Stat 243 PS 7

A.J. Torre

November 13, 2018

Comments: I worked with Malvika, JD, Hatim, Phil, Huy, and Vaibhav on this homework assignment.

Questions

Question 1)

Looking at Section 1.2.1 Unit 10, we can use Monte Carlo estimation to see if an estimator is biased or not.

Question 2)

a)

If this data set is floating point numbers, then we can look at our number of observations, n, then we have $(1 * 10^9) * 8 = 8000000000$ bytes in our datase. If we account for p, $n * p = (1 * 10^9) * 8 = 8000000000$ total numbers at 8 bytes each, which will be 64000000000 bytes, which is $64,000$ megabytes. This is too much data for a laptop to be able to handle and do computations with.

b)

One way to solve this could be to just store the unique 10,000 values so we would have a m by p matrix with m equals 10,000 and p still equals 8. Then, we could have a separate vector that is length 10ˆ9 that serves as a pointer for which entry in the smaller/ unique value matrix that the index i (i out of 10^9) corresponds to. So, for memory, our smaller matrix would be $10,000$ (number of rows) times 8 (number of columns) times 8 (number of bytes per value) is $64,000 bytes$, which is about 159 mb. Then, for our pointer vector, we could store each entry as an integer, so 4 bytes each so we would $4 * 10^9$ bytes, which is 4,000 megabytes. So, in total, this approach will take about 4,150 megabytes of data, which is still a lot, but much more manageable than the matrix in part a.

c)

One issue is that we can't physically just enter in the small matrix and a pointer vector into the lm, glm, etc. functions as our data input. Another issue is that the way that the lm/ glm code works is that it will need to remake the big X matrix anyways to do its computations so we wouldn't be saving any memory once lm or glm is actually called.

d)pseduo code

We want to solve $X^T XB = X^T Y$.One idea is to use a weighted matrix that's constructed from the matrix with the 10,000 unique rows and the pointer vector. We'll use a vector to count how many times a certain value appears and we'll weight those rows more. Then, we will follow the notes from class to do a QR decomposition (where we end up with the final equation $RB = Q^T Y$, where R\$ is upper triangular so we can backsolve to solve for B as this problem (with the exception of the very large data set/ need to make use of our current data structures), is

```
pointer= 10^9 vector of indices
small_x= unique values of our big X matrix so has dimensions 10,000 by 8

#this part is to use how many times a value appears in our pointer vector to see if
for element in pointer
  element_weight<-0
  for value in small_x
  #if the value of small_x at the index of the element in our
  #pointer vector equals the value we're currently iterating over
  #we add one to our count
    if small_x[element]==value
        element_weight<-element_weight +1
    #divide by 1 to get some proportion
      result<-element_weight/length(pointer)
    #to overwrite (save memory) of the weighted value in
    #the old value's place
   small_x[value]<-value * result
#this creates new_weighted_matrix

#Next we do the QR decomposition of our weighted x matrix
new_weighted_matrix.qr<-qr(new_weighted_matrix)
weighted_q<-qr.Q(new_weighted_matrix.qr)
weighted_r<-qr.R(new_weighted_matrix.qr)
#for some given y, we calculate
rhs<-crossprod(q,y)
result<-backsolve(weighted_r,rhs)
```

Question 3)

We want to compute the generalized least squares $B = (X^T S^{-1} X)^{-1} X^T S^{-1} Y$, for X is an n by p matrix, S is n by n, and we assume n is greater than p. So, first to start with solving this, we use some linear algebra facts to help simplify our equation. So, we start with $B = (X^T S^{-1} X)^{-1} X^T S^{-1} Y$, and then since S is a covariance matrix, we know it should be symmetric (so S^{-1} is also symmetric) and positive definite so we know that we can say that $S^{-1} = S^{-1/2} S^{-1/2}$ so we can rewrite our equation as $B = (X^T S^{-1/2} S^{-1/2} X)^{-1} X^T S^{-1/2} S^{-1/2} Y$ and we can say a new matrix Z is such that $Z = S^{-1/2} X$ so that $Z^T = X^T S^{-1/2}$. Then, we have $B = (Z^T Z)^{-1} Z^T Y$, and we set $Y* = S^{-1/2} Y$. Then, we get $B = (Z^T Z)^{-1} Z^T Y*$, and thus, $Z^T Z B = Z^T Y$ so we end with $ZB = Y*$.

