8.4 Exercises

- Exercise 7
- Exercise 8
- Exercise 9
- Exercise 10
- Exercise 11
- Exercise 12

```r
# load all libraries needed for these exercises
library(MASS)
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 3.4.4
```

```
## randomForest 4.6-14
```

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

```r
library(tree)
```

```
## Warning: package 'tree' was built under R version 3.4.4
```

```r
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 3.4.4
```

```r
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 3.4.4
```

```
## Loading required package: survival
```

```
## Loading required package: lattice
```

```
## Loading required package: splines
```

```
## Loading required package: parallel
```

```
## Loaded gbm 2.1.3
```

```
library(leaps)
```

```
## Warning: package 'leaps' was built under R version 3.4.4
```

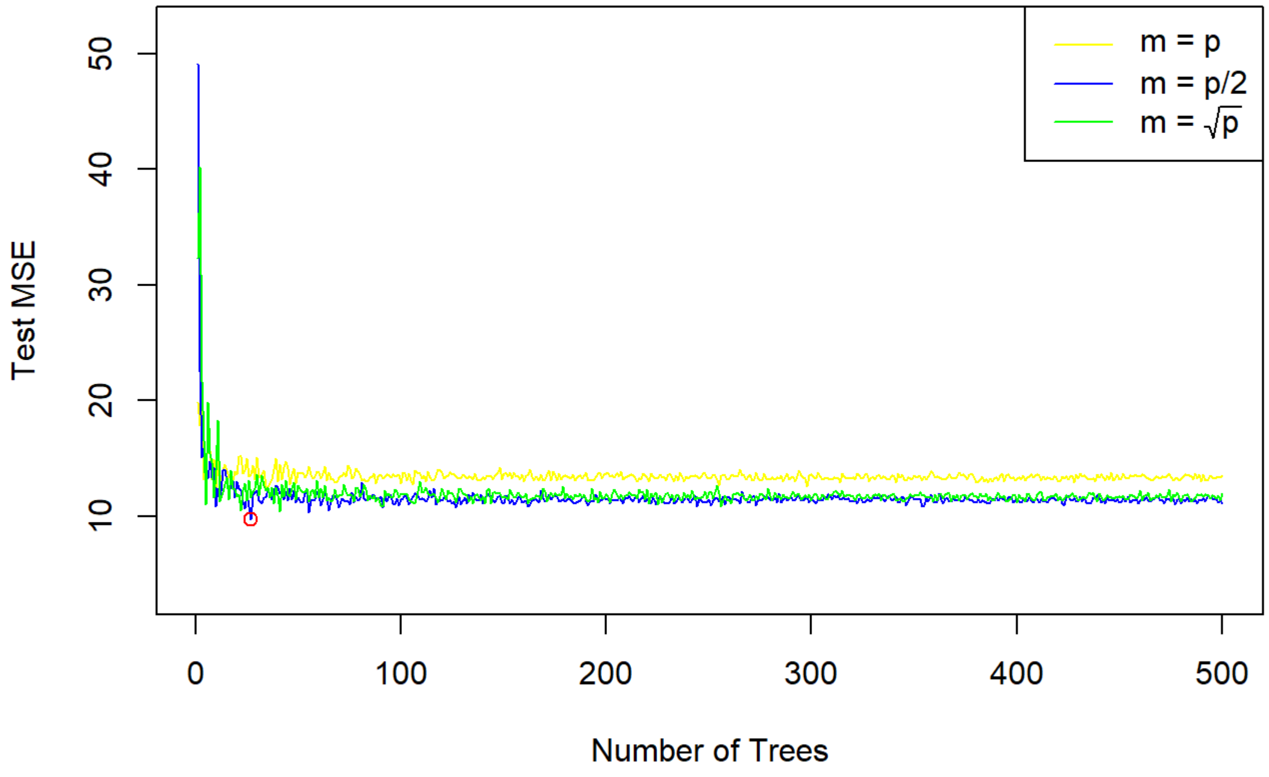
```
library(class)
library(datasets)
```

Exercise 7

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 data set for a more comprehensive range of values for **mtry** and **ntree**. You can model your plot after Figure 8.10. Describe the results obtained.

```
# Boston data set is in the MASS library
library(MASS)

# create a training set index with half the data
set.seed(1)
train = sample(1:nrow(Boston), size = nrow(Boston)/2, replace = FALSE)

# per Figure 8.10, we will test 1 to 500 trees, with m = p, m = p/2, and m = sqrt(p)
ntrees = 1:500
test.mse = data.frame(p = rep(NaN, length(ntrees)), p2 = rep(NaN, length(ntrees)), sqrt.
p = rep(NaN, length(ntrees)))

# one of the columns is the response and therefore not a valid predictor
p = ncol(Boston) - 1
columns = c('p', 'p2', 'sqrt.p')
m.values = c(p, p/2, sqrt(p))

# Load the randomForest() function from the randomForest library
library(randomForest)

getTestMSE = function(ntree, mtry){
  rf.model = randomForest(medv ~ ., data = Boston, subset = train, mtry = mtry, ntree =
 ntree)
  rf.pred = predict(rf.model, newdata = Boston[-train,])
  return(mean((rf.pred - Boston[-train, 'medv'])^2))
}

# loop through different numbers of trees and find the three test errors
# record the minimum test error
min.error = Inf
for (ntree in ntrees){
  for (i in 1:length(m.values)){
    this.col = columns[i]
    m = m.values[i]
    this.error = getTestMSE(ntree, m)
    test.mse[ntree, this.col] = this.error
    if (this.error < min.error){
      min.error = this.error
      min.ntree = ntree
      min.m = m
    }
  }
}
```

```
# Bound the y-axis limits of the plot to avoid cutting off the lines
y.min = min.m - 3
y.max = max(test.mse) + 3

# plot test errors in the same fasion as Figure 8.10
plot(ntrees, test.mse$p, xlab = 'Number of Trees', ylab = 'Test MSE', col = 'yellow', ty
pe = 'l', ylim = c(y.min, y.max))
lines(ntrees, test.mse$p2, col = 'blue')
lines(ntrees, test.mse$sqrt.p, col = 'green')
points(min.ntree, min.error, col = 'red')
legend('topright', col = c('yellow', 'blue', 'green'), legend = c('m = p', 'm = p/2', ex
pression(paste('m = ', sqrt(p)))), lty = 1)
```



In this plot, it's hard to see much of a difference between the different values of m*, but we can clearly see that additional trees after about* 50 *are not providing additional test accuracy.*

```
# print the minimum error and the associated ntree and mtry
min.error
```

```
## [1] 9.782963
```

```
min.ntree
```

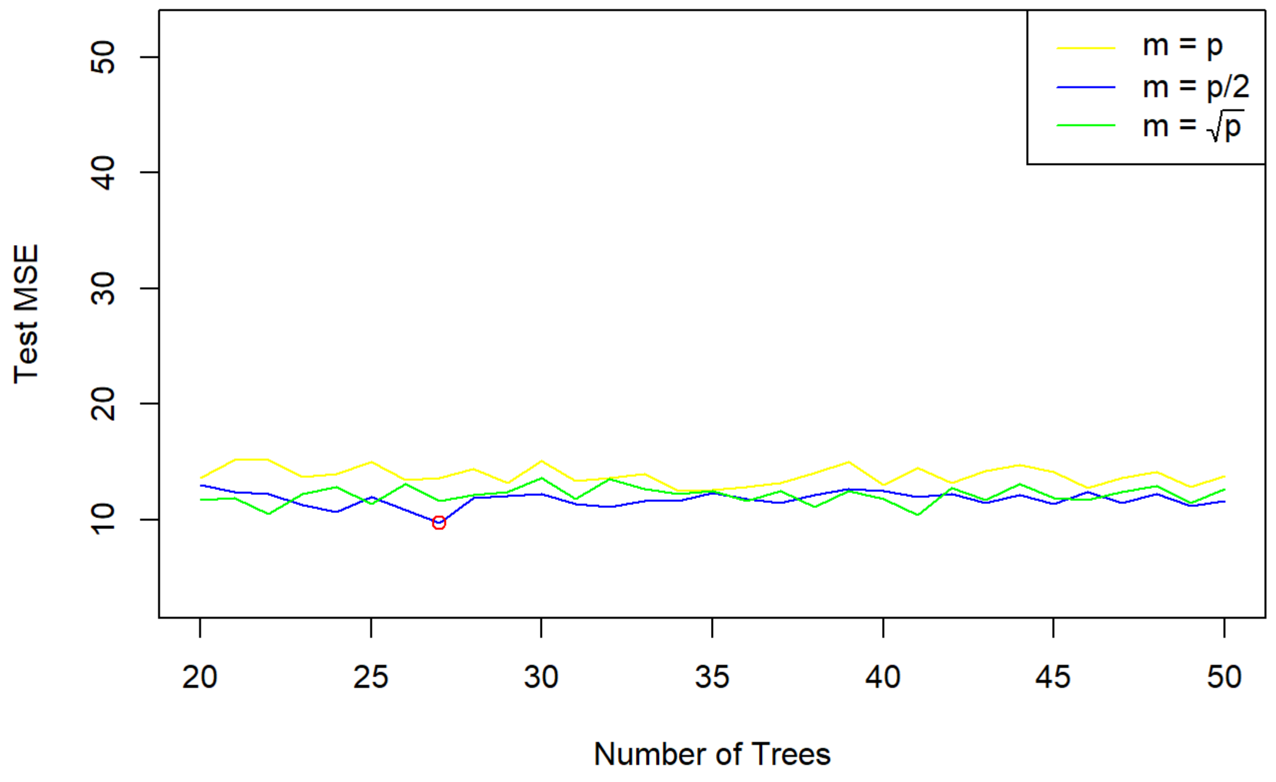
```
## [1] 27
```

```
min.m
```

```
## [1] 6.5
```

