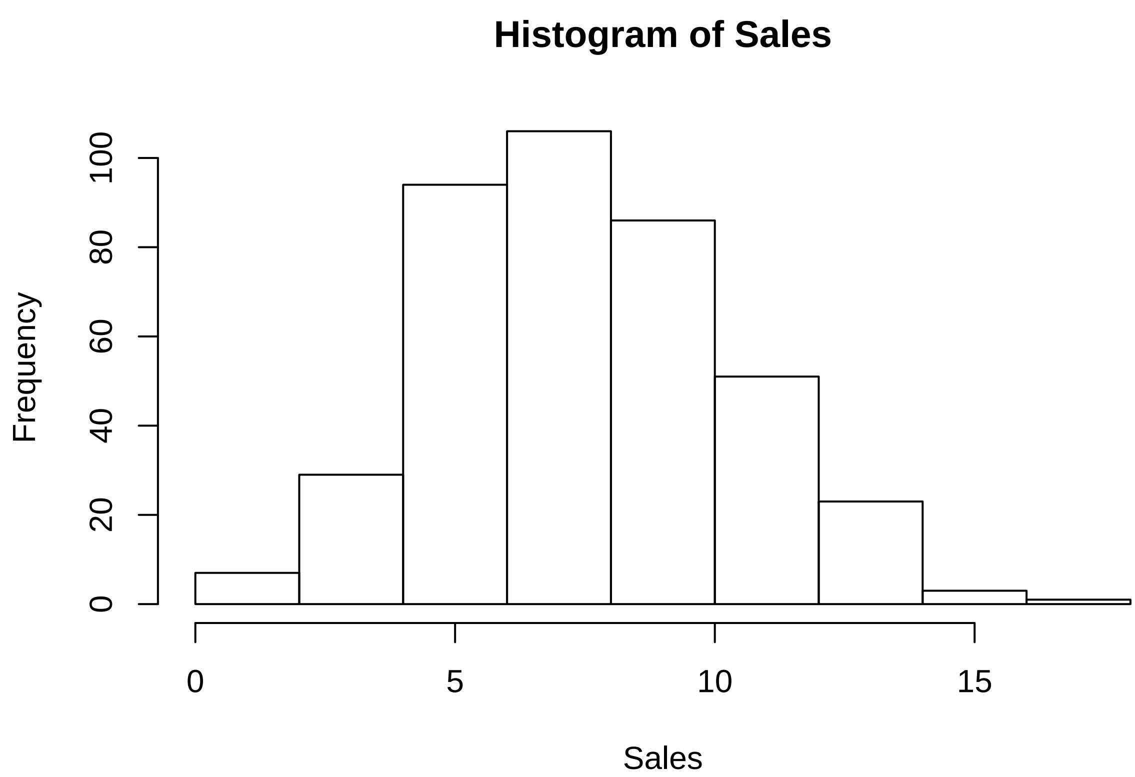
Decision Trees

We will have a look at the `Carseats` data using the `tree` package in R, as in the lab in the book. We create a binary response variable `High` (for high sales), and we include it in the same dataframe.

```
library(ISLR)
library(tree)
attach(Carseats)
hist(Sales)
```




```
High <- ifelse(Sales <= 8, "No", "Yes")
Carseats <- data.frame(Carseats, High)
```

