Scientific Computing and Data Science

Yiqiao Yin
Department of Statistics
Columbia University

Abstract

This document notes the core materials of Scientific Computing and Data Science course from Department of statistics at Columbia University. This is an introduction to programming in the R, including statistical packages, functions, objects, data structures, flow control, input and output, debugging, logical design, and abstraction, witing code for numerical and graphical statistical analyses, writing maintainable code and testing, stochastic simulations, paralleizing data analyses, and working with large data sets. I want to specially thank Professor Gabriel Young for his wonderful teachings.

Contents

This document is dedicated to Professor Gabriel Young.

1 INTRODUCTION TO R

1.1 Background: The Normal Distribution

Recall from your probability class that a random variable X is normally-distributed with mean μ and variance σ^2 (denoted $X \sim N(\mu, \sigma^2)$) if it has a probability density function, or *pdf*, equal to

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

In R we can simulate $N(\mu, \sigma^2)$ random variables using the `rnorm()` function. For example,

```
rnorm(n = 5, mean = 10, sd = 3)
```

```
## [1]  8.120639 10.550930  7.493114 14.785842 10.988523
```

outputs 5 normally-distributed random variables with mean equal to 10 and standard deviation (this is σ) equal to 3. If the second and third arguments are ommited the default rates are **mean = 0** and **sd = 1**, which is referred to as the "standard normal distribution".

1.1.1 Tasks

Sample means as sample size increases

1) Generate 100 random draws from the standard normal distribution and save them in a vector named **normal100**. Calculate the mean and standard deviation of **normal100**. In words explain why these values aren't exactly equal to 0 and 1.

```
# You'll want to type your response here.  Your response should look like:
# normal100 <-
# Of course, your answer should not be commented out.
## Answer here:
normal100 <- rnorm(100)
# Check here:
summary(normal100)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -2.21500 -0.54910  0.05823  0.08257  0.63730  2.40200
```

```
mean(normal100)
```

```
## [1] 0.08256659
```

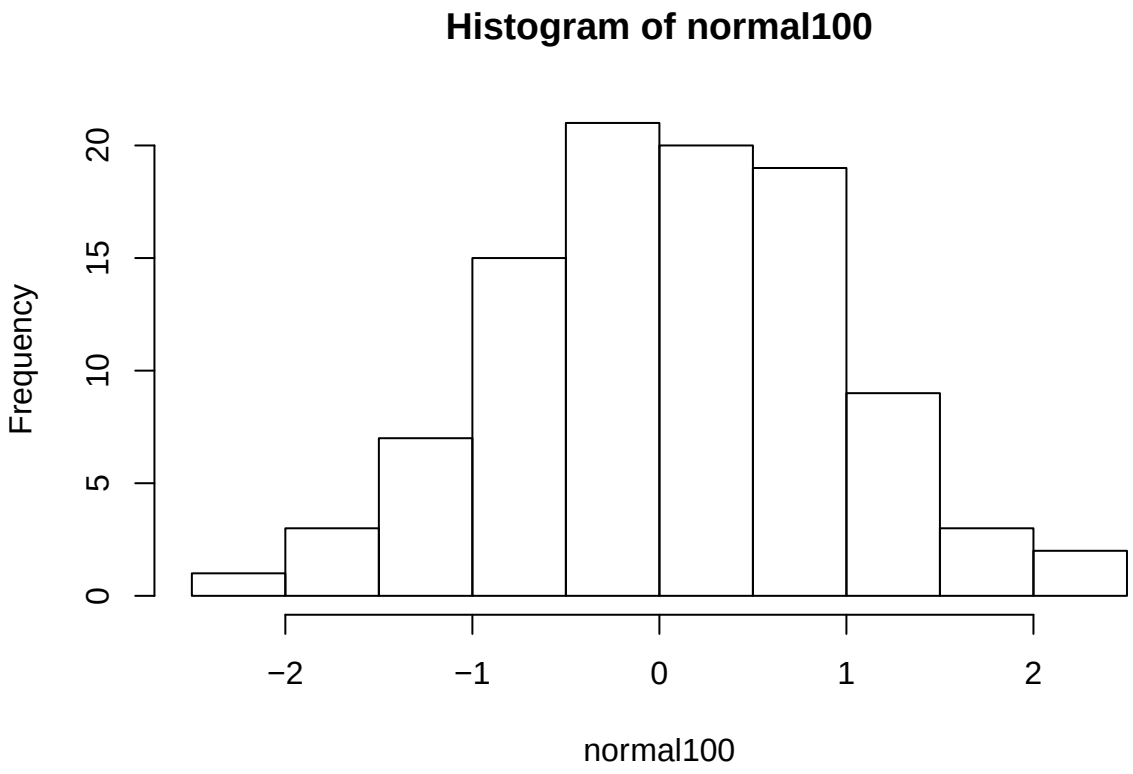
```
sd(normal100)
```

```
## [1] 0.8891336
```

```
# Comment: these values are randomly generated and there are 100 values.
# The sample mean and sample standard deviation would tend to the true
# mean and standard deviation, but they wouldn't be equal. However,
# if we increase size from 100 to 1000 or even bigger, I suspect the mean
# and standard deviation would tend to 0 and 1, respectively.
```

2) The function **hist()** is a base R graphing function that plots a histogram of its input. Use **hist()** with your vector of standard normal random variables from question (1) to produce a histogram of the standard normal distribution. Remember that typing **?hist** in your console will provide help documents for the **hist()** function. If coded properly, these plots will be automatically embedded in your output file.

```r
# Use hist() to produce histogram of
# the standard normal distribution:
hist(normal100)
```



3) Repeat question (1) except change the number of draws to 10, 1000, 10,000, and 100,000 storing the results in vectors called **normal10**, **normal1000**, **normal10000**, **normal100000**.

```r
# use rnorm():
normal10 <- rnorm(10)
normal1000 <- rnorm(1000)
normal10000 <- rnorm(10000)
normal100000 <- rnorm(1000000)
```

4) We want to compare the means of our four random draws. Create a vector called **sample_means** that has as its first element the mean of **normal10**, its second element the mean of **normal100**, its third element the mean of **normal1000**, its fourth element the mean of **normal10000**, and its fifth element the mean of **normal100000**. After you have created the **sample_means** vector, print the contents of the vector and use the **length()** function to find the length of this vector. (it should be five). There are, of course, multiple ways to create this vector. Finally, explain in words the pattern we are seeing with the means in the **sample_means** vector.

```r
# Create a mean vector:
sample_means <- c(
  mean(normal10),
  mean(normal100),
  mean(normal1000),
  mean(normal10000),
```

```
  mean(normal100000)
); sample_means
```

```
## [1]  0.4937354371  0.0825665889 -0.0268757231 -0.0067198069  0.0003208042
```

```
# Now print contents of vector
# using length()
length(sample_means)
```

```
## [1] 5
```

```
# Comment: as we can see, the mean of the groups of normal
# distribution actually decreases as the size of the
# normal distribution increases. Moreover, the mean
# is tending to 0 as we expected as theoretical mean for
# normal distribution.
```

Sample distribution of the sample mean

5) Let's push this a little farther. Generate 1 million random draws from a normal distribution with $\mu = 3$ and $\sigma^2 = 4$ and save them in a vector named **normal1mil**. Calculate the mean and standard deviation of **normal1mil**.

```
# Use rnorm() function but we need
# to specify mean and sd:
normal1mil <- rnorm(10^6,mean=3,sd=2)
mean(normal1mil); sd(normal1mil)
```

```
## [1] 2.999172
```

```
## [1] 2.001561
```

6) Find the mean of all the entries in **normal1mil** that are greater than 3. You may want to generate a new vector first which identifies the elements that fit the criteria.

```
# Find mean of all the entries:
mean_all <- mean(normal1mil); mean_all
```

```
## [1] 2.999172
```

```
find_which <- which(normal1mil > 3); length(find_which); head(find_which)
```

```
## [1] 499830
```

```
## [1]  1  3  5  7  9 11
```

```
# Comment: from above, we have observed that
# that there are 500,078 entries greater than 3.
# Generate a new vector:
new_vector <- normal1mil[find_which]; length(new_vector); head(new_vector); mean(new_vector)
```

```
## [1] 499830
```

```
## [1] 3.484544 4.967129 4.159350 3.392708 4.330506 5.601582
```

```
## [1] 4.596509
```

```
# Comment: from answer above, the estimated mean is very
# close to the theoretical mean we want.
```

7) Create a matrix **normal1mil_mat** from the vector **normal1mil** that has 10,000 columns (and therefore should have 100 rows).

```
# Create a new matrix with size 100x10,000
normal1mil_mat <- matrix(normal1mil,ncol=10000); dim(normal1mil_mat)
```

```
## [1]   100 10000
```

8) Calculate the mean of the 1234^{th} column.

```
# Mean of 1234th column:
mean_1234 <- mean(normal1mil_mat[,1234]); mean_1234
```

```
## [1] 3.160561
```

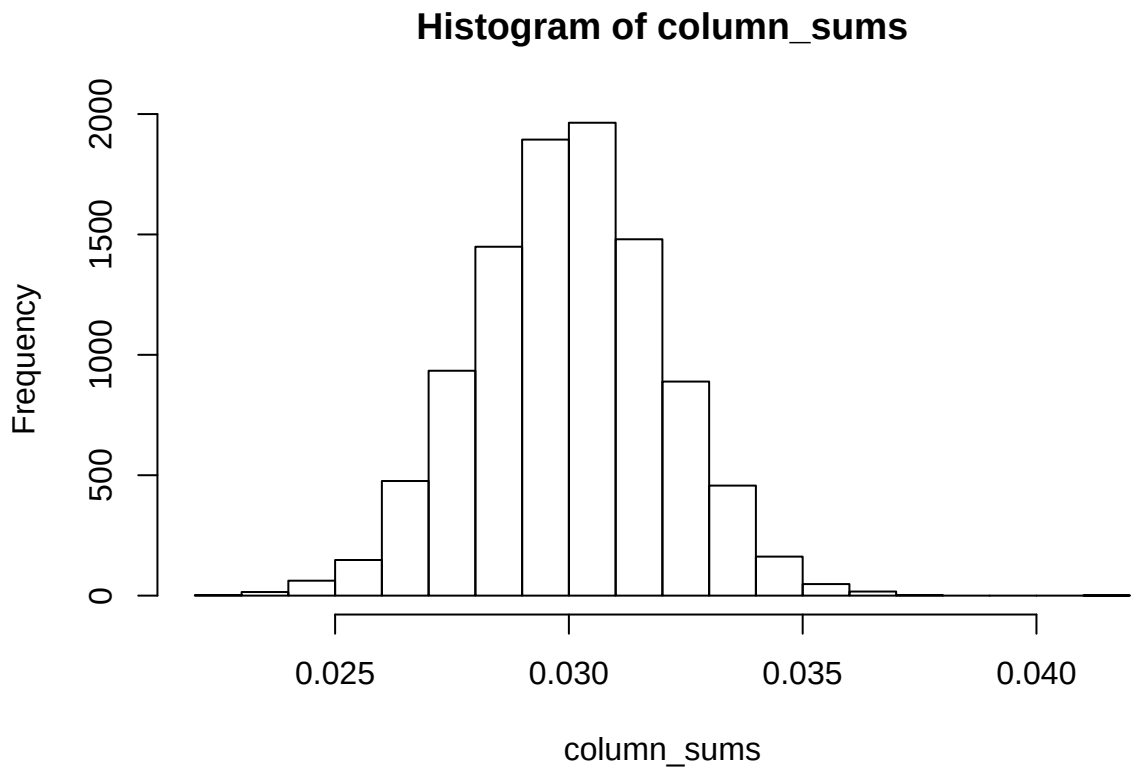
9) Use the **colSums()** functions to calculate the *means* of each column of **normal1mil_mat**. Remember, **?colSums** will give you help documents about this function. Save the vector of column means with an appropriate name as it will be used in the next task.

```
# Check definition:
# ?colSums # Form row and column sums and means for numeric arrays (or data frames)
# Create vector of column sums by using colSums():
column_sums <- colSums(normal1mil_mat)/10000; length(column_sums)
```

```
## [1] 10000
```

10) Finally, produce a histogram of the column means you calculated in task (9). What is the distribution that this histogram approximates (i.e. what is the distribution of the sample mean in this case)?

```
# Produce histogram
hist(column_sums)
```



```r
summary(column_sums); sd(column_sums)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.02236 0.02863 0.03001 0.02999 0.03134 0.04153
```

```
## [1] 0.00202249
```

```r
# Comment: This is a standard normal distribution
# and we observe the mean and standard deviation
# from the above code. Both mean and std are
# close to zero and this is a piece of evidence
# that part (9) above in this section
# is coded correctly.
```

1.2 Dataframe

For dataframe, we consider two-dimensional tables of data. As in matrices, we have rows and columns. We can also assign names for each row and each column. This subsection we introduce how to work with data frame and we look at several examples for illustration.

Sometimes the dataframe can get very dirty and difficult to read. In this case, we clean the dataset. We can look for "NA" entries as well as "NULL" entries. We can also re-edit the row and/or column names (etc.) to make the tables more readable. A few functions introduced here are: sort(), findtext(), alphabetized(), and etc. Moreover, we can write control statements, i.e. if-statement.

2 EXPLORATORY DATA

2.1 Background: Edgar Anderson's Iris Data

The R data description follows:

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

2.2 Purpose of Iris Dataset

The purpose of this lab is to preform simple filtering tasks, use the function **tapply()** and successfully write a loop.

2.2.1 Tasks

1) Run the following code and briefly explain what the two functions are doing.

```r
## head function
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
```

```
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# Comment: this is giving us a quick view
# of the first 6 rows of the data frame.

## names function
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

```
# Comment: this is giving us all the names of the data frame, iris,
# which are "Sepal.Length", etc.
```

2) In one line of code, create a vector named **Versicolor** which contains a 1 if the species is versicolor and contains a 0 otherwise. Use the function **ifelse()** to accomplish this task.

```
#-- R code goes here ----
Versicolor <- ifelse(iris$Species == "versicolor", 1, 0)
Versicolor
```

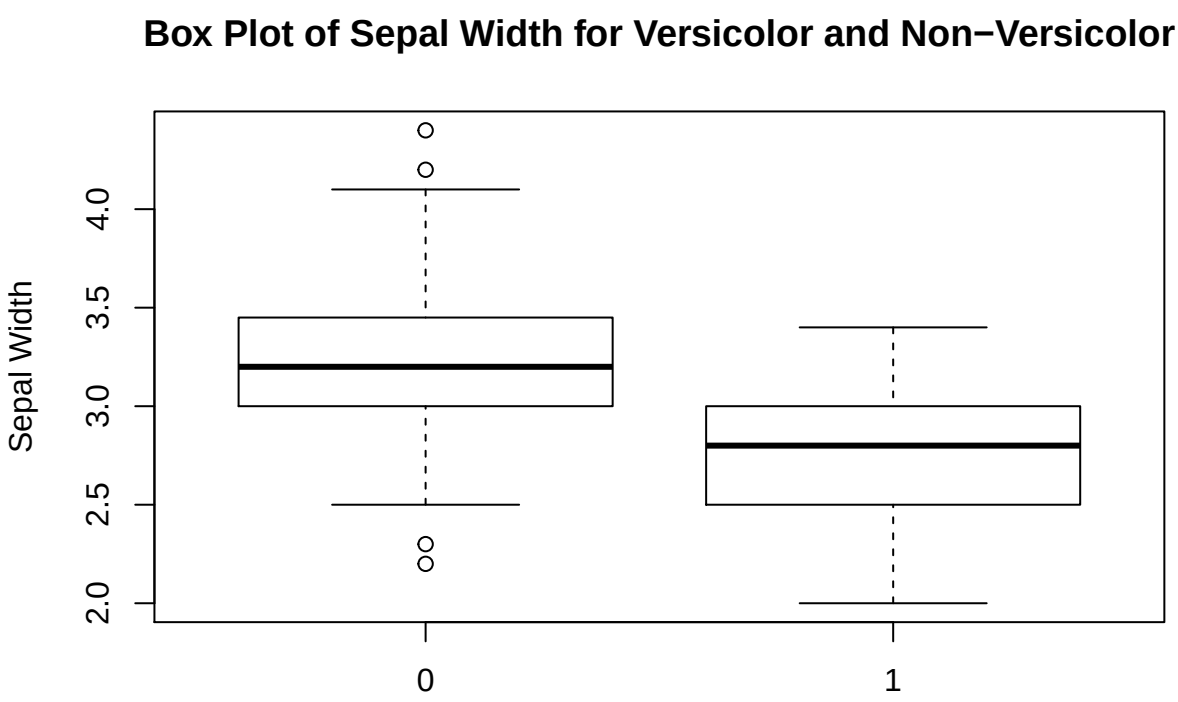
```
##   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [106] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [141] 0 0 0 0 0 0 0 0 0 0
```

3) Based on the vector **Versicolor** defined in Part 2, construct a comparative boxplot of sepal width split by whether of not the species is versicolor. The code is given below. Change the main title and y-label of the plot.

```
# Don't forget to uncomment the code below
# ?boxplot()
summary(iris$Sepal.Width)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.000   2.800   3.000   3.057   3.300   4.400
```

```
boxplot(Sepal.Width~Versicolor,
        main="Box Plot of Sepal Width for Versicolor and Non-Versicolor",
        ylab="Sepal Width",
        data=iris
        )
```



4) Using filtering techniques and the function **mean()**, compute the mean sepal length of only the setosa species. Write only one line of code to accomplish this task.

```
#-- R code goes here ----
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# Setosa <- ifelse(iris$Species == "setosa", 1, 0)
tapply(iris$Sepal.Length, ifelse(iris$Species == "setosa", 1, 0), mean)
```

```
##     0     1
## 6.262 5.006
```

5) Run the following code and briefly explain what the function **tapply()** is accomplishing.

```
tapply(iris$Sepal.Length,iris$Species,mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

```
# Comment: tapply() is a function calculating the mean of Sepal Length
# based on Species.
```

6) Write a loop that that computes the mean of each quantitative variable split by each iris species. Store the computed means in a matrix named **MeanFlowers**. This matrix should have dimension 4 by 3. Also display the matrix after you run the loop. I started the loop below:

```
# Quick view:
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# define a matrix of zeros
MeanFlowers <- matrix(0,nrow=4,ncol=3)

# define a character vector corresponding to the numeric variable names
measurements <- c("Sepal.Length","Sepal.Width","Petal.Length","Petal.Width")

# name the rows and columns of the matrix MeanFlowers
rownames(MeanFlowers)  <- measurements
colnames(MeanFlowers)  <- c("setosa","versicolor","virginica")

# Loop
for (i in 1:4) {

  #-- R code goes here ----
  #-- Don't forget to uncomment the loop

  MeanFlowers[i,] <- tapply(iris[,i],iris$Species,mean)
}
MeanFlowers
```

```
##              setosa versicolor virginica
## Sepal.Length  5.006      5.936     6.588
## Sepal.Width   3.428      2.770     2.974
## Petal.Length  1.462      4.260     5.552
## Petal.Width   0.246      1.326     2.026
```

2.3 Background: Edgar Anderson's Iris Data

The R data description follows:

This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica.

2.4 Goal

The purpose of this lab is to preform simple filtering tasks, use the function **tapply()** and successfully write a loop.

2.4.1 Tasks

1) Run the following code and briefly explain what the two functions are doing.

```
## head function
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# Comment: this is giving us a quick view
# of the first 6 rows of the data frame.

## names function
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

```
# Comment: this is giving us all the names of the data frame, iris,
# which are "Sepal.Length", etc.
```

2) In one line of code, create a vector named **Versicolor** which contains a 1 if the species is versicolor and contains a 0 otherwise. Use the function **ifelse()** to accomplish this task.

```
#-- R code goes here ----
Versicolor <- ifelse(iris$Species == "versicolor", 1, 0)
Versicolor
```

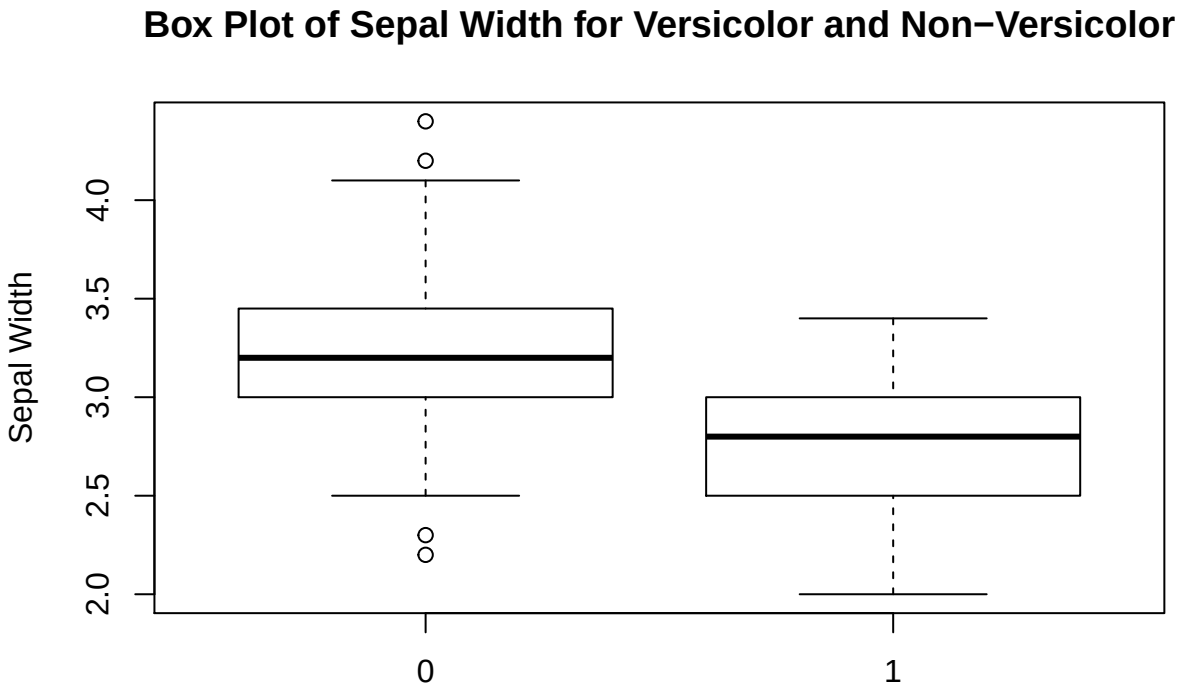
```
##   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [106] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [141] 0 0 0 0 0 0 0 0 0 0
```

3) Based on the vector **Versicolor** defined in Part 2, construct a comparative boxplot of sepal width split by whether of not the species is versicolor. The code is given below. Change the main title and y-label of the plot.

```
# Don't forget to uncomment the code below
# ?boxplot()
summary(iris$Sepal.Width)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   2.000   2.800   3.000   3.057   3.300   4.400
```

```
boxplot(Sepal.Width~Versicolor,
        main="Box Plot of Sepal Width for Versicolor and Non-Versicolor",
        ylab="Sepal Width",
        data=iris
        )
```



4) Using filtering techniques and the function **mean()**, compute the mean sepal length of only the setosa species. Write only one line of code to accomplish this task.

```
#-- R code goes here ----
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# Setosa <- ifelse(iris$Species == "setosa", 1, 0)
tapply(iris$Sepal.Length, ifelse(iris$Species == "setosa", 1, 0), mean)
```

```
##     0     1
## 6.262 5.006
```

5) Run the following code and briefly explain what the function **tapply()** is accomplishing.

```
tapply(iris$Sepal.Length,iris$Species,mean)
```

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

```
# Comment: tapply() is a function calculating the mean of Sepal Length
# based on Species.
```

6) Write a loop that that computes the mean of each quantitative variable split by each iris species. Store the computed means in a matrix named **MeanFlowers**. This matrix should have dimension 4 by 3. Also display the matrix after you run the loop. I started the loop below:

```
# Quick view:
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# define a matrix of zeros
MeanFlowers <- matrix(0,nrow=4,ncol=3)

# define a character vector corresponding to the numeric variable names
measurements <- c("Sepal.Length","Sepal.Width","Petal.Length","Petal.Width")

# name the rows and columns of the matrix MeanFlowers
rownames(MeanFlowers)  <- measurements
colnames(MeanFlowers)  <- c("setosa","versicolor","virginica")

# Loop
for (i in 1:4) {

  #-- R code goes here ----
  #-- Don't forget to uncomment the loop

  MeanFlowers[i,] <- tapply(iris[,i],iris$Species,mean)
}
MeanFlowers
```

```
##              setosa versicolor virginica
## Sepal.Length  5.006      5.936     6.588
## Sepal.Width   3.428      2.770     2.974
## Petal.Length  1.462      4.260     5.552
## Petal.Width   0.246      1.326     2.026
```

3 BOOTSTRAP

3.1 Part (A): Simple Linear Regression Model

1) Import the **diamonds_small.csv** dataset into R and store in a dataframe called **diamonds**. Use the **lm()** command to regress **price** (response) on **carat** (predictor) and save this result as **lm0**. What are the coefficients of **lm0**? (Some of this problem is solved for you below.)

```
# You'll want to type your response to question A(1) here.  Your response should look like:
# setwd("~/Desktop/")
diamonds <- read.csv("diamonds_small.csv", as.is = TRUE, header = TRUE)
rows <- dim(diamonds)[1]; rows
```

```
## [1] 53940
```

```
diamonds <- diamonds[sample(1:rows, 2000), ]
# head(diamonds)

# Linear Model:
lm0 <- lm(price ~ carat, data = diamonds)
coefficients(lm0)
```

```
## (Intercept)       carat
##   -2163.510    7564.318
```

```
# Of course, your answer should not be commented out.
```

Recall from lecture that the estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ that you just calculated with **lm()** are functions of the data values and are therefore themselves are random (they inherit variability from the data). If we were to recollect the diamonds data over and over again, the estimates would be different each time.

In this lab we'll use bootstrapping to answer the following questions:

1. **"How much does $\hat{\beta}_1$ vary from one replication of the experiment to the other?"**
2. **"What are all the values of β_1 that would have produced this data with high probability?"**

3.2 Part (B): How Does Estimator Vary?

Strategy: we'll re-sample **(price, carat)** pairs in order to provide an estimate for how $\hat{\beta}_1$ varies across samples.

1) How many rows are in the **diamonds** dataset? Call this value **n**.

```
# You'll want to type your response to question B(1) here.  Your response should look like:
n <- nrow(diamonds)
n
```

```
## [1] 2000
```

2) We'll next use the **sample()** function to re-sample **n** rows of the **diamonds** dataset *with replacement*. The following code provides a single re-sample of the values $1, 2, \ldots, n$, or a single re-sample of the rows of the dataset.

```
resample1 <- sample(1:n, n, replace = TRUE)
head(resample1)
```

```
## [1] 1869 1383  918 1961 1000 1566
```

```
# Remove the comment symbol in front of the above after n is assigned in B(1)
```

Now write a loop to calculate **B <- 1000** such re-samples and store them as rows of the matrix **resampled_values** which will have **B** rows and **n** columns.

```
# You'll want to type your response to question B(2) here.  Your response should look like:
B <- 1000
resampled_values <- matrix(NA, nrow = B, ncol = n)
for (b in 1:B) {
#   Write code that fills in the matrix resample_values with re-samples.
  resampled_values[b,] <- sample(1:n, n, replace = TRUE)
}
resampled_values[1:5,1:5] # 5x5 quick view
```

```
##      [,1] [,2] [,3] [,4] [,5]
```

```
## [1,] 1747  695  514 1589   68
## [2,]  896 1340  209 1474  768
## [3,]   65  968  906  155  341
## [4,]  946  936  346 1502  579
## [5,]  604  508   79  755 1216
```

3) Now we'll use each re-sampled dataset to provide a new estimate of $\hat{\beta}_1$. Write a line of code that uses **resample1** above to produce a resamples dataset of **(price, carat)** pairs. Using the re-sampled dataset, use **lm()** to produce new estimates of $\hat{\beta}_0$ and $\hat{\beta}_1$. These values should be stored in a vector called **resample1_ests**.

```
# You'll want to type your response to question B(3) here.  Your response should look like:
resample_dataset <- matrix(NA, nrow=n)
for (b in 1:B) {
  # First, write a line of code that will give us a bootstrap matrix that has
  # 2000 rows, and 2x1000 columns.
  resample_dataset <- cbind(resample_dataset, diamonds[sample(1:n, n, replace = TRUE),])
}
resample_dataset <- resample_dataset[,-1]; head(resample_dataset)[,1:6]; dim(resample_dataset)
```

```
##       carat price carat.1 price.1 carat.2 price.2
## 34336  0.42   862    1.02    4983    0.99    5671
## 39195  0.38  1064    1.63   10479    1.54   12061
## 2714   0.32   564    0.34    1014    1.53    8996
## 13169  1.00  5445    1.01    4129    1.14    5858
## 28665  0.31   677    1.01    6267    0.41     791
## 48931  0.74  2042    1.00    4032    0.32     756
```

```
## [1] 2000 2000
```

```
resample1_ests <- matrix(NA,nrow=2*B,ncol=2)
for (b in 1:(2*B)) {
  # Next, using the above matrix, we produce, for each pair of carat and price,
  # a paramter space, \hat{\beta}_0 and \hat{\beta}_1
  ifelse(
    (b-1)%%2 == 0,
    resample1_ests[b,] <- data.frame(summary(lm(resample_dataset[,b+1]~resample_dataset[,b]))$coef)[,1]
    resample1_ests[b,] <- resample1_ests[b,]
  )
  # This will give us odd rows with data and even rows with NA entries.
}
resample1_ests <- resample1_ests[-which(is.na(resample1_ests)),] # Get rid of NA entries.
head(resample1_ests); dim(resample1_ests)
```

```
##           [,1]     [,2]
## [1,] -2128.797 7579.115
## [2,] -2129.252 7520.124
## [3,] -2143.534 7480.379
## [4,] -2185.858 7586.105
## [5,] -2221.456 7650.216
## [6,] -2175.162 7534.163
```

```
## [1] 1000    2
```

```
# Finally, we have a dataset, namely resample1_ests, that stores
# estimated parameters column 1 is for all \beta_0 (intercepts) and
# column 2 is for all \beta_1 (coefficients of carat variable).
```

Hint: (a) Note that the following code produces the re-sampled dataset from the re-sampled values:

```
# resampled_data <- diamonds[resample1, ]
# Remove the comment symbol in front of the above after resample1 is assigned in B(2)
# Code is provided above.
```

Hint: (b) You'll probably want to use the **coefficients()** function.

```
# Comment: I used summary(lm())$coef function discussed in class.
```

4) Repeat the above call for each re-sampled dataset produced from the **resampled_values** matrix. We'll store the new coefficient estimates in a matrix **resampled_ests** with **B** rows and **2** columns. Again you'll want to write a loop, this time that iterates over the rows of **resampled_values**. (Note that if you are very clever this could be done using **apply()**.) Make sure to print **head(resample_ests)** at the end.

```
# You'll want to type your response to question B(4) here.  Your response should look like:
# resampled_ests <- matrix(NA, nrow = B, ncol = 2)
# names(resampled_ests) <- c("Intercept_Est", "Slope_Est")
# for (b in 1:B) {
#   Write code that fills in the matrix resample_ests with coefficent estimates.
# }
# head(resampled_ests)

# Comment:
# Code is written above in 3)
head(resample1_ests); dim(resample1_ests)
```

```
##           [,1]     [,2]
## [1,] -2128.797 7579.115
## [2,] -2129.252 7520.124
## [3,] -2143.534 7480.379
## [4,] -2185.858 7586.105
## [5,] -2221.456 7650.216
## [6,] -2175.162 7534.163
```

```
## [1] 1000    2
```

Hint: You may want to use multiple lines of code within the for loop. One idea is to first use the rows corresponding to re-sample **b** provided in **resampled_values** to created a resampled dataset. Then use the new dataset to provide new estimates of the coefficients.

5) Recall from lecture that $(\hat{\beta}_1^{(b)})_{b=1}^{B} - \hat{\beta}_1$ approximates the sampling distribution of $\hat{\beta}_1 - \beta_1$ where β_1 is the population parameter, $\hat{\beta}_1$ is the estimate from out original dataset, and $(\hat{\beta}_1^{(b)})_{b=1}^{B}$ are the B bootstrap estimates.

Make a vector **diff_estimates** that holds the differences between the original estimate of $\hat{\beta}_1$ from **lm0** and the bootstrap estimates. It should have length **B**.

```
# You'll want to type your response to question B(5) here.  Your response should look like:
summary(lm(diamonds$price~diamonds$carat, data=diamonds))$coef[,1]
```

```
##    (Intercept) diamonds$carat
##      -2163.510       7564.318
```

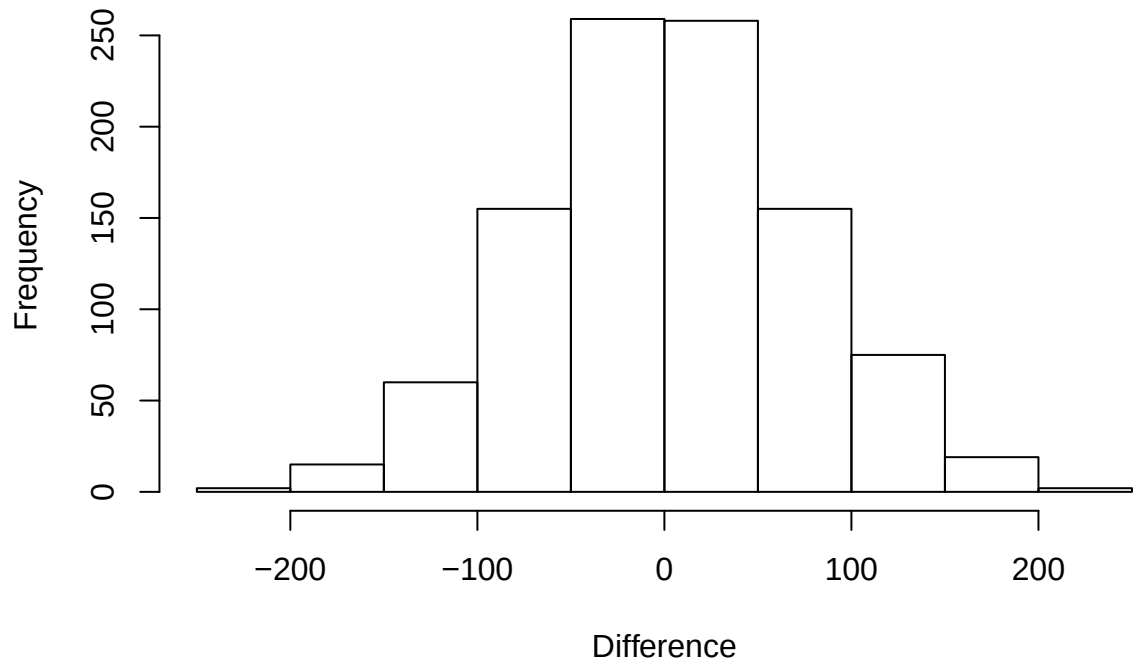
```
diff_estimates <- resample1_ests - cbind(
  matrix(summary(lm(diamonds$price~diamonds$carat, data=diamonds))$coef[1,1],nrow=B),
  matrix(summary(lm(diamonds$price~diamonds$carat, data=diamonds))$coef[2,1],nrow=B)
```

```
)
head(diff_estimates)
```

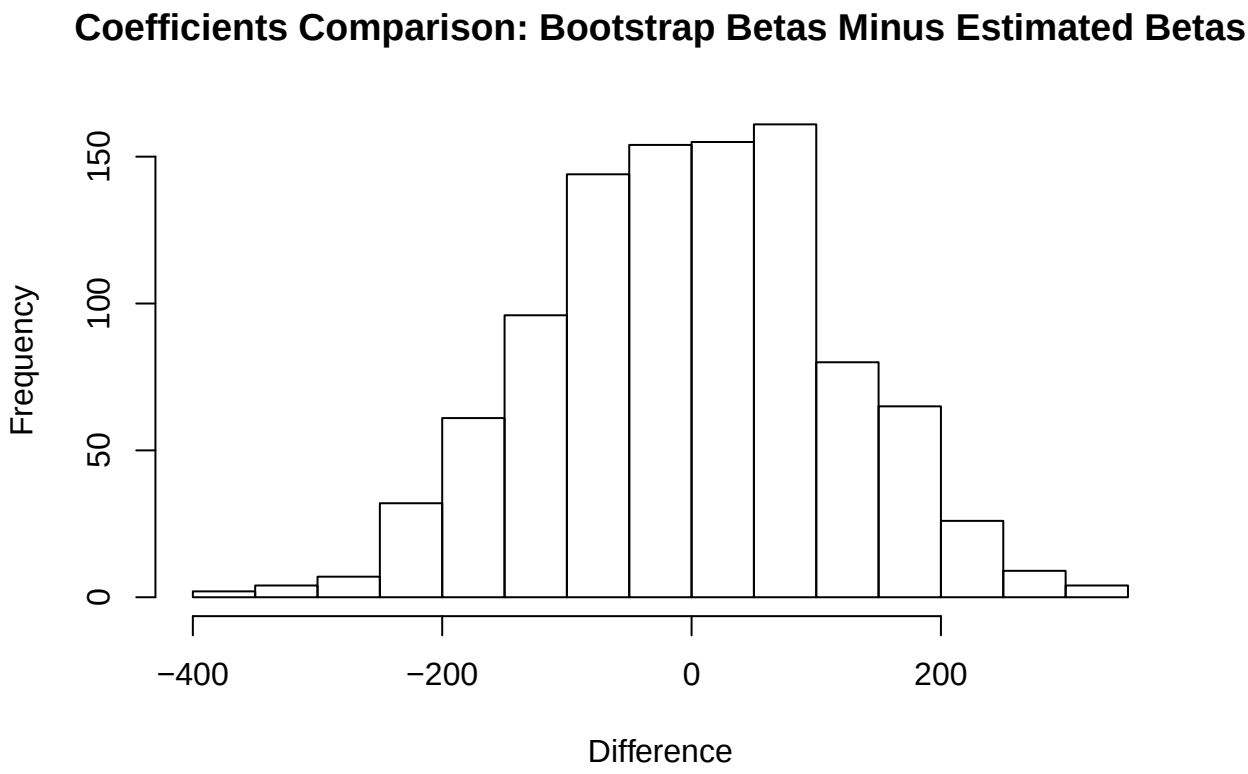
```
##           [,1]      [,2]
## [1,]  34.71229  14.79775
## [2,]  34.25778 -44.19329
## [3,]  19.97550 -83.93907
## [4,] -22.34815  21.78725
## [5,] -57.94646  85.89792
## [6,] -11.65215 -30.15504
```

6) Plot a histogram of the bootstrap estimates of $\hat{\beta}_1$ (they're in the 'Slope_Est' column). Label the x-axis appropriately.