The lowest test MSE resulted from aggregating 27 *trees and considering* p/2 *predictors at each branch. Let's zoom in our plot around here to get a better look at what's going on.*

```
zoom.index = 20:50
plot(zoom.index, test.mse[zoom.index, 'p'], xlab = 'Number of Trees', ylab = 'Test MSE',
col = 'yellow', type = 'l', ylim = c(y.min, y.max))
lines(zoom.index, test.mse[zoom.index, 'p2'], col = 'blue')
lines(zoom.index, test.mse[zoom.index, 'sqrt.p'], col = 'green')
points(min.ntree, min.error, col = 'red')
legend('topright', col = c('yellow', 'blue', 'green'), legend = c('m = p', 'm = p/2', ex
pression(paste('m = ', sqrt(p)))), lty = 1)
```



So here we can see that in this range of numbers of trees, there really isn't much difference between the different values for m *or the number of trees being used. In fact, our selection of this particular pair of* m *and* ntrees *seems like it could just be a bit of noise in the variations in test error. In conclusion, for this data set, it seems like the only important thing is to use a sufficient number of trees in the bagging and/or random forests.* 25 *or more trees seems sufficient based on this data.*

Exercise 8

In the lab, a classification tree was applied to the **Carseats** data set 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 quantitative variable.

```
# the Carseats data set resides in the ISLR library
library(ISLR)
```

a. Split the data set into a training set and a test set.

```
# we'll go 50-50 on training and test data
train = sample(nrow(Carseats), nrow(Carseats)/2, replace = F)
```

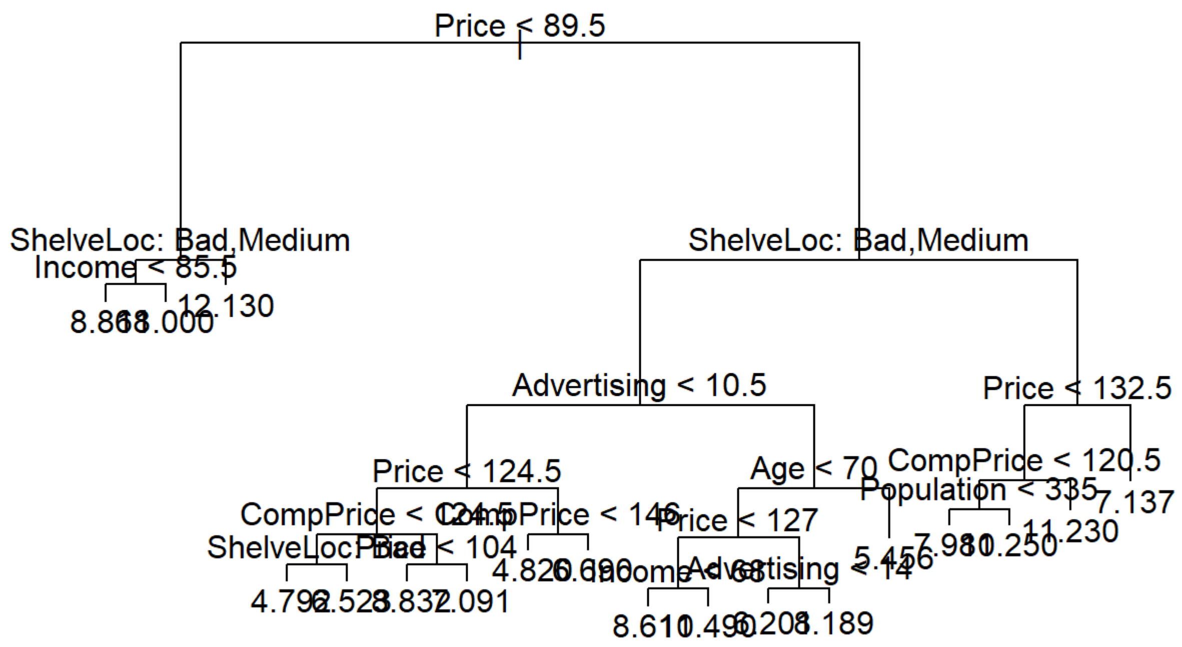
b. Fit a regression tree to the training set. Plot the tree and interpret the results. What test MSE do you obtain?

```
# the tree() function is in the tree library
library(tree)

# Make a reusable function for getting the test mse on this data set
getTestMSE = function(model){
  model.pred = predict(model, newdata = Carseats[-train,])
  return(mean((model.pred - Carseats[-train, 'Sales'])^2))
}

tree.model = tree(Sales ~ ., data = Carseats, subset = train)
tree.mse = getTestMSE(tree.model)
tree.mse
```

```
## [1] 4.730486
```

A simple regression tree yields a test MSE of 4.73.

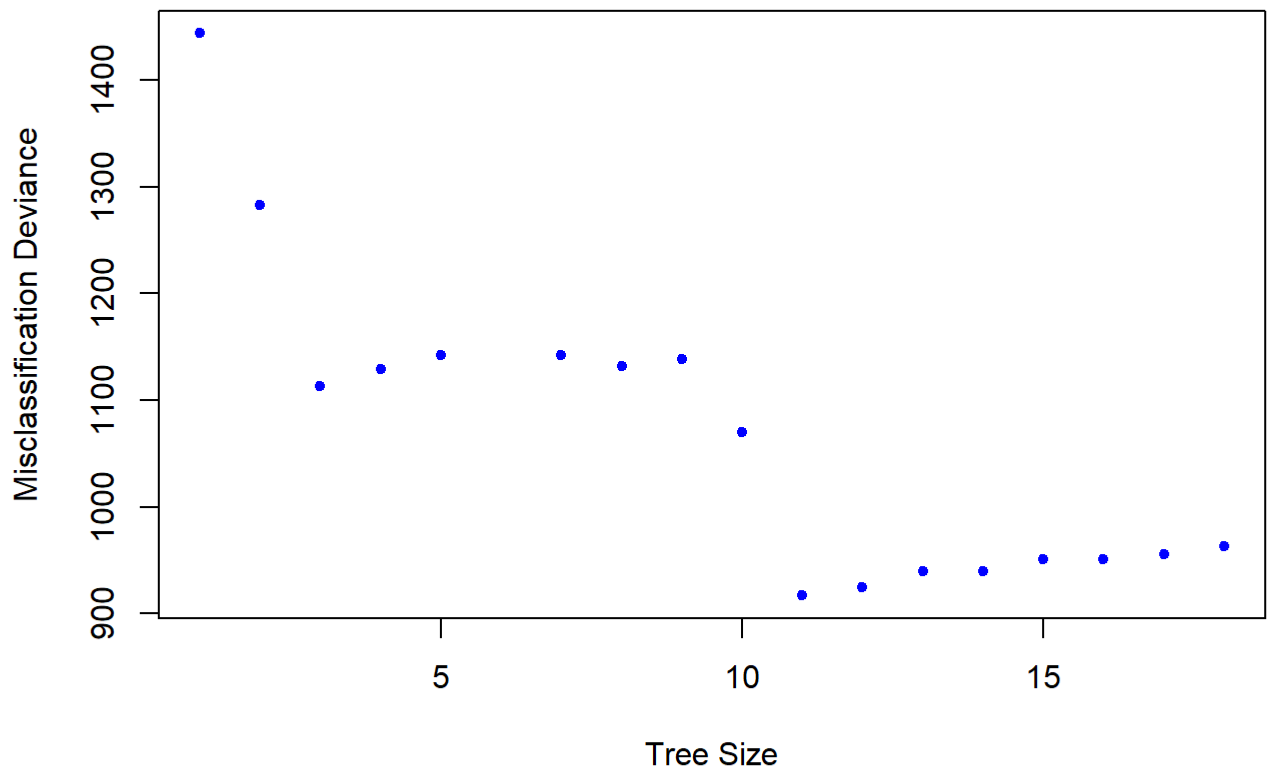
```
plot(tree.model)
text(tree.model, pretty = 0)
```



In this tree, the most important predictors are considered to be **Price** *and* **ShelveLoc**.