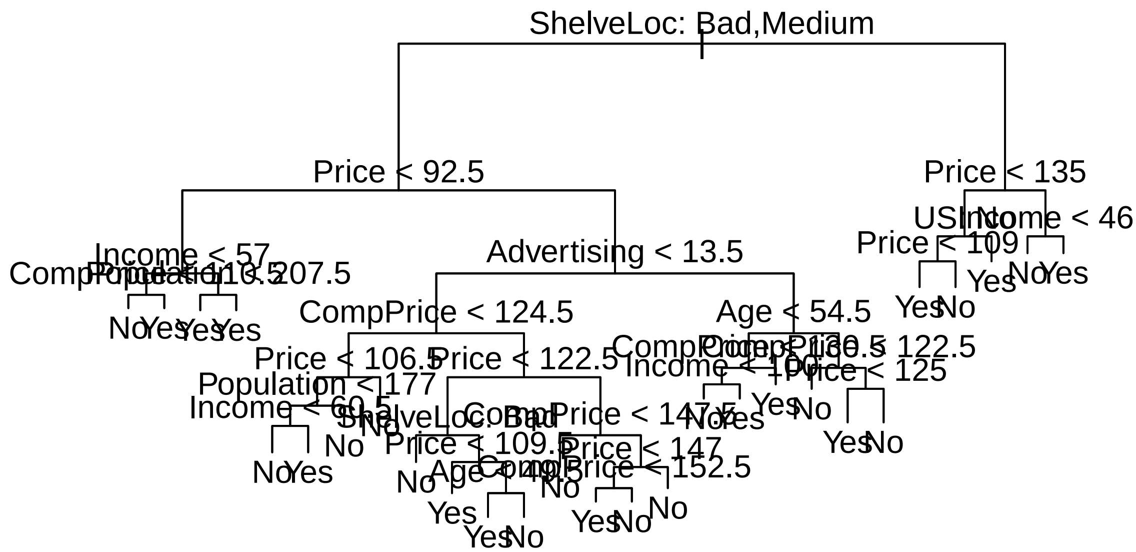
Now we fit a tree to these data, and summarize and plot it. Notice that we have to *exclude* `Sales` from the right-hand side of the formula, because the response is derived from it.

```
tree.carseats <- tree(High~.-Sales, data = Carseats)

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
```

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



For a detailed summary of the tree, print it:

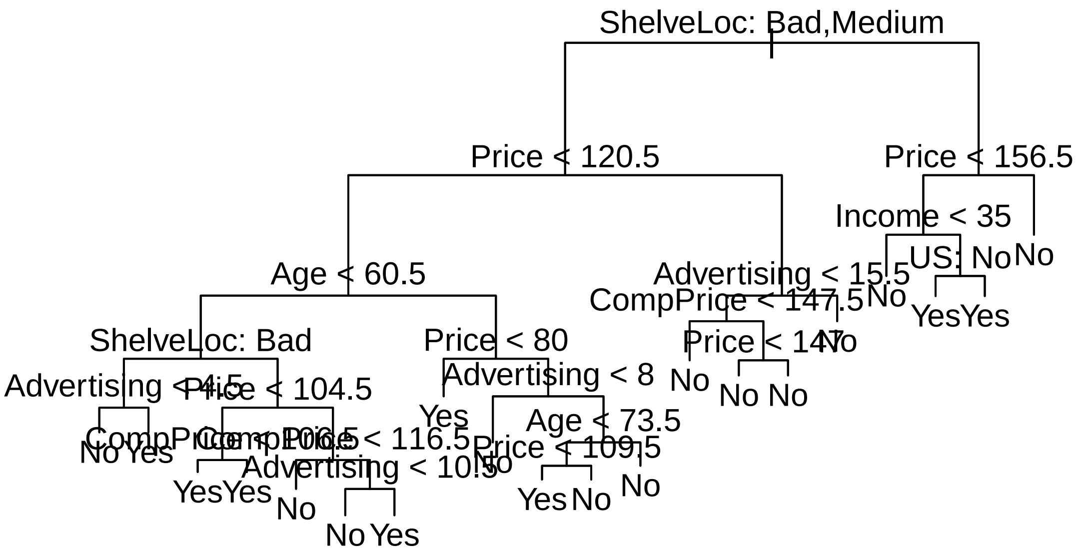
```
tree.carseats
```

```
## 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 ) *
```

