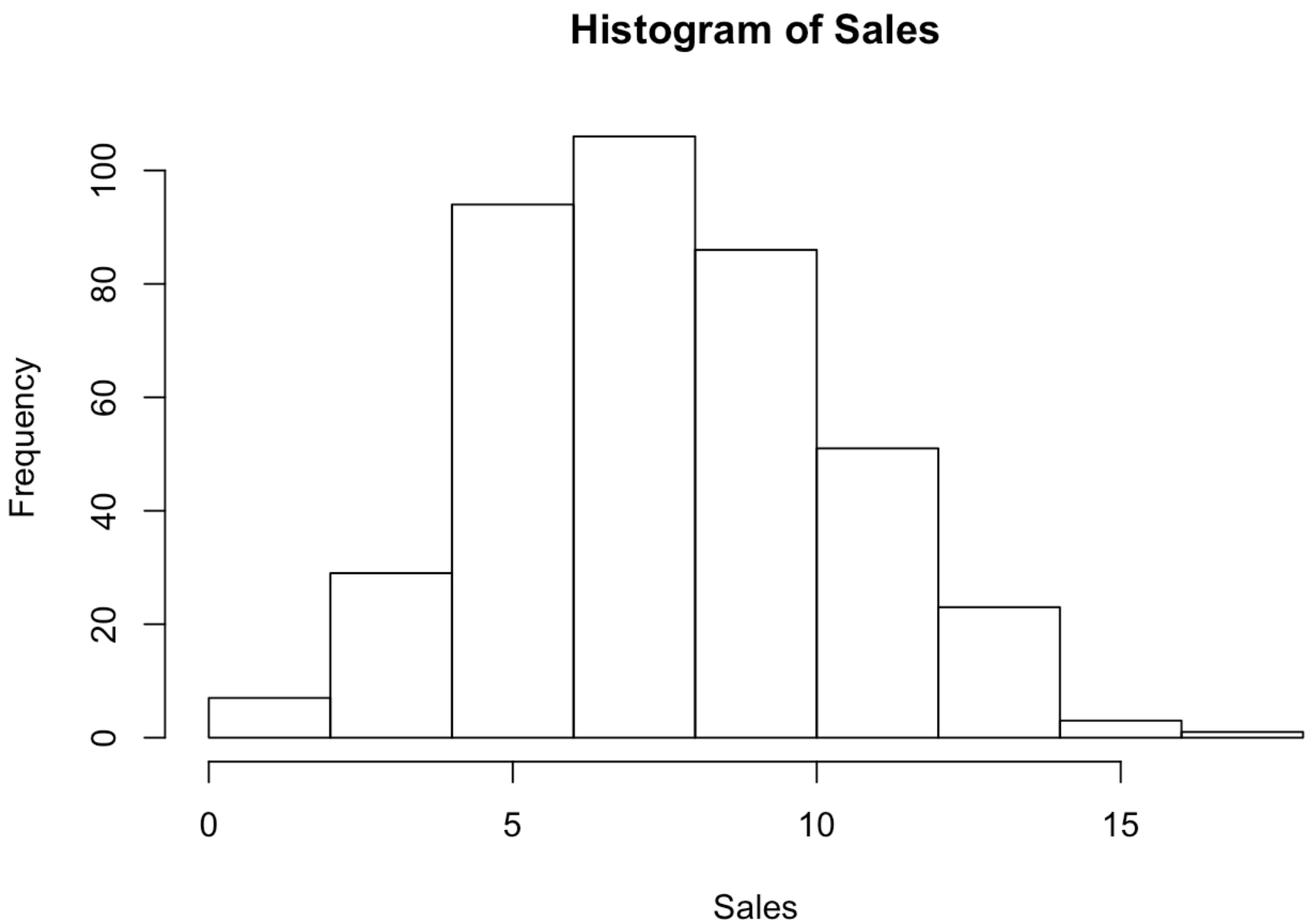
SL RLab8: DECISION TREE METHODS

Fitting Classification Trees

Load tree package and Carseats data Create binary response variable: 'High' (high sales), add new variable to Carseats using data.frame

