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 higg sales), and we include it in the same dataframe.

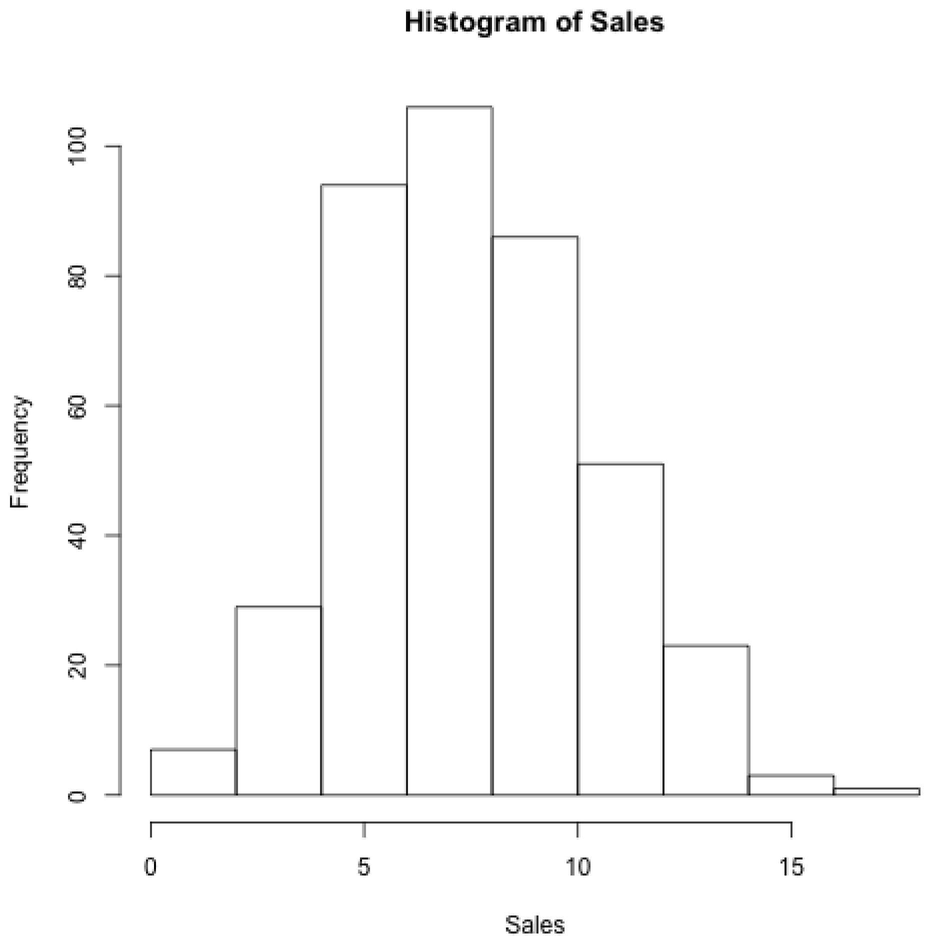
```
require(ISLR)
```

```
## Loading required package: ISLR
```

```
require(tree)
```

```
## Loading required package: tree
```

```
attach(Carseats)
hist(Sales)
```