Lets create a training and test set (250,150) split of the 400 observations, grow the tree on the training set, and evaluate its performance on the test set.

```r
set.seed(1011)
train <- sample(1:nrow(Carseats), 250)
tree.carseats <- tree(High ~ .-Sales, Carseats,subset = train)
plot(tree.carseats);text(tree.carseats, pretty=0)
```




```
tree.pred <- predict(tree.carseats, Carseats[-train,], type="class")
with(Carseats[-train,], table(tree.pred,High))
```

```
##           High
## tree.pred No  Yes
##       No   72  27
##       Yes  18  33
```

```
(72+33)/150
```

```
## [1] 0.7
```

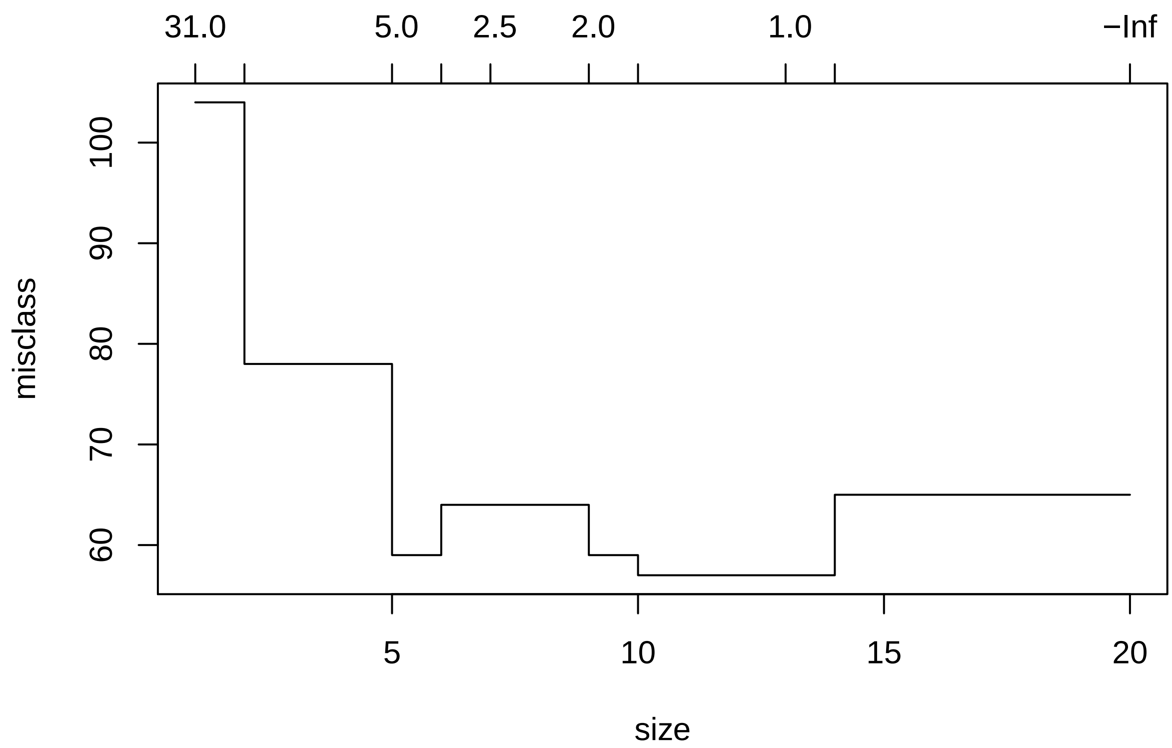
This tree was grown to full depth, and might be too variable. We now use CV to prune it.

```
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
cv.carseats
```

