Chapter 8 Lab

Contents

Setup **1**

Decision Trees: Classification **2**
The Choice of Sub-Tree 6
Summary 9

Regression Trees **10**
Caution 14

Bagging and Random Forests **14**
Summary 17

Boosting **17**
Summary 20

Setup

We work on the `Carseats` data, and we are all about growing a tree from carseats!

```
rm(list = ls())
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

```
library(ISLR)
library(tree)
library(gmodels)
# knowing the data and variables
dim(Carseats)
```

```
## [1] 400  11
```

```
summary(Carseats)
```

```
##      Sales          CompPrice       Income        Advertising
##  Min.   : 0.000   Min.   : 77   Min.   : 21.00   Min.   : 0.000
##  1st Qu.: 5.390   1st Qu.:115   1st Qu.: 42.75   1st Qu.: 0.000
##  Median : 7.490   Median :125   Median : 69.00   Median : 5.000
##  Mean   : 7.496   Mean   :125   Mean   : 68.66   Mean   : 6.635
##  3rd Qu.: 9.320   3rd Qu.:135   3rd Qu.: 91.00   3rd Qu.:12.000
##  Max.   :16.270   Max.   :175   Max.   :120.00   Max.   :29.000
##    Population        Price        ShelveLoc        Age
##  Min.   : 10.0   Min.   : 24.0   Bad   : 96   Min.   :25.00
##  1st Qu.:139.0   1st Qu.:100.0   Good  : 85   1st Qu.:39.75
##  Median :272.0   Median :117.0   Medium:219   Median :54.50
##  Mean   :264.8   Mean   :115.8                Mean   :53.32
##  3rd Qu.:398.5   3rd Qu.:131.0                3rd Qu.:66.00
```

```
##  Max.   :509.0   Max.   :191.0                Max.   :80.00
##    Education    Urban       US
##  Min.   :10.0   No :118   No :142
##  1st Qu.:12.0   Yes:282   Yes:258
##  Median :14.0
##  Mean   :13.9
##  3rd Qu.:16.0
##  Max.   :18.0
```

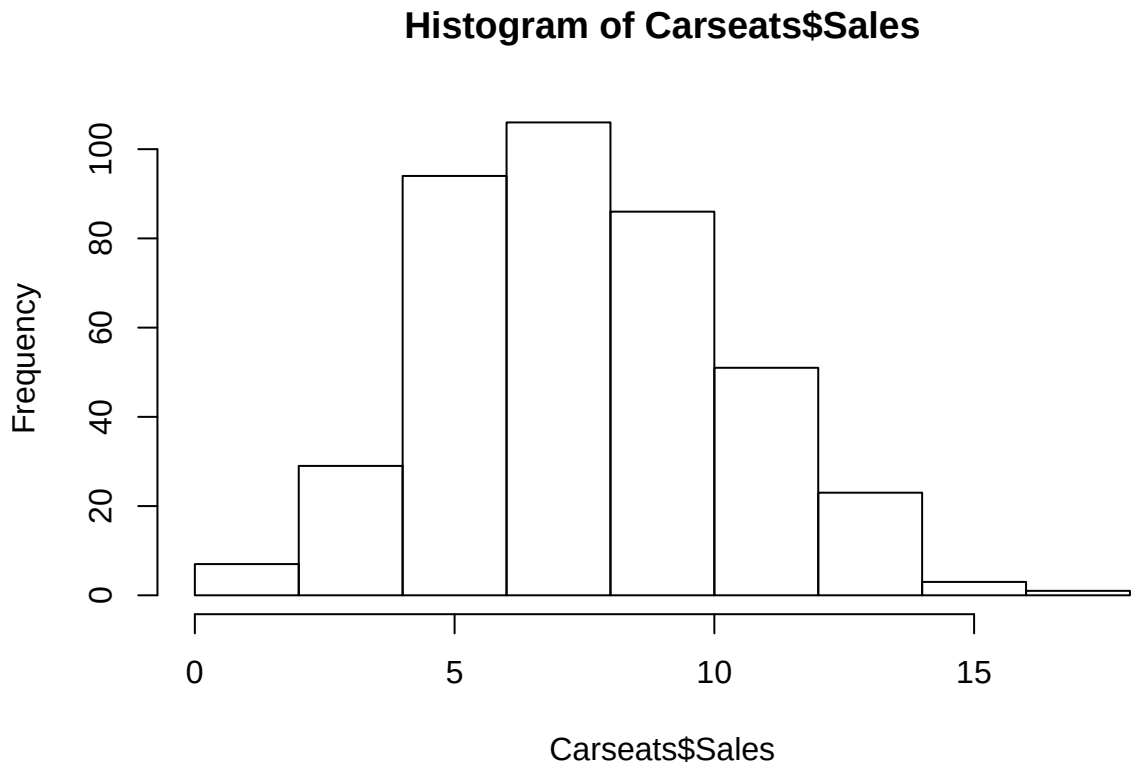
```
sum(is.na(Carseats))  # no missing
```

```
## [1] 0
```

```
# learning about response
summary(Carseats$Sales)  # unit unknown, mean and median 7.5
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   5.390   7.490   7.496   9.320  16.270
```

```
hist(Carseats$Sales)  # not much skewed
```




```
# attaching the data: not recommended, but we follow the ISLR book
attach(Carseats)
```

Decision Trees: Classification

After the initial setup, we create a dummy variable from `Sales` to work on as the response.