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



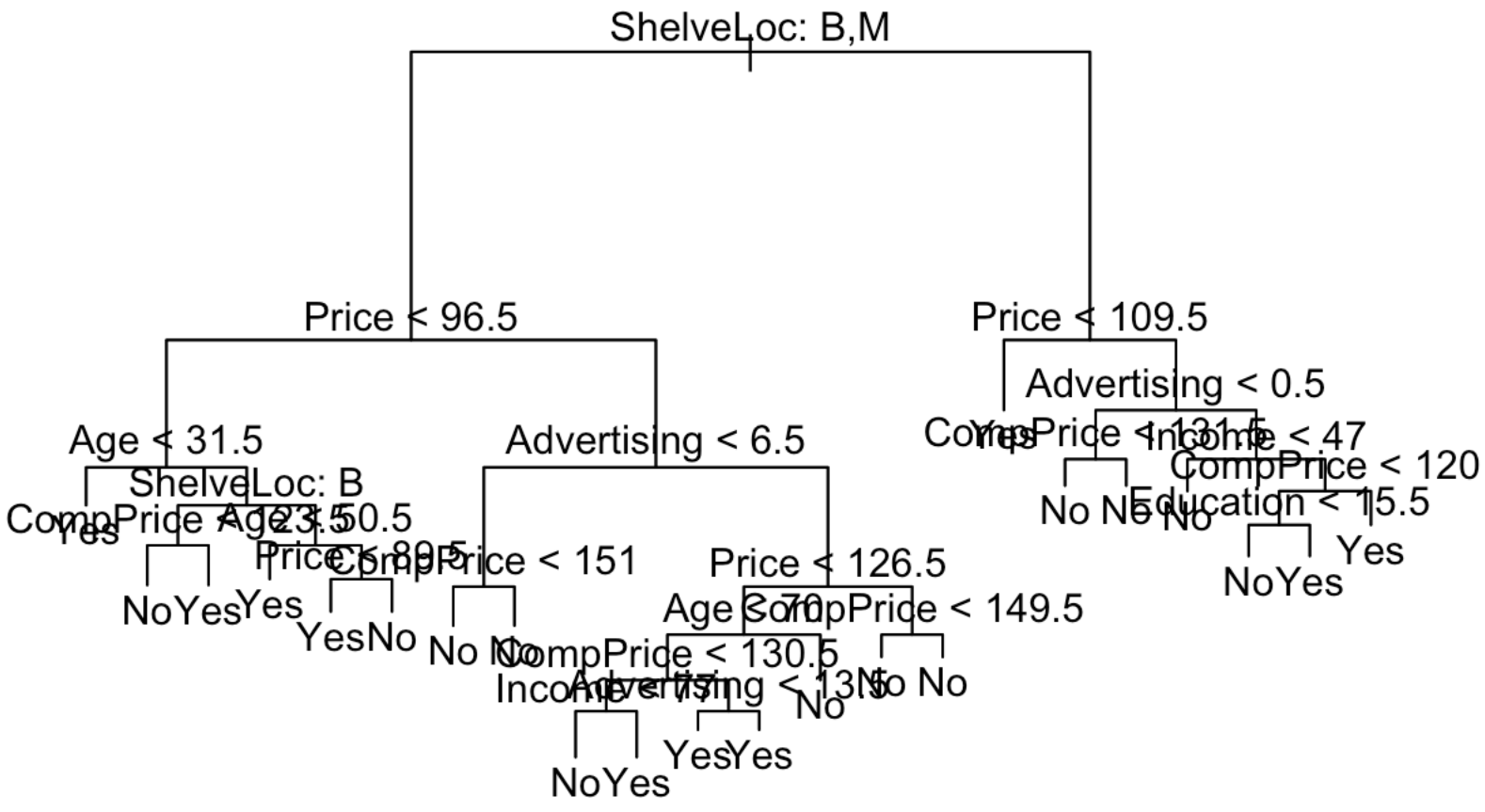

```
High = ifelse(Sales<=9, "No", "Yes")
Carseats = data.frame(Carseats, High)
```

Now, fit a tree to the data, summarize and plot it. Must *exclude* 'Sales' from right side of forumula because the response variable was 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"       "Age"         "CompPrice"   "Advertising"
## [6] "Income"      "Education"
## Number of terminal nodes:  22
## Residual mean deviance:  0.3897 = 147.3 / 378
## Misclassification error rate: 0.09 = 36 / 400
```