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

```
set.seed(4)
cv.carseats = cv.tree(tree.model)
plot(cv.carseats$size, cv.carseats$dev, pch = 20, col = 'blue', xlab = 'Tree Size', ylab
= 'Misclassification Deviance')
```



```
best.size = cv.carseats$size[which.min(cv.carseats$dev)]
best.size
```

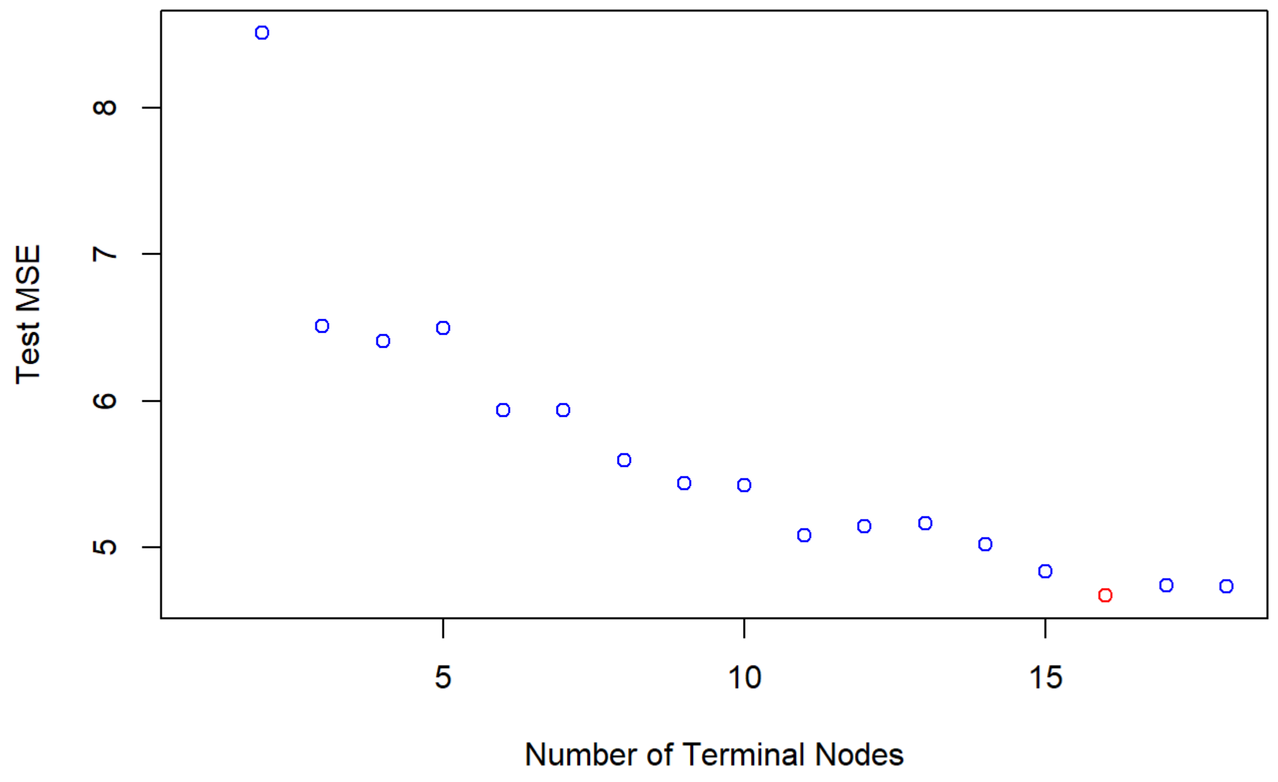
```
## [1] 11
```

```
prune.model = prune.tree(tree.model, best = best.size)
getTestMSE(prune.model)
```

```
## [1] 5.081844
```

The pruned model with 11 terminal nodes does not yield a lower test MSE. Note, I played with the seed value I set before taking the CV so that I would get a smaller tree. Seeds 1-3 actually did not result in a smaller tree.

```
# Let's plot the test MSE of different sizes to see if any of them yield a smaller test
 MSE
max.size = length(cv.carseats$size) + 1
test.mse = rep(NA, max.size)
for (this.size in 2:max.size){
  test.mse[this.size] = getTestMSE(prune.tree(tree.model, best = this.size))
}
best.size = which.min(test.mse)
plot(test.mse, xlab = "Number of Terminal Nodes", ylab = "Test MSE", col = "blue")
points(best.size, test.mse[best.size], col = "red")
```



```
best.size
```

```
## [1] 16
```

```
test.mse[best.size]
```

```
## [1] 4.670888
```

The pruned tree with only 16 terminal nodes has a slightly lower test MSE than the fully grown tree in this case.

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.

```
set.seed(1)

# We need to set mtry equal to the number of predictors in the model in order to perform
bagging instead of random forests
p = ncol(Carseats) - 1
bag.carseats = randomForest(Sales ~ ., data = Carseats, subset = train, mtry = p, import
ance = TRUE)
bag.mse = getTestMSE(bag.carseats)
bag.mse
```

```
## [1] 3.158778
```

Wow! It's a huge performance improvement. The test MSE obtained from baggging is much smaller than the test MSE from the pruned tree.

```
importance(bag.carseats)
```

```
##                %IncMSE IncNodePurity
## CompPrice   20.77899423    130.943589
## Income       7.25172186     80.581428
## Advertising 21.54915861    185.810892
## Population   0.41671466     55.003974
## Price       54.13927377    431.922698
## ShelveLoc   41.38863827    248.451451
## Age         12.63004181    111.973059
## Education    3.32618162     35.487637
## Urban        0.01734356      6.507844
## US           2.33948140      7.191645
```

In terms of training MSE, the bagging model lists **Price** *and* **ShelveLoc** *as the two most important predictors.* **Advertising** *and* **CompPrice** *are also considered important, but they don't impact the training MSE as much. Recall that the fully grown tree also considered* **Price** *and* **ShelveLoc** *to be the most important factors in predicting* **Sales**.

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

```
set.seed(1)

# First we'll use m = p/2
m = p/2
rf.carseats = randomForest(Sales ~ ., data = Carseats, subset = train, mtry = m, importa
nce = TRUE)
rf.mse = getTestMSE(rf.carseats)
rf.mse
```

```
## [1] 3.475288
```

Random forests with $m = p/2$ doesn't yield quite as low a test MSE as bagging, but it is still a significant improvement over the full tree and pruned tree test MSE.

```
importance(rf.carseats)
```

```
##                %IncMSE IncNodePurity
## CompPrice   14.1636754    119.055120
## Income       4.4456966     93.357926
## Advertising 19.3332935    169.700848
## Population  -0.9103591     77.596018
## Price       43.1857963    387.705189
## ShelveLoc   34.1445934    232.120164
## Age         10.4461913    118.779148
## Education    0.9741664     51.301962
## Urban        0.7789178      8.517634
## US           4.8030455     20.208178
```

Nothing too different about the order of predictor importance. It may be worth noting that the predictors which were considered much more important in the bagging model are a bit closer in importance to the other predictors.

```
set.seed(1)

# Now we'll try m = sqrt(p)
m = sqrt(p)
rf.carseats = randomForest(Sales ~ ., data = Carseats, subset = train, mtry = m, importa
nce = TRUE)
rf.mse = getTestMSE(rf.carseats)
rf.mse
```

```
## [1] 4.107447
```

Random forests with $m = \sqrt{p}$ yields a test MSE similar to the pruned tree test MSE - not great.

```
importance(rf.carseats)
```

```
##                %IncMSE IncNodePurity
## CompPrice    9.8053219     111.37676
## Income       1.5185003     102.50976
## Advertising 16.4317019     156.84286
## Population   0.2753494     101.21260
## Price       36.0495211     336.83466
## ShelveLoc   29.1520438     196.30048
## Age          9.1381909     129.59990
## Education    3.6112924      68.37053
## Urban        0.1102256      11.87917
## US           6.6298204      36.73006
```

The importance table yielded with $m = \sqrt{p}$ is similar to the one with $m = p/2$ with respect to predictor importance on training MSE. However, predictor importance has become even more "flat".

Overall, random forests yielded the best performance on this data set. All of the models observed considered **Price** *and* **ShelveLoc** *to be the best indicators of* **Sales**.

