Stat 243 PS 5

A.J. Torre

October 14, 2018

Comments: I worked with Andrea, Norae, and Malvika on this homework assignment.

Questions

Question 1

To complete tihs problem I read the intro of unit 9 as well as looked up general linear algebra/ matrix properties. If we consider A, a square N by N matrix with N linearly independent eigenvectors. We can factorize A as $A = QVQ^{-}1$. We can decompose A this way because of the properties of eigenvectors. We know that $Ab = bv$, where v is an eigenvector of A. So, then $AQ = QV$, and thus $A = QVQ^{-}1$. Further, if A is symmetric we can use that $A = QVQ^{T}$, where Q is an orthogonal matrix whose vectors are the eigenvectors of A and V is a matrix where the diagonal is the eigenvalues of A. I got this from reading about how matrix decomposition online.

So, then we consider $AQ = QV$ by the properties of eigenvectors. Next, we have $(A - V)Q = 0$ so $det(A - V) = 0$ as Q is non-zero as it's the orthogonal matrix of eigenvectors. Since V is a diagonal matrix, its determinant is the product of its diagonal entries, which are A's eigenvalues. Therefore, $det(A)$ is the product of its eigenvalues.

Another way to see this is if $A = QVQ^{T}$, then $det(A) = det(Q)det(V)det(Q^{T})$ by the properties of determinants. Then, since Q is orthognal, $det(Q) = 1$ or $det(Q) = -1$, and also, from the properties of determinants, $det(Q^{T}) = det(Q)$.Lastly, since V is a diagonal matrix, its determinant is the product of its diagonal entries, which are the eigenvalues of A. Therefore, $det(A)$ is the product of A's eigenvalues.

Question 2

We will show some examples of the expit function in R to see how it's unstable for large values of z.

```
#expit function,
#unstable for large z values
expit<-function(z){
  exp(z)/(1+exp(z))
}

expit(10)
```

```
## [1] 0.9999546
```

```
expit(709)
```

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

```
exp(709)
```

```
## [1] 8.218407e+307
```

```
expit(710)
```

```
## [1] NaN
```

```
expit(100000)
```

```
## [1] NaN
```

So, from this, we see that the largest value that the expit function can give finite answers for is 709 by using that exp(709) is 8.218407e+307, and in class, we discussed that the max that R can handle is an exponential to the 308th power. In order to make expit numerically stable, we can divide both the numerator and denominator by exp(z). I realized that when z is very large, exp(z) and exp(z)+1 are very close to being the same number as exp(z) is so large that adding one won't make a big difference. So, we will get one (in R) when we divide exp(z) by exp(z)+1 for large values of z.

```
#expit function,
#stable for large values of z
expit2<-function(z){
  1/(exp(-z)+1)

}

expit2(10)
```

```
## [1] 0.9999546
```

```
expit2(709)
```

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

```
expit2(710)
```

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

```
expit2(100000)
```

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

So, we see that this version of the expit function is stable as it doesn't give Inf or NaN answers. We will get 1 for large values of z as explained above that exp(z)= exp(z)+1 approximately when z is large.

Question 3

The code is running the one provided in ps5 and also just seeing how to compare what x and z look like.

```
#running code from ps5 file
set.seed(1)
z<-rnorm(10,0,1)
x<-z+1e12
formatC(var(z), 20, format='f')
```

```
## [1] "0.60931443706111987346"
```

```
formatC(var(x), 20, format='f')
```

```
## [1] "0.60931216345893013386"
```

```
#to look at what z looks like
#compared to x
set.seed(1)
z<-rnorm(10,0,1)
x<-z+1e12
z
```

```
##  [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
##  [7]  0.4874291  0.7383247  0.5757814 -0.3053884
```

```
formatC(x, 20, format='f')
```

```
##  [1] "999999999999.37353515625000000000"
##  [2] "1000000000000.18359375000000000000"
##  [3] "999999999999.16442871093750000000"
##  [4] "1000000000001.59533691406250000000"
##  [5] "1000000000000.32946777343750000000"
##  [6] "999999999999.17956542968750000000"
##  [7] "1000000000000.48742675781250000000"
##  [8] "1000000000000.73828125000000000000"
##  [9] "1000000000000.57580566406250000000"
## [10] "999999999999.69458007812500000000"
```

We have discussed in class that R is accurate up to sixteen digits for floating point numbers. So, when we look at z, each number is more than 4 digits long, and when we create x, we're adding on 12 digits so R will only store up to 4 digits of our original z values in our x vector. So, when we take the variance of both, they are both accurate to a point (the 5th value) as x can store some of the original digits/ values of z, but because of R holding values to sixteen digits, x's variance in R is different than Z's.

Above, we see that when x is printed we can compare the decimal values to that of z. If a z value was negative, then we add 1e12 to it, our decimal on x is 1 plus the z value so, to compare easily, I'll just explain three examples where the z values were positive. We can compare 0.1836433 and .18359375000000000000, the second values of z and x. We see that the decimal digit in x will be correct to 4 digits if we count the fifth digit to round up, i.e. we use the 9 to round up the 5 to a 6. If we compare 1.5952808 to 1.59533691406250000000, we see here that we only get exactly 4 digits of accuracy and using the fifth digit to round won't help. Lastly, we can compare 0.4874291 and .48742675781250000000, which is correct up to 5 digits past the decimal point. We saw in class that R guarantees 16 digits of accuracy and then from there, R will try to match/ guess to the best of its ability. So, we saw from looking at three examples of comparing z to x that we can get some variance of 4 to 5 digits of accuracy, depending on R's rounding. So, while, just straight applying the we would think that the variance should only be correct to 4 decimal places (because we add on 12 digits when we create x), it looks like with rounding, R can be correct to the 5th in calculating the variance using rounding/ trying to match the remaining digits after the 16th one as seen with the previous examples.

Question 4

a) As we discussed in class, we want 1) each core to be doing similar amounts of work and 2) the cores do not have to communicate too often as loading information from one core to another can be rather slow.

When we use parallelization, we also want that it'll speed up the original process by a factor of p if we have p cores. So, by splitting up the number of columns into $m = n/p$, each core will be doing the same amount of work and the cores also will not have to communicate with each other frequently in order to complete the calculation as each core is calculating its own multiplication of a smaller submatrix.

b) In Approach A, each time we pass tasks, we are passing the full matrix X and some submatrix of Y to get a n by m matrix each time. In Approach B, each time we pass tasks, we are passing submatrices of m by p each time. So, Approach A has less tasks (order of p), but passes the full X matrix everytime, while Approach B has more tasks (order of p^2), but passes smaller submatrices of X and Y each time. Thus, Approach A has less communication but uses more memory, and Approach B uses less memory but involves more communication time. Depending on how many cores we have and how quickly each core can compute, each approach has its benefits and flaws and would be ideal in different situations.