```
plot(tree.carseats)
text(tree.carseats, pretty=0.25)
```



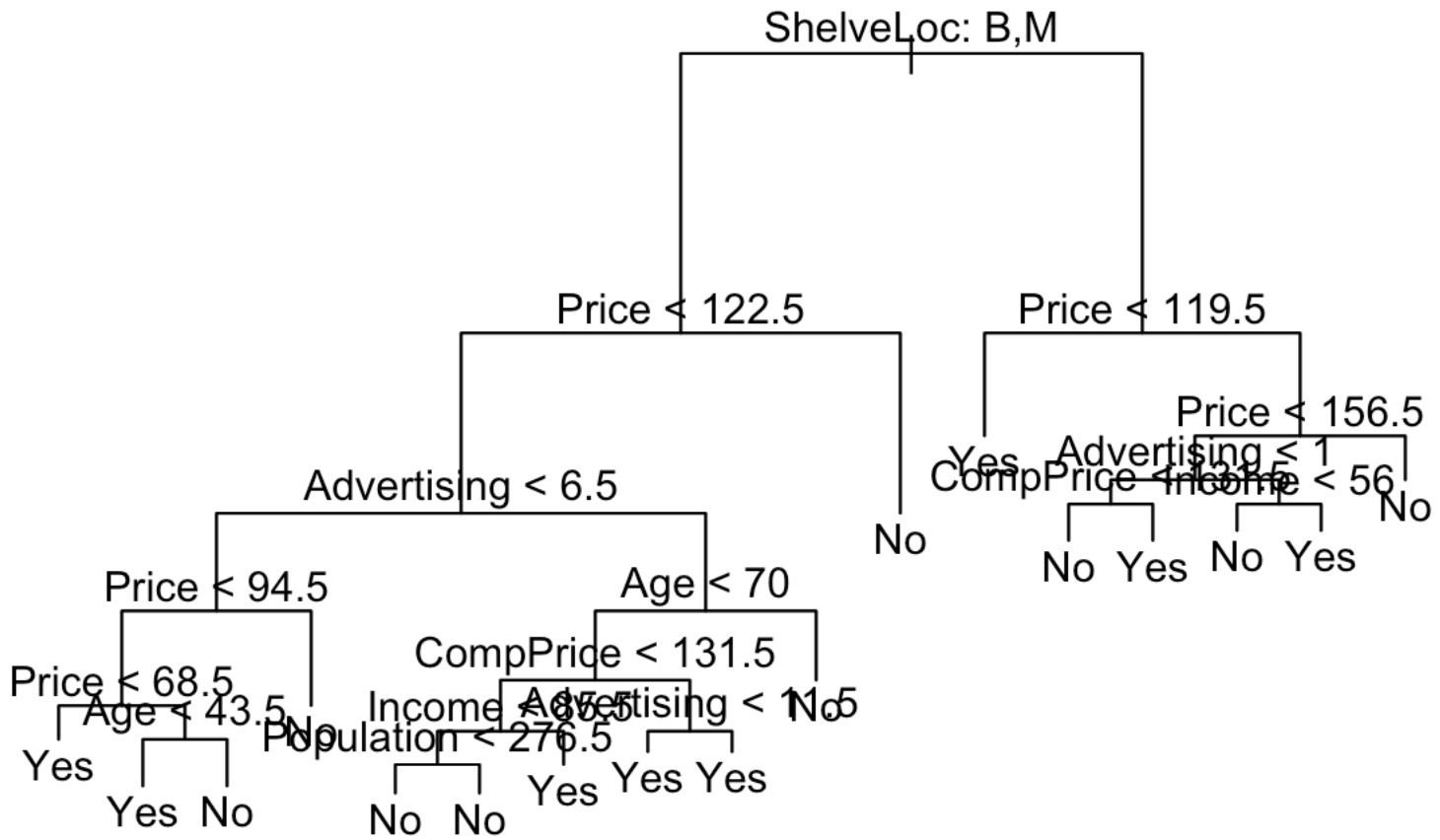
For detailed summary of the tree, print it:

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 476.200 No ( 0.71750 0.28250 )
##     2) ShelveLoc: Bad,Medium 315 291.800 No ( 0.82540 0.17460 )
##       4) Price < 96.5 62  85.370 No ( 0.54839 0.45161 )
##         8) Age < 31.5 6   0.000 Yes ( 0.00000 1.00000 ) *
##         9) Age > 31.5 56  75.040 No ( 0.60714 0.39286 )
##          18) ShelveLoc: Bad 22  17.530 No ( 0.86364 0.13636 )
##            36) CompPrice < 123.5 17   0.000 No ( 1.00000 0.00000 ) *
##            37) CompPrice > 123.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##          19) ShelveLoc: Medium 34  46.660 Yes ( 0.44118 0.55882 )
##            38) Age < 50.5 11   6.702 Yes ( 0.09091 0.90909 ) *
##            39) Age > 50.5 23  30.790 No ( 0.60870 0.39130 )
##              78) Price < 89.5 12  15.280 Yes ( 0.33333 0.66667 ) *
##              79) Price > 89.5 11   6.702 No ( 0.90909 0.09091 ) *
##       5) Price > 96.5 253 171.800 No ( 0.89328 0.10672 )
##        10) Advertising < 6.5 142  21.020 No ( 0.98592 0.01408 )
##          20) CompPrice < 151 131   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 151 11  10.430 No ( 0.81818 0.18182 ) *
##        11) Advertising > 6.5 111 118.400 No ( 0.77477 0.22523 )
##          22) Price < 126.5 65  83.200 No ( 0.66154 0.33846 )
##            44) Age < 70 52  70.850 No ( 0.57692 0.42308 )
##              88) CompPrice < 130.5 37  45.030 No ( 0.70270 0.29730 )
##               176) Income < 77 19   7.835 No ( 0.94737 0.05263 ) *
##               177) Income > 77 18  24.730 Yes ( 0.44444 0.55556 ) *
##              89) CompPrice > 130.5 15  17.400 Yes ( 0.26667 0.73333 )
##               178) Advertising < 13.5 9  12.370 Yes ( 0.44444 0.55556 ) *
##               179) Advertising > 13.5 6   0.000 Yes ( 0.00000 1.00000 ) *
##            45) Age > 70 13   0.000 No ( 1.00000 0.00000 ) *
##          23) Price > 126.5 46  22.180 No ( 0.93478 0.06522 )
##            46) CompPrice < 149.5 41   9.403 No ( 0.97561 0.02439 ) *
##            47) CompPrice > 149.5 5   6.730 No ( 0.60000 0.40000 ) *
##     3) ShelveLoc: Good 85 106.300 Yes ( 0.31765 0.68235 )
##       6) Price < 109.5 28   8.628 Yes ( 0.03571 0.96429 ) *
##       7) Price > 109.5 57  78.580 Yes ( 0.45614 0.54386 )
##        14) Advertising < 0.5 19  19.560 No ( 0.78947 0.21053 )
##          28) CompPrice < 131.5 10   0.000 No ( 1.00000 0.00000 ) *
##          29) CompPrice > 131.5 9  12.370 No ( 0.55556 0.44444 ) *
##        15) Advertising > 0.5 38  45.730 Yes ( 0.28947 0.71053 )
##          30) Income < 47 11  14.420 No ( 0.63636 0.36364 ) *
##          31) Income > 47 27  22.650 Yes ( 0.14815 0.85185 )
##            62) CompPrice < 120 10  13.460 Yes ( 0.40000 0.60000 )
##             124) Education < 15.5 5   5.004 No ( 0.80000 0.20000 ) *
##             125) Education > 15.5 5   0.000 Yes ( 0.00000 1.00000 ) *
##            63) CompPrice > 120 17   0.000 Yes ( 0.00000 1.00000 ) *
```

Create TRAIN, TEST sets (250, 150) based on a split of 400 observations, grow the tree on Training set, and evaluate its performance 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=0.25)
```



Next, use this tree to make prediction for fit on TEST set, evaluate performance and prediction accuracy in table

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

```
##          High
## tree.pred No Yes
##       No  95  16
##       Yes 13  26
```

