Stat 427/627 Statistical Machine Learning

In-class Lab 9: Tree Methods

Contents

1 Fitting and Pruning Classification Tree

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

```
# install.packages("tree")
library(tree)
```

1.1 The `Carseats` data

We first use classification trees to analyze the `Carseats` data set in `ISLR2` package. In these data, `Sales` is a continuous variable, and so we begin by recoding it as a binary variable. We use the `ifelse()` function to create a variable, called `High`, which takes on a value of `Yes` if the `Sales` variable exceeds 8, and takes on a value of `No` otherwise.

```
library(ISLR2)
names(Carseats)
```

```
##  [1] "Sales"       "CompPrice"   "Income"      "Advertising" "Population"
##  [6] "Price"       "ShelveLoc"   "Age"         "Education"   "Urban"
## [11] "US"
```

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

1.2 `tree()` function and its output.

We now use the `tree()` function to fit a classification tree in order to predict `High` using all variables but `Sales`. The syntax of the `tree()` function is quite similar to that of the `lm()` function.

```
tree.carseats <- tree(High ~ . - Sales, carseat.data)  # All predictors except for Sales
```

1.2.1 The `summary()` function lists the variables that are used as internal nodes in the tree, the number of terminal nodes, and the (training) error rate.

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

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

- We see that the training error rate is 9%.
- For classification trees, the deviance reported in the output of `summary()` is given by

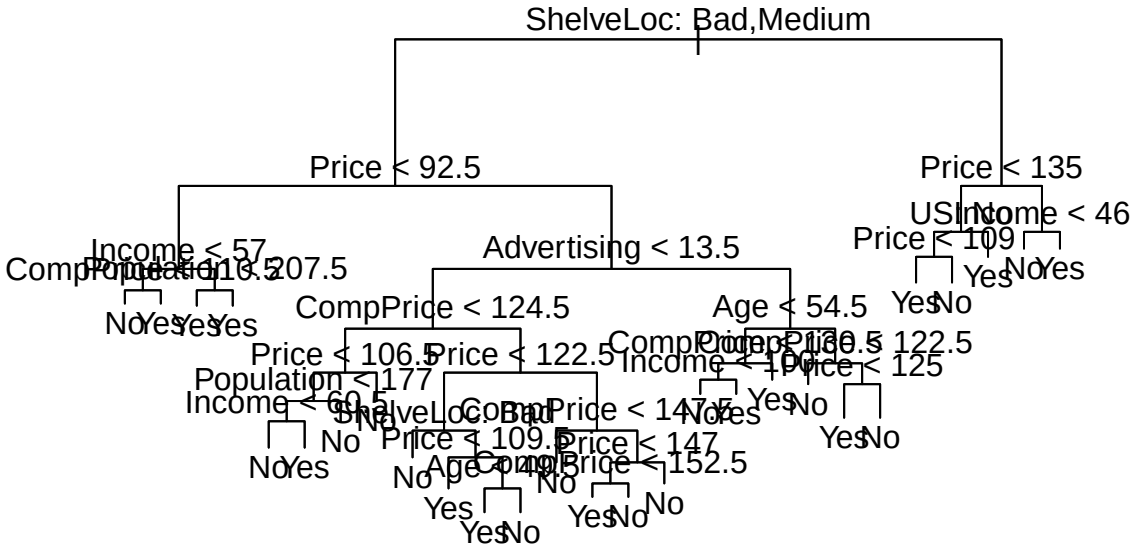
$$-2\sum_m\sum_k n_{mk}\log\hat{p}_{mk},$$

  where n_{mk} is the number of observations in the mth terminal node that belong to the kth 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 divided by $n-T $, i.e., (sample size) - (number of terminal nodes}). (In this case, 400-27=373.)
- **Caution:** The "Residual mean deviance" and "Misclassification error rate" on the training data may be deceptive since they are based on the training data.

1.2.2 Visualize a tree using `plot()` and `text()` function.

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.

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



- The most important indicator of `Sales` appears to be shelving location, since the first branch differentiates `Good` locations from `Bad` and `Medium` locations.

1.2.3 Other details about the tree.

```
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 the above table, we can find:

- The split criterion. E.g. `Price < 92.5`, `Income < 57`, `CompPrice < 110.5`.
- The number of observations in that branch. E.g, 46 observations at node (4).
- The overall prediction for the branch (`Yes` or `No`) by majority vote.
- The fraction of observations in that branch that take on values of `No` and `Yes` (in alphabetical or numerical order). E.g., node (4) has 46 observation: 14 `No` ($14/46 = 0.3043$), 32 `Yes` ($32/46 = 0.6957$).
- For classification tree, the deviance of each branch is calculated by the number of observations in that branch. E.g., for node (4),

$$deviance = -2 \cdot \log(Likelihood) = -2 \cdot (14 \log(14/46) + 32 \log(32/46)) = 56.53$$

- Terminal nodes (aka. leaves) are marked with `*`.

1.2.4 `predict()` the classes.

Use the `predict()` function to predict the responsive, given a set of values of the predictors.

- In the case of a classification tree, the argument `type = "class"` instructs `R` to return the class prediction. Majority vote approach is used, with random draw in case of a tie.

```
set.seed(2023)
train <- sample(1:nrow(carseat.data), 200)
carseat.train <- carseat.data[train, ]
carseat.test <- carseat.data[-train, ]
High.test <- carseat.data$High[-train]

traintree.carseats <- tree(High ~ . - Sales, carseat.data, subset = train)

traintree.pred <- predict(traintree.carseats, newdata=carseat.train, type = "class")
testtree.pred <- predict(traintree.carseats, newdata=carseat.test, type = "class")

table(traintree.pred, carseat.train$High) # Confusion matrix on training data
```

```
##
## traintree.pred  No Yes
##            No  112  11
##            Yes  12  65
```

```
table(testtree.pred, High.test) # Confusion matrix on testing data
```

```
##              High.test
## testtree.pred No Yes
##           No  78  21
##           Yes 34  67
```

```
# Classifications rate
noquote(paste("Misclassfication rate on the TRAINING data:",
              mean(traintree.pred != carseat.train$High)))
```

