Big Data Analytics with R - Solution to Coursework 3

1. Random Forest (Textbook 8.4.7)

1.a. First we load the `Boston` dataset and the `randomForest` library for analysis, then generate the training and test datasets by splitting the dataset in two. We also keep track of the actual values of the response variable for later.

```r
library(MASS)
library(randomForest)
```

```
## Warning: package 'randomForest' was built under R version 3.1.2
```

```
## randomForest 4.6-10
## Type rfNews() to see new features/changes/bug fixes.
```

```r
set.seed(1)
train <- sample(nrow(Boston), nrow(Boston)/2)
test <- Boston[-train, ]
actual <- Boston[-train, "medv"]
```

Since we will be generating data using different values for the `mtry` and `ntree` hyperparameters, we also write a function to more easily generate the data we need for the plot.

```r
get_test_mse <- function(mtry, ntree) {
  fit <- randomForest(medv ~., data=Boston, subset=train, mtry=mtry,
                      importance=TRUE, ntree=ntree)
  preds <- predict(fit, newdata=test)
  return(mean((preds - actual)^2))
}
```

The function `get_test_mse(mtry, ntree)` allows us to get the test mean squared error (MSE) for a given value of `mtry` and `ntree`. In order to be able to replicate the style of plot shown in figure 8.10 in the book, we need to compute the test MSE for a range of tree sizes for a particular value of `mtry` and then repeat that at various values of `mtry`.

To make this process slightly easier, we create a new function `get_mse_plot_data` that will compute the test MSE for trees of sizes in the range [1, ntree] and return a list of corresponing test MSE values. Note that since we need a partial function here, we use the R package functional to curry the function `get_test_mse(mtry, ntree)` to make it easier to use `sapply` rather than more costly for loops.

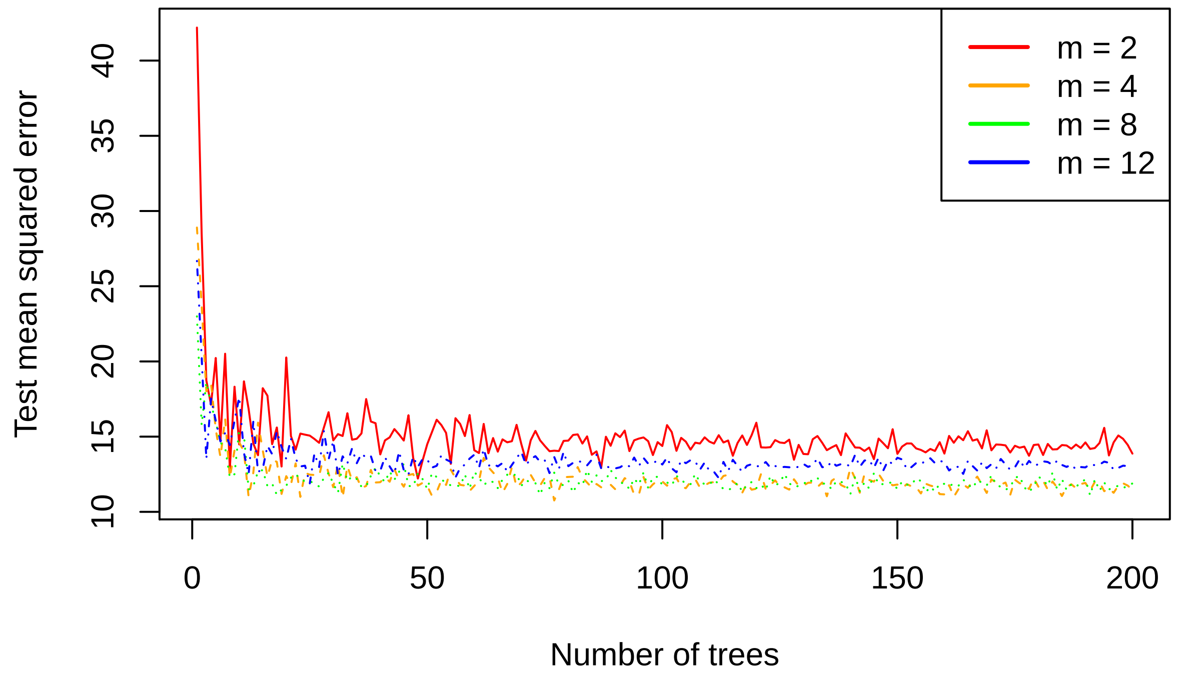
```r
library(functional)
```

```
## Warning: package 'functional' was built under R version 3.1.2
```

```r
get_mse_plot_data <- function(mtry, ntree) {
  get_test_mse_by_ntree <- Curry(get_test_mse, mtry=mtry)
  treesize <- 1:ntree
  return(sapply(treesize, get_test_mse_by_ntree))
}
```

