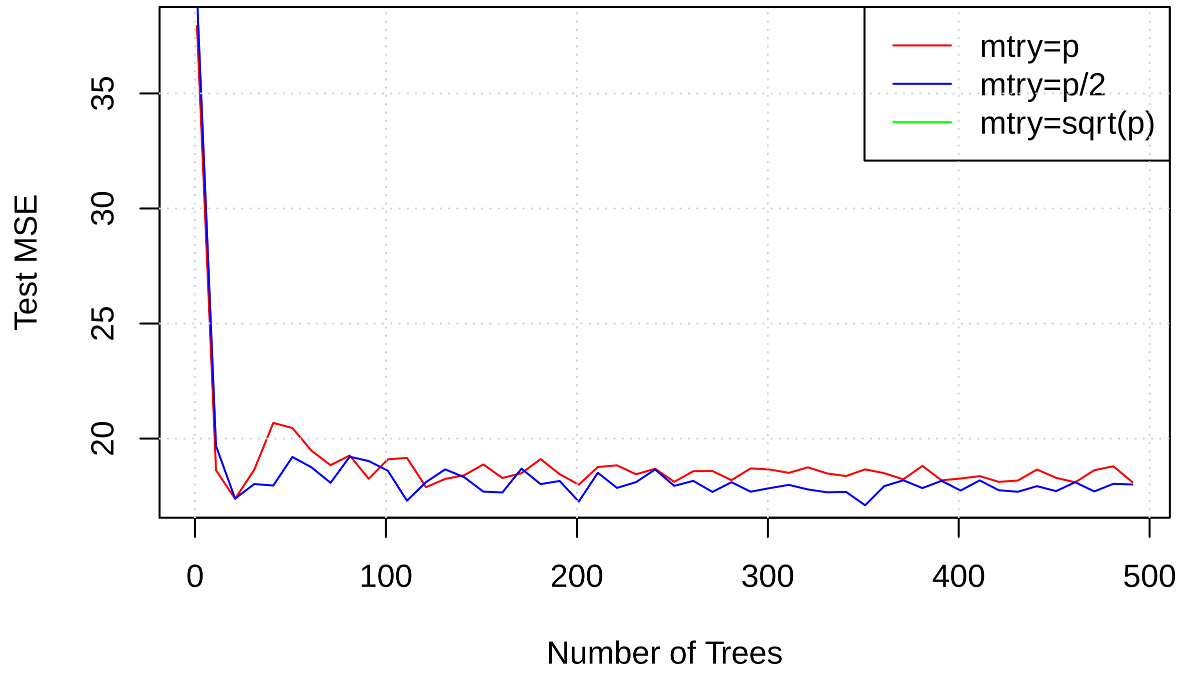
```
# For a number of mtry values and a number of trees look at the test error rate:
```

```
# For mtry == p:
mse.bag = rep(NA, length(ntree_to_test))
for(nti in 1:length(ntree_to_test)){
  nt = ntree_to_test[nti]

  # Grow a tree with "nt" trees:
  boston.bag = randomForest(medv ~ ., data=Boston, mtry = p, ntree = nt, importance=TRUE, subset=train)

  # Make predictions with this tree on the test dataset:
  y_hat = predict(boston.bag, newdata=Boston[test,])
  mse.bag[nti] = mean((Boston[test,]$medv - y_hat)^2)
}
```

```
# For mtry = p/2:
mse.p_over_two = rep(NA, length(ntree_to_test))
for(nti in 1:length(ntree_to_test)){
  nt = ntree_to_test[nti]

  # Grow the tree with "nt" trees:
  boston.bag = randomForest(medv ~ ., data=Boston, mtry=p/2, ntree=nt, importance=TRUE, subset=train)

  # Make predictions with this tree on the test dataset:
  y_hat = predict(boston.bag, newdata=Boston[test,])

  mse.p_over_two[nti] = mean((Boston[test,]$medv - y_hat)^2)
}
```

```
# Run random forest with mtry=sqrt(p) and test on test set
mse.sqrt_p = rep(NA, length(ntree_to_test))
for(nti in 1:length(ntree_to_test)){
  nt = ntree_to_test[nti]

  # Grow a tree with "nt" trees:
  boston.bag = randomForest(medv ~ ., data=Boston, mtry=sqrt(p), ntree=nt, importance=TRUE, subset=trai

  # Make predictions with this tree on the test dataset:
  y_hat = predict(boston.bag, newdata=Boston[test,])

  mse.sqrt_p[nti] = mean((Boston[test,]$medv - y_hat)^2)
}
```

```
plot(ntree_to_test, mse.bag, xlab = "Number of Trees", ylab="Test MSE", col='red', type='l')
lines(ntree_to_test, mse.p_over_two, xlab="Number of Trees", ylab="Test MSE", col='blue', type='l')
legend('topright', c('mtry=p', 'mtry=p/2', 'mtry=sqrt(p)'), col=c('red', 'blue', 'green'), lty=c(1,1,1)
grid()
```



Question Eight

In the lab, a classification tree was applied to the `Carseats` dataset after converting `Sales` into a qualitative response variable. Now we will seek to predict `Sales` using regression trees and related approaches, treating the response as a quanitative variable.

A. Split the dataset into a training set and test set

```
set.seed(0)

n = nrow(Carseats)
p = ncol(Carseats)-1 # Remove the column we seek to predict, i.e., "Sales"
```

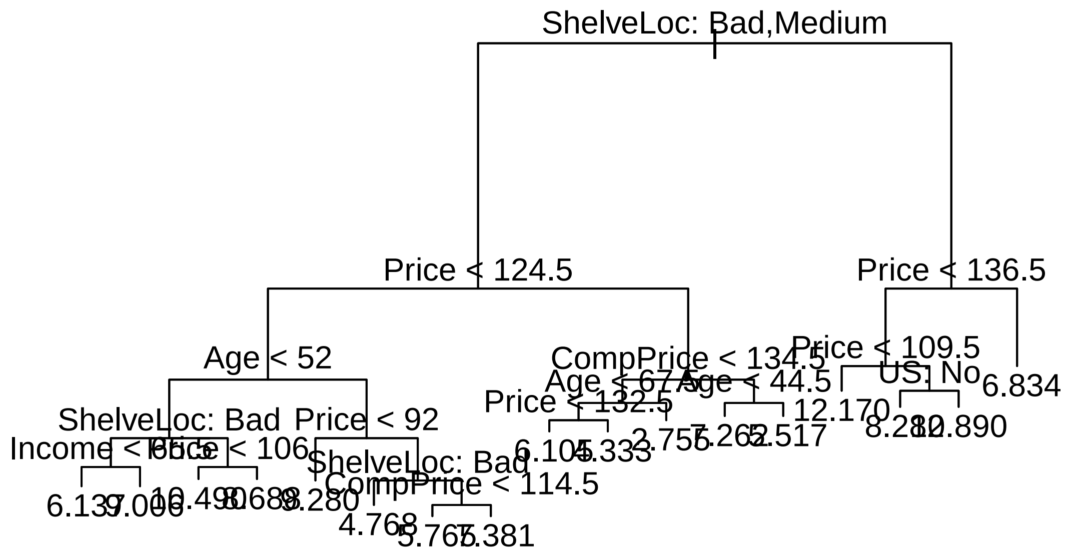
```
train = sample(1:n, n/2)
test = (1:n)[-train]
```