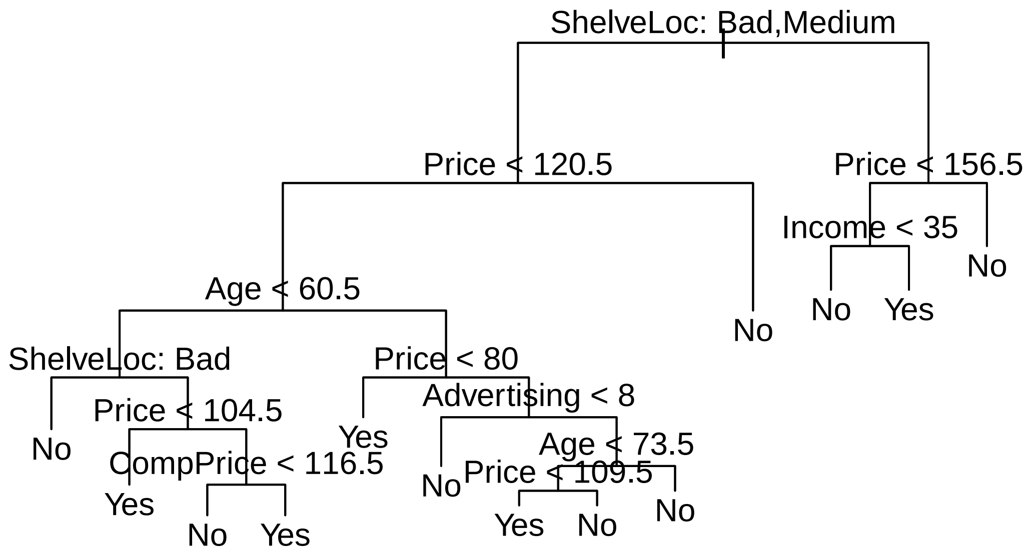
```
## $size
##  [1] 20 14 13 10  9  7  6  5  2  1
##
## $dev
##  [1]  65  65  57  57  59  64  64  59  78 104
##
## $k
##  [1]      -Inf  0.000000  1.000000  1.333333  2.000000  2.500000  4.000000
##  [8]  5.000000  9.000000 31.000000
##
## $method
## [1] "misclass"
```

```
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```r
plot(cv.carseats)
```




```r
prune.carseats <- prune.misclass(tree.carseats, best=13)
plot(prune.carseats);text(prune.carseats, pretty=0)
```



Now lets evaluate this pruned tree on the test data.

```
tree.pred <- predict(prune.carseats, Carseats[-train,], type="class")
with(Carseats[-train, ], table(tree.pred, High))
```

```
##           High
## tree.pred No Yes
##       No  72  28
##       Yes 18  32
```

```
(72+32)/150
```

```
## [1] 0.6933333
```

It has done about the same as our original tree. So pruning did not hurt us wrt misclassification errors, and gave us a simpler tree.

Random Forests and Boosting

These methods use trees as building blocks to build more complex models. Here we will use the Boston housing data to explore random forests and boosting. These data are in the `MASS` package. It gives housing values and other statistics in each of 506 suburbs of Boston based on a 1970 census.

Random Forests

Random forests build lots of bushy trees, and then average them to reduce the variance.

```r
library(randomForest)
```

```
## randomForest 4.6-12
## Type rfNews() to see new features/changes/bug fixes.
```

```r
library(MASS)

?Boston
```

```
## starting httpd help server ...
##  done
```

```r
dim(Boston)
```

```
## [1] 506  14
```

```r
set.seed(101)
train <- sample(1:nrow(Boston), 300)
```

Lets fit a random forest and see how well it performs. We will use the response `medv`, the median housing value (in $1K dollars)

```r
rf.boston <- randomForest(medv ~ ., data = Boston, subset = train)
rf.boston
```