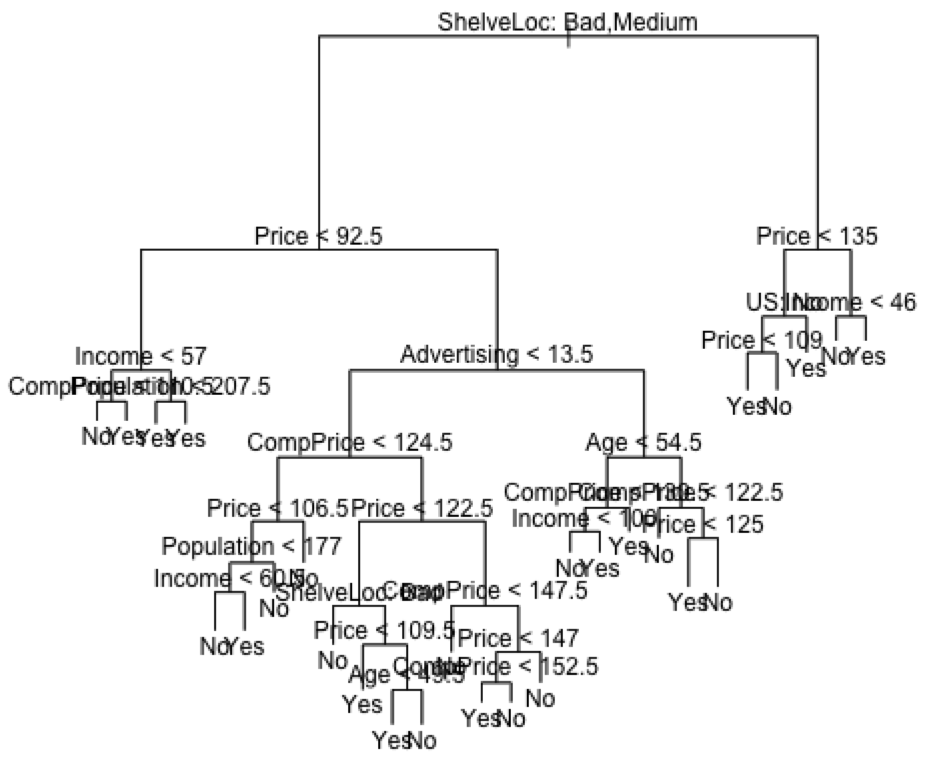
```
High = ifelse(Sales <= 8, "No", "Yes")
Carseats = data.frame(Carseats, High)
```

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:  0.458 = 171 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

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



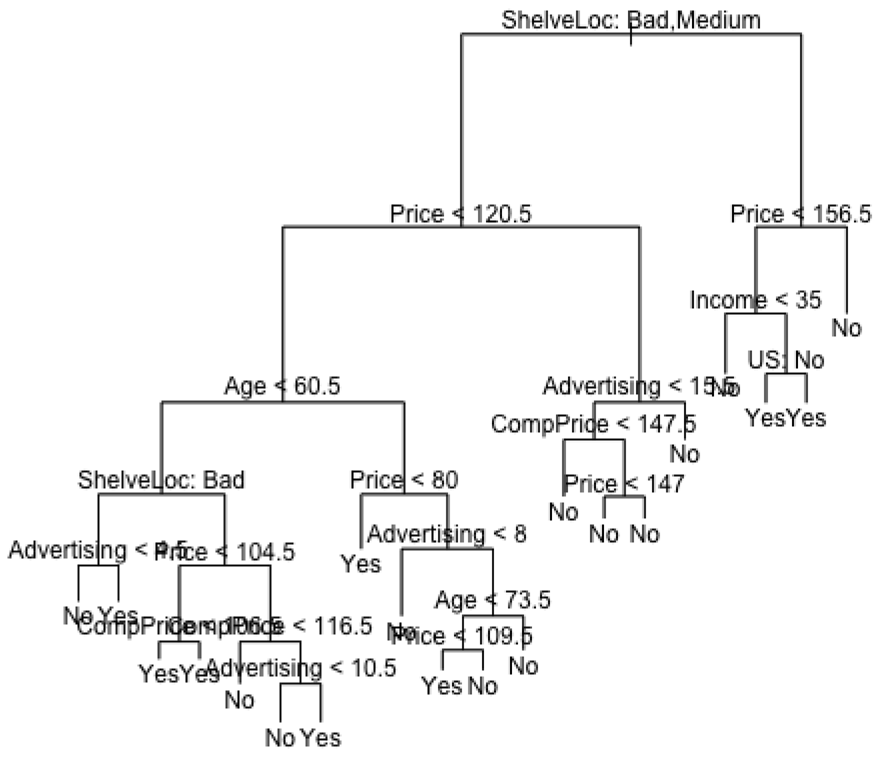
For a detailed summary of the tree, print it:

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 500 No ( 0.59 0.41 )
##     2) ShelveLoc: Bad,Medium 315 400 No ( 0.69 0.31 )
##       4) Price < 92.5 46  60 Yes ( 0.30 0.70 )
##         8) Income < 57 10  10 No ( 0.70 0.30 )
##          16) CompPrice < 110.5 5   0 No ( 1.00 0.00 ) *
##          17) CompPrice > 110.5 5   7 Yes ( 0.40 0.60 ) *
##         9) Income > 57 36  40 Yes ( 0.19 0.81 )
##          18) Population < 207.5 16  20 Yes ( 0.38 0.62 ) *
##          19) Population > 207.5 20   8 Yes ( 0.05 0.95 ) *
##       5) Price > 92.5 269 300 No ( 0.75 0.25 )
##        10) Advertising < 13.5 224 200 No ( 0.82 0.18 )
##          20) CompPrice < 124.5 96  40 No ( 0.94 0.06 )
##            40) Price < 106.5 38  30 No ( 0.84 0.16 )
##              80) Population < 177 12  20 No ( 0.58 0.42 )
##               160) Income < 60.5 6   0 No ( 1.00 0.00 ) *
##               161) Income > 60.5 6   5 Yes ( 0.17 0.83 ) *
##              81) Population > 177 26   8 No ( 0.96 0.04 ) *
##            41) Price > 106.5 58   0 No ( 1.00 0.00 ) *
##          21) CompPrice > 124.5 128 200 No ( 0.73 0.27 )
##            42) Price < 122.5 51  70 Yes ( 0.49 0.51 )
##              84) ShelveLoc: Bad 11   7 No ( 0.91 0.09 ) *
##              85) ShelveLoc: Medium 40  50 Yes ( 0.38 0.62 )
##               170) Price < 109.5 16   7 Yes ( 0.06 0.94 ) *
##               171) Price > 109.5 24  30 No ( 0.58 0.42 )
##                 342) Age < 49.5 13  20 Yes ( 0.31 0.69 ) *
##                 343) Age > 49.5 11   7 No ( 0.91 0.09 ) *
##            43) Price > 122.5 77  60 No ( 0.88 0.12 )
##              86) CompPrice < 147.5 58  20 No ( 0.97 0.03 ) *
##              87) CompPrice > 147.5 19  30 No ( 0.63 0.37 )
##               174) Price < 147 12  20 Yes ( 0.42 0.58 )
##                 348) CompPrice < 152.5 7   6 Yes ( 0.14 0.86 )
*
##                 349) CompPrice > 152.5 5   5 No ( 0.80 0.20 ) *
##               175) Price > 147 7   0 No ( 1.00 0.00 ) *
##        11) Advertising > 13.5 45  60 Yes ( 0.44 0.56 )
##          22) Age < 54.5 25  30 Yes ( 0.20 0.80 )
##            44) CompPrice < 130.5 14  20 Yes ( 0.36 0.64 )
##              88) Income < 100 9  10 No ( 0.56 0.44 ) *
##              89) Income > 100 5   0 Yes ( 0.00 1.00 ) *
##            45) CompPrice > 130.5 11   0 Yes ( 0.00 1.00 ) *
##          23) Age > 54.5 20  20 No ( 0.75 0.25 )
    ##             46) CompPrice < 122.5 10   0 No ( 1.00 0.00 ) *
    ##             47) CompPrice > 122.5 10  10 No ( 0.50 0.50 )
    ##               94) Price < 125 5   0 Yes ( 0.00 1.00 ) *
    ##               95) Price > 125 5   0 No ( 1.00 0.00 ) *
    ##      3) ShelveLoc: Good 85  90 Yes ( 0.22 0.78 )
    ##        6) Price < 135 68  50 Yes ( 0.12 0.88 )
    ##         12) US: No 17  20 Yes ( 0.35 0.65 )
    ##           24) Price < 109 8   0 Yes ( 0.00 1.00 ) *
    ##           25) Price > 109 9  10 No ( 0.67 0.33 ) *
    ##         13) US: Yes 51  20 Yes ( 0.04 0.96 ) *
    ##        7) Price > 135 17  20 No ( 0.65 0.35 )
    ##         14) Income < 46 6   0 No ( 1.00 0.00 ) *
    ##         15) Income > 46 11  20 Yes ( 0.45 0.55 ) *
```

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