```
## [1] Misclassfication rate on the TRAINING data: 0.115
```

```
noquote(paste("Misclassfication rate on the TESTING data:",
              mean(testtree.pred != High.test)))
```

```
## [1] Misclassfication rate on the TESTING data: 0.275
```

- Recall the discussion on model accuracy using training data vs. testing data.
- Recall the discussion on the bias-variance trade-off and the model flexibility.
- Use cross-validation to prune the tree and determine the optimal tree size.

1.3 Pruning the tree

1.3.1 Determining the tree size using cross-validation

The function `cv.tree()` performs cross-validation in order to determine the optimal level of tree complexity.

- Use the argument `FUN = prune.misclass` in order to use the classification error rate to guide the cross-validation and pruning process. If not specified, the default default for the `cv.tree()` function is to use deviance.
- Cost complexity pruning (textbook, p.332, Expression 8.4) is used in order to select a sequence of trees for consideration.
- 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 (8.4)).

- Since I use `FUN = prune.misclass` in the following code, despite its name, `dev` in the output corresponds to the number of cross-validation errors.
- Note the the results from 10-fold CV may vary since the algorithem involves random sampling.

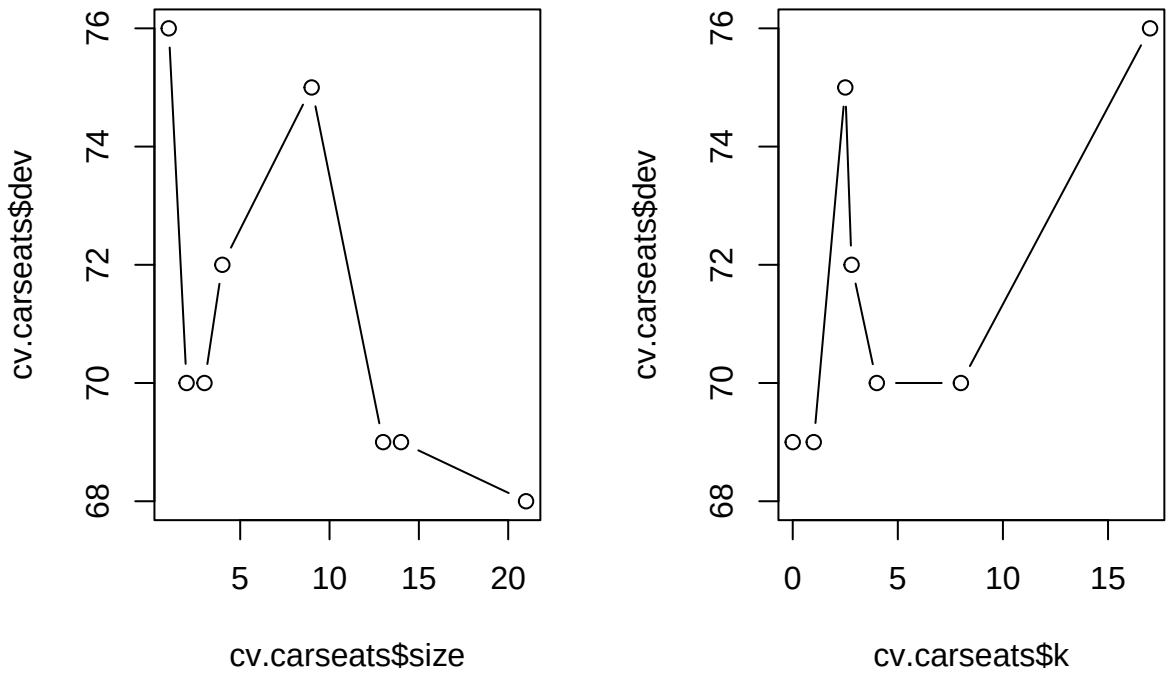
```
set.seed(2023)
cv.carseats <- cv.tree(traintree.carseats, FUN = prune.misclass, K=10)
names(cv.carseats)
```

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

```
cv.carseats
```

```
## $size
## [1] 21 14 13  9  4  3  2  1
##
## $dev
## [1] 68 69 69 75 72 70 70 76
##
## $k
## [1] -Inf  0.0  1.0  2.5  2.8  4.0  8.0 17.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

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

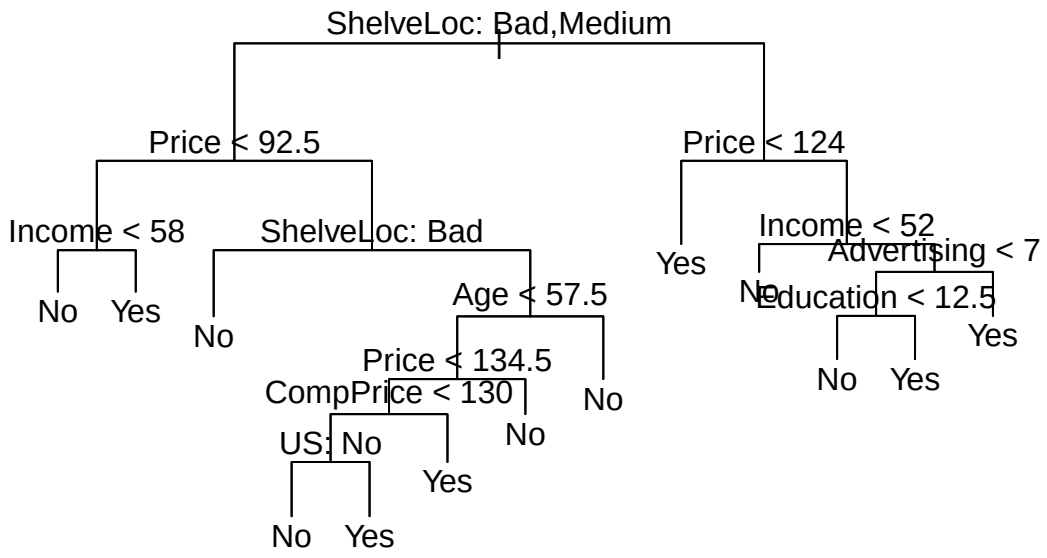


- Though the 10-fold CV shows the unpruned tree (with 21 nodes) has the least error classification (68 misclassifications), we can prune the tree to 13 nodes since the tree is much smaller and produces similar number of misclassfications in cross-validation.