```
(95+26)/150
```

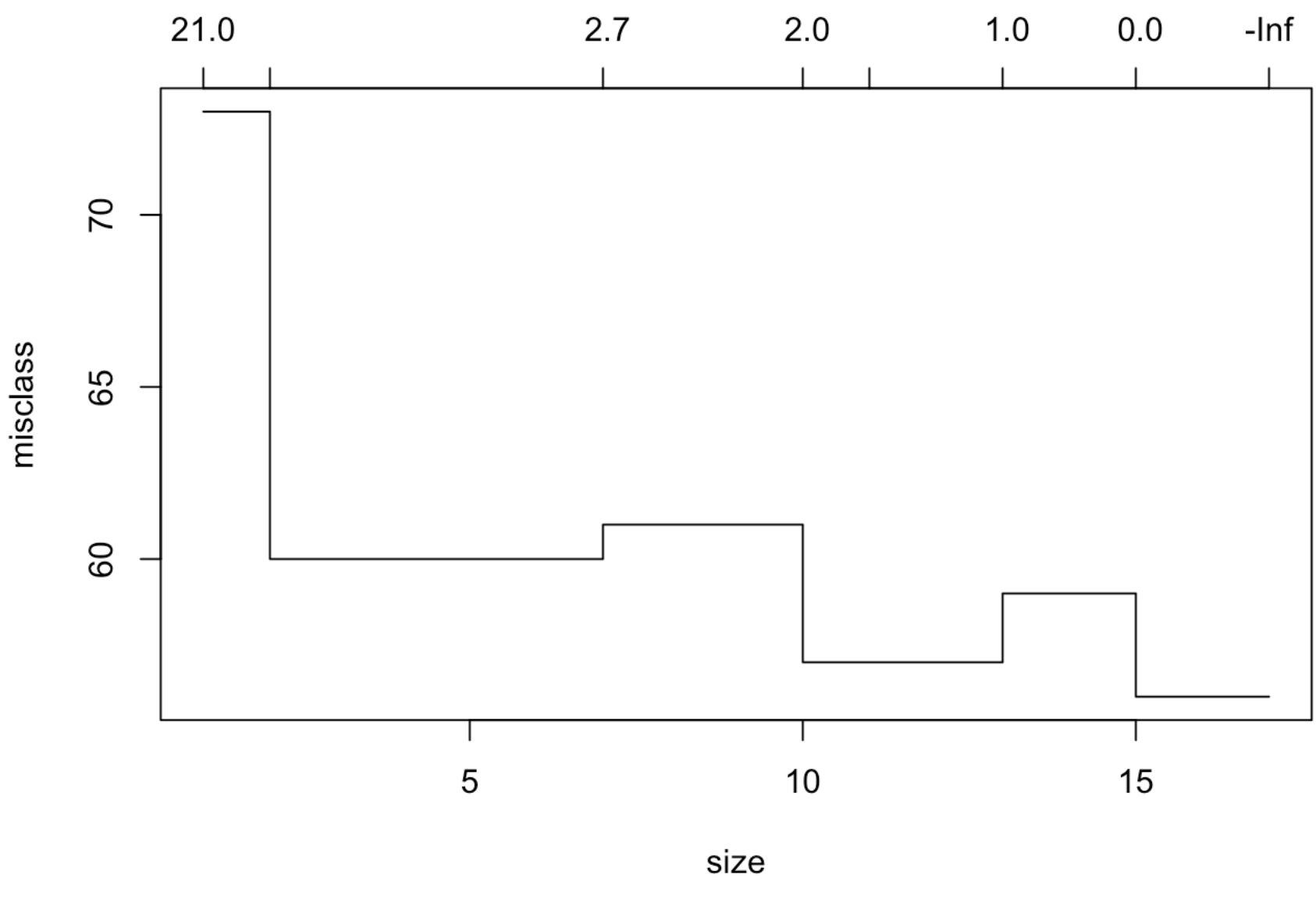
```
## [1] 0.8066667
```

This bushy tree, grown to full depth has too much variance; use 10-fold Cross Validation to prune by misclassification, and plot CV by size of trees and deviance (error)

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

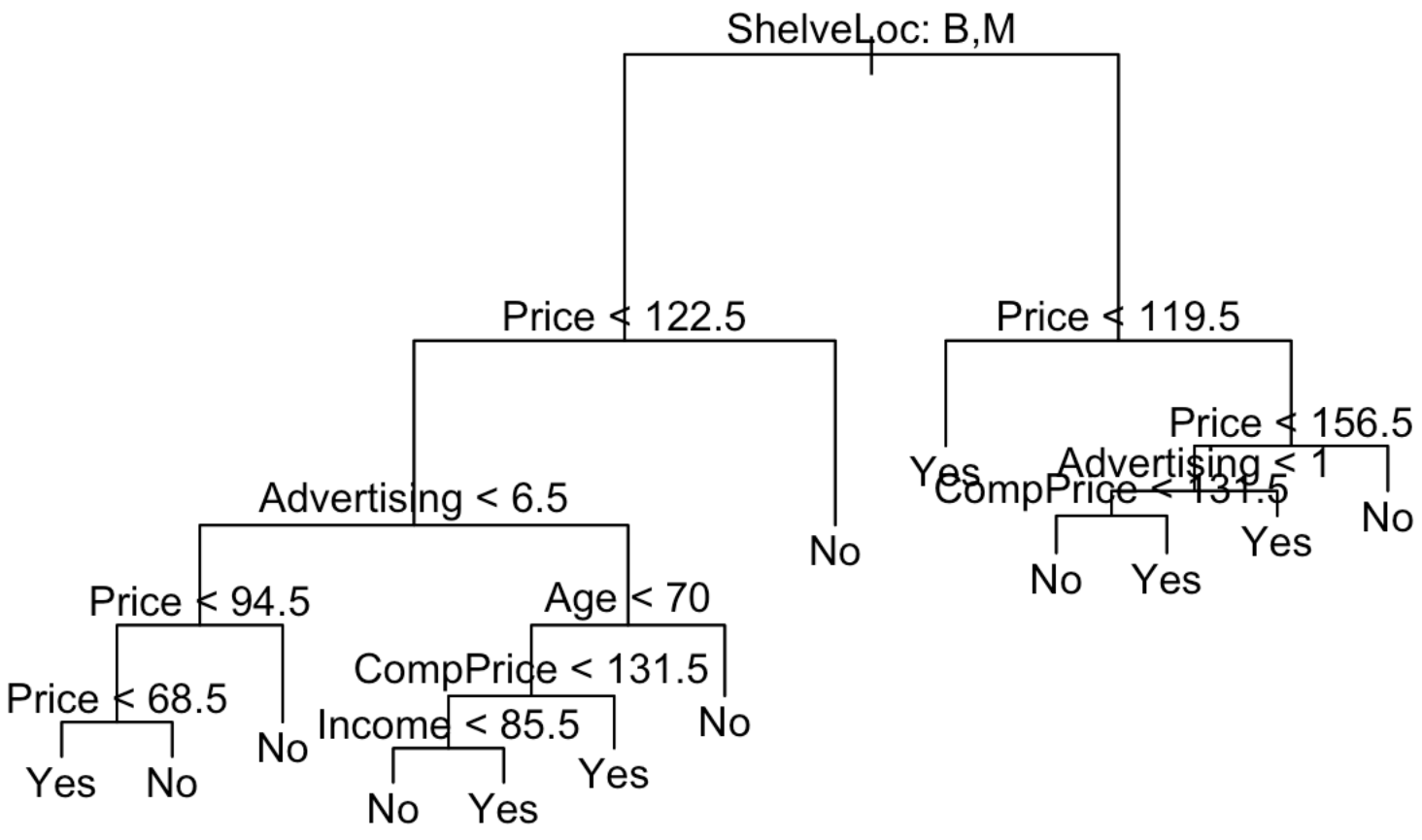
```
## $size
## [1] 17 15 13 11 10  7  2  1
##
## $dev
## [1] 56 56 59 57 57 61 60 73
##
## $k
## [1]      -Inf  0.000000  1.000000  1.500000  2.000000  2.666667  2.800000
## [8] 21.000000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

