HW4_jaeyounglee

Jaeyoung Lee

October 13, 2020

Problem 2

Given **X** and $\vec{h}$ below, implement the above algorithm and compare the results with lm(h~0+**X**). State the tolerance used and the step size, α.

```r
# Random seed and true values
set.seed(1256)
theta <- as.matrix(c(1,2),nrow=2) # True parameter (theta_0, theta_1)
X <- cbind(1,rep(1:10,10))        # Design matrix
h <- X%*%theta+rnorm(100,0,0.2)   # True y value

# Parameters for the algorithm
tolerance <- 1e-9                 # Tolerance
alpha <- 1e-7                     # Step size
result <- NULL                    # Collect interesting information


# Set initial values
initial_value <-c(runif(1,0,2), runif(1,1,3)) # Initial value of theta
theta_new <- initial_value                    # Initial value of theta
theta_old <- c(10,10)                         # Initial value to operate loop
no_iter <- 1                                  # Number of iterations

# Gradient Descent algorithm
# If both theta values are smaller than tolerance, than it converges
while(all(abs(theta_new - theta_old) > tolerance) & no_iter < 5e+6 ){
  # Update new x value
  theta_old <- theta_new

  # Gradient Descent formula
  theta_new[1] <- theta_old[1] - alpha/length(h) * sum(X%*%theta_old - h)
  theta_new[2] <- theta_old[2] - alpha/length(h) * sum((X%*%theta_old - h)*X[,2])

  # Count the number of iteration
  no_iter <- no_iter + 1
}
# Collect iterations, initial values used, estimated theta
result <- rbind(result, c(no_iter, initial_value, theta_new))
result <- result %>% data.frame
names(result) <- c('no_iter', 'theta_0_start', 'theta_1_start',
                   'theta_0', 'theta_1')
result
```

```
##   no_iter theta_0_start theta_1_start   theta_0  theta_1
## 1 1588513     0.4746255      1.763589 0.5328478 2.063692
```

```
# Comparison with lm function
lmfit <- lm(h~0+X) # Fitting with simple linear regression
lmfit$coefficients
```

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

The outputs above are result of Gradient Descent and simple linear regression. Tolerance is $1e-9$ and the step size α is $1e-7$. The starting values for the Gradient descent algorithm are generated from uniform random numbers. From the outputs we can notice that the estimates for Θ_0 of two methods are quite different. On the other hand, Θ_1 values are quite similar. We expected that two methods have similar results. The difference might come from initial values and the step size α. If we use different initial values and step size, then they will have similar result. Thus, in *Problem 3* we will generate 10000 initial values.