For the `Boston` dataset we have 13 predictors, excluding the response variable `medv`. Since it is customary to use m $= \sqrt{p} = 3.6055513$, we generate test MSEs for $m \in \{2, 4, 8, 12\}$, for tree sizes ranging from 1 to 200. Note that m $=$ 13 leads to a bagging classifier, not a random forest classifier.

```r
mse02 <- get_mse_plot_data(2, 200)
mse04 <- get_mse_plot_data(4, 200)
mse08 <- get_mse_plot_data(8, 200)
mse13 <- get_mse_plot_data(12, 200)
matplot(1:200, cbind(mse02, mse04, mse08, mse13), type="l",
        col=c("red", "orange", "green", "blue"),
        xlab="Number of trees", ylab="Test mean squared error")
legend(x="topright", legend=c("m = 2", "m = 4", "m = 8", "m = 12"),
       col=c("red", "orange", "green", "blue"), lwd=2)
```



From the plot we see that the random forest with the smallest number of predictors to choose from (m $=$ 2) performs worst in this situation, and there is considerable variance when the number of trees is small, as might be expected. The models with m $=$ 4 (closest to $\sqrt{p}$) and m $=$ 8 have the lowest mean square error on test data and also show lower model variance with small forests.

```r
# clean up before moving on to next question
rm(list = ls())
```

2. Regression Tree (Textbook 8.4.8)

2.a. First load the `ISLR` package containing the `Carseats` dataset and set the seed for the session, before splitting the data set evenly into training and test sets.

```r
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 3.1.1
```

```r
set.seed(1)
training_indices <- sample(nrow(Carseats), nrow(Carseats)/2)
train <- Carseats[training_indices, ]
test <- Carseats[-training_indices, ]
```

2.b. Load the `tree` package, which contains the `tree()` regression tree function.

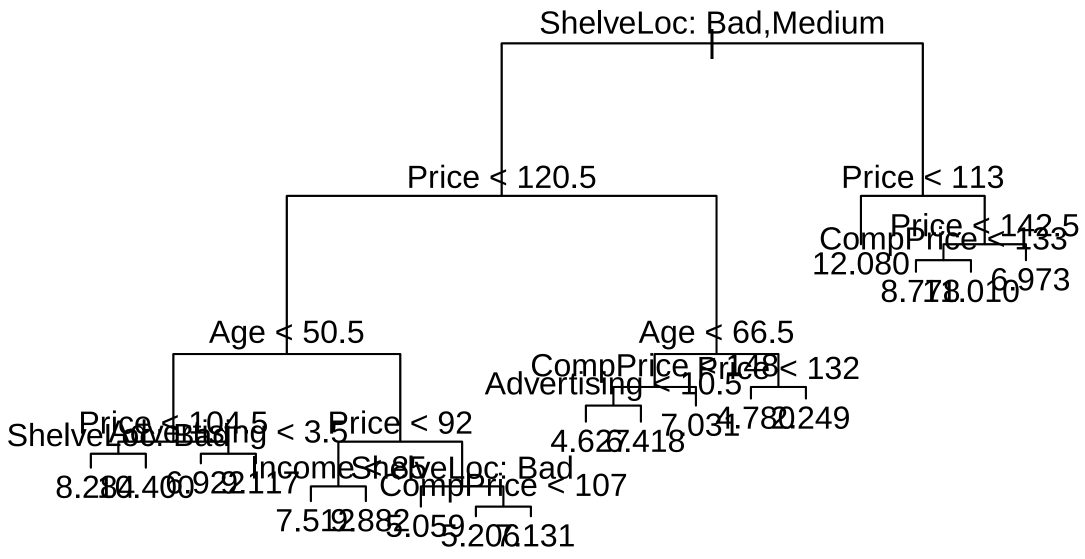
```r
library(tree)
```

```
## Warning: package 'tree' was built under R version 3.1.2
```

```r
fit <- tree(Sales ~ ., data=Carseats, subset=training_indices)
summary(fit)
```

```
##
## Regression tree:
## tree(formula = Sales ~ ., data = Carseats, subset = training_indices)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Age"         "Advertising" "Income"
## [6] "CompPrice"
## Number of terminal nodes:  18
## Residual mean deviance:  2.36 = 429.5 / 182
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -4.2570 -1.0360  0.1024  0.0000  0.9301  3.9130
```

```r
plot(fit)
text(fit, pretty=0)
```