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



Select around 13 trees as best, and prune by these splits; Fit the tree on full training dataset, and plot the data.

```
prune.carseats = prune.misclass(tree.carseats, best=13)
plot(prune.carseats); text(prune.carseats, pretty=0.25)
```



Evaluate pruned tree on TEST data; similar performance as above, pruning did not affect misclassification errors. However, a simpler tree is easier to interpret.

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

```
##          High
## tree.pred No Yes
##       No  95  17
##       Yes 13  25
```

```
(95+25)/150
```

```
## [1] 0.8
```

========== ## RANDOM FORESTS and BOOSTING Use trees as building blocks for more complex models. ———————— Random Forests build lots of bushy trees, and then averages them to reduce variance. In the process, it decorrelates the trees.

Load randomForest package and Boston housing data, from ‘MASS’ package; housing values, other statistics for 506 suburbs of Boston from 1970 census.

```
library(randomForest)
```

```
## randomForest 4.6-12
```

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

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

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

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

Fit a random forest and see how well it performs. Use 'medv' as response, median housing value ($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: 12.34243
##                     % Var explained: 85.09
```

MSR and % variance explained are based on OOB (out-of-bag) estimates, a clever way to get honest error estimates.

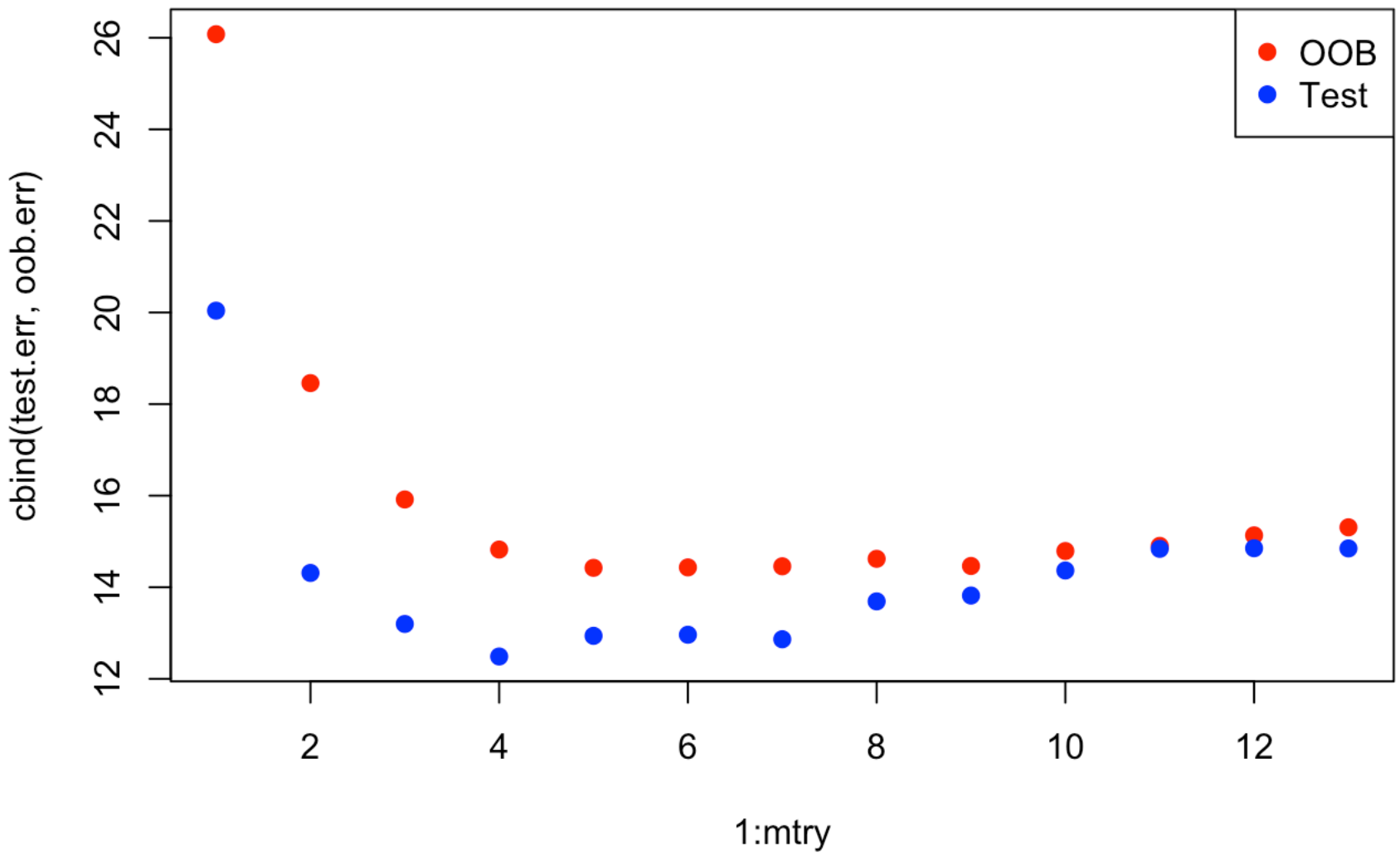
The Model reports 'mtry=4', number of randomly chosen variables at each split. Since $p = 13$, we could try all possible values of 'mtry'. We do so (generating 13 times 400 trees), record the results, and make a plot.

