Lab 08 - Trees

Ken Ye

10/24/2019

Note: some of the results for this lab depend on your version of R and the version of the packages that are installed on your computer. My results differ from the ones already in the textbook. The interpretations after printing numerical results are meant as general trends, so don't worry if specific numbers don't match exactly.

1. Classification Trees

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

```
#install.packages("tree")  ## might need to update R to use
#install.packages("gbm")
```

```
library(tree)
```

We will first use classification trees to analyze the Carseats data set. In this data, Sales is a continuous variable and we begin by first recoding it as a binary variable, using the ifelse() function. We will create a new variable called High that will take on a value of Yes if Sales > 8 and will take on a value of No otherwise.

```
library(ISLR)
attach(Carseats)
High <- ifelse(Sales > 8, "Yes", "No")
```

We can then use the data.frame() function to merge High with the rest of the Carseats data.

```
Carseats <- data.frame(Carseats, High)
Carseats$High = as.factor(Carseats$High)
```

Now, we can use the tree() function to fit a classification tree in order to predict High using all variables but Sales. The tree() function has syntax that is quite similar to the syntax of the lm() function.

```
tree.carseats <- tree(High ~. -Sales, Carseats)
# Fit the model on all variables except for Sales
```

The summary() function can again be used to list the variables that are used as internal nodes in the tree, the number of terminal nodes and the (training) error rate.

```r
class(Carseats$High)
```

```
## [1] "factor"
```

```r
summary(tree.carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## 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
```

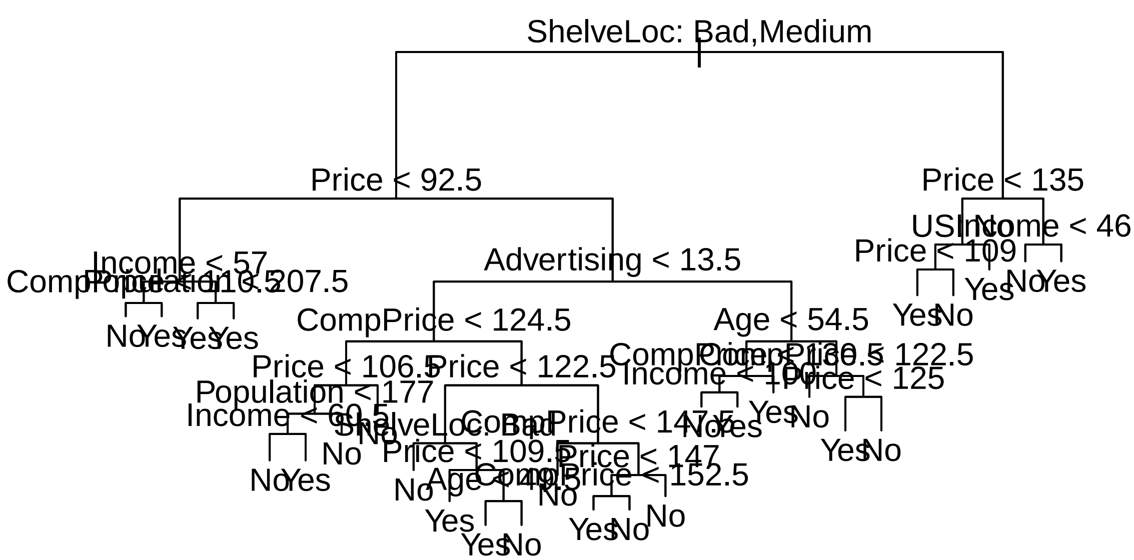
The training error is around 9%. For classification trees, the deviance reported in the output of `summary()` is given by:

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

where n_{mk} is the number of observations in the m^{th} terminal node that belong to the k^{th} 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 divded 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.

```r
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



The most important predictor 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 criterior (e.g. `Price < 92.5`), the number of observations in that branch, the deviance, the overall prediction for the branch (`Yes` or `No`), and the fraction of observations in that branch that take on values of `Yes` and `No`. Branches that lead to terminal nodes are indicated using asterisks.

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##          18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##              80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##               160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##               161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##              81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##            42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                 342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
##                 343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##            43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##              86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##              87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##               174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                 348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                 349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##               175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##        11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##          22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##            44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##              88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##              89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##            45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
##          23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##            46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
```

```
##           47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##             94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##             95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##    3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##      6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##       12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
##         24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##         25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##       13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##      7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##       14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##       15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

In order to properly evaluate the performance of a classification tree on this data, we must estimate the test error rather than just the training error. We can split the observations into a training set and a test set, build the tree using the training set, then 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 71.5% of the locations of the test data set.

```
set.seed(2)
train <- sample(1:nrow(Carseats), 200) ## split data into train and test
Carseats.test <- Carseats[-train,]
High.test <- High[-train]
tree.carseats <- tree(High ~ . - Sales, Carseats, subset = train)
tree.pred <- predict(tree.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##       No  104  33
##       Yes  13  50
```

```
sum(diag(table(tree.pred, High.test)))/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 the `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 in the equation below:)

$$\sum_{m=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y}_{R_m})^2 + \alpha|T|.$$

```
set.seed(3)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
names(cv.carseats)
```

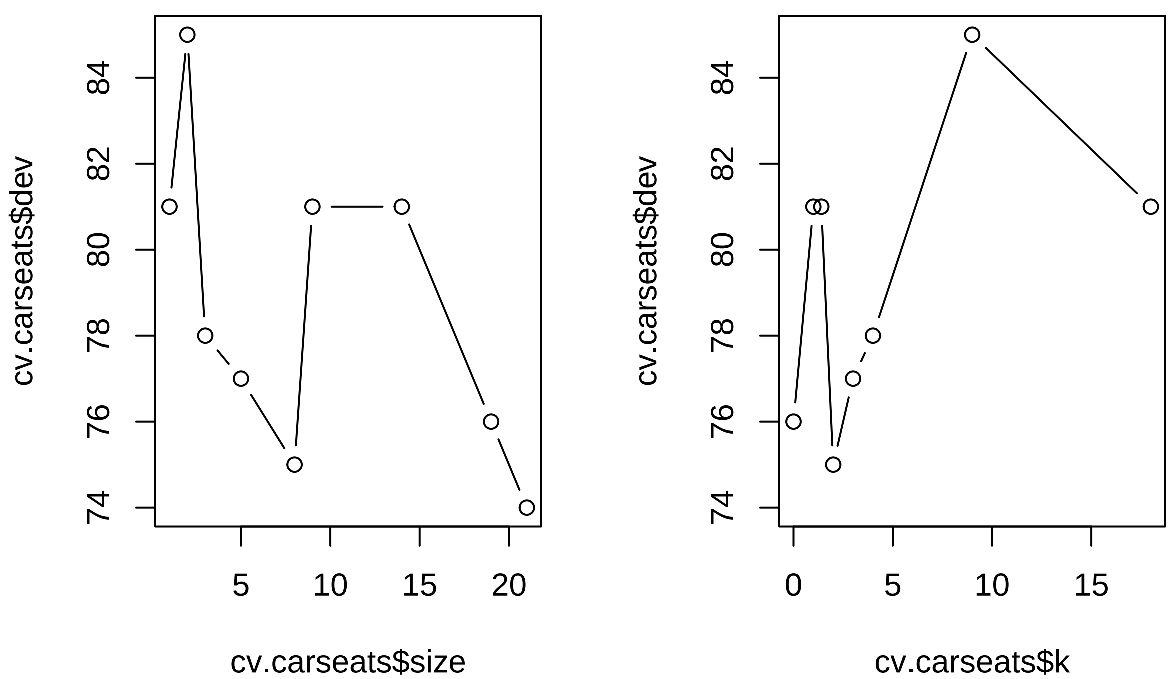
```
## [1] "size"   "dev"    "k"      "method"
```

```
cv.carseats
```

```
## $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"
```