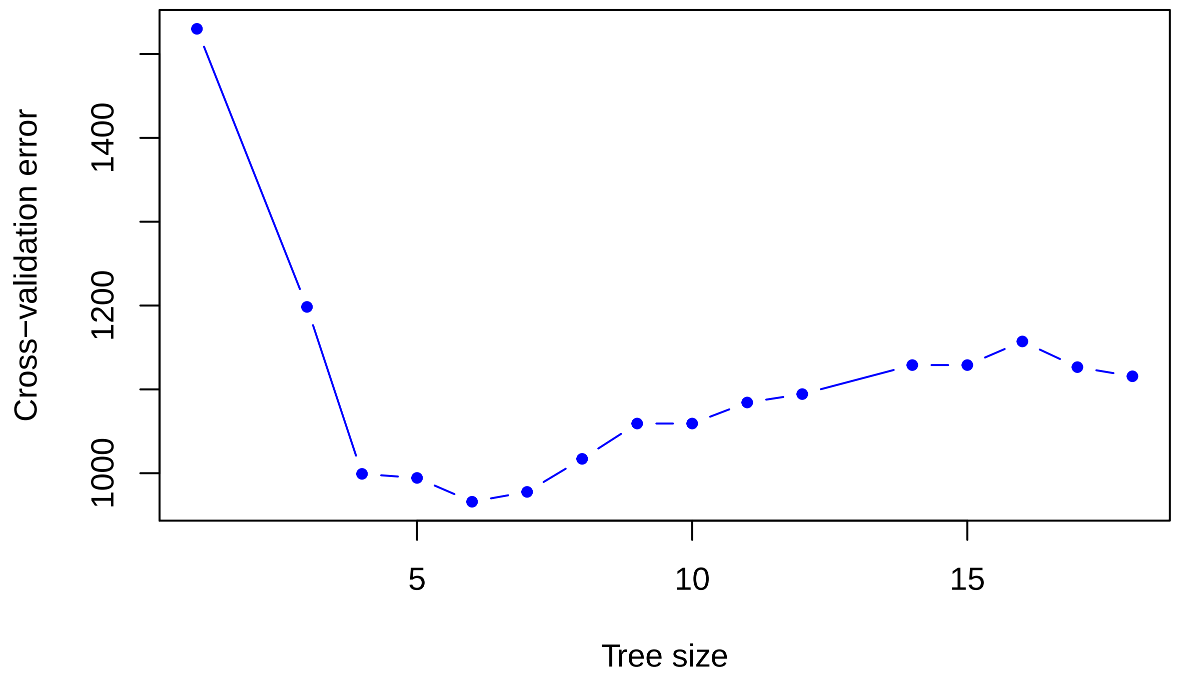
```r
preds <- predict(fit, newdata=test)
actual <- test$Sales
mean((preds - actual)^2)
```

```
## [1] 4.148897
```

Based on the tree obtained from the training data, the most important variables are `ShelveLoc` (shelf location) and `Price`. The tree regression model gives a test MSE of 4.1488975.