```
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
```

Use matplot to make plot because we have two columns, test.err, oob.err; cbind them together to make two column matrix, and make a single plot in matplot, add a legend.

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



Although the test-error curve drops below the OOB curve, these estimates are based on data, and have their own std errors (typically large). Notice the point on the far left indicates a single tree and points with ‘mtry=13’ correspond to bagging.

BOOSTING

Boosting builds lots of smaller trees; each new tree tries to patch up the deficiencies of the current ensemble of trees. Boosting grows smaller, stubbier trees, and goes after bias.

Import gbm package (“Gradient Boosted Machines”, Friedman) Call gbm, “gaussian distribution”, 10000 shallow trees, with shrinkage parameters=0.01, and interaction depth of 4 splits

```
library(MASS)
library(gbm)
```

```
## Warning: package 'gbm' was built under R version 3.3.2
```

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

```
## Warning: package 'survival' was built under R version 3.3.2
```

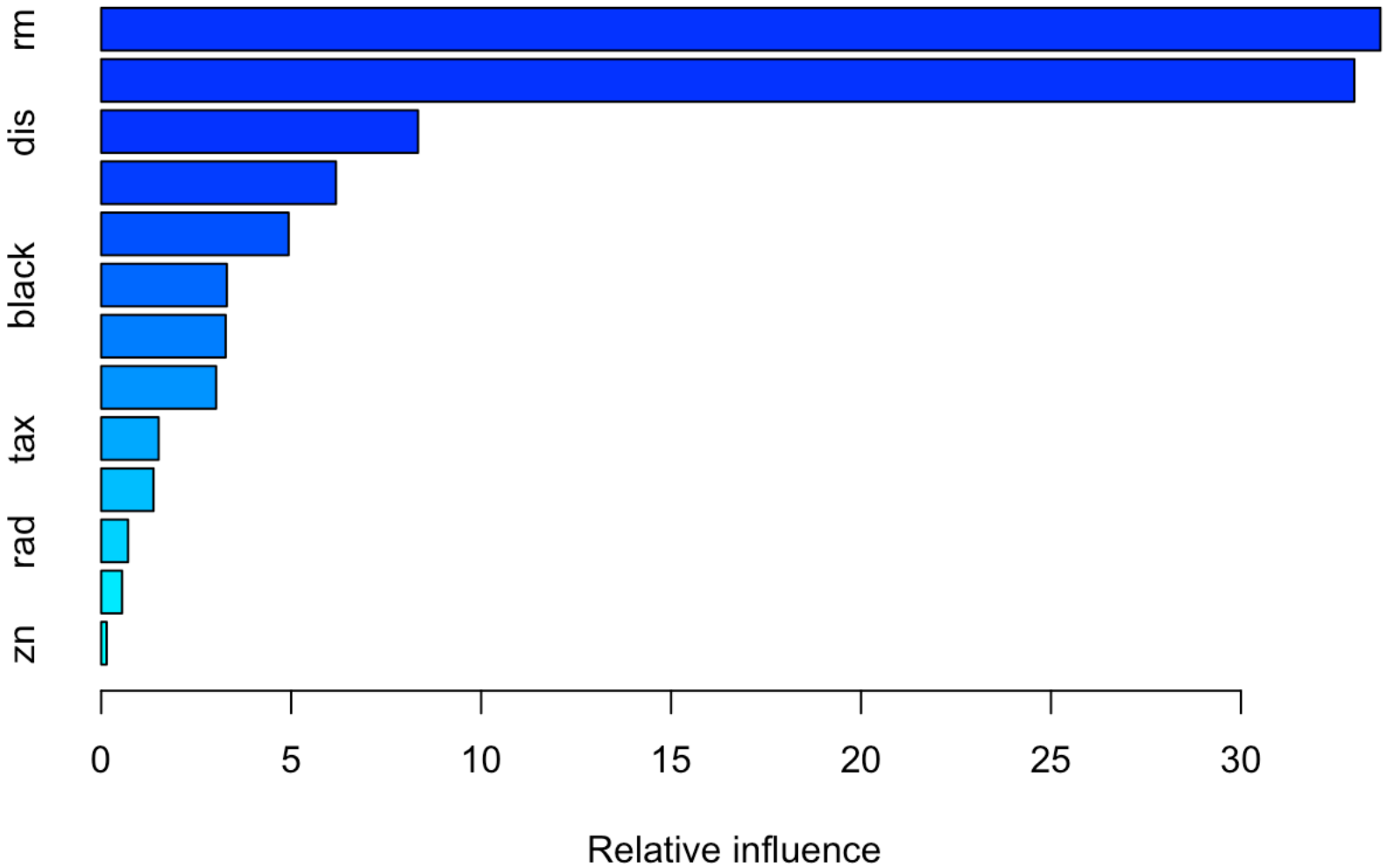
```
## Loading required package: lattice
```

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

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