1.3.2 Pruning the tree to the optimal size

We now apply the `prune.misclass()` function in order to prune the tree based on misclassification rate.

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




```
tree.pred <- predict(prune.carseats, carseat.test, type = "class")
noquote(paste("Misclassfication rate on the TESTING data using pruned tree:", mean(tree.pred != High.te
```

```
## [1] Misclassfication rate on the TESTING data using pruned tree: 0.255
```

- Recall the misclassifcation rate on the testing data using unpruned tree was 27.5%.
- If we increase the value of `best`, we obtain a larger pruned tree with lower classification accuracy:

```
prune.carseats <- prune.misclass(traintree.carseats, best = 18)
tree.pred <- predict(prune.carseats, carseat.test,
    type = "class")
noquote(paste("Misclassfication rate on the TESTING data using bigger tree:", mean(tree.pred != High.te
```

```
## [1] Misclassfication rate on the TESTING data using bigger tree: 0.28
```

2 Fitting and Pruning Regression Trees

2.1 Data: Boston housing values

Here we fit a regression tree to the `Boston` data set in `ISLR2`. We'll set aside 200 observations as the hold-out testing data. Note that those 200 observations will not be used in cross-validation. They will be treated as the "benchmark" to test the performance of the trees at the end.

```
set.seed(2023)
names(Boston)
```

```
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "lstat"   "medv"
```

```
test <- sample(1:nrow(Boston), 200)
boston.train <- Boston[-test, ]
boston.test <- Boston[test, ]
```

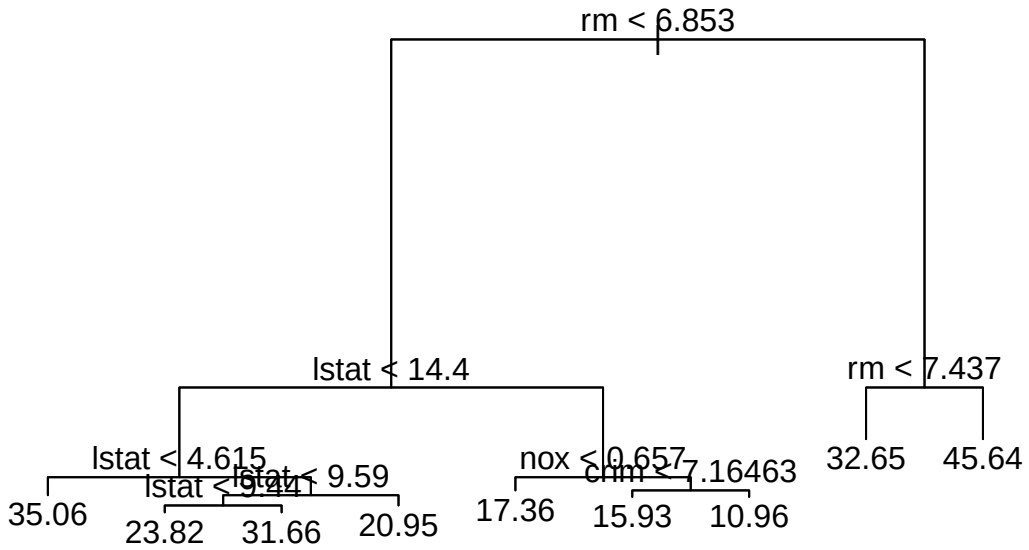
2.2 Fitting a regression tree

```
tree.boston <- tree(medv ~ ., data=boston.train)
summary(tree.boston)
```

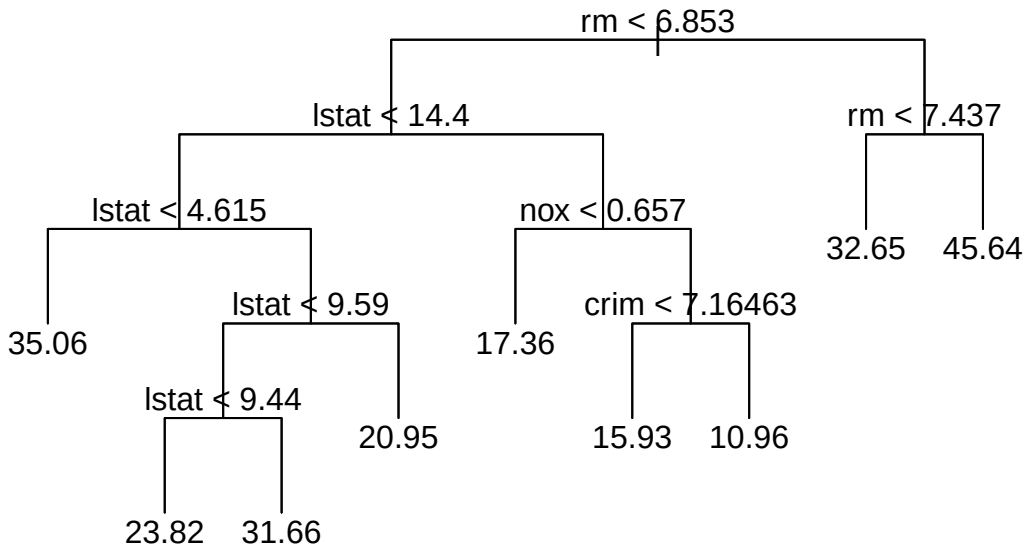
```
##
## Regression tree:
## tree(formula = medv ~ ., data = boston.train)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "nox"   "crim"
## Number of terminal nodes:  9
## Residual mean deviance:  12.93 = 3841 / 297
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.4400  -2.1530  -0.1345   0.0000   1.9280  18.3400
```

- Only four 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 $\sum (Y_i - \widehat{Y}_i)^2$.