Exercise 9

This problem involves the **OJ** data set, 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(1)
train = sample(nrow(OJ), 800)
```

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.

```
tree.model = tree(Purchase ~ ., data = OJ, subset = train)
summary(tree.model)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ, subset = train)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "SpecialCH"     "ListPriceDiff"
## Number of terminal nodes:  8
## Residual mean deviance:  0.7305 = 578.6 / 792
## Misclassification error rate: 0.165 = 132 / 800
```

The resulting tree has 8 terminal nodes and only uses 4 predictors from the data set, which has 17 available predictors to use. The training misclassification error rate was 0.165.

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.

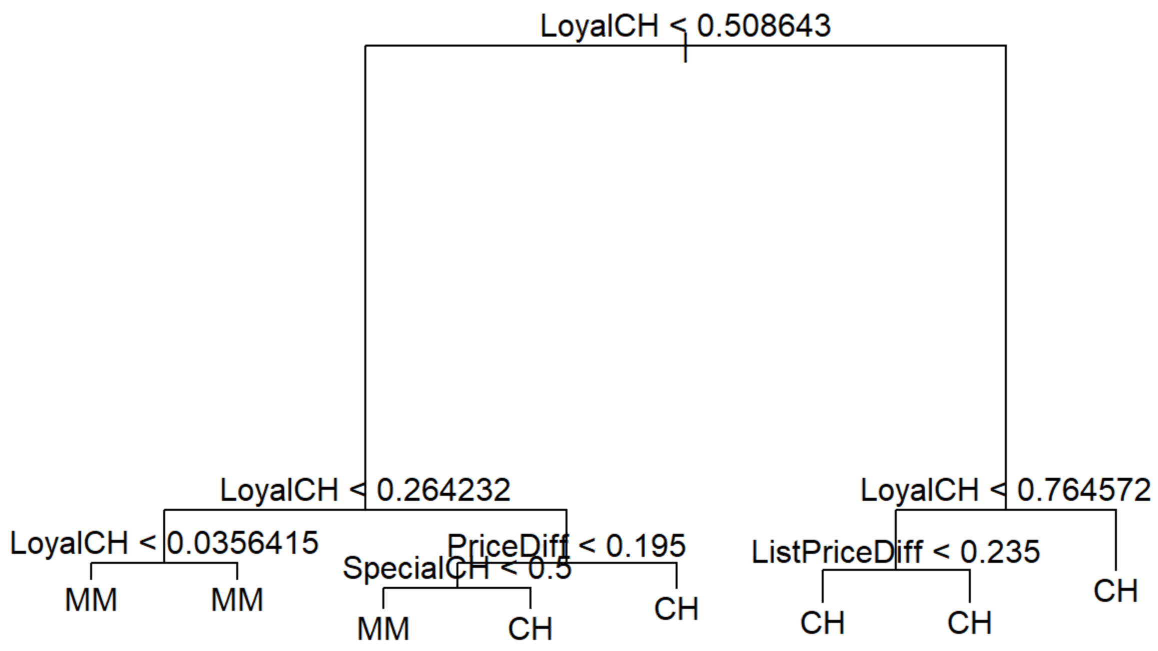
```
tree.model
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1064.00 CH ( 0.61750 0.38250 )
##    2) LoyalCH < 0.508643 350  409.30 MM ( 0.27143 0.72857 )
##      4) LoyalCH < 0.264232 166  122.10 MM ( 0.12048 0.87952 )
##        8) LoyalCH < 0.0356415 57   10.07 MM ( 0.01754 0.98246 ) *
##        9) LoyalCH > 0.0356415 109  100.90 MM ( 0.17431 0.82569 ) *
##      5) LoyalCH > 0.264232 184  248.80 MM ( 0.40761 0.59239 )
##       10) PriceDiff < 0.195 83   91.66 MM ( 0.24096 0.75904 )
##         20) SpecialCH < 0.5 70   60.89 MM ( 0.15714 0.84286 ) *
##         21) SpecialCH > 0.5 13   16.05 CH ( 0.69231 0.30769 ) *
##       11) PriceDiff > 0.195 101  139.20 CH ( 0.54455 0.45545 ) *
##    3) LoyalCH > 0.508643 450  318.10 CH ( 0.88667 0.11333 )
##      6) LoyalCH < 0.764572 172  188.90 CH ( 0.76163 0.23837 )
##       12) ListPriceDiff < 0.235 70   95.61 CH ( 0.57143 0.42857 ) *
##       13) ListPriceDiff > 0.235 102   69.76 CH ( 0.89216 0.10784 ) *
##      7) LoyalCH > 0.764572 278   86.14 CH ( 0.96403 0.03597 ) *
```

Let’s look at terminal node (8). The predictor region defined by this terminal node encompasses 57 of the original 800 training observations, which is 7.125%. This region is defined by **LoyalCH** < 0.0356415 *(the conditions for branches leading to this terminal node all happened to be checking for* **LoyalCH** *less than a certain value as well). The model predicts any observations meeting that criteria to have a response value of* **MM** *(indicating a purchase of Minute Maid orange juice). 98.246% of the training observations in this region had a response value of* **MM**.

d. Create a plot of the tree and interpret the results.

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



Again, we see that we have 8 terminal nodes, 3 of which correspond to a prediction of **MM**. **LoyalCH** *is considered to be the most important predictor in determining* **Purchase**.

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?

```
tree.prob = predict(tree.model, newdata = OJ[-train,])
tree.predict = rep('MM', nrow(tree.prob))
tree.predict[tree.prob[, 'CH'] >= .5] = 'CH'
table(tree.predict, OJ[-train, 'Purchase'])
```

```
##
## tree.predict  CH  MM
##           CH 147  49
##           MM  12  62
```

```
tree.testError = (12 + 62) / length(tree.predict)
tree.testError
```

```
## [1] 0.2740741
```

The full grown tree yields a test error rate of 27.4%.

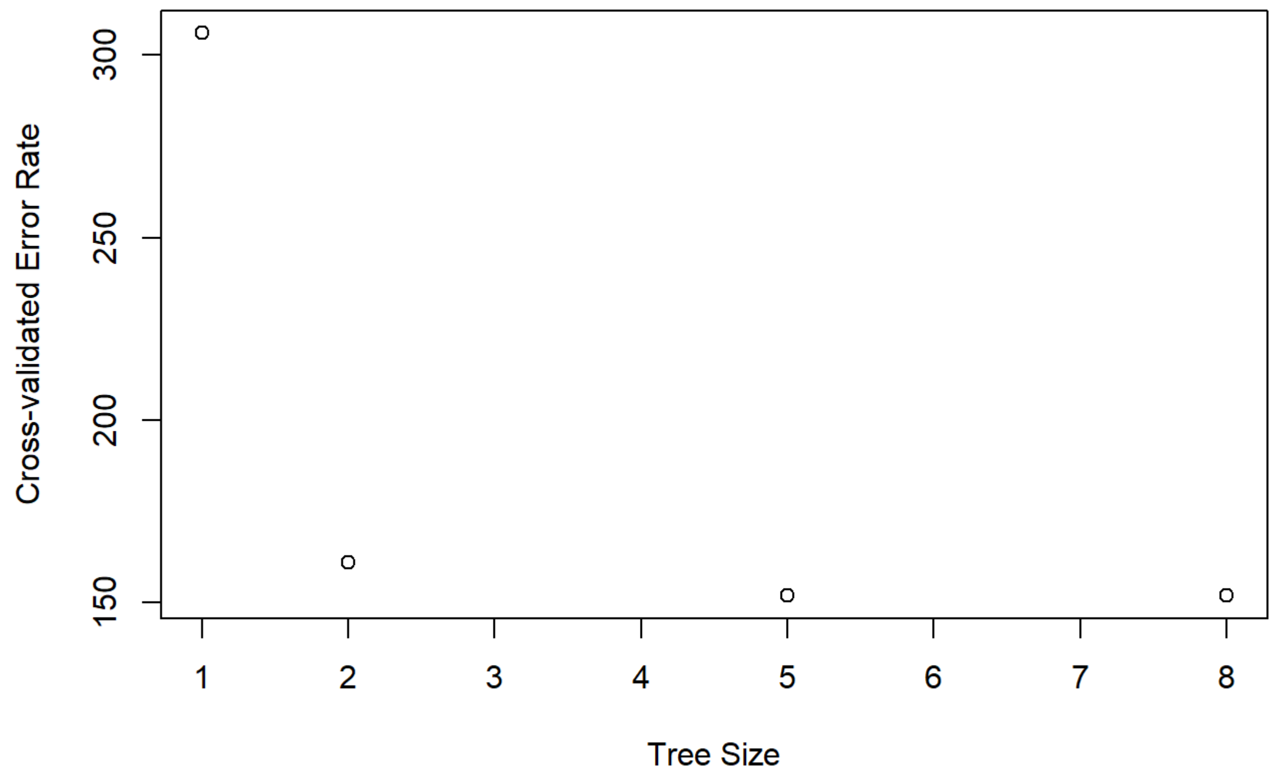
f. Apply the **cv.tree()** function to the training set in order to determine the optimal tree size.