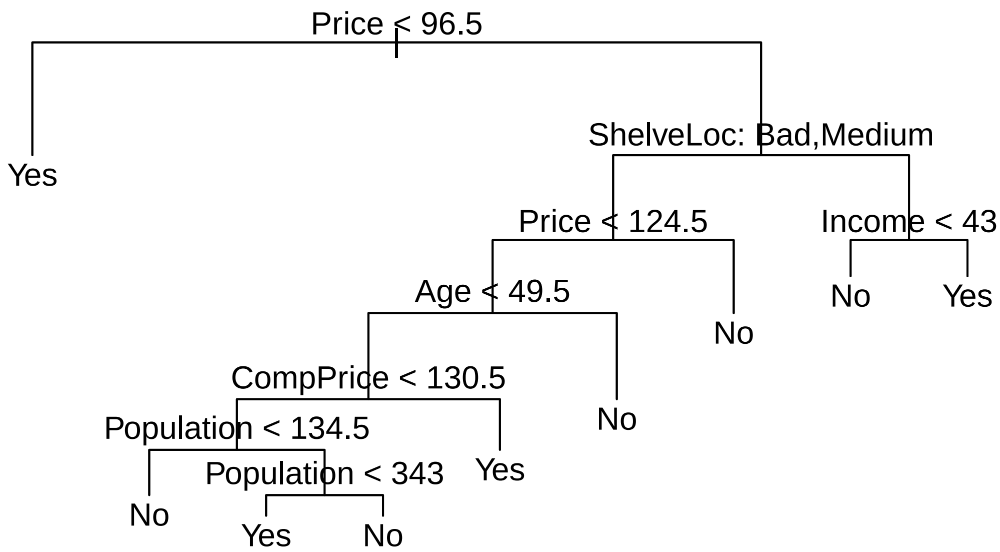
Note that, despite the name, `dev` corresponds to the cross-validation error rate in this instance. The tree with 9 terminal nodes results in the lowest cross-validation error rate, with 50 cross-validation errors. We can plot the error rate as a function of both `size` and `k`.

```
par(mfrow = c(1,2))
plot(cv.carseats$size, cv.carseats$dev, type = "b")
plot(cv.carseats$k, cv.carseats$dev, type = "b")
```



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

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



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

```
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##           High.test
## tree.pred No Yes
##       No  97  25
##       Yes 20  58
```

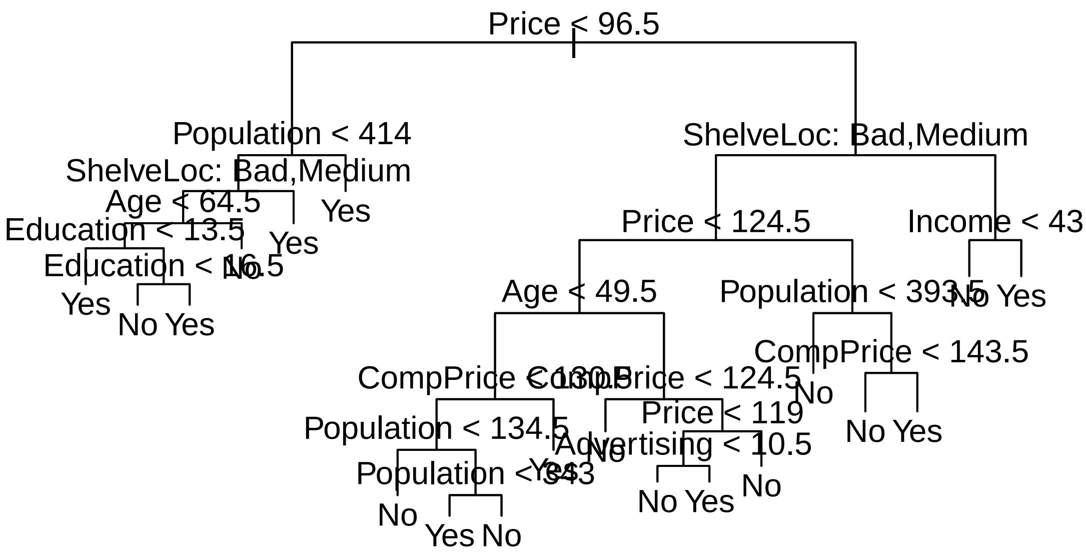
```
sum(diag(table(tree.pred, High.test)))/200
```

```
## [1] 0.775
```

Now, 77% of the test observations are correctly classified, so not only has the pruning process produced a more interpretable tree, but it has also improved the classification accuracy (slightly).