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



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



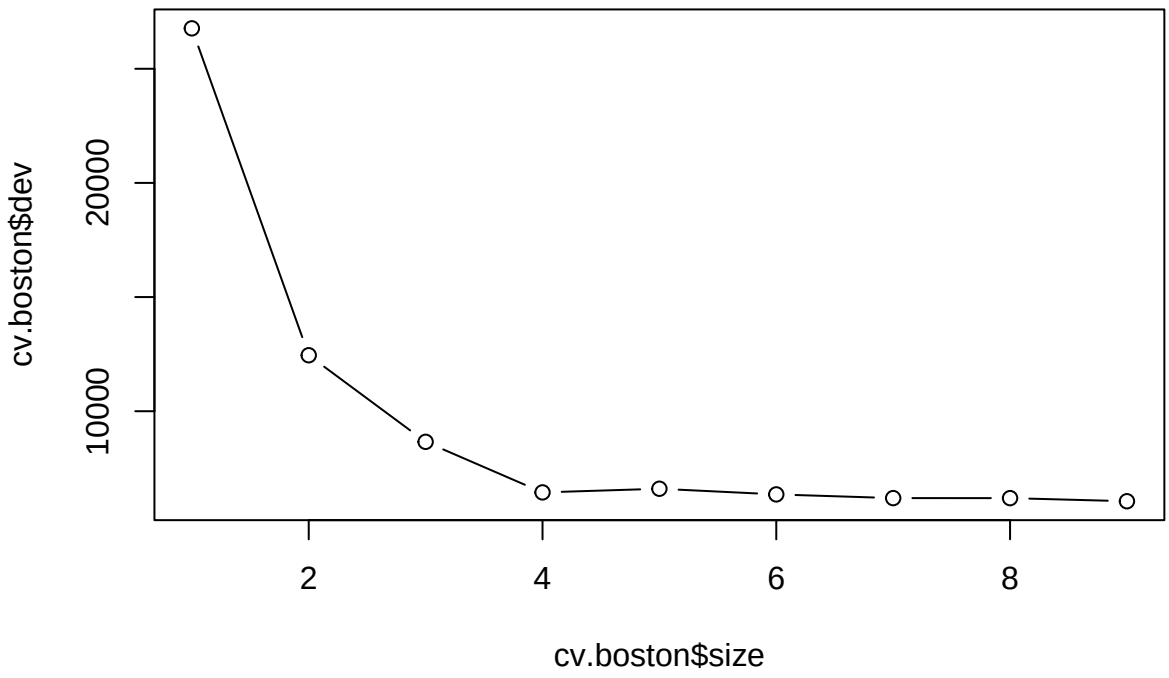
- The variable `lstat` measures the percentage of individuals with *lower* socioeconomic status. The variable `rm` corresponds to the average number of rooms. The tree indicates that larger values of `rm`, or lower values of `lstat`, correspond to more expensive houses.
- For example, the tree predicts a median house price of $\$45,640$ for homes in census tracts in which $rm \geq 7.437$. The tree predicts a median house price of $\$35,060$ for homes in census tracts in which $rm < 6.853$ and $lstat < 4.615$.
- It is worth noting that we could have fit a much bigger tree, by passing `control = tree.control(nobs = length(train), mindev = 0)` into the `tree()` function.

2.3 Pruning the tree.

2.3.1 Determining the tree size using cross-validation

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

```
set.seed(2023)
cv.boston <- cv.tree(tree.boston, K=10)
plot(cv.boston$size, cv.boston$dev, type = "b")
```




```
cv.boston
```

```
## $size
## [1] 9 8 7 6 5 4 3 2 1
##
## $dev
## [1]  6057.679  6194.328  6194.688  6357.571  6605.092  6444.781  8660.108
## [8] 12451.770 26769.779
##
## $k
## [1]       -Inf   268.3272   285.7027   424.6813   546.1001   744.9931  2140.9020
## [8]  3717.3208 14415.2842
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

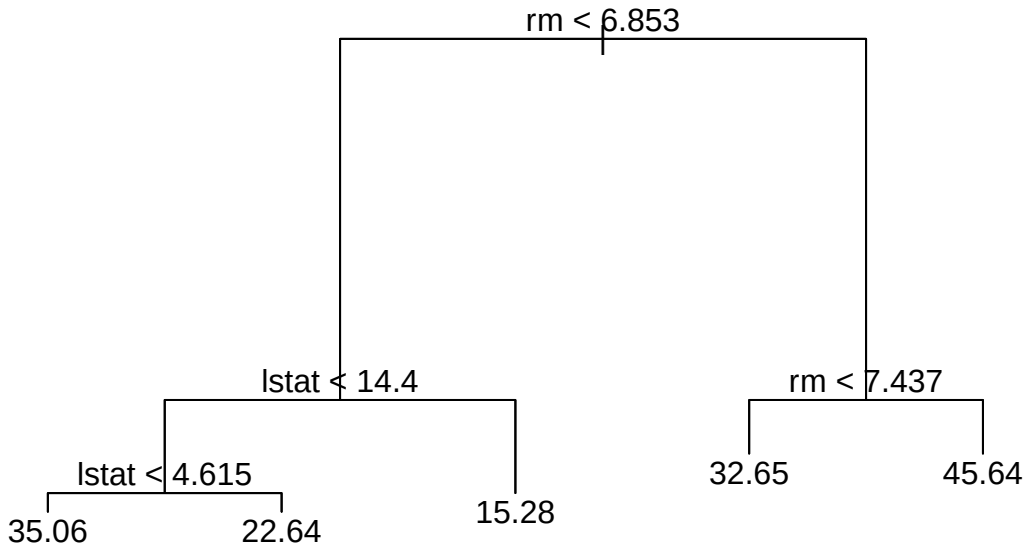
- In this case, the initial most complex tree with 9 nodes has the smallest deviation by cross-validation.

2.3.2 Pruning the tree to the optimal size

Use the `prune.tree()` function to prune the tree when needed. (Recall that for classification tree, use `prune.misclass()`.)

