HW5_boyuj

Boyu Jiang

11/11/2021

Problem 2

Part a.

The posted code uses a mismatch variable name while bootstrapping. It should be "lm(AAPL.Adjusted ~ IXIC.Adjusted, data = bootdata)".

Part b.

```
library(dplyr)
library(tidyr)
library(reshape)
# load data set
measure <- read.delim("https://www2.isye.gatech.edu/~jeffwu/wuhamadabook/data/Sensory.dat",
                      header = FALSE, skip = 2, sep=" ")
# organize data
a <- data.frame(cbind(measure[1,], measure[2,], measure[3,]))
for (i in 2:10) {
  a[i,] <- data.frame(cbind(measure[3*i-2,], measure[3*i-1,], measure[3*i,]))
}
# remove NA column
a <- a[,colSums(is.na(a))<nrow(a)]
# rename column
colnames(a) <- c("item",
                 "1.1st","1.2nd","1.3rd",
                 "2.1st","2.2nd","2.3rd",
                 "3.1st","3.2nd","3.3rd",
                 "4.1st","4.2nd","4.3rd",
                 "5.1st","5.2nd","5.3rd")
# build data set for modeling
a <- melt(a, id.vars = "item")

# separate operator and measurement into 2 columns
a <- separate(data = a, col = 'variable',
              into = c("operator", "TimeOfMeasurement"))
a$operator <- as.numeric(a$operator)

operators <- a$operator
measurevalue <- a$value
```

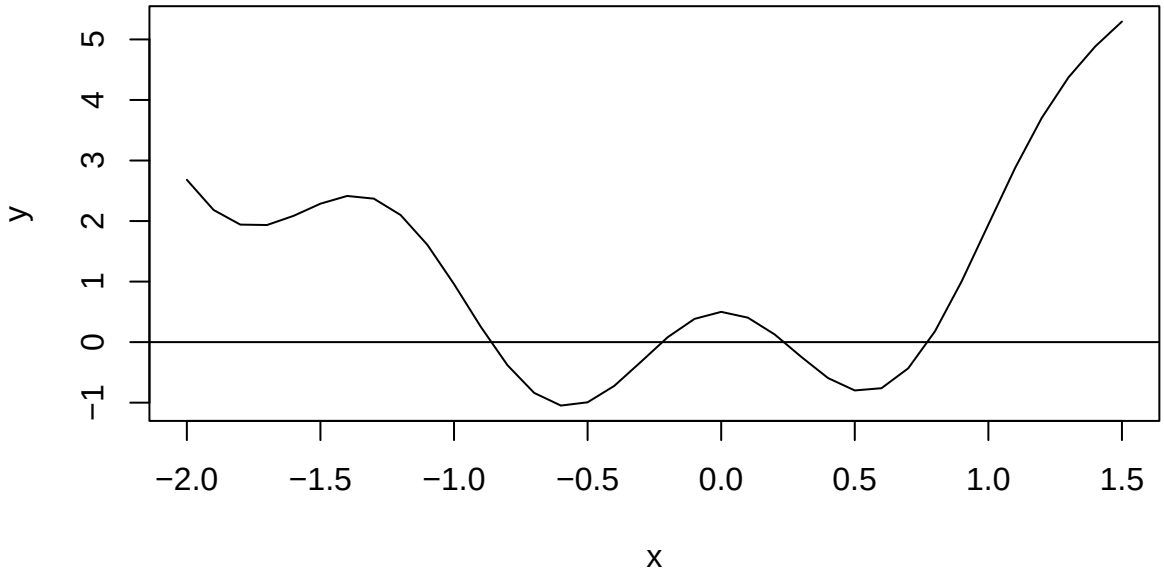
```r
# bootstrap function
bootstrap <- function(boot_samplesize){
    b0 <- c(NA)
    b1 <- c(NA)
    for (i in 1:100){
      data_boot <- a[sample(1:boot_samplesize, boot_samplesize, replace = TRUE),]
      b0[i] <- lm(value~operator, data=data_boot)$coefficients[1]
      b1[i] <- lm(value~operator, data=data_boot)$coefficients[2]
      }
    print(paste("Bootstrap estimate for intercept: ", round(mean(b0),2)))
    print(paste("Bootstrap estimate for operator: ", round(mean(b1),2)))
}

system.time(
  {
    set.seed(7)
    bootstrap(150)
  }
)
```

```
## [1] "Bootstrap estimate for intercept:  4.66"
## [1] "Bootstrap estimate for operator:  0.01"
```

```
##    user  system elapsed
##    0.11    0.00    0.12
```

Problem 3

Part a.



From the plot, there are four roots in all.
In the following function, the last position of x will be returned when it fails to converge.

