Stats 401: Homework 2

Yuandong David Sun

10/27/2020

Introduction

In this homework assignment, I'll try to lead you through coding up algorithms for bootstrap, cross-validation, and tree-making from scratch. The hope is that by coding these algorithms by hand, you'll gain a deeper understanding of how these tools work.

1) Bootstrap

Bootstrapping refers to the technique of resampling our sample with replacement. This is equivalent to creating a "pseudo-population" of infinite duplicates of our sample and then drawing a random sample the same size as our original sample.

We often use bootstrap to get an estimate of the standard error (the standard deviation of the sampling distribution) of a statistic when we do not have access to the actual population.

In this example, we will try to estimate the standard error of the correlation of a sample.

We will begin with a synthetic population.

```r
set.seed(1)  # for reproducibility
# x is 10,000 random normal values, centered at 100, sd 10, rounded to 1 decimal place
x <- round(rnorm(10^4, 100, 10), 1)
# y is created to be correlated with x, but with error
y <- x + round(rnorm(10^4, 0, 5), 1)
# the correlation between x and y in the population is 0.8986
cor(x, y)
```

```
## [1] 0.8986356
```

```r
# plot(x,y) # optional if you want to see the plot
```

The population we created has a correlation of 0.8986. In real life, we would not know what the population actually looks like, and this value of ρ would be unknown to us.

```r
# we randomly sample 100 of these values to create our sample
samp <- sample(10^4, 100)
x_samp <- x[samp] # we subset x and y accordingly
y_samp <- y[samp]
cor(x_samp, y_samp)  # the correlation in our sample is 0.86296
```

```
## [1] 0.8983157
```

We have a sample of 100 values. The correlation between x and y in our sample is 0.86296. Without knowing what the population looks like, we may wish to estimate the standard error of this value.

One way we can do this is via bootstrap resampling. To perform bootstrap resampling, we resample our sample with replacement. In our case, there are 100 observations in our sample. So we will sample the integers 1:100 with replacement to choose which values will be in our bootstrap sample.

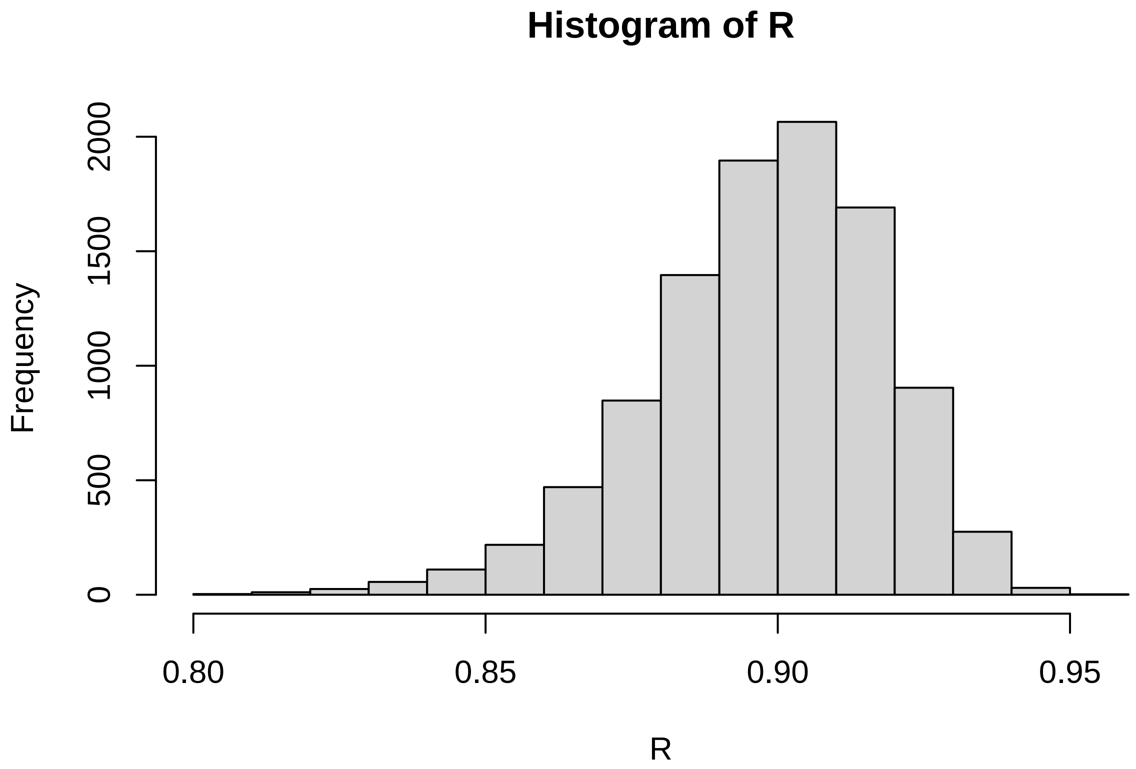
We subset our current sample accordingly and calculate the resulting correlation.

we repeat this many times (say, 1000), and record the correlation for each bootstrap sample. This effectively builds up a sampling distribution of different values of the correlation *r*. When we take the standard deviation of all those values, we have an estimate of the standard error.

```r
# one instance
set.seed(1)
index <- sample(100, replace = TRUE)
x_boot <- x_samp[index]
y_boot <- y_samp[index]
cor(x_boot, y_boot)
```

```
## [1] 0.8761554
```

```r
# your turn...
# Perform 1000 replicates of bootstrap
# For each replicate, calculate the resulting correlation, and store that value.
# So everyone gets the same results, make a for loop, and
# include the line set.seed(i) before you perform the sampling
set.seed(1)
R <- rep(NA, 10000)
for(i in 1:10000){
  samp<-sample(10^4,100)
  x_samp<-x[samp]
  y_samp <-y[samp]
  R[i]<-cor(x_samp,y_samp)
}

hist(R)
```



```
sd(R)
```

```
## [1] 0.02003223
```

```
mean(R)
```

```
## [1] 0.897836
```

```
# after performing the replicates, produce a histogram of the different observed correlations
# Calculate the standard deviation of the correlation estimates
```

2) Cross-Validation

We can use cross-validation to evaluate the predictive performance of several competing models.

