Homework 4

Yixin Chen

2020-10-16

Problem 2: Using the dual nature to our advantage

```r
set.seed(1256)
theta <- as.matrix(c(1,2),nrow=2)
X <- cbind(1,rep(1:10,10))
h <- X%*%theta+rnorm(100,0,0.2)

alpha<-0.001 # learning rate
tol<-1e-6 # tolerance
m<-nrow(X)
# starting values for Beta
beta0.0<-0
beta1.0<-0
h0<-function(x){beta0.0+beta1.0*x}
beta0.1<-beta0.0-alpha*(1/m)*sum(h0(X[,2])-h)
beta1.1<-beta1.0-alpha*(1/m)*sum((h0(X[,2])-h)*X[,2])

while( (abs(beta0.0-beta0.1) > tol) || (abs(beta1.0-beta1.1) > tol) ) {
  beta0.0<-beta0.1
  beta1.0<-beta1.1
  beta0.1<-beta0.0-alpha*(1/m)*sum(h0(X[,2])-h)
  beta1.1<-beta1.0-alpha*(1/m)*sum((h0(X[,2])-h)*X[,2])
}

cat("beta0:",beta0.1,"beta1:",beta1.1,"\n","tolerance:",tol,"learning rate/stepsize:",alpha)
```

```
## beta0: 0.9648095 beta1: 2.002247
##  tolerance: 1e-06 learning rate/stepsize: 0.001
```

```r
fit<-lm(h~0+X)
fit$coefficients
```

```
##        X1        X2
## 0.9695707 2.0015630
```

The result of gradient decent is very close to the result of `lm~0+X`.

Problem 3

In the first part of this problem, I tried with $\alpha = 1 \times 10^{-7}$ and tolerance= 1×10^{-9}, but it took me more than 30 seconds when I was using only 1 starting value. As you can see as following:

```r
system.time(GradDesc(c(0,0),h,X,1e-7,1e-9))
```

```
##    user  system elapsed
##  26.560   6.354  33.171
```

As a result, when I apply this function to the 1000×1000 grid, my laptop kept running for 24 hours and still didn't get the result. Therefore, I had to change my α and tolerance to 1×10^{-3} and 1×10^{-7}, otherwise I wouldn't be able to submit my homework. And it's not because of the issue of the code. Because I tried the same code on another student's laptop, it took only 5 seconds for 1 starting value. My laptop is just too slow to do this project.

Besides, after many trials, I found $\alpha = 1 \times 10^{-3}$ is a very good stepsize for this problem. The estimated β values are very close to the truth and it's not time consuming.

Part a.

Defining the grid and the Gradient Descent function:

```r
beta0.val<-seq(0,2,length.out = 100) # possible start values for beta_0
beta1.val<-seq(1,3,length.out = 100) # possible start values for beta_1
# a grid containing all possible combinations for beta_0 and beta_1
Beta.grid<-expand.grid(beta0.val,beta1.val)
# A function to implement gradient descent
GradDesc<-function(beta.start,h,X,alpha,tol){
  m<-nrow(X)
  # beta old values
  beta.old<-as.numeric(beta.start)
  # beta new values
  beta.new<-c(NA,NA)
  h0 <- X %*% beta.old
  beta.new[1]<-beta.old[1]-alpha*(1/m)*sum(h0-h)
  beta.new[2]<-beta.old[2]-alpha*(1/m)*sum((h0-h)*X[,2])
  # iteration times
  iter<-1
  while( (abs(beta.old[1]-beta.new[1]) > tol) || (abs(beta.old[2]-beta.new[2]) > tol) ) {
    iter<-iter+1
    if (iter > 5e+6) break
    beta.old<-beta.new
    h0 <- X %*% beta.old
    beta.new[1]<-beta.old[1]-alpha*(1/m)*sum(h0-h)
    beta.new[2]<-beta.old[2]-alpha*(1/m)*sum( (h0-h)*X[,2] )
  }
  return(c(iter,beta.new,beta.start))
}
```

```r
beta0.val<-seq(0,2,length.out = 100) # possible start values for beta_0
beta1.val<-seq(1,3,length.out = 100) # possible start values for beta_1
```

```r
# a grid containing all possible combinations for beta_0 and beta_1
Beta.grid<-expand.grid(beta0.val,beta1.val)
# parallel computing
core <- detectCores()-1
cl<-makeCluster(core)
clusterExport(cl,"h")
clusterExport(cl,"X")
system.time(result<-parApply(cl, Beta.grid, 1, GradDesc,h,X,1e-3,1e-7))
```

```
##    user  system elapsed 
##   0.052   0.004 975.582
```

```r
stopCluster(cl)
rownames(result)<-c("iteration","beta_0_hat","beta_1_hat","beta_0_start","beta_1_start")
# minimum number of iteration:
min(result[1,])
```

```
## [1] 316
```

```r
# starting value lead to minimum number of iteration:
result[4:5,which.min(result[1,])]
```

```
## beta_0_start beta_1_start 
##    0.8484848    1.1616162
```

```r
# maximum number of iteration:
max(result[1,])
```

```
## [1] 37096
```

```r
# starting value lead to maximum number of iteration:
result[4:5,which.max(result[1,])]
```

```
## beta_0_start beta_1_start 
##            2            1
```

```r
# mean of beta_0
mean(result[2,])
```

```
## [1] 0.969585
```

```r
# standard deviation of beta_0
sd(result[2,])
```

```
## [1] 0.00047588
```

```r
# mean of beta_1
mean(result[3,])
```

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

```r
# standard deviation of beta_1
sd(result[3,])
```

```
## [1] 6.835527e-05
```

Part b.