```
## Loaded gbm 2.1.3
```

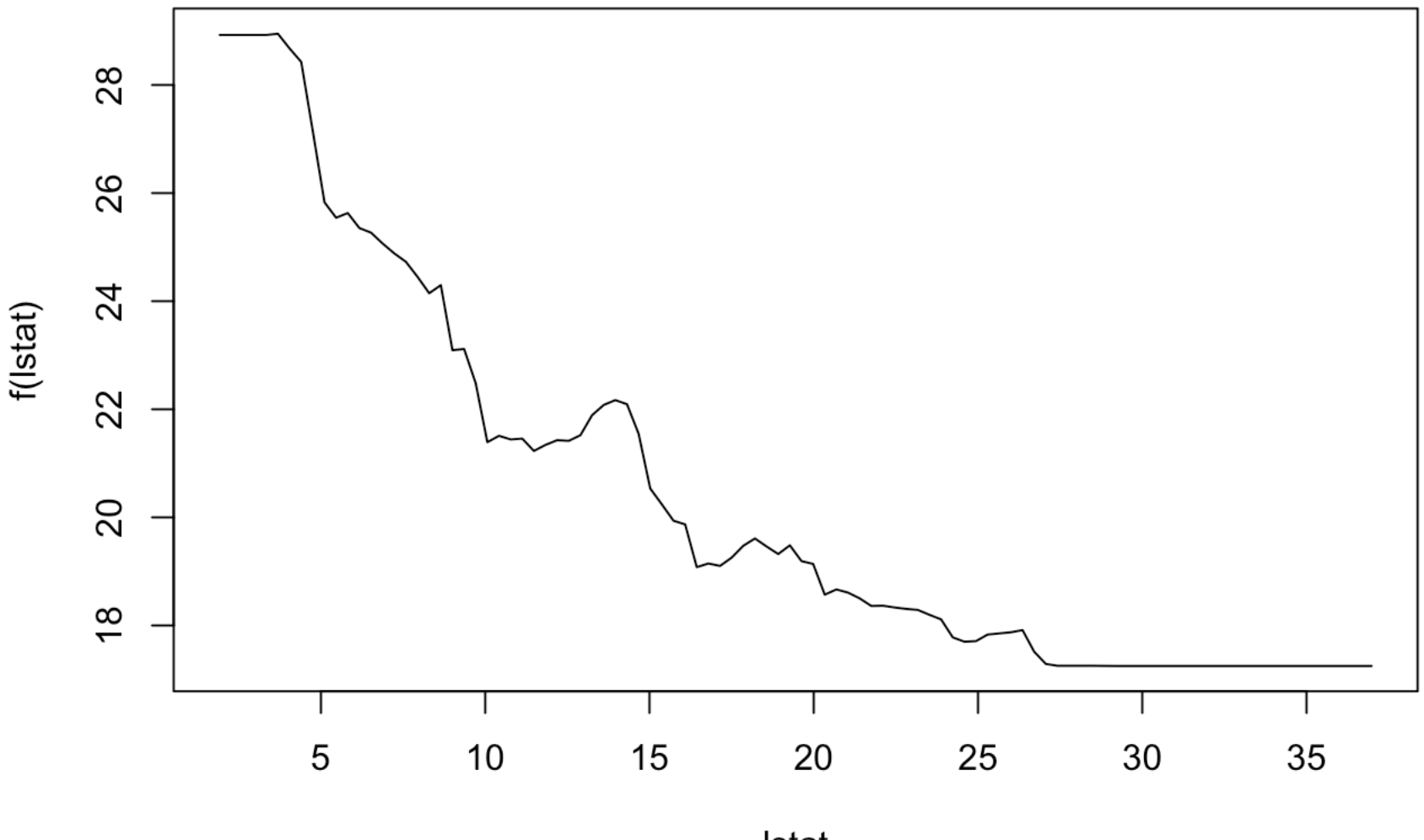
```
set.seed(101)
boost.boston = gbm(medv~., data=Boston[train,], distribution="gaussian", n.trees=1000
0, shrinkage=0.01, interaction.depth=4)
summary(boost.boston)
```



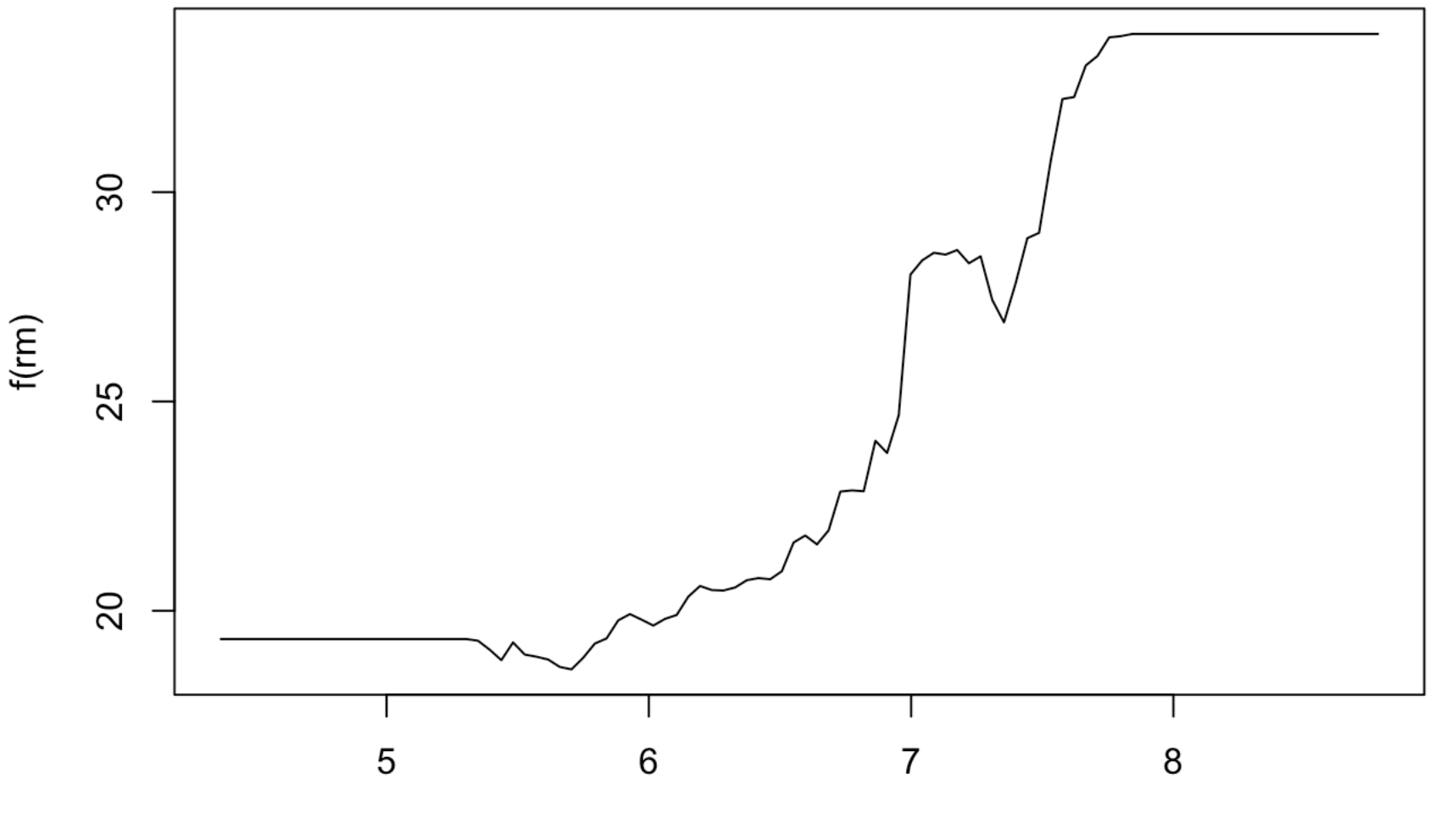