Again, we will manually implement cross-validation from scratch first, and then use the built-in function in R.

We will use the dataset `ironslag` from the package `DAAG` (a companion library for the textbook Data Analysis and Graphics in R).

The description of the data is as follows: The iron content of crushed blast-furnace slag can be determined by a chemical test at a laboratory or estimated by a cheaper, quicker magnetic test. These data were collected to investigate the extent to which the results of a chemical test of iron content can be predicted from a magnetic test of iron content, and the nature of the relationship between these quantities. [Hand, D.J., Daly, F., et al. (1993) **A Handbook of Small Data Sets**]

The `ironslag` data has 53 observations, each with two values - the measurement using the chemical test and the measurement from the magnetic test.

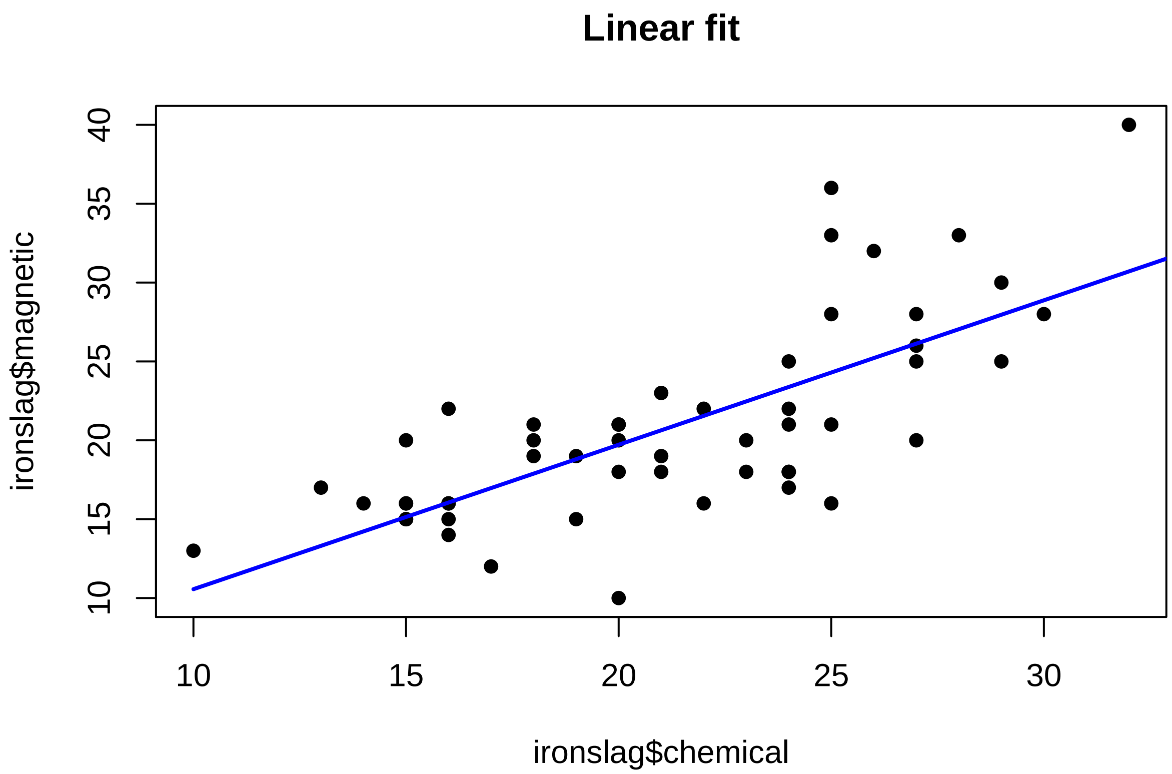
We can start by fitting a linear regression model $Y = \beta_0 + \beta_1 X + \epsilon$. A quick look at the scatterplot seems to indicate that the data may not be linear.

```
# install.packages("DAAG") # if necessary
library(DAAG)
```

```
## Warning: package 'DAAG' was built under R version 4.0.3
```

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

```
x <- seq(10,40, .1) # a sequence used to plot lines

L1 <- lm(magnetic ~ chemical, data = ironslag)
plot(ironslag$chemical, ironslag$magnetic, main = "Linear fit", pch = 16)
yhat1 <- L1$coef[1] + L1$coef[2] * x
lines(x, yhat1, lwd = 2, col = "blue")
```