If we increase the value of `best`, we obtain a larger pruned tree with lower classification accuracy:

```
prune.carseats <- prune.misclass(tree.carseats, best = 15)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```




```r
tree.pred <- predict(prune.carseats, Carseats.test, type = "class")
table(tree.pred, High.test)
```

```
##          High.test
## tree.pred  No Yes
##       No  102  30
##       Yes  15  53
```

```r
sum(diag(table(tree.pred, High.test)))/200
```

```
## [1] 0.775
```

2. Regression Trees

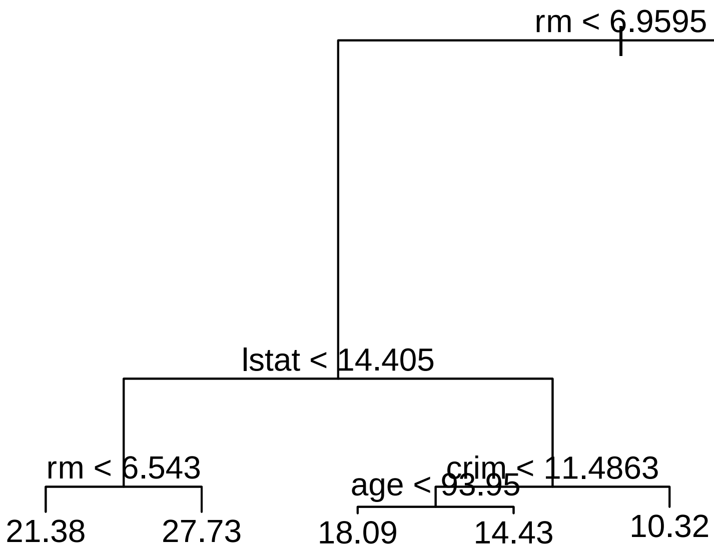
Here we fit a regression tree to the `Boston` data set. First, we create a training set and fit the tree to the training data.

```r
library(MASS)
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston)/2) # 50% split
tree.boston <- tree(medv ~., Boston, subset = train)
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = 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
```

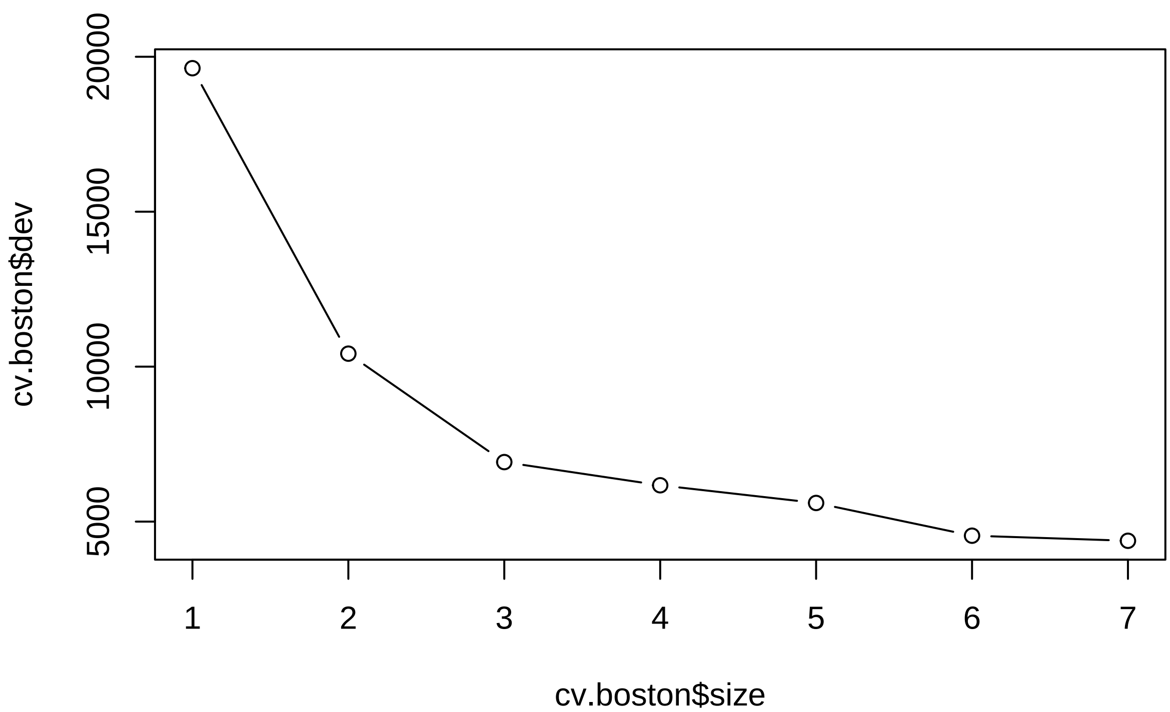
Notice that the output of `summary()` indicates that only three of the variables have been used in constructing the tree. In the context of a regression tree, the deviance is simply the sum of squared errors for the tree. We now plot the tree:

```r
plot(tree.boston)
text(tree.boston, pretty = 0)
```



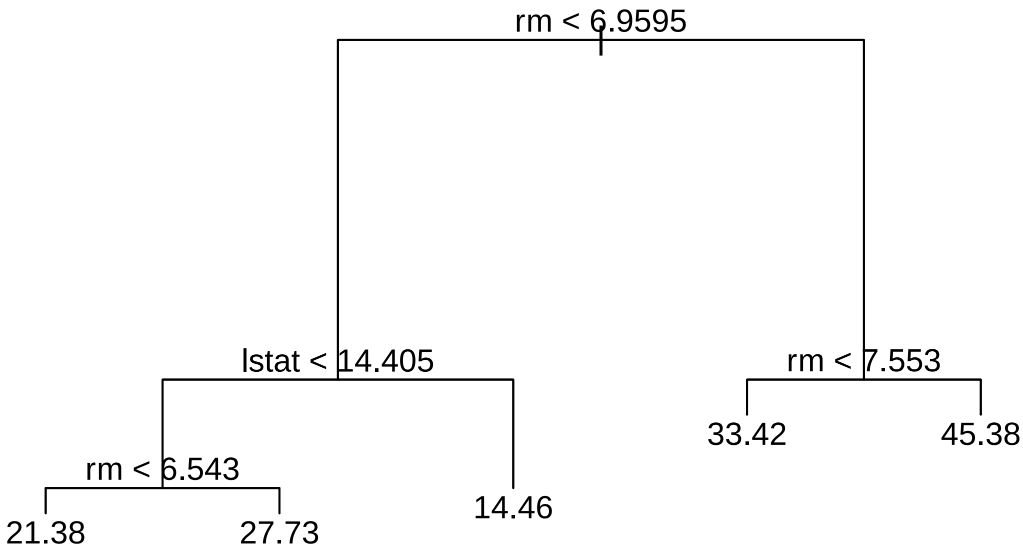
We now use the `cv.tree()` function to see whether pruning the tree will improve performance.

