ps7

Weijie Yuan

11/11/2018

Problem 1

Each simulation study give me an estimate of the coefficient and its standard error. So there are two way for us to estimate the uncertain of the estimated regression coefficient.

For

$$\phi = Var(\hat{\beta})$$

Method I:

$$\hat{\phi} = \frac{1}{1000}\sum_{i=1}^{1000}(\hat{\beta}_i - \hat{E}(\hat{\beta}))^2$$

where

$$\hat{E}(\hat{\beta}) = \frac{1}{1000}\sum_{i=1}^{1000}\hat{\beta}_i$$

Method II:

$$\hat{\phi} = \frac{1}{1000}\sum_{i=1}^{1000}\phi_i$$

where

$$\phi_i = sd^2(\hat{\beta}_i)$$

We can just compare these two quantities to figure out whether the statistical method properly characterizes the uncertainty of the estimated regression coefficient. If these two quantities is close, then the statistical method properly characterizes the uncertainty of the estimated regression coefficient.

Problem 2

(a)

Ordinarily, the dataset would take up about $n * p * 8bytes/1024^3 \approx 64GB$.

(b)

There are only 10000 unique combinations of the p covariates. We can store these 10000 unique combinations in binary format, which takes $10000 * 8 * 8bytes \approx 0.64MB$ to construct a $10000 * p$ matrix $X_{modified}$ to store 10000 unique combinations of the p covariates.

Besides, we need another 10^9 vector to store information about which combination of 10000 each row of original matrix is corresponding. Then, we need $10^9 * 4bytes \approx 4GB$.

The total memory use is obviously less than (a).

(c)

Because lm(), glm() functions require the complete dataframe with real number, so the vector of index can not be imported as input of 'lm' and 'glm'. So the efficient storage of unique combination in part(b) can not work in regular function like 'lm' and 'glm'.

(d)

Firstly, to calculate X^TX, instead of operating the whole X matrix, we can just take two columns from X to compute corresponding element (i,j) in X^TX.

To take advantage of my data structure, we can just construct a loop to calculate X^TX. Denote $X_{modified}$ as Xnew and index vector as ind_vec

```
for(i in 1:p){
  for(j in 1:p){
    XTX[i,j] = sum(Xnew[ind_vec, i] * Xnew[ind_vec, j])
  }
}
```

Because X^TX is symmetric, the lower triangle of it is the same as the transpose of its upper triangle part. Then, we get a $p * p$ matrix, it is easy and time-efficient to calculate the inverse of a $p * p$ matrix.

Besides, as for X^TY, using the similar method, each time, we take a single column of X.

```
for(i in 1:p){
  XTY[i] = sum(Xnew[ind_vec, i] * Y)
}
```

Finally, we get a $p * 1$ vector X^TY and a $p * p$ matrix X^TX. The last calculation is just multiplying them, which is time-efficient, needing just $p * p$ operations.

Problem 3

To compute the generalized least squares estimator, $\hat{\beta} = (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}Y$, we can solove systems of the following equations,

$$(X^T\Sigma^{-1}X)\hat{\beta} = X^T\Sigma^{-1}Y$$

We can find $\Sigma^{-1/2} = \Gamma\Lambda^{-1/2}$ by eigendecomposition.

Then using QR decomposition, let

$$\tilde{X} = \Sigma^{-1/2}X = QR$$

and

$$\tilde{Y} = \Sigma^{-1/2}Y$$

As a result, we get

$$(X^T\Sigma^{-1}X)\hat{\beta} = X^T\Sigma^{-1}Y$$

$$R\hat{\beta} = Q^T\tilde{Y}$$

The above linear algebra steps are the order of $O(n^3)$.

```
gls <- function(X, Y, sigma){
  # eigendecomposition
  e = eigen(sigma)
  eigen_value = e$values
  eigen_vector = e$vectors
  # QR
  X_modified = eigen_value^(-1/2)*eigen_vector%*%X
  X.qr = qr(X_modified)
  q = qr.Q(X.qr)
  r = qr.R(X.qr)
  Y_modified = eigen_value^(-1/2)*eigen_vector%*%Y
  # backsolve
  beta = backsolve(R,t(Q)%*%Y_modified)
}
```

Problem 4

(a)

Gaussian elimination: $\frac{1}{3}n^3 + O(n)$ (excluding subtractions).