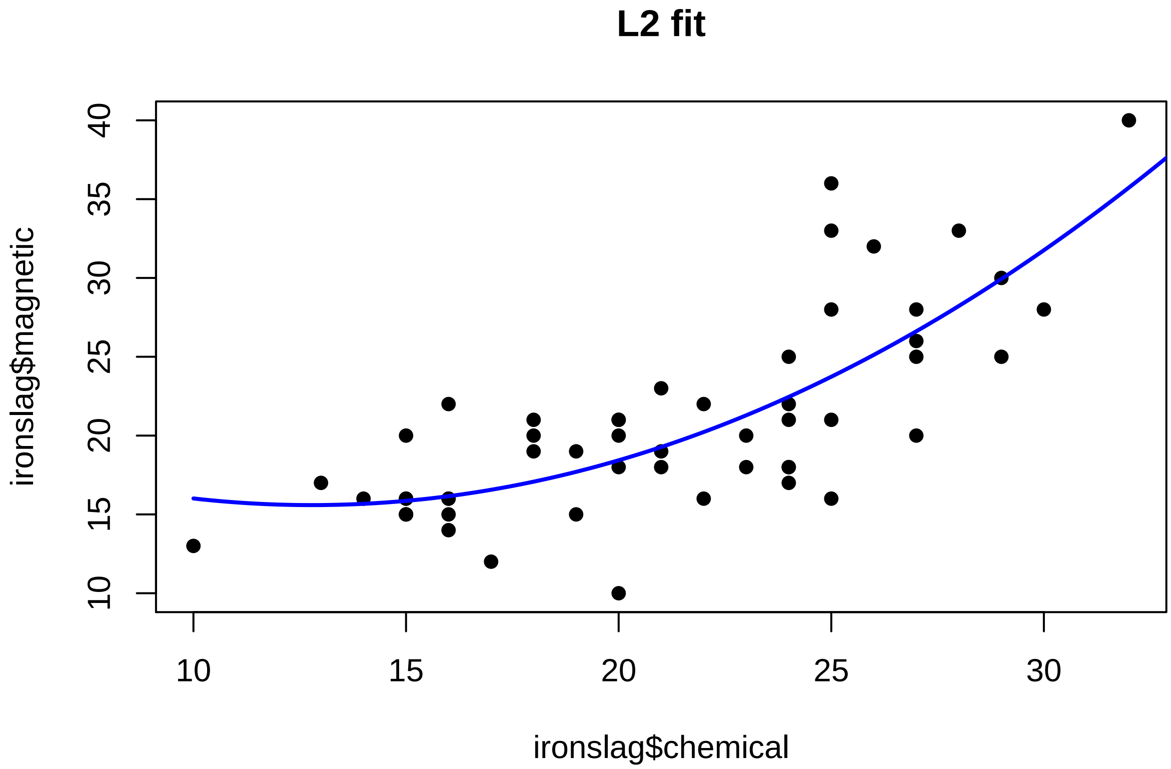
In addition to the linear model, fit the following models that predict the magnetic measurement (Y) from the chemical measurement (X).

Quadratic: $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \epsilon$

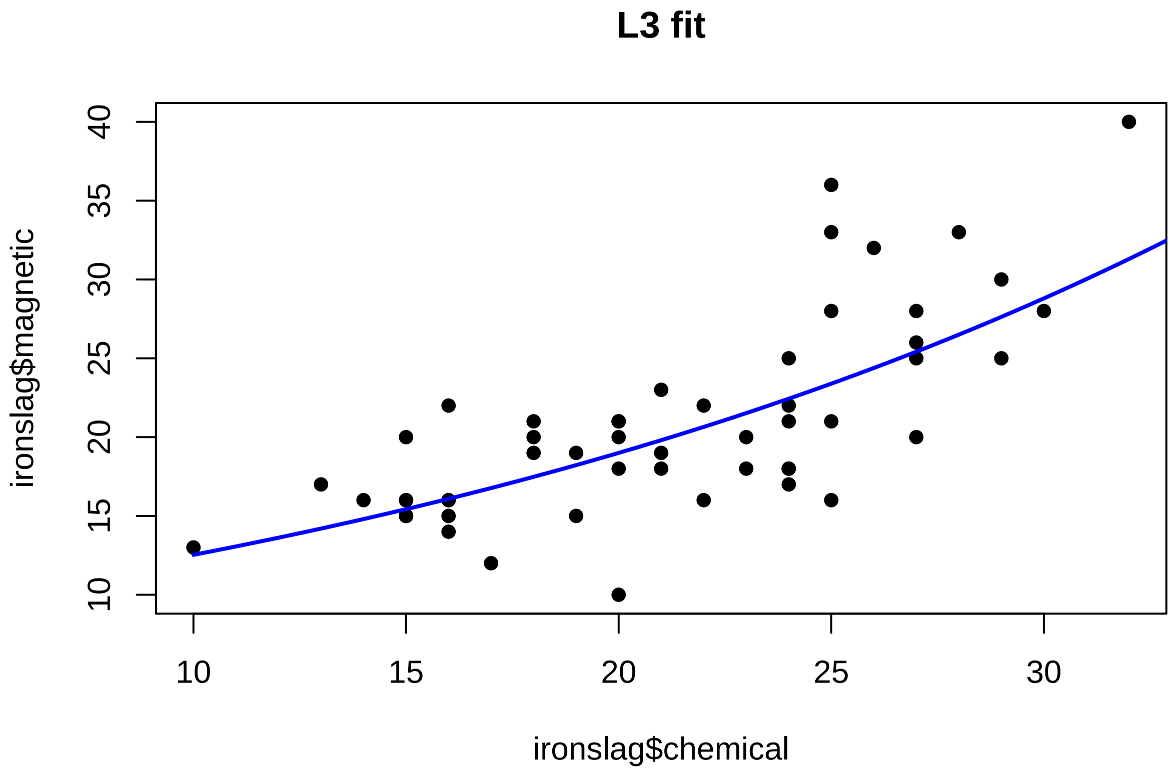
Exponential: $\log(Y) = \beta_0 + \beta_1 X + \epsilon$, equivalent to $Y = \exp(\beta_0 + \beta_1 X + \epsilon)$

