HW7_Brown

Nick Brown

10/26/2019

```
knitr::opts_chunk$set(echo = TRUE)
```

Problem 2:

Part a–First, the question asked in the stackexchange was why is the supplied code not working. This question was actually never answered. What is the problem with the code? If you want to duplicate the code to test it, use the quantreg package to get the data.

Answer: It appears that the code does not work because Adam did not iterate through the bootstrapping 1000 times. He created a variable called Boot_times, but creates a for loop to run through Boot, which does not have the 1000 stored in it.

Part b–Bootstrap the analysis to get the parameter estimates using 100 bootstrapped samples. Make sure to use system.time to get total time for the analysis. You should probably make sure the samples are balanced across operators, ie each sample draws for each operator.

Answer:

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------------------- tidyvers
```

```
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   0.8.3     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------------------------------------------- tidyverse_con
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
url <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
Sensory_raw <- read.table(url, header = F, skip = 1, fill = T,
stringsAsFactors = F)
Sensory_tidy <- Sensory_raw[-1, ]
Sensory_tidy_a <- filter(.data = Sensory_tidy, V1 %in% 1:10) %>%
rename(Item = V1, V1 = V2, V2 = V3, V3 = V4, V4 = V5,
V5 = V6)
Sensory_tidy_b <- filter(.data = Sensory_tidy, !(V1 %in%
1:10)) %>% mutate(Item = rep(as.character(1:10), each = 2)) %>%
mutate(V1 = as.numeric(V1)) %>% select(c(Item, V1:V5))
Sensory_tidy <- bind_rows(Sensory_tidy_a, Sensory_tidy_b)
colnames(Sensory_tidy) <- c("Item", paste("Person", 1:5,
sep = "_"))
Final_table <- Sensory_tidy %>% gather(Person, value, Person_1:Person_5) %>%
mutate(Person = gsub("Person_", "", Person)) %>% arrange(Item)
lm(value~Person, Final_table)
```

```
##
## Call:
## lm(formula = value ~ Person, data = Final_table)
##
```

```
## Coefficients:
## (Intercept)      Person2      Person3      Person4      Person5
##      4.5933       0.4700      -0.4267       0.6000      -0.3267
```

```
local_bootstrap <- system.time({
  coeff <- c()
for (i in 1:100) {
  ind <- sample(1:nrow(Final_table), replace = TRUE)
  sample_table<-Final_table[ind,]
  mdl=lm(value~Person, sample_table)
  coeff[i]=mdl['coefficients']
}
})
```

Part c–Redo the last problem but run the bootstraps in parallel (cl <- makeCluster(8)), don't forget to stopCluster(cl)). Why can you do this? Make sure to use system.time to get total time for the analysis.

Answer:

```
library(parallel)
library(pracma)
```

```
##
## Attaching package: 'pracma'
```

```
## The following object is masked from 'package:purrr':
##
##     cross
```

```
#install.packages("parallel")
cl <- makeCluster(8)
library(tidyverse)
url <- "http://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
Sensory_raw <- read.table(url, header = F, skip = 1, fill = T,
stringsAsFactors = F)
Sensory_tidy <- Sensory_raw[-1, ]
Sensory_tidy_a <- filter(.data = Sensory_tidy, V1 %in% 1:10) %>%
rename(Item = V1, V1 = V2, V2 = V3, V3 = V4, V4 = V5,
V5 = V6)
Sensory_tidy_b <- filter(.data = Sensory_tidy, !(V1 %in%
1:10)) %>% mutate(Item = rep(as.character(1:10), each = 2)) %>%
mutate(V1 = as.numeric(V1)) %>% select(c(Item, V1:V5))
Sensory_tidy <- bind_rows(Sensory_tidy_a, Sensory_tidy_b)
colnames(Sensory_tidy) <- c("Item", paste("Person", 1:5,
sep = "_"))
Final_table <- Sensory_tidy %>% gather(Person, value, Person_1:Person_5) %>%
mutate(Person = gsub("Person_", "", Person)) %>% arrange(Item)
lm(value~Person, Final_table)
```