```
High = ifelse(Sales > 8, "YES", "No")
summary(High)
```

```
##    Length     Class      Mode
##       400 character character
```

The variable `High` is a vector of strings. It resides in the Global Environment and that is why `summary(High)` gives us no information about the values it takes. So the next step is to add the variable to the working data:

```
Carseats = data.frame(Carseats, High)
```

Time to grow a tree:

```
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
```

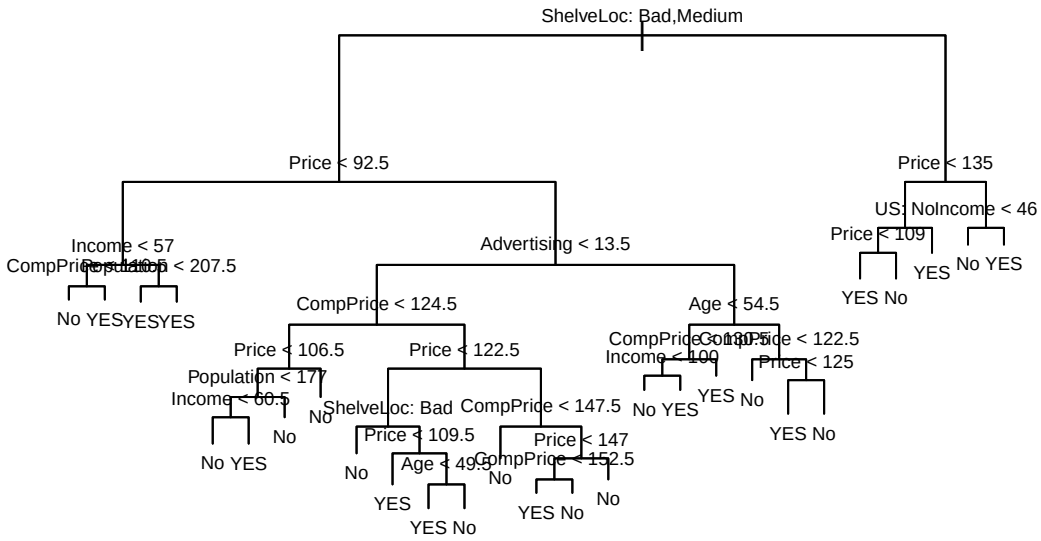
What is the stopping criteria? The default criterion is used above. The stopping criterion is controlled by the argument `control = tree.control(...)` for `tree()` whose default is to have `tree.control(nobs, mincut = 5, minsize = 10, mindev = 0.01)`.

We see residual mean deviance in the output. How is the residual mean deviance computed? It is equal to two times the ratio of log-likelihoods of the saturated model to the model being considered:

$$Dev = 2 \times \frac{\mathcal{L}(y|\theta_s)}{\mathcal{L}(y|\theta_0)} = 2 \times \frac{1}{\log\left(\prod_{m,k}\left(\hat{p}_{mk}^{n_{mk}}\right)\right)} = -2\sum_m \sum_k n_{mk} \log \hat{p}_{mk}\,,$$

where θ_s is the vector of parameters for the saturated model, and θ_0 is the one for the model we consider (which is of course nested in the saturated model). The probability distributions are assumed to be multimonial. The saturated model is tree with a leaf for every observation. Hence, the saturated tree perfectly fits the data and its estimates $\hat{p}_{mk}$ are all equal to 1. The mean deviance is computed by dividing deviance by $n - |T|$. Next, we graphically represent the tree:

```
plot(tree.carseats)
text(tree.carseats, pretty = 0, cex = 0.6)
```



The ability to visualize a tree is one of the most attractive properties of trees. Shelving location appears to be the most important indicator of Sales. The option `pretty = 0` makes the graph display category names, rather than single letters for each category. More details on the tree can be seen by typing the tree's name:

```
summary(Carseats$High)
```

```
##  No YES
## 236 164
```

```
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 ) *
```

Each node is represented with a number. One can trace back a node by dividing this number by two and taking the integer part of it. The resulting number denotes the parent node. By sequentially doing this, we can trace the node back to the root. * denotes the terminal nodes.
The other information are the number of observations in each branch, the deviance, the overall prediction for the branch, and the fraction of leaves in that branch that take on values "NO" and "YES", respectively.
Why is "NO" presented before "YES"? It is probably due to the way the variable is defined. However, which value is the baseline does not make much of a difference, here (although it makes a difference in other cases, e.g. when we want to interpret linear regression results).

What about the test error rate?

```
set.seed(2)
train = sample(nrow(Carseats), 200)
test = -train
test.carseats = Carseats[test, ]
```

```
test.high = High[test]
tree.carseats = tree(High ~ . - Sales, data = Carseats, subset = train)
tree.preds = predict(tree.carseats, newdata = test.carseats, type = "class")
head(tree.preds)
```

```
## [1] No  YES No  No  No  YES
## Levels: No YES
```

```
table(tree.preds, test.high)
```

```
##           test.high
## tree.preds No YES
##        No  86  27
##        YES 30  57
```

```
(86+57)/200
```

```
## [1] 0.715
```

The test error rate estimate is 72%, while the training error rate was estimated to be 91%.

The Choice of Sub-Tree

Pruning the tree is done through cross-validation over a sequence of trees found by cost-complexity pruning.

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

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

```
cv.carseats
```

```
## $size
## [1] 19 17 14 13  9  7  3  2  1
##
## $dev
## [1] 55 55 53 52 50 56 69 65 80
##
## $k
## [1]       -Inf  0.0000000  0.6666667  1.0000000  1.7500000  2.0000000
## [7]  4.2500000  5.0000000 23.0000000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