log-log: $\log(Y) = \beta_0 + \beta_1 \log(X) + \epsilon$

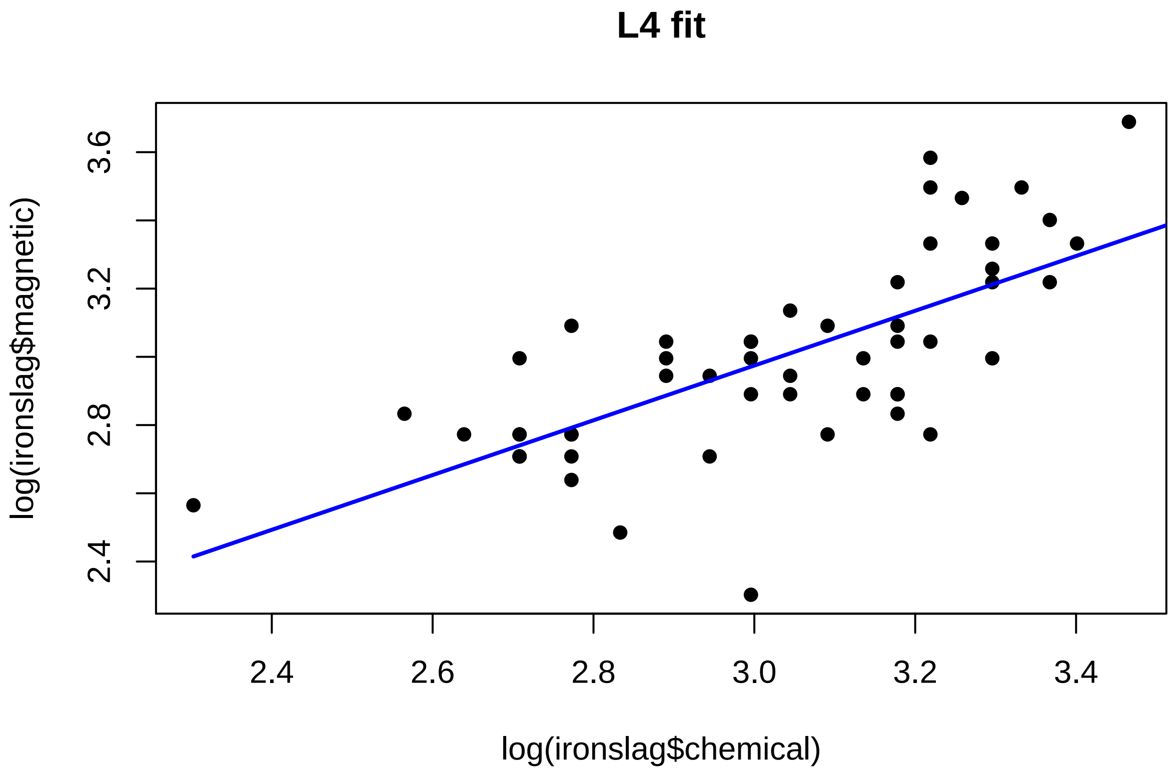
```
# I've started each of these for you.
# Your job is to create the plots with fitted lines.

L2 <- lm(magnetic ~ chemical + I(chemical^2), data = ironslag)
plot(ironslag$chemical, ironslag$magnetic, main = "L2 fit", pch = 16)
yhat2 <- L2$coef[1] + L2$coef[2]*x + L2$coef[3] * I(x^2)
lines(x,yhat2,lwd=2,col="blue")
```



```
L3 <- lm(log(magnetic) ~ chemical, data = ironslag)
# when plotting the fitted line for this one, create estimates of log(y-hat) linearly
# then exponentiate log(y-hat)
plot(ironslag$chemical, ironslag$magnetic, main = "L3 fit", pch = 16)
yhat3 <-exp(L3$coef[1] + L3$coef[2]*x)
lines(x,yhat3,lwd=2,col='blue')
```



```
L4 <- lm(log(magnetic) ~ log(chemical), data = ironslag)
# for this one, use plot(log(chemical), log(magnetic))
# the y-axis is now on the log-scale, so you can create and plot log(y-hat) directly
# just remember that you’ll use log(x) rather than x directly
plot(log(ironslag$chemical), log(ironslag$magnetic), main = "L4 fit", pch = 16)
x_log <- log(x)
yhat4 <- L4$coef[1] + L4$coef[2]*log(x)
lines(x_log,yhat4,lwd=2,col=’blue’)
```



Leave-one-out Cross validation

We will first attempt leave-one-out cross validation. In LOOCV, we remove one data point from our data set. We fit the model to the remaining 52 data points. With this model, we make a prediction for the left-out point. We then compare that prediction to the actual value to calculate the squared error. Once we find the squared error for all 53 points, we can take the mean to get a cross-validation error estimate of that model.

To test out our four models, we will build a loop that will remove one point of data at a time. Thus, we will make a `for(i in 1:53)` loop. For each iteration of the loop, we will fit the four models on the remaining 52 data points, and make a prediction for the remaining point.

```
# create vectors to store the validation errors for each model
error_model1 <- rep(NA, 53)
error_model2 <- rep(NA, 53)
error_model3 <- rep(NA, 53)
error_model4 <- rep(NA, 53)

for(i in 1:53){

  # write a line to select the ith line in the data
  # store this line as the 'test' case
  # store the remaining as the 'training' data
  test <- ironslag[i,]
  training <-ironslag[-i,]
```

```r
  # fit the four models and calculate the prediction error for each one
  # hint: it will be in the form

  model1 <- lm(magnetic ~ chemical, data = training)
  fitted_value1 <- predict(model1,test)
  error_model1[i] <- test$magnetic- fitted_value1

  model2 <- lm(magnetic ~ chemical + I(chemical^2), data = training)
  fitted_value2 <- predict(model2,test)
  error_model2[i] <- test$magnetic - fitted_value2

  model3 <- lm(log(magnetic) ~ chemical, data = training)
  fitted_value3 <- predict(model3,test)
  error_model3[i] <- log(test$magnetic) - fitted_value3

  model4 <- lm(log(magnetic) ~ log(chemical), data = training)
  fitted_value4<- predict(model4,test)
  error_model4[i] <- log(test$magnetic) - fitted_value4

}


# once all of the errors have been calculated, find the mean squared error
# ...

mean(error_model1^2)
```

```
## [1] 19.55644
```

```r
mean(error_model2^2)
```

```
## [1] 17.85248
```

```r
mean(error_model3^2)
```

```
## [1] 0.0397944
```

```r
mean(error_model4^2)
```

```
## [1] 0.04343859
```

Compare the sizes of the cross validation error to help you decide which model does the best job of predicting the test cases.