B. Fit a regression tree to the training set. Plot the tree, and interpret the results. What test MSE do you obtain?

```
rtree.carseats = tree(Sales ~., data=Carseats[train,])
summary(rtree.carseats)
```

```
##
## Regression tree:
## tree(formula = Sales ~ ., data = Carseats[train, ])
## Variables actually used in tree construction:
## [1] "ShelveLoc" "Price"     "Age"       "Income"    "CompPrice" "US"
## Number of terminal nodes:  17
## Residual mean deviance:  2.441 = 446.7 / 183
## Distribution of residuals:
##     Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
## -3.90700 -0.99560 -0.03944  0.00000  1.01200  3.77600
```

```
plot(rtree.carseats)
text(rtree.carseats, pretty=0)
```



```
y_hat = predict(rtree.carseats, newdata=Carseats[test,])
test.MSE = mean((y_hat - Carseats[test,]$Sales)^2)
print(test.MSE)
```

```
## [1] 4.276852
```

C. Use cross-validation in order to determine the optimal level of tree complexity. Does prunting the tree improve the test MSE?

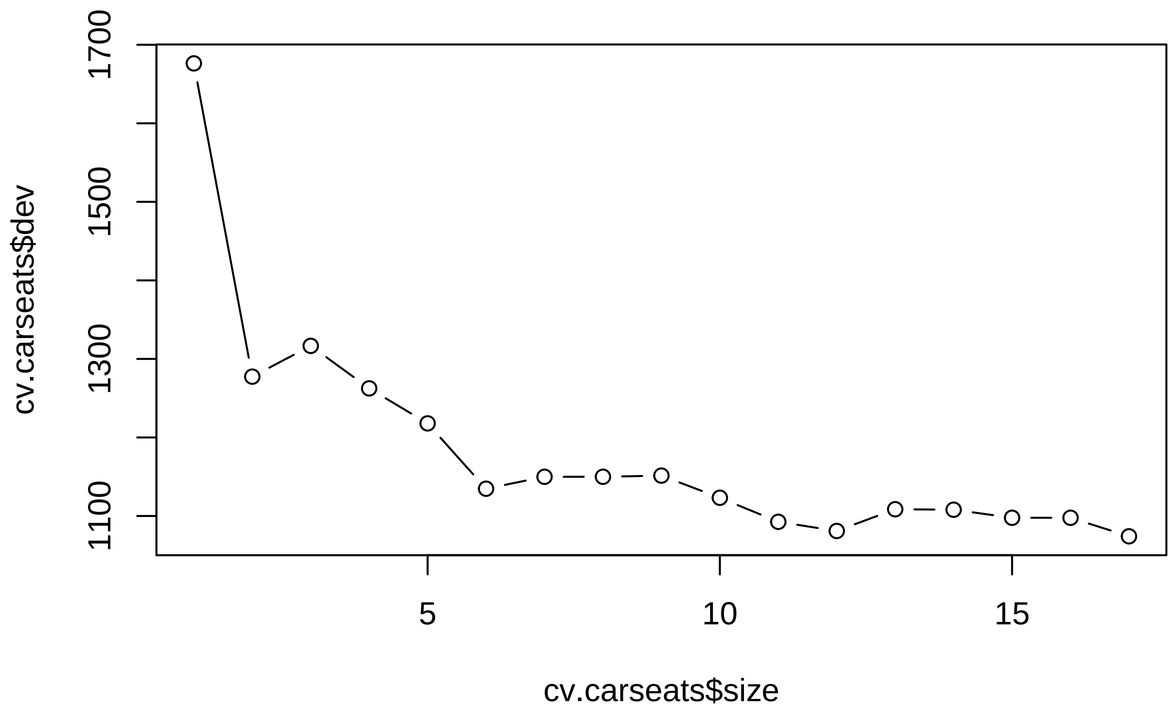
```
# Use cross-validation to determine optimal tree complexity
cv.carseats = cv.tree(rtree.carseats)
names(cv.carseats)
```

```
## [1] "size"   "dev"    "k"      "method"
```

```
print(cv.carseats)
```

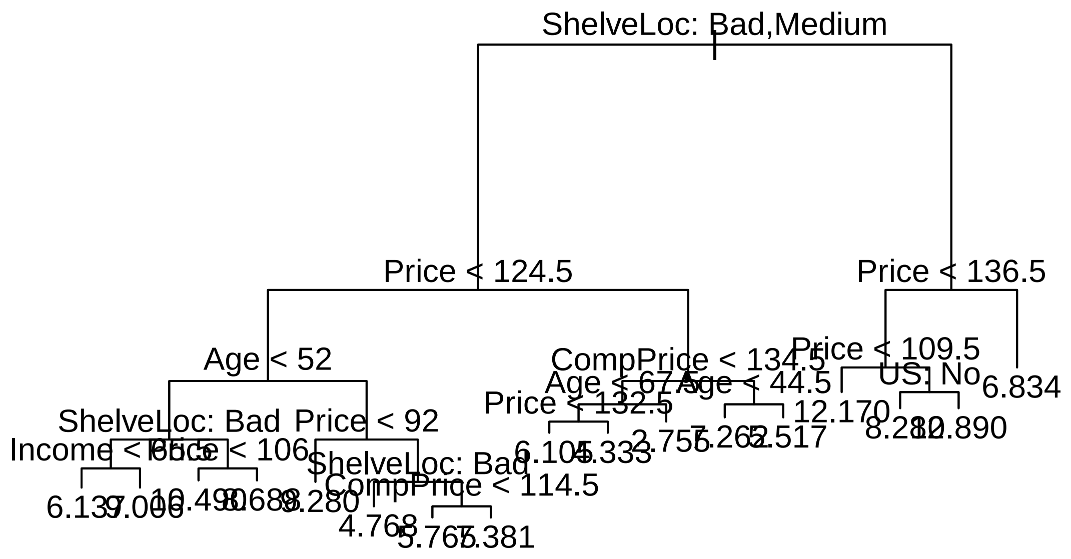
```
## $size
##  [1] 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
## 
## $dev
##  [1] 1074.122 1097.694 1097.694 1107.972 1108.590 1080.886 1092.554 1123.180
##  [9] 1151.382 1149.899 1149.899 1134.674 1217.885 1262.475 1316.610 1277.419
## [17] 1676.291
## 
## $k
##  [1]      -Inf  18.83282  18.87206  20.27236  22.04537  27.21361  29.14616
##  [8]  32.41127  39.33786  41.23946  41.60250  48.67832  71.43696  98.75811
## [15] 130.73089 153.56960 413.77594
## 
## $method
## [1] "deviance"
## 
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
plot(cv.carseats$size, cv.carseats$dev, type="b") #plot the tree size
```



```
# Pick the size of the tree you want to prune to:

#It looks like k = 17 is the smallest tree with an error close to the minimum

prune.carseats = prune.tree(rtree.carseats, best=17)
```