```
prune.boston <- prune.tree(tree.boston, best = 5)
plot(prune.boston)
```

```
text(prune.boston, pretty = 0)
```



3 Summary of coding difference between regression tree and classification tree.

The **regression** tree uses the default "deviance" $\sum(\widehat{Y}_i - \widehat{Y}_i)^2$ to measure the model accuracy.

- Can use the default argument in `cv.tree()`, `predict()`.
- Use `prune.tree()` for pruning.

The **classification** tree usually use "misclassification rate" as the accuracy metric.

- Make sure the response variable is a factor.
- In function `cv.tree()`, set argument `FUN = prune.misclass`.
- For pruning, use function `prune.misclass()`
- In function `predict()`, set `type = "class"`

4 Bagging and Random Forests

We will keep using the `Boston` (training) data.

Function `randomForest()` in the `randomForest` package conducts both bagging and random forest.

```
randomForest(formula, data =  , mtry =  , ntree =  , ...)
```

- For bagging, set the argument `mtry =` to the number of all predictors for bagging. (Recall that bagging is a special case of a random forest with $m = p$.)
- For random forest, by default, the function uses $m = p/3$ for regression tree, and $m = \sqrt{p}$ for classification tree. You can use argument `mtry =` to choose a different m for random forest. (Recall p is the total number of predictors, m is the number of predictors to be considered at each split.)
- By default, 500 trees will be used. (Argument `ntree =`)
- The exact results obtained may depend on the versions of `R`, the version of `randomForest` package, and the random seed.

4.1 Bagging

Recall there are 12 predictors.

```
library(randomForest)
set.seed(2023)
bag.boston <- randomForest(medv ~ ., data = boston.train,
                           mtry = 12, importance = TRUE)
bag.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston.train, mtry = 12,      importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 12
##
##           Mean of squared residuals: 11.82295
##                     % Var explained: 86.29
```

4.2 Random Forest

As an example, I'll use `mtry = 5` below. By default, at each split, `randomForest()` uses $p/3$ variables for regression trees, and $\sqrt{p}$ variable for classification trees.

```
rf.boston <- randomForest(medv ~ ., data = boston.train, mtry = 5,
                          importance = TRUE)
rf.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston.train, mtry = 5,      importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 11.60879
##                     % Var explained: 86.54
```

4.3 Variable importance

Use the `importance()` and `varImpPlot()` functions to view importance of each variable. For regression tree, we have two measures:

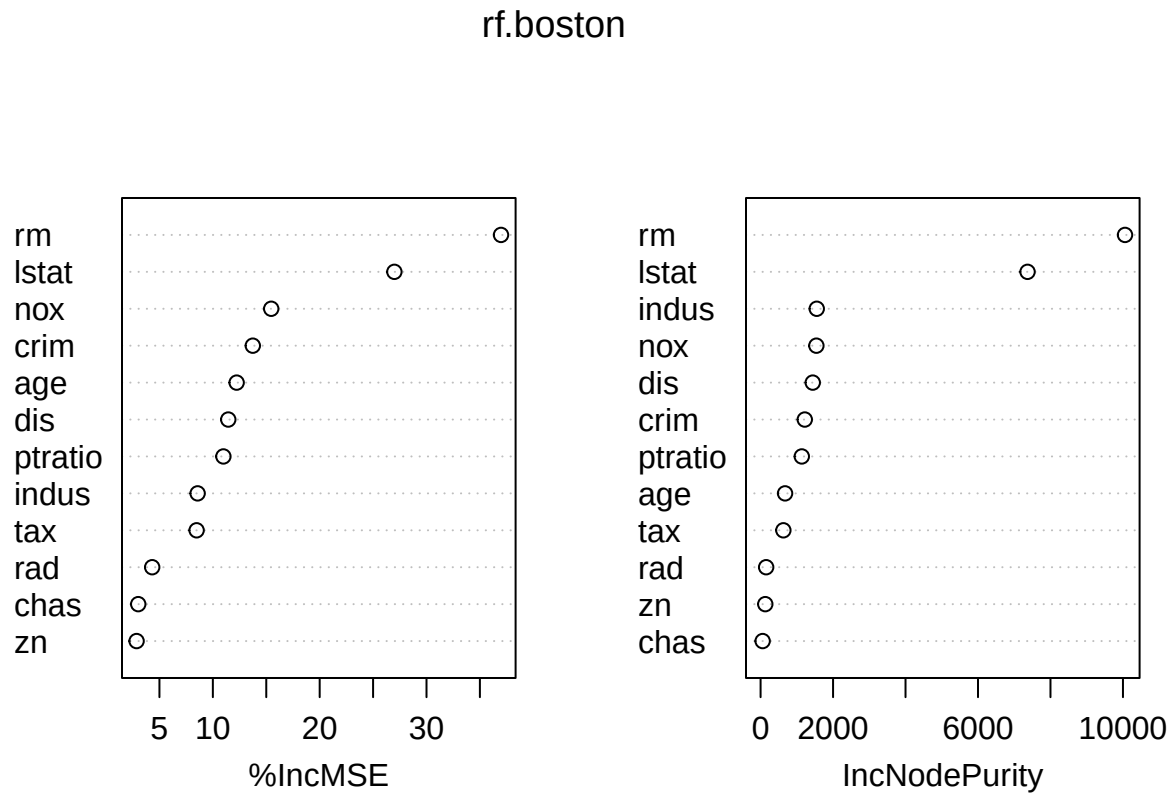
- `%IncMSE`: averaging over all trees, how much a predictor is associated with reducing the MSE on the out-of-bag samples? Large value indicate important predictor.

- `IncNodePurity`: averaging over all trees, how much a predictor is associated with reducing the module purity on the training (in-bag) samples? Large value indicate important predictor.

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