We know the true value for β are: $\beta_0 = 1$, $\beta_1 = 2$. Let's try with starting values that are close to the truth:

```r
system.time(start.true<-GradDesc(c(1.001,2.001),h,X,1e-7,1e-9))
```

```
##    user  system elapsed
##   3.620   0.816   4.478
```

```r
start.true
```

```
## [1] 6.704540e+05 1.000052e+00 1.997468e+00 1.001000e+00 2.001000e+00
```

We can see that the number of iterations is 6.70454×10^5. One may consider changing the stopping rule to `if (iter > 1e+6) break`, i.e, break from the loop when iteration times is greater than 1×10^6. However, this is not a good way to run this algorithm. Because we barely know the true result or true parameter values in practice. If we know some knowledge about the truth, there are many alternatives that are not as computationally expensive as the Gradient Descent algorithm, such as newton's method.

Part c. What are your thoughts on this algorithm?

This algorithm is very computationally expensive for large datasets. Sometimes it cause you to zig zag back and forth. I wouldn't use this algorithm to run regression just in order to get an parameter estimate.

Problem 4: Inverting matrices

Here is what I'll do in R:

```r
solve(t(X)%*%X) %*% t(X) %*% h
```

```
##           [,1]
## [1,] 0.9695707
## [2,] 2.0015630
```

This is obtained by minimizing the sum of square function:

$$Q(\beta) = (y - X\beta)'(y - X\beta)$$

If we take derivative on both sides with respect to β, we have

$$\frac{\partial Q}{\partial \beta} = -2X'y + 2X'X\beta$$

This is known as the Normal Equation. Setting this to zero, we get the least square solution:

$$\beta = (X'X)^{-1}X'y$$

Problem 5: Need for speed challenge

```r
set.seed(12456)

G <- matrix(sample(c(0,0.5,1),size=16000,replace=T),ncol=10)
R <- cor(G) # R: 10 * 10 correlation matrix of G
C <- kronecker(R, diag(1600)) # C is a 16000 * 16000 block diagonal matrix
id <- sample(1:16000,size=932,replace=F)
q <- sample(c(0,0.5,1),size=15068,replace=T) # vector of length 15068
A <- C[id, -id] # matrix of dimension 932 * 15068
B <- C[-id, -id] # matrix of dimension 15068 * 15068
p <- runif(932,0,1)
r <- runif(15068,0,1)
C<-NULL #save some memory space
```

Part a.

```r
object.size(A)
```

```
## 112347224 bytes
```

```r
object.size(B)
```

```
## 1816357208 bytes
```

```r
system.time(y<-p+A%*%solve(B)%*%(q-r))
```

```
##    user  system elapsed
## 964.678   9.327 983.650
```

Part b.

I would compute $B^{-1}(q-r)$ first. Because if we compute AB^{-1} first, then we need to store a 932×15068 matrix. If we compute $B^{-1}(q-r)$ first, we only need to store a 15068×1 vector.

There is a simplification we can make. Notice that B is just an identity matrix. Thus, multiplying by B^{-1} is just mutiplying by I. The expression comes down to $y = p + A(q-r)$.

Part c.

```r
system.time(y<-p%*%A%*%(q-r))
```

```
##    user  system elapsed
##   0.643   0.058   0.702
```

Use ANY means (ANY package, ANY trick, etc) necessary to compute the above, fast. Wrap your code in "system.time({})", everything you do past assignment "C <- NULL".

Problem 3

a.

```r
# a function that computes the proportion of successes in a vector
bin.p<-function(x){
  p<-mean(x)
  return(p)
}
```

b. Create a matrix to simulate 10 flips of a coin with varying degrees of "fairness" (columns = probability) as follows:

```r
    set.seed(12345)
    P4b_data <- matrix(rbinom(10, 1, prob = (31:40)/100), nrow = 10, ncol = 10, byrow = FALSE)
```

c.

```r
col.p<-apply(P4b_data, 2, bin.p)
row.p<-apply(P4b_data, 1, bin.p)
col.p<-data.frame(t(col.p))
row.p<-data.frame(t(row.p))
colnames(col.p)<-c("p=0.31","0.32","0.33","0.34","0.35","0.36","0.37","0.38","0.39","0.40")
row.names(col.p)<-c("By Column")
colnames(row.p)<-c(seq(1,10,1))
row.names(row.p)<-c("By Row")
knitr::kable(col.p, booktabs = T) %>% kable_styling(position = "center")
```

	p=0.31	0.32	0.33	0.34	0.35	0.36	0.37	0.38	0.39	0.40
By Column	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6

```r
knitr::kable(row.p, booktabs = T) %>% kable_styling(position = "center")
```

	1	2	3	4	5	6	7	8	9	10
By Row	1	1	1	1	0	0	0	0	1	1

We can see that the proportion of success of each column is the same with others. The proportion of success for the rows is either 1 or 0, indicating that the elements in each row are the same. This implies that the columns are just the same binomial samples, not independent with each other.

d.