```
# You'll want to type your response to question B(6) here.
hist(diff_estimates[,1],
     xlab="Difference",
     main="Coefficients Comparison: Bootstrap Intercepts Minus Estimated Intercepts")
```




```
hist(diff_estimates[,2],
     xlab="Difference",
     main="Coefficients Comparison: Bootstrap Betas Minus Estimated Betas")
```



7) Calculate the standard deviation of the bootstrap estimates.

```
# You'll want to type your response to question B(7) here.
sd_intercept <- sd(diff_estimates[,1])
sd_beta <- sd(diff_estimates[,2])
sd_intercept; sd_beta
```

```
## [1] 71.74653
```

```
## [1] 117.1836
```

3.3 [Optional] Part (C): Bootstrap Confidence Intervals

Note: This section is optional. If you get the chance to do it during lab, great, but it's not necessary that this part is completed when you turn in the lab.

Finally we'd like to approximate confidence intervals for the regression coefficients. Recall that a confidence interval is a random interval which contains the truth with high probability (the confidence level). If the confidence interval for β_1 is C, and the confidence level is $1 - \alpha$, then we want

$$Pr(\beta_1 \in C) = 1 - \alpha$$

no matter what the true value of β_1.

We estimate the confidence interval from the bootstrap estimates by finding a range of $(\hat{\beta}_1^{(b)})_{b=1}^{B} - \hat{\beta}_1$ which holds $1 - alpha$ percent of the values. In our case, let $\alpha = 0.05$, so we estimate a confidence interval with level 0.95.

(1) Let **Cu** and **Cl** be the upper and lower limits of the confidence interval. Use the **quantile()** function to find the 0.025 and 0.975 quantiles of the vector **diff_estimates** calculated in B(5). Then **Cu** is the sum of the original estimate of $\hat{\beta}_1$ from **lm0** with the upper quantile and **Cl** is the sum of the original estimate of $\hat{\beta}_1$ from **lm0** with the lower quantile.

```
# You'll want to type your response to question C(1) here.  Your response should look like:
# Recall the difference matrix:
head(diff_estimates)
```

```
##           [,1]      [,2]
## [1,]  34.71229  14.79775
## [2,]  34.25778 -44.19329
## [3,]  19.97550 -83.93907
## [4,] -22.34815  21.78725
## [5,] -57.94646  85.89792
## [6,] -11.65215 -30.15504
```

```
# For difference of coefficients for beta_1:
quantile(diff_estimates[,2], seq(0,0.03,by=.005))
```

```
##        0%      0.5%        1%      1.5%        2%      2.5%        3%
## -389.0323 -308.6272 -285.6410 -247.0473 -239.6702 -226.1844 -221.2920
```

```
quantile(diff_estimates[,2], seq(0.97,1,by=.005))
```

```
##      97%    97.5%      98%    98.5%      99%    99.5%     100%
## 216.3015 222.1620 233.9376 240.0890 263.9001 285.5751 348.3497
```

```
Cl <- quantile(diff_estimates[,2], seq(0,0.03,by=.005))[6]
Cu <- quantile(diff_estimates[,2], seq(0.97,1,by=.005))[2]
int <- c(Cl, Cu)
int
```

```
##      2.5%     97.5%
## -226.1844  222.1620
```

```
# Comment:
# int gives us [-236.5608, 235.6642], therefore we have confidence interval of int
# for this range at alpha = 0.05 significance level.
```

4 GRADIENT DESCENT

This section we introduce some R coding sources about optimization. We land on gradient descent in the end as a final section.

4.1 Optimization

The goal of this lab is to write a simple optimization function in **R** which estimates the global minimum of a convex differentiable function $f(x)$. Specifically, consider the function

$$f(x) = \frac{-\log(x)}{1+x}, \quad x > 0,$$

where $\log(x)$ is the natural logarithm of x. We seek to estimate the value of $x > 0$ such that $f(x)$ achieves its global minimum. For example, the global minimum of the function $g(x) = x^2 - 4x + 3$ is at $x = 2$. The minimum of $g(x)$ can easily be computed using the vertex formula for quadratic functions, i.e.,

$x = -b/(2a) = 4/(2 * 1) = 2$. In most cases, the minimum does not have a closed form solution and must be computed numerically. Hence we seek to estimate the global minimum of $f(x)$ numerically via gradient descent.

4.2 Tasks

1) Using **R**, define the function

$$f(x) = \frac{-\log(x)}{1+x}, \quad x > 0.$$

Test the points $f(0)$ and $f(2)$.

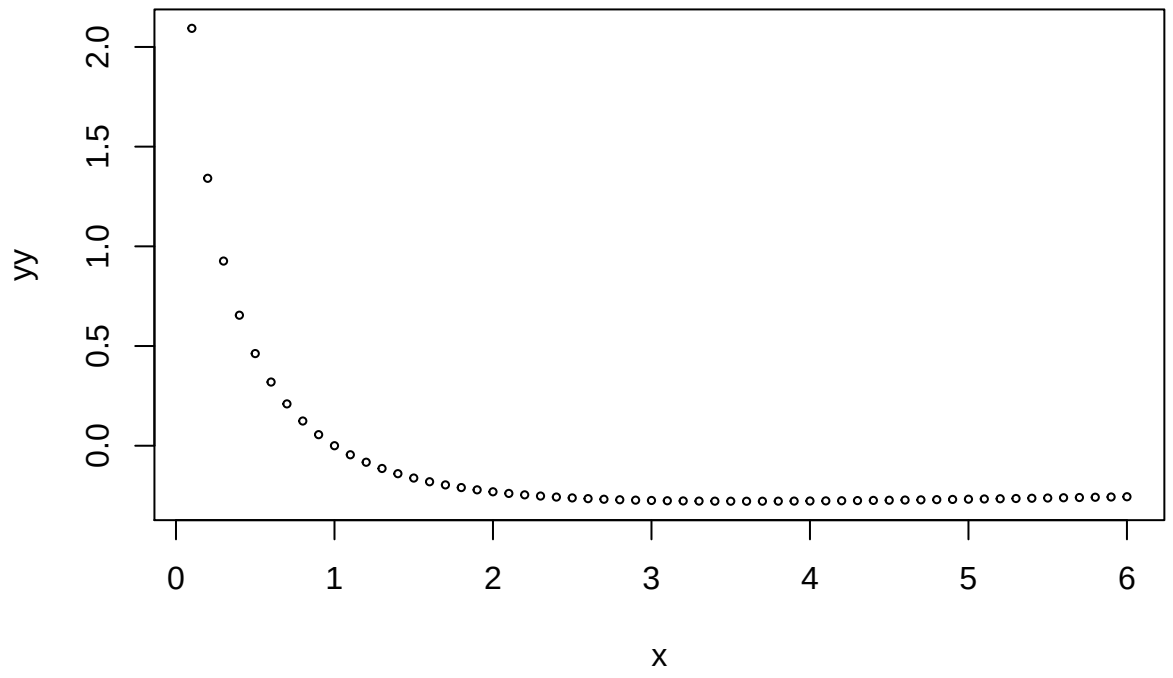
```
# Define a function named y
y <- function(x){
  y <- -log(x)/(1+x)
}
y(0) # Answer is Inf
y(1) # Answer is 0
```

2) Plot the function $f(x)$ over the interval $(0, 6]$.

```
# Plot the function
x <- data.frame(seq(0,6,0.1)); dim(x)
```

```
## [1] 61  1
```

```
yy <- data.frame(y(x))
x <- x[-1,] # Get rid of x == 0 entry
yy <- yy[-1,] # Get rid of y(0) entry
plot(x,yy,cex=0.5)
```



3) By inspection, were do you think global minimum is located at?

```
# Comment:
# Based on the graph from part 2) above,
# I would suspect global minimum is at
# x -> infinity
```

4) Define a **R** function which computes the difference quotient of $f(x)$, i.e., for $h > 0$,

$$\frac{f(x+h) - f(x)}{h}.$$

This function should have two inputs; h and x. Name the difference quotient function **diff.quot**. Note that for small h, this function is the approximate derivative of $f(x)$.

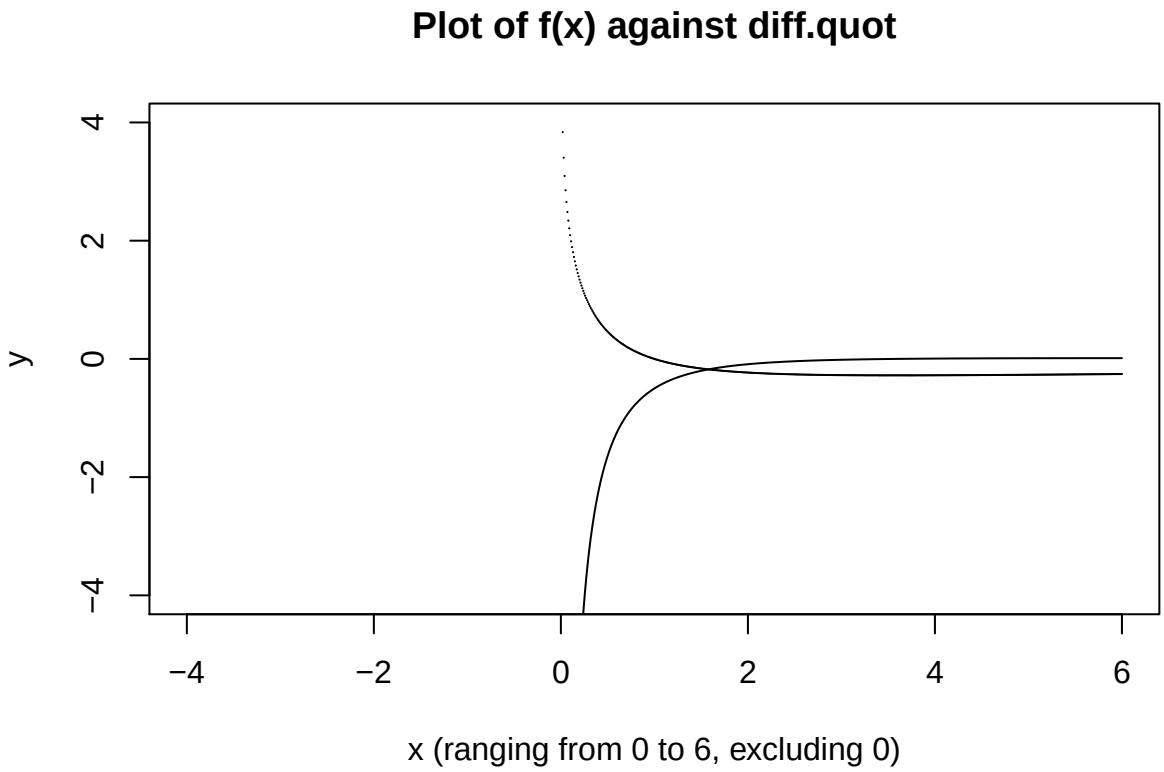
```
# Define a diff.quot function:
x <- 1
h <- 0.1
diff.quot <- function(x,h){
  diff <- (y(x+h) - y(x))/h
}
diff.quot(x,h)
# Comment:
# The first term is y(x+h), the
# second term is y(x). The last term is
# the final result.
```

5) Plot both the difference quotient function **diff.quot** and $f(x)$ over the interval $(0, 6]$. Fix $h = .0001$ to construct this plot. Comment on any interesting features.

```r
x <- data.frame(seq(0,6,0.01)); dim(x)
```

```
## [1] 601   1
```

```r
yy <- data.frame(y(x))
x <- x[-1,] # Get rid of x == 0 entry
yy <- yy[-1,] # Get rid of y(0) entry
h <- 0.0001 # Fix h
diff.yy <- diff.quot(x,h)
plot(x, yy, cex=0.01,
     xlab="x (ranging from 0 to 6, excluding 0)",
     xlim=c(-4,6),
     ylab="y",
     ylim=c(-4,4),
     main="Plot of f(x) against diff.quot")
lines(x, diff.yy)
```




```r
# Comment:
# Observe that there is an intersection between the two
# plots. Moreover, I observe that both functions converges
# to y=0 area.
```

6) Write a **R** function named **basic.grad.descent** that runs the basic gradient descent algorithm on the function $f(x)$. The function should have inputs:

   (a) Initial value **x**
   (b) Maximum iterations **max.iter** with default 10000.

(c) Stopping criterion **stop.deriv** with default 1e-10.
(d) Derivative step size **h** with default .0001.
(e) Step size **step.scale** with default .5.

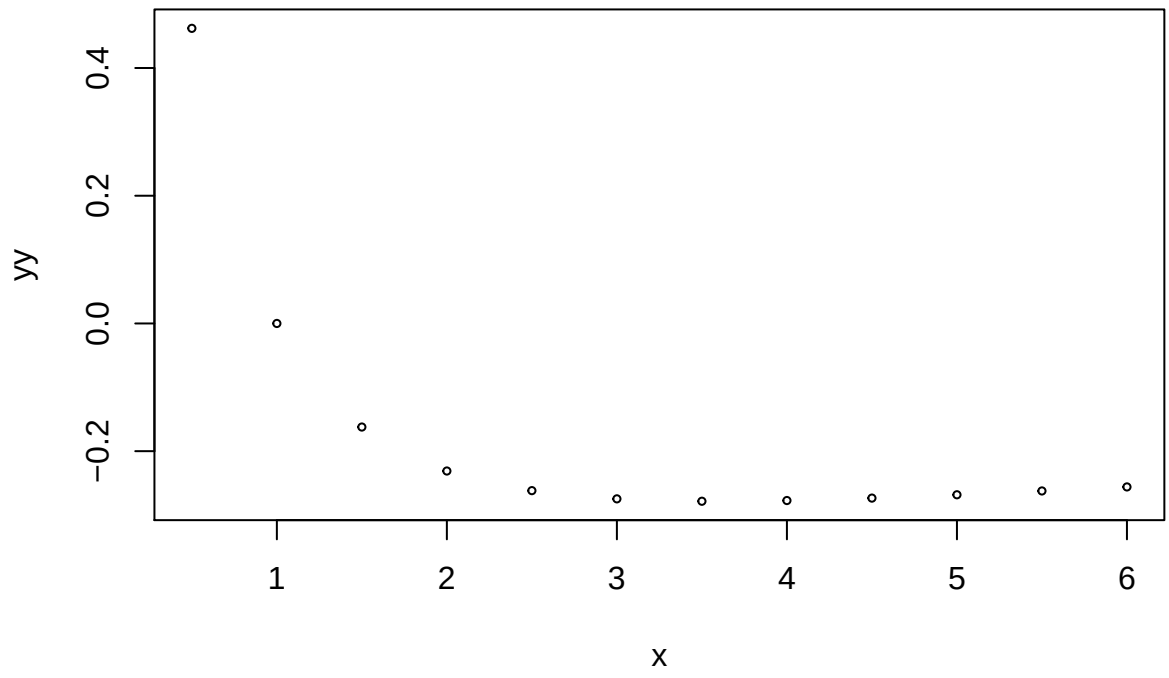
The function should have outputs:

(a) The value x that yields the minimum of $f(x)$.
(b) The minimum value of $f(x)$.
(c) The number of iterations the algorithm took to reach the minimum.
(d) A logical indicator displaying whether or not the algorithm converged.

```
# This answer I code with materials from class;
# and then I provide some extra sources taken from R-blogger:

# ------------------------------------------------------
# Code is here:

# Recall function:
# Define a function named y
y <- function(x){
  y <- -log(x)/(1+x)
}

# Recall data:
# x taken from (0,6]
# y is the range taken from the function:
x <- data.frame(seq(0,6,0.5)); dim(x)
```

```
## [1] 13  1
```

```
yy <- data.frame(y(x))
x <- x[-1,] # Get rid of x == 0 entry
yy <- yy[-1,] # Get rid of y(0) entry
plot(x,yy,cex=0.5)
```




```
# Parameters
max.iter         <- 4000000         # How long we run the alg.
stop.deriv       <- 1/10000000    # If derivative is small, stop
deriv.step       <- 1/10        # This is h
step.scale       <- 1/10         # This is c

# Initializations
iter             <- 0       # Compare to max.iteration
deriv            <- Inf     # Compare to stop.deriv
x                <- 1       # Arbitrary start

while((iter < max.iter) & (deriv > stop.deriv)) {
  iter  <- iter + 1
  deriv <- (y(x+deriv.step) - y(x))^2/deriv.step
  x     <- x + step.scale*deriv
}
list(
  x = x,
  min.y = y(x),
  iteration = iter,
  converged = (iter < max.iter)
)
```

```
## $x
## [1] 3.475576
##
## $min.y
```

```
## [1] -0.2783463
##
## $iteration
## [1] 4e+06
##
## $converged
## [1] FALSE
```

```
# Comment:
# observe that we need an iteration of 4e+06 rounds to
# arrive a minimum y value of 3.5.
# With Newton's Method from the last part,
# we are able to achieve more optimal results with
# a lot less rounds.
```

```
# ------------------------------------------------------
# Here is an alternative way to code Gradient Descent in R
# from R-blogger:
# https://www.r-bloggers.com/regression-via-gradient-descent-in-r/
# Define a function named y
fct.of.x <- function(x){
  y <- -log(x)/(1+x)
  print(y)
}
x0 <- c(1,1,1,1,1) # column of 1's
x1 <- c(1,2,3,4,5) # original x-values

# create the x- matrix of explanatory variables

x <- as.matrix(cbind(x0,x1))

# create the y-matrix of dependent variables

y <- as.matrix(fct.of.x(x[,2]))
```

```
## [1]  0.0000000 -0.2310491 -0.2746531 -0.2772589 -0.2682397
```

```
m <- nrow(y)

# implement feature scaling
x.scaled <- x
x.scaled[,2] <- (x[,2] - mean(x[,2]))/sd(x[,2])

# analytical results with matrix algebra
solve(t(x)%*%x)%*%t(x)%*%y # w/o feature scaling
```

```
##           [,1]
## x0 -0.03543340
## x1 -0.05826891
```

```
solve(t(x.scaled)%*%x.scaled)%*%t(x.scaled)%*%y # w/ feature scaling
```

```
##           [,1]
## x0 -0.21024013
## x1 -0.09213124
```

```r
# results using canned lm function match results above
summary(lm(y ~ x[, 2])) # w/o feature scaling
```

```
##
## Call:
## lm(formula = y ~ x[, 2])
##
## Residuals:
##        1        2        3        4        5
##  0.09370 -0.07908 -0.06441 -0.00875  0.05854
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.03543    0.09120  -0.389    0.724
## x[, 2]      -0.05827    0.02750  -2.119    0.124
##
## Residual standard error: 0.08696 on 3 degrees of freedom
## Multiple R-squared:  0.5995, Adjusted R-squared:  0.466
## F-statistic:  4.49 on 1 and 3 DF,  p-value: 0.1243
```

```r
summary(lm(y ~ x.scaled[, 2])) # w/feature scaling
```

```
##
## Call:
## lm(formula = y ~ x.scaled[, 2])
##
## Residuals:
##        1        2        3        4        5
##  0.09370 -0.07908 -0.06441 -0.00875  0.05854
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)   -0.21024    0.03889  -5.406   0.0124 *
## x.scaled[, 2] -0.09213    0.04348  -2.119   0.1243
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.08696 on 3 degrees of freedom
## Multiple R-squared:  0.5995, Adjusted R-squared:  0.466
## F-statistic:  4.49 on 1 and 3 DF,  p-value: 0.1243
```

```r
# define the gradient function dJ/theta: 1/m * (h(x)-y))*x where h(x) = x*theta
# in matrix form this is as follows:
grad <- function(x, y, theta) {
  gradient <- (1/m)* (t(x) %*% ((x %*% t(theta)) - y))
  return(t(gradient))
}

# define gradient descent update algorithm
grad.descent <- function(x, maxit){
    theta <- matrix(c(0, 0), nrow=1) # Initialize the parameters

    alpha = .05 # set learning rate
    for (i in 1:maxit) {
      theta <- theta - alpha  * grad(x, y, theta)
```

```
    }
 return(theta)
}

# results without feature scaling
print(grad.descent(x,1000))
```

```
##               x0          x1
## [1,] -0.03542971 -0.05826993
```

```
# results with feature scaling
print(grad.descent(x.scaled,1000))
```

```
##              x0          x1
## [1,] -0.2102401 -0.09213124
```

```
# ---------------------------------------------------------------------
# cost and convergence intuition
# ---------------------------------------------------------------------

# typically we would iterate the algorithm above until the
# change in the cost function (as a result of the updated b0 and b1 values)
# was extremely small value 'c'. C would be referred to as the set 'convergence'
# criteria. If C is not met after a given # of iterations, you can increase the
# iterations or change the learning rate 'alpha' to speed up convergence

# get results from gradient descent
beta <- grad.descent(x,1000)

# define the 'hypothesis function'
h <- function(xi,b0,b1) {
 b0 + b1 * xi
}

# define the cost function
cost <- t(mat.or.vec(1,m))
for(i in 1:m) {
  cost[i,1] <-  (1 /(2*m)) * (h(x[i,2],beta[1,1],beta[1,2])- y[i,])^2
}

totalCost <- colSums(cost)
print(totalCost)
```

```
## [1] 0.002268575
```

```
# save this as Cost1000
cost1000 <- totalCost

# change iterations to 1001 and compute cost1001
beta <- (grad.descent(x,1001))
cost <- t(mat.or.vec(1,m))
for(i in 1:m) {
  cost[i,1] <-  (1 /(2*m)) * (h(x[i,2],beta[1,1],beta[1,2])- y[i,])^2
}
cost1001 <- colSums(cost)
```

```r
# does this difference meet your convergence criteria?
print(cost1000 - cost1001)
```

```
## [1] 2.084617e-14
```

```r
# Comment: The final result is really small. If we keep increase
# iterations, we would simply go close to 0, which means a
# convergence in mathemtaical terms.
```

7) Check the optimal value using the base **R** function **nlm()**.

```r
# Learn about the function nlm()
# ?nlm()
# An example from ?nlm()
f <- function(x) sum((x-1:length(x))^2)
f(3)
```

```
## [1] 4
```

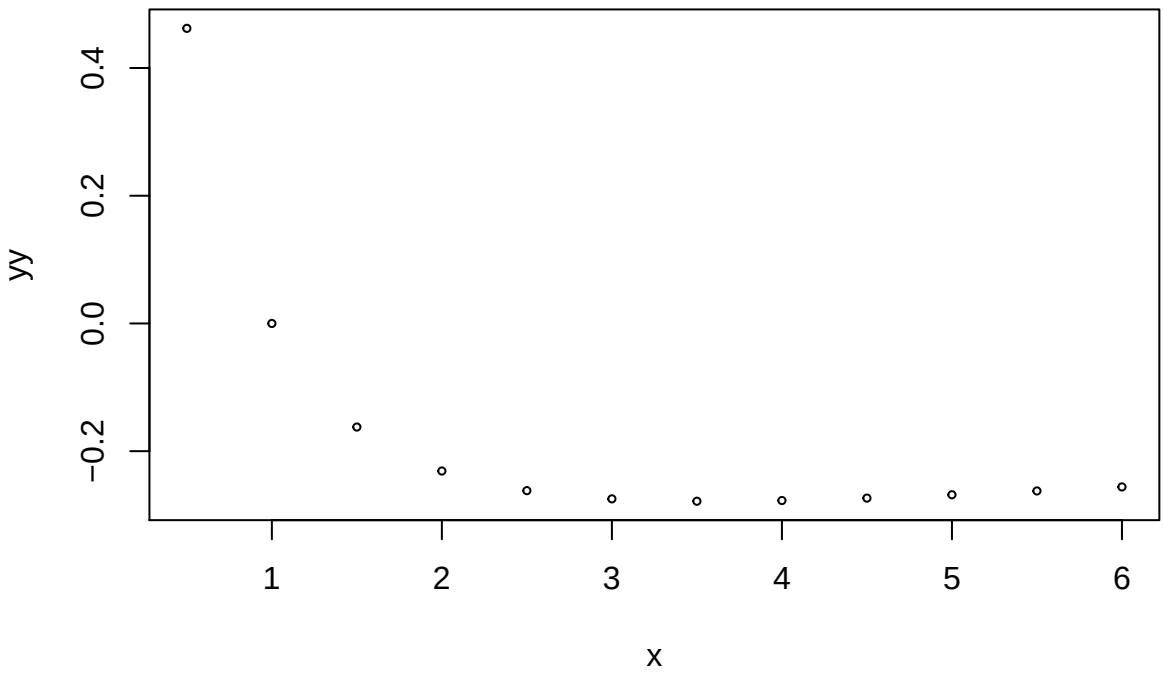
```r
nlm(f, c(10,10))
```

```
## $minimum
## [1] 4.303458e-26
##
## $estimate
## [1] 1 2
##
## $gradient
## [1]  2.757794e-13 -3.099743e-13
##
## $code
## [1] 1
##
## $iterations
## [1] 2
```

```r
# Example ends here.

# Next, we test on our functions:
# x taken from (0,6]
# y is the range taken from the function:
y <- function(x){-log(x)/(1+x)}
x <- data.frame(seq(0,6,0.5)); dim(x)
```

```
## [1] 13  1
```

```r
yy <- data.frame(y(x))
x <- x[-1,] # Get rid of x == 0 entry
yy <- yy[-1,] # Get rid of y(0) entry
plot(x,yy,cex=0.5)
```




```
# Now we test nlm() function
nlm(y, c(1))
```

```
## $minimum
## [1] -0.2784645
##
## $estimate
## [1] 3.591119
##
## $gradient
## [1] -1.273731e-08
##
## $code
## [1] 1
##
## $iterations
## [1] 10
```

5 GRAPHICS

5.1 Background

We consider a dataset containing information about the world's richest people. The dataset is taken from the World Top Incomes Database (WTID) hosted by the Paris School of Economics [http://topincomes.g-mond.parisschoolofeconomics.eu]. This is derived from income tax reports, and compiles information about the

very highest incomes in various countries over time, trying as hard as possible to produce numbers that are comparable across time and space.

For most countries in most time periods, the upper end of the income distribution roughly follows a Pareto distribution, with probability density function

$$f(x) = \frac{(a-1)}{x_{min}} \left(\frac{x}{x_{min}} \right)^{-a}$$

for incomes $X \geq x_{min}$. (Typically, x_{min} is large enough that only the richest 3%-4% of the population falls above it.) As the *Pareto exponent*, a, gets smaller, the distribution of income becomes more unequal, that is, more of the population's total income is concentrated among the very richest people.

The proportion of people whose income is at least x_{min} whose income is also at or above any level $w \geq x_{min}$ is thus

$$\mathbf{Pr}(X \geq w) = \int_w^\infty f(x)dx = \int_w^\infty \frac{(a-1)}{x_{min}} \left(\frac{x}{x_{min}} \right)^{-a} dx = \left(\frac{w}{x_{min}} \right)^{-a+1}.$$

We will use this to estimate how income inequality changed in the US over the last hundred years or so. (Whether the trends are good or bad or a mix is beyond our scope here.) WTID exports its data sets as `.xlsx` spreadsheets. For this lab session, we have extracted the relevant data and saved it as `wtid-report.csv`.

5.1.1 Part 1

1. Open the file and make a new variable containing only the year, "P99", "P99.5" and "P99.9" variables; these are the income levels which put someone at the 99th, 99.5th, and 99.9th, percentile of income. What was P99 in 1972? P99.5 in 1942? P99.9 in 1922? You must identify these using your code rather than looking up the values manually. (You may want to modify the column names to make some of them shorter.)

```
# Solution:

# Load data:
data <- read.csv(
  "F://Course/CU Stats/STATS W4206(S) - Stat Computing Intro to Data Sci/4. Lab/Lab6/wtid-report.csv",
  header = T, sep = ",")
head(data, 3); dim(data) # Quick view
```

```
##         Country Year P99.income.threshold P99.5.income.threshold
## 1 United States 1913             82677.22               135583.5
## 2 United States 1914             76405.62               126910.5
## 3 United States 1915             64409.44               122555.7
##   P99.9.income.threshold
## 1               428630.4
## 2               410528.7
## 3               451668.3
```

```
## [1] 103   5
```

```
# P99 in 1972?
data[data$Year==1972,3] # Just for P99
```

```
## [1] 215836.3
```

```
data[data$Year==1972,] # For all percentile in 1972
```

```
##          Country Year P99.income.threshold P99.5.income.threshold
## 60 United States 1972             215836.3               285575.6
##    P99.9.income.threshold
## 60               504206.7
```

```
# Comment:
# P99 = 21586.3, P99.5 = 285575.6, P99.9 = 504206.7

# P99.5 in 1942:
data[data$Year==1942, 4] # Just for P99.5
```

```
## [1] 189140.6
```

```
data[data$Year==1942,] # For all percentile in 1942
```

```
##          Country Year P99.income.threshold P99.5.income.threshold
## 30 United States 1942             120306.1               189140.6
##    P99.9.income.threshold
## 30               499711.5
```

```
# Comment:
# P99 = 120306.1; P99.5 = 189140.6, P99.9 = 499711.5

# P99.9 in 1922:
data[data$Year==1922, 5] # Just for P99.9
```

```
## [1] 413153.5
```

```
data[data$Year==1922, ] # For all percentile in 1922
```

```
##          Country Year P99.income.threshold P99.5.income.threshold
## 10 United States 1922             91634.52               143497.2
##    P99.9.income.threshold
## 10               413153.5
```

```
# Comment:
# P99 = 91634.52, P99.5 = 143497.2, P99 = 413453.5
```

2. Plot the three percentile levels against time using `ggplot`. Make sure the axes are labeled appropriately, and in particular that the horizontal axis is labeled with years between 1913 and 2012, not just numbers from 1 to 100. Remember `library(ggplot2)`. In my plot I used multiple layers of `geom_line` and didn't include a lengend (but plotted the years in different colors).

```
library(ggplot2)
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     diamonds
```

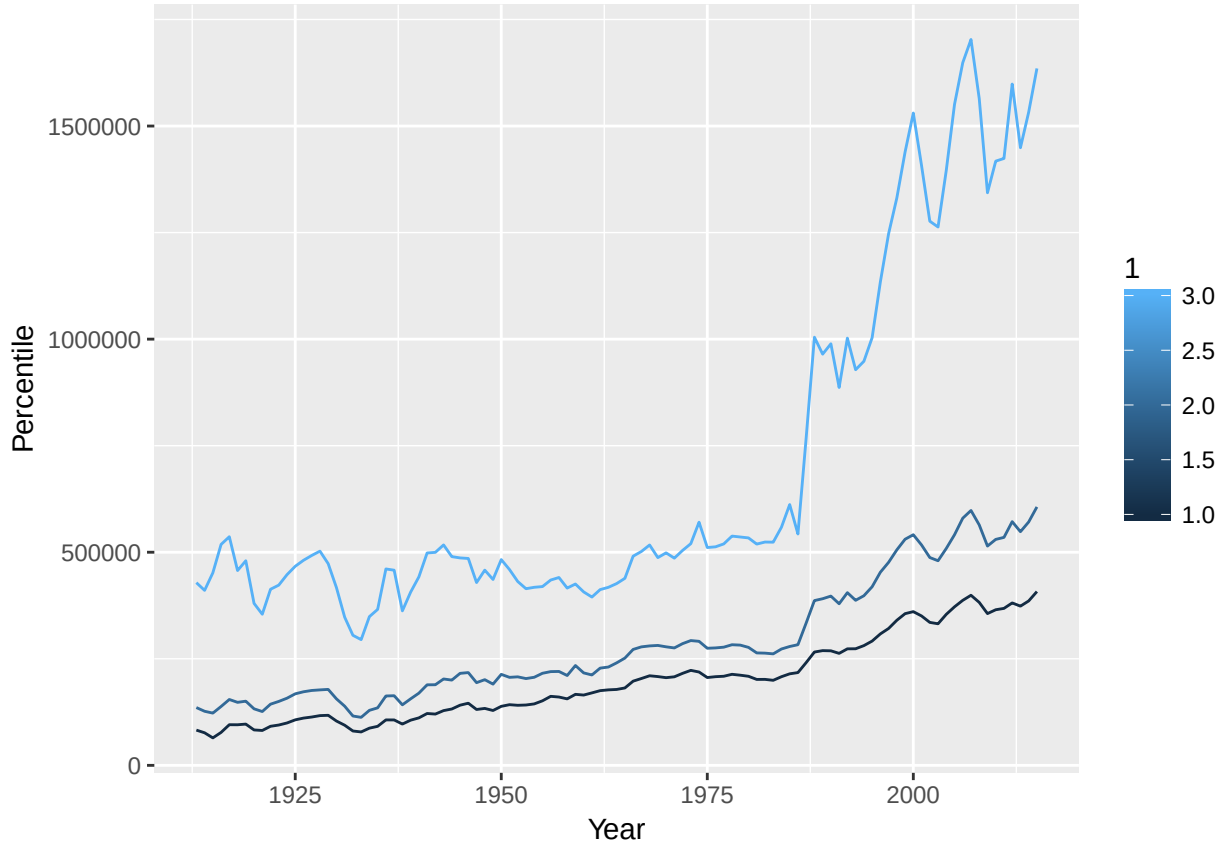
```
# Update the names:
names(data) <- c("Country", "Year", "P99", "P99.5", "P99.9")
head(data,3); names(data) # Check the variable names
```

```
##         Country Year      P99    P99.5    P99.9
## 1 United States 1913 82677.22 135583.5 428630.4
```

```
## 2 United States 1914 76405.62 126910.5 410528.7
## 3 United States 1915 64409.44 122555.7 451668.3
```

```
## [1] "Country" "Year"    "P99"     "P99.5"   "P99.9"
```

```
# Plot:
ggplot(data = data) +
  geom_line(mapping = aes(x=Year, y=P99, color = 1)) +
  geom_line(mapping = aes(x=Year, y=P99.5, color = 2)) +
  geom_line(mapping = aes(x=Year, y=P99.9, color = 3)) +
  labs(x = "Year", y = "Percentile")
```



3. One can show from the earlier equations that one can estimate the exponent by the formula

$$a = 1 - \frac{\log 10}{\log\left(\frac{P99}{P99.9}\right)} \tag{1}$$

Write a function, `exponent.est_ratio()` which takes in values for `P99` and `P99.9`, and returns the value of a implied by (1). Check that if `P99=1e6` and `P99.9=1e7`, your function returns an a of 2.