Using the same form, perform 6-fold cross validation. I have already created the list of cases to include in each fold.

```r
set.seed(1)
shuffled <- sample(53) # shuffles the values 1 to 53
cases <- list(
  test1 = shuffled[ 1:9], # the first 9 go into test set #1
  test2 = shuffled[10:18],
  test3 = shuffled[19:27],
  test4 = shuffled[28:36],
  test5 = shuffled[37:45],
  test6 = shuffled[46:53]
)

#create error models  1 2 3 4
error_model1 <- list(
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 8)
)
error_model2 <- list(
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 8)
)
error_model3 <- list(
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 8)
)
error_model4 <- list(
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 9),
  rep(NA, 8)
)

# There are 6 test sets, each have 9, with the exception of test set #6, which has 8
# you can access each vector in the cases list via cases[[1]], cases[[2]], etc.

for(i in 1:6){
  # write your code here.
  # it will be quite similar to the code you wrote for LOOCV

  test <- ironslag[cases[[i]],]
  training <-ironslag[-cases[[i]],]

  model1 <- lm(magnetic ~ chemical, data = training)
  fitted_value1 <- predict(model1,test)
  error_model1[[i]] <- test$magnetic- fitted_value1

  model2 <-lm(magnetic ~ chemical+I(chemical^2), data = training)
  fitted_value2 <- predict(model2,test)
  error_model2[[i]] <- test$magnetic- fitted_value2

  model3 <-lm(log(magnetic) ~ chemical, data = training)
  fitted_value3 <- predict(model3,test)
  error_model3[[i]] <- log(test$magnetic)- fitted_value3

  model4 <-lm(log(magnetic)~ log(chemical), data = training)
  fitted_value4 <- predict(model4,test)
  error_model4[[i]] <- log(test$magnetic)- fitted_value4
}
mean(unlist(error_model1)^2)
```

```
## [1] 18.97702
```

```
mean(unlist(error_model2)^2)
```

```
## [1] 17.18358
```

```
mean(unlist(error_model3)^2)
```

```
## [1] 0.03894465
```

```
mean(unlist(error_model4)^2)
```

```
## [1] 0.04217244
```

Again, compare the sizes of the cross validation error to help you decide which model does the best job of predicting the test cases.

Cross-validation with R

Now that you have wrote your cross-validation script from scratch, we can use the built-in functions in R. Library(boot) has the function `cv.glm()` which can be used to estimate cross-validation error.

To make use of `cv.glm()` on the linear models, we must first use `glm()` to fit a generalized linear model to our data. If you do not change the attribute “family” in the function `glm()`, it will fit a linear model

```
library(boot)
```

```
##
## Attaching package: ’boot’
```

```
## The following object is masked from ’package:lattice’:
##
##     melanoma
```

```r
L1 <- glm(magnetic ~ chemical, data = ironslag) # equivalent to lm(magnetic ~ chemical)
L2 <- glm(magnetic ~ chemical+I(chemical^2), data = ironslag)
L3 <- glm(log(magnetic) ~ chemical, data = ironslag)
L4 <- glm(log(magnetic) ~ log(chemical), data = ironslag)
L1_loocv <- cv.glm(ironslag, L1)$delta
L2_loocv<- cv.glm(ironslag, L2)$delta
L3_loocv <- cv.glm(ironslag, L3)$delta
L4_loocv <- cv.glm(ironslag, L4)$delta

L1_loocv
```

```
## [1] 19.55644 19.54152
```

```r
L2_loocv
```

```
## [1] 17.85248 17.83187
```

```r
L3_loocv
```

```
## [1] 0.03979440 0.03976952
```

```r
L4_loocv
```

```
## [1] 0.04343859 0.04340963
```

```r
L1_6foldcv <- cv.glm(ironslag, L1, K = 6)$delta
L2_6foldcv <- cv.glm(ironslag, L2, K = 6)$delta
L3_6foldcv <- cv.glm(ironslag, L3, K = 6)$delta
L4_6foldcv <- cv.glm(ironslag, L4, K = 6)$delta


L1_6foldcv
```

```
## [1] 18.92759 18.84351
```

```r
L2_6foldcv
```

```
## [1] 17.75640 17.58247
```

```r
L3_6foldcv
```

```
## [1] 0.04013913 0.03986375
```

```r
L4_6foldcv
```

```
## [1] 0.04278477 0.04256358
```

```r
# find the LOOCV and 6-fold CV values for the other three models
```

Your LOOCV estimates from `cv.glm()` should match your estimates when you coded your algorithm from scratch. The 6-fold CV estimates will vary because row selection will vary due to random sampling.

3) Decision Trees from Scratch

We will start by writing an algorithm to make a decision tree from scratch.