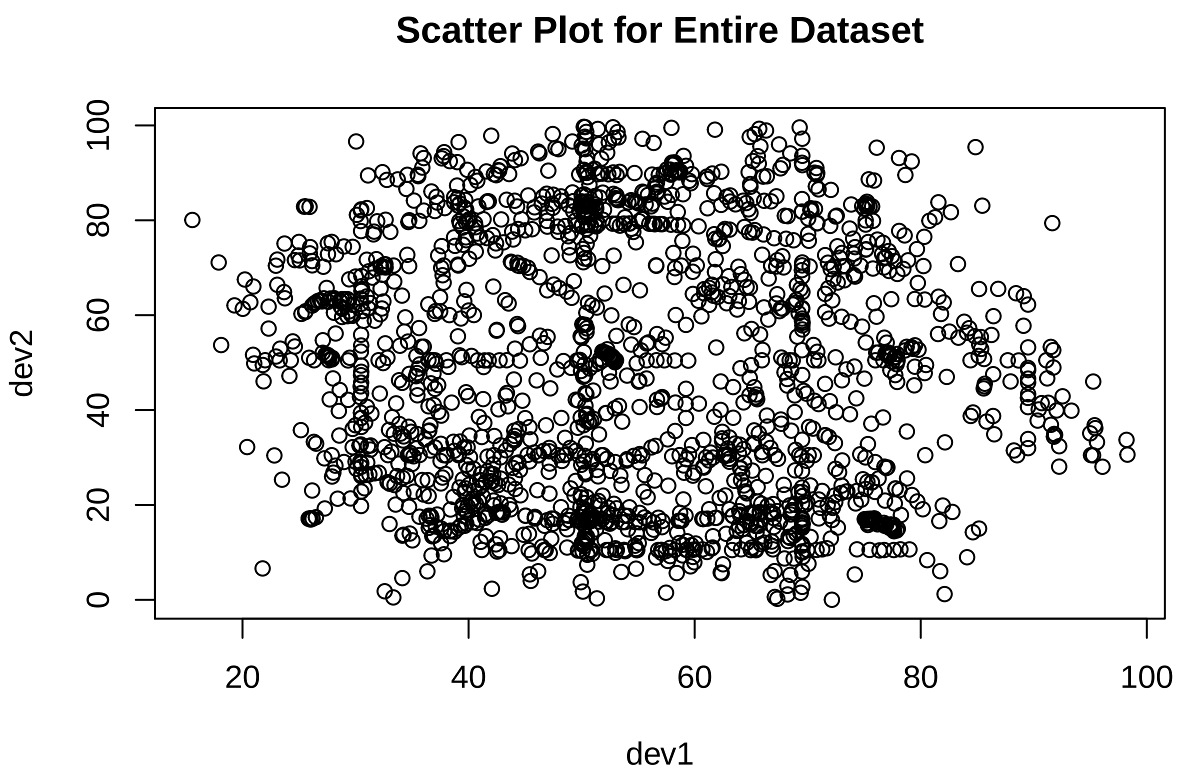
```r
coinflip10<-function(p){
  v<-rbinom(10,1,p)
  return(v)
}
p<-as.matrix((31:40)/100)
prob.matrix<-apply(p, 1, coinflip10)
col.p<-apply(prob.matrix, 2, bin.p)
col.p<-data.frame(t(col.p))
colnames(col.p)<-c("p=0.31","0.32","0.33","0.34","0.35","0.36","0.37","0.38","0.39","0.40")
row.names(col.p)<-"proportion of success"
kable(col.p, booktabs = T)  %>% kable_styling(position = "center")
```

	p=0.31	0.32	0.33	0.34	0.35	0.36	0.37	0.38	0.39	0.40
proportion of success	0.2	0.3	0.4	0.3	0.4	0.6	0.3	0.3	0.5	0.6

Problem 4

a.

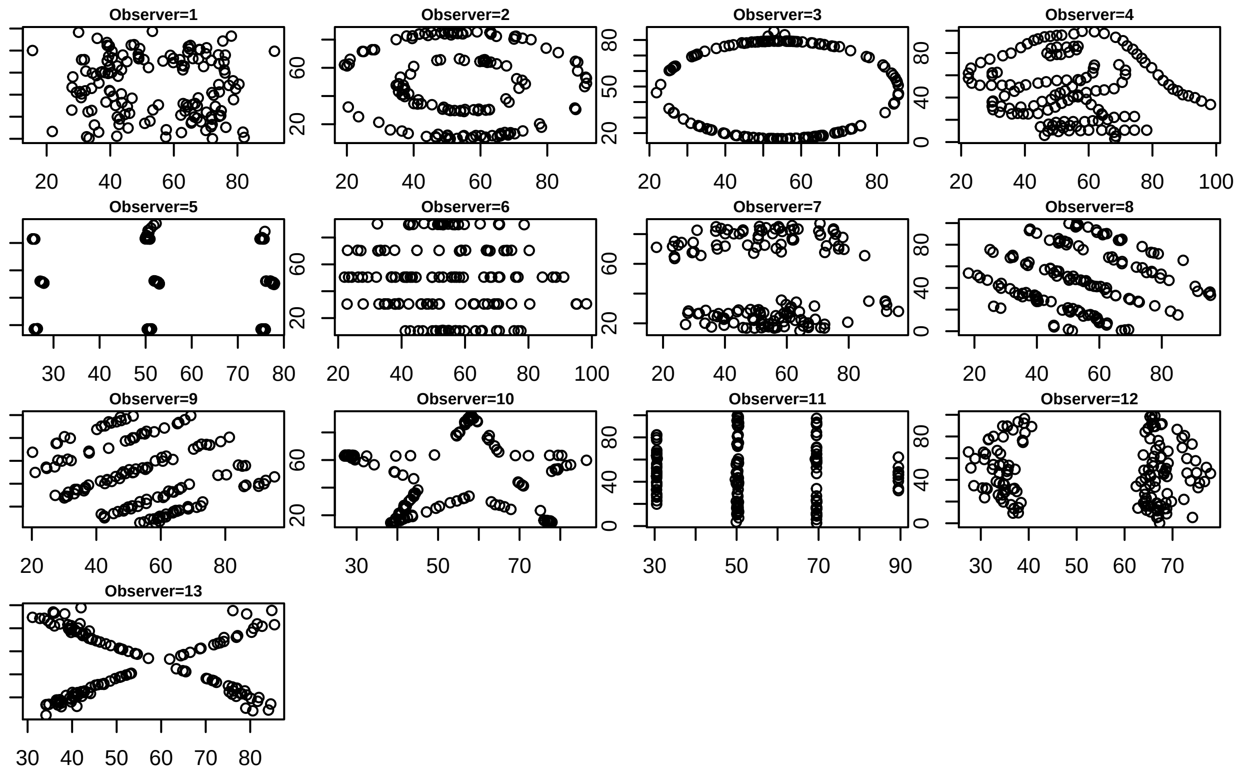
```r
data <- readRDS("HW3_data_canvas.rds")
colnames(data)<-c("Observer","x","y")
scatterplot<-function(df, title, x.lab, y.lab){
    if (length(unique(df[,1])) != 1){
      plot(df[,2], df[,3], main = title, xlab = x.lab, ylab = y.lab)
    }
    else {
      plot(df[,2], df[,3], xlab = x.lab, ylab = y.lab)
      title(paste0("Observer=",unique(df[,1])))
    }
}
scatterplot(data, "Scatter Plot for Entire Dataset", "dev1", "dev2")
```



b.

```
devdata<-array(NA,dim = c(142,3,13))
for (i in 1:13) {
  devdata[,,i]<-as.matrix(data[data$Observer == i,])
}
par(mfrow = c(4,4))
par(mar = c(2,1,1,1))
par(cex.main = 0.8)
apply(devdata, 3, scatterplot, title="Seperate Scatter Plot for Each Observer",x.lab="dev1",y.lab="dev2
```

```
## NULL
```