```
# Solution:

# Write a function:
exponent.est_ratio <- function(P99,P99.9){
  a <- 1 - log(10)/log(P99/P99.9)
  return(a)
}

# Example:
```

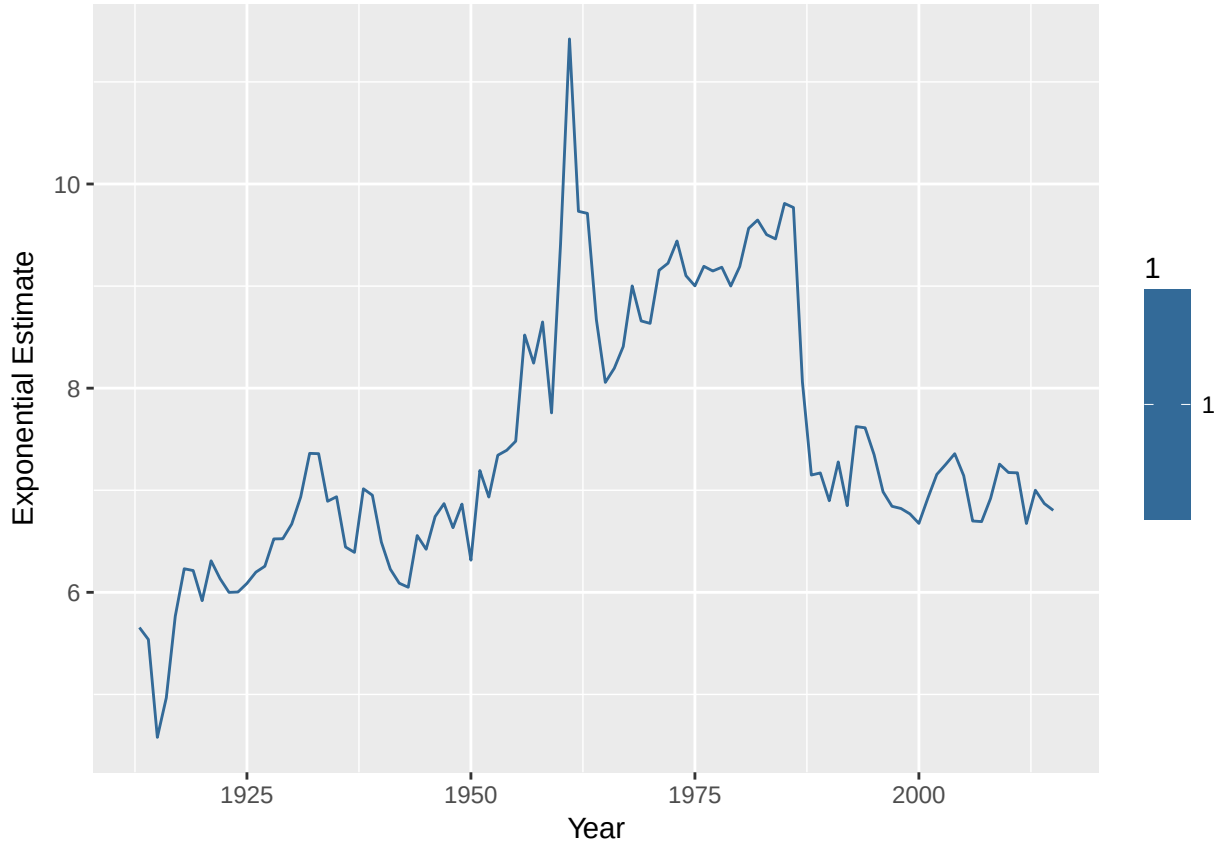
```
exponent.est_ratio(1e6, 1e7) # Function returns 2.
```

```
## [1] 2
```

5.1.2 Part 2

4. Estimate a for each year in the data set, using your `exponent.est_ratio()` function. If the function was written properly, you should not need to use a loop. Plot your estimate of a over time using `ggplot`. Do the results look reasonable? (Remember that smaller exponents mean more income inequality.)

```
# Solution:
# head(data, 3); dim(data) # Quick view

# How many different years?
# levels(factor(data$Year))
# exponent.est_ratio(data$P99, data$P99.5)

# Plot:
ggplot(data = data) +
  geom_line(mapping = aes(x=Year, y=exponent.est_ratio(data$P99, data$P99.5) , color = 1)) +
  labs(x = "Year", y = "Exponential Estimate")
```




```
# Comment:
# There is a sharp rise during the 1950 - 1975
# period. This makes sense in reality because
# we had several crisis during this period
```

```
# and we also had two world wars.
```

5. The logic leading to (1) also implies that

$$\left(\frac{P99.5}{P99.9}\right)^{-a+1} = 5$$

Write a function which takes `P99.5`, `P99.9`, and a, and calculates the left-hand side of that equation. Plot the values for each year using `ggplot`, using the data and your estimates of the exponent. Add a horizontal line with vertical coordinate 5. How good is the fit?

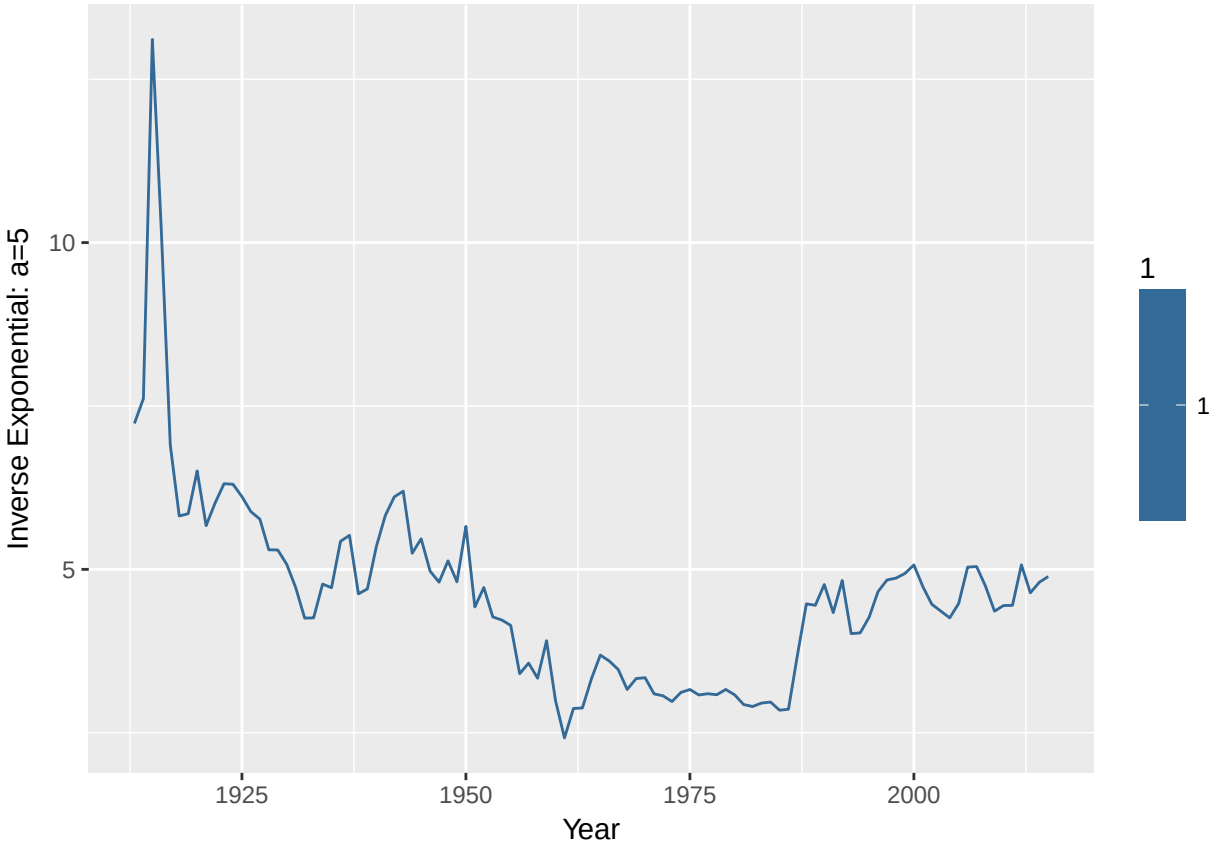
```
# Solution:
# Write a function:
inverse.exponential <- function(
  numerator,
  denominator,
  a){
  left <- (numerator/denominator)^(-a+1)
  left
}

# Example:
(3/2)^(-5+1)
```

```
## [1] 0.1975309
```

```
inverse.exponential(
  numerator = 3,
  denominator = 2,
  a = 5) # Two results match.
```

```
## [1] 0.1975309
```

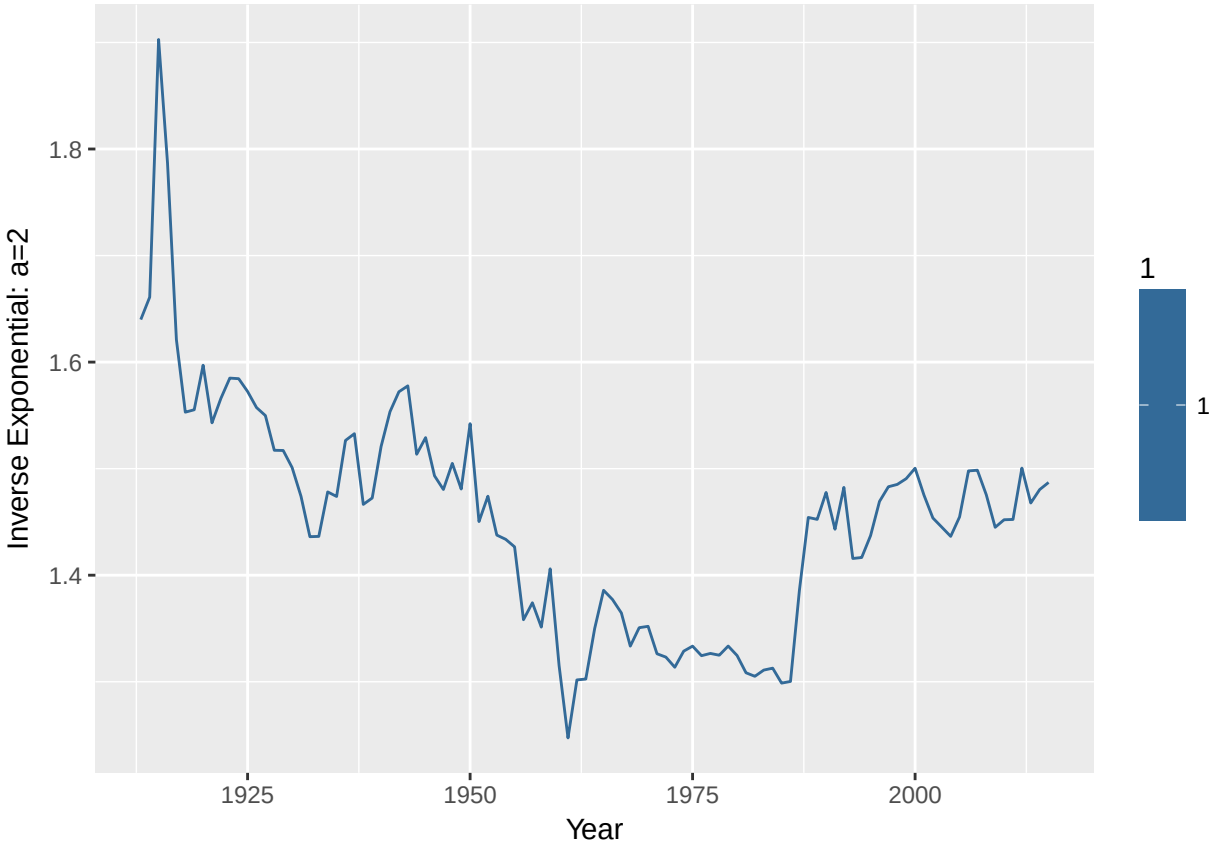
```
# Plot:
ggplot(data = data) +
  geom_line(mapping = aes(
    x=Year,
    y=inverse.exponential(data$P99, data$P99.5, a=5),
    color = 1)) +
  labs(x = "Year", y = "Inverse Exponential: a=5")
```



6. By parallel reasoning, we should have $(P99/P99.5)^{-a+1} = 2$. Repeat the previous step with this formula. How would you describe this fit compared to the previous ones?

(Note: the formula in (1) is not the best way to estimate a, but it is one of the simplest.)

```
# Solution:
# Repeat (5) with a = 2.
# Plot:
ggplot(data = data) +
  geom_line(mapping = aes(
    x=Year,
    y=inverse.exponential(data$P99, data$P99.5, a=2),
    color = 1)) +
  labs(x = "Year", y = "Inverse Exponential: a=2")
```




```
# Comment:
# Observe that this graph takes similar shape as
# the one above in part (5). However, the y-axis
# takes less value for the inverse exponential values
# of each year. We see a smaller variation while
# taking a=2.
```

6 KNN

6.1 Data Clean-up

6.1.1 (a) Load Data

```
# Load data:
rich <- readLines("rich.html")
rich_df <- read.table("rich_dataframe.txt", header = TRUE, as.is = TRUE)

# Quick view:
head(rich)
```

```
## [1] ""
## [2] "<!DOCTYPE HTML>"
## [3] "<html xmlns=\"http://www.w3.org/1999/xhtml\" xmlns:fb=\"http://www.facebook.com/2008/fbml/\" xml
## [4] "<head>"
```

```
## [5] ""
## [6] "\t<title>The Richest People in America List - Forbes</title>"
```

```
head(rich_df)
```

```
##   Rank                    Name Age Networths Worth             Location
## 1    1              Bill Gates  58     $72 B  72.0   Medina, Washington
## 2    2          Warren Buffett  84   $58,5 B  58.5      Omaha, Nebraska
## 3    3           Larry Ellison  70     $41 B  41.0 Woodside, California
## 4    4            Charles Koch  78     $36 B  36.0      Wichita, Kansas
## 5    5              David Koch  74     $36 B  36.0   New York, New York
## 6    6 Christy Walton & family  59   $35,4 B  35.4     Jackson, Wyoming
##              Industry Technology
## 1           Microsoft          1
## 2 Berkshire Hathaway          0
## 3              Oracle          1
## 4         diversified          0
## 5         diversified          0
## 6            Wal-Mart          0
```

```
# Comment: two datasets are loaded into R.
```

6.1.2 (b) Extract Networth People

```
worth_lines <- grep("td class=\"worth\"", rich)
length(worth_lines)
```

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

```
# Comment:
# Provided code above find lines that satisfy conidtioning
# pattern and there are 100 of them from length()

# Next:
networths <- gregexpr("\\$[0-9]+,?[0-9]? B", rich[worth_lines])
networths <- unlist(regmatches(rich[worth_lines], networths))
length(networths)
```

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

```
# Check:
networths[1:5]
```

```
## [1] "$72 B"   "$58,5 B" "$41 B"   "$36 B"   "$36 B"
```

6.1.3 (c) Convert Dollars

```
networths2 <- substring(networths, 2, nchar(networths)-2)
networths2 <- strsplit(networths2, ",")
networths2 <- sapply(networths2, paste, collapse = ".")
networths2 <- as.numeric(networths2)

networths2[1:5]
```

```
## [1] 72.0 58.5 41.0 36.0 36.0
```

```r
# Check:
identical(networths2, rich_df$Worth)
```

```
## [1] TRUE
```

6.1.4 (d) Add a Column, Worth

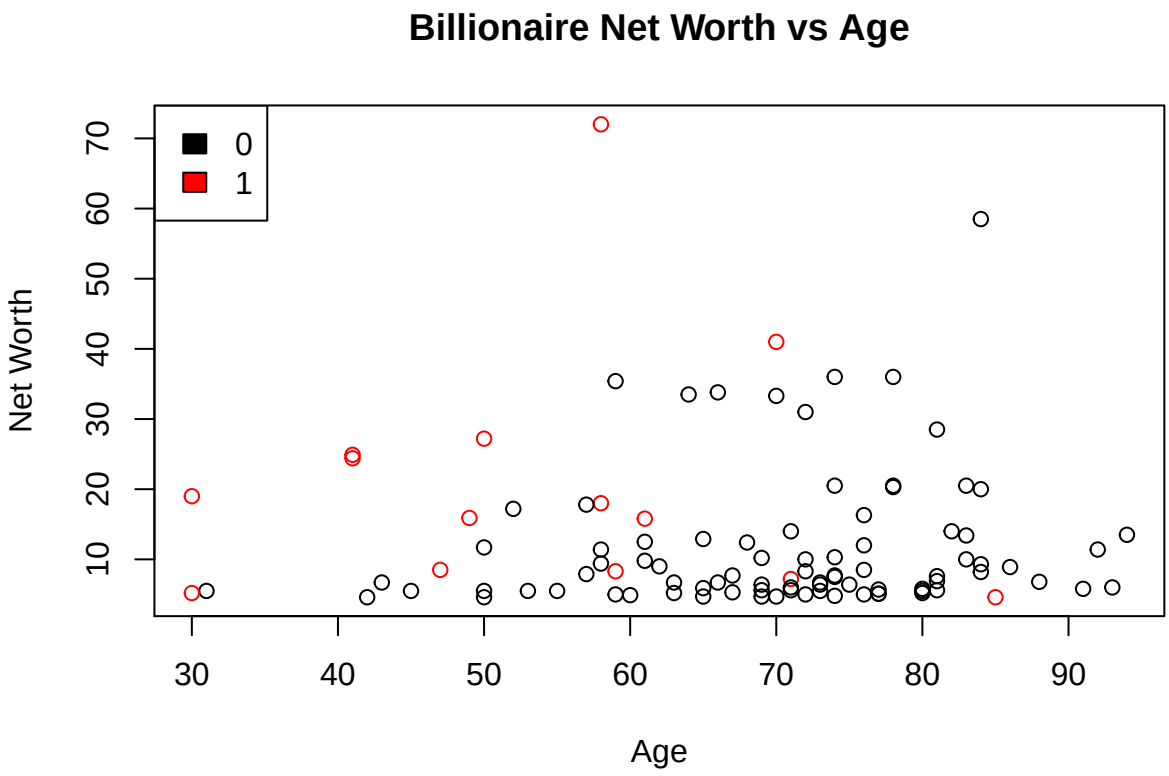
```r
rich_df$MyWorth <- networths2
head(rich_df)
```

```
##   Rank                    Name Age Networths Worth             Location
## 1    1              Bill Gates  58     $72 B  72.0   Medina, Washington
## 2    2          Warren Buffett  84   $58,5 B  58.5      Omaha, Nebraska
## 3    3           Larry Ellison  70     $41 B  41.0 Woodside, California
## 4    4            Charles Koch  78     $36 B  36.0      Wichita, Kansas
## 5    5              David Koch  74     $36 B  36.0   New York, New York
## 6    6 Christy Walton & family  59   $35,4 B  35.4     Jackson, Wyoming
##             Industry Technology MyWorth
## 1          Microsoft          1    72.0
## 2 Berkshire Hathaway          0    58.5
## 3             Oracle          1    41.0
## 4        diversified          0    36.0
## 5        diversified          0    36.0
## 6           Wal-Mart          0    35.4
```

6.2 KNN

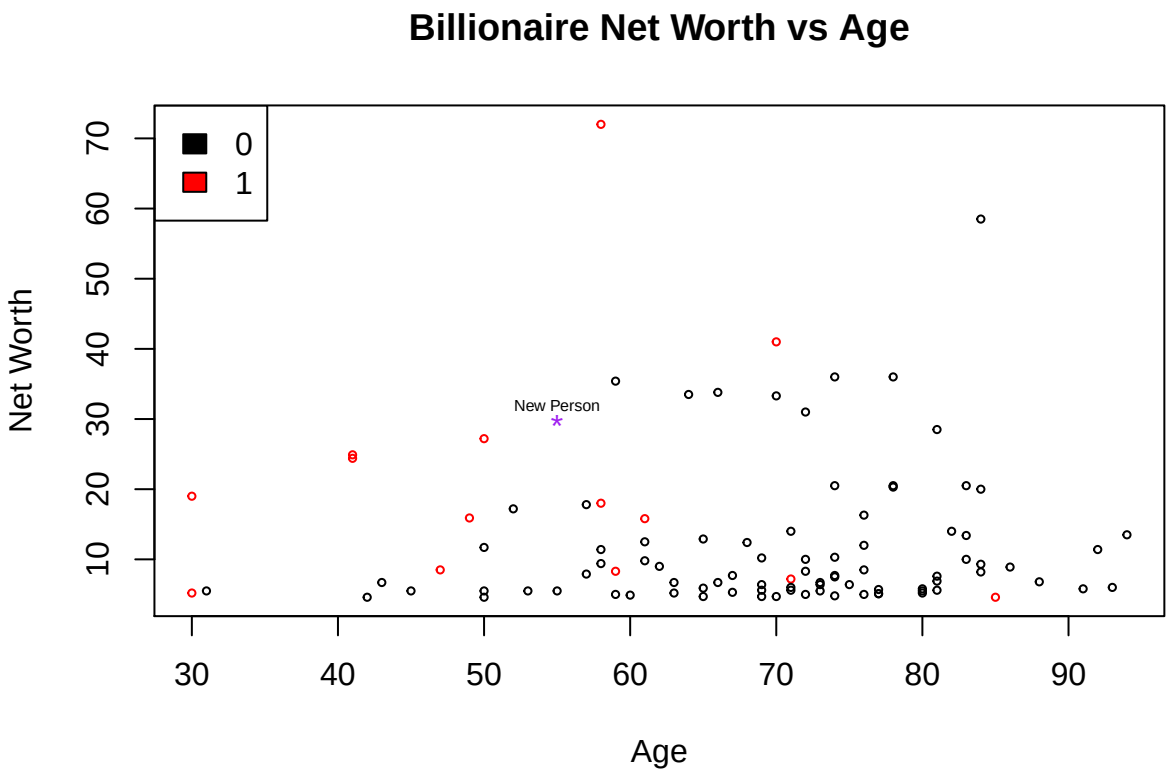
6.2.1 (a) GGPlot2

```r
plot(
  rich_df$Age,
  rich_df$Worth,
  xlab = "Age",
  ylab = "Net Worth",
  main = "Billionaire Net Worth vs Age",
  col = factor(rich_df$Technology)
)
legend(
  "topleft",
  legend = levels(factor(rich_df$Technology)),
  fill = 1:length(levels(factor(rich_df$Technology)))
)
```



6.2.2 (b) Add New Person

```
plot(
  rich_df$Age,
  rich_df$Worth,
  xlab = "Age",
  ylab = "Net Worth",
  main = "Billionaire Net Worth vs Age",
  col = factor(rich_df$Technology),
  cex = .5
)
legend(
  "topleft",
  legend = levels(factor(rich_df$Technology)),
  fill = 1:length(levels(factor(rich_df$Technology)))
)
points(55, 30, pch = "*", col = "purple")
text(55, 30+2, "New Person", cex = 0.5)
```



6.2.3 (c) KNN Classification: a toy ex.

Let's also imagine that we don't know whether the new millionaire's industry is technology or not and we'd like to make a prediction using a KNN Classification scheme. Suppose that the Euclidean distances between the new point and the points in the rich_df dataset are stored in a vector called dists (meaning the first value in dists in the distance between the new point and Bill Gates' point, the second value in dists in the distance between the new point and Warren Buffet's point, and so on).

```
K.nearest.neighbs <- c(12, 6, 8, 7, 22, 21, 23)
rich_df$Technology[K.nearest.neighbs]
```

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

6.3 KNN Classification and Cross-Validation

6.3.1 Background

Today we'll be using the *Weekly* dataset from the *ISLR* package. This data is similar to the *Smarket* data from class. The dataset contains 1089 weekly returns from the beginning of 1990 to the end of 2010. Make sure that you have the *ISLR* package installed and loaded by running (without the code commented out) the following:

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

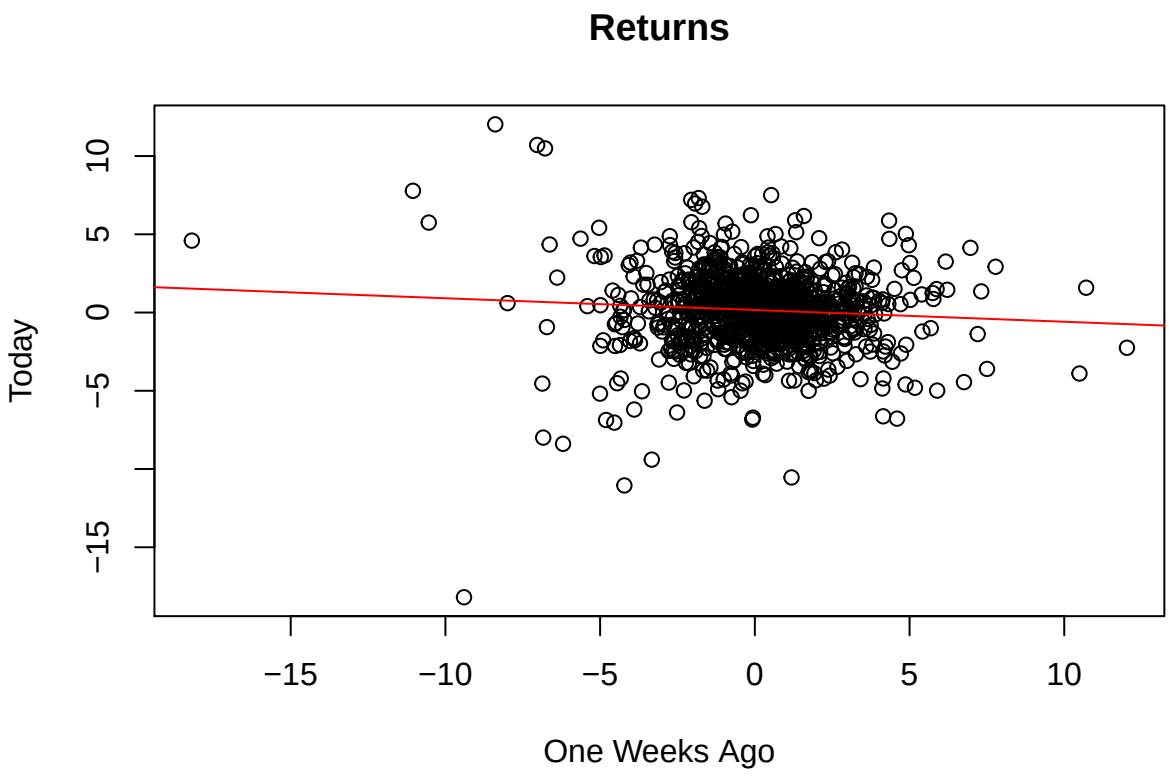
We'd like to see if we can accurately predict the direction of a week's return based on the returns over the last five weeks. *Today* gives the percentage return for the week considered and *Year* provides the year that the observation was recorded. *Lag1* - *Lag5* give the percentage return for 1 - 5 weeks previous and *Direction* is a factor variable indicating the direction ('UP' or 'DOWN') of the return for the week considered.

6.3.2 Part 1: Visualizing the relationship between this week's returns and the previous week's returns.

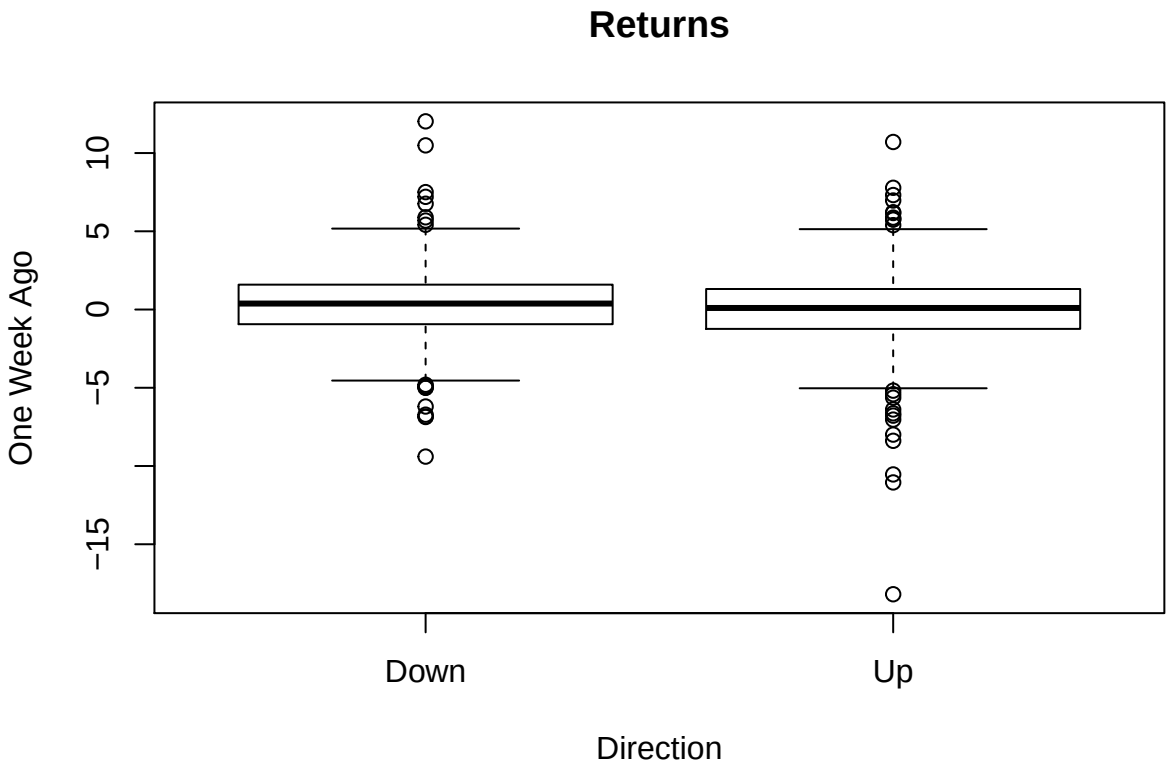
1. Explore the relationship between a week's return and the previous week's return. You should plot more graphs for yourself, but include in the lab write-up a scatterplot of the returns for the weeks considered (*Today*) vs the reutrn from two weeks previous (*Lag2*), and side-by-side boxplots for the lag one week previous (*Lag1*) divided by the direction of this wee'k reutnr (*Direction*).

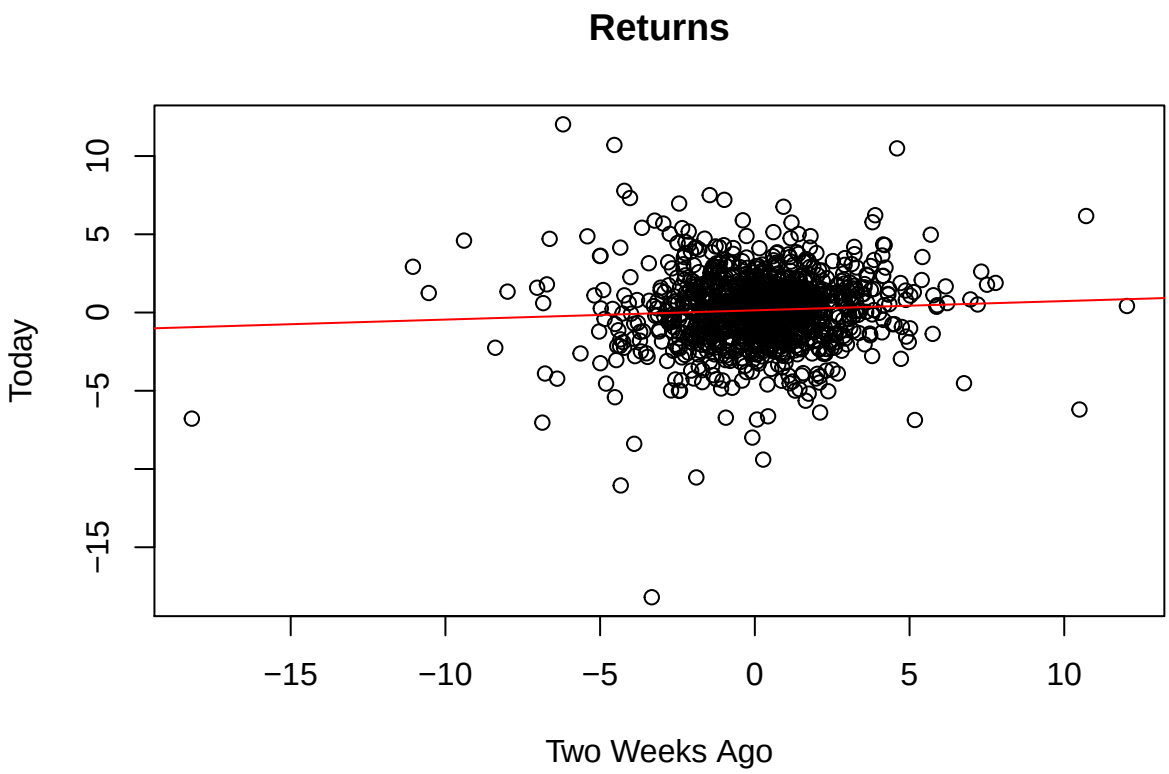
```
##   Year   Lag1   Lag2   Lag3   Lag4   Lag5    Volume  Today Direction
## 1 1990  0.816  1.572 -3.936 -0.229 -3.484 0.1549760 -0.270      Down
## 2 1990 -0.270  0.816  1.572 -3.936 -0.229 0.1485740 -2.576      Down
## 3 1990 -2.576 -0.270  0.816  1.572 -3.936 0.1598375  3.514        Up
## 4 1990  3.514 -2.576 -0.270  0.816  1.572 0.1616300  0.712        Up
## 5 1990  0.712  3.514 -2.576 -0.270  0.816 0.1537280  1.178        Up
## 6 1990  1.178  0.712  3.514 -2.576 -0.270 0.1544440 -1.372      Down
```

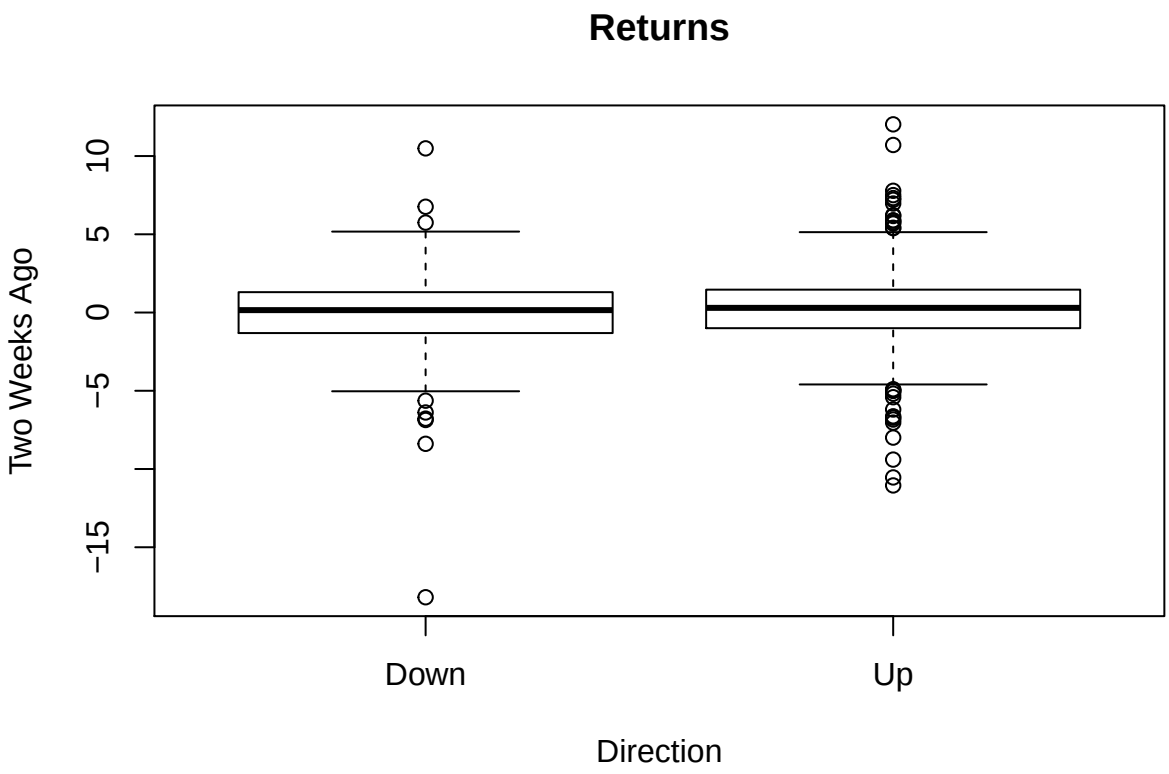
```
##      Year   Lag1   Lag2   Lag3   Lag4   Lag5   Volume  Today Direction
## 1084 2010  0.043 -2.173  3.599  0.015  0.586 4.177436 -0.861      Down
## 1085 2010 -0.861  0.043 -2.173  3.599  0.015 3.205160  2.969        Up
## 1086 2010  2.969 -0.861  0.043 -2.173  3.599 4.242568  1.281        Up
## 1087 2010  1.281  2.969 -0.861  0.043 -2.173 4.835082  0.283        Up
## 1088 2010  0.283  1.281  2.969 -0.861  0.043 4.454044  1.034        Up
## 1089 2010  1.034  0.283  1.281  2.969 -0.861 2.707105  0.069        Up
```

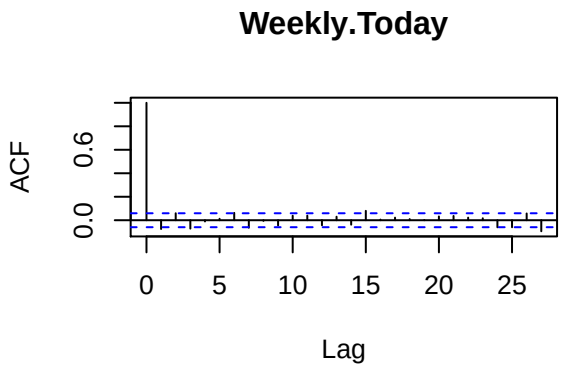


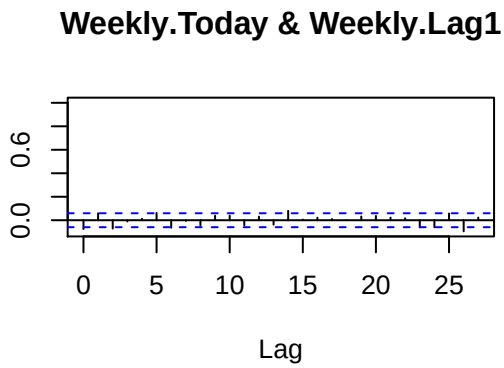