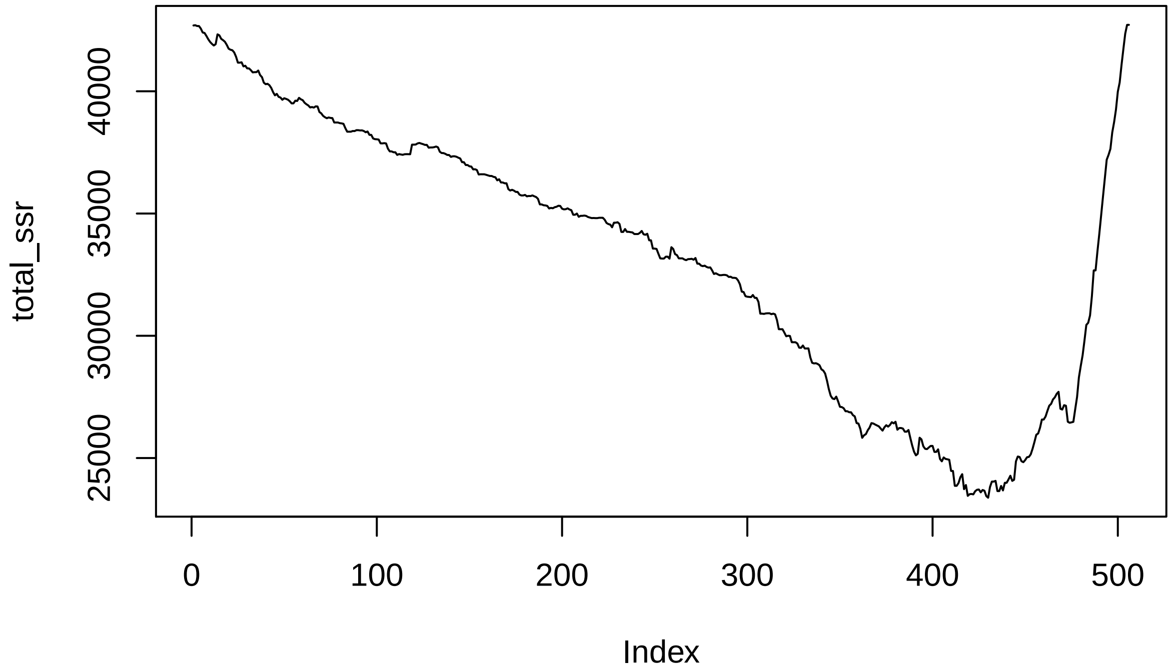
A decision tree will consider all possible splits, and will make a split that results in the smallest possible sum of squared residuals (SSR). It then recursively applies splits onto the resulting partition. We will write a short loop that will consider all possible splits for a chosen variable, and keep track of the resulting SSR.

For our example, we will use the Boston housing data in library MASS. We will try to model the variable `medv` (median home value) based on the other variables. For this first example, we will just consider the variable `rm` (average number of rooms for the district).

```r
library(MASS)
```

```
##
## Attaching package: 'MASS'

## The following object is masked from 'package:DAAG':
##
##     hills
```

```r
data(Boston)

# create an empy vector to store the resulting SSR after a split
total_ssr <- rep(NA, nrow(Boston))

for(i in 1:nrow(Boston)){
  # we will go through every possible split, which can occur between any two observations
  val = sort(Boston$rm)[i] # we sort the values of rm and select the ith value

  # split the median value data into two groups
  # 1) those that correspond to rm being equal to or less than the chosen ith value
  split_se <- subset(Boston,Boston$rm <= val)
  # 2) those that correspond to rm being greater than the chosen value
  split_l <- subset(Boston,Boston$rm > val)

  # find the mean of both groups
  m_se <- mean(split_se$medv)
  m_l <- mean(split_l$medv)
  # calculate the SSR for each group
  SSR_se <- sum((split_se$medv -m_se)^2)
  SSR_l <- sum((split_l$medv -m_l)^2)

  # add the SSRs from both groups together and store the resulting value in total_ssr
  total_ssr[i] <- SSR_se + SSR_l
}

plot(total_ssr, type = "l")
```



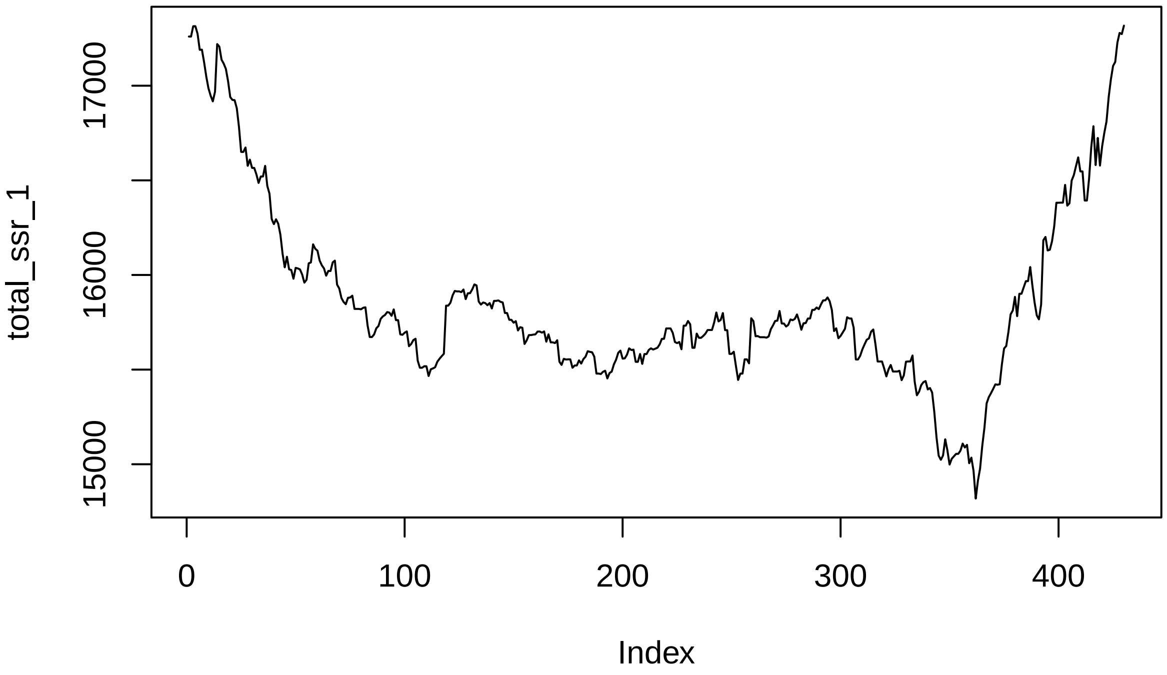
```r
# find the value that minimizes total SSR after the split
(minimizer <- which(total_ssr == min(total_ssr)))
```

```
## [1] 430
```

```r
(val <- sort(Boston$rm)[minimizer])
```

```
## [1] 6.939
```

```r
# use that value to split the data into Boston1 and Boston2
Boston1 <- Boston[Boston$rm <= val,]
Boston2 <- Boston[Boston$rm > val, ]

# repeat the above process on just Boston1 to find where to make another split
total_ssr_1 <- rep(NA, nrow(Boston1))
val1 <- sort(Boston1$rm)
for(i in 1:nrow(Boston1)){
  val1 <- sort(Boston1$rm)[i]
  split_se_1 <-subset(Boston1,Boston1$rm <= val1)
  split_l_1 <-subset(Boston1,Boston1$rm > val1)

  m_se_1 <- mean(split_se_1$medv)
  m_l_1 <- mean(split_l_1$medv)
  SSR_se_1 <- sum((split_se_1$medv - m_se_1)^2)
  SSR_l_1 <- sum((split_l_1$medv - m_l_1)^2)

  total_ssr_1[i] <- SSR_se_1 + SSR_l_1

}


plot(total_ssr_1, type = "l")
```



```
(minimizer_1 <- which(total_ssr_1 == min(total_ssr_1)))
```

```
## [1] 362
```

```
(val_1 <- sort(Boston$rm)[minimizer_1])
```

```
## [1] 6.545
```

