HW7

shuangshuang xu

10/25/2019

problem2

(a)

```r
library(quantreg)
```

```
## Loading required package: SparseM

##
## Attaching package: 'SparseM'

## The following object is masked from 'package:base':
##
##     backsolve
```

```r
library(quantmod)
```

```
## Loading required package: xts

## Loading required package: zoo

##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo

##
## Attaching package: 'xts'

## The following objects are masked from 'package:dplyr':
##
##     first, last

## The following objects are masked from 'package:data.table':
##
##     first, last

## Loading required package: TTR

## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo

## Version 0.4-0 included new data defaults. See ?getSymbols.
```

```r
#fetch data from Yahoo
#AAPL prices
```

```r
apple08 <- getSymbols('AAPL', auto.assign = FALSE, from = '2008-1-1', to =
                        "2008-12-31")[,6]
```

```
## 'getSymbols' currently uses auto.assign=TRUE by default, but will
## use auto.assign=FALSE in 0.5-0. You will still be able to use
## 'loadSymbols' to automatically load data. getOption("getSymbols.env")
## and getOption("getSymbols.auto.assign") will still be checked for
## alternate defaults.
##
## This message is shown once per session and may be disabled by setting
## options("getSymbols.warning4.0"=FALSE). See ?getSymbols for details.
```

```r
#market proxy
rm08<-getSymbols('^ixic', auto.assign = FALSE, from = '2008-1-1', to =
                   "2008-12-31")[,6]

#log returns of AAPL and market
logapple08<- na.omit(ROC(apple08)*100)
logrm08<-na.omit(ROC(rm08)*100)

#OLS for beta estimation
beta_AAPL_08<-summary(lm(logapple08~logrm08))$coefficients[2,1]

#create df from AAPL returns and market returns
df08<-cbind(logapple08,logrm08)
set.seed(666)
Boot=1000
sd.boot=rep(0,Boot)
for(i in 1:Boot){
  # nonparametric bootstrap
  bootdata=df08[sample(nrow(df08), size = 251, replace = TRUE),]
  sd.boot[i]= coef(summary(lm(AAPL.Adjusted~IXIC.Adjusted, data = bootdata)))[2,2]
}
head(sd.boot,10)
```

```
##  [1] 0.06861686 0.05623264 0.05940472 0.07227112 0.04946571 0.06534475
##  [7] 0.05475728 0.06822135 0.05293374 0.04875206
```

The names of y and x in linear model are wrong. I change them to the colname in bootdata, and it works. Here are first 10 values of coefficient.

(b)

```r
#import data
urla <- "https://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat"
sensory_data <- fread(urla,sep=" ",fill=TRUE,skip=1)
colnames(sensory_data)<-c("item","x1","x2","x3","x4","x5")
#get data frame contains NA
temp<-sensory_data %>% filter(is.na(x5)==TRUE) %>% select(1:5) %>% cbind(rep(1:10,each=2))
colnames(temp)<-c("x1","x2","x3","x4","x5","item")
#get data frame without NA
temp1<-sensory_data %>% filter(is.na(x5)==FALSE)
#combine
sensory_data<-rbind(temp1,temp)
sensory_data<-sensory_data[order(sensory_data$item),]
```

```r
rownames(sensory_data)<-NULL
```

```r
#bootstrap
system.time({
  set.seed(666)
  Boot=100
  beta.boot <- matrix(0, nrow = Boot, ncol = 6)
  for(i in 1:Boot){
    # nonparametric bootstrap
    bootdata=sensory_data[sample(nrow(sensory_data), size = nrow(sensory_data), replace = TRUE),]
    beta.boot[i,]= lm(item~., data = bootdata)$coefficients
  }
})
```

```
##    user  system elapsed
##   0.133   0.000   0.133
```

```r
colnames(beta.boot) <- c("intercept", "beta1", "beta2", "beta3", "beta4", "beta5")
kable(beta.boot[1:10, ], caption = "first 10 coefficients' value")
```