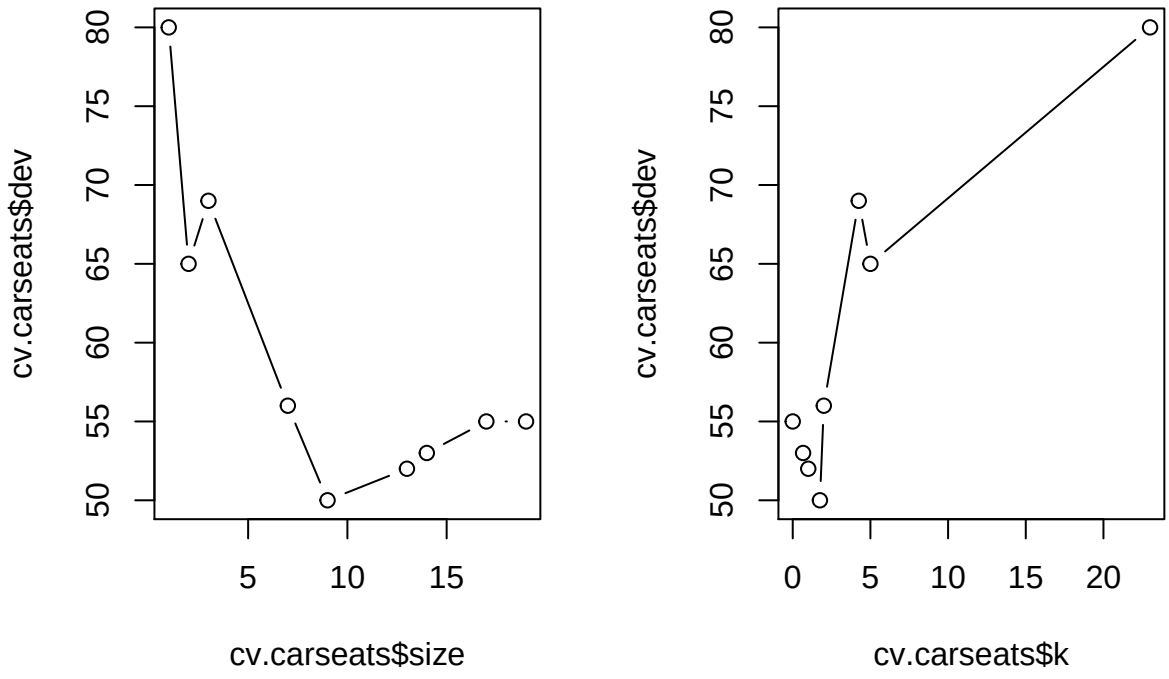
The `FUN` argument in `cv.tree` determines the nested sequence of subtrees and is the output of the command `prune.tree()`, which has uses either of the following estimates for error:

1. Deviance, which is the default
2. Misclassification error, which is done through equating the argument `FUN` to either `prune.tree(method = "misclass")` or its short form `prune.misclass`

In the output of `cv.tree` above, `$size` is the number of *terminal* nodes, and `$k` corresponds to α, the tuning parameter in cost complexity pruning. The output `$dev` corresponds to the error, which is misclassification

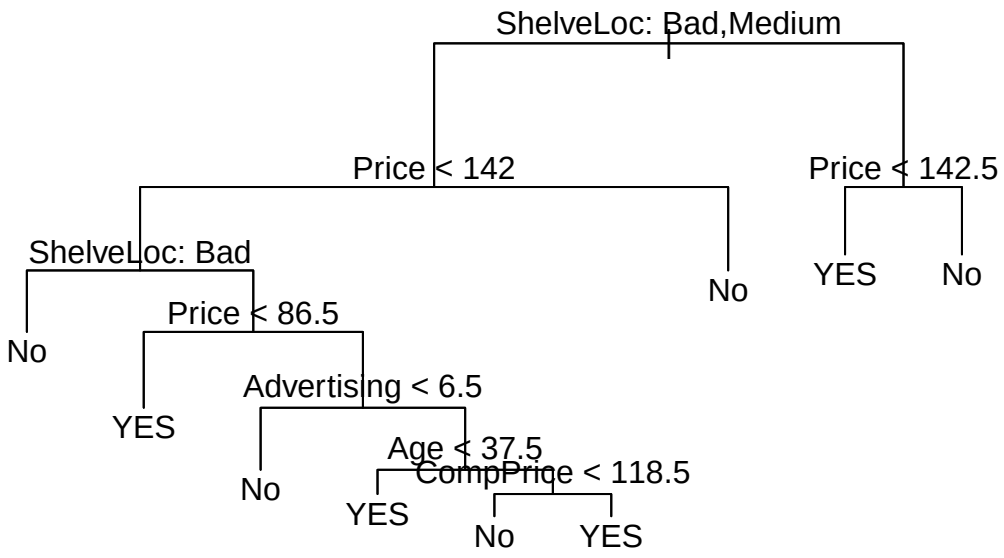
error in this case, despite its name. The value of $\alpha = -\infty$ would be the largest possible tree with RSS = 0, which has 27 leaves here.

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



So the tree with 9 terminal nodes results in the lowest cross-validation error rate.

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




```
tree.prune
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 200 269.200 No ( 0.6000 0.4000 )
##     2) ShelveLoc: Bad,Medium 153 185.400 No ( 0.7059 0.2941 )
##       4) Price < 142 130 167.700 No ( 0.6538 0.3462 )
##         8) ShelveLoc: Bad 39  29.870 No ( 0.8718 0.1282 ) *
##         9) ShelveLoc: Medium 91 124.800 No ( 0.5604 0.4396 )
##          18) Price < 86.5 9   0.000 YES ( 0.0000 1.0000 ) *
##          19) Price > 86.5 82 108.700 No ( 0.6220 0.3780 )
##            38) Advertising < 6.5 52  56.180 No ( 0.7692 0.2308 ) *
##            39) Advertising > 6.5 30  39.430 YES ( 0.3667 0.6333 )
##              78) Age < 37.5 5   0.000 YES ( 0.0000 1.0000 ) *
##              79) Age > 37.5 25  34.300 YES ( 0.4400 0.5600 )
##               158) CompPrice < 118.5 8   8.997 No ( 0.7500 0.2500 ) *
##               159) CompPrice > 118.5 17  20.600 YES ( 0.2941 0.7059 ) *
##       5) Price > 142 23   0.000 No ( 1.0000 0.0000 ) *
##     3) ShelveLoc: Good 47  53.400 YES ( 0.2553 0.7447 )
##       6) Price < 142.5 38  29.590 YES ( 0.1316 0.8684 ) *
##       7) Price > 142.5 9   9.535 No ( 0.7778 0.2222 ) *
```

The pruned tree becomes more interpretable. The test error for the tree with 9 leaves:

```
tree.preds = predict(tree.prune, newdata = Carseats[test, ], type = "class")
table(tree.preds, High[test])
```