```
##
## Call:
## lm(formula = Weekly$Today ~ Weekly$Lag1)
##
## Coefficients:
## (Intercept)  Weekly$Lag1
##     0.16120     -0.07503
```

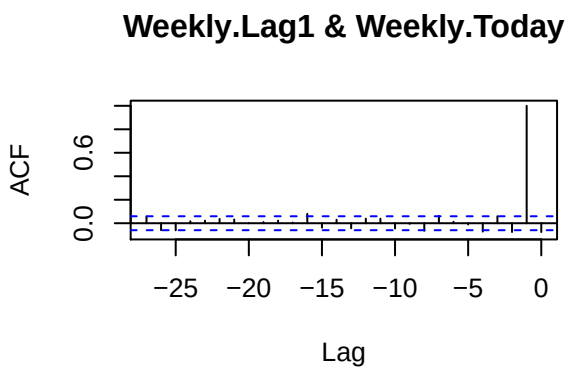

Returns
One Week Ago
10
5
0
−5
−15
Down
Up
Direction

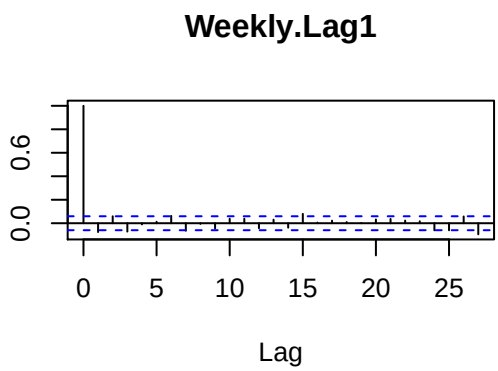

Returns
Today
Two Weeks Ago
10
5
0
−5
−15
−15
−10
−5
0
5
10

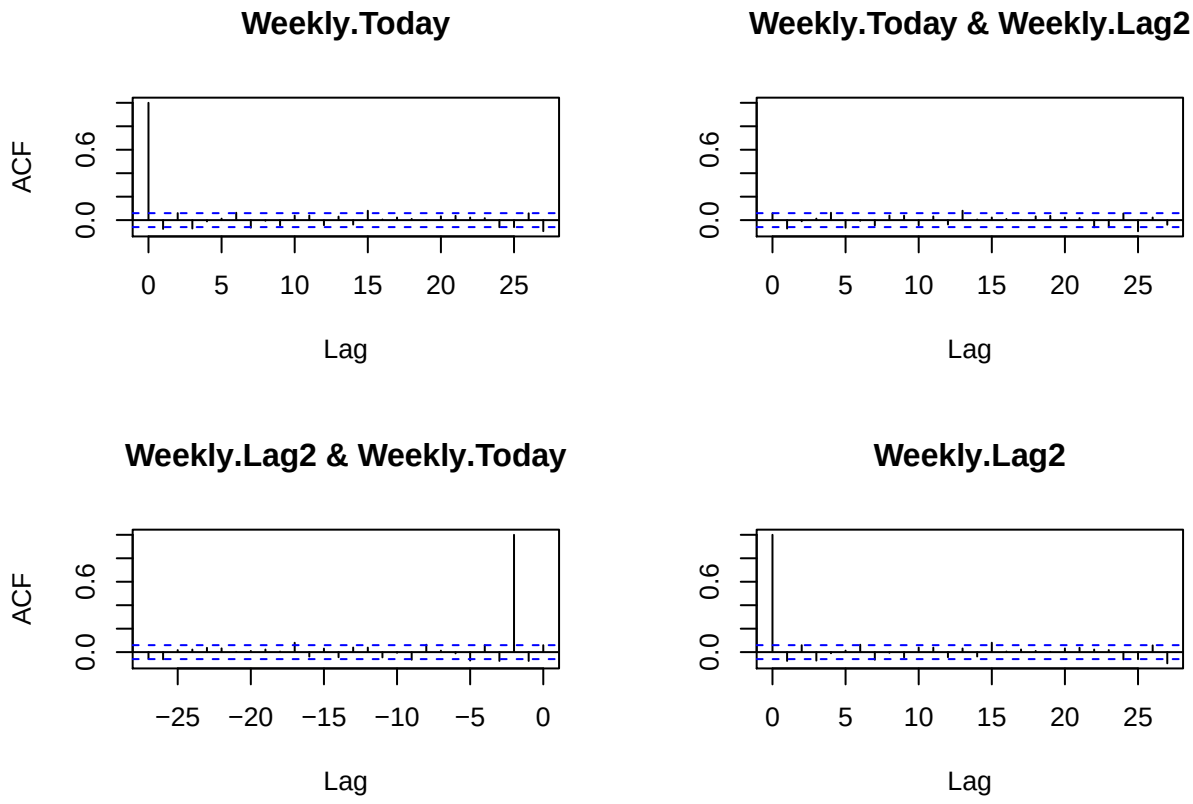

Returns
Two Weeks Ago
10
5
0
−5
−15
Down
Up
Direction


Weekly.Today
ACF
0.0
0.6
0
5
10
15
20
25
Lag


Weekly.Today & Weekly.Lag1
0.0
0.6
0
5
10
15
20
25
Lag


Weekly.Lag1 & Weekly.Today
ACF
0.0
0.6
−25
−20
−15
−10
−5
0
Lag


Weekly.Lag1
0.0
0.6
0
5
10
15
20
25
Lag



6.3.3 Part 2: Building a classifier

Recall the KNN procedure. We classify a new point with the following steps:

– Calculate the Euclidean distance between the new point and all other points.

– Create the set $\mathcal{N}_{new}$ containing the K closest points (or, nearest neighbors) to the new point.

– Determine the number of 'UPs' and 'DOWNs' in $\mathcal{N}_{new}$ and classify the new point according to the most frequent.

2. We'd like to perform KNN on the *Weekly* data, as we did with the *Smarket* data in class. In class we wrote the following function which takes as input a new point $(Lag1_{new}, Lag2_{new})$ and provides the KNN decision using as defaults $K = 5$, Lag1 data given in *Smarket$Lag1*, and Lag2 data given in *Smarket$Lag2*. Update the function to calculate the KNN decision for weekly market direction using the *Weekly* dataset with *Lag1* - *Lag5* as predictors. Your function should have only three input values: (1) a new point which should be a vector of length 5, (2) a value for K, and (3) the Lag data which should be a data frame with five columns (and n rows).

```
# Quick view:
head(Weekly)
```

```
##   Year   Lag1   Lag2   Lag3   Lag4   Lag5    Volume  Today Direction
## 1 1990  0.816  1.572 -3.936 -0.229 -3.484 0.1549760 -0.270      Down
## 2 1990 -0.270  0.816  1.572 -3.936 -0.229 0.1485740 -2.576      Down
## 3 1990 -2.576 -0.270  0.816  1.572 -3.936 0.1598375  3.514        Up
## 4 1990  3.514 -2.576 -0.270  0.816  1.572 0.1616300  0.712        Up
## 5 1990  0.712  3.514 -2.576 -0.270  0.816 0.1537280  1.178        Up
```

```
## 6 1990  1.178  0.712  3.514 -2.576 -0.270 0.1544440 -1.372      Down
```

```
# Here we update the function with Weekly datset:
KNN.decision <- function(
  Lag1.new,
  Lag2.new,
  K,
  Lag1 = Weekly$Lag1,
  Lag2 = Weekly$Lag2) {

   n <- length(Lag1)

   stopifnot(length(Lag2) == n, length(Lag1.new) == 1, length(Lag2.new) == 1, K <= n)

   dists       <- sqrt((Lag1-Lag1.new)^2 + (Lag2-Lag2.new)^2)
   neighbors  <- order(dists)[1:K]
   neighb.dir <- Weekly$Direction[neighbors]
   choice      <- names(which.max(table(neighb.dir)))
   return(choice)
}

# Example:
KNN.decision(
  Lag1.new = 0.1,
  Lag2.new = 0.2,
  K = 5,
  Lag1 = Weekly$Lag1,
  Lag2 = Weekly$Lag2
)
```

```
## [1] "Down"
```

3. Now train your model using data from 1990 - 2008 and use the data from 2009-2010 as test data. To do this, divide the data into two data frames, *test* and *train*. Then write a loop that iterates over the test points in the test dataset calculating a prediction for each based on the training data with $K = 5$. Save these predictions in a vector. Finally, calculate your test error, which you should store as a variable named *test.error*. The test error calculates the proportion of your predictions which are incorrect (don't match the actual directions).

```
# Split Dataset:
levels(factor(Weekly$Year)) # view all the variables in Year
```

```
##  [1] "1990" "1991" "1992" "1993" "1994" "1995" "1996" "1997" "1998" "1999"
## [11] "2000" "2001" "2002" "2003" "2004" "2005" "2006" "2007" "2008" "2009"
## [21] "2010"
```

```
train <- Weekly[
  Weekly$Year != 2009 && Weekly$Year != 2010,
]; head(train); dim(train)
```

```
##   Year   Lag1   Lag2   Lag3   Lag4   Lag5    Volume  Today Direction
## 1 1990  0.816  1.572 -3.936 -0.229 -3.484 0.1549760 -0.270      Down
## 2 1990 -0.270  0.816  1.572 -3.936 -0.229 0.1485740 -2.576      Down
## 3 1990 -2.576 -0.270  0.816  1.572 -3.936 0.1598375  3.514        Up
## 4 1990  3.514 -2.576 -0.270  0.816  1.572 0.1616300  0.712        Up
## 5 1990  0.712  3.514 -2.576 -0.270  0.816 0.1537280  1.178        Up
## 6 1990  1.178  0.712  3.514 -2.576 -0.270 0.1544440 -1.372      Down
```

```
## [1] 1089    9
```

```
test.1 <- Weekly[Weekly$Year == 2009,]; head(test.1); dim(test.1)
```

```
##     Year   Lag1   Lag2   Lag3   Lag4   Lag5   Volume  Today Direction
## 986 2009  6.760 -1.698  0.926  0.418 -2.251 3.793110 -4.448      Down
## 987 2009 -4.448  6.760 -1.698  0.926  0.418 5.043904 -4.518      Down
## 988 2009 -4.518 -4.448  6.760 -1.698  0.926 5.948758 -2.137      Down
## 989 2009 -2.137 -4.518 -4.448  6.760 -1.698 6.129763 -0.730      Down
## 990 2009 -0.730 -2.137 -4.518 -4.448  6.760 5.602004  5.173        Up
## 991 2009  5.173 -0.730 -2.137 -4.518 -4.448 6.217632 -4.808      Down
```

```
## [1] 52  9
```

```
test.2 <- Weekly[Weekly$Year == 2009,]; head(test.2); dim(test.2)
```

```
##     Year   Lag1   Lag2   Lag3   Lag4   Lag5   Volume  Today Direction
## 986 2009  6.760 -1.698  0.926  0.418 -2.251 3.793110 -4.448      Down
## 987 2009 -4.448  6.760 -1.698  0.926  0.418 5.043904 -4.518      Down
## 988 2009 -4.518 -4.448  6.760 -1.698  0.926 5.948758 -2.137      Down
## 989 2009 -2.137 -4.518 -4.448  6.760 -1.698 6.129763 -0.730      Down
## 990 2009 -0.730 -2.137 -4.518 -4.448  6.760 5.602004  5.173        Up
## 991 2009  5.173 -0.730 -2.137 -4.518 -4.448 6.217632 -4.808      Down
```

```
## [1] 52  9
```

```
test <- rbind(test.1,test.2); head(test); dim(test)
```

```
##     Year   Lag1   Lag2   Lag3   Lag4   Lag5   Volume  Today Direction
## 986 2009  6.760 -1.698  0.926  0.418 -2.251 3.793110 -4.448      Down
## 987 2009 -4.448  6.760 -1.698  0.926  0.418 5.043904 -4.518      Down
## 988 2009 -4.518 -4.448  6.760 -1.698  0.926 5.948758 -2.137      Down
## 989 2009 -2.137 -4.518 -4.448  6.760 -1.698 6.129763 -0.730      Down
## 990 2009 -0.730 -2.137 -4.518 -4.448  6.760 5.602004  5.173        Up
## 991 2009  5.173 -0.730 -2.137 -4.518 -4.448 6.217632 -4.808      Down
```

```
## [1] 104   9
```

```
# Set up:
n.test <- nrow(test)
predictions <- rep(NA, n.test)

# Train:
for (i in 1:n.test){
  predictions[i] <- KNN.decision(
  Lag1.new = test$Lag1[i],
  Lag2.new = test$Lag2[i],
  K = 5,
  Lag1 = Weekly$Lag1,
  Lag2 = Weekly$Lag2
  )
}
head(predictions)
```

```
## [1] "Up"   "Up"   "Down" "Down" "Up"   "Down"
```

```
test.error <- sum(predictions != test$Direction)/n.test
test.error
```

```
## [1] 0.2115385
```

```
# Ans:
# The error rate is 0.2115, which means we perform
# roughly 80% testing accuracy at K=5.
```

4. Do the same thing as in question 3, but instead use $K = 3$. Which has a lower test error?

```
# Consider K=3:
# Set up:
n.test <- nrow(test)
predictions <- rep(NA, n.test)

# Train:
for (i in 1:n.test){
  predictions[i] <- KNN.decision(
  Lag1.new = test$Lag1[i],
  Lag2.new = test$Lag2[i],
  K = 3,
  Lag1 = Weekly$Lag1,
  Lag2 = Weekly$Lag2
  )
}
head(predictions)
```

```
## [1] "Down" "Up"   "Down" "Down" "Up"   "Down"
```

```
test.error <- sum(predictions != test$Direction)/n.test
test.error
```

```
## [1] 0.2884615
```

```
# Ans:
# The error rate increased from 0.21 to 0.2885
# which means K=3 is performing worse than K=5.
```

6.3.4 Part 3: Cross-validation

Ideally we'd like to use our model to predict future returns, but how do we know which value of K to choose? We could choose the best value of K by training with data from 1990 - 2008, testing with the 2009 - 2010 data, and selecting the model with the lowest test error as in the previous section. However, in order to build the best model, we'd like to use ALL the data we have to train the model. In this case, we could use all of the *Weekly* data and choose the best model by comparing the training error, but unfortunately this isn't usually a good predictor of the test error.

In this section, we instead consider a class of methods that estimate the test error rate by holding out a (random) subset of the data to use as a test set, which is called k-fold cross validation. (Note this lower case k is different than the upper case K in KNN. They have nothing to do with each other, it just happens that the standard is to use the same letter in both.) This approach involves randomly dividing the set of observations into k groups, or folds, of equal size. The first fold is treated as a test set, and the model is fit on the remaining $k - 1$ folds. The error rate, ERR1, is then computed on the observations in the held-out fold. This procedure is repeated k times; each time, a different group of observations is treated as a test set. This process results in k estimates of the test error: ERR1, ERR2, ..., ERRk. The k-fold CV estimate of the test error is computed by averaging these values,

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} ERR_k.$$

We'll run a 9-fold cross-validation in the following. Note that we have 1089 rows in the dataset, so each fold will have exactly 121 members.

```
# Check number of rows:
nrow(Weekly)
```

```
## [1] 1089
```

```
nrow(Weekly)/9
```

```
## [1] 121
```

5. Create a vector *fold* which has n elements, where n is the number of rows in *Weekly*. We'd like for the *fold* vector to take values in 1-9 which assign each corresponding row of the *Weekly* dataset to a fold. Do this in two steps: (1) create a vector using *rep()* with the values 1-9 each repeated 121 times (note $1089 = 121 \cdot 9$), and (2) use *sample()* to randomly reorder the vector you created in (1).

```
# Create:
n <- nrow(Weekly)
vector <- rep(1:9,121)
fold <- sample(vector, length(vector), replace = TRUE)
```

6. Iterate over the 9 folds, treating a different fold as the test set and all others the training set in each iteration. Using a KNN classifier with $K = 5$ calculate the test error for each fold. Then calculate the cross-validation approximation to the test error which is the average of ERR1, ERR2, ..., ERR9.

```
# Data:
data <- data.frame(
  fold,
  Weekly
); head(data); dim(data)
```

```
##   fold Year   Lag1   Lag2   Lag3   Lag4   Lag5    Volume  Today Direction
## 1    7 1990  0.816  1.572 -3.936 -0.229 -3.484 0.1549760 -0.270      Down
## 2    1 1990 -0.270  0.816  1.572 -3.936 -0.229 0.1485740 -2.576      Down
## 3    4 1990 -2.576 -0.270  0.816  1.572 -3.936 0.1598375  3.514        Up
## 4    5 1990  3.514 -2.576 -0.270  0.816  1.572 0.1616300  0.712        Up
## 5    8 1990  0.712  3.514 -2.576 -0.270  0.816 0.1537280  1.178        Up
## 6    1 1990  1.178  0.712  3.514 -2.576 -0.270 0.1544440 -1.372      Down
```

```
## [1] 1089   10
```

```
# Define an NA-entry error matrix
# which we will fill later:
error.matrix <- matrix(NA,nrow=1,ncol=9)

# This is a function we write to change
# K for KNN so that latter problems
# we can simply the answer:

# Function starts here:
CV.average.error <- function(K){
for (m in 1:9) {
  # Split data to training and testing set
  train <- data[data$fold == m,]; dim(train)
  test <- data[data$fold != m,]; dim(test)

  # Function KNN.decisionn:
  # Here we update the function with arbitrary dataset:
```

```r
KNN.decision <- function(
  Lag1.new,
  Lag2.new,
  K,
  Lag1 = train$Lag1,
  Lag2 = train$Lag2) {

  n <- length(Lag1)
  stopifnot(length(Lag2) == n, length(Lag1.new) == 1, length(Lag2.new) == 1, K <= n)
  dists       <- sqrt((Lag1-Lag1.new)^2 + (Lag2-Lag2.new)^2)
  neighbors  <- order(dists)[1:K]
  neighb.dir <- Weekly$Direction[neighbors]
  choice      <- names(which.max(table(neighb.dir)))
  return(choice)
}

# Consider K=5:
# Set up:
n.test <- nrow(test)
predictions <- rep(NA, n.test)

# Train:
for (i in 1:n.test){
  predictions[i] <- KNN.decision(
  test$Lag1[i],
  test$Lag2[i],
  K = K,
  Lag1 = train$Lag1,
  Lag2 = train$Lag2
  )
}
head(predictions)

# Test:
test.error <- sum(predictions != test$Direction)/n.test
test.error
error.matrix[,m] <- test.error

# Output: average error rate
# error.matrix; rowMeans(error.matrix)
}
return(list(
  Error = error.matrix,
  Mean = rowMeans(error.matrix)
))
}

# Use function CV.average.error()
K <- 5
CV.average.error(K)
```

```
## $Error
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4839045 0.4686848 0.4699793 0.4947808 0.4628866 0.4876543 0.5010288
```

```
##          [,8]      [,9]
## [1,] 0.478125 0.5035247
##
## $Mean
## [1] 0.4833965
```

```
# Ans:
# We get average testing error from
# Cross-Validation to be 0.4566919,
# which is not that great since
# flipping a coin already gives us 0.5
```

7. Repeat step (6) for $K = 1$, $K = 3$, and $K = 7$. For which set is the cross-validation approximation to the test error the lowest?

```
# Data:
data <- data.frame(
  fold,
  Weekly
); head(data); dim(data)
```

```
##   fold Year   Lag1   Lag2   Lag3   Lag4   Lag5    Volume  Today Direction
## 1    7 1990  0.816  1.572 -3.936 -0.229 -3.484 0.1549760 -0.270      Down
## 2    1 1990 -0.270  0.816  1.572 -3.936 -0.229 0.1485740 -2.576      Down
## 3    4 1990 -2.576 -0.270  0.816  1.572 -3.936 0.1598375  3.514        Up
## 4    5 1990  3.514 -2.576 -0.270  0.816  1.572 0.1616300  0.712        Up
## 5    8 1990  0.712  3.514 -2.576 -0.270  0.816 0.1537280  1.178        Up
## 6    1 1990  1.178  0.712  3.514 -2.576 -0.270 0.1544440 -1.372      Down
```

```
## [1] 1089   10
```

```
error.matrix <- matrix(NA,nrow=1,ncol=9)

# Use function CV.average.error()
# K <- 5
CV.average.error(5)
```

```
## $Error
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4839045 0.4686848 0.4699793 0.4947808 0.4628866 0.4876543 0.5010288
##          [,8]      [,9]
## [1,] 0.478125 0.5035247
##
## $Mean
## [1] 0.4833965
```

```
# K = 1:
CV.average.error(1)
```

```
## $Error
##           [,1]      [,2]      [,3]     [,4]      [,5]      [,6]      [,7]
## [1,] 0.4766355 0.5114823 0.5248447 0.493737 0.4835052 0.4897119 0.5102881
##           [,8]      [,9]
## [1,] 0.5104167 0.4974824
##
## $Mean
## [1] 0.4997893
```

```r
# K = 3:
CV.average.error(3)
```

```
## $Error
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4672897 0.4979123 0.4803313 0.5010438 0.4690722 0.4742798 0.5010288
##           [,8]      [,9]
## [1,] 0.4989583 0.4843907
##
## $Mean
## [1] 0.4860341
```

```r
# K = 5:
CV.average.error(5)
```

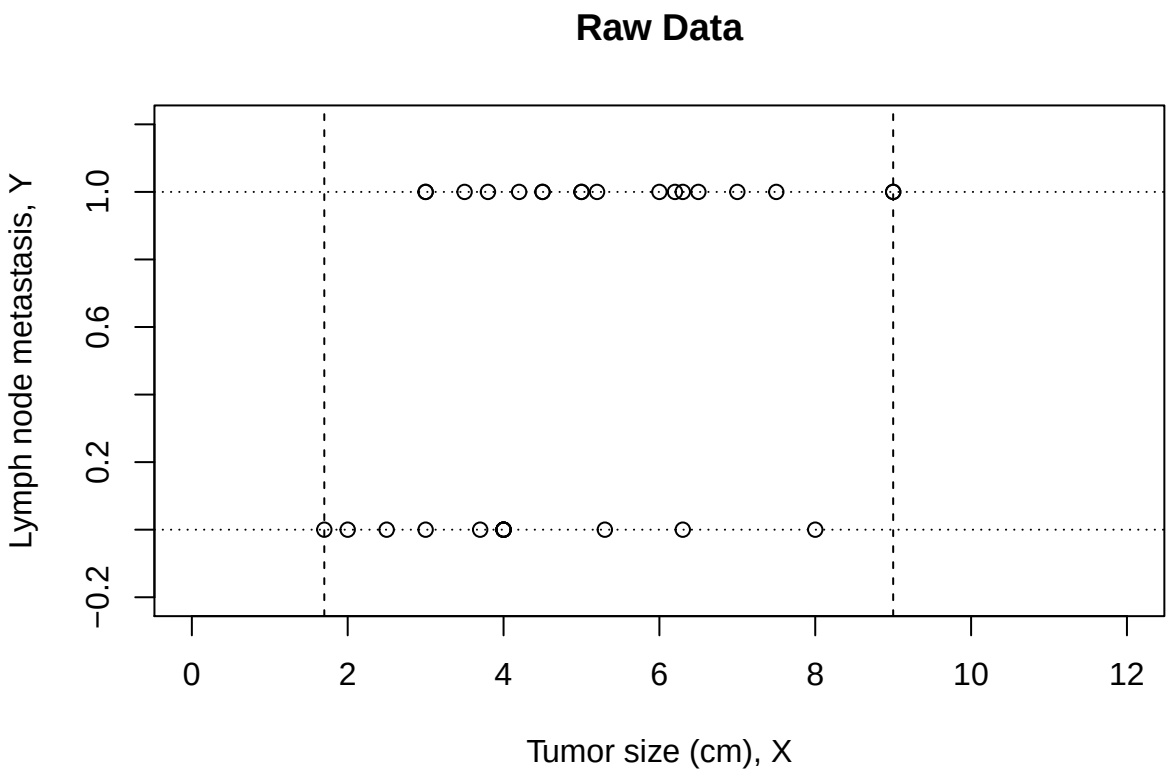
```
## $Error
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4839045 0.4686848 0.4699793 0.4947808 0.4628866 0.4876543 0.5010288
##          [,8]      [,9]
## [1,] 0.478125 0.5035247
##
## $Mean
## [1] 0.4833965
```

```r
# K = 7
CV.average.error(7)
```

```
## $Error
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 0.4932503 0.4530271 0.4575569 0.4885177 0.4618557 0.4804527 0.5082305
##           [,8]      [,9]
## [1,] 0.4770833 0.4874119
##
## $Mean
## [1] 0.4785985
```

7 LOGISTIC AND NEWTON'S METHOD

```r
#########################
# Load in data set:
#########################

#setwd("F://Course/CU Stats/STATS W4206(S) - Stat Computing Intro to Data Sci/3. R")
data <- read.table("Example71.txt")

#########################
# Plot of raw data
#########################
plot(data$x,data$y,xlab="Tumor size (cm), X", ylab="Lymph node metastasis, Y",main="Raw Data",ylim=c(-.1
abline(h=0,lty=3)
abline(h=1,lty=3)
abline(v=min(data$x),lty=2,col=1)
abline(v=max(data$x),lty=2,col=1)
```




```
#########################
# Estimate logistic model using glm()
#########################

model <- glm(y~x,data=data,family=binomial(link = "logit"))
linear.pred <- predict(model,newdata=data.frame(x=7))
probs <- exp(linear.pred)/(1+exp(linear.pred))
probs
```

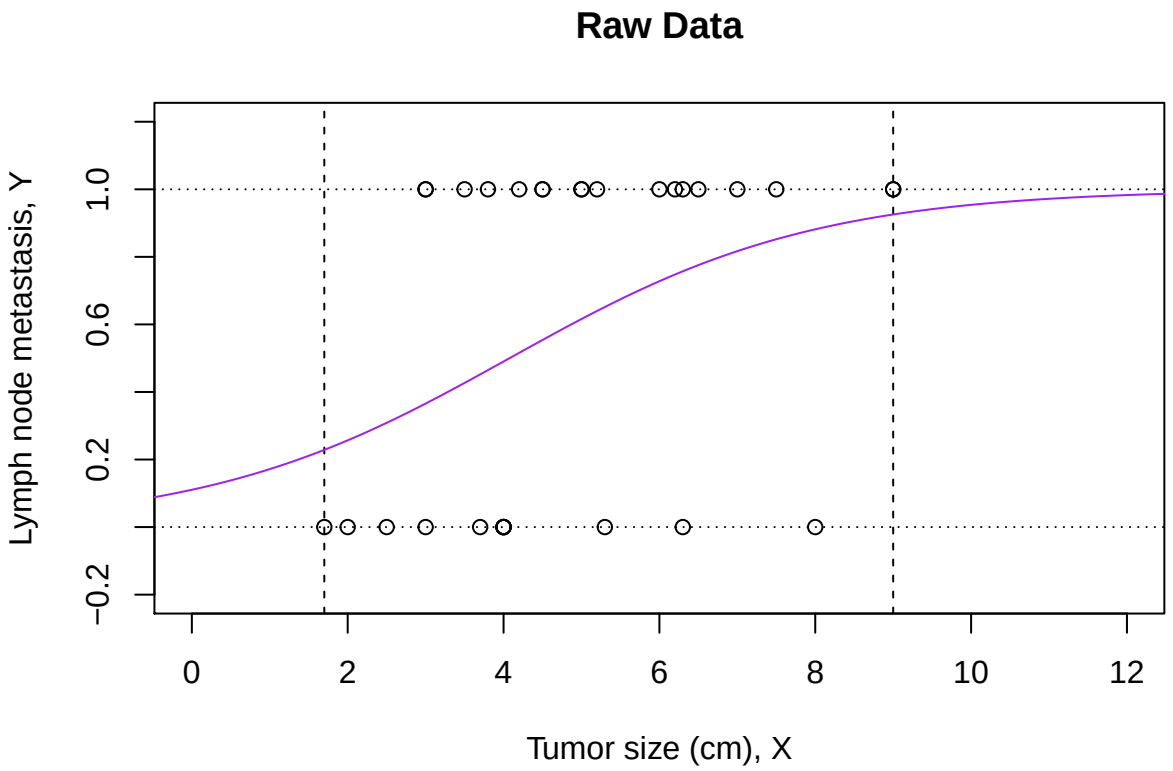
```
##         1
## 0.8169462
```

```
log(probs/(1-probs))
```

```
##        1
## 1.495793
```

```
#########################
# Plot of raw data and estimated model
#########################

plot(data$x,data$y,xlab="Tumor size (cm), X", ylab="Lymph node metastasis, Y",main="Raw Data",ylim=c(-.
abline(h=0,lty=3)
abline(h=1,lty=3)
abline(v=min(data$x),lty=2,col=1)
abline(v=max(data$x),lty=2,col=1)
x.pred <- seq(-1,13,by=.1)
linear.pred <- predict(model,newdata=data.frame(x=x.pred))
```

```r
probs <- exp(linear.pred)/(1+exp(linear.pred))
lines(x.pred,probs,col="purple")
```




```r
##########################
# Maximum likelihood estimation
##########################

##########################
# Log-likelihood function (negative)
##########################

logistic.Neg.LL <- function(b,data=data) {

  b0 <- b[1]
  b1 <- b[2]
  x <- data$x
  y <- data$y
  p.i <- exp(b0+b1*x)/(1+exp(b0+b1*x))
  return(-sum(dbinom(y,size=1,prob=p.i,log=TRUE)))

}

logistic.Neg.LL(b=c(0,0),data=data)
```

```
## [1] 21.48756
```

```r
logistic.Neg.LL(b=c(-3,.6),data=data)
```

```
## [1] 19.37619
```

```r
#########################
# grad.descent from class
#########################

library(numDeriv)
```

```
##
## Attaching package: 'numDeriv'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     grad
```

```r
grad.descent <- function(f, x0, max.iter = 200, step.size = 0.05, stopping.deriv = 0.01, ...) {

  n    <- length(x0)
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0

  for (k in 2:max.iter) {
    # Calculate the gradient
    grad.cur <- grad(f, xmat[ ,k-1], ...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k <- k-1; break
    }

    # Move in the opposite direction of the grad
    xmat[ ,k] <- xmat[ ,k-1] - step.size * grad.cur
  }

  xmat <- xmat[ ,1:k] # Trim
  return(list(x = xmat[,k],
              xmat = xmat,
              k = k,
              minimum=f(xmat[,k],...)
  )
  )
}

#########################
# Optimization using grad.descent()
#########################

#x0 <- c(0,0)
#gd <- grad.descent(logistic.Neg.LL,x0,max.iter = 2000, step.size = 0.01, stopping.deriv = 0.001,data=d
#gd$minimum
#gd$k
#gd$x
#coef(model)
```

```
#########################
# Optimization using nlm()
#########################

#
#nml.opt <- nlm(logistic.Neg.LL,c(0,0),data=data)
#nml.opt


########################
Newton.Method <- function(f, x0, max.iter = 200, stopping.deriv = 0.01, ...) {

  n    <- length(x0)
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0

  for (k in 2:max.iter) {
    # Calculate the gradient
    grad.cur <- grad(f, xmat[ ,k-1], ...)

    # Calculate the hessian
    hess.cur <- hessian(f, xmat[, k-1], ...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k <- k-1; break
    }

    # Move in the opposite direction of the grad
    xmat[ ,k] <- xmat[ ,k-1] - solve(hess.cur) %*% grad.cur
  }

  xmat <- xmat[ ,1:k] # Trim
  return(list(x = xmat[,k],
              xmat = xmat,
              k = k,
              minimum=f(xmat[,k],...)
              )
         )
}

#x0 <- c(0,0)
#gd <- Newton.Method(logistic.Neg.LL,x0,max.iter = 2000, stopping.deriv = 0.001,data=data)
#gd$minimum
#gd$k
#gd$x
#coef(model)
```

8 Homework 1

In this problem we study *Titanic* dataset. The dataset provides information on each passenger including, for example, economic status, sex, age, cabin, name, and survival status. This is a training dataset taken from the Kaggle competition website; for more information on Kaggle competitions, please refer to https://kaggle.com. Students should download the data set on Canvas. Below is a more detailed description of the variables. We approach this problem by the following part:

8.1 Part 1: Importing Data to R

8.1.1 i. Import

We important data to R and call this matrix titanic. Use function read.table(). Use the argument as.is = TRUE. The dataset should be stored in a data frame called titanic.

```
getwd() # Check to make sure this is correct location that the dataset is stored
```

```
## [1] "F:/Course/CU Stats/STATS W4206(S) - Stat Computing Intro to Data Sci/7. Notes"
```

```
# ?read.table
titanic <- read.table("Titanic.txt",
                      as.is = TRUE,
                      header = TRUE); head(titanic, 2); str(titanic)
```

```
##   PassengerId Survived Pclass
## 1           1        0      3
## 2           2        1      1
##                                                 Name    Sex Age SibSp
## 1                            Braund, Mr. Owen Harris   male  22     1
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38     1
##   Parch    Ticket    Fare Cabin Embarked
## 1     0 A/5 21171  7.2500              S
## 2     0  PC 17599 71.2833   C85        C
```

```
## 'data.frame':    891 obs. of  12 variables:
##  $ PassengerId: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Survived   : int  0 1 1 1 0 0 0 0 1 1 ...
##  $ Pclass     : int  3 1 3 1 3 3 1 3 3 2 ...
##  $ Name       : chr  "Braund, Mr. Owen Harris" "Cumings, Mrs. John Bradley (Florence Briggs Thayer)"
##  $ Sex        : chr  "male" "female" "female" "female" ...
##  $ Age        : num  22 38 26 35 35 NA 54 2 27 14 ...
##  $ SibSp      : int  1 1 0 1 0 0 0 3 0 1 ...
##  $ Parch      : int  0 0 0 0 0 0 0 1 2 0 ...
##  $ Ticket     : chr  "A/5 21171" "PC 17599" "STON/O2. 3101282" "113803" ...
##  $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
##  $ Cabin      : chr  "" "C85" "" "C123" ...
##  $ Embarked   : chr  "S" "C" "S" "S" ...
```

8.1.2 ii. Dimension

How many rows and columns does titanic have?

```
dim(titanic)
```

```
## [1] 891  12
```

```
# Observe the output is a dim = 891 x 12, which matches
# the answer notated in the instrubtion.
```

8.1.3 iii. Create: Survived.Word

Create a new variable in the data frame called Survived.Word. It should read either "survived" or "died" indicating whether the passenger survived or died. This variable should be of type 'character'.

```
head(titanic, 2)
```

```
##   PassengerId Survived Pclass
## 1           1        0      3
## 2           2        1      1
##                                                 Name    Sex Age SibSp
## 1                             Braund, Mr. Owen Harris   male  22     1
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38     1
##   Parch    Ticket    Fare Cabin Embarked
## 1     0 A/5 21171  7.2500              S
## 2     0  PC 17599 71.2833   C85        C
```

```
# Create a new variable:
# read either "survived" or "died".

# The interpretation is to create a control-statement
# such that the Survived: 0 == Died and 1 == Survived.
summary(titanic$Survived) # All entries are between 0 and 1
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.0000  0.0000  0.3838  1.0000  1.0000
```

```
Survived.Word <- matrix()
for (i in 1:nrow(titanic)){
  unit <- ifelse(titanic$Survived[i] == 0, "Died", "Survived")
  Survived.Word <- rbind(Survived.Word, unit)
}; mode(Survived.Word); length(Survived.Word)
```

```
## [1] "character"
```

```
## [1] 892
```

```
head(titanic$Survived); head(Survived.Word)
```

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

```
##      [,1]
##      NA
## unit "Died"
## unit "Survived"
## unit "Survived"
## unit "Survived"
## unit "Died"
```

```
# Comment: this is a variable, namely Survived.Word,
# that it is created by a for a loop. The loop loops through
# each row in the variable Survived of the data frame titanic
# and takes Survived == 0 as "Died" and "Survifed" elsewhere.
# mode() gives us "character". "
```

8.2 Part 2: Exploring Data in R

8.2.1 i. Calculate Mean

Use the apply() function to calculate the mean of the variables Survived, Age, and Fare. This will require using the apply() function on a sub-matrix of dimension 891.3. Explain what the mean of Survived tells us. One of the mean values is NA. Which variable has a mean value of NA and why is this the case?