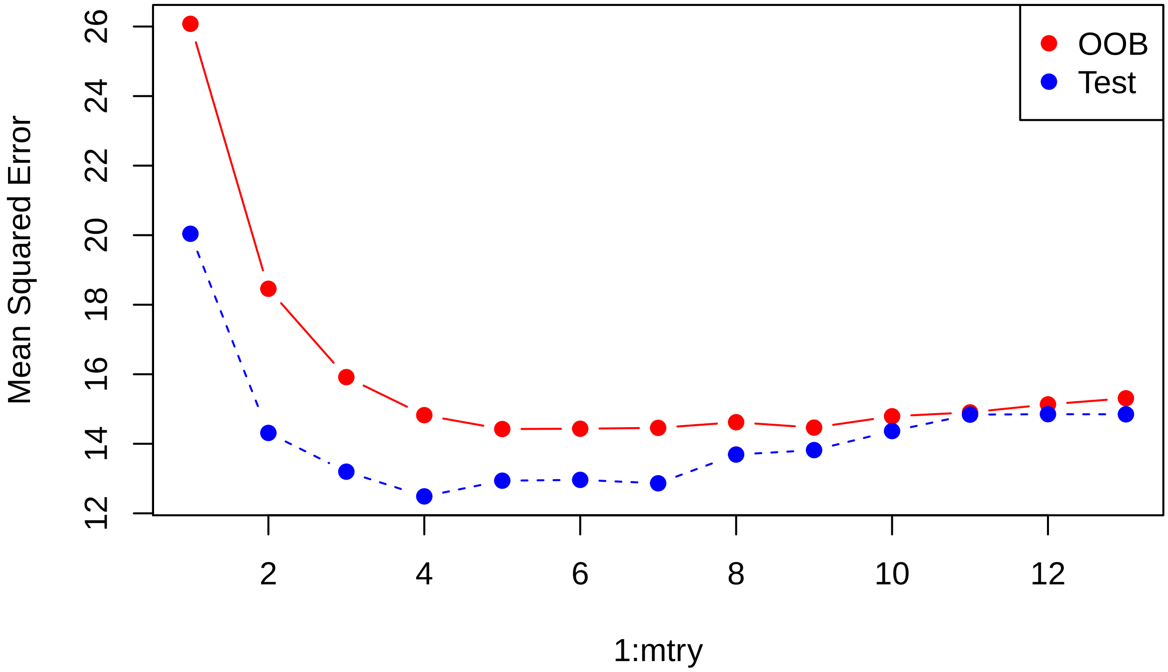
```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, subset = train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 12.34243
##                     % Var explained: 85.09
```

The MSR and % variance explained are based on OOB or *out-of-bag* estimates, a very clever device in random forests to get honest error estimates. The model reports that `mtry=4`, which is the number of variables randomly chosen at each split. Since $p = 13$ here, we could try all 13 possible values of `mtry`. We will do so, record the results, and make a plot.

```r
oob.err <- double(13)
test.err <- double(13)

for(mtry in 1:13){
                   fit <- randomForest(medv ~., data=Boston, subset=train, mtry=mtry, ntree=400)
         oob.err[mtry] <- fit$mse[400]
                  pred <- predict(fit, Boston[-train,])
        test.err[mtry] <- with(Boston[-train,], mean((medv-pred)^2))
        cat(mtry," ")
}
```

```
## 1  2  3  4  5  6  7  8  9  10  11  12  13
```

```r
matplot(1:mtry, cbind(test.err,oob.err), pch=19, col=c("red","blue"), type="b", ylab="Mean Squared Erro
legend("topright", legend=c("OOB","Test"), pch=19, col=c("red","blue"))
```



Not too difficult! Although the test-error curve drops below the OOB curve, these are estimates based on data, and so have their own standard errors (which are typically quite large). Notice that the points at the end with `mtry=13` correspond to bagging.