Problem 5

Part a.

```
    library(downloader)
    download("http://www.farinspace.com/wp-content/uploads/us_cities_and_states.zip",dest="us_cities_sta
    unzip("us_cities_states.zip", exdir="./")

    #read in data, looks like sql dump, blah
    states <- fread(input = "./us_cities_and_states/states.sql",skip = 23,sep = "'", sep2 = ",", header
    ### YOU do the CITIES
    cities <- fread(input = "./us_cities_and_states/cities_extended.sql",sep = "'", sep2 = ",", header =
    states<-states[-which(states$V4 == "DC"),]
    cities<-cities[-which(cities$V4  %in% c("DC","PR")),]
    colnames(cities)<-c("City","StateAbbrev")
    colnames(states)<-c("State","Abbrev")
```

Part b.

```
n.cities <- rep(NA,50)
for (i in 1:50) {
  n.cities[i] <- length(which(cities$StateAbbrev == states$Abbrev[i]))
}
city.count<-cbind(states[,1],n.cities)
colnames(city.count)<-c("state","NumberofCities")
city.count1<-city.count[1:25,]
city.count2<-city.count[26:50,]
disp<-cbind(city.count1,city.count2)
knitr::kable(disp,"latex", booktabs = T) %>% kable_styling(position = "center")
```

state	NumberofCities	state	NumberofCities
Alaska	273	Montana	405
Alabama	838	North Carolina	1090
Arkansas	709	North Dakota	407
Arizona	532	Nebraska	620
California	2651	New Hampshire	284
Colorado	659	New Jersey	733
Connecticut	438	New Mexico	426
Delaware	98	Nevada	253
Florida	1487	New York	2207
Georgia	972	Ohio	1446
Hawaii	139	Oklahoma	774
Iowa	1060	Oregon	484
Idaho	325	Pennsylvania	2208
Illinois	1587	Rhode Island	91
Indiana	989	South Carolina	539
Kansas	756	South Dakota	394
Kentucky	961	Tennessee	795
Louisiana	725	Texas	2650
Massachusetts	703	Utah	344
Maryland	619	Virginia	1238
Maine	489	Vermont	309
Michigan	1170	Washington	732
Minnesota	1031	Wisconsin	898
Missouri	1170	West Virginia	859
Mississippi	533	Wyoming	195

Part c. Create a function that counts the number of occurances of a letter in a string. The input to the function should be "letter" and "state_name". The output should be a scalar with the count for that letter.

Create a for loop to loop through the state names imported in part a. Inside the for loop, use an apply family function to iterate across a vector of letters and collect the occurance count as a vector.

```
    letter_count <- data.frame(matrix(NA,nrow=50, ncol=26))
    getCount <- function(letter,state_name){
        temp <- strsplit(tolower(state_name),"")
        count<-length(which(temp[[1]] == letter))
```

```r
        return(count)
    }
    for(i in 1:50){
        letter_count[i,] <- sapply(letters,getCount,state_name = states$State[i])
    }
    colnames(letter_count)<-letters
    rownames(letter_count)<-states$State
    kable(letter_count,"latex", booktabs = T) %>%
      kable_styling(latex_options = "scale_down")
```