```
set.seed(1)
cv.model = cv.tree(tree.model, FUN = prune.misclass)
cv.model
```

```
## $size
## [1] 8 5 2 1
##
## $dev
## [1] 152 152 161 306
##
## $k
## [1]       -Inf   0.000000   4.666667 160.000000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

g. Produce a plot with tree size on the *x*-axis and cross-validated classification error rate on the *y*-axis.

```
plot(cv.model$size, cv.model$dev, xlab = "Tree Size", ylab = "Cross-validated Error Rat
e")
```

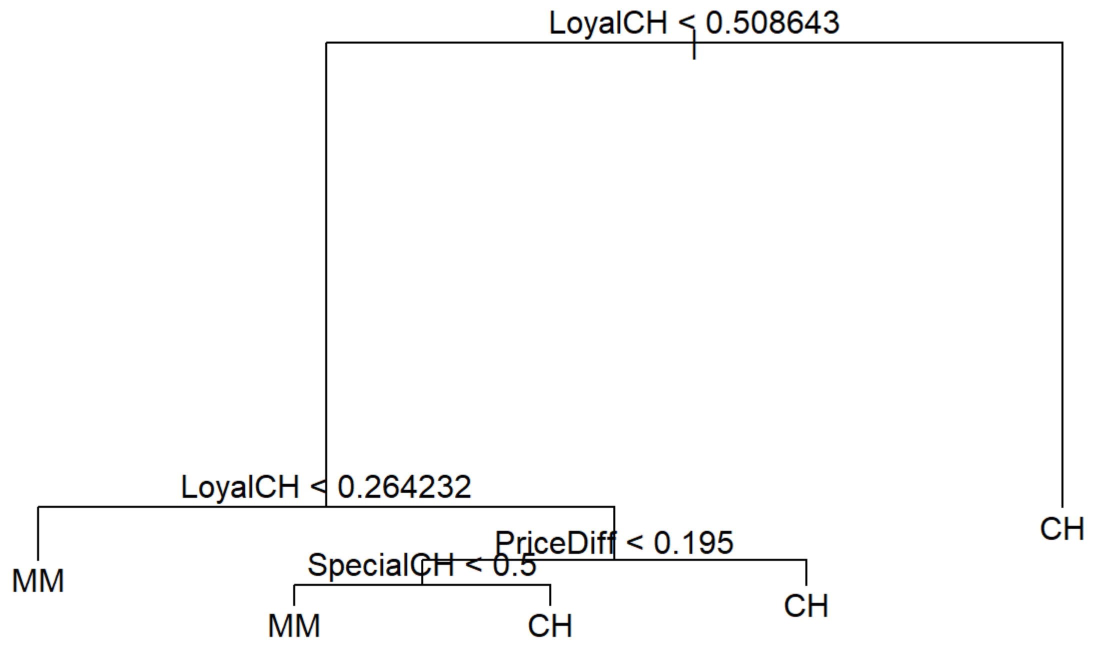


h. Which tree size corresponds to the lowest cross-validated classification error rate?

Both 5 and 8 terminal nodes yield the lowest CV error rate. We'll look at pruning the tree to 5 terminal nodes, since we already examined the full tree with 8 terminal nodes.

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.

```
prune.model = prune.misclass(tree.model, best = 5)
plot(prune.model)
text(prune.model, pretty = 0)
```



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

```
summary(prune.model)
```

```
##
## Classification tree:
## snip.tree(tree = tree.model, nodes = 3:4)
## Variables actually used in tree construction:
## [1] "LoyalCH"   "PriceDiff" "SpecialCH"
## Number of terminal nodes:  5
## Residual mean deviance:  0.8256 = 656.4 / 795
## Misclassification error rate: 0.165 = 132 / 800
```

The training error rate for the pruned tree is 16.5%, which is exactly the same as the training error rate observed with the full tree.

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

```
prune.prob = predict(prune.model, newdata = OJ[-train,])
prune.predict = rep('MM', nrow(prune.prob))
prune.predict[prune.prob[, 'CH'] >= .5] = 'CH'
prune.testError = mean(prune.predict != OJ[-train, "Purchase"])

tree.testError
```

```
## [1] 0.2740741
```

```
prune.testError
```

```
## [1] 0.2259259
```

The pruned tree has a test error rate of 22.6%, *which is a bit lower than the full tree test error rate.*

Exercise 10

We now use boosting to predict **Salary** in the **Hitters** data set.

a. Remove the observations for which the salary information is unknown and then log-transform the salaries.

```
Hitters = Hitters[complete.cases(Hitters$Salary),]
Hitters$Salary.log = log(Hitters$Salary)
```

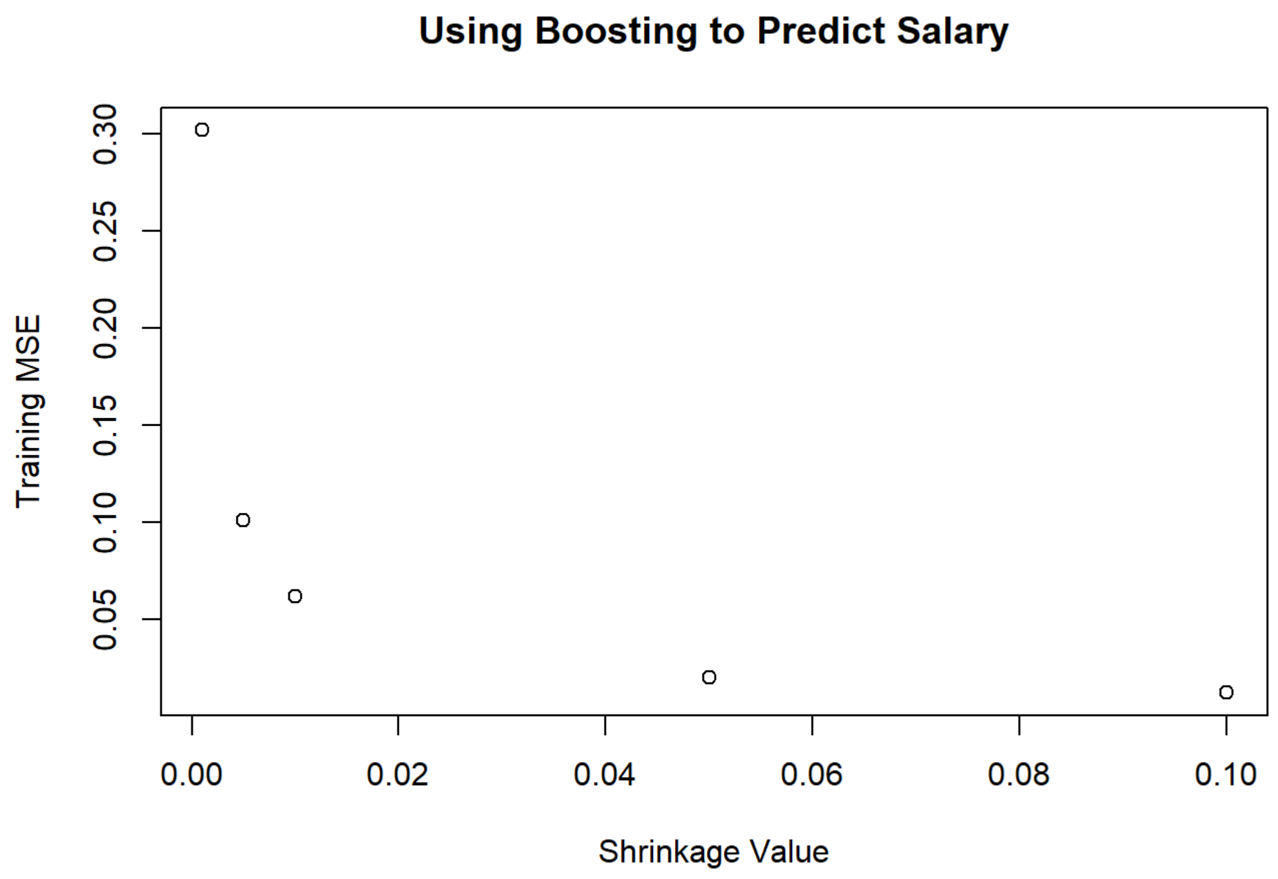
b. Create a training set consisting of the first 200 observations and a test set consisting of the remaining observations.

```
train = 1:200
```