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



```
# Predict the MSE using this tree:
y_hat = predict(prune.carseats, newdata=Carseats[test,])
prune.MSE = mean((y_hat - Carseats[test,]$Sales)^2)
print(prune.MSE)
```

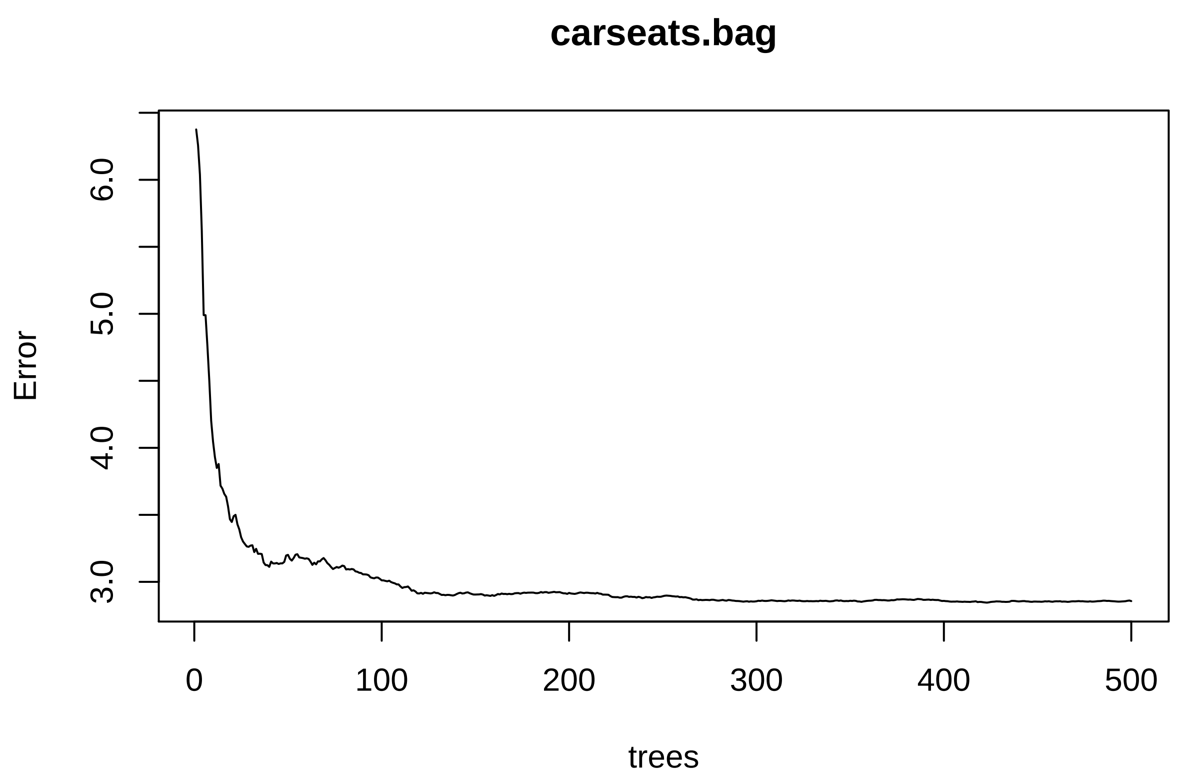
```
## [1] 4.276852
```

D. Use the bagging approach in order to analyze this data. What test MSE do you obtain? Use the `importance()` function to determine which variables are most important.

```
# Use bagging
carseats.bag = randomForest(Sales ~ ., data=Carseats, mtry=p, ntree=500, importance=TRUE, subset=train)
y_hat = predict(carseats.bag, newdata=Carseats[test,])
mse.bag = mean((Carseats[test,]$Sales - y_hat)^2)
print(mse.bag)
```

```
## [1] 2.597362
```

```
plot(carseats.bag)
```



```
ibag = importance(carseats.bag)
print(ibag[order(ibag[,1]),])
```

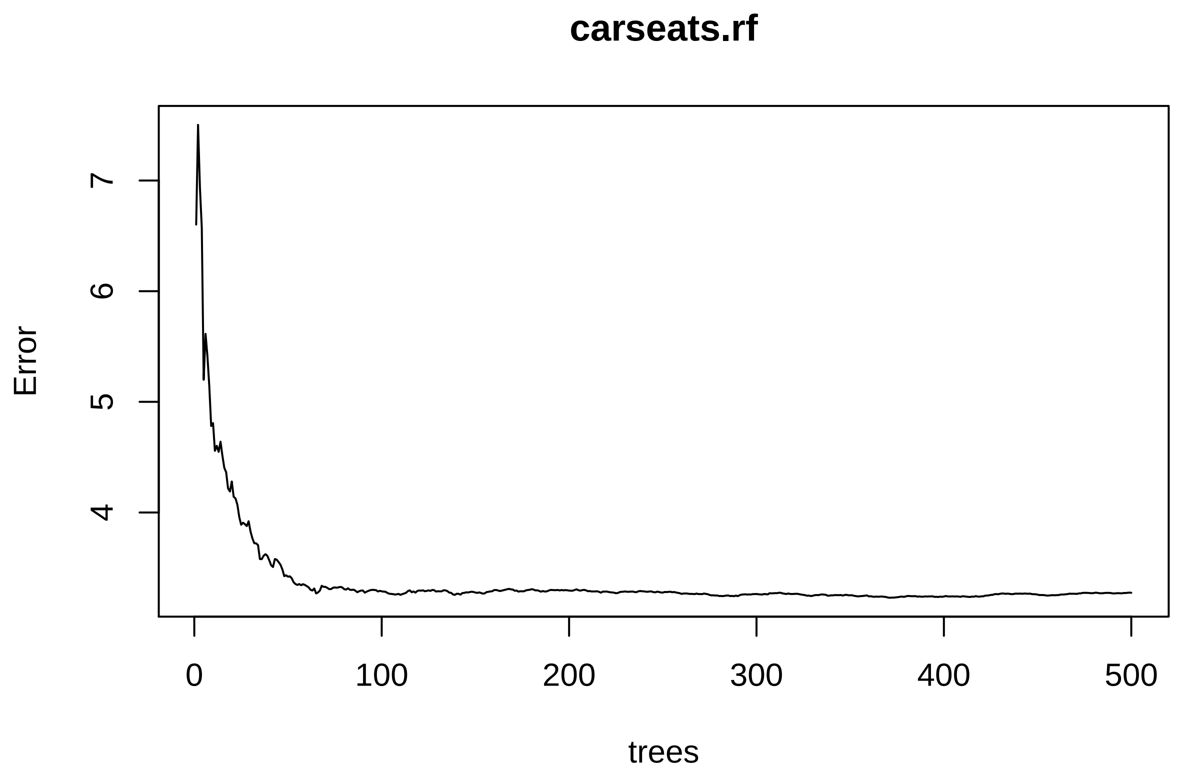
```
##                %IncMSE IncNodePurity
## Population  -2.660192      49.23199
## Education    2.396716      44.92580
## US           2.711138      13.52099
## Urban        3.303058      11.36380
## Income       3.331959      79.14007
## Advertising 16.217643     116.83583
## Age         17.283679     167.54657
## CompPrice   22.815491     171.93564
## Price       52.792602     495.11024
## ShelveLoc   54.280908     464.99885
```