```
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
```

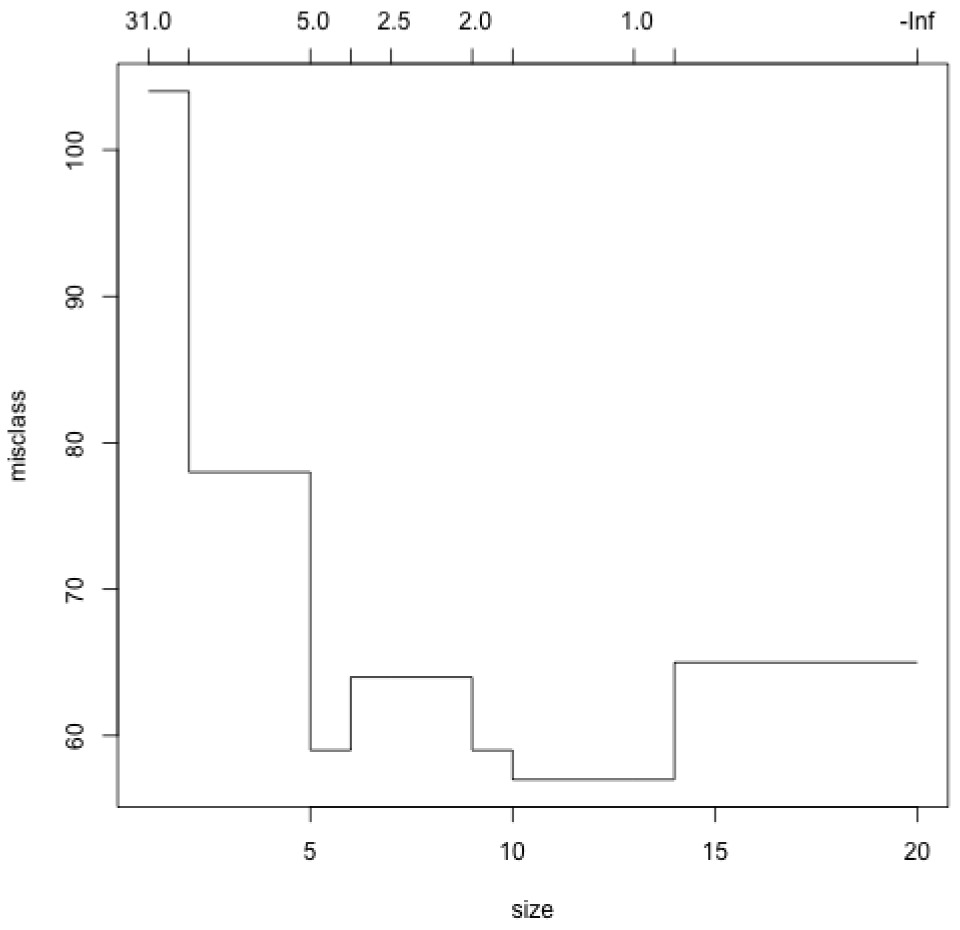
```
## [1] 0.7
```

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

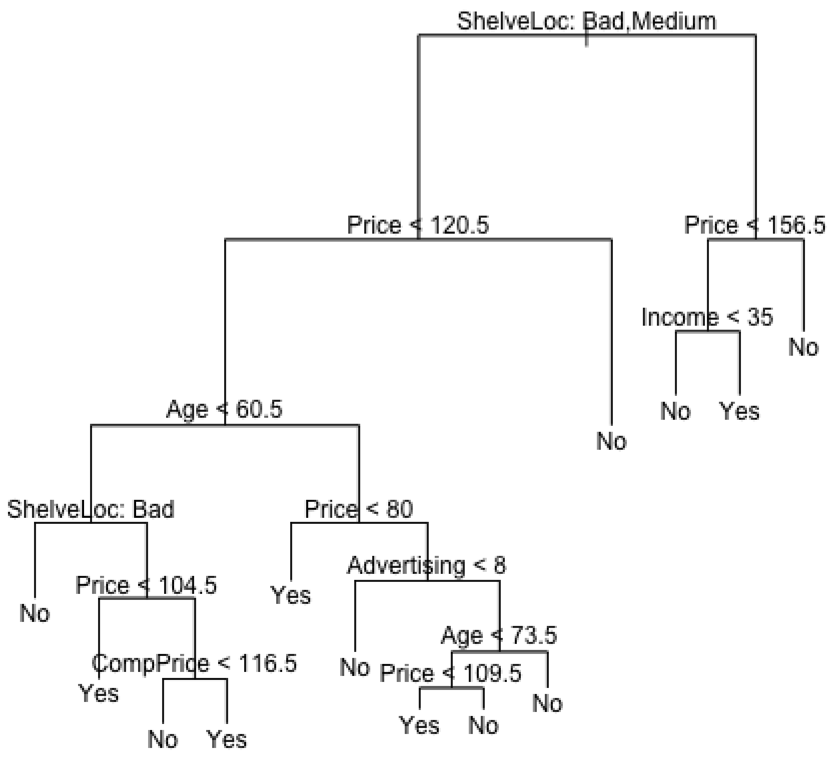
```
cv.carseats = cv.tree(tree.carseats, FUN = prune.misclass)
cv.carseats
```