```r
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size, cv.boston$dev, type = "b")
```



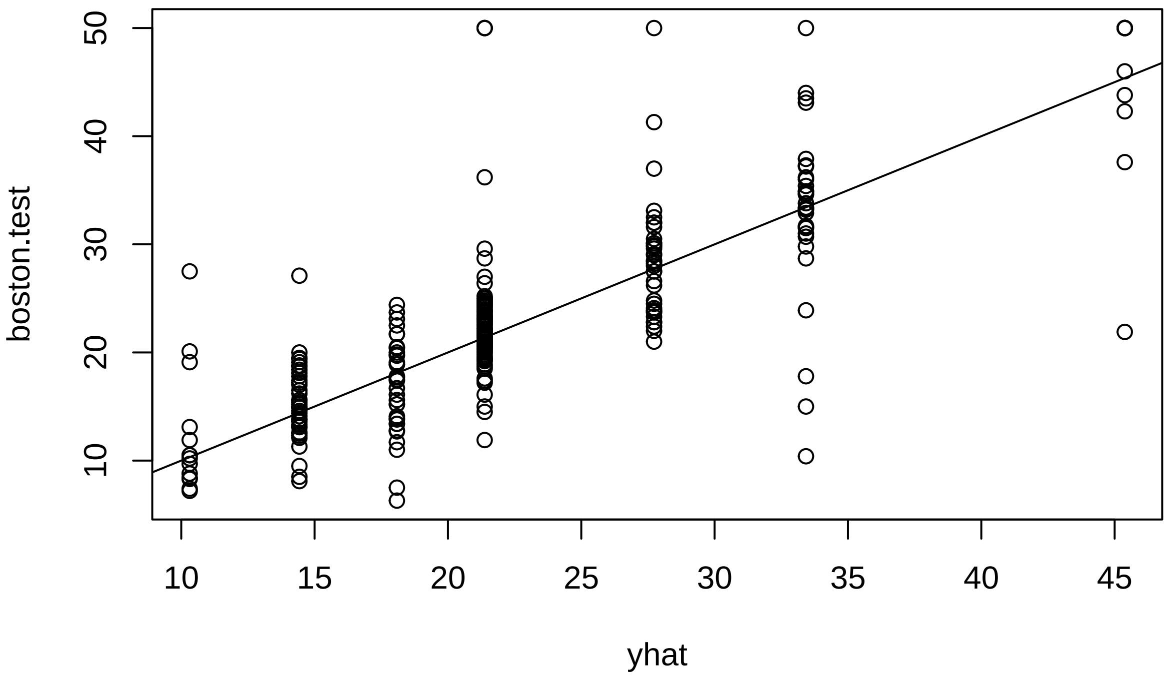
In this case, the most complex tree is selected by cross-validation. However, if we wish to prune the tree, we could do so as follows, using the `prune.tree()` function:

```r
prune.boston <- prune.tree(tree.boston, best = 5)
plot(prune.boston)
text(prune.boston, pretty = 0)
```



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

```r
yhat <- predict(tree.boston, newdata = Boston[-train,])
boston.test <- Boston[-train, "medv"]
plot(yhat, boston.test)
abline(0,1)
```




```r
mean((yhat-boston.test)^2)
```

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

In other words, the test set MSE associated with the regression tree is 25.05. The square root of the MSE is therefore around 5.005, indicating that this model leads to test predictions that are within around $5,005 of the true median home value for the suburb.

3. 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 `randomForest` installed on your computer.

Recall that bagging is simply a special case of random forest with $m = p$. Therefore, the `randomForest()` function can be used to perform both random forests and bagging. We perform bagging as follows:

```
library(randomForest)
```

```
## randomForest 4.7-1.1
```

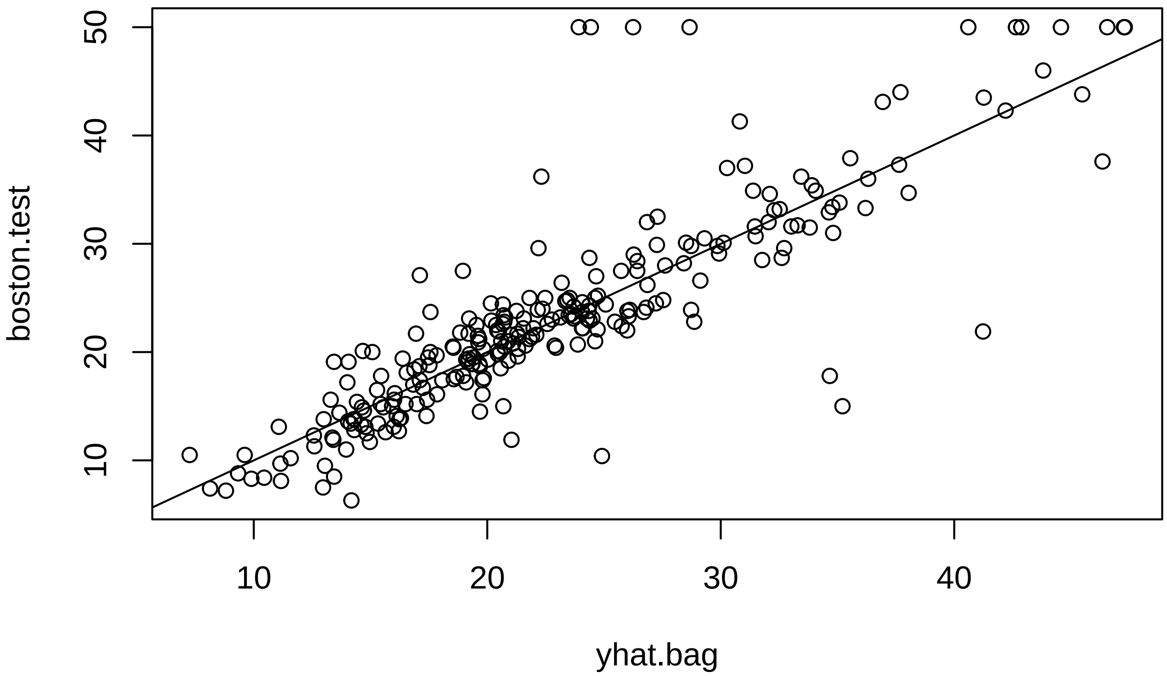
```
## Type rfNews() to see new features/changes/bug fixes.
```

```
set.seed(1)
bag.Boston <- randomForest(medv~., data = Boston, subset = train,
                           mtry = 13, importance = TRUE)
bag.Boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = TRUE,      subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 13
##
##           Mean of squared residuals: 11.39601
##                     % Var explained: 85.17
```

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

```
yhat.bag <- predict(bag.Boston, newdata = Boston[-train,])
plot(yhat.bag, boston.test)
abline(0,1)
```




```
mean((yhat.bag - boston.test)^2)
```

```
## [1] 23.59273
```

The test set MSE associated with the bagged regression tree is 13.16, almost half that obtained by an optimally-pruned single tree. We could change the number of trees grown by `randomForest()` using the `ntree` argument.

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

```
## [1] 23.66716
```

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`.