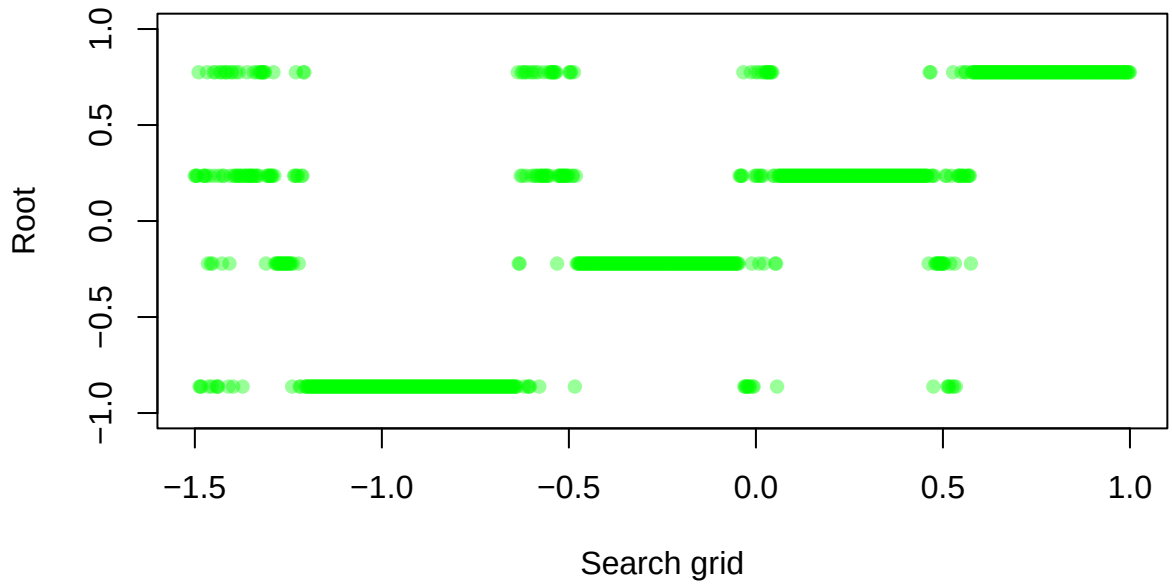
```
# function of Newtons method
# x0-starting point, N-max number of iterations
newton <- function(x0) {
  f <- function(x){3^x - sin(x) + cos(5*x) + x^2 - 1.5}
  h <- 0.001
  i <- 1
  x1 <- x0
  p <- numeric(100)
  while (i <= 100) { # max iteration = 100
    df.dx <- (f(x0+h)-f(x0))/h
    x1 <- (x0 - (f(x0)/df.dx))
    p[i] <- x1
    i <- i + 1
    c <- abs(x1-x0)
    x0 <- x1
    if(c < 0.00001){ # tolerance = 10^(-5)
      break
    }
  }
  return(p[i-1])
}
```

Part b.

```
gridvec <- seq(from = -1.5, to = 1, length.out = 1000)

system.time({
  sapply(X = gridvec[-363], FUN = newton)
})
```

```
##    user  system elapsed
##    0.05    0.00    0.05
```

```
plot(x = gridvec[-363],
     y = sapply(X = gridvec[-363], FUN = newton),
     ylim = c(-1,1), pch = 16,
     col = rgb(red = 0, green = 1, blue = 0, alpha = 0.4),
     xlab = 'Search grid',
     ylab = 'Root')
```



Problem 4

Part a.

The goal of gradient descent algorithm is to find the intercept and gradient values which correspond to the lowest MSE. It estimates intercept and slope through iteration over each set of X and Y data pairs whereby new intercept and gradient values are calculated as well as a new MSE.

```r
gradDesc <- function(c, m) {
  # initial values for intercept - 'c' and slope - 'm'
  yhat <- m * operators + c
  MSE <- sum((measurevalue - yhat) ^ 2) / 150
  converged = F
  iterations = 0
  while(converged == F) {
    # Implement the gradient descent algorithm
    m_new <- m - 0.001 * ((1 / 150) * (sum((yhat - measurevalue) * operators)))
    c_new <- c - 0.001 * ((1 / 150) * (sum(yhat - measurevalue)))
    m <- m_new
    c <- c_new
    yhat <- m * operators + c
    MSE_new <- sum((measurevalue - yhat) ^ 2) / 150
    if(MSE - MSE_new <= 0.00001) {
      converged = T
      return(c(c, m))
    }
    iterations = iterations + 1
    if(iterations > 5000) {
      converged = T
      return(c(c, m))
    }
  }
}

# test the function
# initial values for intercept - 'c' and slope - 'm'
gradDesc(c = runif(1, 3, 5), m = runif(1, 0, 1))
```

```
## [1] 4.1374397 0.1465223
```

Part b.