Table 1: first 10 coefficients' value

intercept	beta1	beta2	beta3	beta4	beta5
10.378107	0.8735692	-1.2023224	1.7815910	-2.1501746	0.3561322
7.399245	-0.0833066	-0.5642760	0.4117166	-1.3015785	1.4914741
6.524012	-0.1033929	-0.9785161	1.2801961	-1.3523566	1.2829954
7.722459	0.6705703	-0.1596634	1.2360311	-1.9597120	0.1495749
7.981068	-0.3650556	-0.1470396	2.6312412	-2.6143482	0.4755146
6.642910	-1.3735202	-0.2674568	2.1296914	-0.1553975	-0.3845279
7.079447	0.8219657	-0.2740008	0.3282275	-1.3958379	0.6873465
7.715345	1.1020509	-0.4818613	-1.7981322	-0.7571586	1.5882752
8.059349	0.2679573	-1.8842193	1.0650332	-1.1086626	1.7777014
7.353870	-0.5243531	-1.6478045	1.4581122	-0.9409016	1.7384898

part c

```r
cl <- makeCluster(8)
system.time({

  f <- function(x){

    #beta coefficient from bootstrap data
    bootdata=sensory_data[sample(nrow(sensory_data), size = nrow(sensory_data), replace = TRUE),]
    beta.boot <- lm(item~., data = bootdata)$coefficients
    return(beta.boot)
  }

  Boot <- matrix(1:100, ncol = 100)
  clusterExport(cl=cl, varlist=c("Boot", "f", "sensory_data"), envir=environment())
  beta_table <- t(parSapply(cl, Boot, f))

})
```

```
##    user  system elapsed
```

```
##   0.009   0.001   0.208
```

```
stopCluster(cl)
```

```
#kable(beta_table[1:10, ], caption = "first 10 coefficients' value")
```

Since each bootstrap do not relate with others, that's why we can calculate them in the same time. For b part, we used 0.143s, but in c part, we used 0.094s. Paralell processing helps save time.

problem 3

part a

```
func1 <- function(x){
  #f(x)
  return(3^x-sin(x)+cos(5*x))
}

x_upper <- -2
x_lower <- -22
x <- seq(from = x_lower, to = x_upper, by = 0.02)
y <- apply(as.matrix(x, nrow = 1), 2, func1)

#plot(x, y, type = "l")

func2 <- function(z){
  # find approximate x when f(x-e)*f(x+e)<0, e very small
  if (y[z]*y[z+1] < 0){
    return(mean(x[z],x[z+1]))
  } else {
    return(NA)
  }
}

i <- as.matrix(seq(1:(length(x)-1)), nrow = 1)

system.time({
  result <- sapply(i, func2)
})
```

```
##    user  system elapsed
##   0.009   0.000   0.008
```

```
result[!is.na(result)]
```

```
##  [1] -21.74 -20.82 -20.70 -19.64 -19.26 -18.60 -17.68 -17.56 -16.50 -16.12
## [11] -15.46 -14.54 -14.40 -13.36 -12.96 -12.32 -11.40 -11.26 -10.22  -9.82
## [21]  -9.18  -8.26  -8.12  -7.08  -6.68  -6.04  -5.12  -4.98  -3.94  -3.54
## [31]  -2.90
```

part b

```
cl <- makeCluster(8)
clusterExport(cl=cl, varlist=c("i", "func2", "y", "x"), envir=environment())
```

```r
system.time({
  result2 <- parSapply(cl, i, func2)
})
```

```
##    user  system elapsed
##   0.003   0.001   0.044
```

```r
stopCluster(cl)

result2[!is.na(result2)]
```

```
##  [1] -21.74 -20.82 -20.70 -19.64 -19.26 -18.60 -17.68 -17.56 -16.50 -16.12
## [11] -15.46 -14.54 -14.40 -13.36 -12.96 -12.32 -11.40 -11.26 -10.22  -9.82
## [21]  -9.18  -8.26  -8.12  -7.08  -6.68  -6.04  -5.12  -4.98  -3.94  -3.54
## [31]  -2.90
```

part a uses only 0.001s, however part b uses 0.043s. I think, if using apply family only needs extremely short time, there is no need to use parapply family.