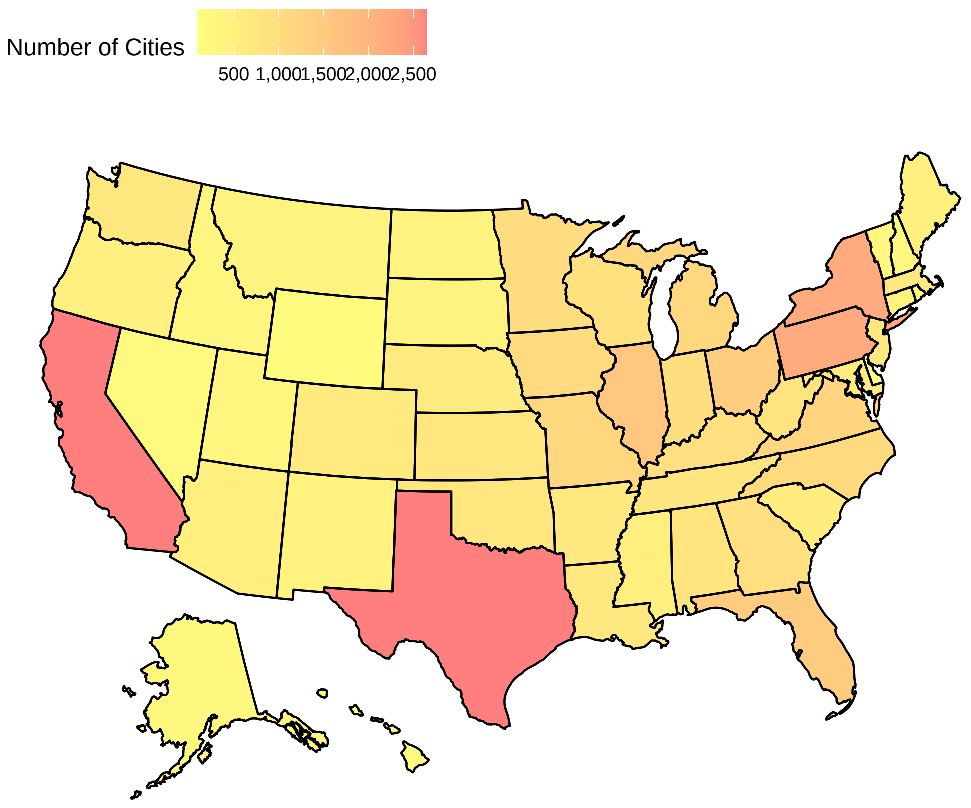
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
Alaska	3	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Alabama	4	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Arkansas	3	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1	2	0	0	0	0	0	0	0
Arizona	2	0	0	0	0	0	0	0	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1
California	2	0	1	0	0	1	0	0	2	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0	0	0
Colorado	1	0	1	1	0	0	0	0	0	0	0	1	0	0	3	0	0	1	0	0	0	0	0	0	0	0
Connecticut	0	0	3	0	1	0	0	0	1	0	0	0	0	2	1	0	0	0	0	2	1	0	0	0	0	0
Delaware	2	0	0	1	2	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0
Florida	1	0	0	1	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
Georgia	1	0	0	0	1	0	2	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
Hawaii	2	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Iowa	1	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0
Idaho	1	0	0	1	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Illinois	0	0	0	0	0	0	0	0	3	0	0	2	0	1	1	0	0	0	1	0	0	0	0	0	0	0
Indiana	2	0	0	1	0	0	0	0	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
Kansas	2	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	2	0	0	0	0	0	0	0
Kentucky	0	0	1	0	1	0	0	0	0	0	2	0	0	1	0	0	0	0	0	1	1	0	0	0	1	0
Louisiana	2	0	0	0	0	0	0	0	2	0	0	1	0	1	1	0	0	0	1	0	1	0	0	0	0	0
Massachusetts	2	0	1	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	4	2	1	0	0	0	0	0
Maryland	2	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	1	0
Maine	1	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Michigan	1	0	1	0	0	0	1	1	2	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Minnesota	1	0	0	0	1	0	0	0	1	0	0	0	1	2	1	0	0	0	1	1	0	0	0	0	0	0
Missouri	0	0	0	0	0	0	0	0	2	0	0	0	1	0	1	0	0	1	2	0	1	0	0	0	0	0
Mississippi	0	0	0	0	0	0	0	0	4	0	0	0	1	0	0	2	0	0	4	0	0	0	0	0	0	0
Montana	2	0	0	0	0	0	0	0	0	0	0	0	1	2	1	0	0	0	0	1	0	0	0	0	0	0
North Carolina	2	0	1	0	0	0	0	1	1	0	0	1	0	2	2	0	0	2	0	1	0	0	0	0	0	0
North Dakota	2	0	0	1	0	0	0	1	0	0	1	0	0	1	2	0	0	1	0	2	0	0	0	0	0	0
Nebraska	2	1	0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
New Hampshire	1	0	0	0	2	0	0	2	1	0	0	0	1	1	0	1	0	1	1	0	0	0	1	0	0	0
New Jersey	0	0	0	0	3	0	0	0	0	1	0	0	0	1	0	0	0	1	1	0	0	0	1	0	1	0
New Mexico	0	0	1	0	2	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	1	1	0	0
Nevada	2	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0
New York	0	0	0	0	1	0	0	0	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1	0	1	0
Ohio	0	0	0	0	0	0	0	1	1	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0
Oklahoma	2	0	0	0	0	0	0	1	0	0	1	1	1	0	2	0	0	0	0	0	0	0	0	0	0	0
Oregon	0	0	0	0	1	0	1	0	0	0	0	0	0	1	2	0	0	1	0	0	0	0	0	0	0	0
Pennsylvania	2	0	0	0	1	0	0	0	1	0	0	1	0	3	0	1	0	0	1	0	0	1	0	0	1	0
Rhode Island	1	0	0	2	1	0	0	1	1	0	0	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0
South Carolina	2	0	1	0	0	0	0	1	1	0	0	1	0	1	2	0	0	1	1	1	1	0	0	0	0	0
South Dakota	2	0	0	1	0	0	0	1	0	0	1	0	0	0	2	0	0	0	1	2	1	0	0	0	0	0
Tennessee	0	0	0	0	4	0	0	0	0	0	0	0	0	2	0	0	0	0	2	1	0	0	0	0	0	0
Texas	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
Utah	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
Virginia	1	0	0	0	0	0	1	0	3	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0
Vermont	0	0	0	0	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0	1	0	1	0	0	0	0
Washington	1	0	0	0	0	0	1	1	1	0	0	0	0	2	1	0	0	0	1	1	0	0	1	0	0	0
Wisconsin	0	0	1	0	0	0	0	0	2	0	0	0	0	2	1	0	0	0	2	0	0	0	1	0	0	0
West Virginia	1	0	0	0	1	0	1	0	3	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	0	0
Wyoming	0	0	0	0	0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0	1	0

Part d.

Create 2 maps to finalize this. Map 1 should be colored by count of cities on our list within the state. Map 2 should highlight only those states that have more than 3 occurances of ANY letter in thier name.