```
set.seed(1)
rf.boston <- randomForest(medv~., data = Boston, subset = train,
                          mtry = 6, importance = TRUE)
yhat.rf <- predict(rf.boston, newdata = Boston[-train,])
mean((yhat.rf - boston.test)^2)
```

```
## [1] 19.62021
```

The test set MSE is 11.31; this indicates that random forests yielded an improvement over bagging in this case.

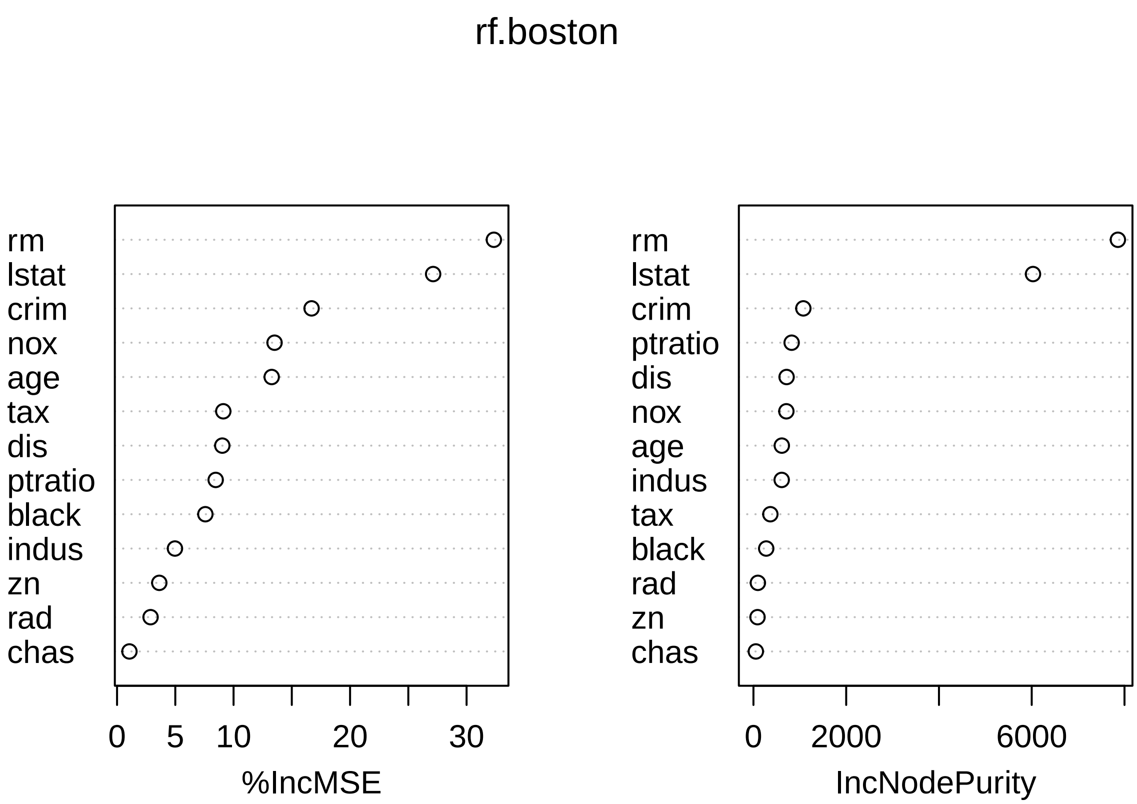
Using the `importance()` function, we can view the importance of each variable.

```r
importance(rf.boston)
```

```
##           %IncMSE IncNodePurity
## crim    16.697017    1076.08786
## zn       3.625784      88.35342
## indus    4.968621     609.53356
## chas     1.061432      52.21793
## nox     13.518179     709.87339
## rm      32.343305    7857.65451
## age     13.272498     612.21424
## dis      9.032477     714.94674
## rad      2.878434      95.80598
## tax      9.118801     364.92479
## ptratio  8.467062     823.93341
## black    7.579482     275.62272
## lstat   27.129817    6027.63740
```

Two measures of variable importance are reported. The former is based on the mean decrease in 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 (this was plotted in Figure 8.9 in the text). 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.

```r
varImpPlot(rf.boston)
```



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

4. Boosting

Here we use the `gbm()` package, and within it the `gbm()` function, to fit boosted regression trees to the `Boston` data set. We run `gbm()` with the option `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 5000 trees, and the option `interaction.depth = 4` limits the depth of each tree.

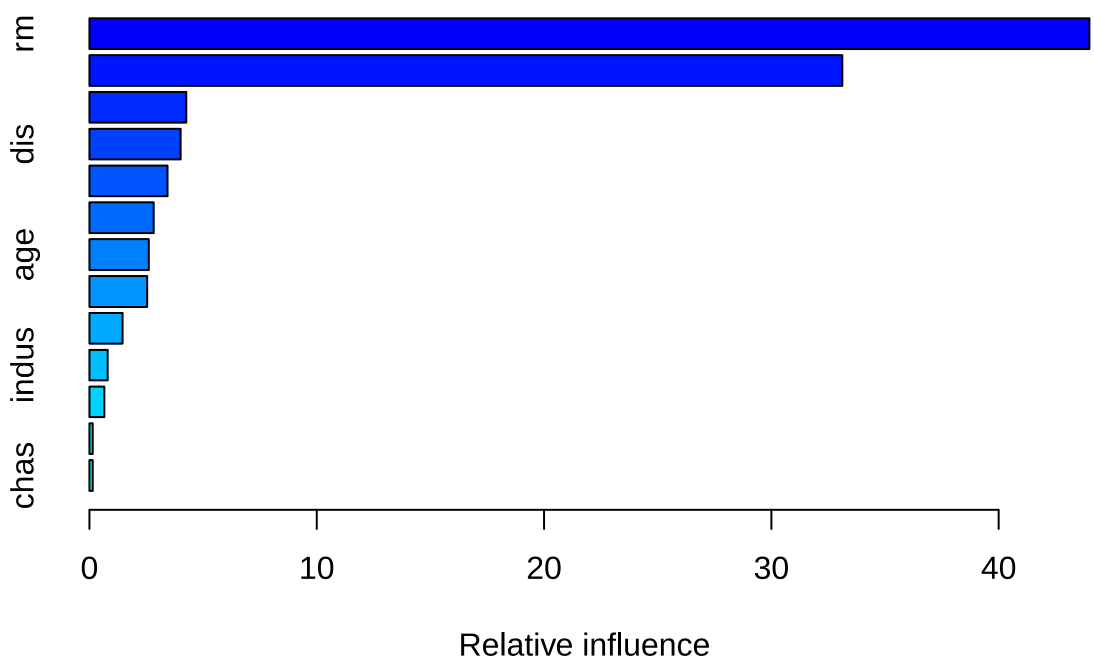
```
library(gbm)
```