E. Use random forests to analyze this data. What test MSE do you obtain? Use the `importance()` function to determine which variables are the most important. Describe the effect of m, the number of variables considered at each split, on the error rate obtained.

```
# Use random forests
carseats.rf = randomForest(Sales ~ ., data=Carseats, mtry=p/3, ntree=500, importance=TRUE, subset=train
y_hat = predict(carseats.rf, newdata=Carseats[test,])
mse.rf = mean((Carseats[test,]$Sales - y_hat)^2)
print(mse.rf)
```

```
## [1] 2.988168
```

```
plot(carseats.rf)
```



```
irf = importance(carseats.rf)
print(irf[order(irf[,1]),])
```

```
##                %IncMSE IncNodePurity
## Population  -0.9029990      97.56670
## Urban        0.1259916      19.06768
## Income       2.1277547     115.04105
## Education    3.1350209      76.48365
## US           6.6536420      30.78831
## CompPrice   13.1921517     166.43324
## Advertising 13.5428052     121.69371
## Age         14.3583472     205.55877
## ShelveLoc   34.0735184     330.22667
## Price       36.8150937     387.12126
```

Question Nine

This problem involves the `OJ` dataset which is part of the `ISLR` package.

A. Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

```
set.seed(0)

n = nrow(OJ)
p = ncol(OJ)-1 # remove the response Purchase
```

```
train = sample(1:n, 800)
test = (1:n)[-train]
```

B. Fit a tree to the training data, with `Purchase` as the response and the other variables as predictors. Use the `summary()` function to produce summary statistics about the tree, and describe the results obtained.

What is the training error rate? How many terminal nodes does the tree have?

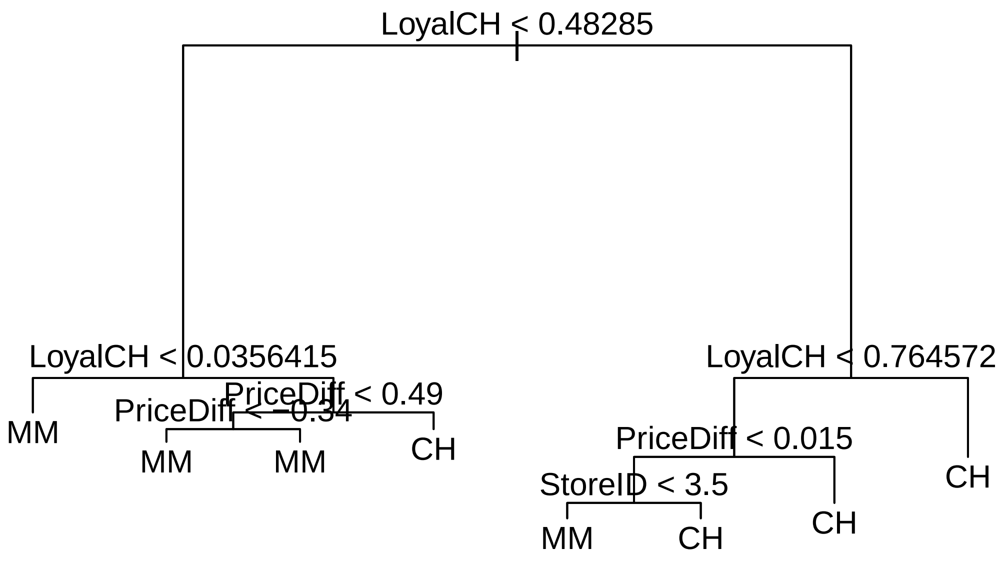
```
tree.OJ = tree(Purchase ~ ., data=OJ[train,])
summary(tree.OJ)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ[train, ])
## Variables actually used in tree construction:
## [1] "LoyalCH"   "PriceDiff" "StoreID"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7679 = 608.2 / 792
## Misclassification error rate: 0.1588 = 127 / 800
```

C. Type in the name of the tree object in order to get a detailed text output. Pick one of the terminal nodes, and interpret the information displayed.

D. Create a plot of the tree and interpret its results.

```
plot(tree.OJ)
text(tree.OJ, pretty=0)
```



E. Predict the response on the test data, and produce a confusion matrix comparing the test labels to the predicted test labels. What is the test error rate?

```
y_hat = predict(tree.OJ, newdata=OJ[test,], type="class") #gives classification labels
CT = table(y_hat, OJ[test,]$Purchase)
print(CT)
```

```
##
## y_hat  CH  MM
##    CH 134  24
##    MM  28  84
```

```
print('original tree: classification error rate on the test dataset')
```

```
## [1] "original tree: classification error rate on the test dataset"
```

```
print((CT[1,2] + CT[2,1]) / sum(CT))
```

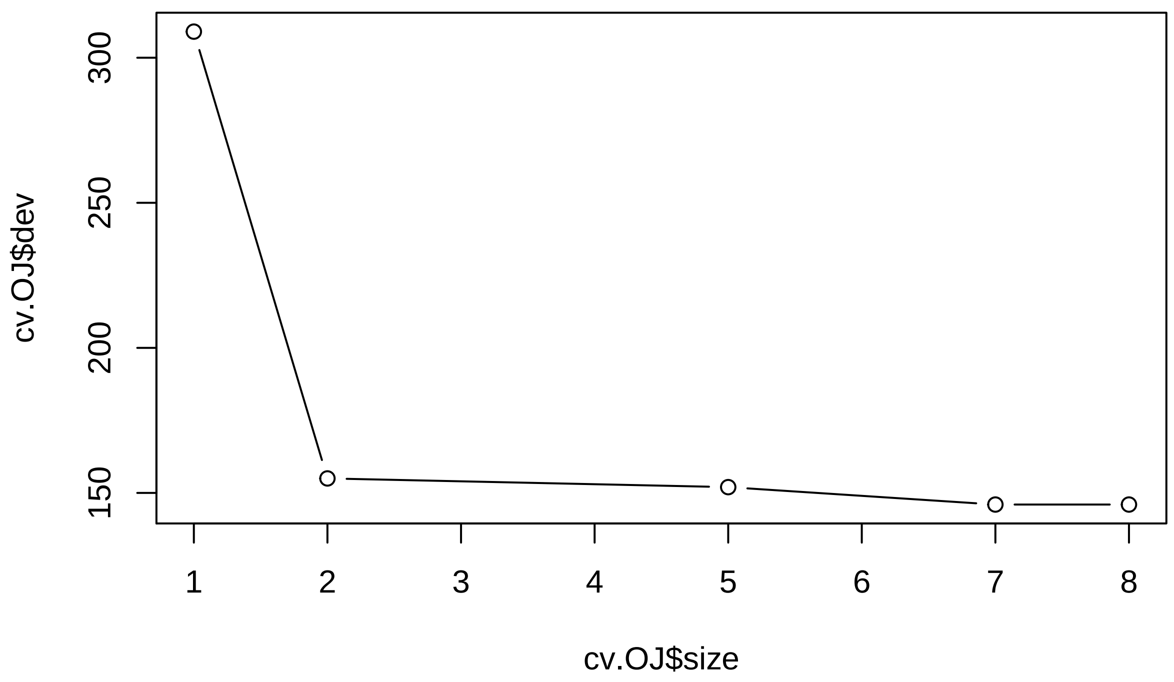
```
## [1] 0.1925926
```

F. Apply the `cv.tree()` function to the training set in order to determine the optimal tree size.

```
cv.OJ = cv.tree(tree.OJ, FUN=prune.misclass)
```