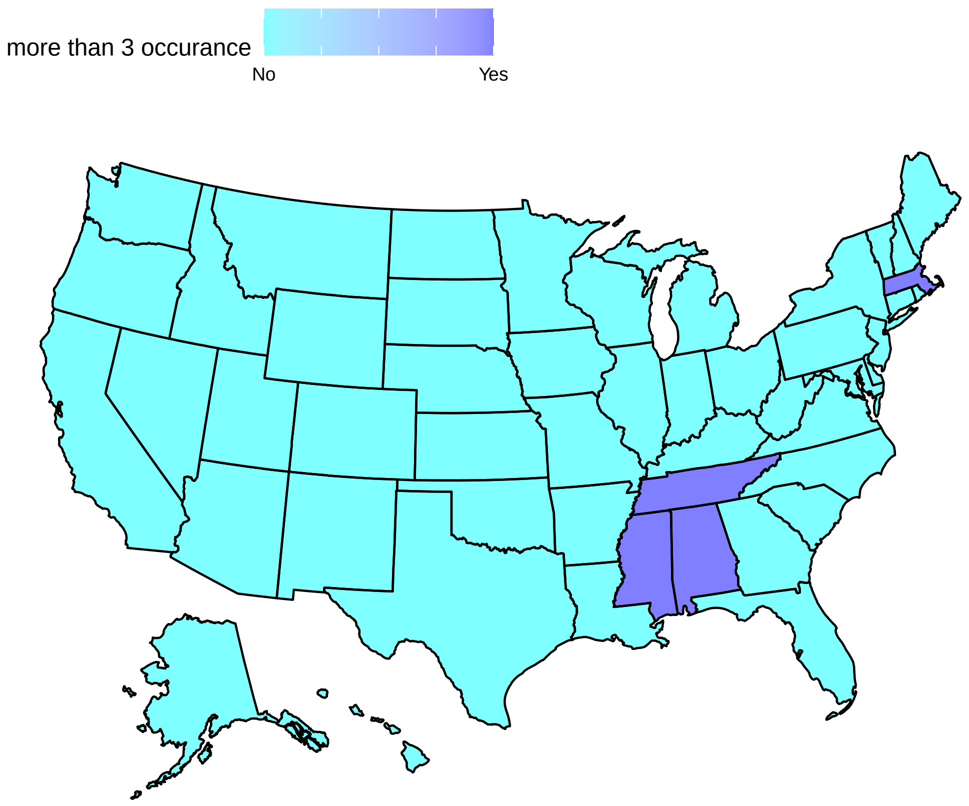
Quick and not so dirty map:

```
    library(ggplot2)
    library(usmap)

    plot_usmap(data = city.count, values = "NumberofCities", color = "black") +
      scale_fill_continuous( low = rgb(1,1,0,0.5), high = rgb(1,0,0,0.5),
                             name = "Number of Cities", label = scales::comma) +
      theme(legend.position = "top")
```



```
ifgt3<-function(x){
  x<-as.numeric(x)
  if (length(which(x > 3)) >= 1) {
    flag<-1
  }
  else {
    flag<-0
  }
  return(flag)
}

states.gt3<-apply(letter_count,1,ifgt3)
states.gt3<-cbind(states[,1],states.gt3)
colnames(states.gt3)<-c("state","states.gt3")
```

```r
plot_usmap(data = states.gt3, values = "states.gt3", color = "black") +
    scale_fill_continuous( low = rgb(0,1,1,0.5), high = rgb(0,0,1,0.5), name = "more than 3 occurance",
    theme(legend.position = "top")
```



Problem 2

```r
url1 <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
sensory_raw <- fread(url1,fill = T, skip = 1, data.table = F)
saveRDS(sensory_raw, "sensory.RDS")
sensory_raw <- readRDS("sensory.RDS")
sensory <- data.frame(item = rep(seq(1,10), each=15),operator = rep(c(1:5),30),n = NA)
temp_n<-matrix(NA,15,10)
for (i in 1:10) {
   temp_n[,i] <- as.numeric(c(sensory_raw[3*i-2,-1], sensory_raw[3*i-1,-6],sensory_raw[3*i,-6]))
}
sensory$n <- as.vector(temp_n)
```

Part a.

In the last line of his code:

```r
sd.boot[i]= coef(summary(lm(logapple08~logrm08, data = bootdata)))[2,2]
```

he wasn't supposed to use `logapple08` and `logrm08`, this two are the vectors he generated previously, not the variable name of the `bootdata`. Instead, he should use `AAPL.adjusted` and `IXIC.adjusted`.

Part b.

```r
set.seed(27356981)
id<-matrix(NA,5,30)
# Make sure the samples are balanced across operators
for (i in 1:5) {
  id[i,]<-which(sensory[,2] == i)
}
para.est<-matrix(NA,100,2)
boot.reg<-function(data){
  for (i in 1:100) {
    # Get sample id and sample data
    samp.id<-as.numeric(apply(id, 1, sample,size=30,replace=T))
    samp.data<-data[samp.id,]
    para.est[i,]<- lm(n~operator,data = samp.data)$coefficients
  }
  return(para.est)
}
time.reg<-system.time(result.reg<-boot.reg(sensory))
colnames(result.reg)<-c("(Intercept)","operator")
kable(summary(result.reg),"latex", booktabs = T) %>% kable_styling(position = "center")
```

(Intercept)	operator
Min. :3.470	Min. :-0.52500
1st Qu.:4.519	1st Qu.:-0.12992
Median :4.826	Median :-0.05767
Mean :4.827	Mean :-0.05619
3rd Qu.:5.113	3rd Qu.: 0.03500
Max. :6.147	Max. : 0.36033

Part c.

```r
  cores <- detectCores()-1
  cl <- makeCluster(cores)
  registerDoParallel(cl)
  set.seed(27356981)
  time.par<-system.time({
    result.par<- foreach(i=1:100,.combine = rbind) %dopar% {
    samp.id<-as.numeric(apply(id, 1, sample,size=30,replace=T))
    samp.data<-sensory[samp.id,]
    lm(n~operator,data = samp.data)$coefficients
  }
  })
  stopCluster(cl)
kable(summary(result.par),"latex", booktabs = T) %>% kable_styling(position = "center")
```

(Intercept)	operator
Min. :3.907	Min. :-0.38467
1st Qu.:4.486	1st Qu.:-0.15408
Median :4.719	Median :-0.05617
Mean :4.806	Mean :-0.06195
3rd Qu.:5.127	3rd Qu.: 0.05050
Max. :6.170	Max. : 0.19000

The reason why we can do this is that we were doing independent jobs that are not related to previous loops.

Create a single table summarizing the results and timing from part a and b. What are your thoughts?