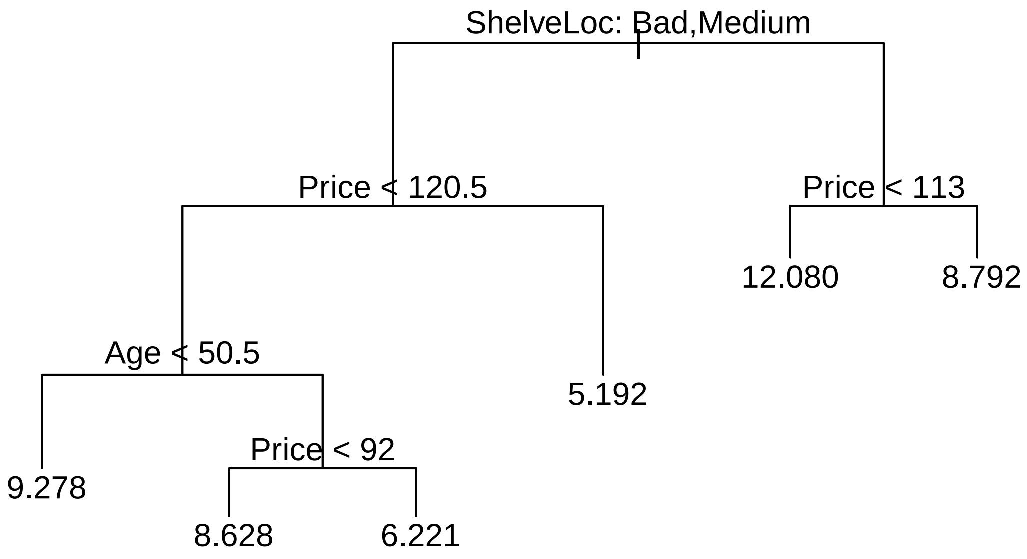
2.c. We use the `cv.tree()` function to perform cross validation using the regression tree fit in part (b). Since there are 400 rows in the `Carseats` dataset, we perform K-fold cross validation with k = 5 to get an even split across the five groups.

```r
tree_cv <- cv.tree(fit, K=5)
plot(tree_cv$size, tree_cv$dev, type="b",
     xlab="Tree size", ylab="Cross-validation error", pch=20, col="blue")
```



From the plot we see that the minimum cross-validation error is obtained with a tree size of 6. Using the `tree.prune()` function, we can prune the tree to size 6.

```r
pruned_tree <- prune.tree(fit, best=6)
plot(pruned_tree)
text(pruned_tree, pretty=0)
```



Using this pruned tree, we can again calcuated the MSE for the test data.

```r
preds <- predict(pruned_tree, newdata=test)
mean((preds - actual)^2)
```

```
## [1] 5.334766
```

The test MSE for the pruned tree is 5.3347657, which is higher than that for the unpruned tree. This is typical of simple regression trees, which have high variance. Note however that the interpretation has been simplified.

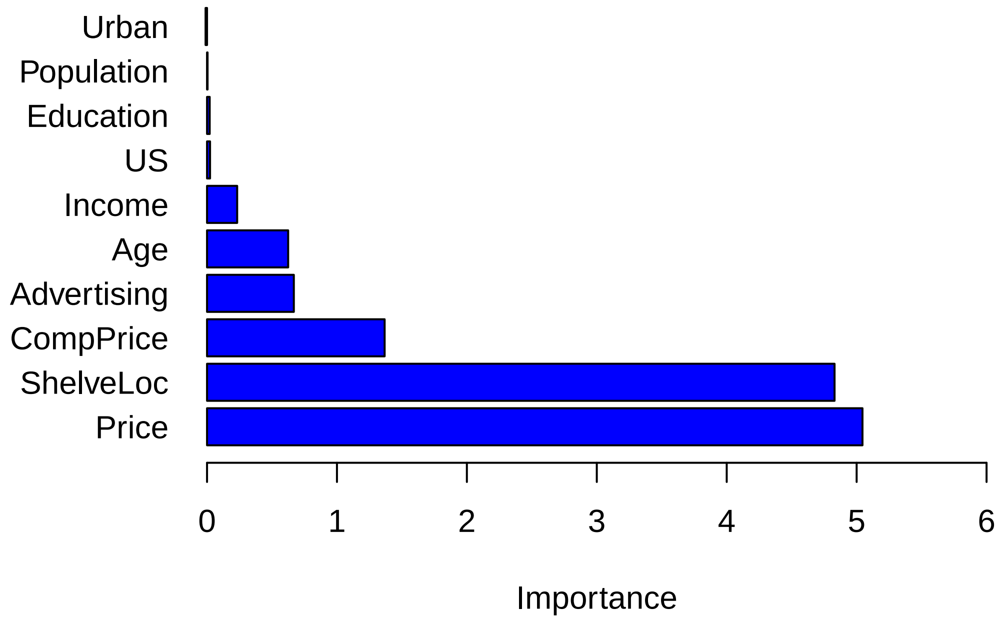
2.d. In order to use bagging, we again use the `randomForest` package. Bagging chooses randomly from all of the p = 10 predictors (excluding `Sales`) at each new branch.

```r
set.seed(1)
bag_fit <- randomForest(Sales ~ ., data=Carseats, mtry=10, importance=TRUE)
preds <- predict(bag_fit, newdata=test)
mean((preds - actual)^2)
```

```
## [1] 0.376406
```