Problem 3

Part a. Making sure to take advantages of parallel computing opportunities.

```
# Random seed and true values
set.seed(1256)
theta <- as.matrix(c(1,2),nrow=2) # True parameter (theta_0, theta_1)
X <- cbind(1,rep(1:10,10))        # Design matrix
h <- X%*%theta+rnorm(100,0,0.2)   # True y value
m <- length(h)                    # length of y (To speed up)

# Parameters for the algorithm
tolerance <- 1e-9                 # Tolerance
alpha <- 1e-7                     # Step size

# Generate 10000 initial values
n <- 10000                         # The number of initial value vector
initial_value <- cbind(runif(n,0,2), runif(n,1,3))

# Speed up using parallel computing
cores <- detectCores() - 1 # Use almost all the cores
cl <- makeCluster(cores)   # Create a cluster via makeCluster
registerDoParallel(cl)     # Register the cluster

# Making advantage of parallel computing
# Collect the interesting information from foreach
gradient_result <- foreach(init=1:n, .combine = rbind) %dopar% {
  # Set initial values
  theta_new <- initial_value[init,]  # Initial value of theta
  theta_old <- c(10,10)              # Initial value to operate loop
  no_iter <- 1                       # Number of iterations

  # Gradient Descent algorithm
  # If both theta values are smaller than tolerance, than it converges
```

```
  while(all(abs(theta_new - theta_old) > tolerance) & no_iter < 5e+6 ){
    # Update new x value
    theta_old <- theta_new
    h0 <- X%*%theta_old

    # Gradient Descent formula
    theta_new[1] <- theta_old[1] - alpha/m * sum(h0 - h)
    theta_new[2] <- theta_old[2] - alpha/m * sum((h0 - h)*X[,2])

    # Count the number of iteration
    no_iter <- no_iter + 1
  }
  c(no_iter, initial_value[init,], theta_new)
}

# Stop the cluster
stopCluster(cl)

# Save R Data
save.image(file = 'gradient_descent.RData')
```

```
# Load R Data
load(file = 'gradient_descent.RData')

# Make data frame of the result
gradient_result <- gradient_result %>% data.frame()
names(gradient_result) <- c('no_iter', 'theta0_start', 'theta1_start', 'theta0', 'theta1')
head(gradient_result)
```

```
##          no_iter theta0_start theta1_start     theta0   theta1
## result.1 1554759    0.4746255     1.800113 0.52756250 2.064447
## result.2 1764551    0.7635888     1.647124 0.82310918 2.022226
## result.3 1673410    1.5041688     2.367106 1.42569420 1.936662
## result.4  446942    1.7567372     1.700835 1.77098642 1.854033
## result.5  743555    0.1541603     2.783213 0.07975509 2.165166
## result.6  848185    1.6003094     1.078961 1.70002120 1.866936
```

```
# Minimum and maximum number of iteration, their initial values and theta estimates
gradient_result %>% filter(no_iter == min(no_iter))
```

```
##             no_iter theta0_start theta1_start     theta0   theta1
## result.689        2   1.99933259     1.814882 1.99933259 1.814882
## result.952        2   0.09146533     2.161888 0.09146533 2.161888
## result.1748       2   1.17564578     1.964011 1.17564578 1.964011
## result.2176       2   1.76543262     1.857382 1.76543262 1.857382
## result.2983       2   1.76630558     1.856044 1.76630558 1.856044
## result.3536       2   1.85590821     1.840816 1.85590821 1.840816
## result.3798       2   0.74341594     2.033758 0.74341594 2.033758
## result.3964       2   0.45559490     2.093276 0.45559490 2.093276
## result.5105       2   1.52871654     1.921588 1.52871653 1.921588
## result.5558       2   0.10102632     2.158865 0.10102632 2.158865
## result.5585       2   1.10913568     1.976064 1.10913568 1.976064
```

```
## result.6195       2   1.66264843     1.876521 1.66264843 1.876521
## result.6339       2   1.75351724     1.859784 1.75351724 1.859784
## result.6750       2   1.34015900     1.948744 1.34015899 1.948744
## result.7261       2   1.28847813     1.945134 1.28847812 1.945134
## result.7293       2   1.50114430     1.925518 1.50114429 1.925518
## result.7792       2   1.50086171     1.906308 1.50086171 1.906308
## result.8044       2   1.76310672     1.858735 1.76310672 1.858735
## result.8369       2   0.93829139     2.008499 0.93829139 2.008499
## result.8437       2   1.51125022     1.904692 1.51125022 1.904692
## result.8633       2   0.54479622     2.079628 0.54479622 2.079628
## result.9007       2   0.18449957     2.145518 0.18449957 2.145518
## result.9167       2   1.02181562     1.990324 1.02181562 1.990324
## result.9514       2   1.46134723     1.912146 1.46134723 1.912146
## result.9963       2   0.26106641     2.129458 0.26106641 2.129458
```

```
gradient_result %>% filter(no_iter == max(no_iter))
```

```