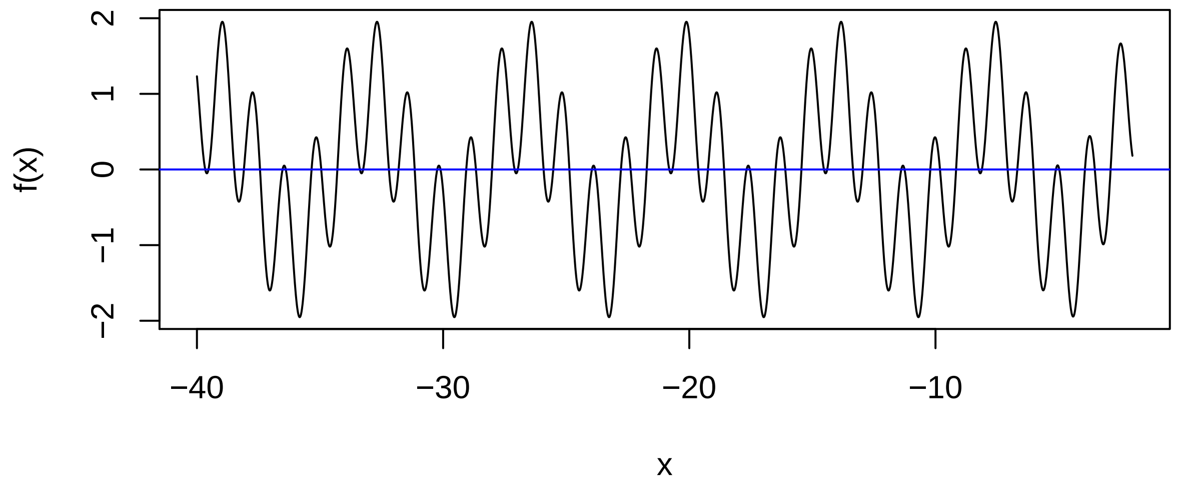
(Intercept).regular	operator.regular	(Intercept).parallel	operator.parallel
Min. :3.470	Min. :-0.52500	Min. :3.907	Min. :-0.38467
1st Qu.:4.519	1st Qu.:-0.12992	1st Qu.:4.486	1st Qu.:-0.15408
Median :4.826	Median :-0.05767	Median :4.719	Median :-0.05617
Mean :4.827	Mean :-0.05619	Mean :4.806	Mean :-0.06195
3rd Qu.:5.113	3rd Qu.: 0.03500	3rd Qu.:5.127	3rd Qu.: 0.05050
Max. :6.147	Max. : 0.36033	Max. :6.170	Max. : 0.19000

	time.reg	time.par
user.self	0.079	0.033
sys.self	0.002	0.004
elapsed	0.081	0.107
user.child	0.000	0.000
sys.child	0.000	0.000

Obviously, parallel computing take less time to do bootstrap even if we are dealing with such a small project. Since we are sampling independently in each bootstrap loop, we are able to use parallel computing. And the results are similar. We should always think of using parallel computing whenever we are faced with bootstrapping problems or other similar problems in order to increase our efficency.

Problem 3

A first skim at the function plot:



We can see that this function have infinite number of roots. Therefore, we will only deal with the roots that lie within the plotted interval: $[-40, -2]$.

Recall from previous homework, here is the implementation of Newton's method:

```
# function (1):
f <- function(x){
  func<-3^x-sin(x)+cos(5*x)
  return(func)
}
# First derivative:
df <- function(x){
  3^x*log(3) - cos(x) - 5*sin(5*x)
}
# Root finding using Newton's method
newton_root <- function(start.val,tol){
  # tol: tolerance used to terminate the loop
  # start.val: containing the 2 ending points of the chosen interval [a,b]

  # Table that contains x_0,x_1,...,x_n+1. Initial guess of the root: (a+b)/2
  # The 1st row of the table is x_n in nth loop, 2nd row is x_n+1 in nth loop
  root<-matrix(c((start.val[1]+start.val[2])/2,NA))
  # plot the function
  #x<-seq(start.val[1],start.val[2],0.01);y<-f(x)
  #plot(x,y,type = "l", xlim = c(start.val[1],start.val[2]))
  #abline(h = 0, col="blue")
  maxit <- 10000
  #iteration times:
  it<-0
  for (i in 1:maxit) {
    it<-it+1
    # calculate x_n+1.
    temp <- root[1,i]-f(root[1,i])/df(root[1,i])
```

```r
    # if x_n+1 < a, let x_n+1 = a
    # if x_n+1 > b, let x_n+1 = b
    if (temp < start.val[1]) {root[2,i] <- start.val[1]}
    else if (temp > start.val[2]) {root[2,i] <- start.val[2]}
    else {root[2,i] <- temp}

    # add lines and points to the plot
    #segments(root[1,i],f(root[1,i]),temp,0,lty = "dashed",col = "darkgreen")
    #abline(v = root[1,i],col="red",lty="dotted")
    #text(root[1,i],0.02, labels = it,cex = 0.7)
    #points(root[1,i], 0, pch = 16, col="red",cex = 0.7)

    # break when x_n+1 -x_n < tolerance
    if(abs(root[2,i] - root[1,i]) < tol) break
    # append the new roots
    root<-cbind(root,c(root[2,i],NA))
  }
  return(root[2,i])
}
```

Newton's method gives an answer for a root. To find multiple roots, you need to try different starting values. There is no guarantee for what start will give a specific root, so you simply need to try multiple. From the plot of the function in HW4, problem 8, how many roots are there?

Create a vector (`length.out=1000`) as a "grid" covering all the roots and extending +/-1 to either end.