We see that the test MSE has been reduced dramatically, to 0.376406. Using the `importance` return value, we can plot the importance values for the predictors.

```r
par(mar = c(6,8,4,4) + 0.1)
barplot(sort(bag_fit$importance[, 1], decreasing=TRUE),
        horiz=TRUE, las=1, col="blue", xlab="Importance", xlim=c(0,6))
```



The plot shows that `Price` and `ShelveLoc` are the most important predictor variables.

2.e. For the random forest approach, we now use a smaller value for m. It is typical to use m $= \sqrt{p}$. For p $=$ 10 we have $\sqrt{p} = 3.1622777$ so we use m $= 3$.

To try a range of values for m, we first create a utility function to more easily compute the test MSE.

```r
get_test_mse <- function(m) {
set.seed(1)
rf_fit <- randomForest(Sales ~ ., data=Carseats, mtry=m, importance=TRUE)
preds <- predict(rf_fit, newdata=test)
return(mean((preds - actual)^2))
}
```

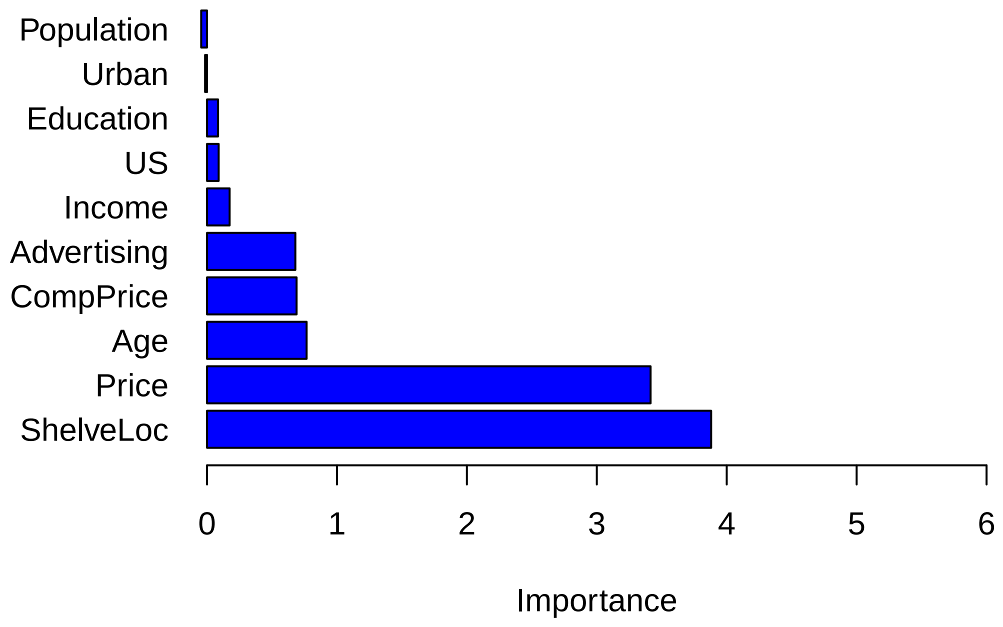
We see that the random forests model with m $= 3$ has a test MSE of

```r
get_test_mse(3)
```

```
## [1] 0.5350276
```

We can again produce a variable importance plot.

```r
rf_fit <- randomForest(Sales ~ ., data=Carseats, mtry=3, importance=TRUE)
par(mar = c(6,8,4,4) + 0.1)
barplot(sort(rf_fit$importance[, 1], decreasing=TRUE),
        horiz=TRUE, las=1, col="blue", xlab="Importance", xlim=c(0,6))
```



The plot shows that `Price` and `ShelveLoc` are once again the most important predictors, and as in the bagging model, we see there is some weight given to `CompPrice`, `Advertising` and `Age`.