When the difference between new MSE and the old MSE is less than the convergence threshold, or the number of iteration reaches the maximum, the program stops.
If the true intercept and slope were known, the program would stop when the difference between estimate values and true values is less than the convergence threshold.
This algorithm may get a local optimal solution instead of a global optimal solution. I think the best guess for initial values is the true intercept and slope.

Part c. & d.

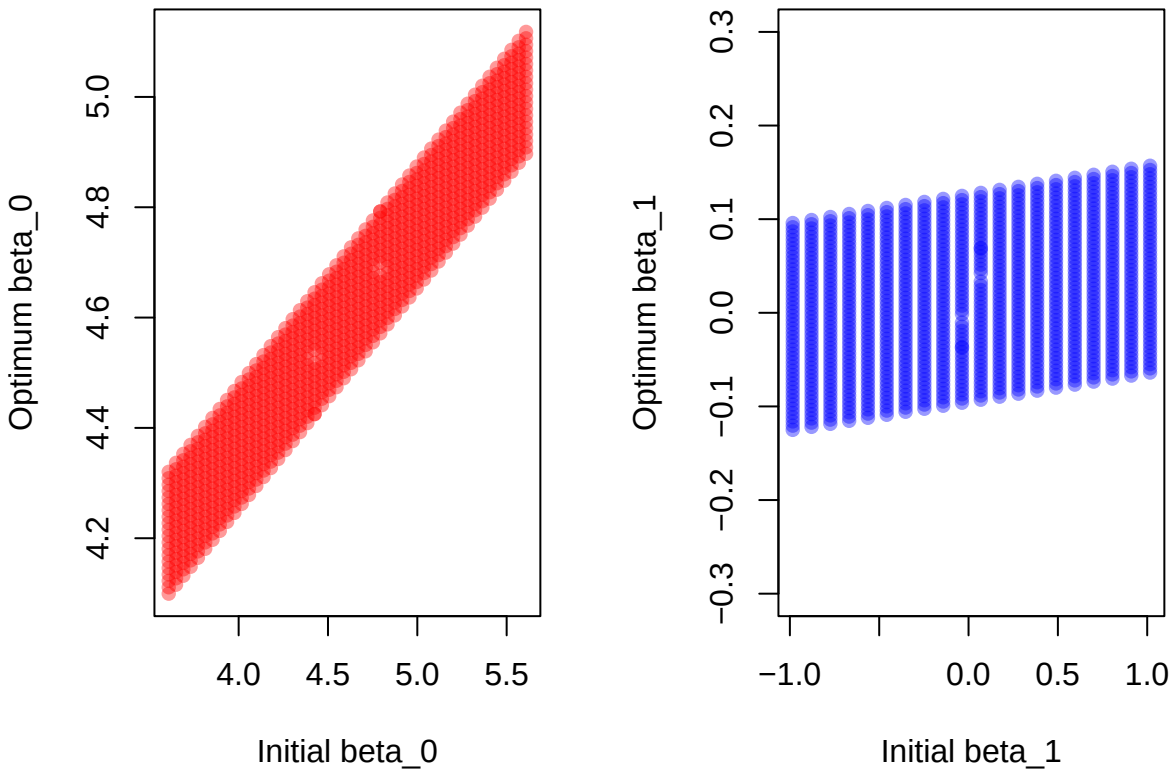
To reduce the running time, the max iteration is set to 5000, the tolerance 10^{-5}, the step size 0.001.

```r
# true value
trueb0 <- lm(value~operator, data=a)$coefficients[1]
trueb1 <- lm(value~operator, data=a)$coefficients[2]
# searching grid
gridb0 <- seq(trueb0 - 1, trueb0 + 1, length.out=50)
gridb1 <- seq(trueb1 - 1, trueb1 + 1, length.out=20)
```

```r
grid_b <- expand.grid(gridb0, gridb1)
colnames(grid_b) <- c('c', 'm')
# estimate optimum values
system.time(
  {
    opt <- mapply(FUN = gradDesc, c=grid_b$c, m=grid_b$m)
  }
)
```

```
##    user  system elapsed
##   17.01    0.03   17.07
```

```r
b0_opt <- opt[1,]
b1_opt <- opt[2,]
# plot
par(mfrow=c(1,2))
plot(x=grid_b$c, y=b0_opt, pch = 16,
     xlab = 'Initial beta_0', ylab = 'Optimum beta_0',
     col = rgb(red = 1, green = 0, blue = 0, alpha = 0.4))
plot(x=grid_b$m, y=b1_opt, pch = 16,
     xlab = 'Initial beta_1', ylab = 'Optimum beta_1',
     col = rgb(red = 0, green = 0, blue = 1, alpha = 0.4),
     ylim = c(-0.3,0.3))
```



From two plots, gradient descent algorithm works well in finding true values of parameters. The defect is that its computational time is long.