```
## Loaded gbm 2.1.8.1
```

```
set.seed(1)
boost.boston <- gbm(medv ~., data = Boston[train,],
                    distribution = "gaussian", n.trees = 5000,
                    interaction.depth = 4)
```

The `summary()` function also provides a relative influence plot and also outputs the relative influence statistics.

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

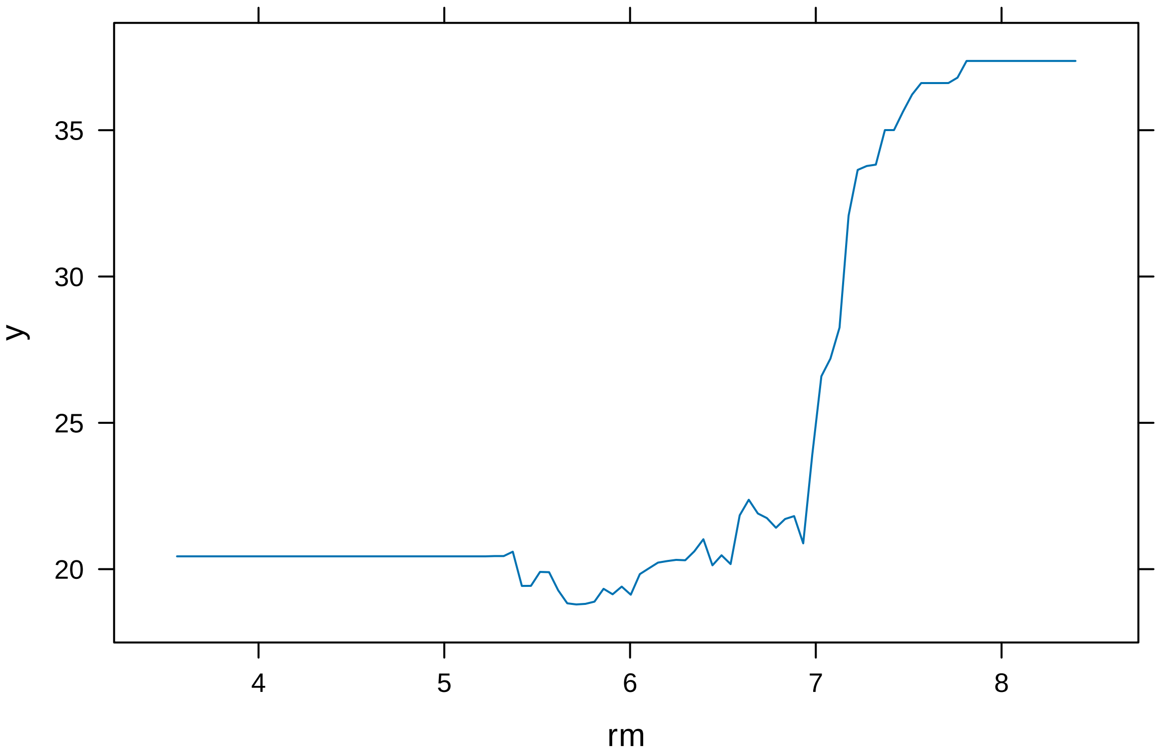



```
##             var    rel.inf
## rm           rm 43.9919329
## lstat     lstat 33.1216941
## crim       crim  4.2604167
## dis         dis  4.0111090
## nox         nox  3.4353017
## black     black  2.8267554
## age         age  2.6113938
## ptratio ptratio  2.5403035
```

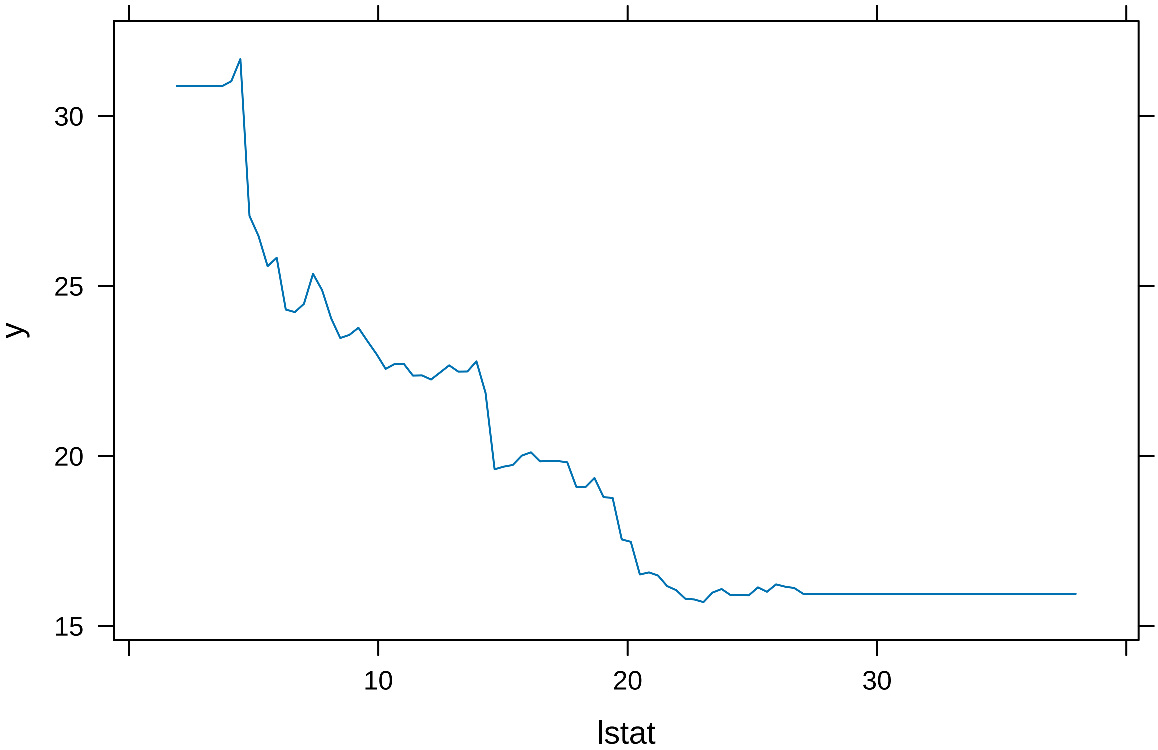
```
## tax          tax  1.4565654
## indus      indus  0.8008740
## rad          rad  0.6546400
## zn            zn  0.1446149
## chas        chas  0.1443986
```

We see that `lstat` and `rm` 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`.