G. Produce a plot with tree size on the x-axis and cross-validation classification error on the y-axis.

```
plot(cv.OJ$size, cv.OJ$dev, type="b")
```

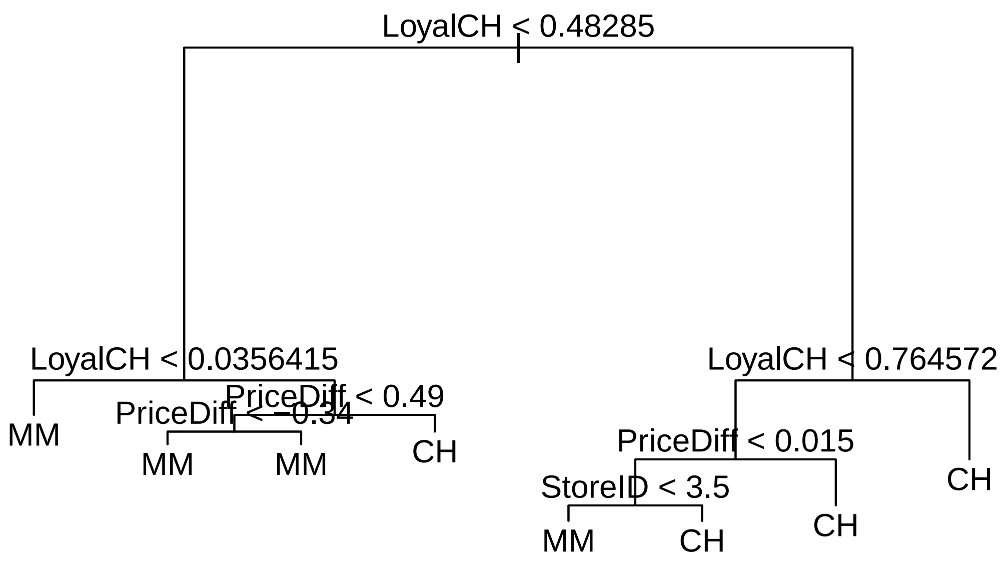


H. Which tree size corresponds to the lowest cross-validated classification error rate? **Answer:** 8

I. Produce a pruned tree corresponding to the optimal tree size obtained using cross-validation. If cross-validation does not lead to selection of a pruned tree, then create a pruned tree with five terminal nodes.

```
# Based on the above, pick the size of the tree you want to prune to:
prune.OJ = prune.misclass(tree.OJ, best=8)
```

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



J. Compare the training error rates between the pruned and unpruned trees. Which is higher?

```
# Compute the training error rates:
y_hat = predict(prune.OJ, newdata=OJ[train,], type="class")
CT = table(y_hat, OJ[train,]$Purchase)
print('pruned tree: classification error rate on the training dataset:')
```

```
## [1] "pruned tree: classification error rate on the training dataset:"
```

```
print((CT[1,2] + CT[2,1])/sum(CT))
```

```
## [1] 0.15875
```

Answer: the training error rate for the unpruned tree is higher.

K. Compare the test error rates between the pruned and unpruned trees. Which is higher?

```
# Compute testing error rates:
y_hat = predict(prune.OJ, newdata=OJ[test,], type='class')
CT = table(y_hat, OJ[test,]$Purchase)
print('pruned tree: classification error rate on the test dataset:')
```

```
## [1] "pruned tree: classification error rate on the test dataset:"
```

```
print((CT[1,2] + CT[2,1])/sum(CT))
```

```
## [1] 0.1925926
```

Answer: the testing error rates for the pruned and unpruned trees are the same.

Question Ten

We now use boosting to predict `Salary` in the `Hitters` dataset.

A. Remove the observations for whom the salary information is unknown, and then log-transform the salaries.

```
set.seed(0)

Hitters = na.omit(Hitters)
Hitters$Salary = log(Hitters$Salary)
```

B. Create a training set consisting of the first 200 observations, and a test set consisting of the remaining observations.

```r
n = nrow(Hitters)
p = ncol(Hitters) - 1 # one column is the response we are trying to model, "Salary"
```

```r
train = 1:200
test = 201:n
```

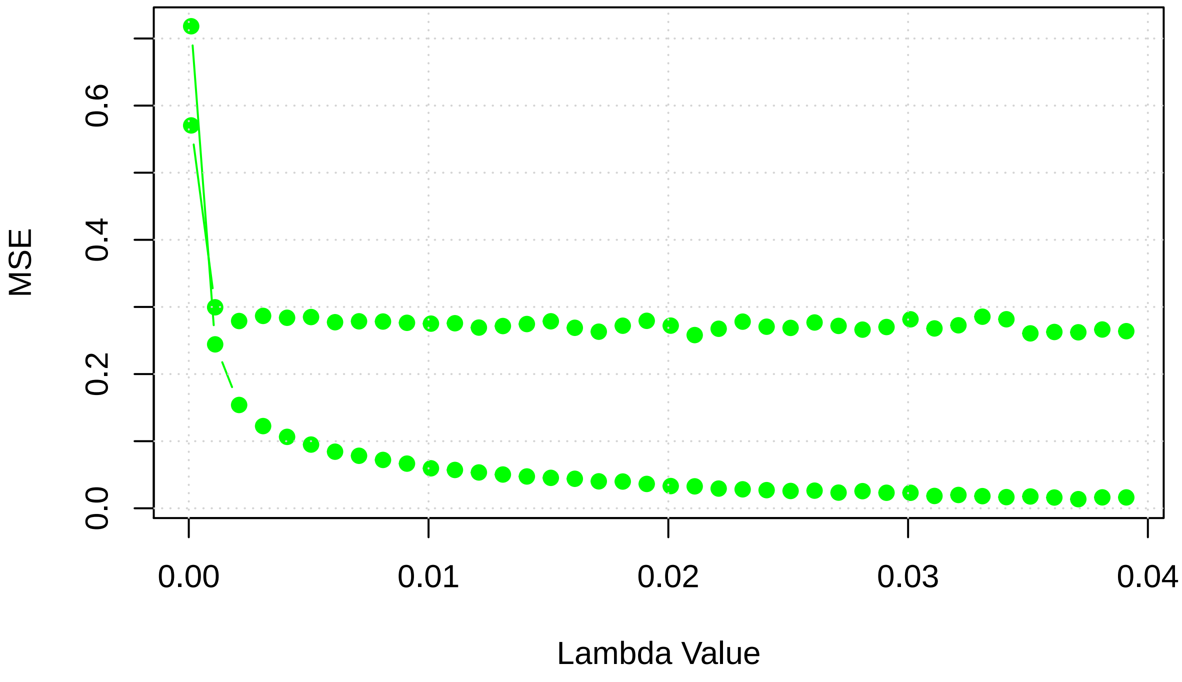
C. Perform boosting on the training set with 1,000 trees for a range of values of the shrinkage parameter λ. Produce a plot with different shrinkage values on the x-axis and the corresponding training set MSE on the y-axis.

```r
lambda_set = seq(1.e-4, 0.04, by=0.001)
```

```r
training_set_mse = rep(NA, length(lambda_set))
test_set_mse = rep(NA, length(lambda_set))
for(lmi in 1:length(lambda_set)){
  lm = lambda_set[lmi]

  boost.hitters = gbm(Salary ~ ., data=Hitters[train,], distribution="gaussian", n.trees=1000, interact

  y_hat = predict(boost.hitters, newdata=Hitters[train,], n.trees=1000)
  training_set_mse[lmi] = mean((y_hat - Hitters[train,]$Salary)^2)

  y_hat = predict(boost.hitters, newdata=Hitters[test,], n.trees=1000)
  test_set_mse[lmi] = mean((y_hat - Hitters[test,]$Salary)^2)
}
```