```
##           %IncMSE IncNodePurity
## crim    13.738627    1219.31706
## zn       2.859644     130.06439
## indus    8.573697    1550.34916
## chas     3.015253      56.57057
## nox     15.464497    1539.83344
## rm      36.976791   10057.02054
## age     12.227347     675.94060
## dis     11.443556    1439.45399
## rad      4.321137     153.96643
## tax      8.479898     628.71632
## ptratio 10.985986    1136.94658
## lstat   26.993054    7367.09899
```

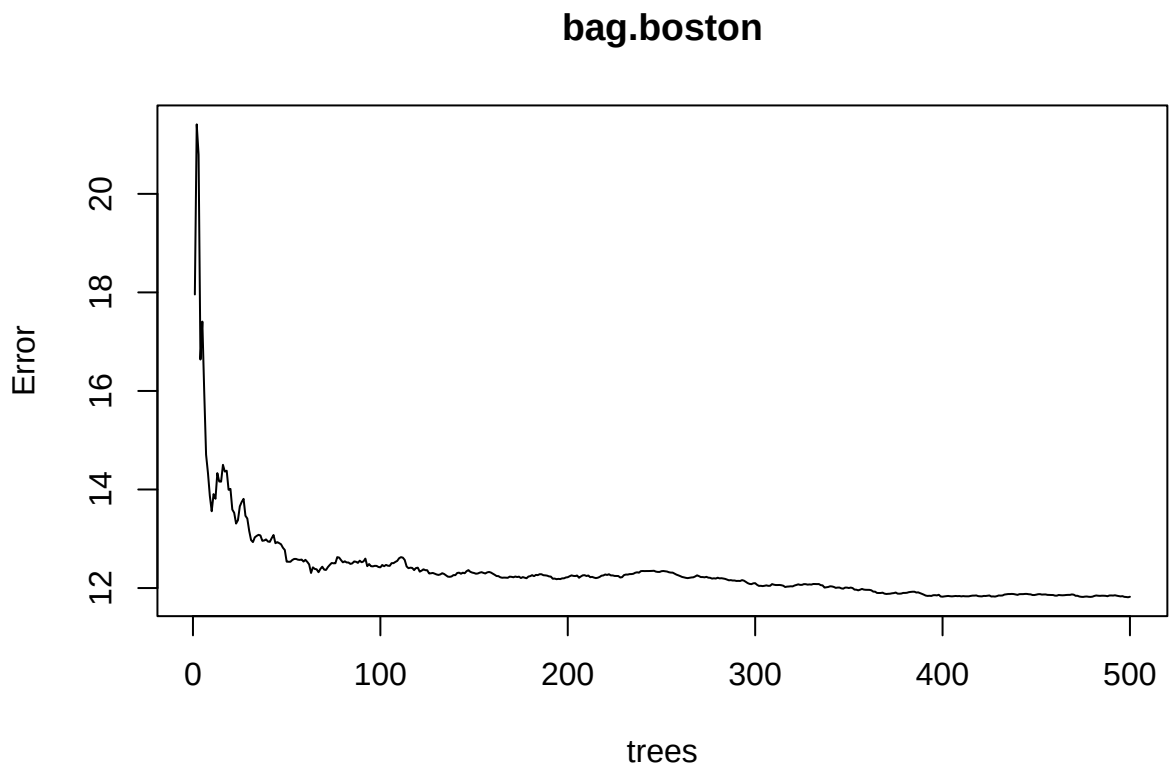
```
varImpPlot(rf.boston)
```



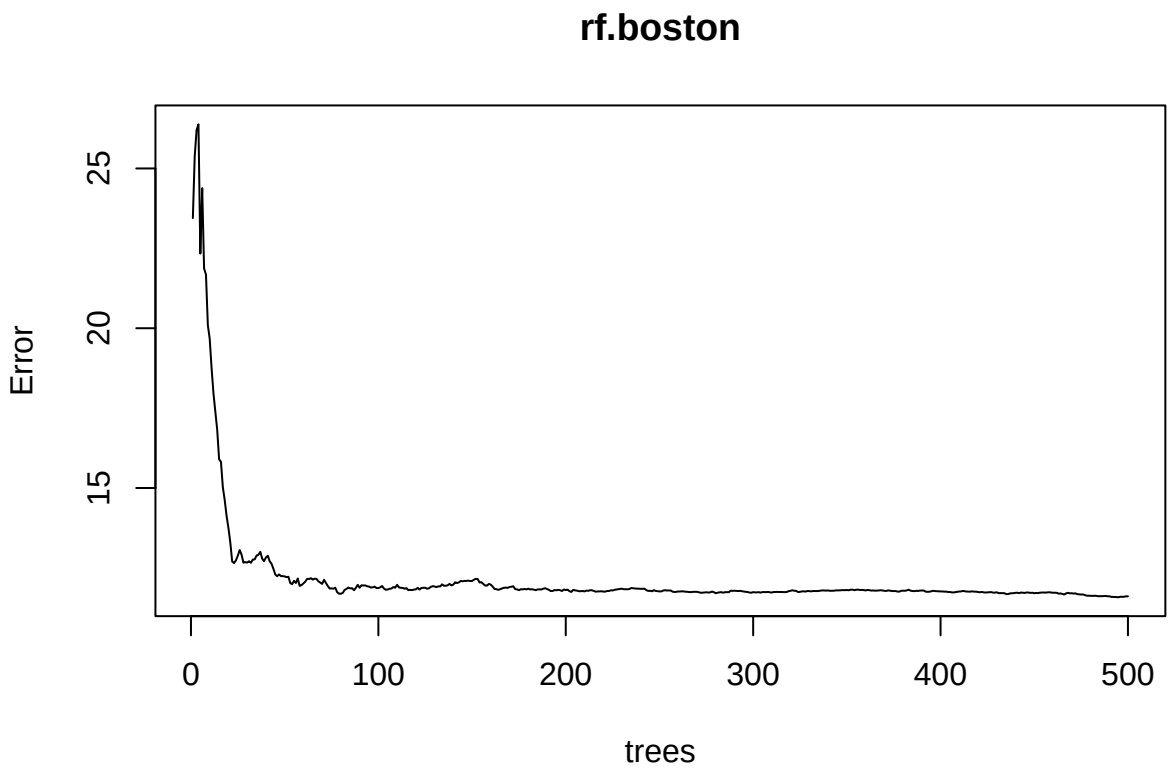
- The results indicate that across all of the trees considered in the random forest, the wealth of the community (`lstat`) and the house size (`rm`) are by far the two most important variables.

4.4 How many trees?