```
## $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  0.000  1.000  1.333  2.000  2.500  4.000  5.000
9.000 31.000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

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



```
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] 0.6933
```

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.

```
require(randomForest)
```

```
## Loading required package: randomForest randomForest 4.6-7 Type
rfNews() to
## see new features/changes/bug fixes.
```

```
require(MASS)
```

```
## Loading required package: MASS
##
## Attaching package: 'MASS'
##
## The following object is masked from 'package:hastie':
##
## enlist
```

```
set.seed(101)
dim(Boston)
```

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

```
train = sample(1:nrow(Boston), 300)
`?`(Boston)
```

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

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

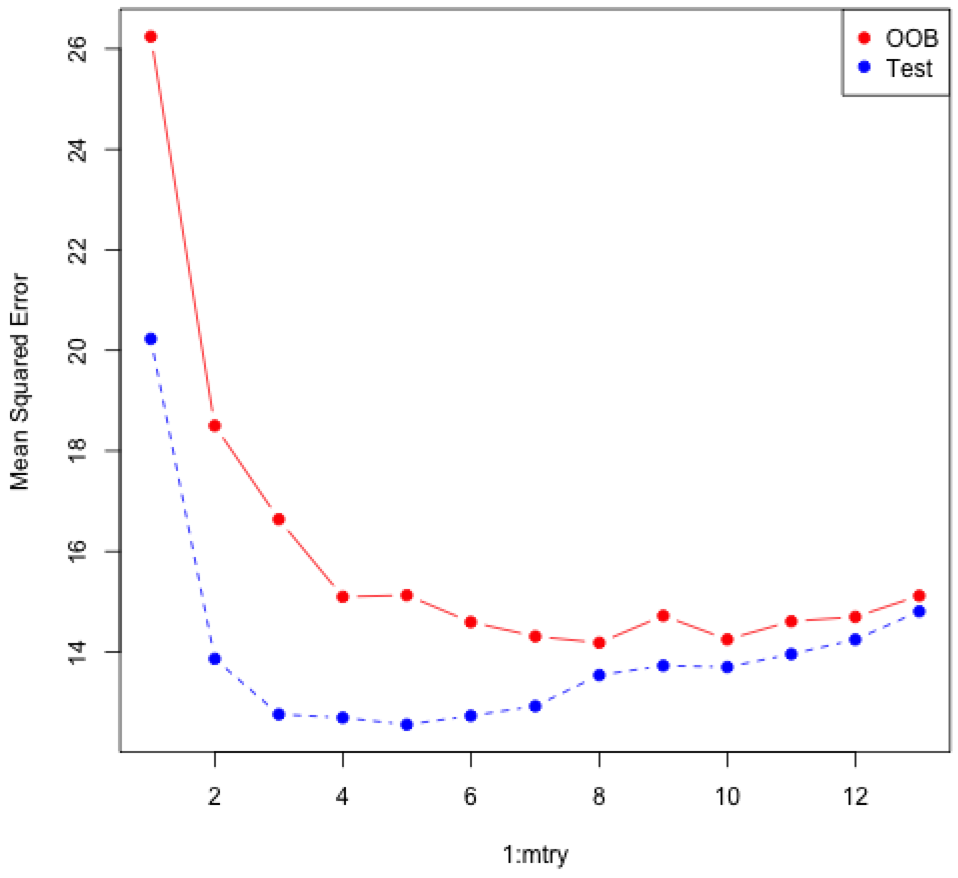
```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, subset =
train)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##           Mean of squared residuals: 12.51
##                     % Var explained: 84.89
```