D. Produce a plot with different shrinkage values on the x-axis and the corresponding test MSE on the y-axis.

```r
plot(lambda_set, training_set_mse, type="b", pch=19, col='green', xlab="Lambda Value", ylab="MSE")
lines(lambda_set, test_set_mse, type="b", pch=19, col='green', xlab="Lambda Value", ylab="Test Set MSE"
grid()
```



E. Compare the test MSE of boosting to the test MSE that results from applying two regression approaches?

```
# Looks like the test MSE results are insensitive to the exact value of lambda as long as its small eno
lm = 0.01
boost.hitters = gbm(Salary ~ ., data=Hitters[train,], distribution="gaussian", n.trees=1000, interactio
y_hat = predict(boost.hitters, newdata=Hitters[test,], n.trees=1000)
print('regression boosting test MSE:')
```

```
## [1] "regression boosting test MSE:"
```

```
print(mean((y_hat - Hitters[test,]$Salary)^2))
```

```
## [1] 0.2756909
```

```
# Try linear regression:
m = lm(Salary ~ ., data=Hitters[train,])
y_hat = predict(m, newdata=Hitters[test,])
print('linear regression test MSE:')
```

```
## [1] "linear regression test MSE:"
```

```
print(mean((y_hat - Hitters[test,]$Salary)^2))
```

```
## [1] 0.4917959
```

```
# Try the lasso:
library(glmnet)
```

```
## Loading required package: Matrix

##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack

## Loaded glmnet 4.1-2
```

```
MM = model.matrix( Salary ~ ., data=Hitters[train,] )
cv.out = cv.glmnet( MM, Hitters[train,]$Salary, alpha=1 )
bestlam = cv.out$lambda.1se
print( "lasso CV best value of lambda (one standard error)" )
```

```
## [1] "lasso CV best value of lambda (one standard error)"
```

```
print( bestlam )
```

```
## [1] 0.1725201
```

```
lasso.mod = glmnet(MM, Hitters[train,]$Salary, alpha=1)
```

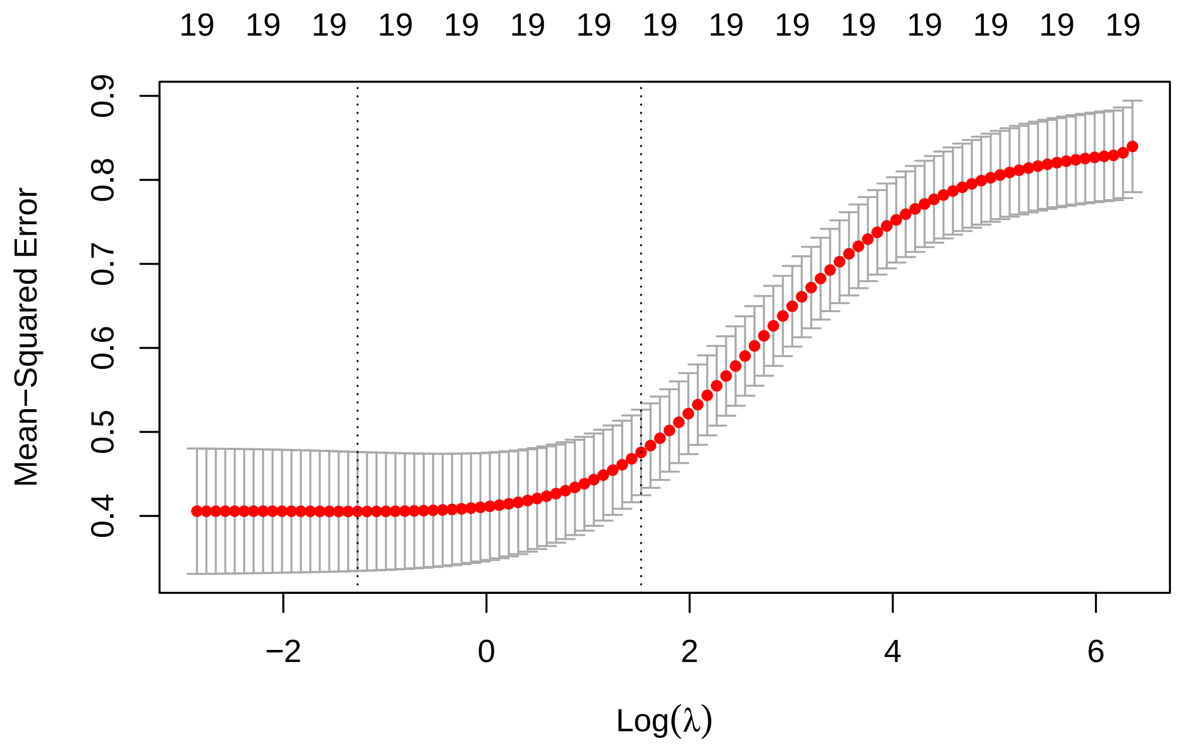
```
MM_test = model.matrix(Salary ~ ., data=Hitters[test,])
y_hat = predict(lasso.mod, s=bestlam, newx=MM_test)
print('lasso regression test MSE:')
```

```
## [1] "lasso regression test MSE:"
```

```
print(mean((y_hat = Hitters[test,]$Salary)^2))
```

```
## [1] 35.29202
```

```r
# Try Ridge Regression:
cv.out = cv.glmnet(MM, Hitters[train,]$Salary, alpha=0)
plot(cv.out)
```



```r
bestlam = cv.out$lambda.1se
print("ridge CV best value of lambda (one standard error)")
```

```
## [1] "ridge CV best value of lambda (one standard error)"
```

```r
print(bestlam)
```

```
## [1] 4.582284
```

```r
ridge.mod = glmnet(MM, Hitters[train,]$Salary, alpha=0)
y_hat = predict(ridge.mod, s=bestlam, newx=MM_test)
print("ridge regression test MSE:")
```

```
## [1] "ridge regression test MSE:"
```