```
##             var    rel.inf
## rm           rm 33.6702467
## lstat     lstat 32.9837784
## dis         dis  8.3394238
## crim       crim  6.1793462
## nox         nox  4.9333527
## black     black  3.3059634
## ptratio ptratio  3.2753985
## age         age  3.0250287
## tax         tax  1.5112966
## chas       chas  1.3771420
## rad         rad  0.7065527
## indus     indus  0.5492869
## zn           zn  0.1431833
```

Relative influence plot shows that Rooms and LStatus are two most influential predictors above all others in model Construct partial dependence plots for top two predictors.

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




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



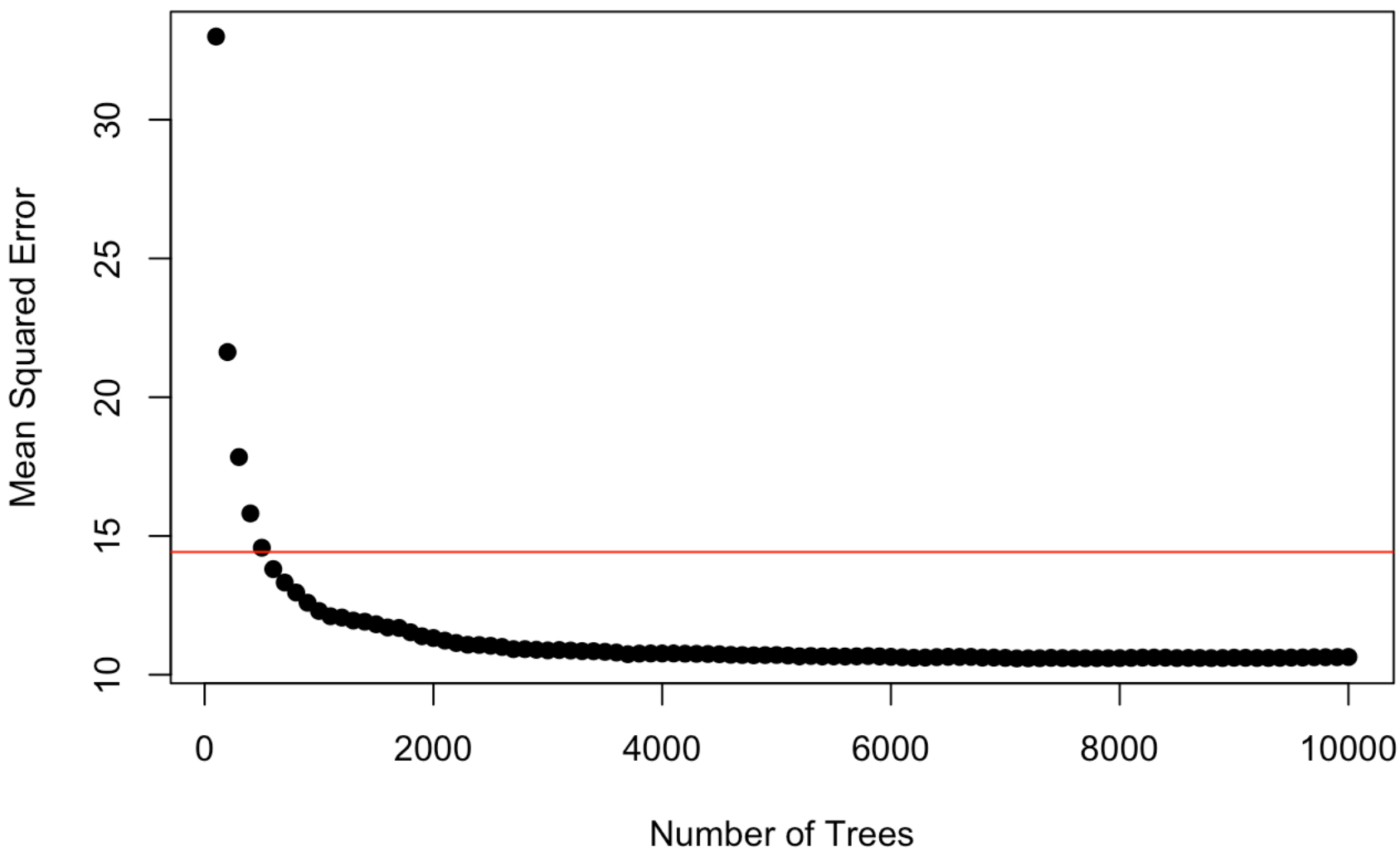
Make a prediction on the TEST set. With boosting, the number of trees is a tuning parameter; if we have too many we can overfit. Use cross validation to select number of trees, do as exercise.

Calculate test error as function of number of trees, and plot. Create grid of number of trees, run predict function on boosted model, which returns a matrix of predictions on test data. Compute columnwise TEST MSE for each of those

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

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

```
b.err = with(Boston[-train,], apply( (pred.mat-medv)^2, 2, mean))
plot(n.trees, b.err, pch=19, ylab="Mean Squared Error", xlab="Number of 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
Number of Trees