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.

```
# need to gbm library to perform boosting
library(gbm)

# According to the book, typical values are 0.01 or 0.001, so we'll use that as a starti
ng point
shrinkage.values = c(0.01, 0.001, 0.1, 0.005, 0.05)
test.mse = rep(NA, length(shrinkage.values))
training.mse = rep(NA, length(shrinkage.values))

for (index in 1:length(shrinkage.values)){
  boost.model = gbm(Salary.log ~ . - Salary, data = Hitters[train,], distribution = "gau
ssian", n.trees = 1000, shrinkage = shrinkage.values[index])
  boost.predict = predict(boost.model, newdata = Hitters[-train,], n.trees = 1000)
  test.mse[index] = mean((Hitters[-train, "Salary.log"] - boost.predict)^2)
  training.mse[index] = mean(boost.model$train.error^2)
}
```

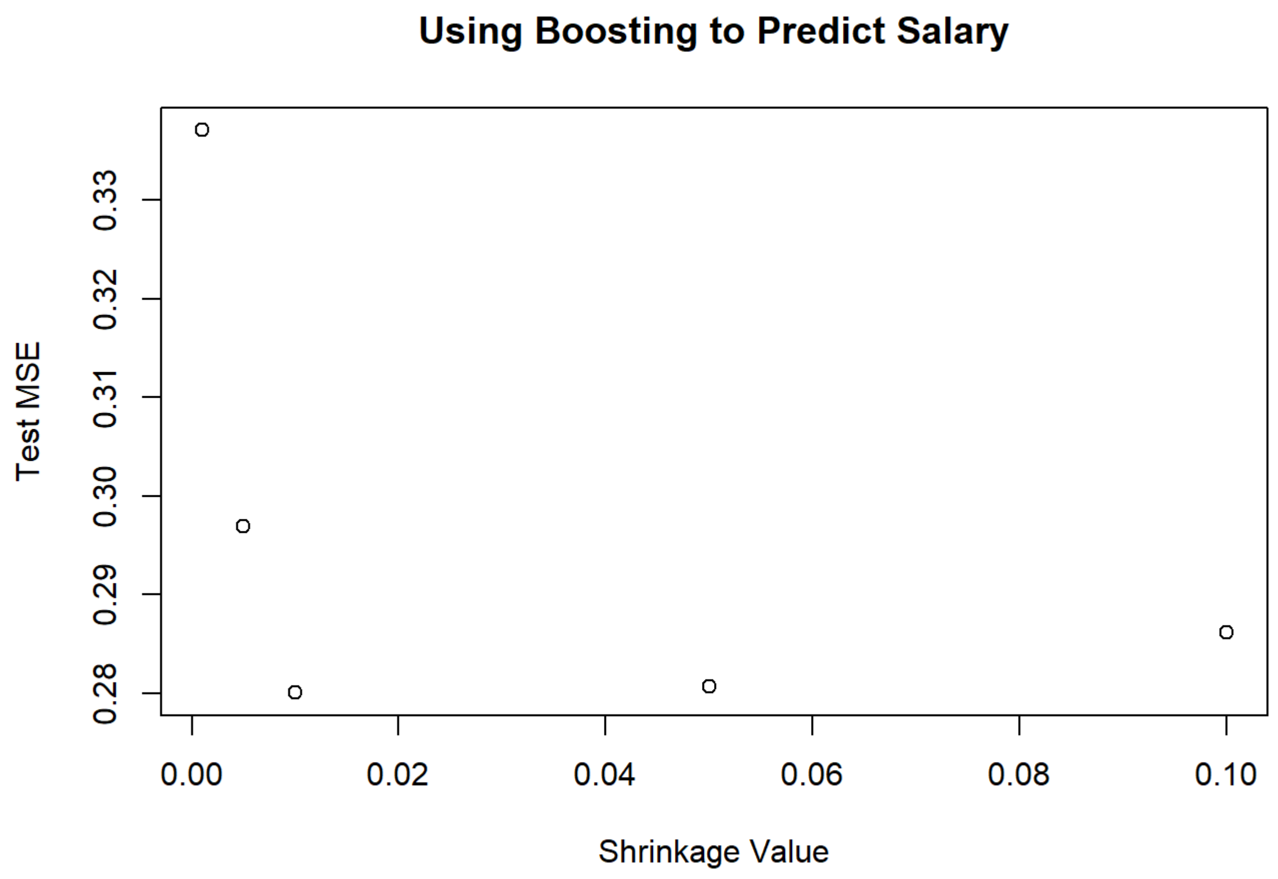
```
plot(shrinkage.values, training.mse, xlab = 'Shrinkage Value', ylab = 'Training MSE', ma
in = 'Using Boosting to Predict Salary')
```



And of course, the largest shrinkage value produces the smallest training MSE due to the boosting algorithm, which focusing on minimizing residuals.

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

```
plot(shrinkage.values, test.mse, xlab = 'Shrinkage Value', ylab = 'Test MSE', main = 'Us
ing Boosting to Predict Salary')
```



The boosting model using $\lambda = 0.05$ *was by far the best in terms of test MSE.*

e. Compare the test MSE of boosting to the test MSE that results from applying two of the regression approaches seen in Chapters 3 and 6.

```
best.index = which.min(test.mse)
best.shrinkage = shrinkage.values[best.index]
best.mse = test.mse[best.index]
best.shrinkage
```

```
## [1] 0.01
```

```
best.mse
```

```
## [1] 0.2800611
```

Boosting yields a test MSE of 0.259.

```
# In Chapter 3, we looked at linear regression, so we'll just do a multiple linear regre
ssion on the transformed Salary data
lm.fit = lm(Salary.log ~ . - Salary, data = Hitters, subset = train)
lm.predict = predict(lm.fit, newdata = Hitters[-train,])
lm.mse = mean((Hitters[-train, "Salary.log"] - lm.predict)^2)
lm.mse
```

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

The multiple linear regression yields a test MSE of 0.492, which is almost double that of the boosting model.

```
# In Chapter 6, we looked at a number of methods, including Best Subset Selection, Ridge
Regression, Lasso, PCR, and PLS
# For this exercise, we'll utilize Best Subset Selection, which requires the leaps libra
ry
library(leaps)

# Number of predictors; -2 for Salary and Salary.log columns
p = ncol(Hitters) - 2

# We need a matrix to get response predictions from the best subset method, because ther
e is no predict method for regsubsets
test.mat = model.matrix(Salary.log ~ . - Salary, data = Hitters[-train,])

# Find which subset yields the lowest test MSE and roll with that one
regfit.best = regsubsets(Salary.log ~ . - Salary, data = Hitters[train,], nvmax = p)
best.mse = Inf
for (i in 1:p){
  coefi = coef(regfit.best, id = i)
  pred = test.mat[,names(coefi)]%*%coefi
  this.mse = mean((Hitters[-train, "Salary.log"] - pred)^2)
  if (this.mse < best.mse){
    best.mse = this.mse
    best.coef = coefi
  }
}

length(best.coef)
```

```
## [1] 9
```

```
best.coef
```

```
##   (Intercept)         AtBat          Hits         Walks         Years
##  4.4722395349 -0.0032378215  0.0149552471  0.0105182255  0.0647798866
##         CRuns        CWalks     DivisionW       PutOuts
##  0.0012175332 -0.0010136188 -0.1482230049  0.0005050225
```