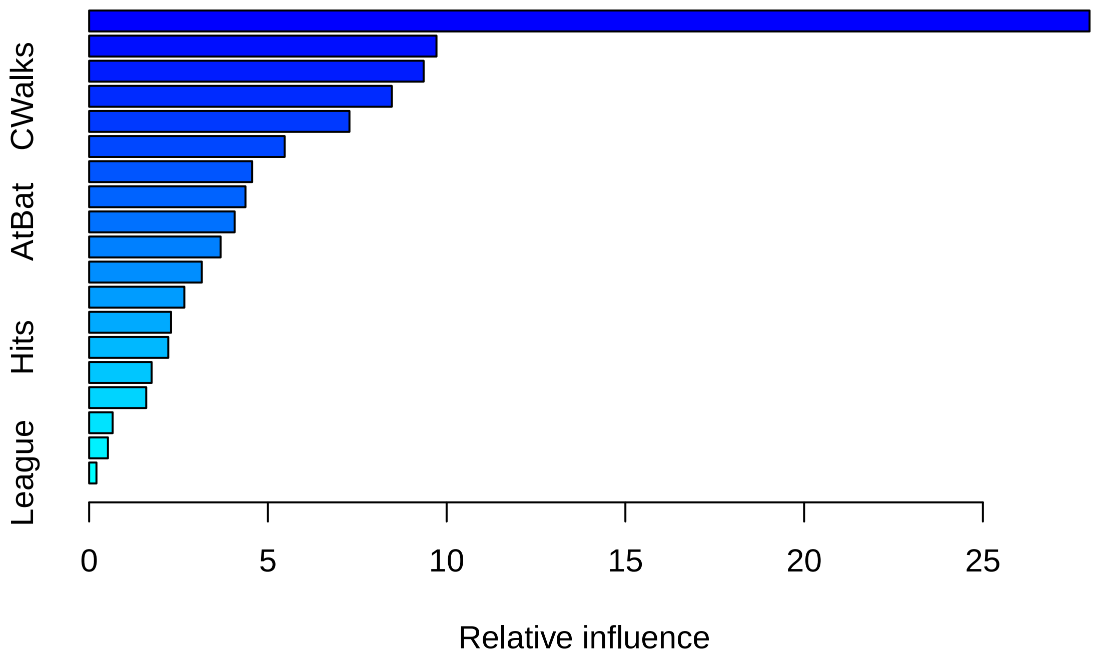
```r
print(mean((y_hat - Hitters[test,]$Salary)^2))
```

```
## [1] 0.454797
```

Answer: Lasso regression has the lowest test MSE.

F. Which variables appear to be the most important predictors in the boosted model?

```r
# What are the most important variables
summary(boost.hitters)
```



```
##                 var    rel.inf
## CAtBat       CAtBat 27.9843348
## CHits         CHits  9.7163571
## CRBI           CRBI  9.3564948
## CWalks       CWalks  8.4641693
## CRuns         CRuns  7.2820304
## Years         Years  5.4674280
## PutOuts     PutOuts  4.5604591
## Walks         Walks  4.3724216
## AtBat         AtBat  4.0692012
## CHmRun       CHmRun  3.6759612
## Assists     Assists  3.1511172
## RBI             RBI  2.6630040
## Errors       Errors  2.2909979
## Hits           Hits  2.2133744
## HmRun         HmRun  1.7480868
## Runs           Runs  1.5973072
## NewLeague NewLeague  0.6577638
## Division   Division  0.5252455
## League       League  0.2042459
```

G. Now apply bagging to the training set. What is the test set MSE for this approach?

```
# Try randomForests on the Hitters dataset (not asked for above)
rf.hitters = randomForest(Salary ~ ., data=Hitters, mtry=p/3, ntree=1000, importance=TRUE, subset=train
y_hat = predict(rf.hitters, newdata=Hitters[test,])
mse.rf = mean((Hitters[test,]$Salary - y_hat)^2)
print('randomForest test MSE:')
```

```
## [1] "randomForest test MSE:"
```

```
print(mse.rf)
```

```
## [1] 0.2141533
```

```r
# Try Bagging on the Hitters dataset:
bag.hitters = randomForest(Salary ~ ., data=Hitters, mtry=p, ntree=1000, importance=TRUE, subset=train)
y_hat = predict(bag.hitters, newdata=Hitters[test,])
mse.bag = mean((Hitters[test,]$Salary - y_hat)^2)
print('Bagging test MSE:')
```

```
## [1] "Bagging test MSE:"
```

```r
print(mse.bag)
```

```
## [1] 0.2293072
```

Random forest has the lowest test MSE.

Question Eleven

This question uses the `Caravan` dataset.

A. Create a training set consisting of the first 100 observations, and create a test set consisting of the remaining observations.

```r
set.seed(0)

Caravan = na.omit(Caravan)
```

```r
n = nrow(Caravan)
p = ncol(Caravan) - 1 # one column is the response we are trying to model, i.e., "Purchase"
```

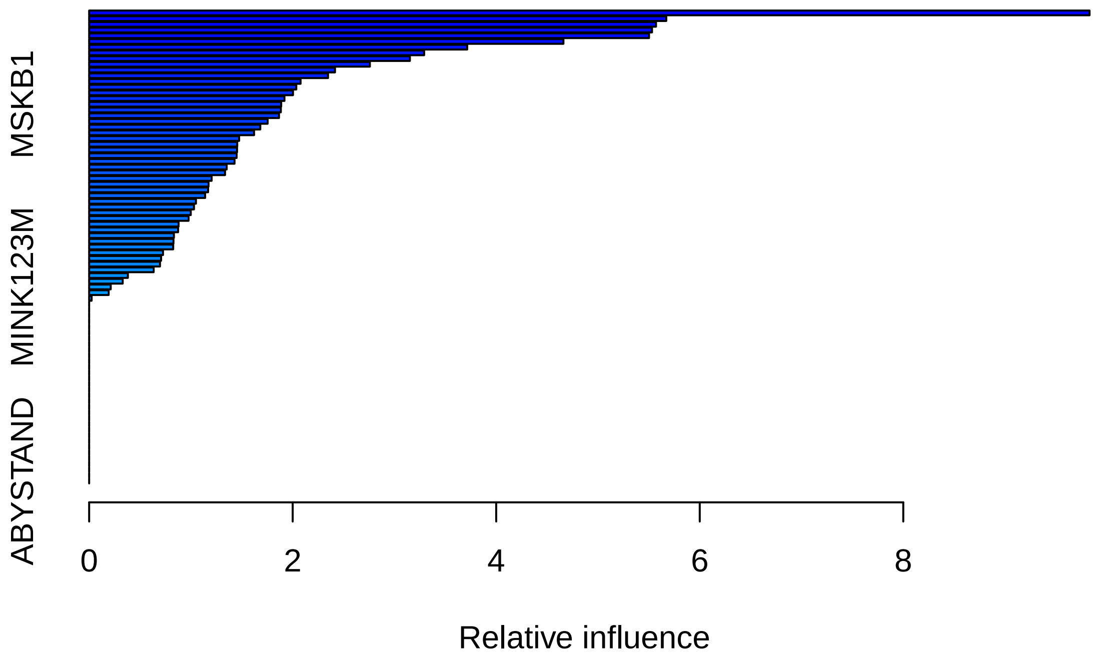
```r
train = 1:1000
test = 1001:n
```

```r
# Transform the response "Purchase" to be in [0,1] as required by gbm:
PurchaseBinary = rep(0,n)
PurchaseBinary[Caravan$Purchase == 'Yes'] = 1
Caravan$Purchase = PurchaseBinary
```

```r
# Some variables seen to be very noninformative (have zero variance as reported by gbm):
Caravan$PVRAAUT = NULL
Caravan$AVRAAUT = NULL
```