```
##
## Call:
## lm(formula = value ~ Person, data = Final_table)
##
## Coefficients:
## (Intercept)      Person2      Person3      Person4      Person5
##      4.5933       0.4700      -0.4267       0.6000      -0.3267
```

```
cluster_bootstrap <- system.time({
  coeff <- c()
for (i in 1:100) {
  ind <- sample(1:nrow(Final_table), replace = TRUE)
  sample_table<-Final_table[ind,]
  mdl=lm(value~Person, sample_table)
  coeff[i]=mdl['coefficients']
}
})
stopCluster(cl)
```

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

Answer:

```
library(knitr)
kable(cbind(local_bootstrap,cluster_bootstrap), caption="Summary Speedtimes for Bootstrap")
```

Table 1: Summary Speedtimes for Bootstrap

	local_bootstrap	cluster_bootstrap
user.self	0.152	0.149
sys.self	0.015	0.000
elapsed	0.167	0.149
user.child	0.000	0.000
sys.child	0.000	0.000

Thoughts: The cluster and local bootstrap times are very similar. This is perhaps because I am not using a parApply command. Instead, I am running a for loop and it requires only 1 core; hence, even though I am using the cluster, it uses only one core.

Problem 3:

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?

Answer. Based on the plot of the function in HW4, problem 8, there are four roots, i.e., values that cross the X axis at 0.

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.

```
newton <- function(initGuess){
fx <- 3^initGuess - sin(initGuess) + cos(5*initGuess)
fxprime <- log(3)*3^(initGuess) - cos(initGuess) - 5*sin(5*initGuess)
f <- initGuess - fx/fxprime
}
find_root<-function(x){
roots <- c(x,rep(0,1000))
i<-2
tolerance <- 0.01
move <- 2
while(move>tolerance && i < 1001){
roots[i] <- newton(roots[i-1])
```

```
move <- abs(roots[i-1]-roots[i])
if (is.na(move)) {move<-0}
i <- i+1
}
return(roots[i-1])
}
library(pracma)
x = linspace(-6,-2,1000)
transpose_x <- t(x)
find_root(-5)
```

```
## [1] -4.970865
```

```
local_newton <- system.time({
  roots_grid <- sapply(x, find_root)
  })
```

Part b. Repeat the apply command using the equivelant parApply command. Use 8 workers. cl <- makeCluster(8).

```
library(parallel)
library(pracma)
#install.packages("parallel")
cl <- makeCluster(8)
newton <- function(initGuess){
fx <- 3^initGuess - sin(initGuess) + cos(5*initGuess)
fxprime <- log(3)*3^(initGuess) - cos(initGuess) - 5*sin(5*initGuess)
f <- initGuess - fx/fxprime
}
find_root<-function(x){
roots <- c(x,rep(0,1000))
i<-2
tolerance <- 0.01
move <- 2
while(move>tolerance && i < 1001){
roots[i] <- newton(roots[i-1])
move <- abs(roots[i-1]-roots[i])
if (is.na(move)) {move<-0}
i <- i+1
}
return(roots[i-1])
}
library(pracma)
clusterExport(cl, list("newton", "find_root"))
x = linspace(-6,-2,1000)
transpose_x <- t(x)
find_root(-5)
```

```
## [1] -4.970865
```

```
cluster_newton <- system.time({
  roots_grid <- parSapply(cl,x, find_root)
  })
stopCluster(cl)
```

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

```
library(knitr)
kable(cbind(local_newton,cluster_newton), caption="Summary Speedtimes for Newton")
```

Table 2: Summary Speedtimes for Newton

	local_newton	cluster_newton
user.self	0.037	0.005
sys.self	0.000	0.000
elapsed	0.037	0.099
user.child	0.000	0.000
sys.child	0.000	0.000

Thoughts: Because I am using the cluster computing, the time to complete the newton function with the cluster is 11 times faster than the local computing. I computed this by taking the local user time of .033 and divided by cluster user time of .003 to determine the 11 times factor speed.

Problem 4

Part a. What if you were to change the stopping rule to include our knowledge of the true value? Is this a good way to run this algorithm? What is a potential problem?

Answer: In order to include our knowledge of the true value in the algorithm, we should replace the stopping rule (z<5000000). This condition limits the total number of iterations. We should replace it with two conditions testing the distance between the last guess and the true value with using a tolerance value. Two problems could occur with this approach. If the tolerance is too small, it might take very long to converge to the true value; or, it may not converge to the true value at all.

If the tolerance is too large, the likelihood of convergence will increase. However, our estimations will always have higher errors.

Part c. Make a table of each starting value, the associated stopping value, and the number of iterations it took to get to that value. What fraction of starts ended prior to 5M? What are your thoughts on this algorithm?