##              no_iter theta0_start theta1_start     theta0   theta1
## result.138    5e+06   1.74756654     1.886413 1.67045812 1.900887
## result.293    5e+06   1.78014603     1.878087 1.70025262 1.896607
## result.464    5e+06   0.28335306     2.110530 0.35042742 2.090497
## result.666    5e+06   1.57958792     1.907647 1.51958790 1.922558
## result.1143   5e+06   1.74689646     1.872616 1.67161522 1.900721
## result.1146   5e+06   1.84444338     1.869189 1.75809892 1.888298
## result.1425   5e+06   1.84036444     1.863119 1.75526985 1.888705
## result.1691   5e+06   1.96382697     1.854022 1.86533308 1.872895
## result.1807   5e+06   1.87244503     1.850030 1.78522774 1.884402
## result.2129   5e+06   0.13289936     2.125936 0.21575503 2.109842
## result.2224   5e+06   1.50075406     1.910348 1.44970363 1.932596
## result.2613   5e+06   1.46723031     1.929465 1.41770885 1.937192
## result.2689   5e+06   1.99221445     1.827241 1.89376803 1.868811
## result.2743   5e+06   1.69171391     1.888681 1.62090116 1.908006
## result.2770   5e+06   0.35499132     2.094591 0.41564165 2.081130
## result.3516   5e+06   1.53595385     1.906790 1.48120535 1.928072
## result.3551   5e+06   1.71655623     1.892971 1.64227193 1.904936
## result.3669   5e+06   0.00473579     2.171038 0.09698262 2.126902
## result.3854   5e+06   0.28241245     2.113455 0.34922700 2.090670
## result.4018   5e+06   1.49228553     1.917685 1.44130357 1.933803
## result.4106   5e+06   1.82716083     1.876926 1.74187299 1.890629
## result.4226   5e+06   0.30828619     2.105231 0.37309320 2.087241
## result.4350   5e+06   0.58537200     2.067509 0.62229986 2.051445
## result.4770   5e+06   0.15744212     2.133423 0.23645628 2.106868
## result.4902   5e+06   1.50132135     1.922191 1.44870342 1.932740
## result.4950   5e+06   0.01960615     2.163726 0.11102682 2.124885
## result.4961   5e+06   1.56554162     1.900184 1.50814273 1.924202
## result.4983   5e+06   0.46906421     2.080242 0.51808744 2.066414
## result.5153   5e+06   0.30164686     2.105272 0.36723125 2.088083
## result.5410   5e+06   1.89788150     1.850378 1.80762208 1.881185
## result.5622   5e+06   0.19895716     2.116461 0.27522734 2.101299
## result.5753   5e+06   1.93725466     1.851306 1.84223685 1.876213
## result.5756   5e+06   0.21710443     2.118526 0.29097410 2.099037
## result.5776   5e+06   1.80466675     1.873684 1.72244109 1.893420
## result.5877   5e+06   1.59413507     1.896878 1.53378491 1.920519
## result.5937   5e+06   1.74379599     1.887420 1.66700442 1.901383
```

```
## result.5988   5e+06   1.64454378     1.903043 1.57747101 1.914244
## result.6236   5e+06   0.45535907     2.084893 0.50540823 2.068236
## result.6641   5e+06   0.07068289     2.132251 0.16007153 2.117840
## result.7165   5e+06   1.82281058     1.855818 1.74071020 1.890796
## result.7187   5e+06   1.87881384     1.867238 1.78866555 1.883908
## result.7486   5e+06   0.14548822     2.145469 0.22438498 2.108602
## result.7511   5e+06   0.32408807     2.096221 0.38817423 2.085075
## result.7698   5e+06   0.13116371     2.150524 0.21110836 2.110509
## result.7827   5e+06   1.84741708     1.848181 1.76338406 1.887539
## result.8073   5e+06   1.69986073     1.871932 1.63021012 1.906668
## result.8162   5e+06   1.64930585     1.884928 1.58396714 1.913311
## result.8220   5e+06   1.98397376     1.838855 1.88502713 1.870066
## result.8239   5e+06   1.63992160     1.904401 1.57322152 1.914854
## result.8502   5e+06   1.81994178     1.862101 1.73738334 1.891274
## result.8632   5e+06   1.95177501     1.845765 1.85574793 1.874272
## result.8680   5e+06   1.67110384     1.876968 1.60420451 1.910404
## result.8710   5e+06   0.03940556     2.137320 0.13183839 2.121895
## result.8777   5e+06   1.86817923     1.864892 1.77958167 1.885213
## result.8832   5e+06   1.85311941     1.866522 1.76609033 1.887151
## result.8958   5e+06   0.53591922     2.070670 0.57827531 2.057769
## result.8959   5e+06   1.65613557     1.889674 1.58939052 1.912532
## result.8973   5e+06   0.24339819     2.112969 0.31487282 2.095604
## result.9155   5e+06   1.90527147     1.842255 1.81517027 1.880101
## result.9404   5e+06   1.72321818     1.876916 1.65018289 1.903800
## result.9486   5e+06   0.23440426     2.114394 0.30675843 2.096770
## result.9502   5e+06   0.23606745     2.110775 0.30868414 2.096493
## result.9524   5e+06   0.10224674     2.147931 0.18592831 2.114126
## result.9700   5e+06   0.31272438     2.106667 0.37682635 2.086705
## result.9760   5e+06   0.40132312     2.086628 0.45752153 2.075114
## result.9866   5e+06   0.03180393     2.169428 0.12106439 2.123443
```

```r
# Mean of theta estimates
gradient_result %>% select(theta0, theta1) %>% apply(2, mean)  %>% data.frame
```

```
##                .
## theta0 1.001573
## theta1 1.996530
```

```r
# Standard deviation of theta estimates
gradient_result %>% select(theta0, theta1) %>% apply(2, sd) %>%  data.frame
```

```
##                  .
## theta0 0.55864683
## theta1 0.09313795
```