Using a slightly higher value for m, m = p/2 = 5, we have

```r
get_test_mse(5)
```

```
## [1] 0.4361195
```

and for m = 8 we obtain

```r
get_test_mse(8)
```

```
## [1] 0.3841172
```

The results are comparable to those obtained for the bagging model and there is the suggestion that allowing more predictors to be available at each split allows for a better test MSE, as evinced by the lower test MSE and also that the variable importance plot now has up to five predictors with relatively high importance.

```
# clean up before moving on to next question
rm(list = ls())
```

3. Classification Tree (Textbook 8.4.9)

3.a. We first load the `ISLR` package to access the `OJ` dataset, then create a training set of 800 randomly chosen observations and a test set containing the remaining 270 observations.

```
library(ISLR)
set.seed(1)
training_indices <- sample(nrow(OJ), 800)
train <- OJ[training_indices, ]
test <- OJ[-training_indices, ]
nrow(train)
```

```
## [1] 800
```

```
nrow(test)
```

```
## [1] 270
```

3.b. To create a tree classifier, we use the `tree` package, then fit a model with `Purchase` as the response and the remaining 17 variables as the predictors.

```
tree_fit <- tree(Purchase ~ ., data=OJ, subset=training_indices)
summary(tree_fit)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ, subset = training_indices)
## 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
```

From the fit summary we see that just four of the 17 predictors were used - `LoyalCH`, `PriceDiff`, `SpecialCH`, and `ListPriceDiff`. The resulting tree has 8 terminal nodes and a training misclassification error rate of 0.165.

3.c. To get a detailed text representation of the tree, we enter the tree object name.

```
tree_fit
```

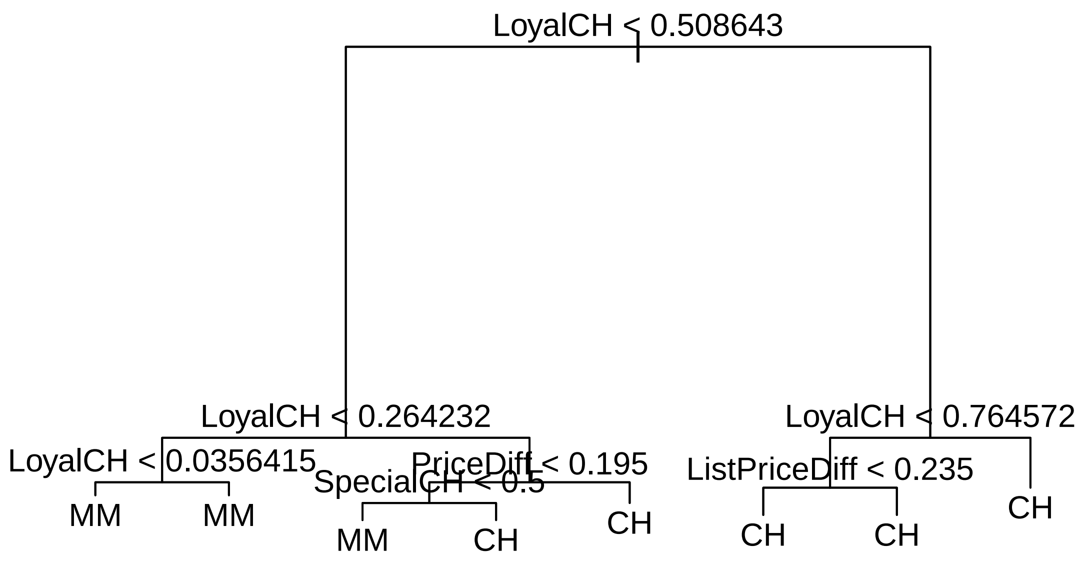
```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1064.00 CH ( 0.61750 0.38250 )
##    2) LoyalCH < 0.508643 350  409.30 MM ( 0.27143 0.72857 )
##      4) LoyalCH < 0.264232 166  122.10 MM ( 0.12048 0.87952 )
##         8) LoyalCH < 0.0356415 57   10.07 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 ) *
```