At the mean time, we need to transform the matrix I to I^* at the right side, which means we need another $\frac{1}{2}n^3 + O(n^2)$ operations.

(b)

n backsolves: $n^2(n+1)/2$

(c)

Matrix of $n * n$ times vector of $n * 1$ takes n^2 computation.

The total cost is $\frac{4}{3}n^3 + O(n^2)$, it is obviously larger than $\frac{1}{3}n^3$ cost of what we saw in Gaussian elimination.

Problem 5

```
library(rbenchmark)
set.seed(1)
n=5000
W=matrix(rnorm(n*n),n)
X=t(W)%*%W
```

```
y=rnorm(n)
cholesky <- function(X,y){
  U = chol(X)
  b = backsolve(U, backsolve(U, y, transpose = TRUE))
  return(b)
}
benchmark(solve(X)%*%y, solve(X,y), cholesky(X,y),
          replications = 1, columns=c('test', 'elapsed', 'replications','relative'))
```

```
##             test elapsed replications relative
## 3 cholesky(X, y)  23.517            1    1.000
## 2    solve(X, y)  25.554            1    1.087
## 1 solve(X) %*% y 113.095            1    4.809
```

(a)

Using cholesky is fastest, and solving the inverse of X to compute b is slowest. It suggests that the order of speed matches our analysis.

However, the multiple of timing is somewhat deflected. According to our analysis, computing the inverse of X and then computing $X^{-1}y$ takes up $\frac{4}{3}n^3 + O(n^2)$ operations. While using cholesky takes $\frac{1}{6}n^3 + O(n^2)$ and using solve(X,y) takes $\frac{1}{3}n^3 + O(n^2)$, which does not match the relative timing showed above.

According to analysis in problem 4, theoretically, using 'solve(X)%*%y' need four times as the operations in using 'solve(X,y)' and 'solve(X,y)' need two times as the operations needed in 'cholesky(X,y)'. However, the relative timing shows that (a) takes more than four times time as that of (b). Besides, (b) and (c) takes similar time to complete. It maybe resulted from the operations of subtraction and addition and lower order operations.

(b)

```
b_b = solve(X,y)
b_c = cholesky(X,y)
```

```
sprintf("%3.20f", b_b[1])
```

```
## [1] "-157.37664324161977447147"
```

```
sprintf("%3.20f", b_b[2])
```

```
## [1] "-266.43747983830246539583"
```

```
sprintf("%3.20f", b_b[3])
```

```
## [1] "-23.64804156417822866842"
```

```
sprintf("%3.20f", b_c[1])
```

```
## [1] "-157.37664330970642367902"
```

```
sprintf("%3.20f", b_c[2])
```

```
## [1] "-266.43747983974492399284"
```

```
sprintf("%3.20f", b_c[3])
```

```
## [1] "-23.648041534140681108 06"
```

The results for b are not the same numerically for methods (b) and (c).

About 9 digits in the elements of b agree.

```
e = eigen(X)
cond = e$values[1]/e$values[5000]
cond
```

```
## [1] 391726566
```

The condition number is about 10^9, which means that the precision of x is of magnitude:

```
delta_x = cond*10^(-16)
delta_x
```

```
## [1] 3.917266e-08
```

which is very closely matched with the number of digits that agree between two results.

Note that

$$\frac{||\delta x||}{||x||} \leq cond(A) * 10^{-16}$$

where $\frac{||\delta x||}{||x||}$ in this case is of magnitude 10^{-9}.