B. Fit a boosting model to the training set with `Purchase` as the response and the other variables as predictors. Use 1,000 trees, and a shrinkage value of 0.01. Which predictors appear the most important?

```r
# Train a gbm:
lm = 0.01
boost.caravan = gbm(Purchase ~ ., data=Caravan[train,], distribution="bernoulli", n.trees=1000, interac
summary(boost.caravan)
```



```
##                var    rel.inf
## PPERSAUT PPERSAUT 9.83047713
## MKOOPKLA MKOOPKLA 5.67101078
## MOPLHOOG MOPLHOOG 5.57015086
## PBRAND     PBRAND 5.53090647
## MGODGE     MGODGE 5.50146339
## MBERMIDD MBERMIDD 4.66053217
## MOSTYPE   MOSTYPE 3.71528842
## MINK3045 MINK3045 3.29167392
## MGODPR     MGODPR 3.15189659
## MAUT2       MAUT2 2.75860820
## MBERARBG MBERARBG 2.41585129
## ABRAND     ABRAND 2.34750399
## MSKC         MSKC 2.07742330
## MSKA         MSKA 2.03539655
## MAUT1       MAUT1 2.00290825
## MRELGE     MRELGE 1.91903070
## MSKB1       MSKB1 1.88713704
## PWAPART   PWAPART 1.88354710
## MFWEKIND MFWEKIND 1.86568208
## MGODOV     MGODOV 1.75362888
## MINK7512 MINK7512 1.68185238
## MFGEKIND MFGEKIND 1.62053883
## MBERHOOG MBERHOOG 1.47386158
## MBERARBO MBERARBO 1.45496977
## MINKM30   MINKM30 1.45332376
## MHKOOP     MHKOOP 1.44894919
## MGODRK     MGODRK 1.42859782
## MRELOV     MRELOV 1.35136073
## MINKGEM   MINKGEM 1.33514219
## MAUT0       MAUT0 1.20352717
## MZFONDS   MZFONDS 1.17292916
## MINK4575 MINK4575 1.16903944
## MOSHOOFD MOSHOOFD 1.13985697
## MHHUUR     MHHUUR 1.04951077
## MGEMLEEF MGEMLEEF 1.03005616
## APERSAUT APERSAUT 0.99864198
## MSKB2       MSKB2 0.97672510
## PBYSTAND PBYSTAND 0.87858256
## MOPLMIDD MOPLMIDD 0.87431940
## PMOTSCO   PMOTSCO 0.83289524
## MFALLEEN MFALLEEN 0.82850043
## MZPART     MZPART 0.82588303
## PLEVEN     PLEVEN 0.72568141
## MGEMOMV   MGEMOMV 0.70719044
## MSKD         MSKD 0.69605715
## MBERBOER MBERBOER 0.63402319
## MBERZELF MBERZELF 0.38036763
## MRELSA     MRELSA 0.32946406
## MINK123M MINK123M 0.21200829
## MOPLLAAG MOPLLAAG 0.19163868
## MAANTHUI MAANTHUI 0.02438834
## PWABEDR   PWABEDR 0.00000000
## PWALAND   PWALAND 0.00000000
## PBESAUT   PBESAUT 0.00000000
## PAANHANG PAANHANG 0.00000000
## PTRACTOR PTRACTOR 0.00000000
## PWERKT     PWERKT 0.00000000
## PBROM       PBROM 0.00000000
## PPERSONG PPERSONG 0.00000000
## PGEZONG   PGEZONG 0.00000000
## PWAOREG   PWAOREG 0.00000000
## PZEILPL   PZEILPL 0.00000000
## PPLEZIER PPLEZIER 0.00000000
## PFIETS     PFIETS 0.00000000
## PINBOED   PINBOED 0.00000000
## AWAPART   AWAPART 0.00000000
## AWABEDR   AWABEDR 0.00000000
## AWALAND   AWALAND 0.00000000
## ABESAUT   ABESAUT 0.00000000
## AMOTSCO   AMOTSCO 0.00000000
## AAANHANG AAANHANG 0.00000000
## ATRACTOR ATRACTOR 0.00000000
## AWERKT     AWERKT 0.00000000
## ABROM       ABROM 0.00000000
## ALEVEN     ALEVEN 0.00000000
## APERSONG APERSONG 0.00000000
## AGEZONG   AGEZONG 0.00000000
## AWAOREG   AWAOREG 0.00000000
## AZEILPL   AZEILPL 0.00000000
## APLEZIER APLEZIER 0.00000000
## AFIETS     AFIETS 0.00000000
## AINBOED   AINBOED 0.00000000
## ABYSTAND ABYSTAND 0.00000000
```

C. Use the boosting model to predict the response on the test data. Predict that a person will make a purchase if the estimated probability of purchase is greater than 2%. Form a confusion matrix. What fraction of the people predicted to make a purchase do in fact make one? How does this compare to results obtained from applying KNN or logistic regression to this dataset?

```
# Predict the testing error:
y_hat = predict(boost.caravan, newdata=Caravan[test,], n.trees=1000)
p_hat = exp(y_hat) / (1 + exp(y_hat)) # covert the logodd output into probabilities
```

```
will_buy = rep(0, length(test))
will_buy[p_hat > 0.2] = 1
```

```
# Create a confusion matrix
table(will_buy, Caravan[test,]$Purchase)
```

```
##
## will_buy    0    1
##        0 4357  253
##        1  176   36
```

```
# Train a logistic regression:
lr_model = glm(Purchase ~ ., data=Caravan[train,], family="binomial")
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

```
y_hat = predict(lr_model, newdata=Caravan[test,])
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

```
p_hat = exp(y_hat) / (1 + exp(y_hat)) # Convert the logodd output into probabilities
```

```
will_buy = rep(0, length(test))
will_buy[p_hat > 0.2] = 1
```

```
# Create a confusion matrix
table(will_buy, Caravan[test,]$Purchase)
```

```
##
## will_buy    0    1
##        0 4183  231
##        1  350   58
```

```
# Try bagging (not implemented/tested):
if(FALSE){
  bag.hitters = randomForest(Purchase ~ ., data=Caravan, mtry=p, ntree=1000, importance=TRUE, subset=tra
  y_hat = predict(bag.hitters, newdata=Caravan[test,])
  mse.bag = mean((Caravan[test,]$Purchase - y_hat)^2)
  print('randomForest test MSE:')
  print(mse.bag)
}
```