Part a. Using one of the apply functions, find the roots noting the time it takes to run the apply function.

```r
# create grid containing all possible roots
root.grid <- seq(-40,-2,length.out = 1000)
# grid containing starting values extending the root grid to +/-1
start.grid<-cbind(root.grid-1,root.grid+1)
time.reg<-system.time( root.reg <- apply(start.grid, 1, newton_root, 1e-7))
# Make sure unique
root.reg<-almost.unique(root.reg,tolerance = 1e-7)
root.reg
```

```
##   [1] -39.662607 -39.531708 -38.091811 -38.484510 -37.437312 -36.717718
##   [7] -36.679680 -36.390115 -38.071071 -36.521015 -34.950218 -35.342917
##  [13] -36.169169 -34.295720 -33.560561 -34.913914 -33.248522 -33.310057
##  [19] -33.379422 -31.808626 -32.201325 -33.012012 -30.441441 -30.403403
##  [25] -31.154127 -31.794795 -31.756757 -30.106930 -30.237829 -30.172611
##  [31] -28.667033 -29.059732 -29.892893 -28.012534 -27.284284 -26.965337
##  [37] -28.637638 -27.096237 -26.029029 -25.525440 -25.918139 -26.735736
##  [43] -24.870942 -24.165165 -24.127127 -25.518519 -25.480480 -23.823744
##  [49] -23.954644 -22.383848 -22.776547 -21.729349 -22.612762 -21.008008
##  [55] -20.813051 -22.361361 -20.682152 -19.242255 -19.634954 -20.459459
##  [61] -18.587757 -17.888889 -17.850851 -19.242242 -19.204204 -17.540559
##  [67] -19.128128 -17.671459 -16.100662 -16.493361 -15.446164 -14.731732
##  [73] -14.693694 -16.085085 -14.398966 -14.529866 -14.468839 -12.959070
##  [79] -13.351769 -14.183183 -12.304571 -11.574575 -11.257371 -12.927928
##  [85] -11.388276 -10.319319  -9.817471 -10.210179 -11.026026  -9.162986
##  [91]  -8.455455  -8.417417  -9.808809  -9.770771  -8.115867  -8.246605
##  [97]  -6.676061  -7.068483  -6.021155  -5.412412  -5.298298  -5.107437
## [103]  -6.651652  -4.971508  -3.528723  -3.930114  -4.749750  -2.887058
## [109]  -2.179179  -2.141141  -3.494494  -1.858987  -1.905678  -2.236672
## [115]  -1.923010  -3.266266  -1.228228  -1.190190  -1.152152  -1.114114
## [121]  -1.076076  -1.038038
```

```
time.reg
```

```
##    user  system elapsed
##   2.276   2.067   4.387
```

There are 122 roots in [-40,-1].

Part b.

```
cores<-detectCores()-1
cl <- makeCluster(cores)
clusterExport(cl, "f")
clusterExport(cl, "df")
time.par<-system.time(root.par <- parApply(cl, start.grid, 1, newton_root, 1e-7))
stopCluster(cl)
# Make sure unique
root.par<-almost.unique(root.par,tolerance = 1e-7)
root.par
```

```
##   [1] -39.662607 -39.531708 -38.091811 -38.484510 -37.437312 -36.717718
##   [7] -36.679680 -36.390115 -38.071071 -36.521015 -34.950218 -35.342917
##  [13] -36.169169 -34.295720 -33.560561 -34.913914 -33.248522 -33.310057
##  [19] -33.379422 -31.808626 -32.201325 -33.012012 -30.441441 -30.403403
##  [25] -31.154127 -31.794795 -31.756757 -30.106930 -30.237829 -30.172611
##  [31] -28.667033 -29.059732 -29.892893 -28.012534 -27.284284 -26.965337
##  [37] -28.637638 -27.096237 -26.029029 -25.525440 -25.918139 -26.735736
##  [43] -24.870942 -24.165165 -24.127127 -25.518519 -25.480480 -23.823744
##  [49] -23.954644 -22.383848 -22.776547 -21.729349 -22.612762 -21.008008
##  [55] -20.813051 -22.361361 -20.682152 -19.242255 -19.634954 -20.459459
##  [61] -18.587757 -17.888889 -17.850851 -19.242242 -19.204204 -17.540559
##  [67] -19.128128 -17.671459 -16.100662 -16.493361 -15.446164 -14.731732
##  [73] -14.693694 -16.085085 -14.398966 -14.529866 -14.468839 -12.959070
##  [79] -13.351769 -14.183183 -12.304571 -11.574575 -11.257371 -12.927928
##  [85] -11.388276 -10.319319  -9.817471 -10.210179 -11.026026  -9.162986
##  [91]  -8.455455  -8.417417  -9.808809  -9.770771  -8.115867  -8.246605
##  [97]  -6.676061  -7.068483  -6.021155  -5.412412  -5.298298  -5.107437
## [103]  -6.651652  -4.971508  -3.528723  -3.930114  -4.749750  -2.887058
## [109]  -2.179179  -2.141141  -3.494494  -1.858987  -1.905678  -2.236672
## [115]  -1.923010  -3.266266  -1.228228  -1.190190  -1.152152  -1.114114
## [121]  -1.076076  -1.038038
```

```r
time.par
```

```
##    user  system elapsed
##   0.005   0.000   2.849
```

There are 122 roots in [-40,-1].

Create a table summarizing the roots and timing from both parts a and b. What are your thoughts?

```r
root.comp<-data.frame(cbind(summary(root.reg),summary(root.par)))
colnames(root.comp)<-c("regular","parallel")
kable(cbind(summary(root.reg),summary(root.par)),"latex", booktabs = T) %>% kable_styling(position = "c
```

Min.	-39.662607	-39.662607
1st Qu.	-29.684603	-29.684603
Median	-19.223223	-19.223223
Mean	-19.186748	-19.186748
3rd Qu.	-9.314932	-9.314932
Max.	-1.038038	-1.038038

```r
# Make sure they are the same
all.equal(root.reg,root.par)
```

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

```r
kable(cbind(time.reg,time.par),"latex", booktabs = T) %>% kable_styling(position = "center")
```

	time.reg	time.par
user.self	2.276	0.005
sys.self	2.067	0.000
elapsed	4.387	2.849
user.child	0.000	0.000
sys.child	0.000	0.000

The results of regular computing and parallel computing are exactly the same. And parallel computing took much less time than regular computing.