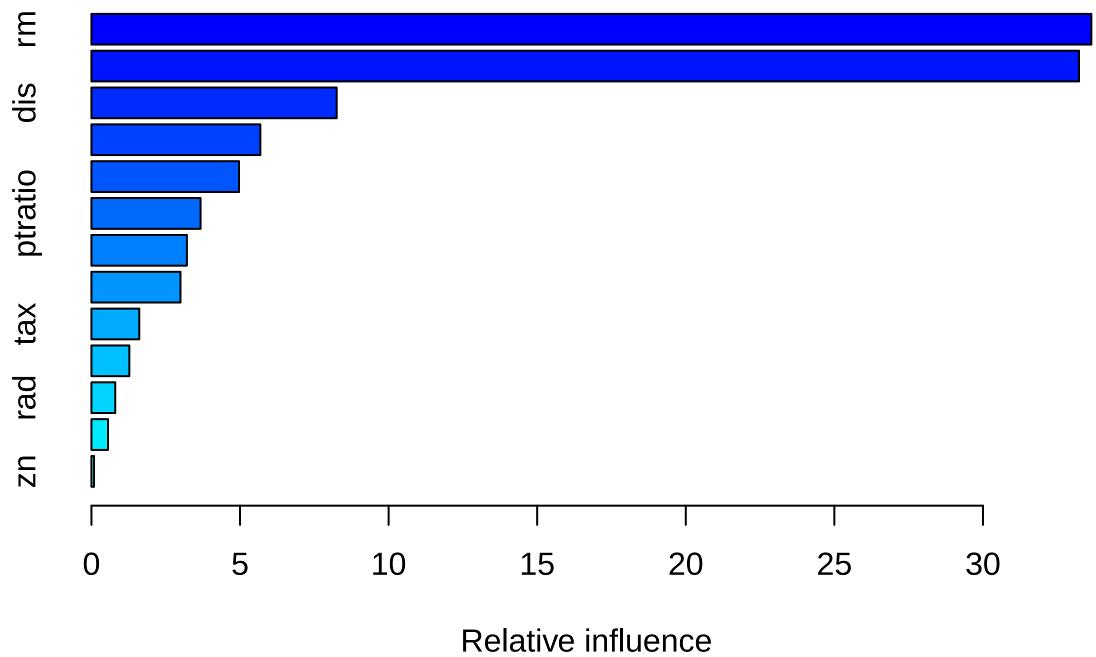
Boosting

Boosting builds lots of smaller trees. Unlike random forests, each new tree in boosting tries to patch up the deficiencies of the current ensemble.

```
library(gbm)
```

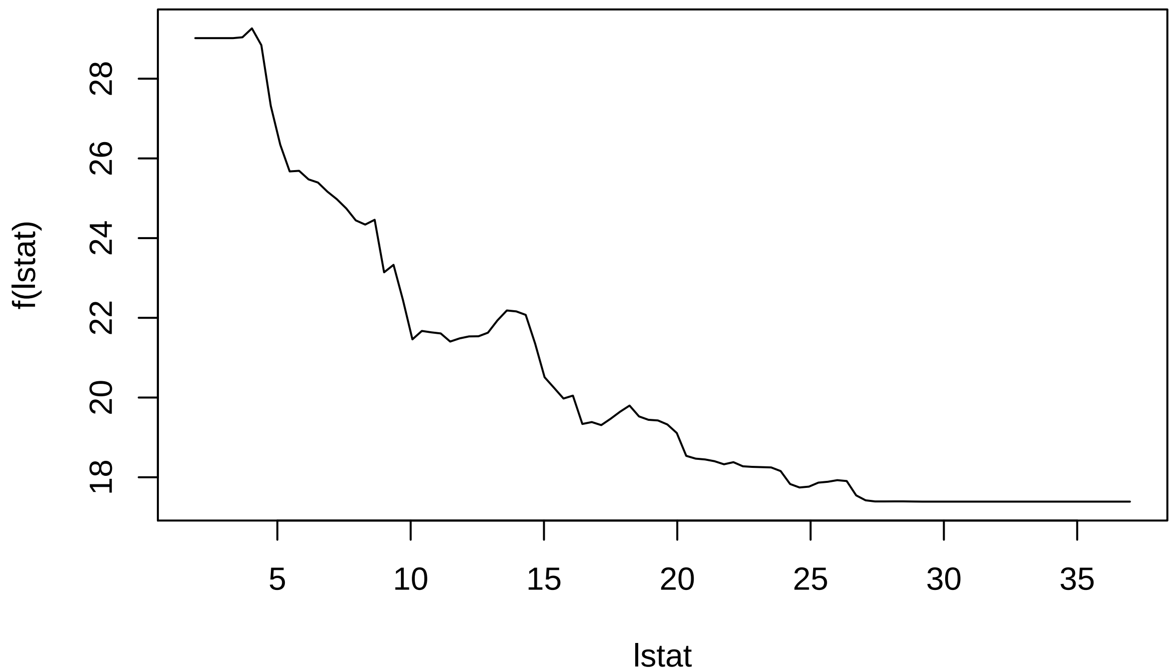
```
## Loading required package: survival

## Loading required package: lattice

## Loading required package: splines

## Loading required package: parallel

## Loaded gbm 2.1.3
```

```
boost.boston <- gbm(medv ~., data = Boston[train, ], distribution="gaussian", n.trees=10000, shrinkage=0

summary(boost.boston)
```