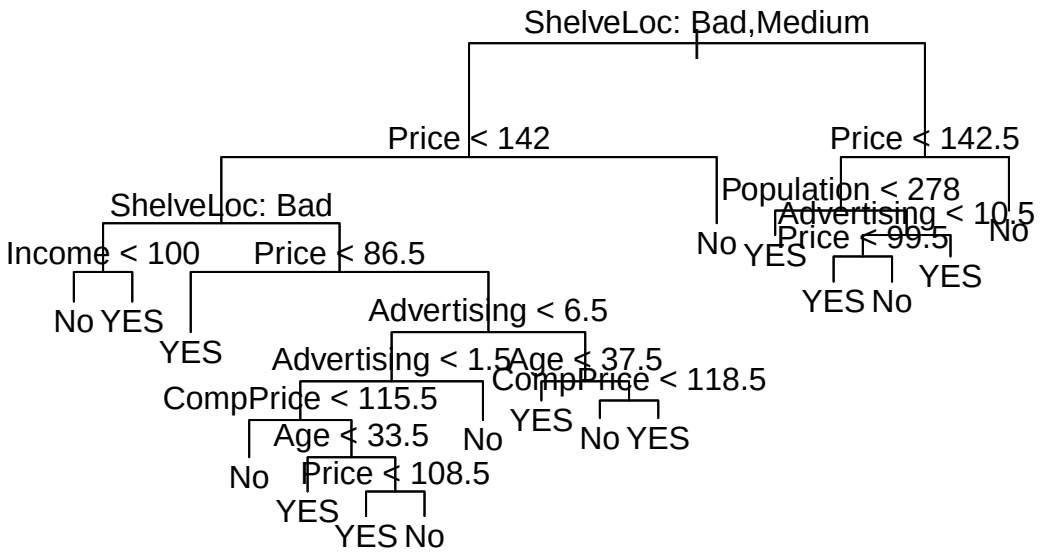
```
##
## tree.preds No YES
##        No  94  24
##        YES 22  60
```

```
(94 + 60)/ 200
```

```
## [1] 0.77
```

The prunin process has not only made the tree more interpretable, but it has also improved the classification accuracy. The prediction accuracy falls if we increase the value of `best`:

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




```
prune.pred.15 = predict(prune.carseats.15, type = "class", newdata = Carseats[test, ])
tab = table(prune.pred.15, High[test])
sum(diag(tab))/sum(tab)
```

```
## [1] 0.74
```

Summary

- Creating qualitative variable: We can use `ifelse` to create new factor variables.
  - Remember to merge the generated variables with the data set
- After fitting the tree and seeing the fit's summary, we learned how to
  - plot it

        * The `plot.tree()` command will plot the tree without text.
        * How to add details? We use `text.tree()` to add text.
            - Factors by name: option `pretty = 0`
    - see its nodes and details: type the name of the tree
    - compute test error rate
        * In `predict.tree()`, the argument `type = class` used to get factor predictions
    - do cost complexity pruning: use `cv.tree` with the argument `FUN = prune.misclass`
        * `cv.tree(...)$k` denotes the number of terminal nodes
    - depict the sub-tree with lowest error.
        * `prune.misclass(tree_name, best = best_leaves)`, where best_leaves is the number of terminal nodes that result in lowest cross-validation error
- Remember to set the seed before doing validation set approach or CV

Regression Trees

We use the `Boston` datat set.

```
library(MASS)
library(tree)
```

Grow on the training set:

```
set.seed(1)
train = sample(nrow(Boston), size = nrow(Boston)/2)
test = -train
tree.boston = tree(medv ~ ., data = Boston, subset = train)
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
## [1] "lstat" "rm"    "dis"
## Number of terminal nodes:  8
## Residual mean deviance:  12.65 = 3099 / 245
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -14.10000  -2.04200  -0.05357   0.00000   1.96000  12.60000
```

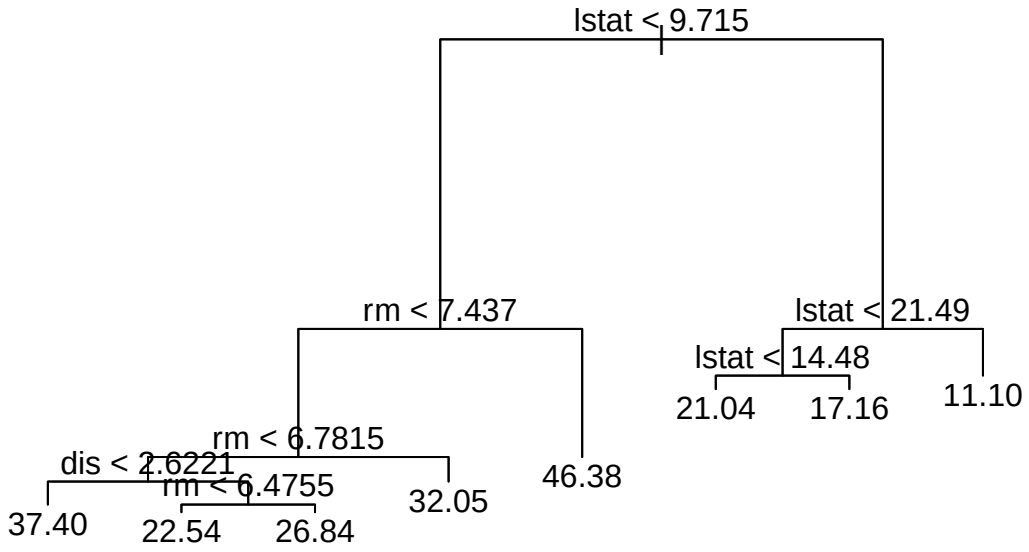
Three variables are used and a tree with 8 leaves is grown. As it is the case with linear regression, deviance is equal to RSS over the variance of error (or equal to RSS, if we use an alternative defintion for deviance), under normal distribution. In linear regression, the assumption was that we have a linear model with Gaussian errors. Under this assumption, we could verify that

$$Dev = 2 \times \frac{\mathcal{L}(y|\theta_s)}{\mathcal{L}(y|\theta_0)} = 2 \times \frac{-1}{\sigma^2}\big(\mathrm{RSS}_s - \mathrm{RSS}_0\big) = \frac{1}{\sigma^2}\mathrm{RSS}_0\,,$$

What about regression trees? It appears that deviance would equal the RSS, under the probability model that considers that the points at each leaf are distributed normally. For details and references (e.g. Venables and Ripley) see here.