Compare the results of your partitioning code to the results of calling tree on the Boston data.

```
library(tree)
tree(medv ~ rm, data = Boston)
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 506 42720 22.53
##   2) rm < 6.941 430 17320 19.93
##     4) rm < 6.5455 362 11670 18.89
##       8) rm < 5.858 111  3806 16.42 *
##       9) rm > 5.858 251  6889 19.98 *
##     5) rm > 6.5455 68  3144 25.50 *
##   3) rm > 6.941 76  6059 37.24
##     6) rm < 7.437 46  1900 32.11 *
##     7) rm > 7.437 30  1099 45.10 *
```

```
# ^^ building a tree, and considering partition only on the variable rm
```

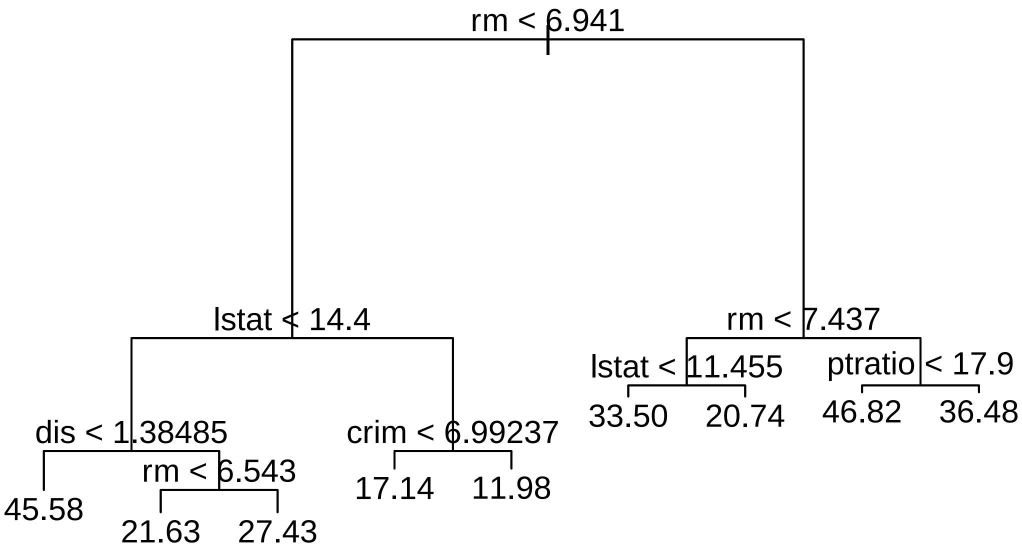
4) Decision Trees with R

Create a decision tree on the Boston data using all predictors with the function `tree()`. (See page 327-328 in ISLR for guidance.)

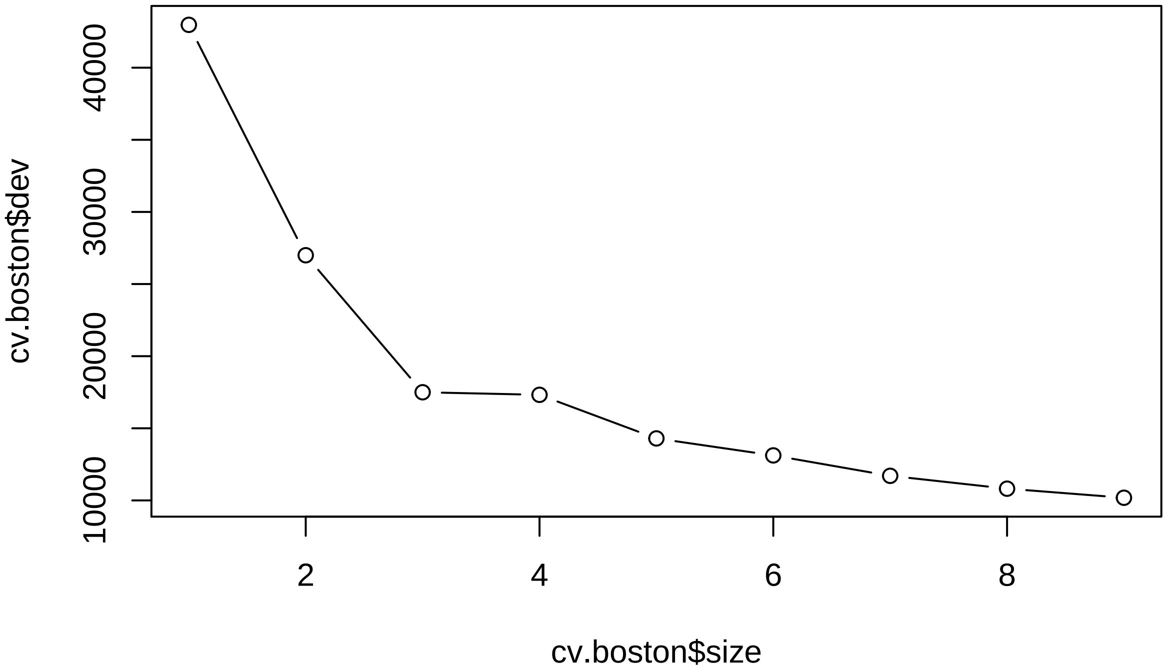
Once you make the tree, use the `cv.tree()` to see if pruning the tree will improve prediction performance. (In the case of the Boston housing data, pruning the tree does not improve prediction performance.)