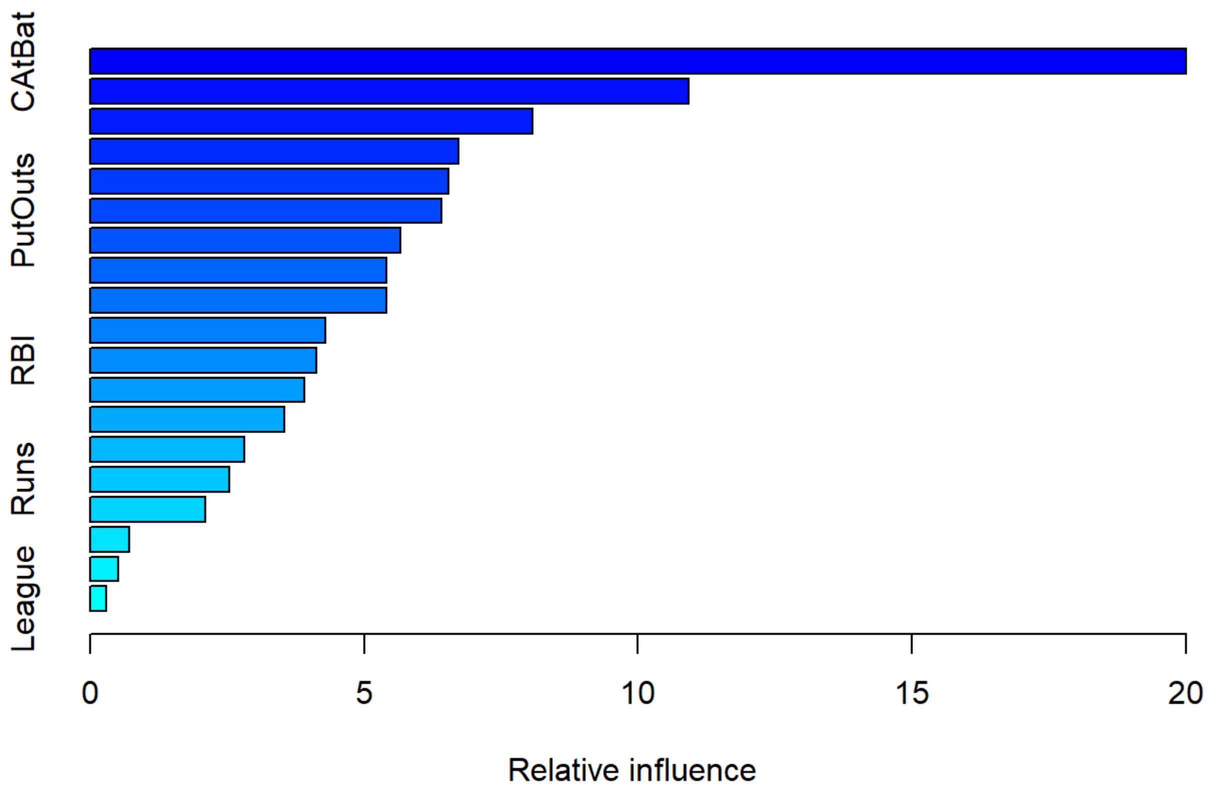
```
best.mse
```

```
## [1] 0.468457
```

The best subset selection yields an 8 predictor model (not counting the intercept) with a test MSE of 0.468, which is only slightly better than the full multiple linear regression. It is still much worse than the test MSE from boosting.

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

```
summary(boost.model)
```



```
##                 var    rel.inf
## CAtBat       CAtBat 20.0016313
## CRuns         CRuns 10.9257398
## CWalks       CWalks  8.0815671
## Years         Years  6.7184333
## CRBI           CRBI  6.5456552
## PutOuts     PutOuts  6.4155962
## CHmRun       CHmRun  5.6683505
## Walks         Walks  5.4062511
## CHits         CHits  5.4028331
## Hits           Hits  4.2915263
## RBI             RBI  4.1287601
## Assists     Assists  3.9069660
## AtBat         AtBat  3.5494994
## HmRun         HmRun  2.8195913
## Runs           Runs  2.5350925
## Errors       Errors  2.0937102
## Division   Division  0.7106661
## NewLeague NewLeague  0.5115559
## League       League  0.2865745
```

In the boosted model, the number of times at bat during a career (**CAtBat**) *had far and away the greatest relative influence on a player's salary. The second most influential predictor, the number of runs during a career* (**CRuns**), *had only half as much relative influence on a player's salary.*

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

```
set.seed(1)
bag.model = randomForest(Salary.log ~ . - Salary, data = Hitters, subset = train, mtry =
p, importance = T)
bag.predict = predict(bag.model, newdata = Hitters[-train,])
bag.mse = mean((Hitters[-train, "Salary.log"] - bag.predict)^2)
bag.mse
```

```
## [1] 0.2301184
```

The bagging approach yields a test MSE of 0.230, which is a bit less than the boost test MSE. Out of all the techniques we applied to this data set, the bagging has yielded the lowest test MSE.

Exercise 11

This question uses the **Caravan** data set.

a. Create a training set consisting of the first 1,000 observations and a test set consisting of the remaining observations.

```
train = 1:1000

# Convert the Purchase response to a factor
Caravan$Purchase.factor = ifelse(Caravan$Purchase == "Yes", 1, 0)
```

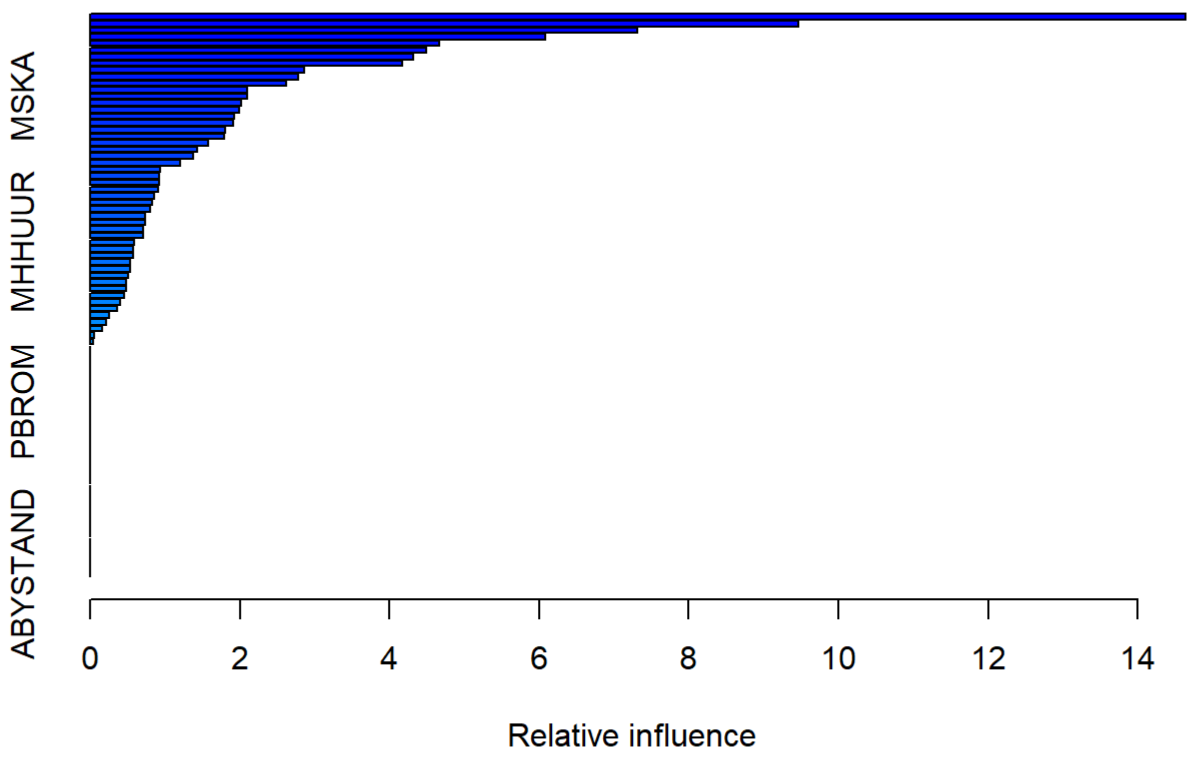
b. Fit a boosting model to the training set with **Purchase** as the response and the other variables as predictors. Use 1,000 trees and a shrinkage value of 0.01. Which predictors appear to be the most important?

```
set.seed(1)
boost.model = gbm(Purchase.factor ~ . - Purchase, data = Caravan[train,], distribution =
"bernoulli", n.trees = 1000, shrinkage = 0.01)
```

```
## Warning in gbm.fit(x, y, offset = offset, distribution = distribution, w =
## w, : variable 50: PVRAAUT has no variation.
```

```
## Warning in gbm.fit(x, y, offset = offset, distribution = distribution, w =
## w, : variable 71: AVRAAUT has no variation.
```

```
summary(boost.model)
```