Plot the tree:

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



Cost complexity pruning:

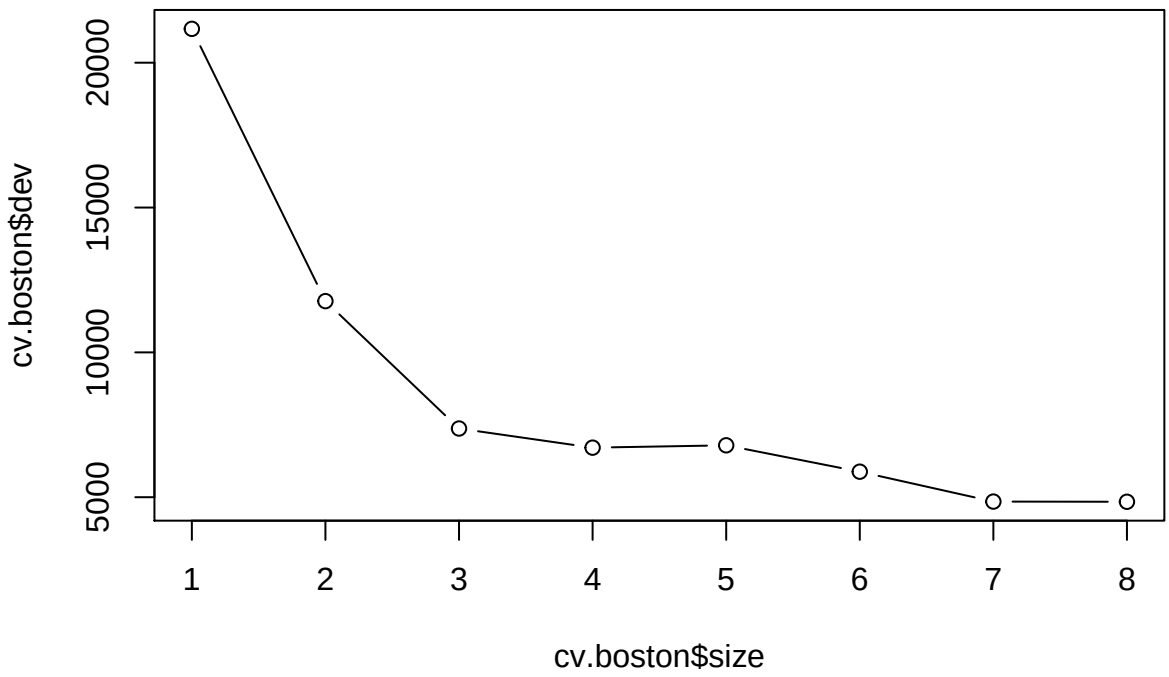
```
set.seed(2)
cv.boston = cv.tree(tree.boston)
best.leaves.num = cv.boston$size[which.min(cv.boston$dev)]
cv.boston
```

```
## $size
## [1] 8 7 6 5 4 3 2 1
##
## $dev
## [1]  4842.966  4849.296  5880.651  6791.346  6710.673  7371.095 11769.678
## [8] 21168.301
##
## $k
## [1]      -Inf  255.6581  451.9272  768.5087  818.8885 1559.1264 4276.5803
## [8] 9665.3582
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
best.leaves.num
```

```
## [1] 8
```

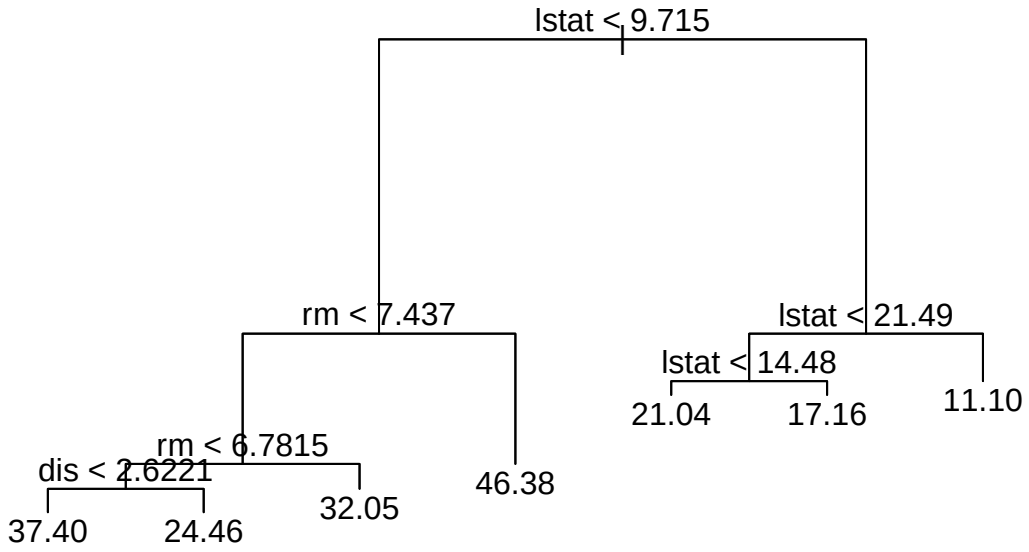
```
plot(cv.boston$size, cv.boston$dev, type = "b")
```



The algorithm chooses the sub-tree with 8 leaves, which is the tree itself. Since the test MSE is very close for a sub-tree with 7 leaves, choosing this smaller tree might be better. We depict the three-leaved tree below. But in order to be consistent with the book, we use the 8-leaved tree to compute the test error.

Depicting the chosen sub-tree:

```
prune.boston = prune.tree(tree.boston, best = 7)
plot(prune.boston)
text(prune.boston, pretty = 0)
```