The possible labels are `CH` (Citrus Hill) and `MM` (Minute Maid) - two brands of orange juice. Taking node 8 as an example of a terminal node (as indicated by the *), we see that if an observation has `LoyalCH` < 0.508643 and `LoyalCH` < 0.264232 and `LoyalCH` < 0.0356415 at this final branch, then the observation is classified as `MM` (i.e. the customer purchased Minute Maid orange juice). Of the 166 training observations considered at the branch, 57 were below 0.0356415 and so were classified as `MM`.

Interestingly, the branch leading to this split leads to two terminal nodes (8 and 9) each being classified as `MM`. They are separated because they have significantly different deviance using this cutpoint.

3.d. A plot of the classification tree is given by

```r
plot(tree_fit)
text(tree_fit, pretty=0)
```



From the heights of the splits, it appears that the main determinant when choosing between the two orange juice brands is `LoyalCH` (loyalty to Citrus Hill). The height of the branches is an indication of the differences in deviance between the nodes and since five of the six branches are at very similar heights, it seems likely – and is entirely consistent – that the these additional factors have little influence on a customer's buying behaviour.

The first split at the root node is `LoyalCH` < 0.508643, and all terminal nodes to the right are classified as `CH`. This just says that those customers that are more loyal to Citrus Hill than not (i.e. > 0.5) are more likely to buy Citrus Hill than Minute Maid orange juice.

3.e. We first predict the responses for the test observations and then compare with the actual purchase decision.

```r
preds <- predict(tree_fit, newdata=test, type="class")
actual <- test$Purchase
table(predicted = preds, actual = actual)
```

```
##          actual
## predicted  CH  MM
##        CH 147  49
##        MM  12  62
```