```
plot(bag.boston)
```




```
plot(rf.boston)
```



5 Boosting

Function `gbm()` in package `gbm` conducts boosting to regression and classification tree.

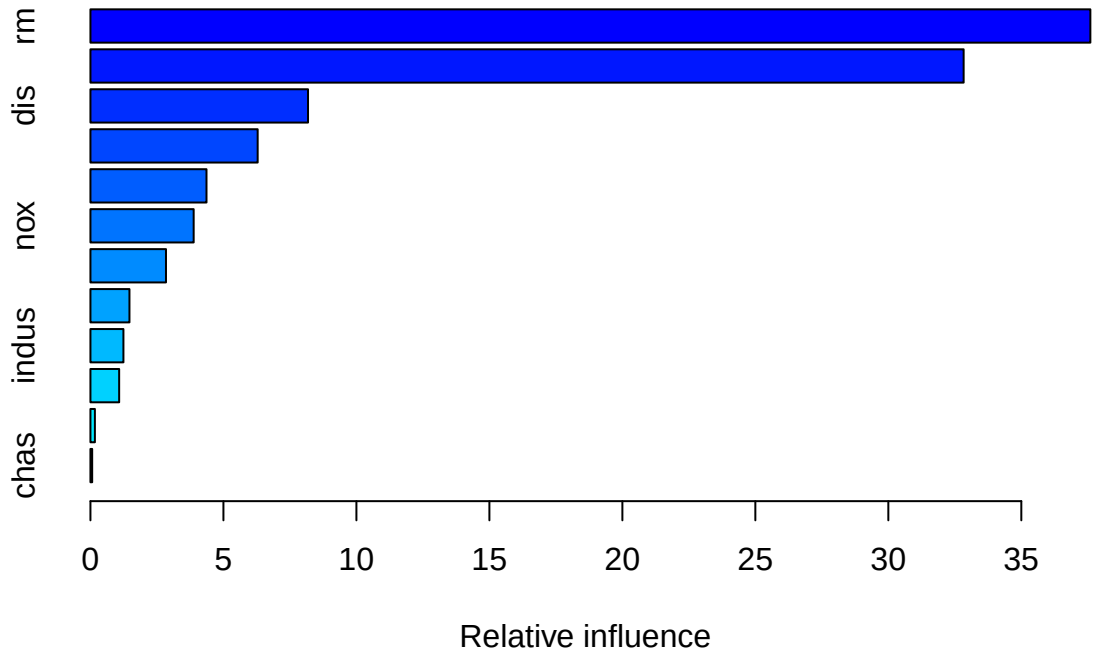
```
# install.packages("gbm")
library(gbm)
# ? gbm
```

- For regression tree, run `gbm()` with argument `distribution = "gaussian"`.
- For binary classification tree, use argument `distribution = "bernoulli"`.
- B: the number of trees is set by argument `n.trees =`. (100 by default.)
- d: the depth of each tree is limited by argument `interaction.depth =`. (1 by default.)
- λ: the shrinkage parameter is set by `shrinkage =`. (0.1 by default.)

```
set.seed(2023)
boost.boston <- gbm(medv ~ ., data = boston.train,
    distribution = "gaussian", n.trees = 500,
    interaction.depth = 4, shrinkage = 0.1)
```

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

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

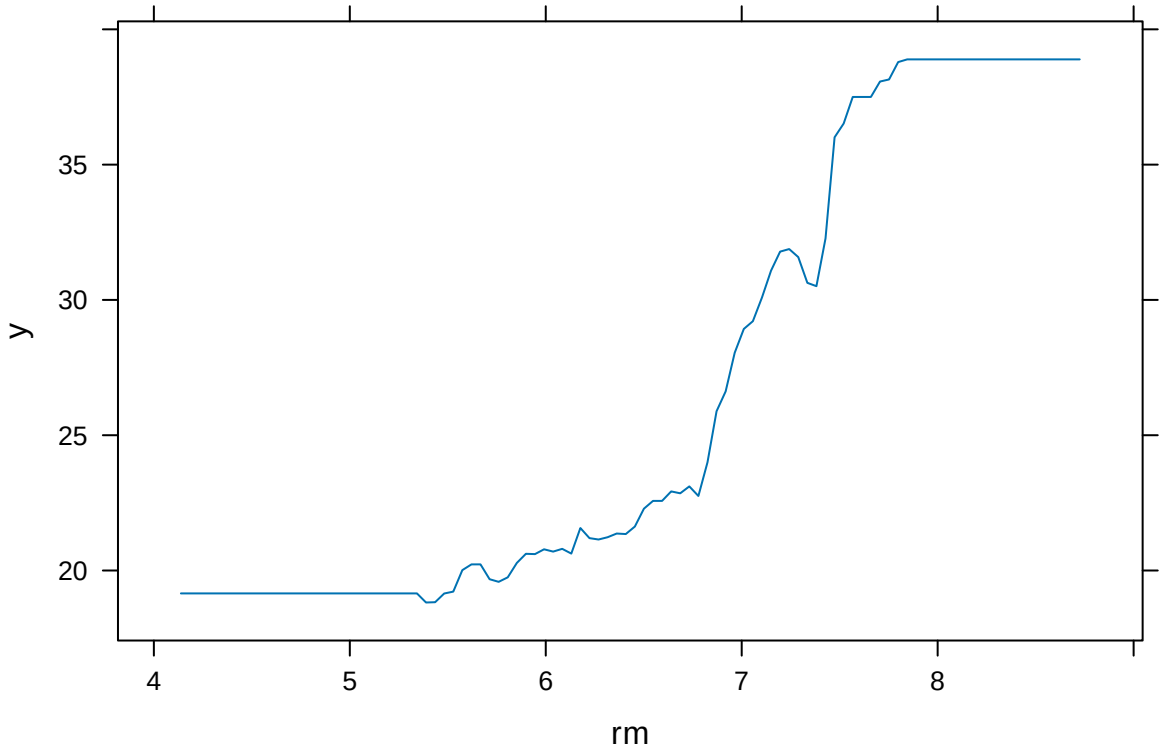