From the output, we can know that some initial values do not work well. The minimum number of iteration is 2 with 25 cases. It means that the algorithm fails to work with the given initial values. The maximum number of iteration is $5M$ with 66 cases. It means than the algorithm did not converge with the given initial values. However, they are extreme cases among 10000 simulations. From the mean of 10000 samples, the estimates are similar to the true value $\Theta_0 = 1$, and $\Theta_1 = 2$. Therefore, the algorithm converges well to the true parameters. To compare the standard deviations, the standard deviation of Θ_0 is larger than that of Θ_1.

In addition, using parallel computing, it truly becomes faster.

Part b.

When we assume certain true values, we can conduct simulations using the assumed values like the result above. In practical situation, we do not know the true values. This means that it is impossible to put the true value into the stopping rule. However, we can conduct 'Explanatory Data Analysis', so from the result from EDA, we can generate random numbers based on the summary statistics. Then, it will have desirable result.

Part c.

The Gradient Descent algorithm we used here can be used as an alternative way of estimating parameters. It is similar to Newton's method. The Gradient Descent is already a popular algorithm, but it requires heavy computation and highly rely on initial values and step size. Therefore, in my opinion, the algorithm is good but need to be careful when we use it.

Problem 4: Inverting matrices

Ok, so John Cook makes some good points, but if you want to do:

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

what are you to do?? Can you explain what is going on?

The above equation is from linear regression. In R, there are `lm` function to find the regression coefficients. However, we can use `solve` function and find the coefficients. Also, instead of using `t` function, `crossprod` function works faster. Therefore, the command `solve(crosprod(x), crossprod(x,y))` would work well.

Problem 5: Need for speed challenge

In this problem, we are looking to compute the following:

$$y = p + AB^{-1}(q - r) \tag{1}$$

Where A, B, p, q and r are formed by:

```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.

How large (bytes) are A and B? Without any optimization tricks, how long does the it take to calculate y?

```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

# How big are A and B?
size_A <- object.size(A)
size_B <- object.size(B)

# Elapsed time computing y
computing_time_y <- system.time(p + A%*%solve(B, (q-r)))

# Save R Data
save.image(file = 'need_for_speed.RData')
```

```
## [1] "Size of A =  112347224 Byte"

## [1] "Size of B =  1816357208 Byte"

## [1] "Original computing time"

##    user  system elapsed
##  561.84    0.59  562.50
```

As we can see, the sizes of A and B are large. Also, the computing time is about nine minutes.

Part b.

The object sizes of A and B matrices are too large. However, they are have a lot of zero elements. Using this fact, we can reduce the object sizes of them using a package named `Matrix`. Using the package we can make the matrices be sparse and reduce the sizes. Since B matrix is too large, it is hard to make it to be sparse. However, if we partition the matrix, we can make the block matrices to be sparse. It is easy to partition and combine matrices. Therefore, it will take much less time than original computation.

Part c.

```r
# Need for speed challenge
library(Matrix) # Using Sparse Matrix
```

```
##
## Attaching package: 'Matrix'

## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```r
# Load R Data
load(file = 'need_for_speed.RData')

# Speed up by making matrices sparse to reduce the object sizes
computing_time_speedup <- system.time({
  A_sparse <- Matrix(A, sparse = TRUE) # Same matrix but reduce the data size
  #B_sparse <- Matrix(B, sparse = TRUE)

  # Make four block matrices of B and be sparse
  B_sparse_11 <- Matrix(B[1:7534, 1:7534], sparse = TRUE)
  B_sparse_12 <- Matrix(B[1:7534, 7535:15068], sparse = TRUE)
  B_sparse_21 <- Matrix(B[7535:15068, 1:7534], sparse = TRUE)
  B_sparse_22 <- Matrix(B[7535:15068, 7535:15068], sparse = TRUE)

  # Sparse matrix of B
  B_sparse <- rbind(cbind(B_sparse_11, B_sparse_12), cbind(B_sparse_21, B_sparse_22))

  p + A_sparse%*%solve(B_sparse, (q-r)) # Same formula with original one
})

computing_time_y       # original computing time
```

```
##    user  system elapsed
##  561.84    0.59  562.50
```

```r
computing_time_speedup # Speed up computing time
```

```
##    user  system elapsed
##    6.25    3.97   10.25
```

From `Matrix` package, there is a function `Matrix`. The function has an argument that make a matrix which has a lot of zeros be sparse, so it can reduce the object size of the matrix. Therefore, using sparse matrices, the computing time is much more faster than the original one.

Problem 3

a.