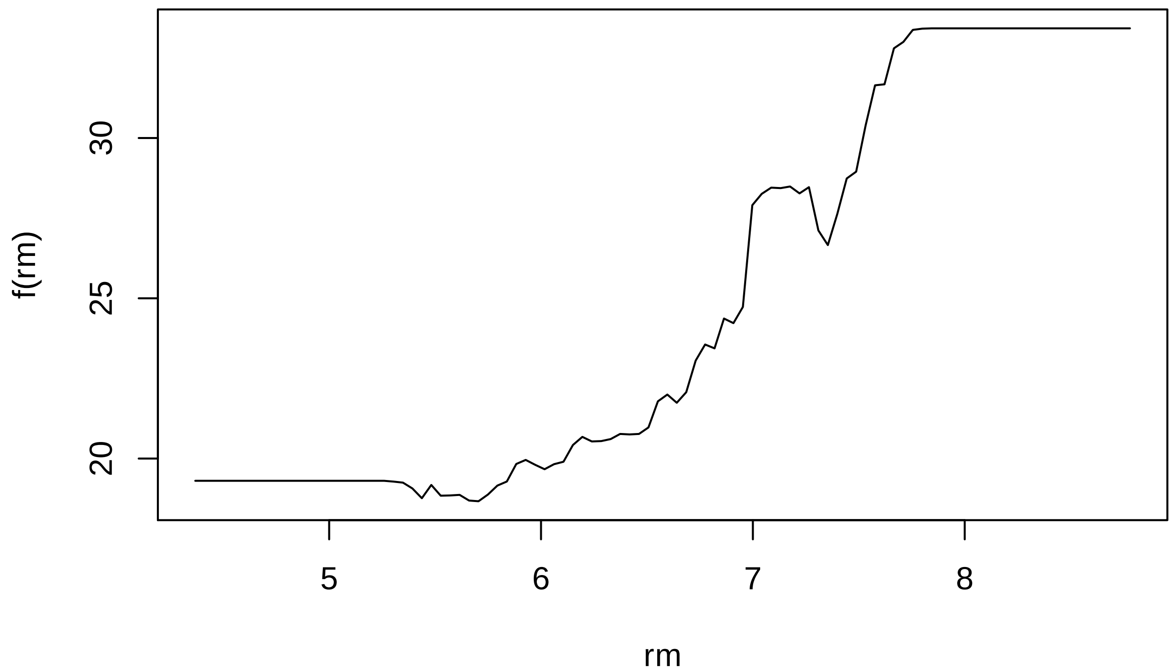



```
##              var     rel.inf
## rm            rm 33.64864169
## lstat      lstat 33.23204546
## dis          dis  8.25135162
## crim        crim  5.68521520
## nox          nox  4.96815276
## ptratio ptratio  3.67173422
## black      black  3.21069543
## age          age  2.99661215
## tax          tax  1.61028754
## chas        chas  1.27643461
## rad          rad  0.80176699
## indus      indus  0.55905474
## zn            zn  0.08800759
```