```
##             var     rel.inf
## rm           rm 37.59555830
## lstat     lstat 32.83130851
## dis         dis  8.18037266
## crim       crim  6.28738582
## age         age  4.36388462
## nox         nox  3.88101683
## ptratio ptratio  2.84050426
## tax         tax  1.46697133
## indus     indus  1.24046052
## rad         rad  1.08016800
## zn           zn  0.16741161
## chas       chas  0.06495754
```

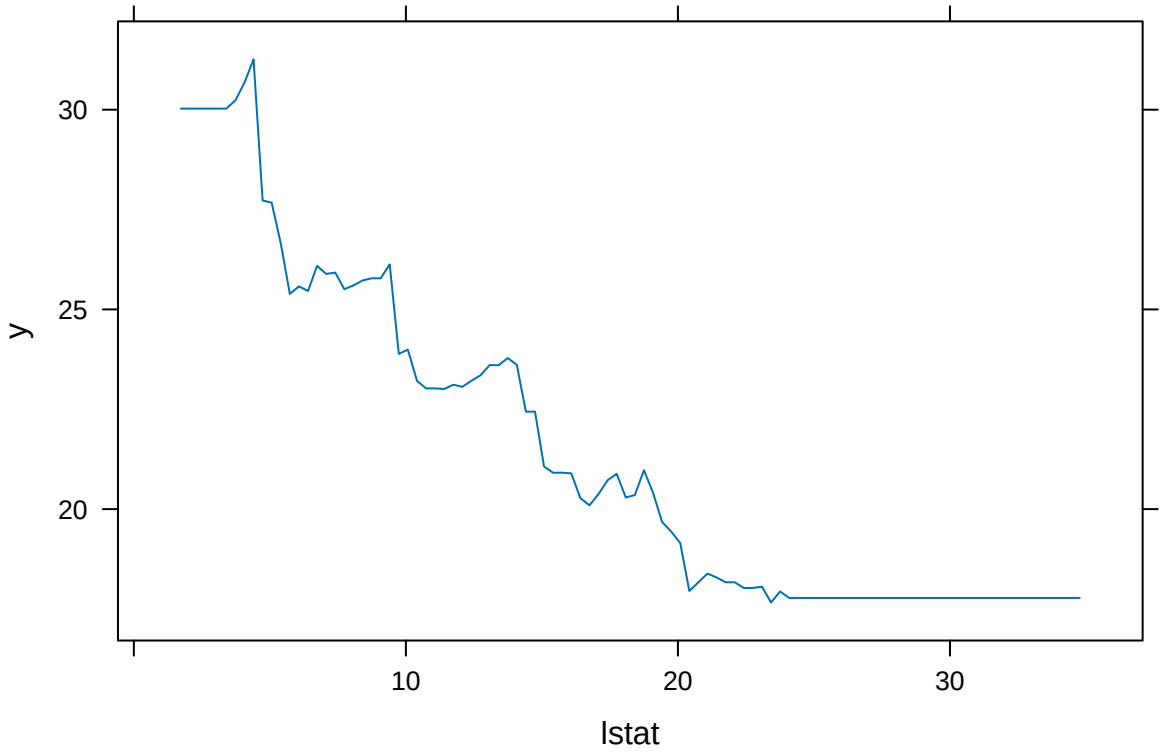
- We see that `lstat` and `rm` are by far the most important variables.

The *partial dependence plots* illustrate the marginal effect of the selected variables on the response after *integrating* out the other variables.

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




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



- In this case, as we might expect, median house prices are increasing with `rm` and decreasing with `lstat`.

6 The prediction accuracy

Recall that prediction accuracy should be assessed using either a testing set, or via cross-validation.

So far, we fitted the following 5 models to the `Boston` data:

```
summary(tree.boston)
```

```
##
## Regression tree:
## tree(formula = medv ~ ., data = boston.train)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "nox"   "crim"
## Number of terminal nodes:  9
## Residual mean deviance:  12.93 = 3841 / 297
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.4400  -2.1530  -0.1345   0.0000   1.9280  18.3400
```

```
summary(prune.boston)
```

```
##
## Regression tree:
## snip.tree(tree = tree.boston, nodes = c(9L, 5L))
## Variables actually used in tree construction:
## [1] "rm"    "lstat"
```

```
## Number of terminal nodes:  5
## Residual mean deviance:  17.83 = 5366 / 301
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.4400  -2.5450  -0.1448   0.0000   2.2470  27.3600
```

```
bag.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston.train, mtry = 12,      importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 12
##
##           Mean of squared residuals: 11.82295
##                     % Var explained: 86.29
```

```
rf.boston
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston.train, mtry = 5,      importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 5
##
##           Mean of squared residuals: 11.60879
##                     % Var explained: 86.54
```

```
boost.boston
```

```
## gbm(formula = medv ~ ., distribution = "gaussian", data = boston.train,
##     n.trees = 500, interaction.depth = 4, shrinkage = 0.1)
## A gradient boosted model with gaussian loss function.
## 500 iterations were performed.
## There were 12 predictors of which 12 had non-zero influence.
```

We set aside a testing data set with 200 observations at the beginning. Estimate the MSE using the hold-out testing.

```
cbind(nrow(Boston), nrow(boston.train), nrow(boston.test))
```

```
##      [,1] [,2] [,3]
## [1,]  506  306  200
```

```
tree.pred <- predict(tree.boston, newdata=boston.test)
tree.mse <- mean((boston.test$medv - tree.pred)^2)

prune.pred <- predict(prune.boston, newdata=boston.test)
prune.mse <- mean((boston.test$medv - prune.pred)^2)

bag.pred <- predict(bag.boston, newdata=boston.test)
bag.mse <- mean((boston.test$medv - bag.pred)^2)

rf.pred <- predict(rf.boston, newdata=boston.test)
rf.mse <- mean((boston.test$medv - rf.pred)^2)
```

```r
boost.pred <- predict(boost.boston, newdata=boston.test, n.trees = 500)
boost.mse <- mean((boston.test$medv - boost.pred)^2)

rbind(tree.mse, prune.mse, bag.mse, rf.mse, boost.mse)
```

```
##                [,1]
## tree.mse  28.71702
## prune.mse 32.71612
## bag.mse   17.44483
## rf.mse    12.42456
## boost.mse 15.51027
```