```
par(mfrow = c(1,2))
plot(boost.boston, i = "rm")
```




```
plot(boost.boston, i = "lstat")
```



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

```
yhat.boost <- predict(boost.boston, newdata = Boston[-train,],
                      n.trees = 5000)
mean((yhat.boost - boston.test)^2)
```

```
## [1] 18.84709
```

The test MSE obtained is 11.8; 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 in Equation 8.10. The default value is 0.001, but this is easily modified. Here, we take $\lambda = 0.2$.

5. Problems

For these problems, we will work with the `Carseats` data again, this time in a regression setting, where the goal is to predict `Sales`. Make sure to drop the `High` variable in every model.

1. Train/Test Split

Split the `Carseats` data into a training and test set, using 30% of the data for the test set.

```
# Reset data set
data(Carseats, package = "ISLR")
```

```
# Split data
set.seed(123)
test_indices <- sample(1:nrow(Carseats), 0.3 * nrow(Carseats))
```

```r
# Create the training and test datasets
train.df <- Carseats[-test_indices, ]
test.df  <- Carseats[test_indices, ]
```

2. Regression Tree

Fit a regular regression tree on the training data using cross validation. Decide at what level to prune the tree and how you decided this. Report the test MSE for your tree and plot your final tree.

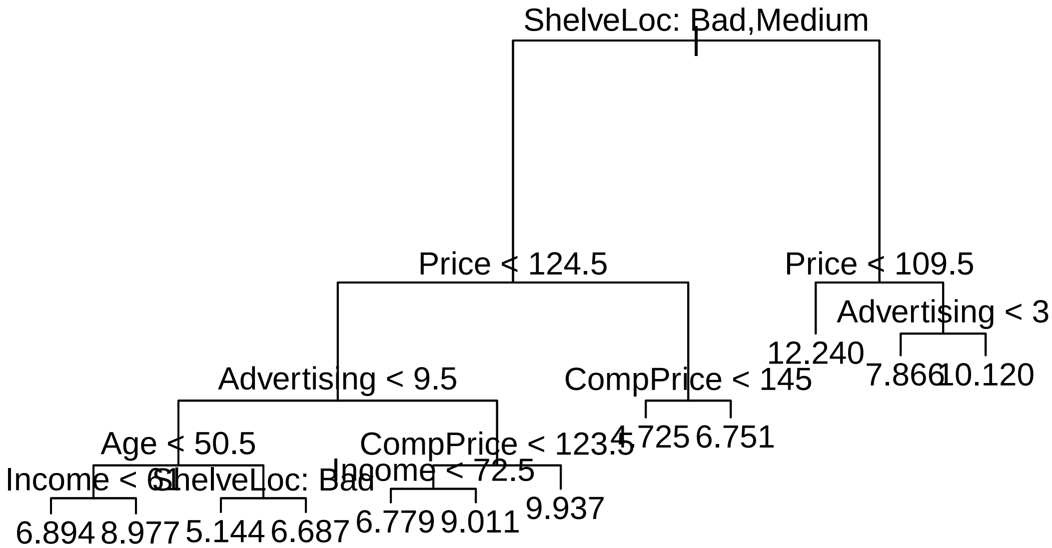
```r
# Fit a regression tree on the training data using cross-validation
cv.tree <- cv.tree(tree(Sales ~ ., data = train.df))

# Determine the optimal tree size
optimal_tree_size <- cv.tree$size[which.min(cv.tree$dev)]

print(optimal_tree_size)
```

```
## [1] 12
```

```r
# Fit a tree with the optimal size on the training data
reg_tree <- tree(Sales ~ .,
                 data = train.df,
                 control = tree.control(nobs = nrow(train.df),
                                        mincut = optimal_tree_size))

# Make predictions on the test data
test_preds <- predict(reg_tree, newdata = test.df)

# Calculate the test MSE
test_mse_reg <- mean((test_preds - test.df$Sales)^2)

# Print the test MSE
print(test_mse_reg)
```

```
## [1] 5.733881
```

```r
# Plot the final tree
plot(reg_tree)
text(reg_tree, pretty = 0)
```



3. Bagging

Perform bagging for the `Carseats` data with 25 trees. Report the MSE on the test set.

```r
set.seed(123)

# Perform bagging with 25 trees
bagging_model <- randomForest(Sales ~ ., data = train.df, ntree = 25)

# Make predictions on the test set
test_preds <- predict(bagging_model, newdata = test.df)

# Calculate the test MSE
test_mse_bag <- mean((test_preds - test.df$Sales)^2)

print(test_mse_bag)
```

```
## [1] 3.32436
```

4. Random Forest

Now, fit a random forest to the `Carseats` data. Report the variable importance as a plot and the MSE on the test set. Use $m = 3$.

```r
set.seed(123)