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.

```
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
```

```
matplot(1:mtry, cbind(test.err, oob.err), pch = 19, col = c("red",
"blue"),
    type = "b", ylab = "Mean Squared Error")
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.

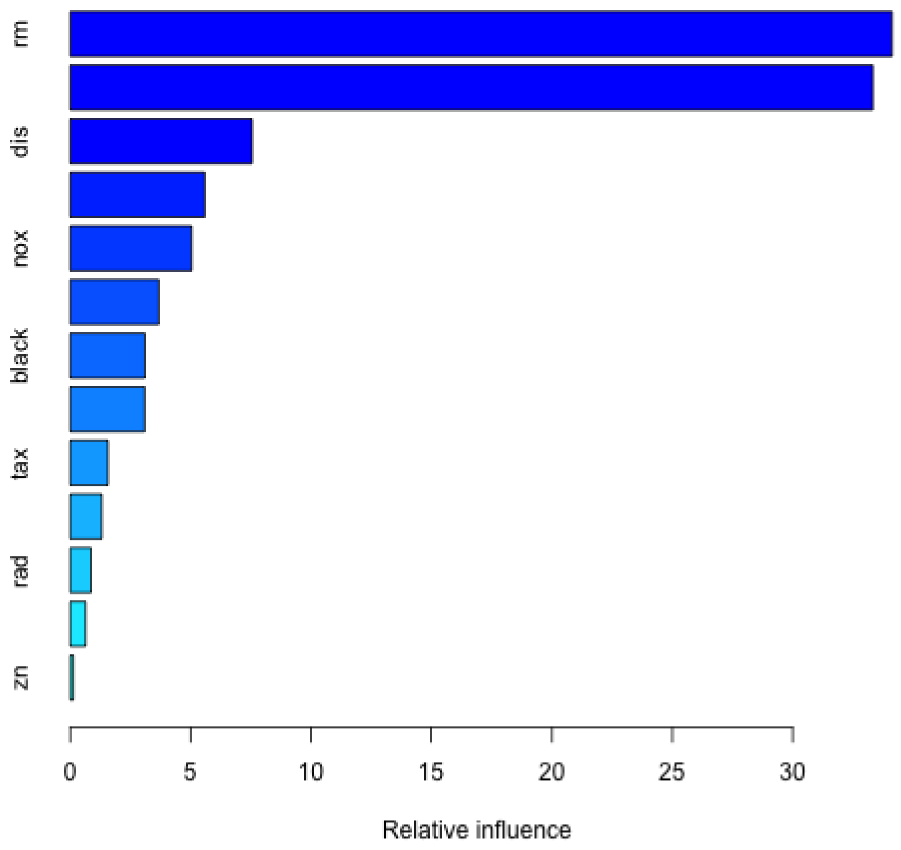
Boosting

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