MSKA
MHHUUR
PBROM
ABYSTAND
0
2
4
6
8
10
12
14
Relative influence

```
##               var     rel.inf
## PPERSAUT PPERSAUT 14.63504779
## MKOOPKLA MKOOPKLA  9.47091649
## MOPLHOOG MOPLHOOG  7.31457416
## MBERMIDD MBERMIDD  6.08651965
## PBRAND     PBRAND  4.66766122
## MGODGE     MGODGE  4.49463264
## ABRAND     ABRAND  4.32427755
## MINK3045 MINK3045  4.17590619
## MOSTYPE   MOSTYPE  2.86402583
## PWAPART   PWAPART  2.78191075
## MAUT1       MAUT1  2.61929152
## MBERARBG MBERARBG  2.10480508
## MSKA         MSKA  2.10185152
## MAUT2       MAUT2  2.02172510
## MSKC         MSKC  1.98684345
## MINKGEM   MINKGEM  1.92122708
## MGODPR     MGODPR  1.91777542
## MBERHOOG MBERHOOG  1.80710618
## MGODOV     MGODOV  1.78693913
## PBYSTAND PBYSTAND  1.57279593
## MSKB1       MSKB1  1.43551401
## MFWEKIND MFWEKIND  1.37264255
## MRELGE     MRELGE  1.20805179
## MOPLMIDD MOPLMIDD  0.93791970
## MINK7512 MINK7512  0.92590720
## MINK4575 MINK4575  0.91745993
## MGODRK     MGODRK  0.90765539
## MFGEKIND MFGEKIND  0.85745374
## MZPART     MZPART  0.82531066
## MRELOV     MRELOV  0.80731252
## MINKM30   MINKM30  0.74126812
## MHKOOP     MHKOOP  0.73690793
## MZFONDS   MZFONDS  0.71638323
## MAUT0       MAUT0  0.71388052
## MHHUUR     MHHUUR  0.59287247
## APERSAUT APERSAUT  0.58056986
## MOSHOOFD MOSHOOFD  0.58029563
## MSKB2       MSKB2  0.53885275
## PLEVEN     PLEVEN  0.53052444
## MINK123M MINK123M  0.50660603
## MBERARBO MBERARBO  0.48596479
## MGEMOMV   MGEMOMV  0.47614792
## PMOTSCO   PMOTSCO  0.46163590
## MSKD         MSKD  0.39735297
## MBERBOER MBERBOER  0.36417546
## MGEMLEEF MGEMLEEF  0.26166240
## MFALLEEN MFALLEEN  0.21448118
## MBERZELF MBERZELF  0.15906143
## MOPLLAAG MOPLLAAG  0.05263665
## MAANTHUI MAANTHUI  0.03766014
## MRELSA     MRELSA  0.00000000
## PWABEDR   PWABEDR  0.00000000
## PWALAND   PWALAND  0.00000000
## PBESAUT   PBESAUT  0.00000000
## PVRAAUT   PVRAAUT  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
## AVRAAUT   AVRAAUT  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
```

There are 85 predictors in this data set, but the **PPERSAUT** *is deemed to be much more important than the others. The next most important predictors are* **MKOOPKLA** *and* **MOPLHOOG**.

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 20%. Form a confusion matrix. What fraction of the people predicted to make a purchase do in fact make one? How does this compare with the results obtained from applying KNN or logistic regression to this data set?

```
boost.prob = predict(boost.model, newdata = Caravan[-train,], n.trees = 1000, type = "re
sponse")
boost.predict = ifelse(boost.prob > .20, "Yes", "No")
table(boost.predict, Caravan[-train, "Purchase"])
```

```
##
## boost.predict   No  Yes
##           No  4410  256
##           Yes  123   33
```

```
33 / (123 + 33)
```

```
## [1] 0.2115385
```

21.15% of people predicted to make a purchase actually do make one.

```
# use KNN, which is part of the class library, to predict purchase
library(class)

# Need to set a seed for consistency because knn() uses a random choice to settle ties
set.seed(1)

# the knn function requires the input to be a matrix, which should be standardized
standardized.mat = scale(subset(Caravan, select = -c(Purchase, Purchase.factor)))
train.mat = standardized.mat[train,]
test.mat = standardized.mat[-train,]

# We'll consider a range of K values, from 1 to the sqrt(n), with n being the number of
 observations in the training set
best.rate = 0
for (k in 1:floor(sqrt(length(train)))){
  knn.pred = knn(train.mat, test.mat, Caravan[train, "Purchase"], k = k)
  this.table = table(knn.pred, Caravan[-train, "Purchase"])
  this.rate = this.table["Yes", "Yes"] / sum(this.table["Yes",])
  if (!is.na(this.rate) & (this.rate > best.rate)){
    best.rate = this.rate
    best.table = this.table
    best.k = k
  }
}

best.table
```

```
##
## knn.pred   No  Yes
##      No  4507  279
##      Yes   26   10
```

```
best.k
```

```
## [1] 5
```

```
best.rate
```

```
## [1] 0.2777778
```

KNN was able correct about 27.78% of the test observations predicted to make a purchase. This percentage was yielded by a KNN with k = 5 and is significantly higher than the boosting model, but there are also much fewer test observations that are predicted to make a purchase. Note that we are not using a 20% probability threshold here; I don't believe that makes as much sense using KNN in R. You could make a prediction of purchase if more than 20% of the K nearest neighbors are predicted to make a purchase, but I do not believe that is convenient to implement in R, so I won't.

```
# use logistic regression to predict purchase
glm.model = glm(Purchase.factor ~ . - Purchase, data = Caravan, subset = train, family =
binomial)
```

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

```
glm.prob = predict(glm.model, newdata = Caravan[-train,], type = "response")
```

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

```
glm.pred = ifelse(glm.prob > .20, "Yes", "No")
table(glm.pred, Caravan[-train, "Purchase"])
```

```
##
## glm.pred   No  Yes
##      No  4183  231
##      Yes  350   58
```

```
58 / (350 + 58)
```

```
## [1] 0.1421569
```

Only 14.22% of the test observations predicted to make a purchase actually did. This percentage is much lower than boosting. It did actually predict much more of the test observations to make a purchase, and we were able to easily implement the 20% threshold in the logistic regression.

Exercise 12

Apply boosting, bagging, and random forests to a data set of your choice. Be sure to fit the models on a training set and to evaluate their performance on a test set. How accurate are the results compared to simple methods like linear or logistic regression? Which of these approaches yields the best performance?

```r
# We'll work with the airquality data set from the datasets package
library(datasets)

# drop any NaNs
airquality = airquality[complete.cases(airquality),]
names(airquality)
```

```
## [1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"
```

```r
dim(airquality)
```

```
## [1] 111   6
```

```r
# We'll create models to predict the maximum daily temperature in degrees Farenheit (Tem
p)

# Number of observations
n = nrow(airquality)

# Number of predictors
p = ncol(airquality) - 1

# First, split the data into a training set and a test set
set.seed(1)
train = sample(n, n/2)

# Create a data frame to store test MSE results for each model
df = data.frame(model.name = rep(NA, 5), test.mse = rep(NA, 5))
```

```r
# Create the boosting model
df[1, "model.name"] = "boosting"
boost.model = gbm(Temp ~ ., data = airquality[train,], distribution = "gaussian", n.tree
s = 1000)
boost.predict = predict(boost.model, newdata = airquality[-train,], n.trees = 1000)
boost.mse = mean((airquality[-train, "Temp"] - boost.predict)^2)
df[1, "test.mse"] = boost.mse
```

```r
# Create the bagging model
set.seed(1)
df[2, "model.name"] = "bagging"
bag.model = randomForest(Temp ~ ., data = airquality, subset = train, mtry = p, importan
ce = T)
bag.predict = predict(bag.model, newdata = airquality[-train,])
bag.mse = mean((airquality[-train, "Temp"] - bag.predict)^2)
df[2, "test.mse"] = bag.mse
```

```
# Create the random forests model
set.seed(1)
df[3, "model.name"] = "random forests"
rf.model = randomForest(Temp ~ ., data = airquality, subset = train, mtry = p/2, importa
nce = T)
rf.predict = predict(rf.model, newdata = airquality[-train,])
rf.mse = mean((airquality[-train, "Temp"] - rf.predict)^2)
df[3, "test.mse"] = rf.mse
```

```
# Create the multiple linear regression model
df[4, "model.name"] = "linear regression"
lm.model = lm(Temp ~ ., data = airquality, subset = train)
lm.predict = predict(lm.model, newdata = airquality[-train,])
lm.mse = mean((airquality[-train, "Temp"] - lm.predict)^2)
df[4, "test.mse"] = lm.mse
```

```
# Logistic regression is specific to qualitative response variables, but we can perform
 a log transformation on Temp and fit a linear regression on that
df[5, "model.name"] = "log transform regression"
log.model = lm(log(Temp) ~ ., data = airquality, subset = train)
log.predict = exp(predict(log.model, newdata = airquality[-train,]))
log.mse = mean((airquality[-train, "Temp"] - log.predict)^2)
df[5, "test.mse"] = log.mse
```

```
# Show the results
df
```

```
##                  model.name test.mse
## 1                  boosting 64.10107
## 2                   bagging 40.37541
## 3            random forests 36.05172
## 4         linear regression 51.59265
## 5 log transform regression 55.33704
```

We can see that the random forests model performed better than any of the other models. It had a much lower test MSE than the linear regression and the log transform regression models. The bagging model had a test MSE which was very close to that of the random forests model. The boosting model performed very poorly on this data set compared to the other models.