```
# ?apply() # To understand what the function does
head(titanic, 3) # Quick view of 3 rows
```

```
##   PassengerId Survived Pclass
## 1           1        0      3
## 2           2        1      1
## 3           3        1      3
##                                                 Name    Sex Age SibSp
## 1                             Braund, Mr. Owen Harris   male  22     1
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38     1
## 3                              Heikkinen, Miss. Laina female  26     0
##   Parch           Ticket    Fare Cabin Embarked
## 1     0        A/5 21171  7.2500              S
## 2     0         PC 17599 71.2833   C85        C
## 3     0 STON/O2. 3101282  7.9250              S
```

```
mean_matrix <- apply(cbind(
  titanic$Survived,
  titanic$Age,
  titanic$Fare
), 2, mean); mean_matrix
```

```
## [1]  0.3838384         NA 32.2042080
```

```
# Comment: observe that the second entry is NA;
# and this entry is the mean for Age. This is
# because one of the entries of the column of Age
# in the data frame is NA or at least one of the entries are
# NA term.
```

8.2.2 ii. Compute Proportion

Compute the proportion of female passengers who survived the titanic disaster. Round your answer to 2 decimals using the round() function. Hint ?round.

```
head(titanic[,c(2,5)],5) # Only show Survived and Sex
```

```
##   Survived    Sex
## 1        0   male
## 2        1 female
## 3        1 female
## 4        1 female
## 5        0   male
```

```
female.survived.total <- tapply(titanic[,2], titanic[,5], sum)[[1]]
count.female.total <- plyr::count(titanic$Sex)$freq[[1]]
count.female.total # How many women are there?
```

```
## [1] 314
```

```
female.survived.total # How many women survived?
```

```
## [1] 233
```

```
female.survived.proportion <- female.survived.total/count.female.total

# Round to 2 decimals by using round()
# ?round()
round(female.survived.proportion, 2)
```

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

```
# Comment: 74.20% of the female passengers survived the
# titanic disasters.
```

8.2.3 iii. Compute Proportion

Of the survivors, compute the proportion of female passengers. Round your answer to 2 decimals. This answer may take a few lines of code. One strategy would be to create a survivors matrix that only includes individuals who survived the disaster. Then using the survived matrix, calculate the proportion of females.

```
head(titanic[,c(2,5)],5) # Only show Survived and Sex
```

```
##   Survived    Sex
## 1        0   male
## 2        1 female
## 3        1 female
## 4        1 female
## 5        0   male
```

```
female.survived.total <- tapply(titanic[,2], titanic[,5], sum)[[1]]
count.survived.total <- sum(tapply(titanic[,2], titanic[,5], sum))

female.survived.total # How many women survived?
```

```
## [1] 233
```

```
count.survived.total # How many survivors are there?
```

```
## [1] 342
```

```
survived.female.proportion <- female.survived.total/count.survived.total
survived.female.proportion; round(survived.female.proportion, 2)
```

```
## [1] 0.6812865
```

```
## [1] 0.68
```

```
# Comment: 68.13% of the survivors are female.
```

8.2.4 iv. Fill In Blanks

Use the following code to create an empty numeric vector of length three called Pclass.Survival. We will fill in the elements of Pclass.Survival with the survival rates of the three classes.

```
head(cbind(titanic$Survived, titanic$Pclass)) # Quick view
```

```
##      [,1] [,2]
## [1,]    0    3
## [2,]    1    1
## [3,]    1    3
## [4,]    1    1
## [5,]    0    3
## [6,]    0    3
```

```
df <- data.frame(cbind(titanic$Survived,titanic$Pclass))
names(df) <- c("Survived", "Pclass"); head(df)
```

```
##   Survived Pclass
## 1        0      3
## 2        1      1
## 3        1      3
## 4        1      1
## 5        0      3
## 6        0      3
```

```
# Create an empty numeric vector of length three:
classes <- sort(unique(titanic$Pclass))
Pclass.Survival <- vector("numeric", length=3)

# Make sure it is a matrix
Pclass.Survival <- data.frame(t(Pclass.Survival)); dim(Pclass.Survival)
```

```
## [1] 1 3
```

```
names(Pclass.Survival) <- classes

# Now we want to fill in the blanks
# ?plyr::count() # Count unique variables:
count <- 0
for (i in 1:3){
  # i <- 1
  for (i.row in 1:nrow(df)){
    unit <- ifelse(df$Pclass[i.row] == i,
                   ifelse(df$Survived[i.row] == 1,
                          1,
                          0),
                   0)
    count <- count + unit
  }
  percent <- count/plyr::count(df, vars = "Pclass")$freq[[i]]
  Pclass.Survival[1,i] <- round(percent,2)
  count <- 0
}; Pclass.Survival
```

```
##      1    2    3
## 1 0.63 0.47 0.24
```

8.2.5 v. Create Variable

```
head(cbind(titanic$Survived, titanic$Pclass)) # Quick view
```

```
##      [,1] [,2]
```

```
## [1,]    0    3
## [2,]    1    1
## [3,]    1    3
## [4,]    1    1
## [5,]    0    3
## [6,]    0    3
```

```
df <- data.frame(cbind(titanic$Survived,titanic$Pclass))
names(df) <- c("Survived", "Pclass"); head(df)
```

```
##   Survived Pclass
## 1        0      3
## 2        1      1
## 3        1      3
## 4        1      1
## 5        0      3
## 6        0      3
```

```
# Create an empty numeric vector of length three:
classes <- sort(unique(titanic$Pclass))
Pclass.Survival.2 <- vector("numeric", length=3)

# Make sure it is a matrix
Pclass.Survival.2 <- data.frame(t(Pclass.Survival.2)); dim(Pclass.Survival.2)
```

```
## [1] 1 3
```

```
names(Pclass.Survival.2) <- classes

# Now we want to fill in the blanks
# ?plyr::count() # Count unique variables:
for (i in 1:3){
  # i <- 1
  percent <- tapply(
    df$Survived,
    df$Pclass, sum)[[i]]/plyr::count(df,
                                     vars = "Pclass")$freq[[i]]
  Pclass.Survival.2[1,i] <- round(percent,2)
}; Pclass.Survival.2
```

```
##      1    2    3
## 1 0.63 0.47 0.24
```

```
# Comment: By using tapply(), the lines of coding are drastically
# shortened.
```

8.2.6 vi. Comment: Relation

Is there a relation with survival rate and class?

```
# Observe the following:
Class <- c(1,2,3)
Survival.Rate <- c(0.63, 0.47, 0.24)
linear.model <- lm(Survival.Rate ~ Class)
summary(linear.model)
```

```
##
```

```
## Call:
## lm(formula = Survival.Rate ~ Class)
##
## Residuals:
##       1       2       3
## -0.01167  0.02333 -0.01167
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.83667    0.04365   19.17   0.0332 *
## Class       -0.19500    0.02021   -9.65   0.0657 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.02858 on 1 degrees of freedom
## Multiple R-squared:  0.9894, Adjusted R-squared:  0.9788
## F-statistic: 93.12 on 1 and 1 DF,  p-value: 0.06574
```

```
# Comment: Observe the t-stat for Class
# is actually -9.65, which is a lot higher than
# 1.96, a 95% confidence interval cutoff line.
# We can conclude that the null hypothesis,
# which is that this coefficient is zero,
# is rejected. We have sufficient evidence that
# the coefficient for Class is not zero.
```

9 Homework 2

The goal is to perform basic data cleaning, EDA, R graphics type of techniques on a sample dataset. The data NYHousing.csv contains property-level data on privately-owned, subsidized rental properties in NYC collected by Furman Center. The data can be found http://www.furmancenter.org/data/. The dataset contains financial and physical information on the properties including geographic, subsidy, ownership, physical, and financial information.

9.1 Part 1: Loading Data and Cleaning the Data in R

9.1.1 i. Import

Load data:

```
housing <- read.csv("NYChousing.csv", header=T)
head(housing,3) # Quick view of the first three rows of the dataset
```

```
##      UID               PropertyName       Lon      Lat
## 1 100000           1018 DEVELOPMENT -73.89244 40.82060
## 2 100001  1041 BUSHWICK AVENUE APTS -73.92065 40.69130
## 3 100002 1085 MANHATTAN DEVELOPMENT -73.95547 40.73569
##                                    AgencyID                        Name
## 1 BBL 2-02723-0040;HUD Property 800016127           1018 DEVELOPMENT
## 2 BBL 3-03331-0036;HUD Property 800016130  1041 BUSHWICK AVENUE APTS
## 3 BBL 3-02495-0042;HUD Property 800016133 1085 MANHATTAN DEVELOPMENT
##     Value               Address Violations2010 REACNumber  Borough
## 1 2041200  1010 EAST 163 STREET              3         94    Bronx
```

```
## 2  897300  1041 BUSHWICK AVENUE              0         92 Brooklyn
## 3  253490 1085 MANHATTAN AVENUE              0         98 Brooklyn
##                              CD CityCouncilDistrict  CensusTract
## 1    BX02: Hunts Point/Longwood                  17     Bronx 89
## 2               BK04: Bushwick                  34 Brooklyn 399
## 3 BK01: Greenpoint/Williamsburg                 33 Brooklyn 563
##   BuildingCount UnitCount YearBuilt                Owner Rental.Coop
## 1             1        97      1909   1018 DEVELOPMENT CO      Rental
## 2             1        48      1924 1041 BUSHWICK AVE ASS      Rental
## 3             1        12      1931  MANHATTAN AVE ASSOCS      Rental
##   OwnerProfitStatus AffordabilityRestrictions
## 1                                  Affordable
## 2                                  Affordable
## 3                                  Affordable
##   StartAffordabilityRestrictions
## 1                           1979
## 2                           1980
## 3                           1978
```

9.1.2 ii. Dimensions

```
# How many rows and columns are there?
orig_dim <- dim(housing); orig_dim
```

```
## [1] 2506   22
```

```
# Comment: this dataset has dimension 2506x22
# which means that there are 2506 observations and there are
# 22 variables in this dataset
```

9.1.3 iii. apply()

```
apply(is.na(housing), 2, sum)
```

```
##                          UID                 PropertyName
##                            0                            0
##                          Lon                          Lat
##                           15                           15
##                     AgencyID                         Name
##                            0                            0
##                        Value                      Address
##                           52                            0
##               Violations2010                    REACNumber
##                            0                         1873
##                      Borough                           CD
##                            0                            0
##          CityCouncilDistrict                  CensusTract
##                           10                            0
##                BuildingCount                    UnitCount
##                            0                            0
##                    YearBuilt                        Owner
##                            0                            0
##                  Rental.Coop             OwnerProfitStatus
```

```
##                              0                              0
##      AffordabilityRestrictions StartAffordabilityRestrictions
##                              0                              5
```

```
# Comment: this line of code is using the apply() function to
# to screen through the housing dataset and looking for the entries
# that are "NA", not applicable, meaning na entries. Then it gives us
# how many NA entries are there in that column (or in that variable).
# For example, looking at the review screen, we observe for UID variable
# there are no NA entries. For Lon variable, there are 15 NA entries.
```

9.1.4 iv. Remove NA entries

```
housing <- read.csv("NYChousing.csv", header=T)
housing.copy <- housing; dim(housing.copy) # Back up
```

```
## [1] 2506   22
```

```
housing <- housing[!is.na(housing$Value), ]; dim(housing)
```

```
## [1] 2454   22
```

9.1.5 v. Difference?

How many rows did you remove?

```
# Check the following:
dim(housing.copy); dim(housing)
```

```
## [1] 2506   22
```

```
## [1] 2454   22
```

```
# The difference for the rwos:
new_dim <- dim(housing)
orig_dim[1] - new_dim[1]
```

```
## [1] 52
```

9.1.6 vi. Create logValue

```
# Create a new variable called logValue:
housing$logValue <- log(housing$Value)
summary(housing$logValue)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    8.41   12.49   13.75   13.68   14.80   20.47
```

```
# Comment:
# min = 10.06;
# median = 14.65;
# mean = 14.65;
# max = 20.22
```

9.1.7 vii. Create logUnits

```
# Create a new variable called logUnits:
housing$logUnits <- log(housing$UnitCount)
summary(housing$logUnits)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   2.773   3.892   3.775   4.691   9.640
```

```
# Comment:
# min = 1.609;
# median = 4.564;
# mean = 3.560;
# max = 8.679;
```

9.1.8 viii. Create after1950

```
# Create a new variable called after1950:
housing$after1950 <- ifelse(housing$YearBuilt > 1950, "TRUE", "FALSE")
# Quick view:
head(housing$YearBuilt)
```

```
## [1] 1909 1924 1931 1915 1905 1974
```

```
head(housing$after1950)
```

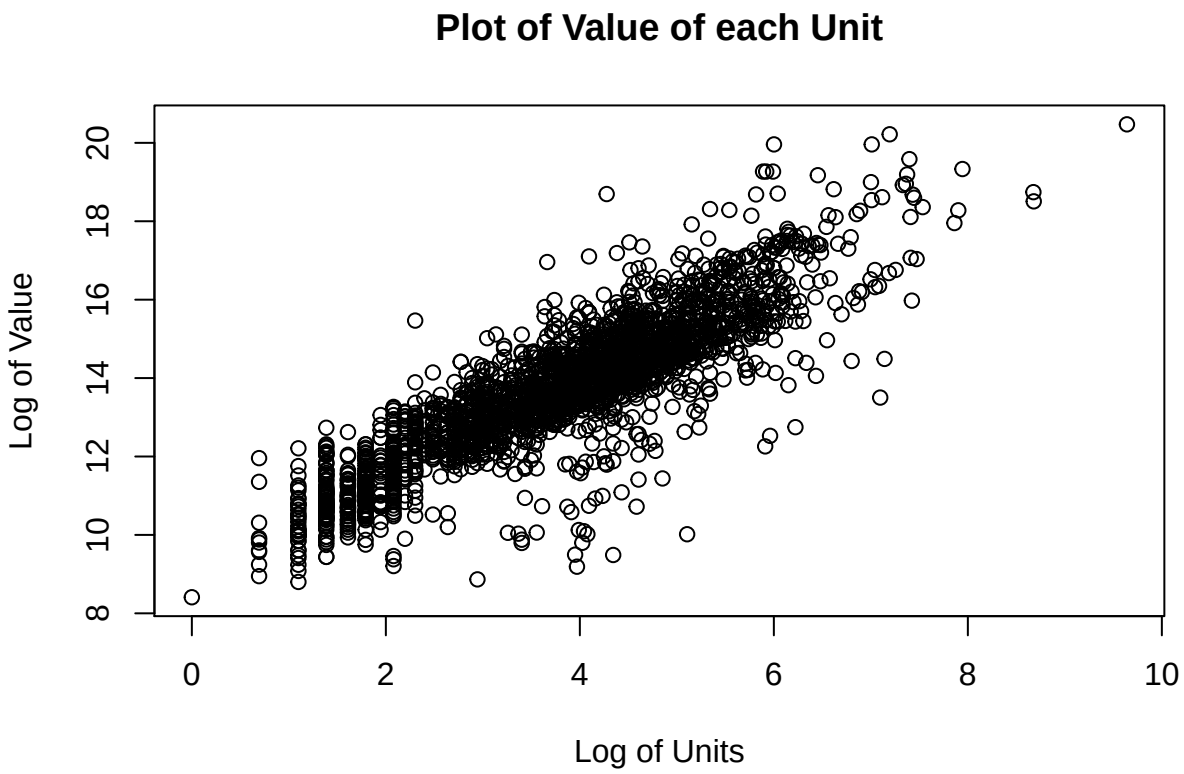
```
## [1] "FALSE" "FALSE" "FALSE" "FALSE" "FALSE" "TRUE"
```

9.2 Part 2:

The column Borough in the data housing contains Borough of each property and is one of either Bronx, Manhattan, Staten Island, Brooklyn, or Queens.

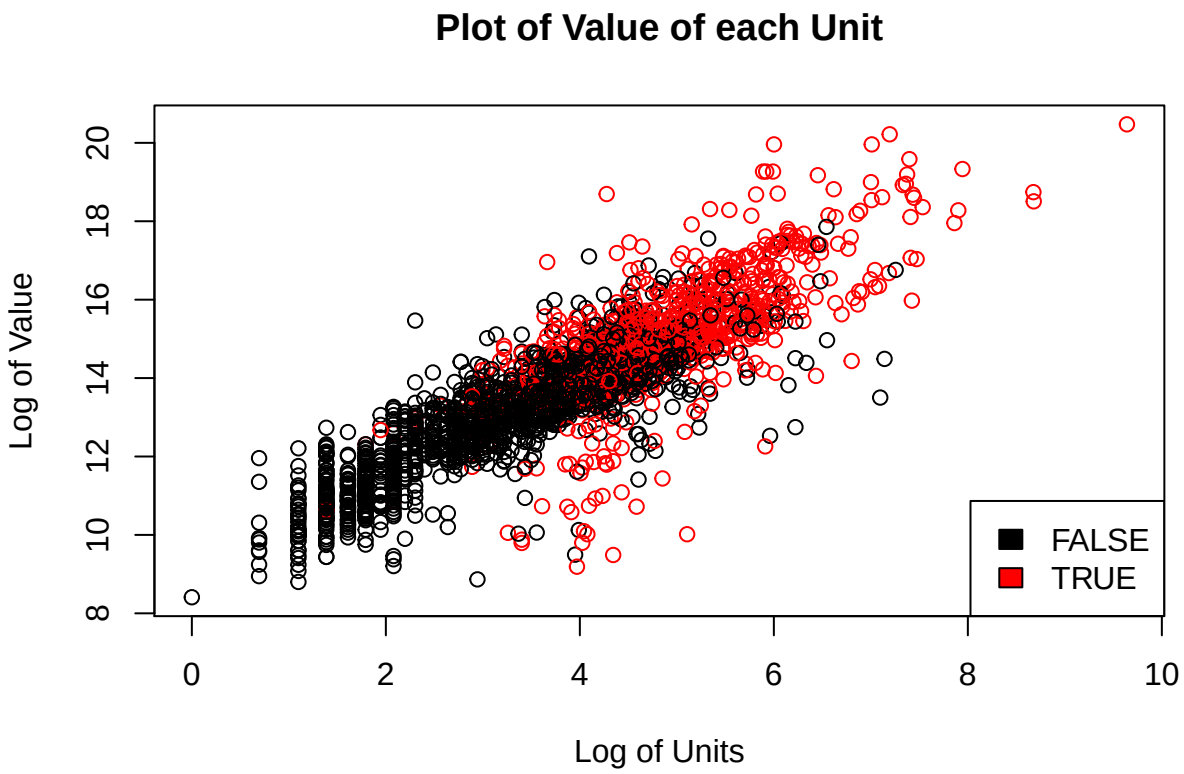
9.2.1 i. Plot

```
plot(housing$logUnits, housing$logValue,
     xlab="Log of Units",
     ylab="Log of Value",
     main="Plot of Value of each Unit")
```



9.2.2 ii. Plot with Columns

```
plot(housing$logUnits, housing$logValue,
     xlab="Log of Units",
     ylab="Log of Value",
     main="Plot of Value of each Unit",
     col=factor(housing$after1950)
)
# Add Legend:
legend(
  "bottomright",
  legend=levels(factor(housing$after1950)),
  fill=1:length(levels(factor(housing$after1950)))
)
```




```
# Observe that the plot generates dots of
# log values of units for two categories, before vs. after 1950,
# with the red dots to be after 1950, and black dots otherwise.
# It makes sense in reality that the units built after 1950 were
# gradually more expensive than those built before 1950. Hence,
# we observe more red dots on upper right corner of the graph.
```

9.2.3 iii. Correlation

```
# Quickview:
freq.borough <- plyr::count(housing$Borough); freq.borough
```

```
##               x freq
## 1         Bronx  664
## 2      Brooklyn  829
## 3     Manhattan  851
## 4        Queens   77
## 5 Staten Island   33
```

```
# Comment:
# plyr::count on housing$Borough gives us frequency each type of Borough occurs in the data

count.by.borough <- plyr::count(
  cbind(housing$Borough, housing$logUnits, housing$logValue))
head(count.by.borough)
```

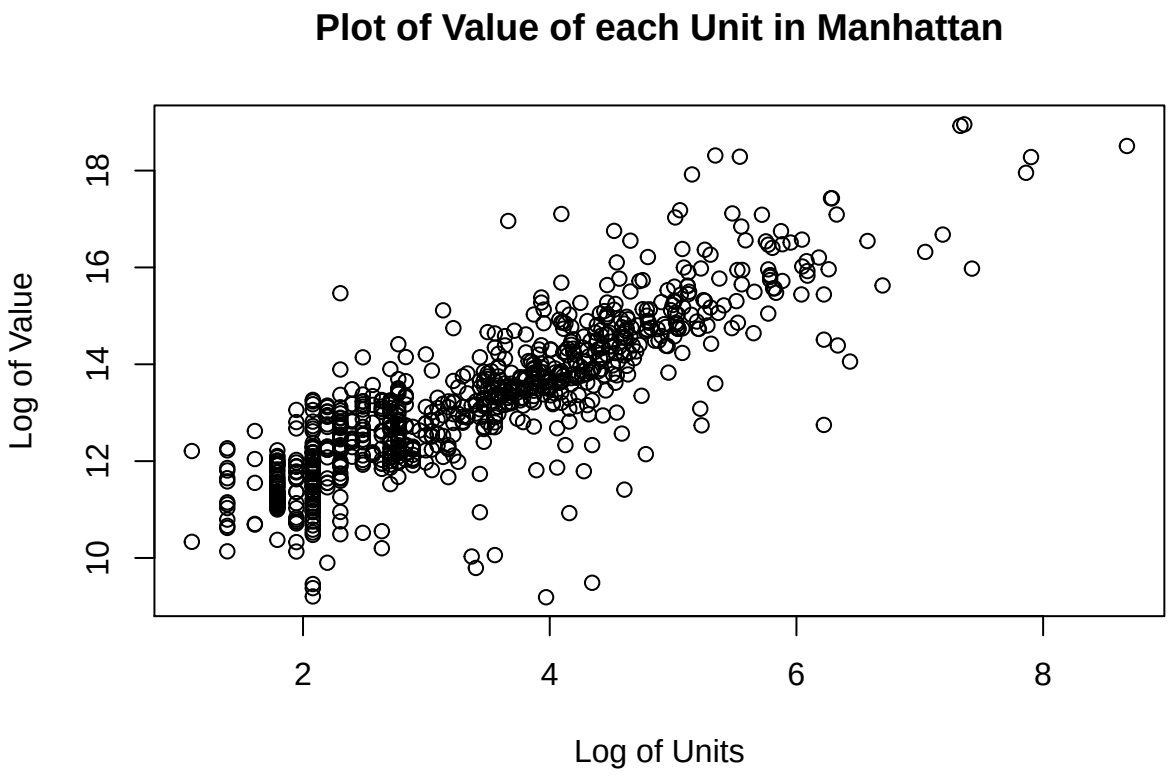
```
##   x.1      x.2      x.3 freq
## 1   1 1.098612  9.48235    1
## 2   1 1.098612 10.09080    1
## 3   1 1.098612 10.11050    1
## 4   1 1.098612 11.75501    1
## 5   1 1.386294 10.46445    2
## 6   1 1.386294 10.72135    1
```

```
# Comment:
# plyrr:count on cbind() of three values gives us frequency of each type categorized by the
# first variable, which is Borough.

# Define new variables:
# correlation in borough of some type is defined by cor() function:
corr.in.Borough.Bronx <- cor(
  count.by.borough$x.2[1:freq.borough$freq[[1]]],
  count.by.borough$x.3[1:freq.borough$freq[[1]]])
corr.in.Borough.Brooklyn <- cor(
  count.by.borough$x.2[freq.borough$freq[[1]]+1:freq.borough$freq[[2]]],
  count.by.borough$x.3[freq.borough$freq[[1]]+1:freq.borough$freq[[2]]])
corr.in.Borough.Manhattan <- cor(
  count.by.borough$x.2[freq.borough$freq[[2]]+1:freq.borough$freq[[3]]],
  count.by.borough$x.3[freq.borough$freq[[2]]+1:freq.borough$freq[[3]]])
corr.in.Borough.Queens <- cor(
  count.by.borough$x.2[freq.borough$freq[[3]]+1:freq.borough$freq[[4]]],
  count.by.borough$x.3[freq.borough$freq[[3]]+1:freq.borough$freq[[4]]])
corr.in.Borough.StatenIsland <- cor(
  count.by.borough$x.2[freq.borough$freq[[4]]+1:freq.borough$freq[[5]]],
  count.by.borough$x.3[freq.borough$freq[[4]]+1:freq.borough$freq[[5]]])
c(
  corr.in.Borough.Bronx,
  corr.in.Borough.Brooklyn,
  corr.in.Borough.Manhattan,
  corr.in.Borough.Queens,
  corr.in.Borough.StatenIsland
)
```

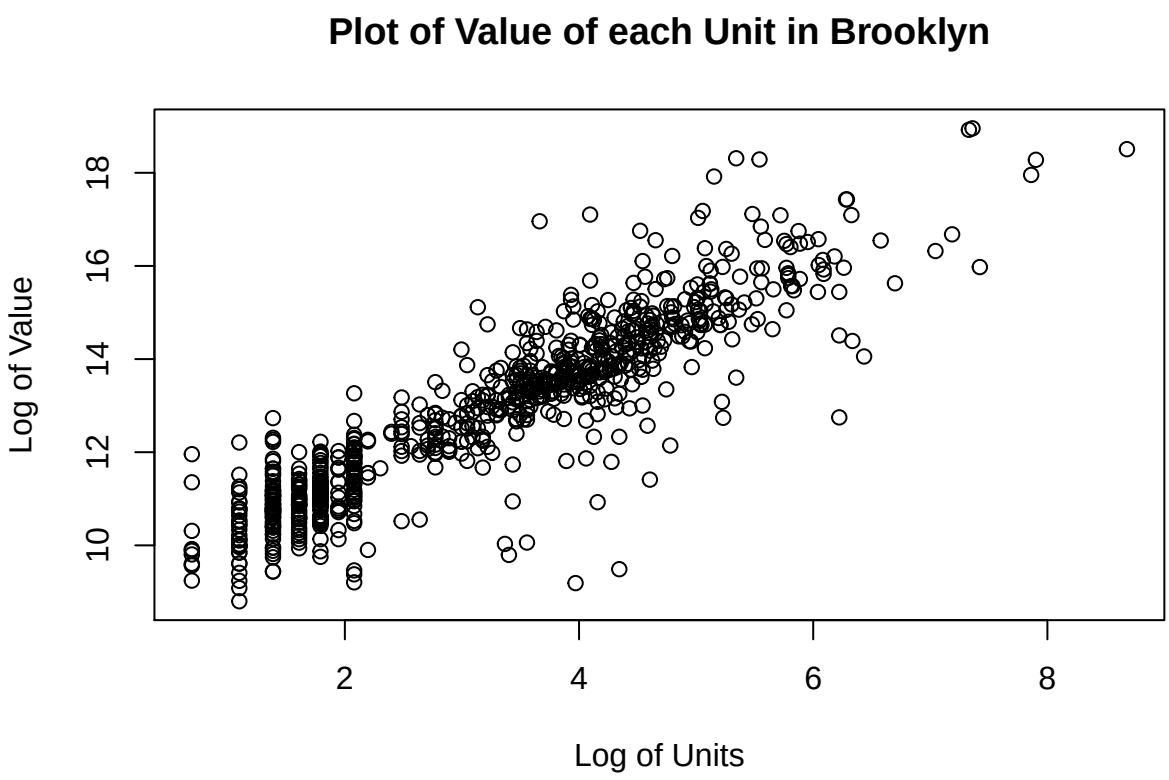
```
## [1]  0.7787119  0.9067442  0.8501742 -0.3325070  0.2761048
```

9.2.4 iv. Plot and Compare

```
plot(
  cbind(
    count.by.borough$x.2[freq.borough$freq[[2]]+1:freq.borough$freq[[3]]],
    count.by.borough$x.3[freq.borough$freq[[2]]+1:freq.borough$freq[[3]]]
    ),
  xlab="Log of Units",
  ylab="Log of Value",
  main="Plot of Value of each Unit in Manhattan",
  type="p"
)
```




```
plot(
  cbind(
    count.by.borough$x.2[freq.borough$freq[[1]]+1:freq.borough$freq[[2]]],
    count.by.borough$x.3[freq.borough$freq[[1]]+1:freq.borough$freq[[2]]]
    ),
  xlab="Log of Units",
  ylab="Log of Value",
  main="Plot of Value of each Unit in Brooklyn",
  type="p"
)
```



9.2.5 v. Simplify Codes

```
manhat.props <- c()
for (props in 1:nrow(housing)) {
  if (housing$Borough[props] == "Manhattan") {
    manhat.props <- c(manhat.props, props)
    }
}

med.value <- c()
for (props in manhat.props) {
  med.value <- c(med.value, housing$Value[props])
}

med.value <- median(med.value, na.rm = TRUE)
# Answer is 3129300

# sort(med.value) # This gives us all the med.vaulue entries in order

#plyr::count(housing$Borough) # plyr::count() gives us frequency
# But this function count duplicates. Therefore median would be
# a different number.

#sort(
```

```
#  as.matrix(
#    plyr::count(
#    cbind(housing$Borough, housing$Value))
#  )[(179+196-1):(179+196+197-3),2]
#) # Sort() gives us all entries in order from plyr::count() function
# and we easily detect the duplicate, which is the second to last term
# that is not counted here. We want to add that term back by
# using c()

median(
  c(as.matrix(
    plyr::count(
    cbind(housing$Borough, housing$Value))
  )[(179+196-1):(179+196+197-3),2],466300792
  )
)
```

```
## [1] 2194200
```

```
# Answer is 3129300

# For learning purpose, there is a soft code we can use
plyr::count(cbind(housing$Borough)) # Notice where 179, 196, and 197 come from
```

```
##   x freq
## 1 1  664
## 2 2  829
## 3 3  851
## 4 4   77
## 5 5   33
```

```
median(
  c(as.matrix(
    plyr::count(
    cbind(housing$Borough, housing$Value))
  )[(
    plyr::count(cbind(housing$Borough))$freq[[1]]
    +plyr::count(cbind(housing$Borough))$freq[[2]]
    -1):(
      plyr::count(cbind(housing$Borough))$freq[[1]]
      +plyr::count(cbind(housing$Borough))$freq[[2]]
      +plyr::count(cbind(housing$Borough))$freq[[3]]
      -3),
    2],466300792 # This number needs to observed from the data.
  )
)
```

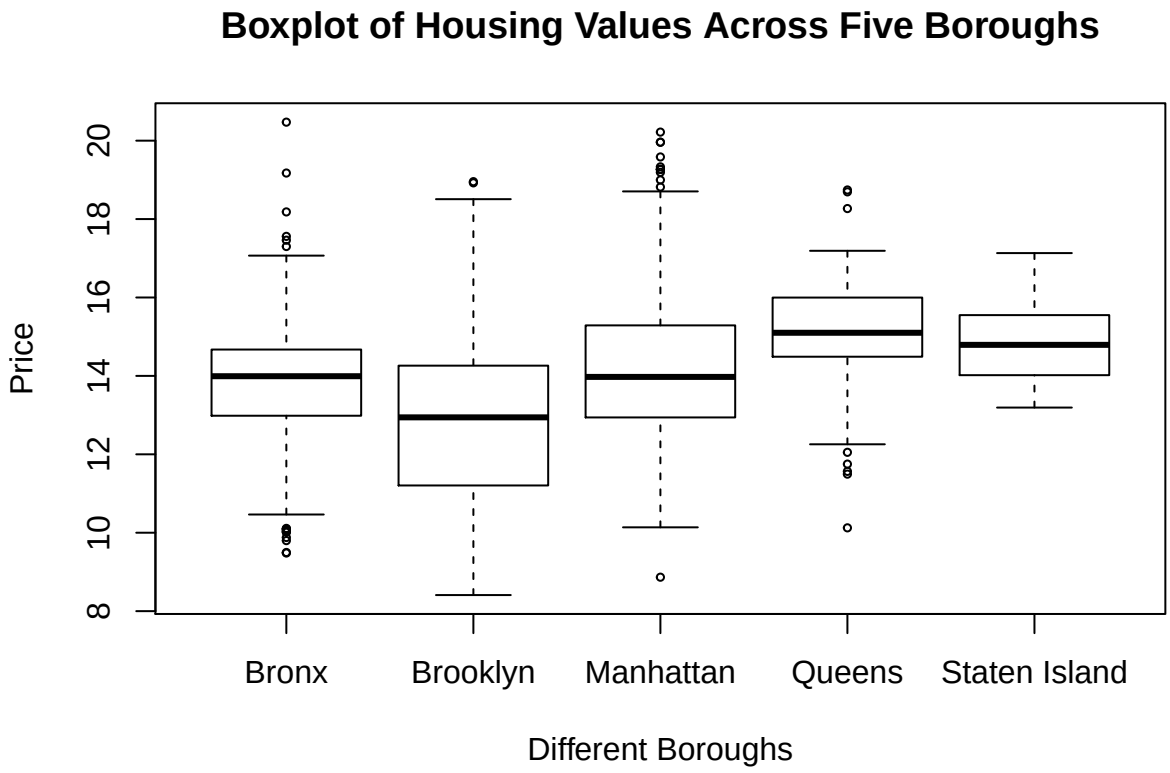
```
## [1] 1447344
```

```
# Answer is 3129300
```

9.2.6 vi. Boxplot

```
boxplot(
  housing$logValue ~ housing$Borough,
```

```
  data = housing,
  ylab = "Price",
  xlab = "Different Boroughs",
  main="Boxplot of Housing Values Across Five Boroughs",
  cex=.5)
```



9.2.7 vii. Median Property Value

```
head(cbind(housing$Value, housing$Borough), 3)
```

```
##         [,1] [,2]
## [1,] 2041200    1
## [2,]  897300    2
## [3,]  253490    2
```

```
median(housing$Value)
```

```
## [1] 934889
```

```
# Comment:
# The median of all five boroughs of housing values, which
# is just the median of all values in the dataset, would
# just be 2,302,650 USD.
```

10 Homework 3

Goals: in this assignment, we are going to scrape the 2016-2017 Brooklyn Nets Regular Season Schedule. We will take regular season schedule from http://www.espn.com/ and resemble the game listings in R data frame

10.1 Parts:

10.1.1 i. Function readLines()

```
# Upload data and call it nets1617:
nets1617 <- readLines("NetsSchedule.rtf",warn = FALSE)

# Check dimensions:
head(nets1617, 3) # Quick view
```

```
## [1] "{\\rtf1\\adeflang1025\\ansi\\ansicpg1252\\uc1\\adeff0\\deff0\\stshfdbch31505\\stshfloch31506\\s
## [2] "{\\f4\\fbidi \\fswiss\\fcharset0\\fprq2{\\*\\panose 020b0604020202020204}Helvetica;}{\\flomajor
## [3] "{\\fdbmajor\\f31501\\fbidi \\froman\\fcharset0\\fprq2{\\*\\panose 02020603050405020304}Times Ne
```

```
dim(data.frame(nets1617))
```

```
## [1] 1641    1
```

```
# Total number of characters:
length(nets1617)
```

```
## [1] 1641
```

```
# Number of characters each line:
# nchar(nets1617) # Too much information so we comment it out

# Total number of characters:
sum(nchar(nets1617))
```

```
## [1] 224592
```

```
# Comment:
# This line of code sums up all of the counts from each line
# of the information in the data.