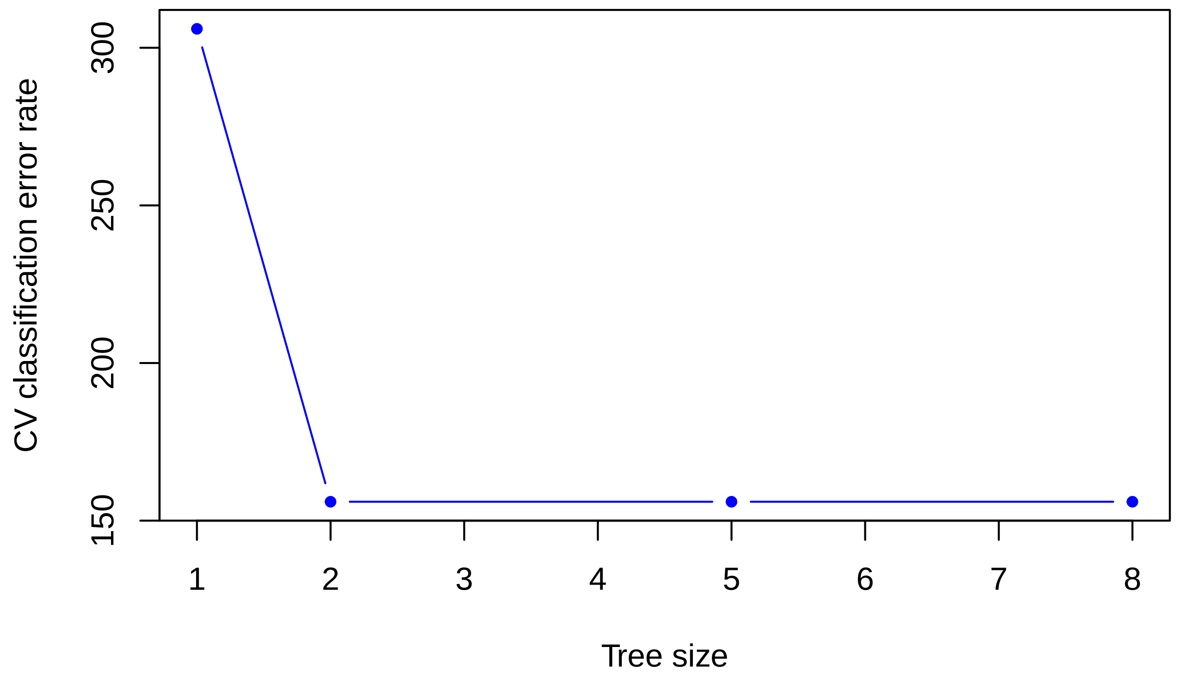
```r
mean(preds != actual)
```

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

With this model, we have a test misclassification error rate of 0.2259259.

3.f-g. We use the `cv.tree()` function to determine the optimal tree size for this model.

```r
cv_fit <- cv.tree(tree_fit, FUN=prune.misclass)
plot(cv_fit$size, cv_fit$dev, type="b",
     xlab="Tree size", ylab="CV classification error rate",
     col="blue", pch=20)
```



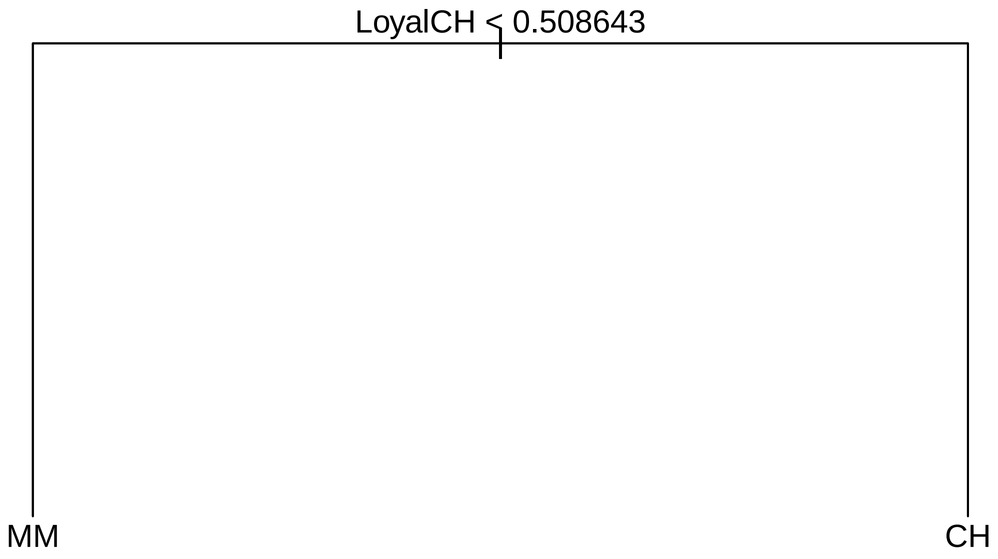
3.h.From the plot we see that the optimal tree size is 2.

3.i. We now use the `prune.misclass()` function to prune the classification tree.

```
pruned_fit <- prune.misclass(tree_fit, best=2)
summary(pruned_fit)
```

```
##
## Classification tree:
## snip.tree(tree = tree_fit, nodes = c(3L, 2L))
## Variables actually used in tree construction:
## [1] "LoyalCH"
## Number of terminal nodes:  2
## Residual mean deviance:  0.9115 = 727.4 / 798
## Misclassification error rate: 0.1825 = 146 / 800
```

```
plot(pruned_fit)
text(pruned_fit, pretty=0)
```



3.j. The pruned tree has a training misclassification error rate of 0.182, which is slightly higher than that for the unpruned tree (0.165).

3.k. The test misclassification error rate is obtained as follows.

```r
preds <- predict(pruned_fit, newdata=test, type="class")
actual <- test$Purchase
table(predicted = preds, actual = actual)
```

```
##          actual
## predicted  CH  MM
##        CH 119  30
##        MM  40  81
```

```r
mean(preds != actual)
```

```
## [1] 0.2592593
```

Here the test misclassification error rate is 0.2592593, which is higher than for the unpruned tree (0.2259). That said, while the pruned tree has slightly worse performance, the interpretability is significantly improved, as there is only one determining predictor, as might have been expected from the unpruned tree.