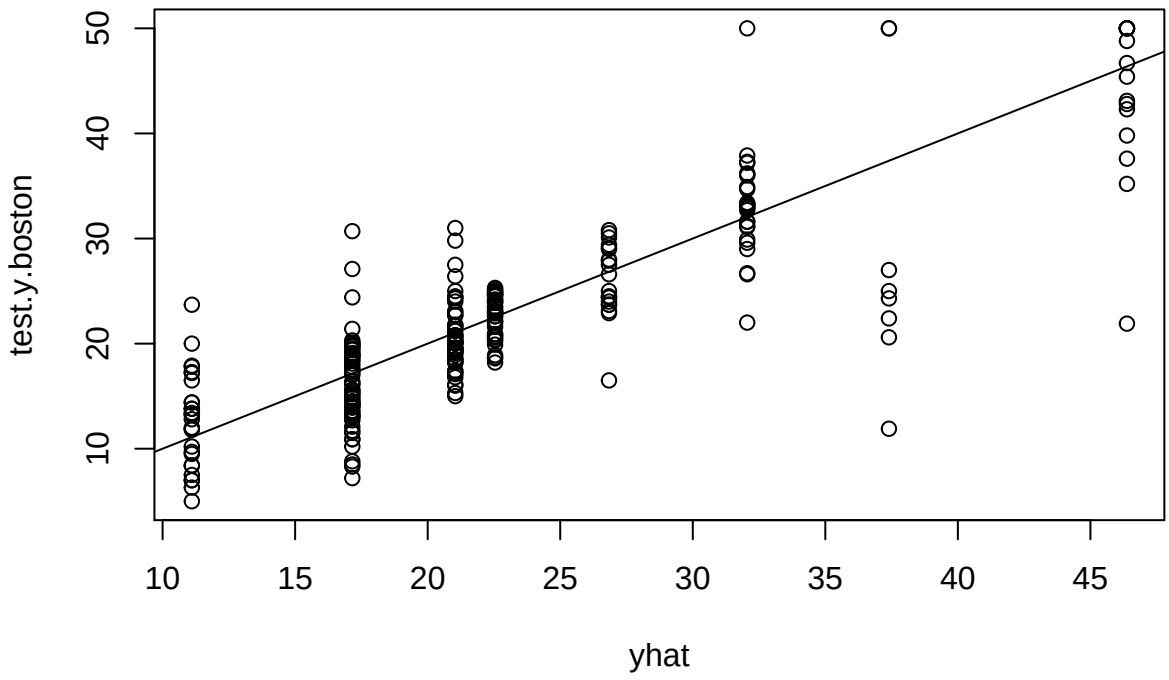
Test error for the chosen sub-tree

```
chosen.tree = tree.boston  # to be consistent with the book, the full tree is considered
yhat = predict(chosen.tree, newdata = Boston[test, ])
test.y.boston = Boston[test, "medv"]
mse.boston = mean((yhat - test.y.boston)^2)
mse.boston
```

```
## [1] 25.04559
```

Visualization of the fit

```
plot(yhat, test.y.boston)
abline(0, 1)
```



Note the discreteness in fitted values, which is a consequence of having a single tree.

Note that we do not specify `prune.misClass`, as we did for classication. The cases where did so was in:

1. `predict.tree()`
2. `cv.tree`

Caution

- identifying the best tree: `which.min(cv.boston$dev)` gives the wrong answer. Why? What should we use?
  - note that in pruning, size (`cv.tree$size`) is decreasing, so the rationale behind using `which.min` alone breaks apart.

- When comparing fit and actual values, draw the 45 degrees line, using `abline`. Otherwise, due to different units on axes, the relationship might not look as meaningful.

Bagging and Random Forests

We first load the library:

```
library(randomForest)
```

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

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

```
library(MASS)
dim(Boston)
```

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

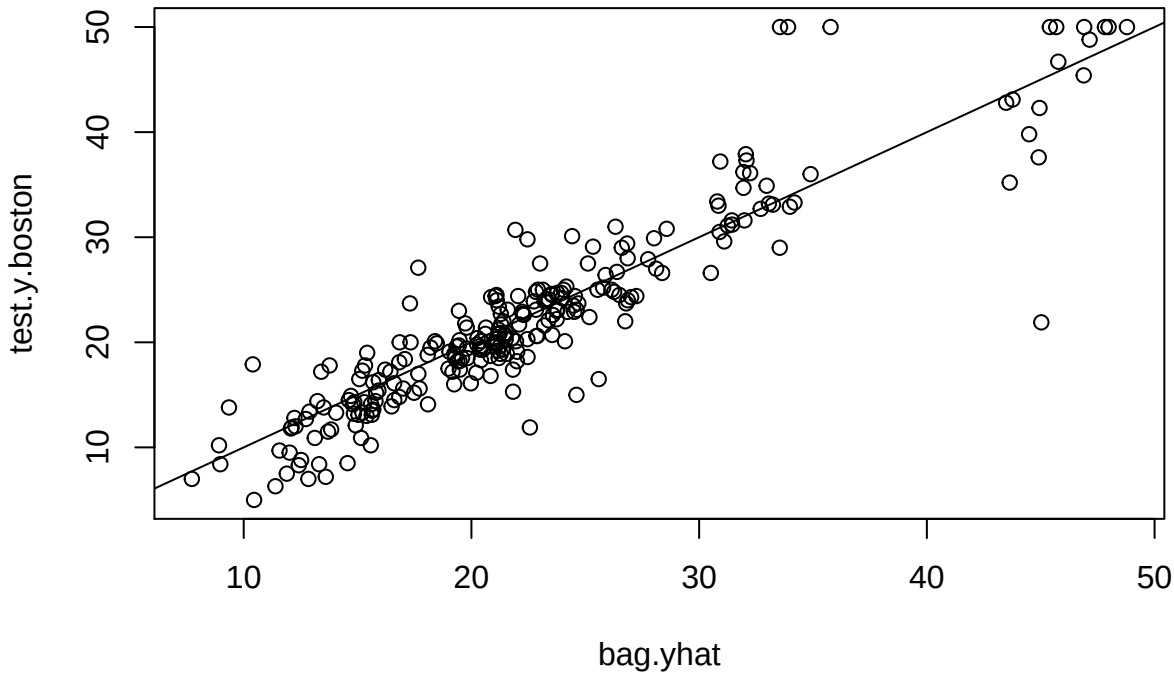
estimate bagging:

```
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: 10.63061
##                     % Var explained: 87.13
```

evaluate bagging:

```
bag.yhat = predict(bag.boston, newdata = Boston[test, ])
test.y.boston = Boston[test, "medv"]
plot(bag.yhat, test.y.boston)
abline(0, 1)
```



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

```
## [1] 13.36681
```

The minimal change with the book results are due to the change in the versions.

estimate rf:

```
rf.boston = randomForest(medv ~ ., data = Boston, subset = train,
                         mtry = 6, importance = TRUE, ntree = 500)
rf.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = Boston, mtry = 6, importance = TRUE,      ntree = 500, subs
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 11.64323
##                     % Var explained: 85.9
```

The default for m, i.e. `mtry`, is $p/3$ for regression and $\sqrt(p)$ for classificatioon. `ntree` is put equal to its default to allow comparability with the book's results.

evaluate rf:

```
yhat.rf = predict(rf.boston, newdata = Boston[test, ])
mean((yhat.rf - test.y.boston)^2)
```

```
## [1] 11.42667
```

Again, there is a small difference with the book. Random forests improves accuracy.