# Maximum number of characters in a single line:
max(nchar(nets1617))
```

```
## [1] 342
```

10.1.2 ii. Who and When

Nets is playing against Boston Celtics the first and playing againt Chicago Bulls last for this season.

10.1.3 iii. Info about the First Game

The line that holds information about the first game is the following

```
#               <div class="mod-page-tabs mod-thirdnav-tabs" style="padding-top: 3px;"><ul class="ui-ta
```

10.1.4 iv. Capture Date

```
# Upload data and call it nets1617:
nets1617 <- readLines("NetsSchedule.rtf",warn = FALSE)

# An example: the first game:
#date_express <- "Wed, Oct 26"
#grep(date_express, nets1617)
#nets1617[[467]]

# In general:
date_express <- "[A-Z][a-z]{2},\\s[A-Z][a-z]{2}\\s[0-9]+"
grep(date_express, nets1617)
```

```
##  [1] 467 473 478 483 490 494 500 505 512 517 522 527 532 537 542 547 552
## [18] 558 563 568 573 578 584 591 598 610 613 619 624 629 634 641 647 650
## [35] 656 661 666 671 676 681 687 693 699 705 712 718 722 729 735 739 744
## [52] 749 754 759 764 770 777 783 788 793 798 803 808 814 821 828 834 838
## [69] 843 848 853 858 863 869 876 881 886 891 896 902 909
```

```
length(grep(date_express, nets1617))
```

```
## [1] 81
```

```
# First and alst:
nets1617[[grep(date_express, nets1617)[[1]]]]
```

```
## [1] "ass=\"stathead\"><td colspan=\"9\">2017 Regular Season Schedule</td></tr><tr class=\"colhead\">
```

```
nets1617[[grep(date_express, nets1617)[[81]]]]
```

```
## [1] "row team-46-4\"><td>Wed, Apr 12</td><td><ul class=\"game-schedule\"><li class=\"game-status\">@
```

```
# Comment: the first and the last in the results of
# grep() function matches the first and the last game.
# However, total length of the match somehow doesn't
# math 82.
```

10.1.5 v. Extract Date

```
# Now we code for this problem:
date_express <- "[A-Z][a-z]+, [A-Z][a-z]+ [0-9]+"
# date_express <- "[A-Z][a-z][a-z] [0-9][0-9]"
date <- regmatches(nets1617, gregexpr(date_express, nets1617))
# Comment:
# Up to this step, we have a variable date. But some of the
# entries will be character(0), and we want to get rid of
# them.

# Store in date without duplicates by using Filter()
date <- as.matrix(Filter(length, date))
head(date); dim(date)
```

```
##      [,1]
## [1,] "Wed, Oct 26"
## [2,] "Fri, Oct 28"
```

```
## [3,] "Sat, Oct 29"
## [4,] "Mon, Oct 31"
## [5,] "Wed, Nov 2"
## [6,] "Fri, Nov 4"
```

```
## [1] 81  1
```

```
# Comment:
# Two hours of pain finally give me a matrix with
# all the dates stored inside.
```

10.1.6 vi. Extract Time

```
# Recall from part iv) and part v) above:
date_express <- "[A-Z][a-z]+, [A-Z][a-z]+ [0-9]+"
date <- regmatches(nets1617, gregexpr(date_express, nets1617))
# Comment:
# Up to this step, we have a variable date. But some of the
# entries will be character(0), and we want to get rid of
# them.

# Store in date without duplicates by using Filter()
date <- as.matrix(Filter(length, date))
head(date); dim(date)
```

```
##      [,1]
## [1,] "Wed, Oct 26"
## [2,] "Fri, Oct 28"
## [3,] "Sat, Oct 29"
## [4,] "Mon, Oct 31"
## [5,] "Wed, Nov 2"
## [6,] "Fri, Nov 4"
```

```
## [1] 81  1
```

```
# Now we extract time the same way:
time_express <- "[0-9]:[0-9][0-9] [A-P]M"
time <- regmatches(nets1617, gregexpr(time_express, nets1617))
time <- as.matrix(Filter(length, time))
head(time); dim(time)
```

```
##      [,1]
## [1,] "7:30 PM"
## [2,] "7:30 PM"
## [3,] "8:00 PM"
## [4,] "7:30 PM"
## [5,] "7:30 PM"
## [6,] "7:30 PM"
```

```
## [1] 80  1
```

10.1.7 vii. Home or Away

```
# Recall from part iv) and part v) above:
date_express <- "[A-Z][a-z]+, [A-Z][a-z]+ [0-9]+"
```

```
date <- regmatches(nets1617, gregexpr(date_express, nets1617))

# Store in date without duplicates by using Filter()
date <- as.matrix(Filter(length, date))
head(date); dim(date)
```

```
##      [,1]
## [1,] "Wed, Oct 26"
## [2,] "Fri, Oct 28"
## [3,] "Sat, Oct 29"
## [4,] "Mon, Oct 31"
## [5,] "Wed, Nov 2"
## [6,] "Fri, Nov 4"
```

```
## [1] 81  1
```

```
# Now we extract time the same way:
time_express <- "[0-9]:[0-9][0-9] [A-P]M"
time <- regmatches(nets1617, gregexpr(time_express, nets1617))
time <- as.matrix(Filter(length, time))
head(time); dim(time)
```

```
##      [,1]
## [1,] "7:30 PM"
## [2,] "7:30 PM"
## [3,] "8:00 PM"
## [4,] "7:30 PM"
## [5,] "7:30 PM"
## [6,] "7:30 PM"
```

```
## [1] 80  1
```

```
# Now we extract home or away:
status_express <- "<li class=\"game-status\">@?"
status <- regmatches(nets1617, gregexpr(status_express, nets1617))
status <- ifelse(
  substring(status, 25, 26) == "@",
  "Home",
  "Away")
status <- as.matrix(Filter(length, status))
head(status); dim(status)
```

```
##      [,1]
## [1,] "Away"
## [2,] "Away"
## [3,] "Away"
## [4,] "Away"
## [5,] "Away"
## [6,] "Away"
```

```
## [1] 1641    1
```

```
status <- ifelse(
  substring(status, 25, 26) == "@",
  "Home",
  "Away")
status <- as.matrix(status)
```

```r
head(status); dim(status)
```

```
##      [,1]
## [1,] "Away"
## [2,] "Away"
## [3,] "Away"
## [4,] "Away"
## [5,] "Away"
## [6,] "Away"
```

```
## [1] 1641    1
```

10.1.8 ix. Opponent

```r
# Extract opponent:
# Now we extract time the same way:
nets1617 <- data.frame(nets1617)
oppo_express <- "<li class=\"team-name\"><.+>([a-zA-Z]|\\s)+</a>"
#oppo_express <- "Bosto"
oppo_express <- regmatches(nets1617[162:nrow(nets1617),], gregexpr(oppo_express, nets1617[162:nrow(nets
oppo_express <- as.matrix(Filter(length, oppo_express))
head(oppo_express); dim(oppo_express)
```

```
##      [,1]
## [1,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/bos/boston-celtics">Boston</
## [2,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/det/detroit-pistons">Detroi
## [3,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/cha/charlotte-hornets">Char
## [4,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/lac/la-clippers">LA</a>"
## [5,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/sa/san-antonio-spurs">San An
## [6,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/hou/houston-rockets">Houston
```

```
## [1] 41  1
```

```r
# Comment:
# This problem is wrong.
```

10.1.9 x. Putting All Together

```r
# Combine Date, Time, Opponent, and Home Together:
data <- cbind(
  time[1:10,],
  date[1:10,],
  status[1:10,],
  oppo_express[1:10,]
)

# Quickview for the first 10 rows:
data[1:10, ]
```

```
##       [,1]      [,2]          [,3]
##  [1,] "7:30 PM" "Wed, Oct 26" "Away"
##  [2,] "7:30 PM" "Fri, Oct 28" "Away"
##  [3,] "8:00 PM" "Sat, Oct 29" "Away"
```

```
##  [4,] "7:30 PM" "Mon, Oct 31" "Away"
##  [5,] "7:30 PM" "Wed, Nov 2"  "Away"
##  [6,] "7:30 PM" "Fri, Nov 4"  "Away"
##  [7,] "7:30 PM" "Tue, Nov 8"  "Away"
##  [8,] "7:00 PM" "Wed, Nov 9"  "Away"
##  [9,] "9:00 PM" "Sat, Nov 12" "Away"
## [10,] "0:30 PM" "Mon, Nov 14" "Away"
##       [,4]
##  [1,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/bos/boston-celtics">Boston
##  [2,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/det/detroit-pistons">Detro
##  [3,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/cha/charlotte-hornets">Cha
##  [4,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/lac/la-clippers">LA</a>"
##  [5,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/sa/san-antonio-spurs">San
##  [6,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/hou/houston-rockets">Houst
##  [7,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/lal/los-angeles-lakers">Los
##  [8,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/orl/orlando-magic">Orlando
##  [9,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/phi/philadelphia-76ers">Phi
## [10,] "<li class="team-name"><a href="http://www.espn.com/nba/team/_/name/tor/toronto-raptors">Toron
```

11 Homework 4

Writing functions and using them as larger parts of code, more practice with bootstrap, fitting models by optimizing loss functions.

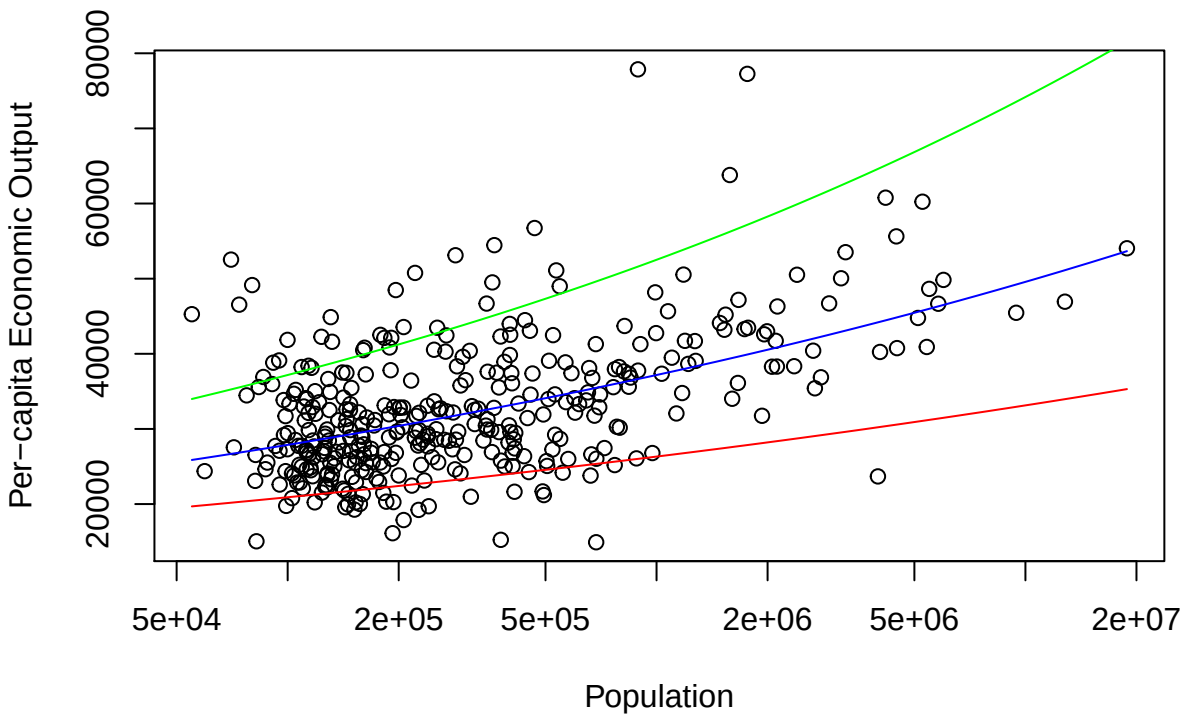
11.1 Parts:

11.1.1 i. Plot the data as in lecture, with per-capita GMP on the y-axis and population on x-axis. Using the ** curve() ** function to add the model using the default values provided in lecture. Add two more curves using $\beta_1 = 0.1$ and $\beta_1 = 0.15$ with the same value of β_0 from class. Make all three curves different colors using the ** col ** option.

```
# Upload data:
gmp <- read.table("gmp.txt", header = TRUE)
gmp$pop <- round(gmp$gmp/gmp$pcgmp)
head(gmp) # Check
```

```
##                          city        gmp pcgmp    pop
## 1                Abilene, TX 3.8870e+09 24490 158718
## 2                  Akron, OH 2.2998e+10 32889 699261
## 3                 Albany, GA 3.9550e+09 24269 162965
## 4 Albany-Schenectady-Troy, NY 3.1321e+10 36836 850282
## 5             Albuquerque, NM 3.0727e+10 37657 815970
## 6              Alexandria, LA 3.8790e+09 25494 152153
```

```
# Plot
library(ggplot2)
plot(gmp$pop, gmp$pcgmp, log = "x", xlab = "Population",
ylab = "Per-capita Economic Output")
curve(6611*x^{1/8}, add = TRUE, col = "blue")
curve(6611*x^{0.1}, add = TRUE, col = "red")
curve(6611*x^{0.15}, add = TRUE, col = "green")
```



11.1.2 ii. Create Function ** mse() **

Calculate the mean squared error of the model on a given data set. ** mse() ** should have three arguments: (1) a numeric vector of length two, with the first component corresponding to β_0 and the second to β_1, (2) a numeric vector containing the population values (X), and (3) a numeric vector containing the values of the per-capita GMP (Y). The output of your function should be a single numeric value (the mean squared error). The second and third arguments should have as default values the variables pop and pcgmp, respectively, from the gmp dataset.

```
# Write function:
mse <- function(vector) {
  #beta0 <- 6611
  #beta1 <- 0.15
  beta0 <- vector[1]
  beta1 <- vector[2]
  X <- gmp$pop
  Y <- gmp$pcgmp
  difference <- Y - beta0*X^(beta1)
  mse <- (1/nrow(gmp))*(sum(difference^2))
  #mse
  return(mse)
}

# Check with:
mse(c(6611, 0.15))
```

```
## [1] 207057513
```

```
mse(c(5000, 0.10))
```

```
## [1] 298459914
```

```
# Comment:
# the first answer is 207057513 and
# the second answer is 298459914
```

11.1.3 iii. Using function ** nlm() **

This is non-linear minimization, ** nlm() ** takes two required arguments, a function to minimize and a starting value for that function. Run ** nlm() ** three times with your function ** mse() ** and three starting pairs for β_0 and β_1 as in

```
nlm(mse, c(beta0 = 6611, beta1 = 1/8))
```

```
## $minimum
## [1] 61857060
##
## $estimate
## [1] 6611.0000000    0.1263177
##
## $gradient
## [1] 50.048639 -9.976327
##
## $code
## [1] 2
##
## $iterations
## [1] 3
```

```
# Comment:
# "minimum" is the minimum value of the function
# defined in the nlm() argument inputs. In this
# case, it is the mse() function.
# "estimate" is the point at which the minimum value
# of the function, mse(), is obtained.

# Note that one can access with $ sign:
nlm(mse, c(beta0 = 6611, beta1 = 1/8))$estimate
```

```
## [1] 6611.0000000    0.1263177
```

```
# Check another pair:
nlm(mse, c(beta0 = 6610, beta1 = 1/9))
```

```
## $minimum
## [1] 61857008
##
## $estimate
## [1] 6610.0000002    0.1263297
##
## $gradient
## [1] 51.33407 10.51508
##
```

```
## $code
## [1] 2
##
## $iterations
## [1] 5
```

```
nlm(mse, c(beta0 = 6610, beta1 = 1/9))$estimate
```

```
## [1] 6610.0000002    0.1263297
```

11.1.4 iv. Write function ** plm() **

Write a function ** plm() ** which estimates the parameters β_0 and β_1 of the model by minimizing the MSE. The function ** plm() ** should take the following four arguments: (1) an initial guess for β_0, (2) an initial guess for β_1, (3) a vector containing the population values (X), and (4) a vector containing the per-capita GMP values (Y).

```
# From previous part:
# Write function:
mse <- function(vector) {
  #beta0 <- 6611
  #beta1 <- 0.15
  beta0 <- vector[1]
  beta1 <- vector[2]
  X <- gmp$pop
  Y <- gmp$pcgmp
  difference <- Y - beta0*X^(beta1)
  mse <- (1/nrow(gmp))*(sum(difference^2))
  #mse
  return(mse)
}

# Check with:
mse(c(6611, 0.15))
```

```
## [1] 207057513
```

```
mse(c(6611, 0.14)) # Fix beta0, change beta1
```

```
## [1] 104079527
```

```
mse(c(6611, 0.13)) # Fix beta0, change beta1
```

```
## [1] 64529166
```

```
mse(c(6611, 0.15))
```

```
## [1] 207057513
```

```
mse(c(6610, 0.15)) # Fix beta1, change beta0
```

```
## [1] 206892998
```

```
mse(c(6609, 0.15)) # Fix beta1, change beta0
```

```
## [1] 206728578
```

```r
# Note that one can access with $ sign:
vector <- c(beta0 = 6611, beta1 = 0.15)
nlm(mse, vector) # Check
```

```
## $minimum
## [1] 61857060
##
## $estimate
## [1] 6610.9999997    0.1263182
##
## $gradient
## [1]   51.76354 -210.18952
##
## $code
## [1] 2
##
## $iterations
## [1] 7
```

```r
# Create function:
plm <- function(vector,
                X,
                Y){
  # Parameters:
  iter <- 0
  max.iter <- 10000
  beta0 <- vector[1]
  beta1 <- vector[2]
  #X <- gmp$pop
  #Y <- gmp$pcgmp
  initial.mse <- mse(vector)

  # Update beta0 and beta1:
  while (iter < max.iter) {
    iter <- iter + 1
    beta0 <- beta0 - mean(X^(beta1))
    beta1 <- beta1 - mean(beta1*X^(beta1-1))
    vector <- c(beta0 = beta0, beta1 = beta1)
    if (mse(vector) < initial.mse) {
    initial.mse <- mse(vector)
    } else break()
  }
  return(
    list(
      estimate = vector,
      min.mse = initial.mse
    )
  )
}

# Ex:
vector <- c(beta0 = 6611, beta1 = 0.15)
plm(vector,X = gmp$pop,Y = gmp$pcgmp)
```

```
## $estimate
## beta0.beta0 beta1.beta1
## 4916.279377    0.148906
##
## $min.mse
## [1] 62607483
```

```
# Ex:
vector <- c(beta0 = 5000, beta1 = 0.10)
plm(vector,X = gmp$pop,Y = gmp$pcgmp)
```

```
## $estimate
##  beta0.beta0  beta1.beta1
## 4996.4409657    0.0999984
##
## $min.mse
## [1] 298459914
```

```
# Comment:
# I don't know what is the correct gradient.
# I am having problems of calculating the correct gradient.
```

11.1.5 v. Bootstrap in a simple example

11.1.5.1 (a) Mean and Standard Deviation

Calculate the mean and standard deviation of the per-capita GMP values in your dataset using built-in function ** mean() ** and ** sd() **.

```
# Mean and SD:
mean(gmp$pcgmp)
```

```
## [1] 32922.53
```

```
sd(gmp$pcgmp)
```

```
## [1] 9219.663
```

11.1.5.2 (b) Mean of Particular Entries

```
# For example:
indices <- c(1,5,1,3)
gmp$pcgmp[indices] # Check values
```

```
## [1] 24490 37657 24490 24269
```

```
# Compute mean:
mean(gmp$pcgmp[indices])
```

```
## [1] 27726.5
```

```
# Mean from randomly drawing n observations::
mean(sample(gmp$pcgmp, nrow(gmp)))
```

```
## [1] 32922.53
```

```
# Write a function:
boot.mean <- function() {
  indices <- sample(1:nrow(gmp), nrow(gmp))
```

```
  bootstrap.mean <- mean(gmp$pcgmp[indices])
  return(bootstrap.mean)
}

# Ex:
boot.mean()
```

```
## [1] 32922.53
```

11.1.5.3 (c) Create function ** bootstrap.means **

Create a vector ** bootstrap.means **, which ahs the mean per-capita GMP for one hundred bootstrap samples. Here you'll want to create a loop where each iteration performs a new bootstrap sample. In each iteration you use** sample() ** to create an indices vector containing the indices of your bootstrap sample and then using indices calculate the mean using your function from (b).

```
# Recall function:
# Write a function:
boot.mean <- function() {
  indices <- sample(1:nrow(gmp), nrow(gmp))
  bootstrap.mean <- mean(gmp$pcgmp[indices])
  return(bootstrap.mean)
}

# Ex:
boot.mean()
```

```
## [1] 32922.53
```

```
# Sample:
n <- nrow(gmp); n
```

```
## [1] 366
```

```
indices <- sample(1:n, n, replace = TRUE)
head(indices)
```

```
## [1]  32 334  67 194 146  39
```

```
# Calculate the mean
bootstrap.means <- mean(gmp$pcgmp[indices])
bootstrap.means
```

```
## [1] 33401.62
```

```
# Write a loop:
i <- 1
B <- 100
bootstrap.means <- NULL
while (i <= B) {
# i <- 2
# Set up samples:
n <- nrow(gmp) #; n
indices <- sample(1:n, n, replace = TRUE)
# head(indices)

# Calculate the mean
boot.mean <- mean(gmp$pcgmp[indices])
```

```
boot.mean

# Store:
bootstrap.means <- cbind(bootstrap.means, boot.mean)
#bootstrap.means

# Loop + 1
i <- i + 1
}

# Mean:
mean(bootstrap.means)
```

```
## [1] 32942.91
```

11.1.5.4 (d) Calculate the SD

```
# Recall the previous code for writing functions:
# Write a loop:
i <- 1
B <- 100
bootstrap.means <- NULL
while (i <= B) {
# i <- 2
# Set up samples:
n <- nrow(gmp) #; n
indices <- sample(1:n, n, replace = TRUE)
# head(indices)

# Calculate the mean
boot.mean <- mean(gmp$pcgmp[indices])
boot.mean

# Store:
bootstrap.means <- cbind(bootstrap.means, boot.mean)
#bootstrap.means

# Loop + 1
i <- i + 1
}

# Here we calculate SD:
# Standard Deviation:
sd(bootstrap.means)
```

```
## [1] 515.3785
```

11.1.6 vi. Write Function ** plm.bootstrap() **

Calculate a bootstrap estimates of the standard errors of β_0 and β_1. It should take the same arguments as ** plm() ** along with an argument B for the number of bootstrap samples to draw. Let the default be $B = 100$. ** plm.bootstrap() ** should return standard errors for both parameters. This function should

call your plm() function repeatedly. What standard errors do you get for the two parameters using initial conditions $\beta_0 = 6611$ and $\beta_1 = 0.15$ Initial conditions $\beta_0 = 5000$, and $\beta_1 = 0.1$.

```
# Solution:
# Write a function
plm.bootstrap <- function() {
i <- 1
B <- 100
se.matrix <- matrix(0, nrow = B, ncol = 2)
while (i <= B) {
# Indices:
n <- nrow(gmp) #; n
indices <- sample(1:n, n, replace = TRUE)
#head(indices)

# Call:
plm(vector,X = gmp$pop[indices],Y = gmp$pcgmp[indices])[1]

# Update se.matrix:
se.matrix[i,] <- t(
  data.frame(
  plm(vector,X = gmp$pop[indices],Y = gmp$pcgmp[indices])[1]
  )
)
i <- i + 1
}
return(
  list(
    # Compute SE:
    SE.Beta0 = sd(se.matrix[,1]),
    SE.Beta1 = sd(se.matrix[,2])
  )
)
}

# Ex:
plm.bootstrap()
```

```
## $SE.Beta0
## [1] 0.02138394
##
## $SE.Beta1
## [1] 5.264048e-08
```

11.1.7 v. Test on 2013 Data

```
# Solution:

# Upload data:
gmp2013 <- read.table("gmp-2013.txt", header = TRUE)
gmp2013$pop <- round(gmp2013$gmp/gmp2013$pcgmp)
head(gmp2013) # Check
```

```
##                          Area        gmp pcgmp    pop
```

```
## 1                 Abilene, TX 5.9240e+09 35366 167506
## 2                   Akron, OH 2.9662e+10 42033 705684
## 3                  Albany, GA 4.8570e+09 31196 155693
## 4                  Albany, OR 3.1300e+09 26354 118768
## 5 Albany-Schenectady-Troy, NY 4.3562e+10 49621 877894
## 6             Albuquerque, NM 3.9618e+10 43884 902789
```

```
# Write a function
plm.bootstrap <- function() {
i <- 1
B <- 100
se.matrix <- matrix(0, nrow = B, ncol = 2)
while (i <= B) {
# Indices:
n <- nrow(gmp) #; n
indices <- sample(1:n, n, replace = TRUE)
#head(indices)

# Call:
plm(vector,X = gmp2013$pop[indices],Y = gmp2013$pcgmp[indices])[1]

# Update se.matrix:
se.matrix[i,] <- t(
  data.frame(
  plm(vector,X = gmp2013$pop[indices],Y = gmp2013$pcgmp[indices])[1]
  )
)
i <- i + 1
}
return(
  list(
    # Compute SE:
    SE.Beta0 = sd(se.matrix[,1]),
    SE.Beta1 = sd(se.matrix[,2])
  )
)
}

# Ex:
plm.bootstrap()
```

```
## $SE.Beta0
## [1] 0.02171628
##
## $SE.Beta1
## [1] 5.240288e-08
```

```
# Comment:
# Observe that from the 2013 data
# both estimates have increased a
# little.
```

12 Homework 5

12.1 Part 1: Inverse Transform Method

Continue working with World Top Incomes Database, and the Pareto distribution as in Lab 4 and Homework 5. Recall that for most countries in most time periods, the upper end of the income distribution roughly follows a Pareto distribution, with probability density function

$$f(x) = \frac{(a-1)}{x_{\min}}\left(\frac{x}{x_{\min}}\right)^{-a}$$

for incomes $X \geq x_{\min}$. In Homework 5, we estimated the parameter a based on the 'wtid-report' data set for years ranging from 1913 to 2015.

Now suppose that we are interested in simulating the upper end of income just for 2015 using the Pareto distribution written above. Let the 'upper end' begin at the 99th annual income percentile for 2015 (meaning, let $x_{\min} = \$407,760$). Then we can model the upper end of income for 2015 by the Pareto distribution having pdf

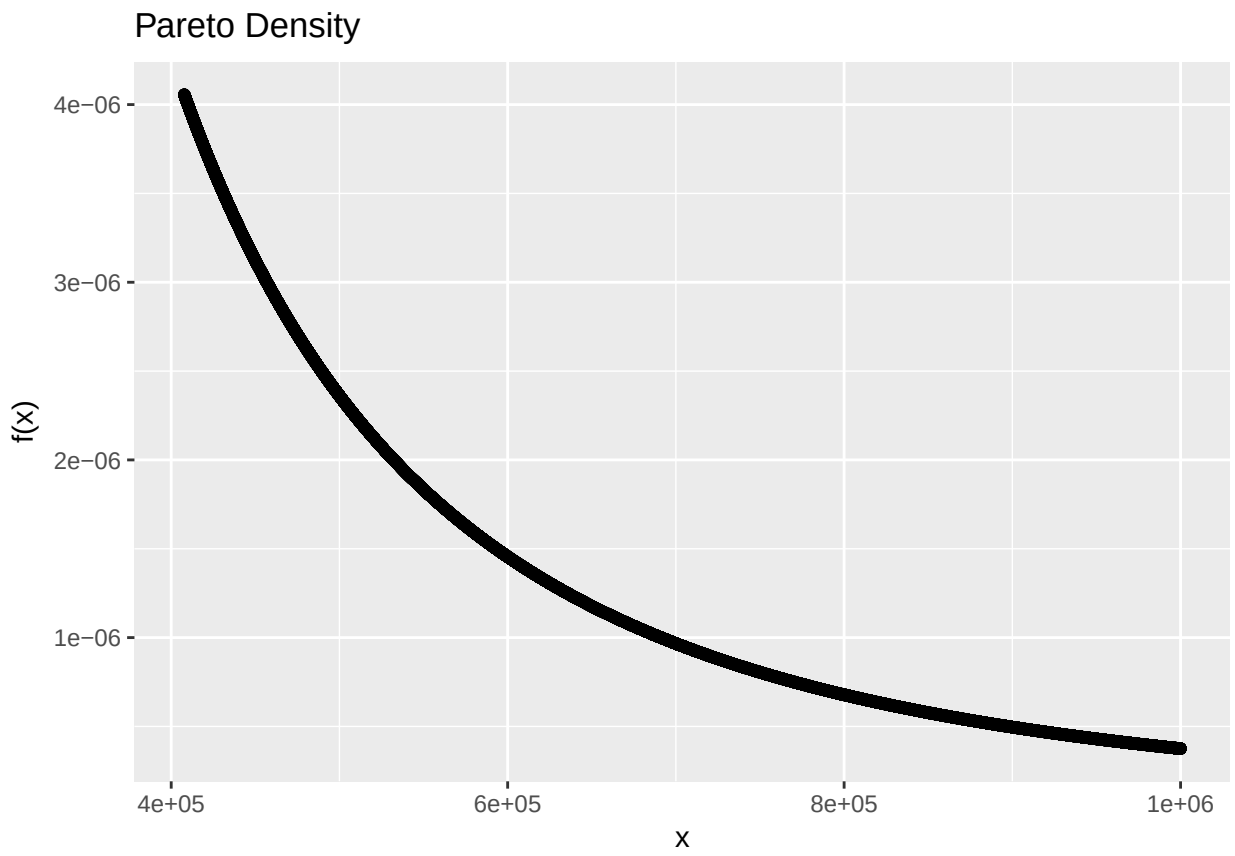
$$f(x) = \frac{\hat{a}-1}{x_{\min}}\left(\frac{x}{x_{\min}}\right)^{-b} = \frac{1.654}{407760}\left(\frac{x}{407760}\right)^{-2.654}$$

using $\hat{a} = 2.654$ which was the Pareto exponent estiamte for 2015 from Homework 5.

Perform the following tasks:

(1) Define a function 'f' which takes three inputs x, a vector and scalars a and $x_{\min}$ having default values of $a = \hat{a}$ and $x_{\min} = \$407,760$. The function should output $f(x)$ for a given input $x > x_{\min}$. Plot the function between $x_{\min}$ and 1,000,000. Make sure your plot is labeled appropriately.

```
f <- function(x, xmin = 407760, a = 2.654) {
  return(ifelse(x < xmin, 0, ((a-1)/xmin)*(x/xmin)^(-a)))
}

xmin <- 407760
x <- seq(xmin, 1000000, by = 10)
library(ggplot2)
ggplot() +
  geom_point(mapping = aes(x = x, y = f(x))) +
  labs(title = "Pareto Density")
```



(2) For $x > x_{\min}$, the cdf equals

$$F(x) = 1 - \left(\frac{x}{x_{\min}}\right)^{-a-1}$$

Find the inverse function $F^{-1}(u)$ and define a function 'upper.income'. The function should have three inputs, a vector u and scalars a and $x_{\min}$ having default values of $a = \hat{a}$ and $x_{\min} = \$407,760$. The function should output $F^{-1}(u)$ for a given input $u \in (0, 1)$. Make sure ** upper.income(.5) ** returns 620020.2

$$u = 1 - \left(\frac{x}{x_{\min}}\right)^{-a-1} \rightarrow x = x_{\min}(1-u)^{-\frac{1}{a-1}}$$

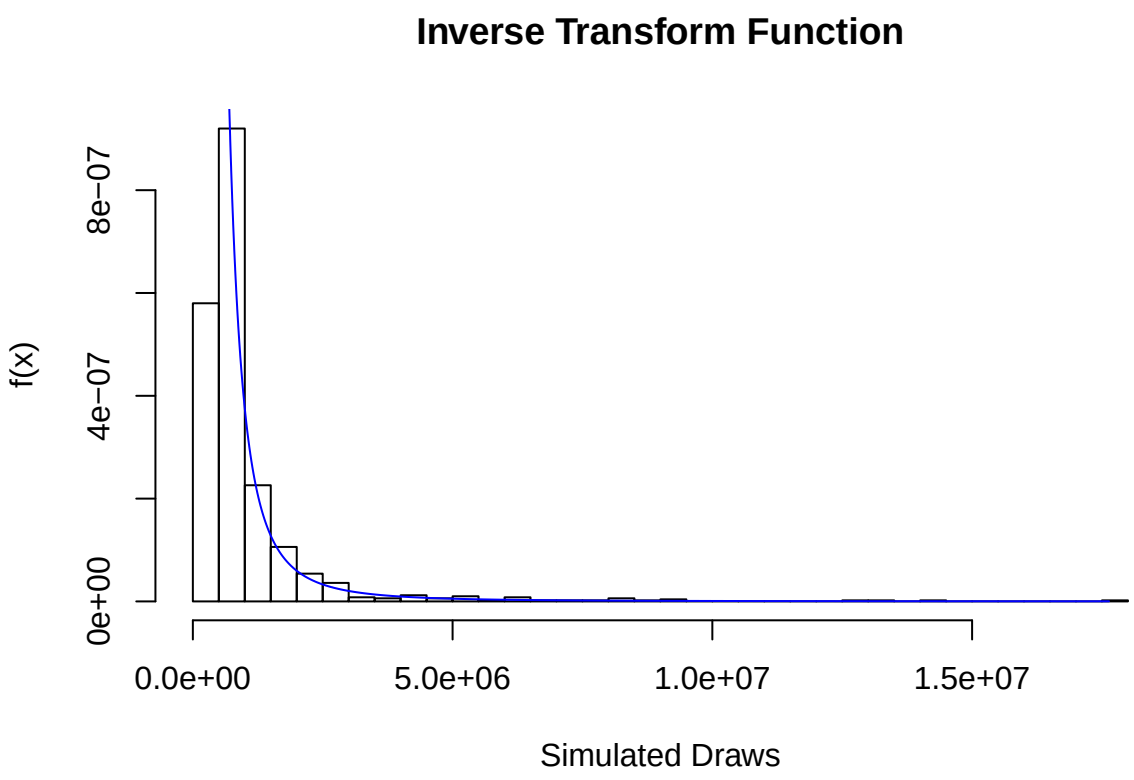
```
upper.income <- function(u, xmin = 407760, a = 2.654) {
  return(ifelse((u < 0 | u > 1), NA, xmin*(1-u)^(-1/(a-1))))
}
upper.income(0.5)
```

```
## [1] 620020.2
```

(3) Using the Inverse Transform Method, simulate 1000 draws from the Pareto distribution and plot a histogram of your values. Overlay the simulated distribution with the Pareto density. Make sure to label the histogram appropriately.

```
n <- 1000
sims <- upper.income(runif(n))

hist(sims, prob = TRUE, breaks = 35,
```

```
     xlab = "Simulated Draws", ylab = "f(x)",
     main = "Inverse Transform Function")
y <- seq(min(sims), max(sims), by = 1000)
lines(y, f(y), col = "blue")
```



(4) Using your simulated set, estimate the median income for the richest 1% of the world. Recall from lab that hte proportion of people whose income is at least $x_{\min}$ whose income is also at or above any level $w \geq x_{\min}$ is

$$Pr(X \geq w) = \left(\frac{w}{x_{\min}}\right)^{-a+1}$$

Compare your estimated 50th percentile to the actual 50th percentile of the Pareto distribution.

The actual 50th percentile of the Pareto distribution is given by

$$0.5 = \left(\frac{w}{x_{\min}}\right)^{-a+1} \rightarrow w = x_{\min}(0.5)^{-\frac{1}{-a+1}}$$

```
ahat <- 2.654
a <- ahat
sim_med <- median(sims)
actual_med <- xmin*(0.5)^(1/(-a+1))
sim_med; actual_med
```

```
## [1] 609700.7
```

```
## [1] 620020.2
```

12.2 Part 2: Reject-Accept Method

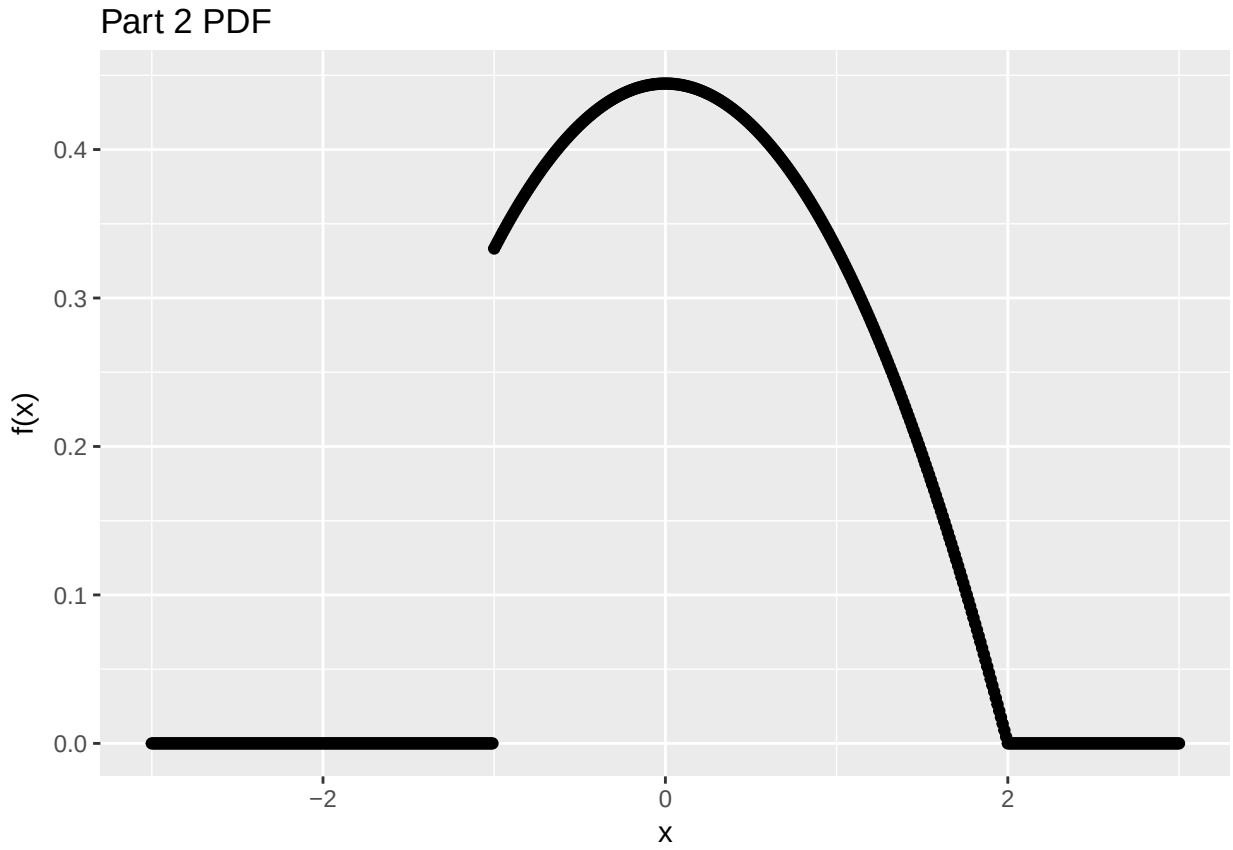
Let random variable X denote the temperature at which a certain chemical reaction takes place. Suppose that X has probability density function

$$f(x) = \begin{cases} \frac{1}{9}(4 - x^2) & -1 \leq x \leq 2 \\ 0 & \text{otherwise} \end{cases}$$

Perform the following tasks:

(5) Write a function 'f' that takes as input a vector x and returns a vector of $f(x)$ values. Plot the function between -3 and 3. Make sure your plot is labeled appropriately.