# Fit a Random Forest with m = 3
rf_model <- randomForest(Sales ~ ., data = train.df, mtry = 3)

# Make predictions on the test set
test_preds <- predict(rf_model, newdata = test.df)

# Calculate the test MSE
test_mse_rf <- mean((test_preds - test.df$Sales)^2)
```

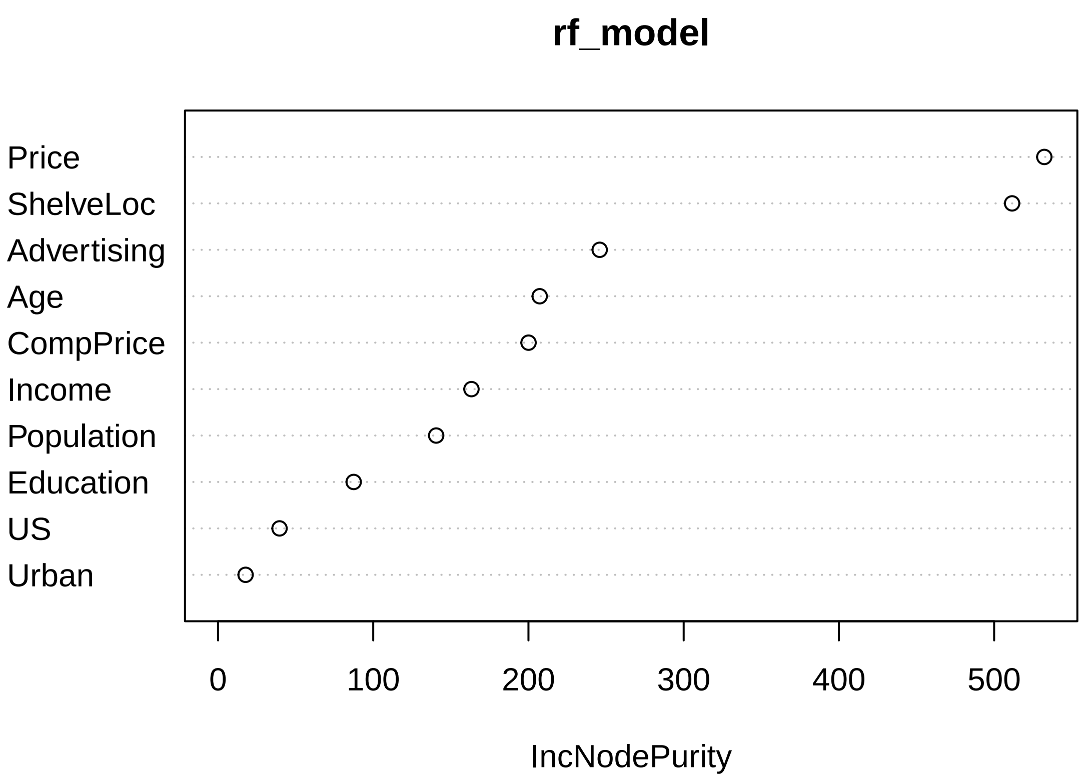
```r
print(test_mse_rf)
```

```
## [1] 3.15142
```

```r
importance(rf_model)
```

```
##             IncNodePurity
## CompPrice       200.04538
## Income          163.27178
## Advertising     245.89220
## Population      140.55079
## Price           532.32107
## ShelveLoc       511.57263
## Age             207.22474
## Education        87.38904
## Urban            17.75263
## US               39.61138
```

```r
# Plot variable importance
varImpPlot(rf_model)
```



5. Boosting

Finally, perform boosting on the `Carseats` data. Again, report the MSE on the test set. Use an interaction depth of 3.

```r
set.seed(123)

# Fit a boosted regression model
boosting_model <- gbm(Sales ~ ., data = train.df, distribution = "gaussian", interaction.depth = 3)

# Make predictions on the test set
test_preds <- predict(boosting_model, newdata = test.df)
```

```
## Using 100 trees...
```

```r
# Calculate the test MSE
test_mse_boost <- mean((test_preds - test.df$Sales)^2)

print(test_mse_boost)
```

```
## [1] 1.629282
```

6. Model Selection

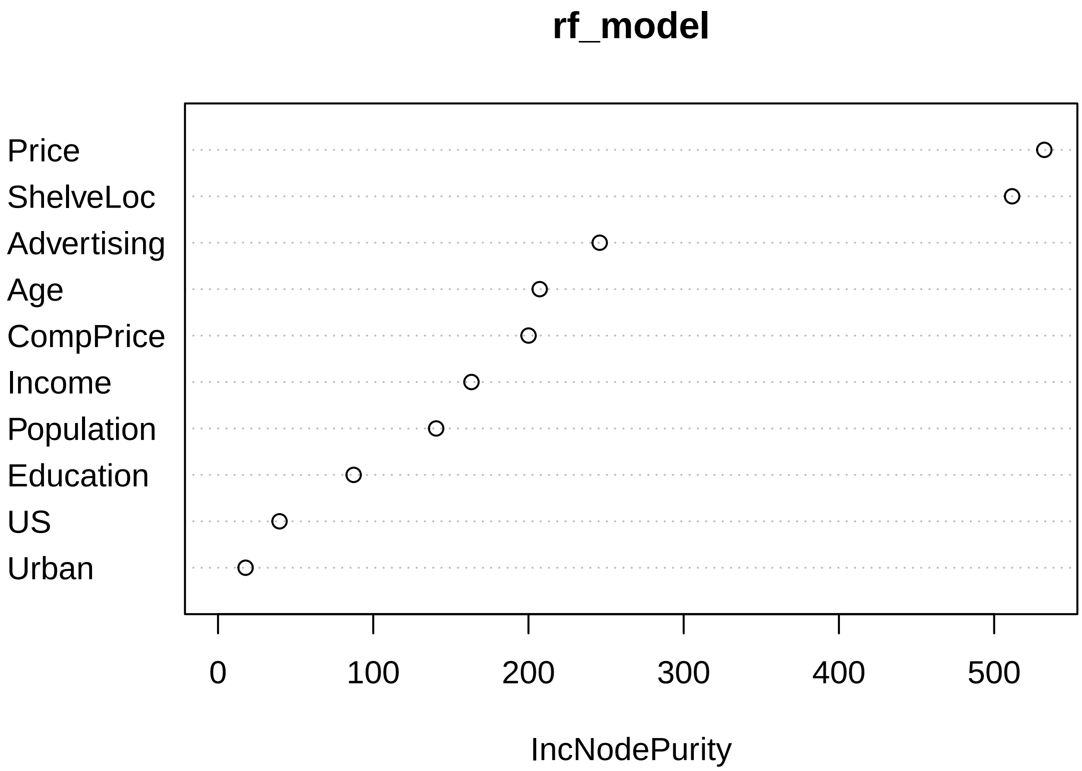
Make a table/dataframe summarizing the MSE results for each model considered above. Which model would you select and why? Which variables appear important to the trees? Does this make sense in the context of the problem?

```r
library(knitr)
library(kableExtra)

# Create a data frame to store the MSE results
model_names <- c("Regression Tree (k = 17)", "Bagging (25 trees)", "Random Forest (m = 3)", "Boosting (i
mse_values <- c(test_mse_reg, test_mse_bag, test_mse_rf, test_mse_boost)
results_df <- data.frame(Model = model_names, Test_MSE = mse_values)

# Print the results
kable(results_df)
```

Model	Test_MSE
Regression Tree (k = 17)	5.733880
Bagging (25 trees)	3.324360
Random Forest (m = 3)	3.151420
Boosting (interaction depth 3)	1.629282

Base on the table, I would choose the Boosting model (interaction depth 3) as it yields the lowest test MSE.

```r
# Plot variable importance
varImpPlot(rf_model)
```



According to this importance plot, we see that price and shelf location have the highest importance. This makes sense because price obviously affect the number of sales as it's the primary factor for most consumers. Shelf location is also important because the more visible the product is, the more likely the consumers will see it and thus purchase the product.