So, to do the pseudo code/ steps needed for our computation in code: Idea Using Choleski & QR Decomposition: - First, we need to get $S^{-1/2}$ from S. So, we consider that S is a covariance matrix and is square and symmetric. So, we do a Choleski decomposition on S to get $S = U^T U$ where $U = chol(S)$ and U will be the square root matrix. - Then, we need the inverse of this to complete finding $S^{-1/2}$ by solving $UU^{-1} = I$ for U^{-1} by using backsolve on U and I. So, thus, $S^{-1/2} = U^{-1}$. - Then, we get our Z matrix by doing our answer for our $S^{-1//2}$ times X, a function input, and then we can solve for $Y*$ for a given Y by doing our answer for $S^{-1/2}$ times Y. - Next, we do a QR decomposition, following the notes on section in 4.4.2 in Unit 9 using Z and Y*. - Lastly, we do a backsolve as we are left with the equation $RB = Q^T Y*$, where $Z = QR$ and we backsolve using R and $Q^T Y*$.

```
#this code is trying the Choleski decomp
#to solve for gls
gls<-function(x,s,y){
  U<-chol(s)
  n<-nrow(U)
  I<-diag(n)
  #solving for S^(-1/2) using chol
  s_squarert_inverse<-backsolve(U,I,n)
  z<-s_squarert_inverse %*% x
  y_star<-s_squarert_inverse%*%y
  #following notes from unit 9 on QR decomposition
  z_qr<-qr(z)
  q<-qr.Q(z_qr)
  r<-qr.R(z_qr)
  rhs<-crossprod(q,y_star)
  #can just backsolve from here since R
  #is upper triangular
  result<-backsolve(r,rhs)
  return(result)}
```

Question 4)

So, we are told that $Ax = b$ we can do solutions $n^3/3$
####a) To solve $UZ = I*$, we know that from a LU decomposition, $A = LU$ for some lower-triangular matrix L and some upper-triangular matrix U so $AZ = I$ is equivalent to $LUZ = I$, then to just get $UZ = I$ is to multiply both sides by L^-1. So, to transform from A to U on the left-hand side is order $n^3/3$ and then to transform I to $I*$ by multiplying by L^-1 is order n^3. So, in total, this order of computation is $4n^3/3$. ####b) Next, to solve for Z, given $UZ = I*$, where U, Z, and $I*$ are n by n matrices. To both multiply and solve this system, without using backsolve, is $n^3 + n^2/2$, so just counting higher order terms is $n^3/2$. ####c) Lastly, to solve $z = Zb$, where Z is an n by n matrix and b is an n by 1 vector is order n^2.

So, to compare this computation cost to $n^3/3$, we will add the highest order terms together, so we get $4n^3/3$ plus $n^3/2$, which equals $11n^3/6$, which is considerably larger than $n^3/3$ and is almost 6 times larger.

Question 5)

Below is the code to compare the different times to solve a matrix system:

```
#constructing our matrix
n<-5000
W<-matrix(rnorm(n^2),n)
X<-t(W)%*%W
y<-rnorm(n)

system.time(output1<-solve(X)%*%y) #super long time!
```

```
##    user  system elapsed
##  139.39    0.50  145.80
```

```
system.time(output2<-solve(X,y)) #okay with solving time
```

```
##    user  system elapsed
##   29.38    0.16   30.39
```

```r
system.time(U<-chol(X))#time to create chol matrix
```

```
##    user  system elapsed
##   38.12    0.14   39.39
```

```r
#fastest actual solving time
system.time(output3<-backsolve(U,backsolve(U,y,transpose = TRUE)))
```

```
##    user  system elapsed
##    0.03    0.00    0.06
```

a)

I think the timing lines up with about what I would think from reading the notes. Chris has mentioned in class before that cholesky is usually quicker as we can see from above. Solve with the matrix multiplication is very slow. Chol takes some time initially to set up the chol/ get U matrix, but backsolve is very quick. While chol takes longer on my laptop (probably something with the BLAS), as discussed in the notes, typically the timing goes (from longest to shortest) is approach a), approach b), and approach c). Also, approach A should be order n^3 as this approach requires doing the explicit multiplication of X by y, and this number of order of computations decreses for b and c.

b)

To look at the matching digits of accuracy in approach b) and c), the code below formats some entries of the matrices to 16 digits, which is R's precision accuracy.

```r
#comparing
identical(output2,output3)
```

```
## [1] FALSE
```

```r
#see that there must be some difference

#comparing digits that match up
#in entries of b) and c)
format(output2[1], digits=16)
```

```
## [1] "92.93266861523028"
```

```r
format(output3[1], digits=16)
```

```
## [1] "92.93266867067018"
```

```r
format(output2[100], digits=16)
```

```
## [1] "148.9921586629336"
```

```r
format(output3[100], digits=16)
```

```
## [1] "148.9921585889197"
```

```r
format(output2[1000], digits=16)
```

```
## [1] "-105.8185180662017"
```

```r
format(output3[1000], digits=16)
```

```
## [1] "-105.81851788511"
```

From notes, we know the condition number is related to the number of accurate digits by the formula $cond(A) = 10^(t - p)$, where t is the number of digits accurate in our matrix and p is 16, machine precision accuracy. Looking at the first entry and comparing the matching digits, we have 8 digits of accuracy. Then, we look at the digits of our 100th entry to match up; again, it looks we have 8 digits of accuracy. We can also look at the 1000th entry, and we get 10 digits of accuracy. To get an idea of what's happening overall in our matrices, we can run all.equal on each entry of the matrices and then look at the entries where there are different values.

```r
#seeing by how much do the values in matrices differ
answer<-c()
for (i in 1:5000){
  answer[i]<-all.equal(output2[i],output3[i])}
```

Looking at this output, we see that numbers that do differ, will differ by around 8 digits (some 7). So, our condition number is about 10ˆ8 as we do 16 (machine precision)- 8 (number of differing digits).