Homework 5 SOLUTIONS

Due October 2, 2019

2019-10-07

Problem 2: Sums of Squares

In this problem, we will calculate sums of squares total using:

a. a for loop to iterate through all data points calculating the summed squared difference between the data points and mean of the data.

b. repeat part a, but use vector operations to effect the same computation

In both cases, wrap the code in "system.time({})". You should report the final answer and timings for both a and b.

To generate the data, use:

```
set.seed(12345)
y <- seq(from=1, to=100, length.out = 1e8) + rnorm(1e8)
```

Part a.

Here we need to use a for loop to calculate sums of squares for the data in $\vec{y}$ created above.

```
#need an accumulator
ss_y_for <- 0
partA <- system.time({
  for(i in 1:length(y)){
    ss_y_for <- ss_y_for + y[i]
  }
})
```

Part b.

Repeat (a), but using vectors.

```
#need a 1's vector of appropriate size
ones <- rep_len(1, length.out = 1e8)
partB <- system.time({
  ss_y_manual <- y %*% ones
})
```

The full problem gives the answers and timings given in Table 1 below. Note, but answers are identical as expected. Vectorizing the math is much faster, also as expected.

Table 1: Result of timing of sum of square methods

SS y	actual time
5049995862	2.817
5049995862	0.424

Problem 3: Using the dual nature to our advantage

As above, sometimes using a mixture of true matrix math plus component operations cleans up our code giving better readibility. Suppose we wanted to form the following computation:

- $while(abs(\Theta_0^i - \Theta_0^{i-1})$ AND $abs(\Theta_1^i - \Theta_1^{i-1}) > tolerance)$ {

$$\begin{aligned}\Theta_0^i &= \Theta_0^{i-1} - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_0(x_i) - y_i)\\ \Theta_1^i &= \Theta_1^{i-1} - \alpha\frac{1}{m}\sum_{i=1}^{m}((h_0(x_i) - y_i)x_i)\end{aligned}$$

}

Where $h_0(x) = \Theta_0 + \Theta_1 x$.

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

```
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)
```

```
#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(1,rep(0,999)) # I want to try a guess at 1, setting up container for max 1000 iters
theta1 <- c(1,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]-theta0[i-1])>tolerance && abs(theta1[i]-theta1[i-1])>tolerance && z<2000000){
  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]

lm_fit <- lm(h~0+X)
```

Strictly speaking, the stepsize as I have defined it is $\alpha/2$ due to the derivative of the function giving rise to an additional factor of 2. But, as written, I have α=10^{-7} and a tolerance=10^{-9}. This gives the following results after 528 iterations.

Table 2: Gradient Descent vs R's lm

	X1	X2
Gradient Descent	1.1190258	1.974507
R's lm	0.9695707	2.001563

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?

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:

```
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?

```
# straight up calc
timerA <- system.time({
  y <- p + A %*% solve(B) %*% (q - r)
})
```

Wow, on this machine, the compute takes 796.681 seconds. About 1/2 of what it took on my old computer. A is 1.1234722×10^8 while B is 1.8163572×10^9 in bytes.

Part b.

How would you break apart this compute, i.e., what order of operations would make sense? Are there any mathmatical simplifications you can make? Is there anything about the vectors or matrices we might take advantage of?

This is a sparce matrix problem. The Kronecker product puts 10x10 (also sparse) matricies on the diagonal, all off diagonal blocks are 0. To make this faster, we should use an algorithm designed for sparce matricies and try to incorporate the solve comments from above.

Part c.

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".

```
# going to use the Matrix package
# going to incorporate the solve
library(Matrix)

timerB <- system.time({
  sB <- Matrix(B, sparse = TRUE)
})
timerC <- system.time({
  y <- p + A %*% solve(sB,(q-r))
})
```

Changing two things, a) converting matrix B to a spare matrix class and b) using solve by providing both the matrix we want inverted and the full system of equations to solve converts a 10+ min problem to ca. 7 seconds!!