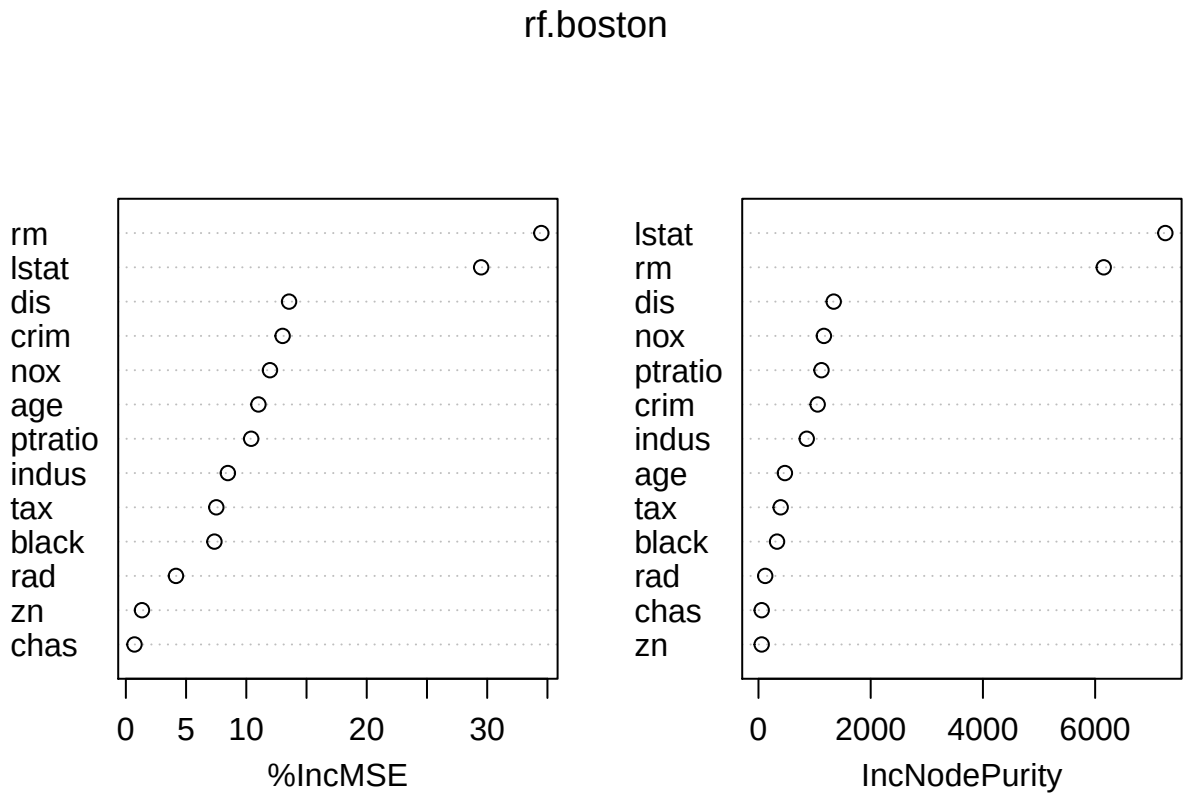
variable imporatnce in rf:

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

```
##           %IncMSE IncNodePurity
## crim    13.0153004    1054.33145
## zn       1.3369584      56.77008
## indus    8.4686600     860.06316
## chas     0.7160271      57.32146
## nox     11.9639834    1165.75054
## rm      34.4934544    6153.31987
## age     11.0067979     470.87378
## dis     13.5510912    1340.66107
## rad      4.1608450     119.80562
## tax      7.5113948     395.23395
## ptratio 10.4038148    1123.02755
## black    7.3529325     330.86874
## lstat   29.4991884    7250.83778
```

The first column shows a measure of importance, which uses out-of-bag data. Hence, it does not use the training data and can be thought of as an estimator for percentage increase in test MSE. In contrast, the second column,uses the training data.
%IncMSE captures the average increase in the out-of-bag MSE, when a given variable is exluded from the model. This measure was not mentioned in the chapter.
The second measure, IncNodePurity, is the measure of importance introduced in the chapter, which captures the total increase in node impurity.

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



The economic status of the community and the size of houses are by far the most important predictors.

Summary

- the library's name, `randomForest` is not plural
- no `summary()` for random forest fit
- new arguments for fit here are `mtry`, `importance` and `ntree`
- `varImpPlot` to plot the variable importance

Boosting

The theory behind boosting is not covered much in ISLR. Therefore, some of the concepts we will explore here will lack the theoretical underpinnings.