```
library(parallel)
library(pracma)
library(knitr)
library(stats)
#install.packages("parallel")
cl <- makeCluster(10)
set.seed(1256)

grad_desc<-function(b)
{
  b0=b[1]
  b1=b[2]
theta <- as.matrix(c(1,2),nrow=2)
X <- cbind(1,rep(1:10,10))
h <- X%*%theta+rnorm(100,0,0.2)
#quick gradient descent
#need to make guesses for both Theta0 and Theta1, might as well be close
alpha <- 0.0000001 # this is the step size
m <- 100 # this is the size of h
tolerance <- 0.000000001 # stopping tolerance
theta0 <- c(b0,rep(0,999)) # I want to try a guess at 1, setting up container for max 1000 iters
```

```
theta1 <- c(b1,rep(0,999))
i <- 2 #iterator, 1 is my guess (R style indices)
#current theta is last guess
current_theta <- as.matrix(c(theta0[i-1],theta1[i-1]),nrow=2)
#update guess using gradient
theta0[i] <-theta0[i-1] - (alpha/m) * sum(X %*% current_theta - h)
theta1[i] <-theta1[i-1] - (alpha/m) * sum((X %*% current_theta - h)*rowSums(X))
rs_X <- rowSums(X) # can precalc to save some time
z <- 0
while(abs(theta0[i]-theta0[i-1])>tolerance && abs(theta1[i]-theta1[i-1])>tolerance && z<5000000){
if(i==1000){theta0[1]=theta0[i]; theta1[1]=theta1[i]; i=1; } ##cat("z=",z,"\n",sep="")}
z <- z + 1
i <- i + 1
current_theta <- as.matrix(c(theta0[i-1],theta1[i-1]),nrow=2)
theta0[i] <-theta0[i-1] - (alpha/m) * sum(X %*% current_theta - h)
theta1[i] <-theta1[i-1] - (alpha/m) * sum((X %*% current_theta - h)*rs_X)
}
theta0 <- theta0[i]
theta1 <- theta1[i]
return(c(theta0,theta1,z))
}
X <- cbind(1,rep(1:10,10))
theta <- as.matrix(c(1,2),nrow=2)
h<-X%*%theta+rnorm(100,0,0.2)
lm_fit <- lm(h~0+X)
true_coefs<-lm_fit$coefficients
library(pracma)

b<-cbind(linspace(true_coefs[1]-1,true_coefs[1]+1,100),linspace(true_coefs[2]-1,true_coefs[2]+1,100))
clusterExport(cl, list("grad_desc"))
answer_output<-t(parApply(cl, b, 1,grad_desc))
stopCluster(cl)

colnames(answer_output) <- c("theta0","theta1","iterations")
kable(head(answer_output,10), caption="Cluster Output Stopping Times")
```

Table 3: Cluster Output Stopping Times

theta0	theta1	iterations
0.1373850	2.126724	1634695
0.1551532	2.118500	1621143
0.1694941	2.116477	1636221
0.1881839	2.121497	1630062
0.2050842	2.117001	1625629
0.2214573	2.118119	1630965
0.2395650	2.113683	1617991
0.2532828	2.107999	1634712
0.2734408	2.108285	1611784
0.2884614	2.101906	1618091

The head is shown for the values. With 1,000 different combinations and a stopping value of 5,000,000, the fraction of starts that ended prior to 5M was 480 out of 1,000.

My thoughts are that the the closer the inital guess is to the true value, the fewer the iterations required to converge.