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




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

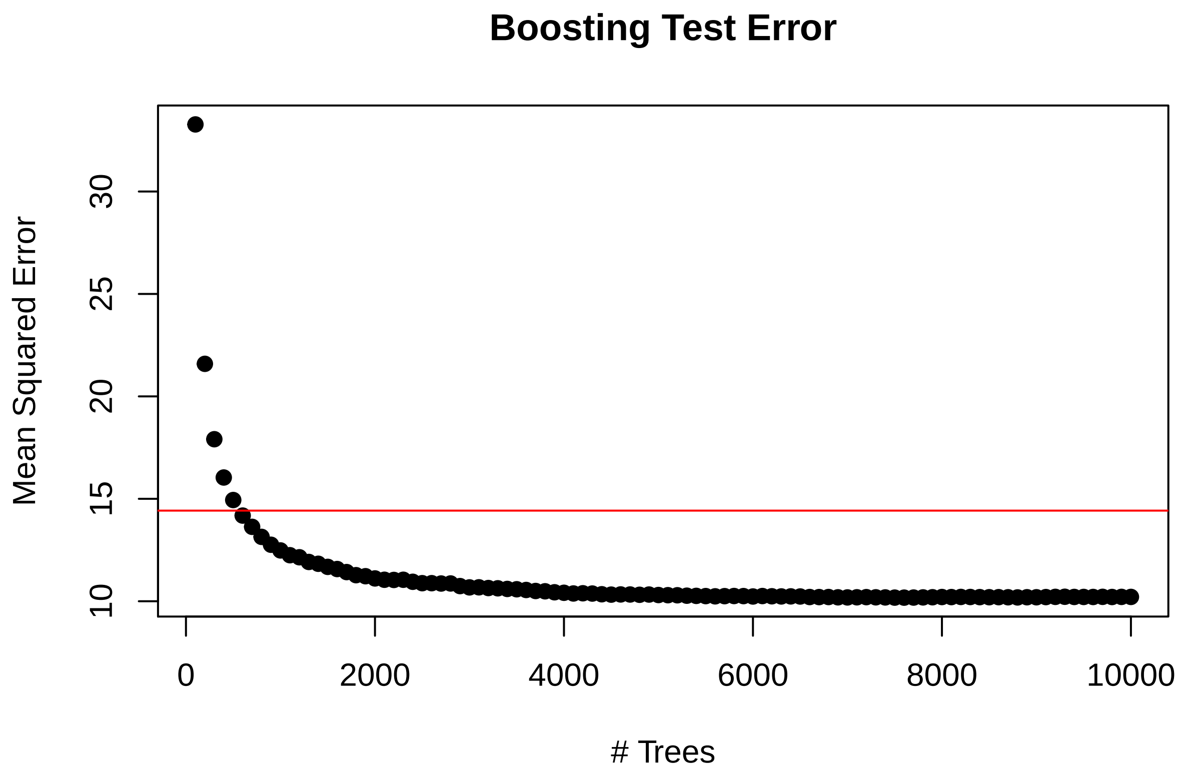


Lets make a prediction on the test set. With boosting, the number of trees is a tuning parameter, and if we have too many we can overfit. So we should use cross-validation to select the number of trees. We will leave this as an exercise. Instead, we will compute the test error as a function of the number of trees, and make a plot.

```
n.trees <- seq(from=100, to=10000, by=100)
predmat <- predict(boost.boston, newdata = Boston[-train, ], n.trees=n.trees)
dim(predmat)
```

```
## [1] 206 100
```

```
berr <- with(Boston[-train, ], apply((predmat-medv)^2,2,mean))
plot(n.trees, berr, pch=19, ylab="Mean Squared Error", xlab="# Trees", main="Boosting Test Error")
abline(h=min(test.err), col="red")
```


Boosting Test Error
Mean Squared Error
10
15
20
25
30
0
2000
4000
6000
8000
10000
Trees