```
require(gbm)
```

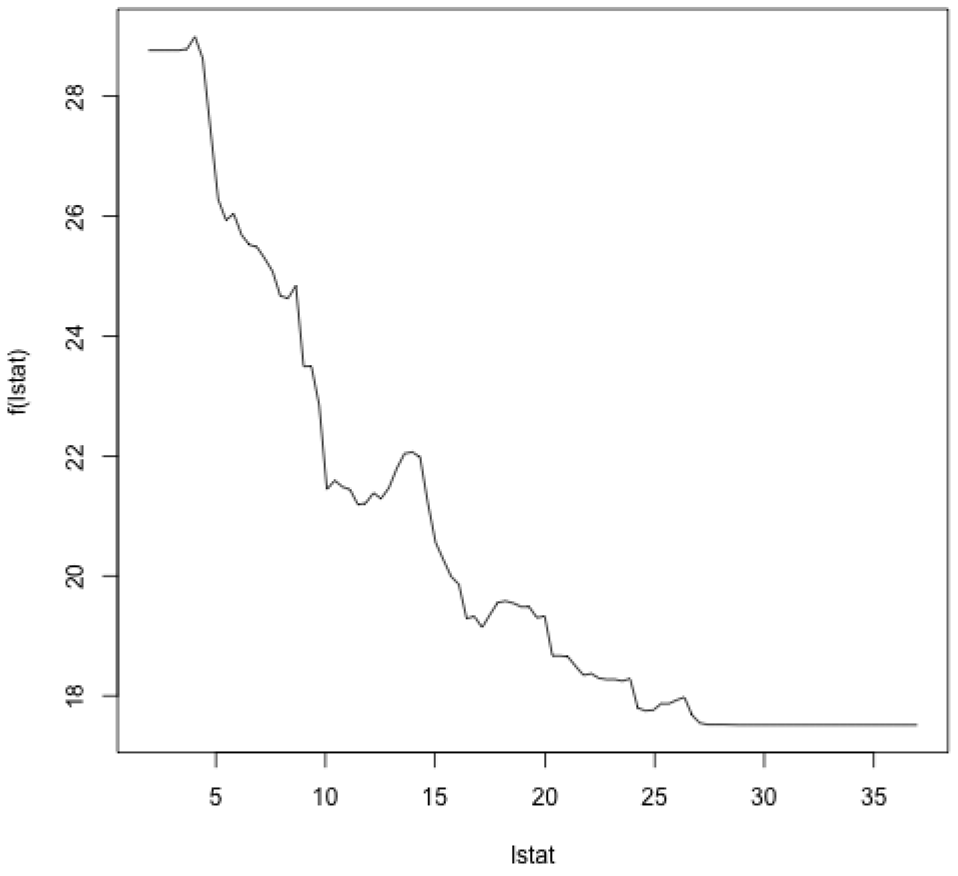
```
## Loading required package: gbm Loading required package:
survival Loading
## required package: splines Loading required package: lattice
Loading
## required package: parallel Loaded gbm 2.1
```

```
boost.boston = gbm(medv ~ ., data = Boston[train, ], distribution
= "gaussian",
    n.trees = 10000, shrinkage = 0.01, interaction.depth = 4)
summary(boost.boston)
```

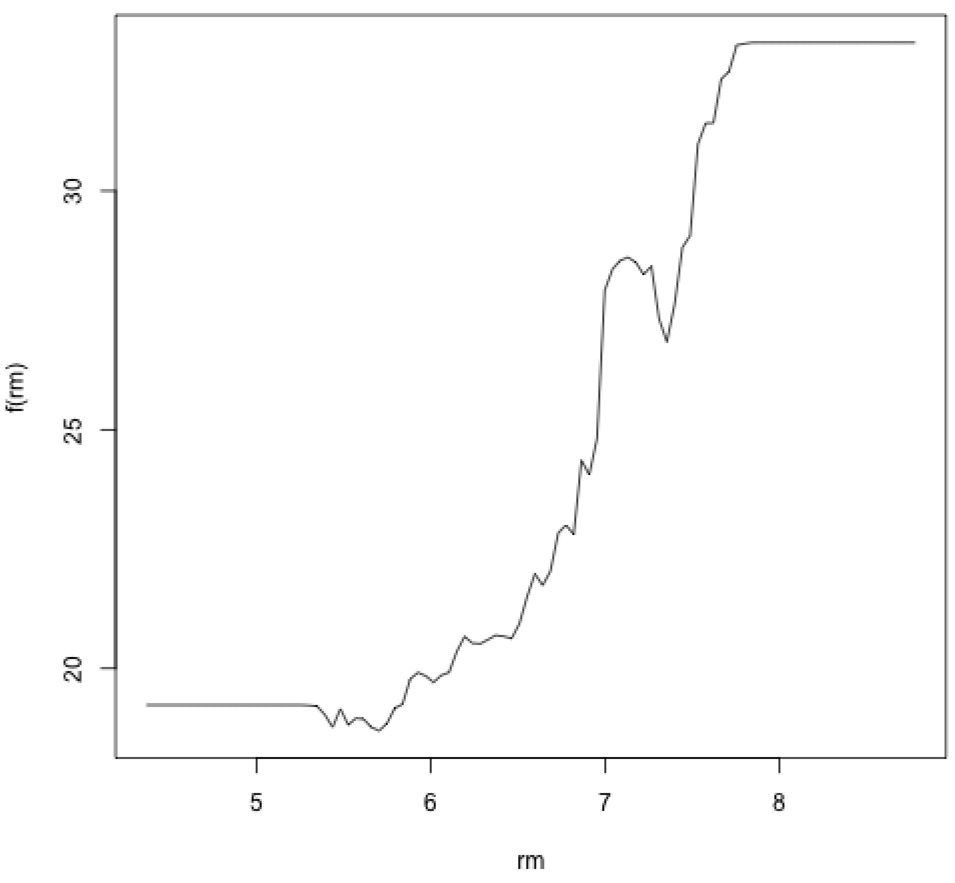