```
f <- function(x) {
  ifelse((x < -1 | x > 2), 0, (1/9)*(4 - x^2))
}
x <- seq(-3,3,by = 0.01)
ggplot() +
  geom_point(mapping = aes(x = x, y = f(x))) +
  labs(title = "Part 2 PDF")
```



(6) Determine the maximum of $f(x)$ and find an envelope function $e(x)$ by using a uniform density for $g(x)$ and setting $\alpha = 1/max_x\{f(x)\}$ as in class. Write a function e which takes as input a vector x and returns a vector of $e(x)$ values.

Note that

$$f'(x) = \begin{cases} -\frac{2x}{9} & -1 \leq x \leq 2 \\ 0 & \text{otherwise} \end{cases}$$

Setting the above equal to 0 we find that the maximum occurs at $x = 0$ and $f(0) = 4/9$.

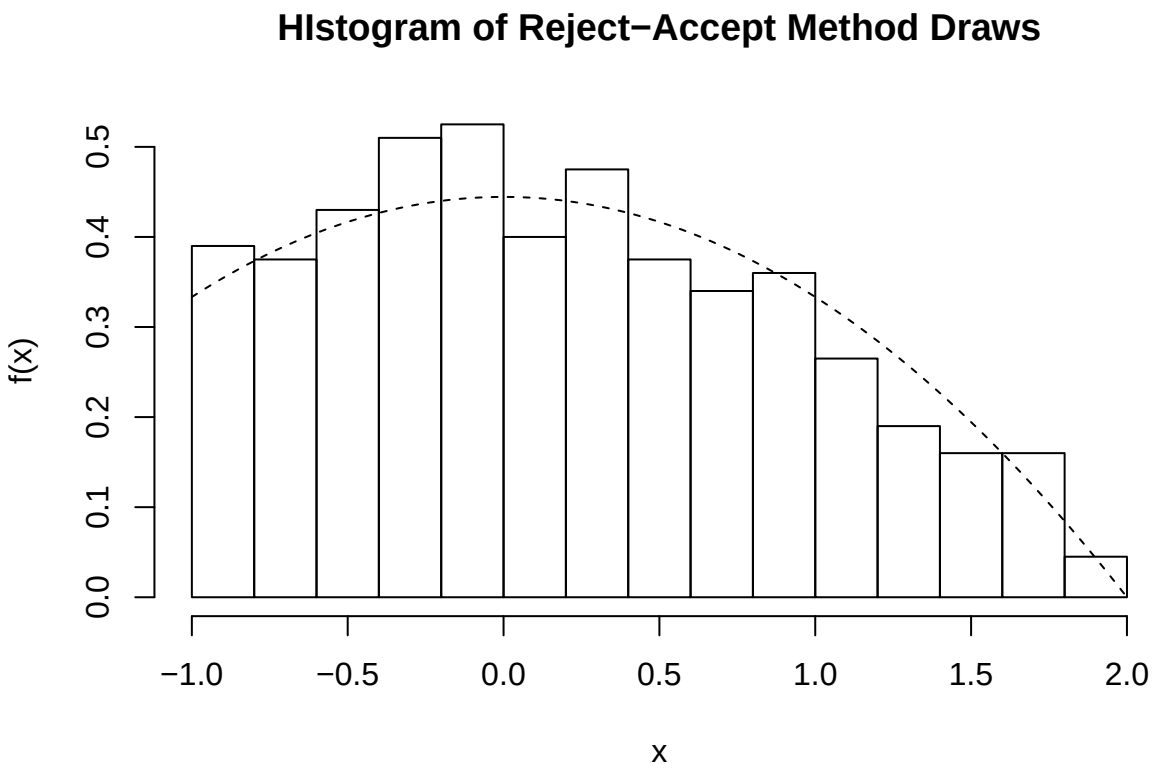
```
f.max <- 4/9
e <- function(x) {
  ifelse((x < -1 | x > 2), Inf, f.max)
}
```

(7) Using the Accept-Reject Algorithm, write a program that simulates 1000 draws from the probability density function $f(x)$ from above equation.

```
n.samps <- 1000 # Samples desired
n <- 0 # Counter for number samples accepted
samps <- numeric(n.samps) # Initialize the vector of output
while ( n < n.samps) {
  y <- runif(1, min = -1, max = 2) # Random draw from g
  u <- runif(1)
  if (u < f(y)/e(y)) {
    n <- n + 1
    samps[n] <- y
  }
}
```

(8) Plot a histogram of your simulated data with the density function f overlaid in the graph. Label plot appropriately.

```
x <- seq(-1, 2, length = 100)
hist(samps, prob = T,
     ylab = "f(x)", xlab = "x",
     main = "HIstogram of Reject-Accept Method Draws")
lines(x, f(x), lty = 2)
```



12.3 Part 3: Simulation with Built-in R Functions

Consider the following "random walk" procedure:

a. Start with $x = 5$

b. Draw a random number r uniformly between -2 and 1.

c. Replace x with $x + r$

d. Stop if $xle0$

e. Else repeat

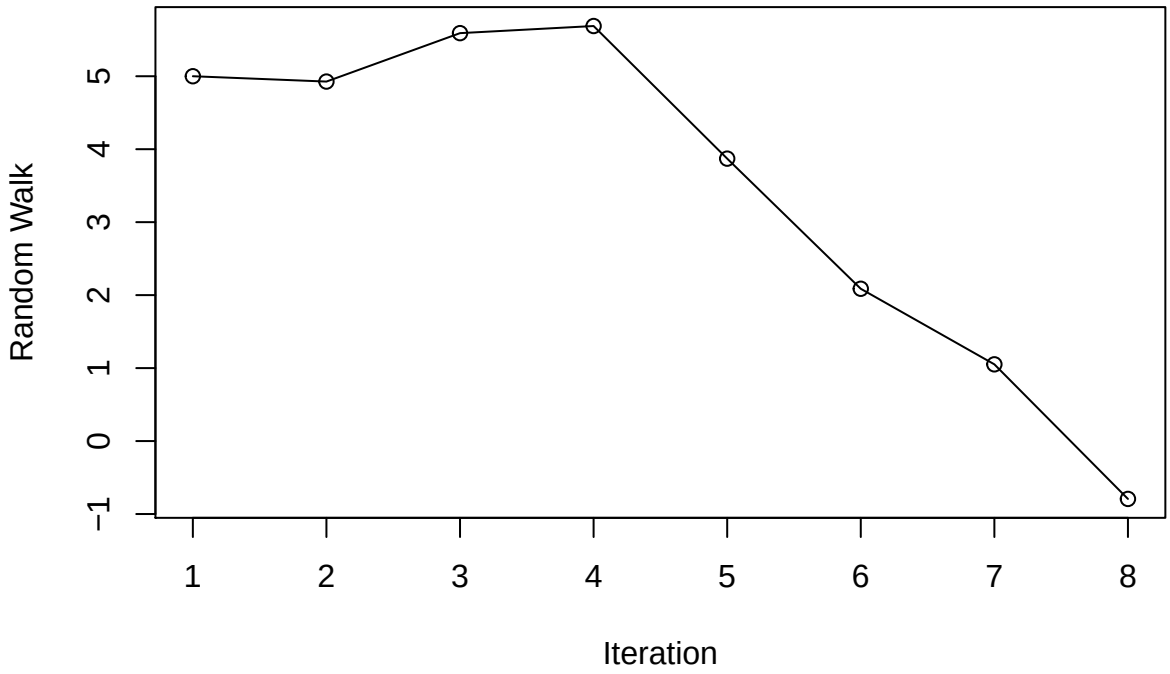
(9) Write a 'while()' loop to implement this procedure. Importantly, save all the positive values of x that were visited in this procedure in a vector called 'x.vals', and display its entries.

```
x.vals <- 5
i <- 1
while(x.vals[i] > 0) {
  r <- runif(1, min = -2, max = 1)
  x.vals <- c(x.vals, x.vals[i] + r)
  i <- i+1
}
x.vals
```

```
## [1]  5.0000000  4.9267869  5.5904759  5.6879449  3.8706945  2.0869534
## [7]  1.0515597 -0.7929799
```

(10) Produce a plot of the random walk values 'x.vals' from above versus the iteration number. Make sure the plot has an appropriately labeled x-axis and y-axis. Also use type = "o" so that we can see both points and lines

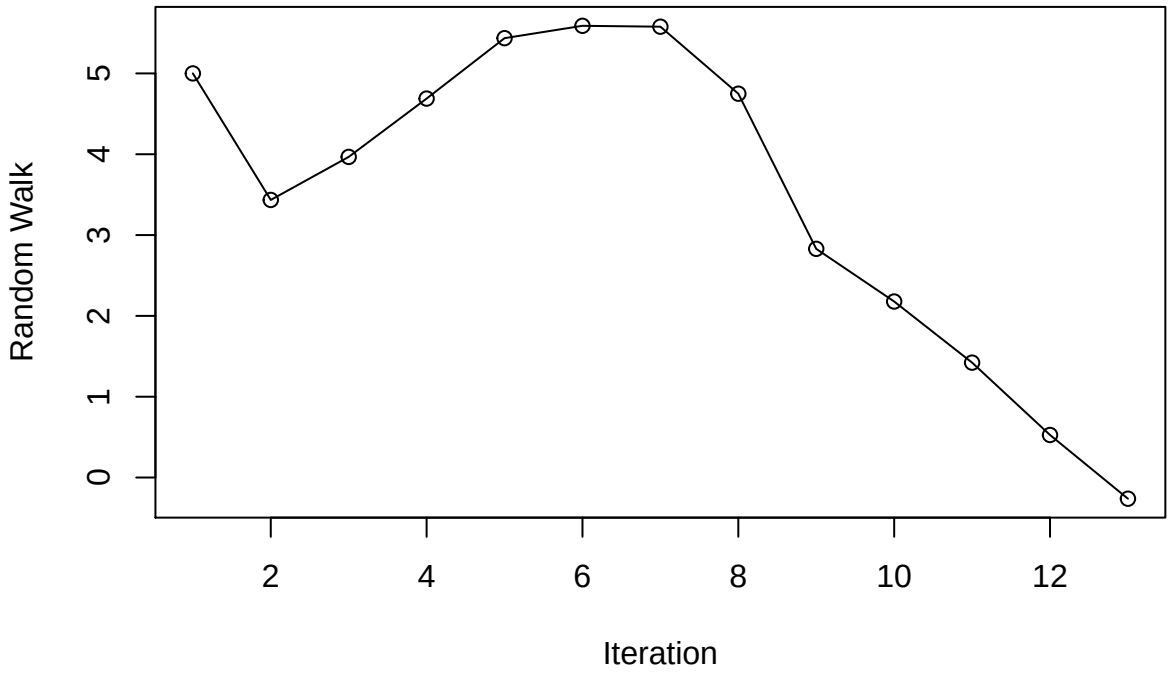
```
plot(1:length(x.vals),
     x.vals,
     type = "o",
     xlab = "Iteration",
     ylab = "Random Walk")
```



(11) Write a function ** random.walk() ** to perform the random walk procedure that you implemented in question (9). Its inputs should be 'x.start', a numeric value at which we will start the random walk, which takes a default value of 5; and 'plot.walk', a boolean value, indicating whether or not we want to produce a plot of the random walk values 'x.vals' versus the iteration number as a side effect, which takes a default value of TRUE. The output of your function should be a list with elements: 'x.vals', a vector of the random walk values as computed above; and 'num.steps', the number of steps taken by the random walk before terminating. Run your function twice with the default inputs, and then twice times 'x.start' equal to 10 and 'plot.walk = FALSE'.

```
random.walk <- function(x.start = 5, plot.walk = TRUE) {
x.vals <- 5
i <- 1
while(all(x.vals[i] > 0)) {
r <- runif(1, min = -2, max = 1)
x.vals <- c(x.vals, x.vals[i] + r)
i <- i+1
}
```

```
if (plot.walk) {plot(1:length(x.vals), x.vals, type = "o", xlab = "Iteration", ylab = "Random Walk")}
return(list(x.vals = x.vals, num.steps = i))
}
random.walk()
```




```
## $x.vals
##  [1]  5.0000000  3.4346394  3.9669876  4.6885661  5.4362406  5.5889408
##  [7]  5.5782259  4.7491101  2.8292691  2.1777066  1.4216262  0.5262276
## [13] -0.2620888
##
## $num.steps
## [1] 13
```

```
random.walk(x.start = 10, plot.walk = FALSE)
```

```
## $x.vals
##  [1]  5.0000000  4.1830537  2.4469732  2.4496717  1.2953851  1.7087114
##  [7]  2.1106721  1.1846789  1.0483359 -0.8863391
##
## $num.steps
## [1] 10
```

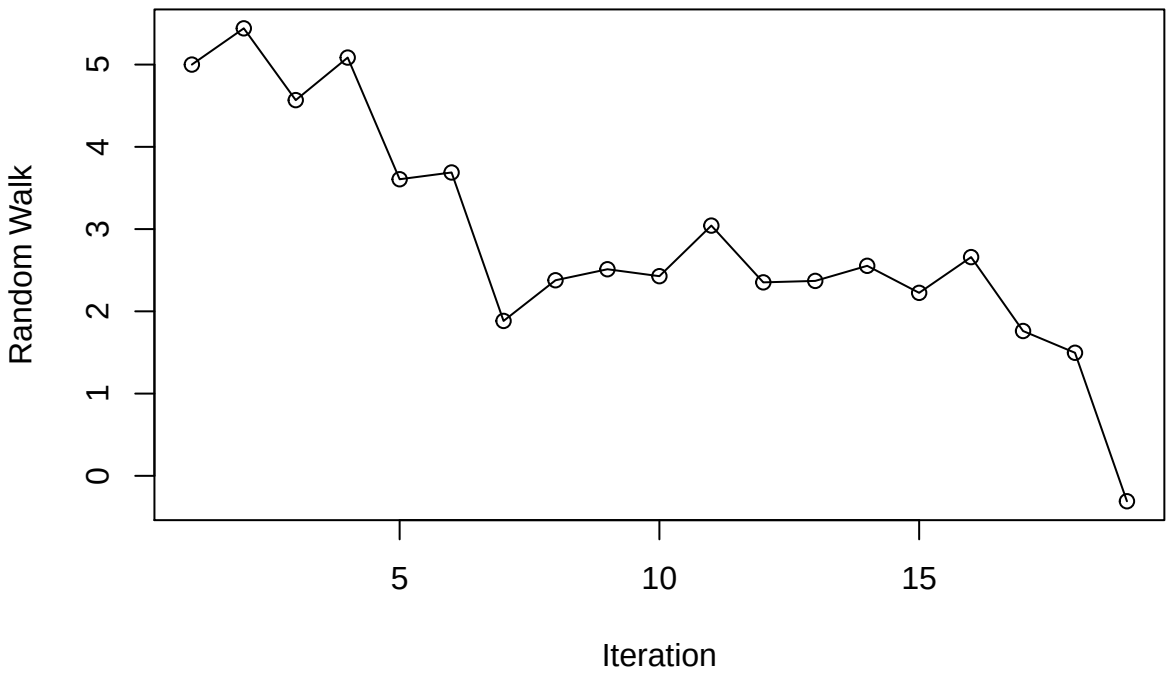
(12) We would like to answer the following question using simulation: if we start our random walk process, as defined above, at $x = 5$, what is the expected number of iterations we need until it terminates? To estiamte the solution produce 10,000 such random walks and calculate the average number of iterations in the 10,000 random walks you produce. You'll want to turn the plot off here.

```
n <- 10000
iters <- rep(NA, n)
for (i in 1:n) {
  iters[i] <- random.walk(plot.walk = FALSE)$num.steps
}
mean(iters)
```

```
## [1] 12.3195
```

(13) Modify your function 'random.walk()' defined previously so that it takes an additional argument seed: this is an integer that should be used to set the seed of the random number generator, before the random walk begins, with set.seed(). But, if seed is NULL, the default, then no seed should be set. Run your modified function 'random.walk()' function several times with the default inputs, then run it several times with the input seed equal to 33.

```
random.walk <- function(x.start = 5, plot.walk = TRUE, seed = NULL) {
  if (!is.null(seed)) {set.seed(seed)}
    x.vals <- 5
    i <- 1
  while(all(x.vals[i] > 0)) {
  r <- runif(1, min = -2, max = 1)
  x.vals <- c(x.vals, x.vals[i] + r)
  i <- i+1
  }
  if (plot.walk) {plot(1:length(x.vals),
                       x.vals, type = "o",
                       xlab = "Iteration",
                       ylab = "Random Walk")}
  return(list(x.vals = x.vals, num.steps = i))
}
random.walk()
```




```
## $x.vals
##  [1]  5.000000  5.440654  4.568056  5.085409  3.606083  3.688294  1.883421
##  [8]  2.377741  2.511765  2.427431  3.042526  2.352339  2.370084  2.553753
## [15]  2.224791  2.659171  1.760933  1.496461 -0.308821
##
## $num.steps
## [1] 19
```

```
random.walk(seed = 33, plot.walk = FALSE)
```

```
## $x.vals
##  [1]  5.0000000  4.3378214  3.5217724  2.9729590  3.7295869  4.2612312
##  [7]  3.8132800  3.1246550  2.1542497  0.2008006 -1.4452259
##
## $num.steps
## [1] 11
```

12.4 Part 4: Monte Carlo Integration

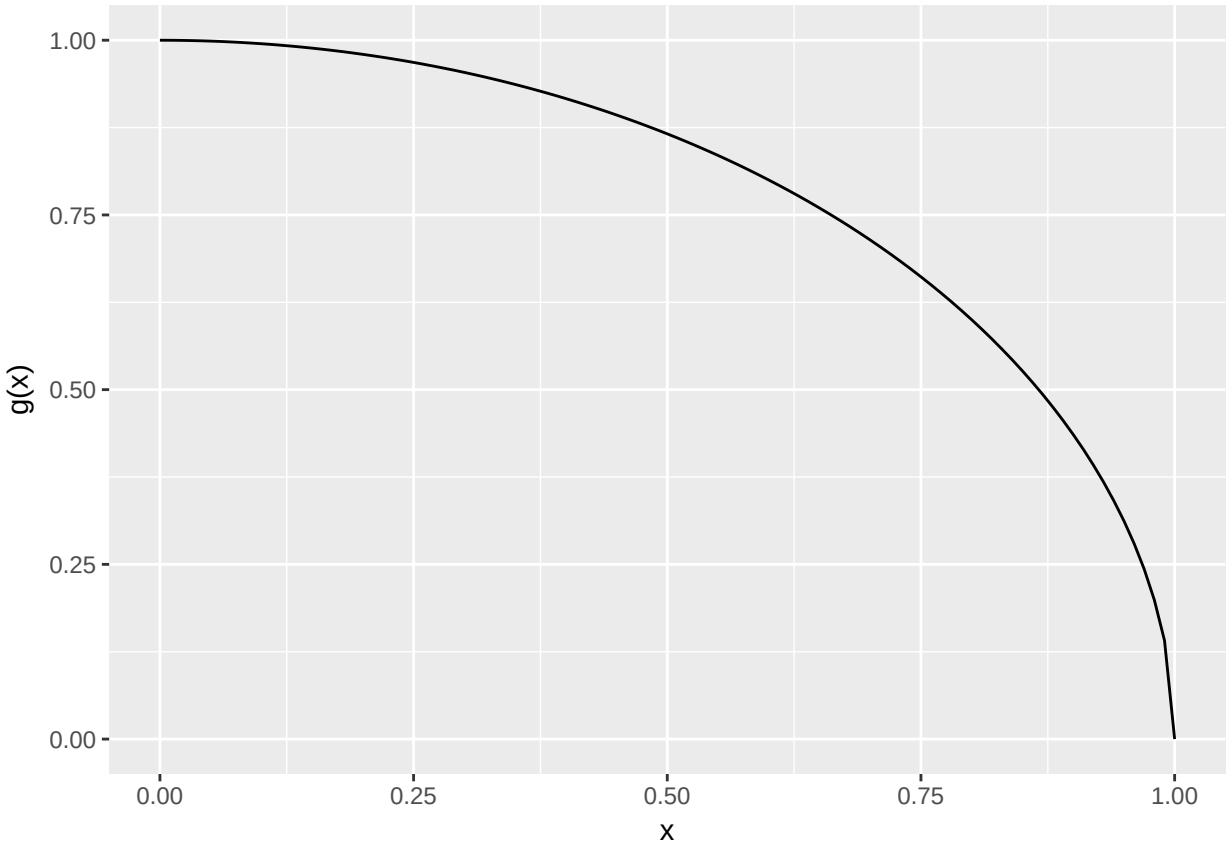
The goal of this exercise is to estiamte the mathematical constant π using MOnte Carlo Integration. Consider the function

$$g(x) = \begin{cases} \sqrt{1-x^2} & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

The above function traces out a quartile circle over the interval $[0, 1]$. Perform the following tasks:

(14) Run the following code:

```
g <- function(x) {
  return(sqrt(1-x^2))
}
library(ggplot2)
ggplot() +
  geom_line(aes(x=seq(0,1,.01),y=g(seq(0,1,.01))))+
  labs(x="x",y="g(x)")
```



(15) Identify the true area under the curve $g(x)$ by using simple geometric formulas.

$$\frac{\pi r^2}{4} = \frac{\pi * 1^2}{4} = \frac{\pi}{4}$$

(16) Using Monte Carlo Integration, approximate the mathematical constant π within a 1/1000 of the true value. When performing this simulation, make sure to choose a probability density function that has support over the unit interval, i.e. uniform(0,1) or beta(α, β)

```
X <- runif(1000^2)
pi.est <- mean(4*sqrt(1-X^2))
pi.est
```

```
## [1] 3.142523
```

```
abs(pi - pi.est) # Small differences
```

```
## [1] 0.0009306794
```

13 Homework 6

13.1 1. Observations

```
n <- 100
p <- 10
s <- 3
set.seed(0)
x <- matrix(rnorm(n * p), n, p)
b <- c(-0.7, 0.7, 1, rep(0, p - s))
y <- x %*% b + rt(n, df = 2)

# Corr:
cors <- apply(x, 2, cor, y)
cors
```

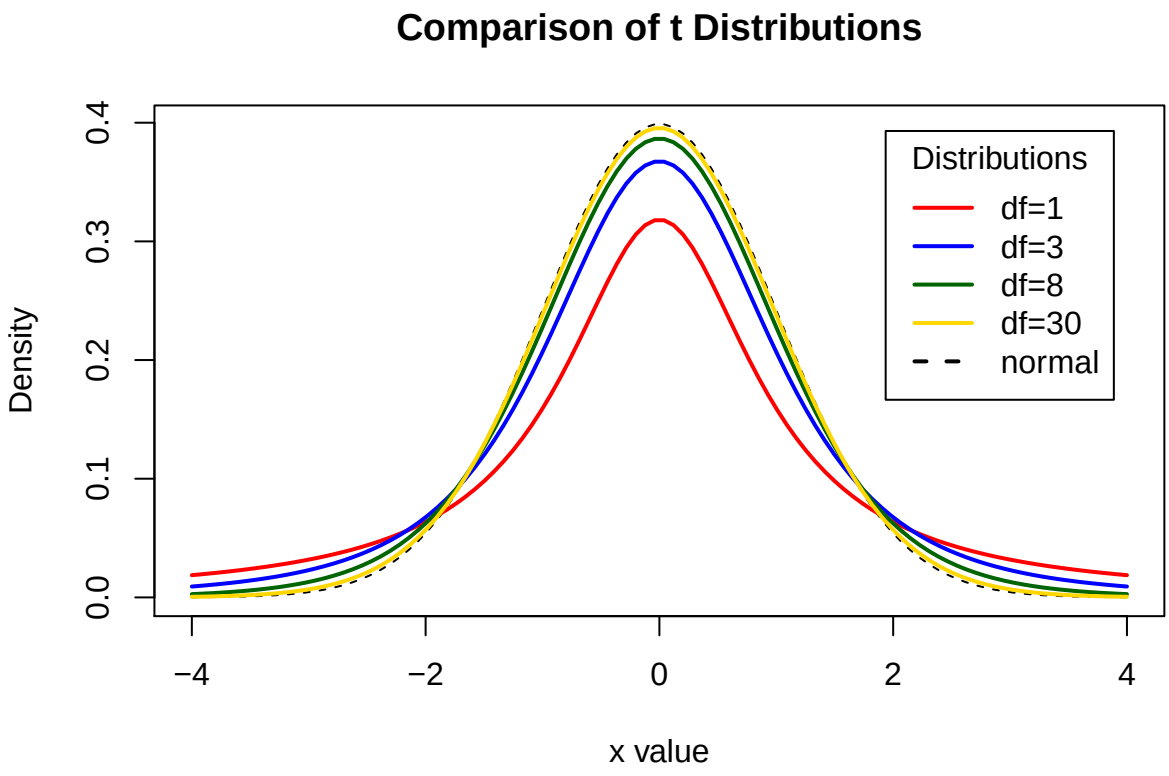
```
##  [1] -0.2526434175  0.1239284685  0.1673840288 -0.2522804417 -0.0371161818
##  [6]  0.1561141420 -0.1175268150 -0.0899681839 -0.0002104895  0.0506851086
```

```
order(abs(cors), decreasing = TRUE)
```

```
##  [1]  1  4  3  6  2  7  8 10  5  9
```

13.2 2. Plot t-distribution

```
# Display the Student's t distributions with various
# degrees of freedom and compare to the normal distribution

x <- seq(-4, 4, length=100)
hx <- dnorm(x)

degf <- c(1, 3, 8, 30)
colors <- c("red", "blue", "darkgreen", "gold", "black")
labels <- c("df=1", "df=3", "df=8", "df=30", "normal")

plot(x, hx, type="l", lty=2, xlab="x value",
  ylab="Density", main="Comparison of t Distributions")

for (i in 1:4){
  lines(x, dt(x,degf[i]), lwd=2, col=colors[i])
}

legend("topright", inset=.05, title="Distributions",
  labels, lwd=2, lty=c(1, 1, 1, 1, 2), col=colors)
```




```
# We can see that the with different degrees of freedom
# we will be able to draw a different shape of
# t-distribution.
```

13.3 3. Huber Loss

```
psi <- function(r, c = 1) {
  return(
    ifelse(
      r^2 > c^2, 2*c*abs(r) - c^2, r^2)
  )
}
```

Write a function called 'huber.loss()' that takes in as an argument a coefficient vector 'beta', and returns the sum of 'psi()' applied to the residuals.

```
huber.loss <- function(beta) {
  sum(psi(y - x %*% beta))
}
```

13.4 4. Function grad.descent()

```
library(numDeriv)
grad.descent <- function(f, x0,
```

```
                          max.iter=  200, step.size = 0.05,
                          stopping.deriv = 0.01, ...) {
  n <- length(x0)
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0

  for (k in 2:max.iter) {

    # Calculate the gradient
    grad.cur <- grad(f, xmat[ ,k-1], ...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k <- k-1; break
      }

    # Move in the opposite direction of the grad
    xmat[ ,k] <- xmat[ ,k-1] - step.size * grad.cur
    }

  xmat <- xmat[ ,1:k] # Trim
  return(list(x = xmat[,k], xmat = xmat, k = k))
}
#gd <- grad.descent(huber.loss, x0 = rep(0,p), max.iter=200, step.size=0.001, stopping.deriv=0.1)
#gd$x
```

13.5 5). Using gd, construct obj

```
#obj <- apply(gd$xmat[, 1:gd$k], 2, huber.loss)
#plot(1:gd$k, obj, xlab = "Iteration Number", ylab = "Objective Function Value", type = "l", main = "Ob
```

13.6 6). Rerun gradient descent with step.size = 0.1

```
#gd2 <- grad.descent(huber.loss, x0 = rep(0,p), max.iter=200, step.size=0.1, stopping.deriv=0.1)
#gd2$x
#plot((gd2$k-49):gd2$k, obj[(gd2$k- 49):gd2$k], xlab = "Iteration Number", ylab = "Objective Function V
```

13.7 7) Sparse Gradient Descent

```
sparse.grad.descent <- function(
  f,
  x0,
  max.iter = 200,
  step.size = 0.05,
  stopping.deriv = 0.01, ...) {

  n <- length(x0)
  xmat <- matrix(0, nrow = n, ncol = max.iter)
  xmat[,1] <- x0
```

```
  for (k in 2:max.iter) {

    # Calculate the gradient
    grad.cur <- grad(f, xmat[ ,k-1], ...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k <- k-1; break
      }

    # Move in the opposite direction of the grad and threshold
    update <- xmat[ ,k-1] - step.size * grad.cur
    update[abs(update) < 0.05] <- 0
    xmat[ ,k] <- update
    }

  xmat <- xmat[ ,1:k] # Trim
  return(list(x = xmat[,k], xmat = xmat, k = k))
}
#gd.sparse <- sparse.grad.descent(huber.loss, x0 = rep(0,p), max.iter=200, step.size=0.001, stopping.de
```

13.8 8) MSE

```
mse.loss <- function(beta) {
mean((b - beta)^2)
}
mse.loss(lm0$coef)
```

```
## [1] 30948909
```

```
# mse.loss(gd$x)
# mse.loss(gd.sparse$x)
```

14 Homework 7

Goals: More practice with simulations. Summarizing data using distributions and estimating parameters

The file 'moretti.csv' contains data compiled by the literary scholar Franco Moretti on the history of genres of novels in Britain betwen 1740 and 1900 (Gothic romances, mystery stories, stories, science fiction, etc.). Each record shows the name of the genre, the year it first appeared, and the year it died out.

It has been conjectured that genres tend to appear together in bursts, bunches, or clusters. We want ot know if this is right. We will simulate what we would expect to see if genres really did appear randomly, at a constant rate - a Poisson process. UNder the assumption, the number of genres which appear in a given year should follow a POisson distribution with some mean λ, and every year should be independent of every other.

14.1 Part 1

14.1.1 i. Function poisLogLik

Assume the variables $x_1, x_2, ..., x_n$ are independent and Poisson-distributed with mean λ then the log likelihood function is given by the following:

$$l(\lambda) = \sum_{i=1}^{n} \log\left(\frac{\lambda^{x_i} e^{-\lambda}}{(x_i)!}\right).$$

Write a function poisLoglik, which takes as inputs a single number λ and a vector data and returns the log-likelihood of that parameter value on that data. What should the value be when data = c(1, 0, 0, 1, 1) and $\lambda = 1$?

```
genres <- read.csv("moretti.csv", as.is = TRUE)
head(genres)
```

```
##          Name Begin  End
## 1   Courtship  1740 1820
## 2 Picareseque  1748 1790
## 3    Oriental  1759 1787
## 4  Epistolary  1766 1795
## 5 Sentimental  1768 1790
## 6         Spy  1770 1800
```

```
# Write a function
poisLoglik <- function(lambda, data) {
  return(
    sum(dpois(data, lambda = lambda, log = TRUE))
  )
}
data1 <- c(1, 0, 0, 1, 1)
lambda1 <- 1
poisLoglik(lambda1, data1)
```

```
## [1] -5
```

```
poisLoglik(0, data1)
```

```
## [1] -Inf
```

14.1.2 ii. Function count_new_genres

Write a function 'count_new_genres' which takes in a year, and returns the number of new genres which appeared in that year: 0 if there were no new genres that year, 1 if there was one, 3 if there were three, etc. What should the values be for 1803 and 1850?

```
count_new_genres <- function(year) {
  return(sum(genres$Begin == year))
}
count_new_genres(1803)
```

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

```
count_new_genres(1850)
```

```
## [1] 3
```

14.1.3 iii. Vector new_genres

Createa vector, new_genres, which counts the number of new genres which appeared in each year of the data, from 1740 to 1900. What positions in the vector correspond to the years 1803 and 1850? What should those values be? Is that what your vector 'new_genres' has for those years?

```
years <- seq(1740, 1900)
num.years <- length(years)
new_genres <- rep(NA, num.years)
names(new_genres) <- years
for (i in 1:num.years) {
  new_genres[i] <- count_new_genres(years[i])
}
twoyears <- which(years == 1803 | years == 1850)
twoyears
```

```
## [1]  64 111
```

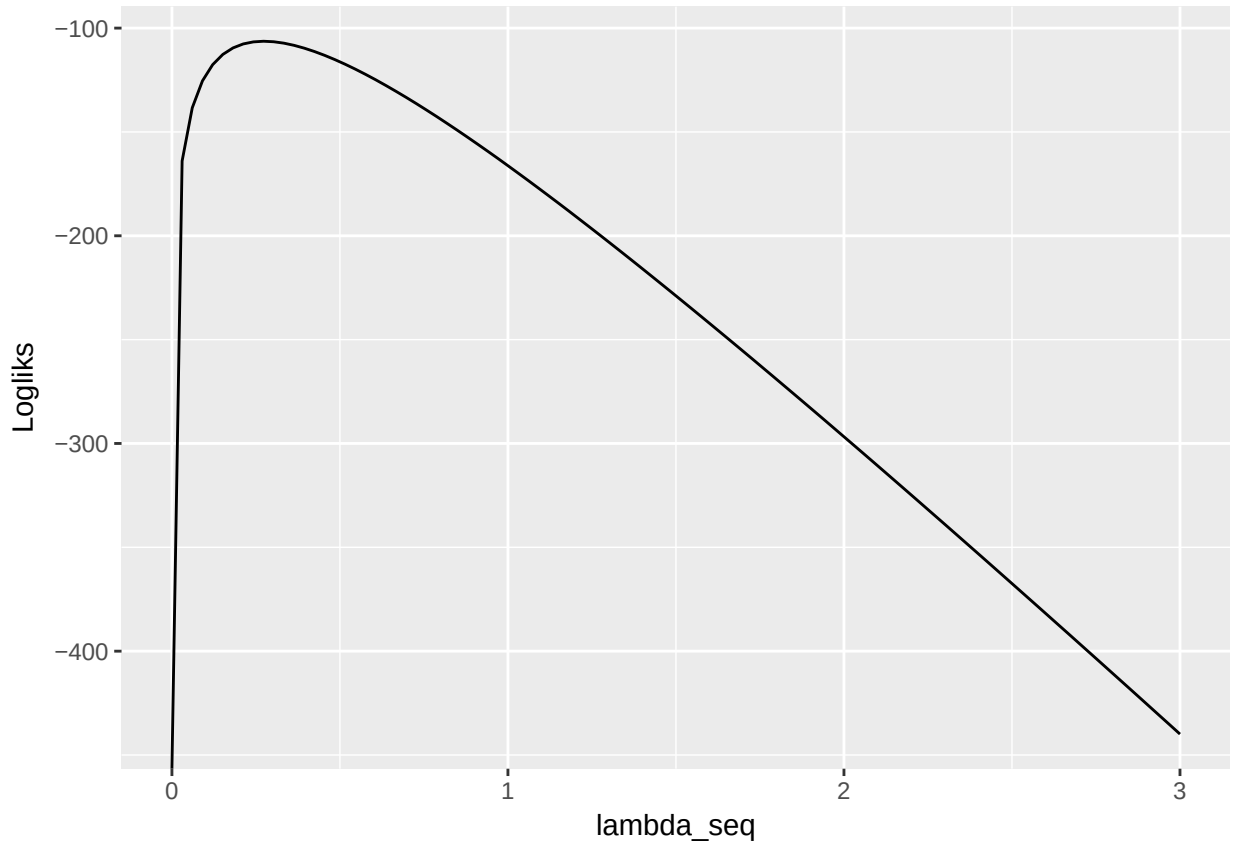
```
new_genres[twoyears]
```

```
## 1803 1850
##    0    3
```

14.1.4 iv. Plot poisLoglik

Plot 'poisLoglik' as a function of λ on the new_genres data. (If the maximum is not at $\lambda = 0.273$, you are doing something wrong.)