Plot the resulting tree with the text labels.

```
tree.boston <- tree(medv ~ ., data = Boston)
plot(tree.boston)
text(tree.boston,pretty=0)
```



```
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size,cv.boston$dev, type='b')
```



We did not bother splitting the data between a training and test set. Still, use the resulting tree to make predictions on the data, to get an estimate of the mean squared error.

```
yhat <- predict(tree.boston, data = Boston)
mean((yhat - tree.boston$y)^2)
```

```
## [1] 13.30788
```

5) Bagging and Random Forests

Before we try to use the Random Forest functions that are available in, let's see how bagging works.

With bagging, we perform bootstrap resampling on the data, and fit separate trees to each of our bootstrapped datasets. At the end, we use each tree to make a prediction, and we average the the different predictions.

For our learning example, we will consider building a tree to predict medv based on the variable rm.

```
set.seed(1) # set seed for reproducibility

# The Boston dataset has 506 rows.
# we randomly sample 506 rows with replacement. This forms our first bootstrap sample
boot1 <- Boston[sample(nrow(Boston), replace = TRUE), ]
# We fit a tree to predict medv from on rm, based on the data in boot1
tree1 <- tree(medv ~ rm, data = boot1)
```

```
# our resulting tree.
tree1
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 506 42450.0 22.34
##   2) rm < 6.978 438 15790.0 19.75
##     4) rm < 6.5455 372 10960.0 18.88
##       8) rm < 5.706 71  3337.0 15.66 *
##       9) rm > 5.706 301  6714.0 19.63 *
##     5) rm > 6.5455 66  2929.0 24.71 *
##   3) rm > 6.978 68  4863.0 39.00
##     6) rm < 7.4545 38  1227.0 32.94 *
##     7) rm > 7.4545 30   482.4 46.66 *
```

```
# The first row in Boston has a rm = 6.575
# According to the tree, we predict a value of 25.82
(p1 <- predict(tree1, newdata = Boston[1,]))
```

```
##        1
## 24.71212
```

```
# We repeat the process and create two additional trees
boot2 <- Boston[sample(nrow(Boston), replace = TRUE), ]
# tree2 <- ...
tree2 <- tree(medv ~ rm, data = boot2)
tree2
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 506 37570.0 23.13
##   2) rm < 6.92 426 12560.0 20.42
##     4) rm < 6.486 328  6387.0 18.92
##       8) rm < 5.758 67  1385.0 15.36 *
##       9) rm > 5.758 261  3931.0 19.84 *
##     5) rm > 6.486 98  2967.0 25.44 *
##   3) rm > 6.92 80  5204.0 37.57
##     6) rm < 7.435 53  1288.0 33.14 *
##     7) rm > 7.435 27   832.7 46.27 *
```

```
# (p2 <- ... )
(p2 <- predict(tree2,newdata = Boston[1,]))
```

```
##        1
## 25.44082
```

```
# boot3 <-
boot3 <- Boston[sample(nrow(Boston), replace = TRUE),]
# tree3 <-
tree3 <- tree(medv ~ rm, data = boot2)
tree3
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 506 37570.0 23.13
##   2) rm < 6.92 426 12560.0 20.42
##     4) rm < 6.486 328  6387.0 18.92
##       8) rm < 5.758 67  1385.0 15.36 *
##       9) rm > 5.758 261  3931.0 19.84 *
##     5) rm > 6.486 98  2967.0 25.44 *
##   3) rm > 6.92 80  5204.0 37.57
##     6) rm < 7.435 53  1288.0 33.14 *
##     7) rm > 7.435 27   832.7 46.27 *
```

```
# tree3
# (p3 <- ... )
p3 <- predict(tree3, newdata = Boston[1,])

# Based on this 'bag' of three trees, what is the predicted median home value
# of a suburb where the average number of rooms is 6.575?
median (p1,p2,p3)
```

```
## [1] 24.71212
```

To do something similar with the function `randomForest()`, we provide the formula of the tree we want to create, and how many trees you want to use. In our case, we make three trees. `randomForest()` pretty much does the rest. Of course, in real life, we would probably never make a tree with only one predictor, and bagging with only three trees seems rather silly.

```
library(randomForest)
```

```
## randomForest 4.6-14
```

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

```
set.seed(1)
bag.boston = randomForest(medv ~ rm, data = Boston, ntree = 3)
yhat.bag = predict(bag.boston, newdata = Boston[1,])
yhat.bag
```

```
##        1
## 24.14167
```

Random Forests

The difference between bagging and Random Forest, is that in Random Forests, we restrict the variables that a tree can use to make a split at each node. This often forces a tree to make a sub-optimal split. The result is that we get trees that are less correlated with each other.

Using `randomForest()`, we can impose this restriction by setting the argument `mtry` to a value smaller than the total number of variables available. In the Boston data, there are 13 predictors, so setting `mtry` to any value less than 13 will cause us to grow a Random Forest. We often select p to be p/3 or something similar.

```r
set.seed(1)
# grow a random forest on the Boston data, using mtry = 5.
# comment on the mean squared residuals compared to the MSE of the single decision tree
rf.Boston <- randomForest(medv~. , data= Boston, mtry = 5)

yhat.rf <- predict(rf.Boston, data = Boston)

mean((yhat.rf - Boston$medv)^2)
```

```
## [1] 9.620774
```

The MSE of Random Forest is much lower than the MSE of the single decision tree. This means we are having an improvement on Random Forest method.