```
##              var rel.inf
## rm            rm 34.1272
## lstat      lstat 33.3280
## dis          dis  7.5532
## crim        crim  5.5853
## nox          nox  5.0416
## ptratio ptratio  3.6750
## black      black  3.1000
## age          age  3.0906
## tax          tax  1.5641
## chas        chas  1.3074
## rad          rad  0.8641
## indus      indus  0.6365
## zn            zn  0.1269
```

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



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

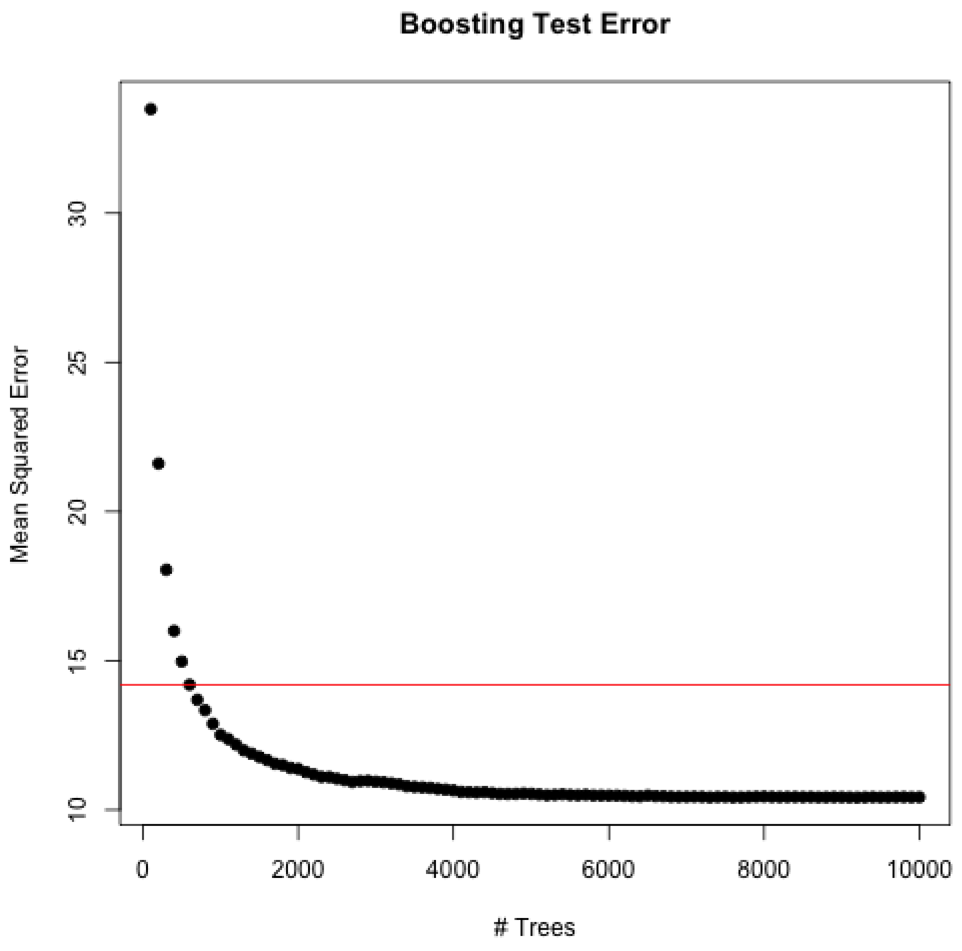


Lets make a prediction on the test set. With boosting, the number of trees is a tuning parameter, and if we have too many we can overfit. So we should use cross-validation to select the number of trees. We will leave this as an exercise. Instead, we will compute the test error as a function of the number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