The results are the same, since I used the same function and input.

problem 4

part a

It might not a good way to include the true value in the stopping rule. If the step size is a little bit large and with a not good start value, we may get a close enough answer, comparing with the true value.

part b

```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)
```

```r
# theta0_start <- seq(from = 0.47, to = 1.46, by = 0.01) # a series of starting value
# theta1_start <- seq(from = 1.5, to = 2.49, by = 0.01)
# thetaMatrix <-list()
# z <- 1
# for(i in 1:100){
#     thetaMatrix[[z]] <- c(theta0_start[i], theta1_start[i])
#     z <- z+1
#   }
#
#   #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
#   f <- function(t){
#     #input starting value of theta0 and theta1, output the value after gradient descent
#     theta0_s <- t[1]
#     theta1_s <- t[2]
#     theta0 <- c(theta0_s,rep(0,999)) # I want to try a guess at theta0_s(from the vector of theta0_st
#                                       # setting up container for max 1000 iters
```

```
#     theta1 <- c(theta1_s,rep(0,999))
#     i <- 2 #iterator, 1 is my guess (R style indecies)
#     #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]-0.9695707)>tolerance && abs(theta1[i]-2.001563)>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[1:i]
#     theta1 <- theta1[1:i]
#     result <- c(theta0_s, theta1_s, z, theta0[i], theta1[i])
#     return(result)
#   }
#
# cl <- makeCluster(8)
# clusterExport(cl=cl, varlist=c("thetaMatrix", "alpha", "m", "tolerance", "f", "theta", "X", "h"), env
#
# result3 <- parSapply(cl, thetaMatrix, f)
#
# stopCluster(cl)
#write.csv(result3, file = "/home/xshuangshuang/result3.csv")
```

Since it really costs time, I save the result and reload it for the next analysis and do not need to run again when knitting.

```
result3 <- read.csv("/home/xshuangshuang/result3.csv")
result3 <- t(result3[, -1])
colnames(result3) <- c("start_value_theta0", "start_value_theta1", "numOfIterations", "output_theta0",
kable(result3, caption = "table of starting value, stopping value and number of iteration")
```

Table 2: table of starting value, stopping value and number of iteration

	start_value_theta0	start_value_theta1	numOfIterations	output_theta0	output_theta1
V1	0.47	1.50	5000000	0.5786318	2.059558
V2	0.48	1.51	5000000	0.5864617	2.058396
V3	0.49	1.52	5000000	0.5942917	2.057235
V4	0.50	1.53	5000000	0.6021217	2.056073
V5	0.51	1.54	5000000	0.6099517	2.054912
V6	0.52	1.55	5000000	0.6177817	2.053750
V7	0.53	1.56	5000000	0.6256116	2.052589
V8	0.54	1.57	5000000	0.6334416	2.051427
V9	0.55	1.58	5000000	0.6412716	2.050266
V10	0.56	1.59	5000000	0.6491016	2.049104
V11	0.57	1.60	5000000	0.6569316	2.047942

	start_value_theta0	start_value_theta1	numOfIterations	output_theta0	output_theta1
V12	0.58	1.61	5000000	0.6647615	2.046781
V13	0.59	1.62	5000000	0.6725915	2.045619
V14	0.60	1.63	5000000	0.6804215	2.044458
V15	0.61	1.64	5000000	0.6882515	2.043296
V16	0.62	1.65	5000000	0.6960815	2.042135
V17	0.63	1.66	5000000	0.7039114	2.040973
V18	0.64	1.67	5000000	0.7117414	2.039811
V19	0.65	1.68	5000000	0.7195714	2.038650
V20	0.66	1.69	5000000	0.7274014	2.037488
V21	0.67	1.70	5000000	0.7352314	2.036327
V22	0.68	1.71	5000000	0.7430613	2.035165
V23	0.69	1.72	5000000	0.7508913	2.034004
V24	0.70	1.73	5000000	0.7587213	2.032842
V25	0.71	1.74	5000000	0.7665513	2.031680
V26	0.72	1.75	5000000	0.7743813	2.030519
V27	0.73	1.76	5000000	0.7822112	2.029357
V28	0.74	1.77	5000000	0.7900412	2.028196
V29	0.75	1.78	5000000	0.7978712	2.027034
V30	0.76	1.79	5000000	0.8057012	2.025873
V31	0.77	1.80	5000000	0.8135312	2.024711
V32	0.78	1.81	5000000	0.8213611	2.023550
V33	0.79	1.82	5000000	0.8291911	2.022388
V34	0.80	1.83	5000000	0.8370211	2.021227
V35	0.81	1.84	5000000	0.8448511	2.020065
V36	0.82	1.85	5000000	0.8526811	2.018903
V37	0.83	1.86	5000000	0.8605110	2.017742
V38	0.84	1.87	5000000	0.8683410	2.016580
V39	0.85	1.88	5000000	0.8761710	2.015419
V40	0.86	1.89	5000000	0.8840010	2.014257
V41	0.87	1.90	5000000	0.8918310	2.013096
V42	0.88	1.91	5000000	0.8996609	2.011934
V43	0.89	1.92	5000000	0.9074909	2.010772
V44	0.90	1.93	5000000	0.9153209	2.009611
V45	0.91	1.94	5000000	0.9231509	2.008449
V46	0.92	1.95	5000000	0.9309809	2.007288
V47	0.93	1.96	5000000	0.9388108	2.006126
V48	0.94	1.97	5000000	0.9466408	2.004965
V49	0.95	1.98	5000000	0.9544708	2.003803
V50	0.96	1.99	532704	0.9615307	2.001563
V51	0.97	2.00	5000000	0.9701308	2.001480
V52	0.98	2.01	5000000	0.9779607	2.000318
V53	0.99	2.02	465587	0.9875311	2.001563
V54	1.00	2.03	5000000	0.9936207	1.997995
V55	1.01	2.04	5000000	1.0014507	1.996834
V56	1.02	2.05	5000000	1.0092807	1.995672
V57	1.03	2.06	5000000	1.0171106	1.994511
V58	1.04	2.07	5000000	1.0249406	1.993349
V59	1.05	2.08	5000000	1.0327706	1.992187
V60	1.06	2.09	483537	1.0481959	2.001563
V61	1.07	2.10	5000000	1.0484306	1.989864
V62	1.08	2.11	5000000	1.0562605	1.988703
V63	1.09	2.12	5000000	1.0640905	1.987541