```
lambda_seq <- seq(0, 3, length.out = 100)
num.lambdas <- length(lambda_seq)
Logliks <- rep(NA, num.lambdas)
for (i in 1:num.lambdas) {
  Logliks[i] <- poisLoglik(lambda_seq[i], new_genres)
}
library(ggplot2)
ggplot() +
  geom_line(mapping = aes(x = lambda_seq, y = Logliks)) +
  labs(main = "Plot of the Log Likelihood Function",
       ylab = "Log Likelihood", xlab = "Lambda")
```




```
lambda_seq[which.max(Logliks)]
```

```
## [1] 0.2727273
```

14.1.5 v. Use nlm()

Use nlm() to maximize the log likelihood to check the $\lambda = 0.273$ value suggested in the previous question. Hint: you may need to rewrite your function from (i.) with some slight alterations.

```
NegpoisLoglik <- function(lambda, data) {
  return(-sum(dpois(data, lambda = lambda, log = TRUE)))
}
nlm(NegpoisLoglik, 1, new_genres)
```

```
## $minimum
## [1] 106.3349
##
## $estimate
## [1] 0.2732914
##
## $gradient
## [1] 2.984279e-07
##
## $code
## [1] 1
##
```

```
## $iterations
## [1] 11
```

14.1.6 vi. Investige genres

To investigate whether genres appear in bunches or randomly, we look at the spacing between genre births. Create a vector, intergenre_intervals, which shows how many years elapsed between new genres appearing. (If two genres appear in the same year, there should be a 0 in your vector, if three genres appear in the same year your vector shuold have two zeros, and so on. For example if the years that new genres appear are 1835, 1837, 1838, 1838, 1838 your vector should be 2,1,0,0.) What is the mean of the time intervals between genre appearances? The standard deviation? The ratio if the standard deviation to the mean, called the coefficient of variation? Hint: The diff() function might help you here.

```
intergenre_intervals <- diff(sort(genres$Begin))
mean(intergenre_intervals)
```

```
## [1] 3.44186
```

```
sd(intergenre_intervals)
```

```
## [1] 3.705224
```

```
moretti.coef <- sd(intergenre_intervals)/mean(intergenre_intervals)
```

14.1.7 vii. Coefficient of Variation

For a Poisson process, the coeffcient of variation is expected to be around 1. However, that calculation doesn't account for the way Moretti's dates are rounded to the nearest year, or tell us how much the coeffcient of variation might fluctuate. We will handle both of these by simulation.

a. Write a function which takes a vector of numbers, representing how many new genres appear in each year, and returns the vector of the in- tervals between appearances. Check that your function works by see- ing that when it is given new genres, it returns intergenre intervals.

```
intergenre_calc <- function(new.genres) {
  names(new.genres) <- 1:length(new.genres)
  new.genres <- new.genres[new.genres !=0]
  years <- as.numeric(rep(names(new.genres), new.genres))
  return(diff(sort(years)))
}
intergenre_intervals2 <- intergenre_calc(new_genres)
all(intergenre_intervals == intergenre_intervals2)
```

```
## [1] TRUE
```

b. Write a function to simulate a Poisson process and calculate the coefficient of variation of its inter-appearance intervals. It should take as arguments the number of years to simulate and the mean number of genres per year. It should return a list, one component of which is the vector of inter-appearance intervals, and the other their coefficient of variation. Run it with 161 years and a mean of 0.273; the mean of the intervals should generally be between 3 and 4.

```
Pois.sim <- function(num.years, mean.genres) {
  samples <- rpois(num.years, lambda = mean.genres)
  intergenre_intervals <- intergenre_calc(samples)
  coef.of.var <- sd(intergenre_intervals)/mean(intergenre_intervals)
  return(list(intergenre_intervals = intergenre_intervals, coef.of.var = coef.of.var))
```

```
}
for (i in 1:10) {
  res <- Pois.sim(141, 0.273)
  print(mean(res$intergenre_intervals))
}
```

```
## [1] 3.341463
## [1] 3.878788
## [1] 3.219512
## [1] 3.621622
## [1] 3.435897
## [1] 3.175
## [1] 2.603774
## [1] 3.3
## [1] 2.555556
## [1] 3.742857
```

14.1.8 viii. Simulation 100,000 times

Run your simulation 100,000 times, taking the coefficient of variation (only) from each. (This should take less than two minutes to run.) What fraction of simulations runs have a higher coefficient of variation than Moretti's data?

```
coef.of.var <- rep(NA, n)
for (i in 1:n) {
  res <- Pois.sim(141, 0.273)
  coef.of.var[i] <- res$coef.of.var
}
mean(coef.of.var > moretti.coef)
```

```
## [1] 0.25
```

14.1.9 ix. Explanation

Explain what this does and does not tell you about the conjecture that genres tend to appear together in burst?

The above result tells us that there isn't really any evidence that new genres appear together in bursts. If the appearance of new genres were truly random (not clustered), meaning they appear according to a Poisson process, then around 23 precent of the time we would see results as or more clustered than Moretti's data (when we consider the cefficient of variation as a measure of the amoutn of cluster).

15 Homework 8

Gross domestic product (GDP) is a measure of the total market value of all goods and services produced in a given country in a given year. The percentage growth rate of GDP in year t is

$$100 \times \left(\frac{GDP_{t+1} - GDP_t}{GDP_t} \right) - 100$$

An important claim in economics is that the rate of GDP growth is closely related to the level of government debt, specifically with the ratio of the government's debt to the GDP. The file debt.csv on the class website

contains measurements of GDP growth and of the debt-to-GDP ratio for twenty countries around the world, from the 1940s to 2010. Note that not every country has data for the same years, nad some years in the middle of the period are missing data for some countries but not others.

```
# Check:
debt <- read.csv("debt.csv", as.is = TRUE)
dim(debt)
```

```
## [1] 1171    4
```

```
head(debt)
```

```
##     Country Year    growth     ratio
## 1 Australia 1946 -3.557951 190.41908
## 2 Australia 1947  2.459475 177.32137
## 3 Australia 1948  6.437534 148.92981
## 4 Australia 1949  6.611994 125.82870
## 5 Australia 1950  6.920201 109.80940
## 6 Australia 1951  4.272612  87.09448
```

15.1 1) Average GDP Growth

Calculate the average GDP growth rate for each country (averaging over years). This is a classic spli/apply/combine problem, and you will use ** daply() ** to solve it.

15.1.1 a. Write a function mean.growth()

```
library(plyr)

# Write function:
mean.growth <- function(country.df) {
  return(
    signif(
      mean(country.df$growth),
      3)
    )
}
```

15.1.2 b. Use 'daply()'

```
country.avgs <- daply(debt, .(Country), mean.growth)
country.avgs["Australia"]
```

```
## Australia
##      3.72
```

```
country.avgs["Netherlands"]
```

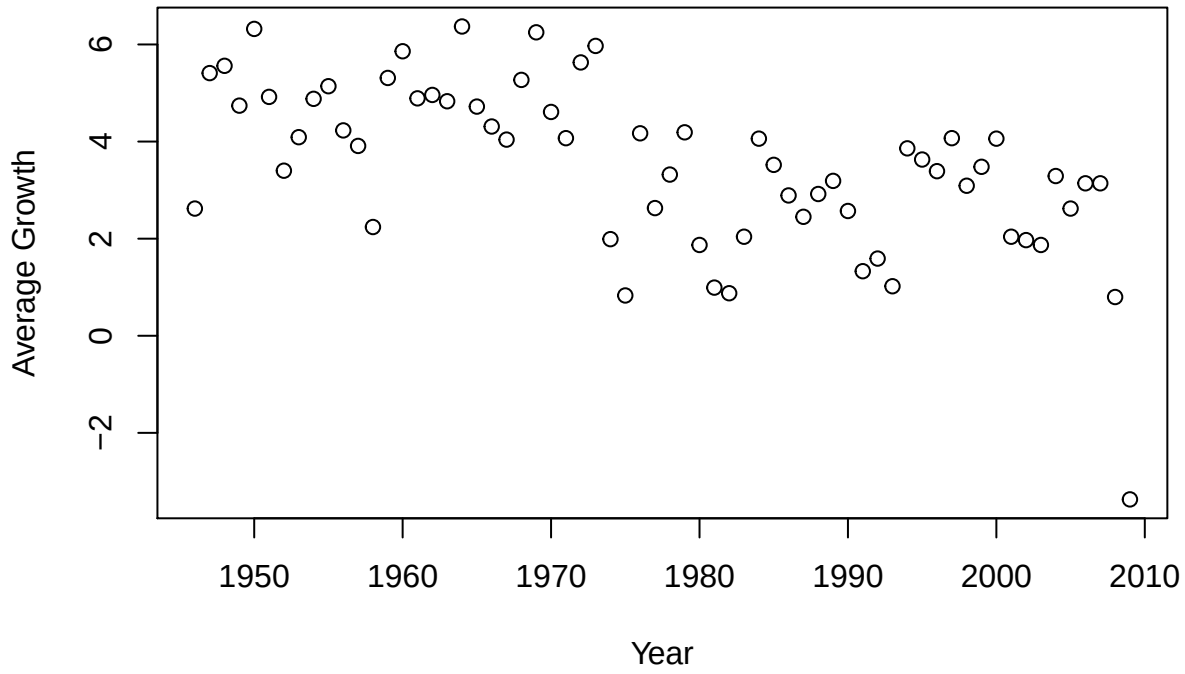
```
## Netherlands
##        3.03
```

15.2 2) Average GDP Growth for each year

```
year.avgs <- daply(debt, .(Year), mean.growth)
year.avgs["1972"]
```

```
## 1972
## 5.63
```

```
plot(names(year.avgs),
          year.avgs,
          xlab = "Year",
          ylab = "Average Growth"
     )
```

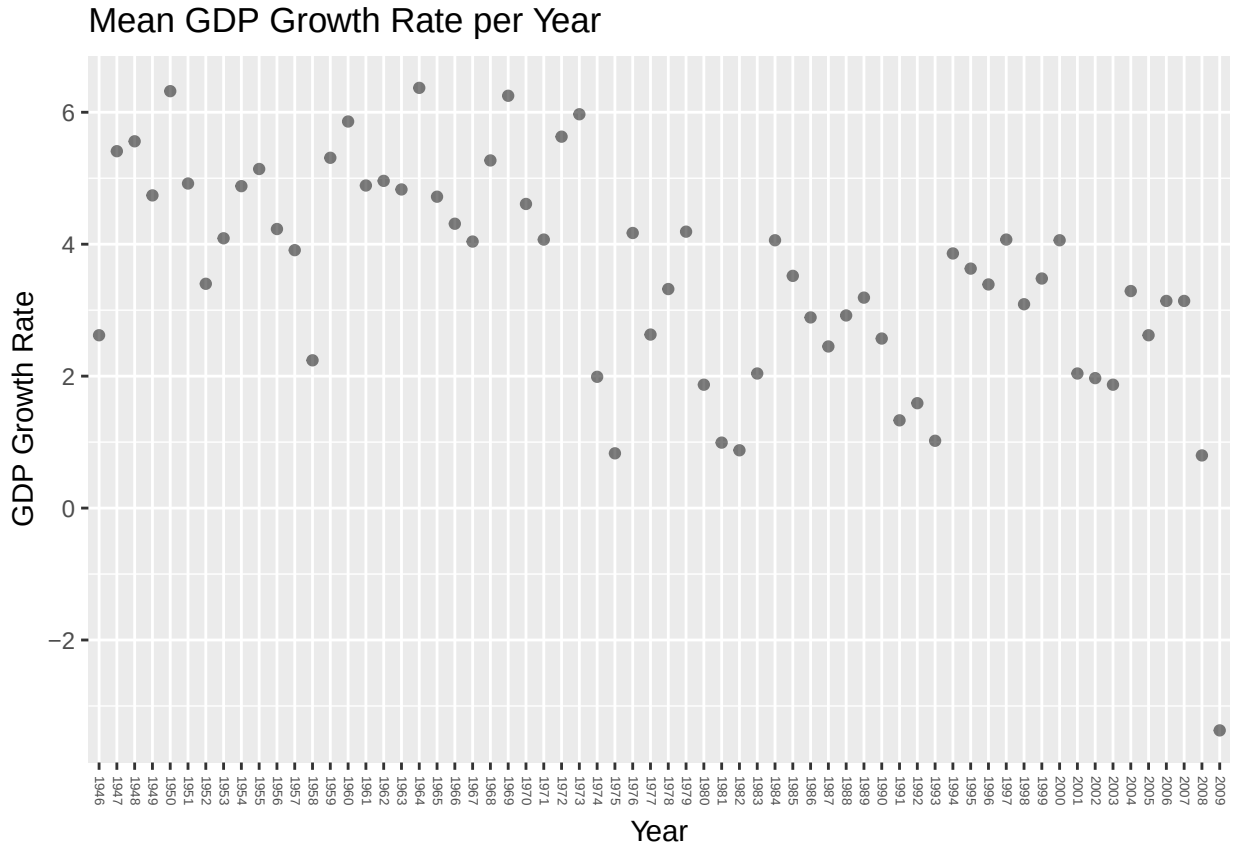



```
# GG Plot:
yg = daply(debt,.(Year),mean.growth)
signif(yg,3)
```

```
##   1946   1947   1948   1949   1950   1951   1952   1953   1954   1955
##  2.620  5.410  5.560  4.740  6.320  4.920  3.400  4.090  4.880  5.140
##   1956   1957   1958   1959   1960   1961   1962   1963   1964   1965
##  4.230  3.910  2.240  5.310  5.860  4.890  4.960  4.830  6.370  4.720
##   1966   1967   1968   1969   1970   1971   1972   1973   1974   1975
##  4.310  4.040  5.270  6.250  4.610  4.070  5.630  5.970  1.990  0.830
##   1976   1977   1978   1979   1980   1981   1982   1983   1984   1985
##  4.170  2.630  3.320  4.190  1.870  0.992  0.876  2.040  4.060  3.520
##   1986   1987   1988   1989   1990   1991   1992   1993   1994   1995
```

```
##  2.890  2.450  2.920  3.190  2.570  1.330  1.590  1.020  3.860  3.630
##   1996   1997   1998   1999   2000   2001   2002   2003   2004   2005
##  3.390  4.070  3.090  3.480  4.060  2.040  1.970  1.870  3.290  2.620
##   2006   2007   2008   2009
##  3.140  3.140  0.798 -3.370
```

```
yg = as.data.frame(yg)
yg$Year = rownames(yg)
##
library(ggplot2)
ggplot(yg) +
  geom_point(mapping = aes(x=Year, y = yg), alpha = 0.5) +
  labs(title = "Mean GDP Growth Rate per Year", x = "Year", y = "GDP Growth Rate") +
  theme(axis.text.x = element_text(angle = 270, vjust=.4, size = 5))
```



15.3 3) Correlation coefficient

15.3.1 a. Calculate Corr between GDP and Debt

```
signif(cor(debt$growth, debt$ratio), 3)
```

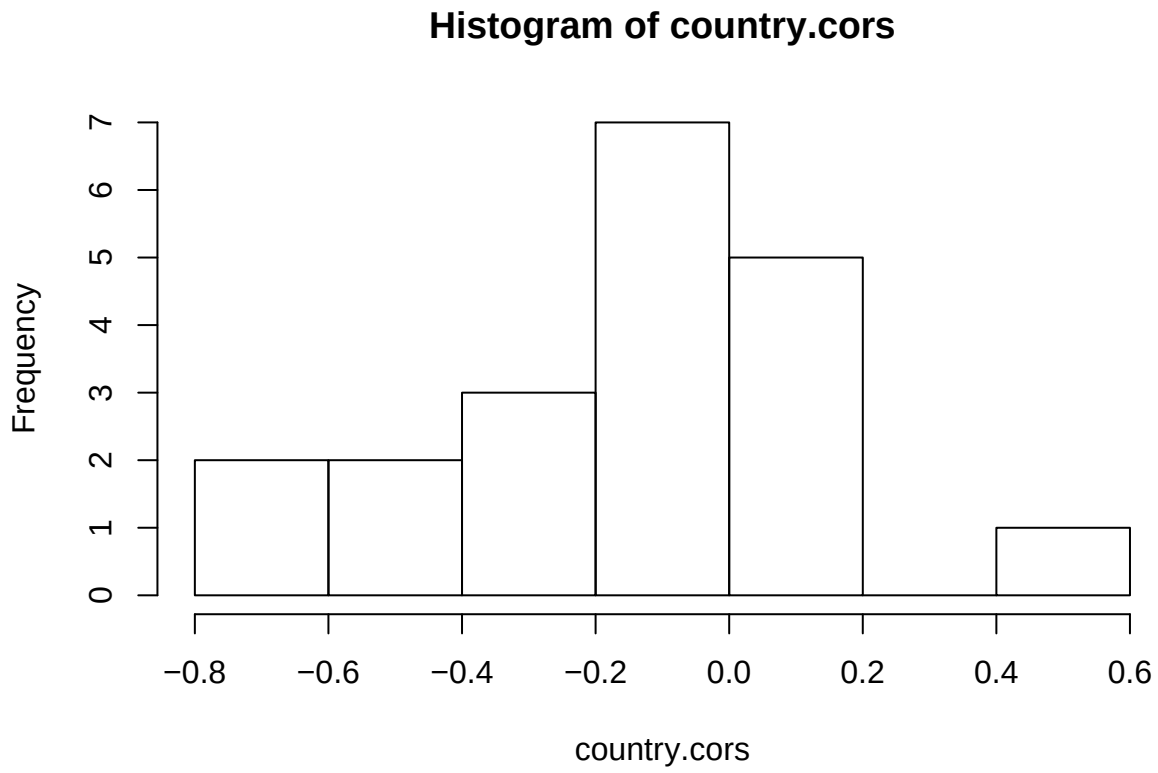
```
## [1] -0.199
```

15.3.2 b. Correlation for each country

```
cor.calc <- function(country.df) {
  return(signif(cor(country.df$growth, country.df$ratio), 3))
}
country.cors <- daply(debt, .(Country), cor.calc)
mean(country.cors)
```

```
## [1] -0.177766
```

```
hist(country.cors)
```

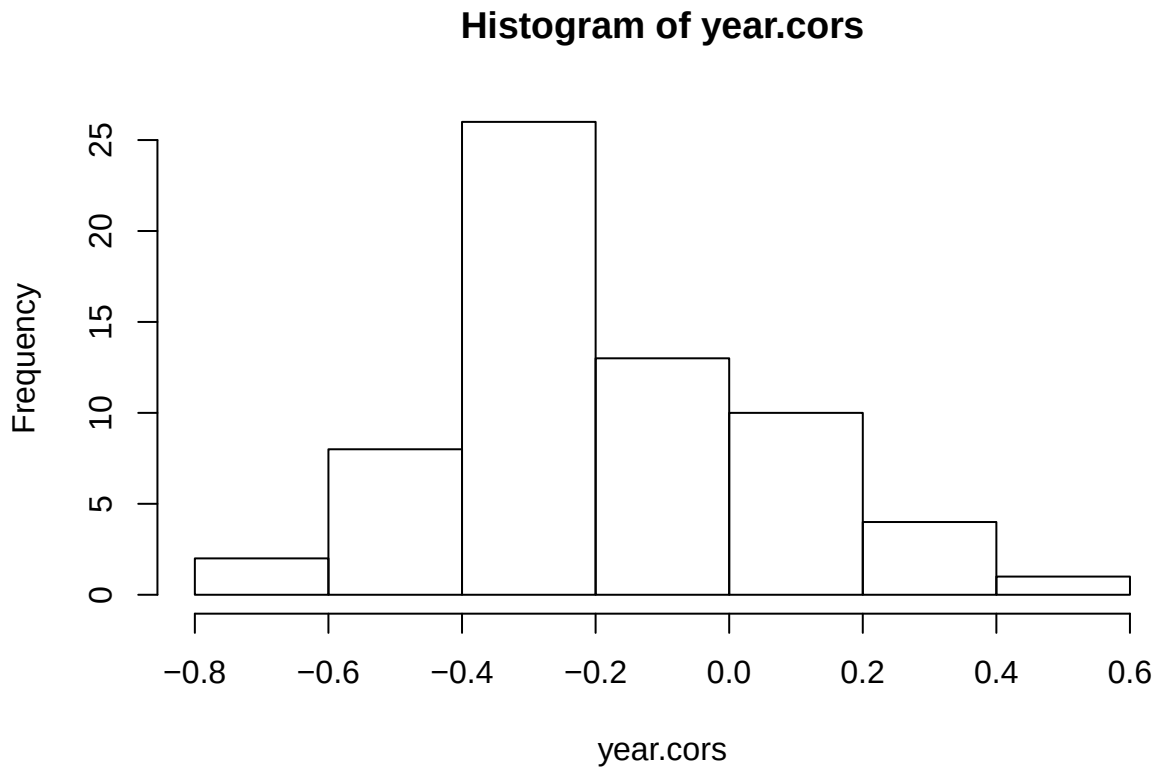


15.3.3 c. Correlation Coefficients

```
year.cors <- daply(debt, .(Year), cor.calc)
mean(year.cors)
```

```
## [1] -0.190525
```

```
hist(year.cors)
```



15.3.4 d. Sort

```
sort(country.cors)
```

```
##       Japan       Italy     Germany      France    Portugal          US
##   -0.702000   -0.645000   -0.576000   -0.502000   -0.352000   -0.341000
##     Austria Netherlands     Belgium     Denmark      Sweden     Ireland
##   -0.253000   -0.199000   -0.192000   -0.168000   -0.161000   -0.140000
##          UK      Greece     Finland   Australia      Canada       Spain
##   -0.137000   -0.093500    0.000581    0.025200    0.075000    0.081400
## New Zealand      Norway
##    0.161000    0.563000
```

```
sort(year.cors)
```

```
##     1957     1946     1961     1970     1960     1956     1958     1985
## -0.75500 -0.62000 -0.53900 -0.51200 -0.50400 -0.45800 -0.45400 -0.44900
##     1979     1951     1962     1993     1983     1964     1986     1996
## -0.42900 -0.41600 -0.38300 -0.37200 -0.36200 -0.36100 -0.35800 -0.35700
##     2002     2007     1948     2005     1965     1966     1959     1967
## -0.34900 -0.34400 -0.34000 -0.31400 -0.31100 -0.31100 -0.28500 -0.27800
##     1952     1954     1947     1998     1999     1969     2001     1955
## -0.27700 -0.27500 -0.27400 -0.26500 -0.25800 -0.25000 -0.23800 -0.22700
```

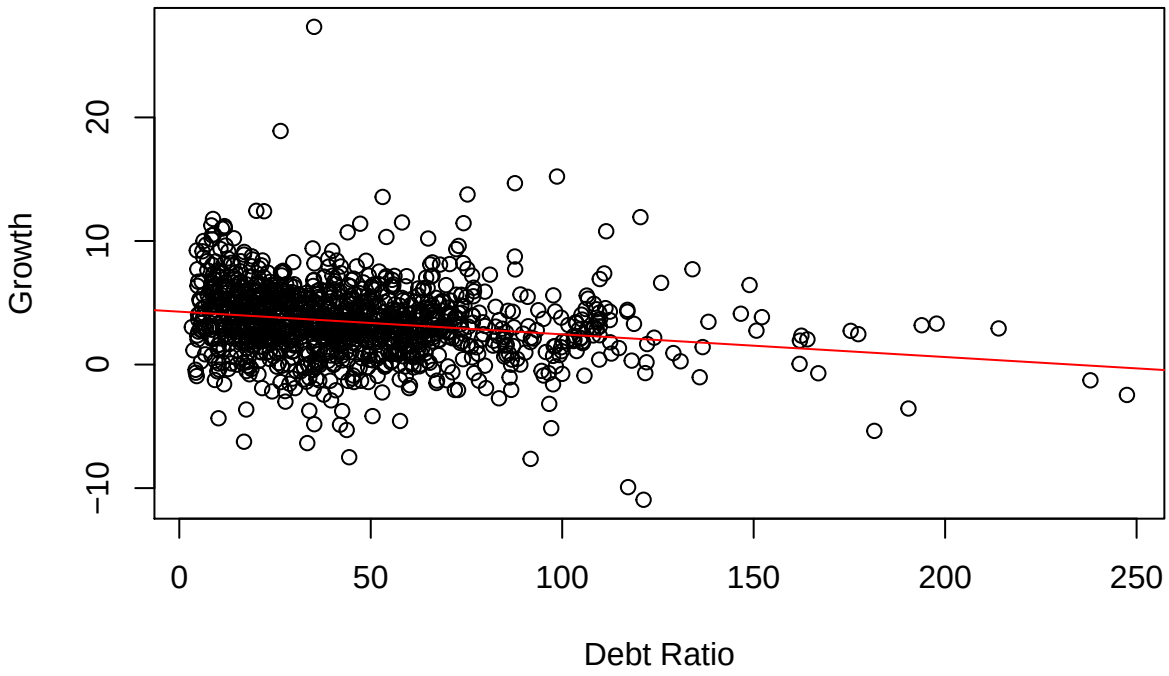
```
##      1994      1953      2009      1949      1972      2006      1968      1976
## -0.22400 -0.20500 -0.20500 -0.20000 -0.19600 -0.19600 -0.18100 -0.17100
##      2004      1984      2000      1980      1997      2008      1987      2003
## -0.17100 -0.15600 -0.13400 -0.12700 -0.11100 -0.09450 -0.06890 -0.06790
##      1992      1971      1981      1950      1995      1989      1988      1973
## -0.00222  0.00872  0.03040  0.03980  0.05190  0.06640  0.07970  0.11400
##      1963      1990      1977      1991      1982      1974      1975      1978
##  0.12800  0.15600  0.16400  0.20200  0.23900  0.26000  0.27100  0.43100
```

15.4 4) Fit Linear Model

```
plot(debt$ratio, debt$growth,
     xlab = "Debt Ratio",
     ylab = "Growth")
lm0 <- lm(debt$growth ~ debt$ratio)
lm0$coef
```

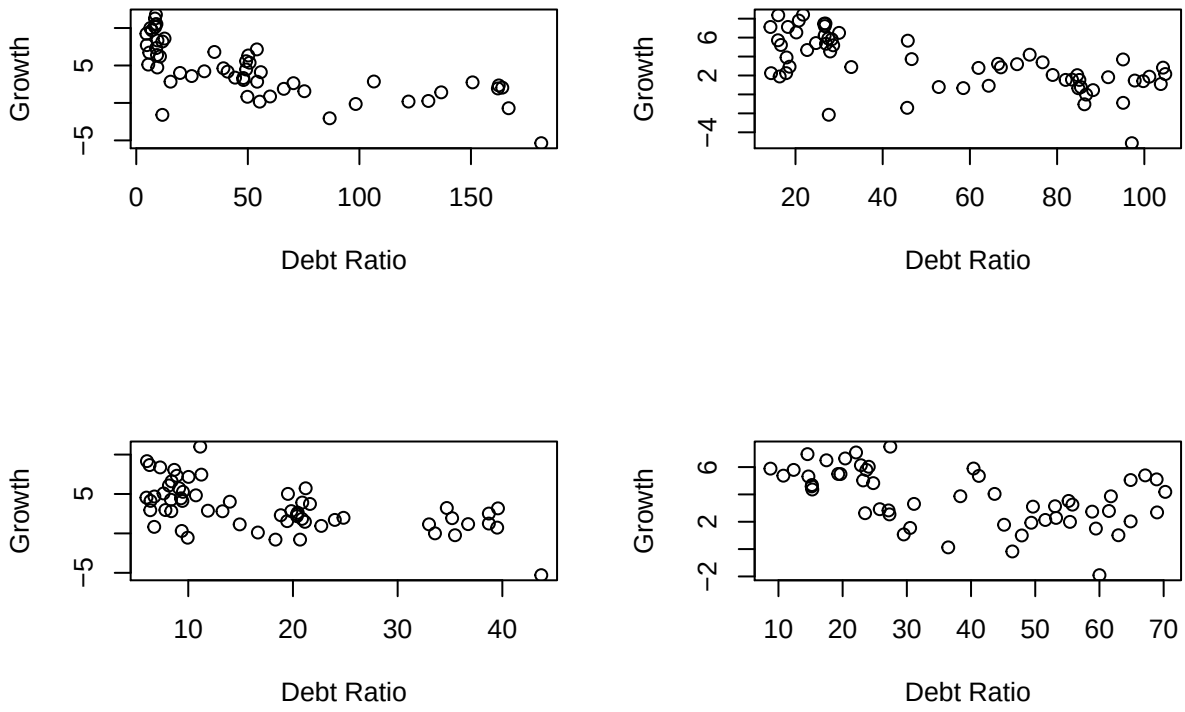
```
## (Intercept)  debt$ratio
##  4.27929049 -0.01835518
```

```
abline(lm0, col = "red")
```



15.5 5) Plot Among Countries

```
par(mfrow = c(2, 2))
four.countries <- names(sort(country.cors))[1:4]
for (i in 1:4) {
  plot(debt$ratio[debt$Country == four.countries[i]],
       debt$growth[debt$Country == four.countries[i]],
       xlab = "Debt Ratio",
       ylab = "Growth"
  )
}
```



15.6 6) Future Growth

15.6.1 a. Create new data frame

```
debt.fr <- debt[debt$Country == "France", ]
dim(debt.fr)
```

```
## [1] 54  4
```

15.6.2 b. Create 'next.growth'

```
next.growth <- function(year, country.df) {
  if(any(country.df$Year == (year + 1))) {
    return(country.df$growth[country.df$Year == (year + 1)])
} else {
  return(NA)
  }
}
debt.fr$next.growth <- sapply(debt.fr$Year, next.growth, debt.fr)
debt.fr$next.growth[debt.fr$Year == 1971]
```

```
## [1] 5.885827
```

```
debt.fr$next.growth[debt.fr$Year == 1972]
```

```
## [1] NA
```

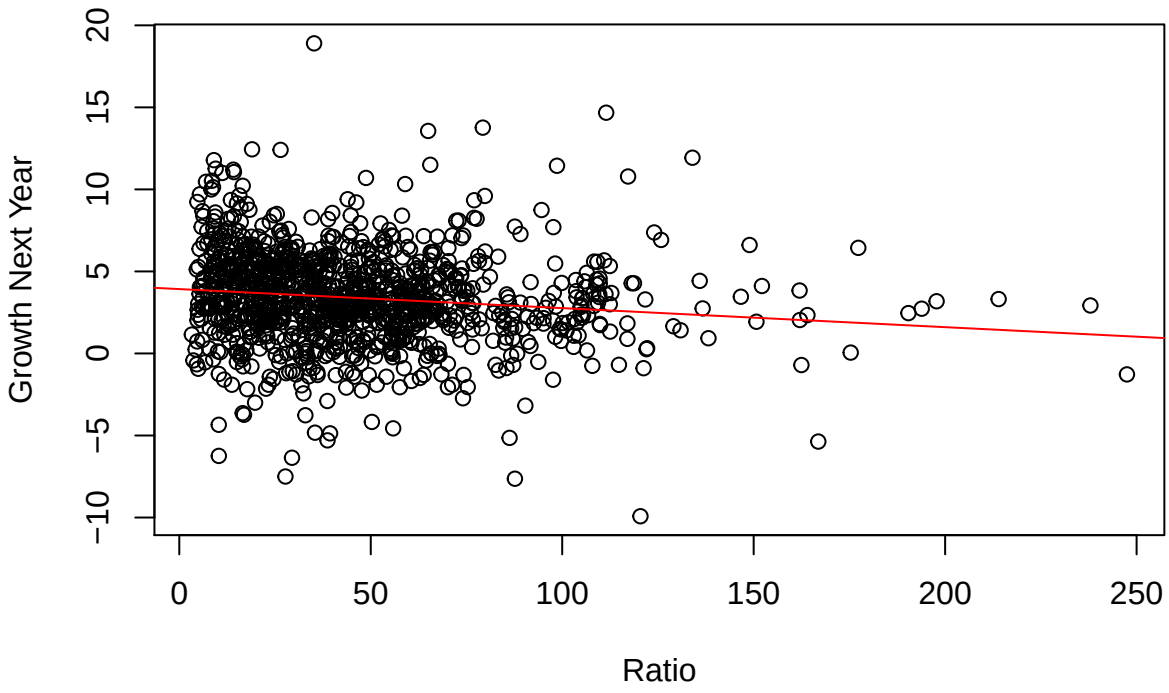
15.7 7) Add next.growth to data

```
next.growth.all <- function(country.df) {
  country.df$next.growth <- sapply(country.df$Year, next.growth, country.df)
  return(country.df)
}
debt <- ddply(debt, .(Country), next.growth.all)
debt$next.growth[debt$Country == "France" & debt$Year == 2009]
```

```
## [1] NA
```

15.8 8) Make Scallter-plot

```
plot(
  debt$ratio,
  debt$next.growth,
  xlab = "Ratio", ylab = "Growth Next Year"
  )
lm1 <- lm(debt$next.growth ~ debt$ratio)
abline(lm1, col = "red")
```




```
lm0$coef
```

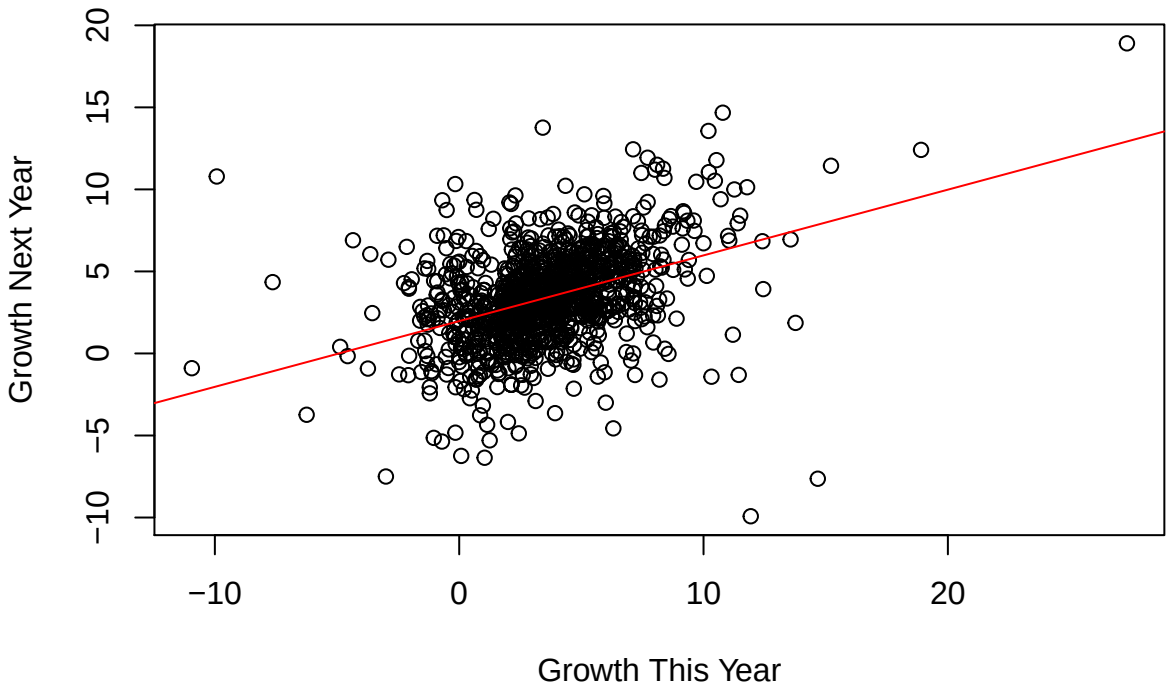
```
## (Intercept)  debt$ratio
##  4.27929049 -0.01835518
```

```
lm1$coef
```

```
## (Intercept)  debt$ratio
##  3.92472155 -0.01160842
```

15.9 9) Scatter-plot GDP

```
plot(
  debt$growth,
  debt$next.growth,
  xlab = "Growth This Year", ylab = "Growth Next Year")
lm2 <- lm(debt$next.growth ~ debt$growth)
abline(lm2, col = "red")
```




```
lm2$coef
```

```
## (Intercept) debt$growth
##   1.9710648   0.4006517
```