fit:

```
library(gbm)
```

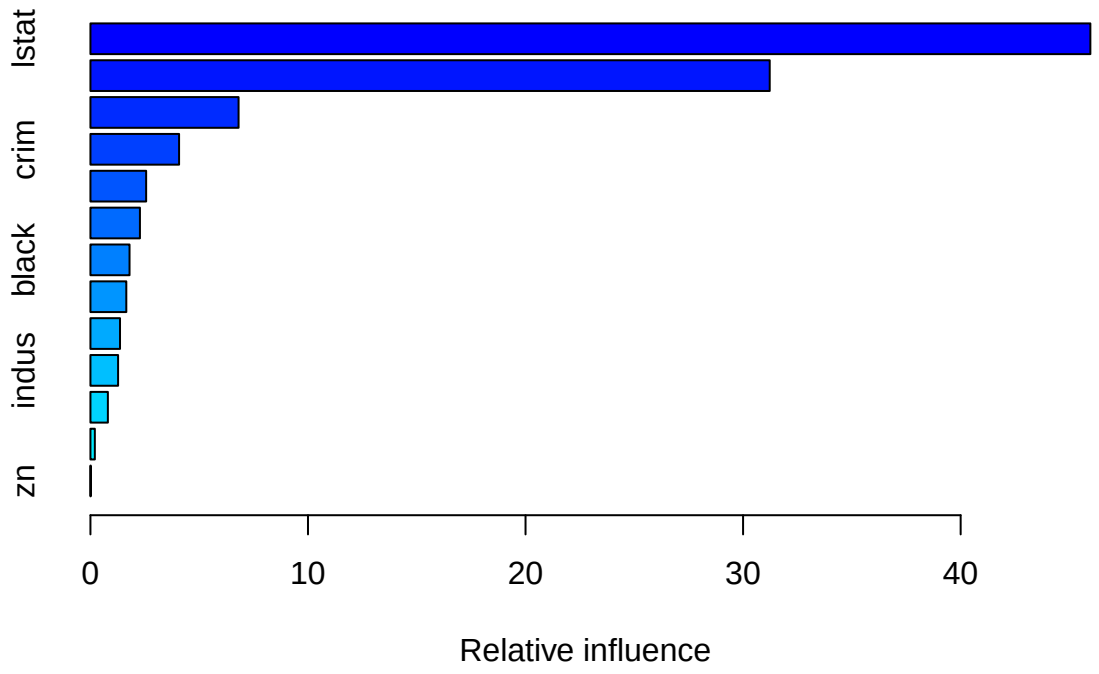
```
## Loading required package: survival
## Loading required package: lattice
## Loading required package: splines
## Loading required package: parallel
```

```
## Loaded gbm 2.1.3
```

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

relative importance:

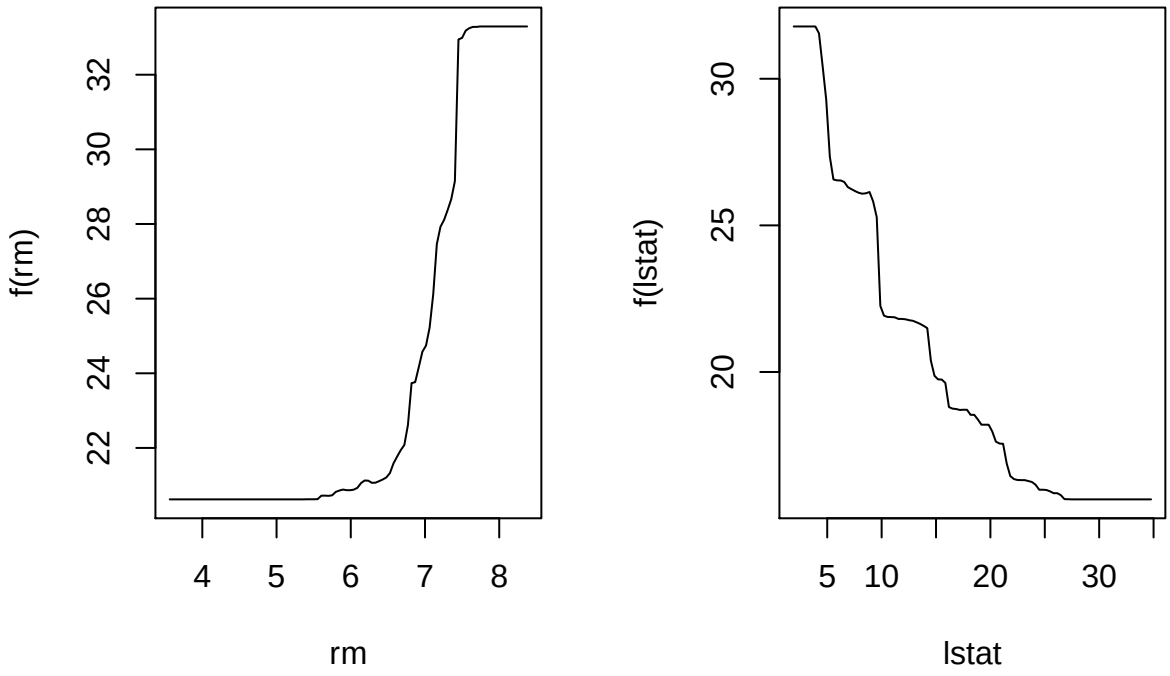
```
summary(boost.boston)
```




```
##             var    rel.inf
## lstat     lstat 45.9627334
## rm           rm 31.2238187
## dis         dis  6.8087398
## crim       crim  4.0743784
## nox         nox  2.5605001
## ptratio ptratio  2.2748652
## black     black  1.7971159
## age         age  1.6488532
## tax         tax  1.3595005
## indus     indus  1.2705924
## chas       chas  0.8014323
## rad         rad  0.2026619
## zn           zn  0.0148083
```

partial dependence plots:

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



These partial dependence plots show the marginal effects of the variables `rm` and `lstat` at the average of other variables, i.e. after integrating out all other variable, except the single one for which the plot is drawn. The interpretation for these plots would be similar to the one for linear regression coefficients, enabling the "having other variables fixed" interpretation; as expected, in least squares regression, partial dependence plots would look like straight lines.

test error:

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

```
## [1] 11.84434
```

different learning speed:

```
boost.boston = gbm(medv ~ ., data = Boston[train, ], interaction.depth = 4,
                   distribution = "gaussian", n.trees = 5000, shrinkage = 0.2,
                   verbose = FALSE)
yhat.boost = predict(boost.boston, newdata = Boston[test, ],
                     n.trees = 5000)
mean((yhat.boost - test.y.boston)^2)
```

```
## [1] 11.51109
```

Summary

- The library is `gbm`
- We estimated the model, saw the relative influence statistics, and depicted partial dependence plots.
- `gbm` does not accept the argument `subset`
- `distribution`, `n.trees`, `shrinkage`, `interaction.depth` with default values `"bernoulli"`, `100`, `0.001`, `1`.
- Set the `distribution` argument to `"gaussian"` if regression and to `"bernouli"` if classification.
- Partial dependence plots can be drawn using `plot.gbm()` with the name of the variable specified in the argument `i.var`
- The argument `n.trees` in `predict.gbm` is used for specifying how many trees from the boosted sequence to use in prediction. As a result, it should be lower than the number used for fitting the boosted model.