	start_value_theta0	start_value_theta1	numOfIterations	output_theta0	output_theta1
V64	1.10	2.13	5000000	1.0719205	1.986380
V65	1.11	2.14	5000000	1.0797505	1.985218
V66	1.12	2.15	5000000	1.0875805	1.984056
V67	1.13	2.16	5000000	1.0954104	1.982895
V68	1.14	2.17	5000000	1.1032404	1.981733
V69	1.15	2.18	5000000	1.1110704	1.980572
V70	1.16	2.19	5000000	1.1189004	1.979410
V71	1.17	2.20	5000000	1.1267304	1.978249
V72	1.18	2.21	5000000	1.1345603	1.977087
V73	1.19	2.22	5000000	1.1423903	1.975926
V74	1.20	2.23	5000000	1.1502203	1.974764
V75	1.21	2.24	5000000	1.1580503	1.973602
V76	1.22	2.25	5000000	1.1658803	1.972441
V77	1.23	2.26	5000000	1.1737102	1.971279
V78	1.24	2.27	5000000	1.1815402	1.970118
V79	1.25	2.28	5000000	1.1893702	1.968956
V80	1.26	2.29	5000000	1.1972002	1.967795
V81	1.27	2.30	5000000	1.2050302	1.966633
V82	1.28	2.31	5000000	1.2128601	1.965472
V83	1.29	2.32	5000000	1.2206901	1.964310
V84	1.30	2.33	5000000	1.2285201	1.963148
V85	1.31	2.34	5000000	1.2363501	1.961987
V86	1.32	2.35	5000000	1.2441801	1.960825
V87	1.33	2.36	5000000	1.2520100	1.959664
V88	1.34	2.37	5000000	1.2598400	1.958502
V89	1.35	2.38	5000000	1.2676700	1.957341
V90	1.36	2.39	5000000	1.2755000	1.956179
V91	1.37	2.40	5000000	1.2833300	1.955018
V92	1.38	2.41	5000000	1.2911599	1.953856
V93	1.39	2.42	5000000	1.2989899	1.952694
V94	1.40	2.43	5000000	1.3068199	1.951533
V95	1.41	2.44	5000000	1.3146499	1.950371
V96	1.42	2.45	5000000	1.3224799	1.949210
V97	1.43	2.46	5000000	1.3303098	1.948048
V98	1.44	2.47	5000000	1.3381398	1.946887
V99	1.45	2.48	5000000	1.3459698	1.945725
V100	1.46	2.49	5000000	1.3537998	1.944563

```r
result3[result3[, 3] == min(result3[, 3]), ]
```

```
## start_value_theta0 start_value_theta1    numOfIterations
##       9.900000e-01       2.020000e+00       4.655870e+05
##      output_theta0      output_theta1
##       9.875311e-01       2.001563e+00
```

The least number of iteration is 4.655870e+05, the starting values (theta0 = 0.99, theta1 = 2.02) are not the closest one to the true value (0.9696, 2.002). So, the closer starting values do not mean the shorter time.