```r
# Define a function that computes the proportion of successes in a vector

prop_success <- function(x, success = 1){
  # The vector x has binary outcomes : 0 = fail, 1 = success
  # Define 'Success' argument what value you want to define as success (Character or numeric)
  x[x == success] <- 1          # Change success to 1

  # Proportion of success
  if(mode(x) == 'character'){
    prop <- sum(as.numeric(x[x == '1']))/length(x)  # Change the type of data
  }else{
    prop <- sum(x[x == 1])/length(x)                # Sample proportion
  }
```

```r
  return(prop)
}

# Example using the function
set.seed(10122020)
bin_outcome <- sample(c('f','s'), size = 100, replace = TRUE) # Binary outcomes
prop_success(bin_outcome, 's') # Proportion of success
```

```
## [1] 0.57
```

The defined function above computes the proportion of successes in a vector. The function is made for any type of vector. The function accept both numeric and character vector with binary outcomes.

b.

```r
# A matrix to simulate 10 flips of a coin with varying degrees of "fairness"
set.seed(12345)
P4b_data <- matrix(rbinom(10, 1, prob = (31:40)/100), nrow = 10, ncol = 10, byrow = FALSE)
colnames(P4b_data) <- c('.31', '.32', '.33', '.34', '.35', '.36', '.37', '.38', '.39', '.40')
P4b_data
```

```
##       .31 .32 .33 .34 .35 .36 .37 .38 .39 .40
##  [1,]   1   1   1   1   1   1   1   1   1   1
##  [2,]   1   1   1   1   1   1   1   1   1   1
##  [3,]   1   1   1   1   1   1   1   1   1   1
##  [4,]   1   1   1   1   1   1   1   1   1   1
##  [5,]   0   0   0   0   0   0   0   0   0   0
##  [6,]   0   0   0   0   0   0   0   0   0   0
##  [7,]   0   0   0   0   0   0   0   0   0   0
##  [8,]   0   0   0   0   0   0   0   0   0   0
##  [9,]   1   1   1   1   1   1   1   1   1   1
## [10,]   1   1   1   1   1   1   1   1   1   1
```

Above is a pre-defined matrix from the problem to simulate 10 flips of a coin with varying degrees of "fairness"

c.

```r
# Apply function with the custum function
apply(P4b_data, 2, prop_success)
```

```
## .31 .32 .33 .34 .35 .36 .37 .38 .39 .40
## 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6 0.6
```

It seems working well. The function brings correct result.

d.

```r
# Coinflip function based on given probabilities
coinflip <- function(p, n = 10){
  # The input n is the number of flips and p is probability
  flips <- sample(c(0,1), size = n, prob = c(1-p, p), replace = TRUE)
  return(flips)
}
# Apply the function on a probability vector using sapply
sapply((31:40)/100,  coinflip)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    0    0    1    1    1    1    1    1    1     0
##  [2,]    0    0    0    0    1    0    0    0    1     1
##  [3,]    1    1    0    1    0    1    0    0    0     1
##  [4,]    0    1    1    1    0    1    0    0    0     1
##  [5,]    0    0    0    0    1    1    0    0    1     0
##  [6,]    0    0    0    0    0    0    0    0    0     1
##  [7,]    0    1    1    0    1    1    1    1    1     1
##  [8,]    0    0    1    0    0    0    1    0    1     1
##  [9,]    0    0    0    0    0    0    0    1    0     0
## [10,]    1    0    0    0    0    1    0    0    0     0
```

The newly defined function is coin flipping function based on given probabilities. Using `sapply` function, we can easily make matrix with the custom function. The matrix is an output of `sapply` and the custom function.

Problem 4

```r
# Load data
# Multiple repeated measurements from two devices (dev1 and dev2) by thirteen Observers.
devices <- readRDS('HW3_data.rds')
names(devices) <- c('Observer', 'x', 'y')

# A function of data frame
scatter <- function(devices, observer = 1, col1 = 'brown', col2 = 'black',
                    main = 'Scatter plot of X and Y'){
  # The inputs are data, observer #, title of plot, and colors
  # Choose colors of single plot and a plot of observer
  # We can choose a scatter plot of certain observer we want to see

  # This function is Based on ggplot2, tidyverse, ggpubr package
  require(ggplot2)
  require(tidyverse)
  require(ggpubr)

  # A single scatter plot of the entire dataset
  singleplot <- ggplot(data = devices, aes(x=x, y=y)) +
    geom_point(col = col1, size = 3, shape = 19) +
    labs(title = main)

  # A separate scatter plot using the apply function
  devices_obs <- devices %>% filter(Observer == observer) # Part of data by observer
```

```
  separateplot <- ggplot(data = devices_obs, aes(x=x, y=y)) +
    geom_point(col = col2, size = 3, shape = 19) +
    labs(title = paste('Scatter plot of X & Y of Obs', observer))

  ggarrange(singleplot, separateplot, ncol=2)

}

# Single scatter plot and a scatter plot by observers
for(i in 1:2){
  scatter(devices, i, col2 = i)
}
```