Linear Algebra on Computers

Matthew Seguin

Preliminary Importing of Packages

```
import time
import numpy as np
import scipy as sp
```

1.

We want to solve for $\beta = (X^T\Sigma^{-1}X)^{-1}X^T\Sigma^{-1}Y$

a.

The first step is to notice we use Σ^{-1} twice so we will want to make that computation simpler. We can use a Cholesky decomposition to get $\Sigma = U^T U$ where U is upper triangular (and hence U^T is lower triangular) which is $O(n^3/6)$. Quickly note that U and hence also U^T are invertible. If say U isn't invertible there exists some $x \neq 0$ such that $Ux = 0$ but then $\Sigma x = U^T Ux = U^T 0 = 0$ contradicting the invertibility of Σ so U and U^T are indeed invertible.

Then notice that $\Sigma^{-1} = (U^T U)^{-1} = U^{-1}(U^T)^{-1} = U^{-1}(U^{-1})^T$ so that our entire model is now

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

$$= (((U^{-1})^T X)^T (U^{-1})^T X)^{-1}((U^{-1})^T X)^T (U^{-1})^T Y = (((U^T)^{-1}X)^T (U^T)^{-1}X)^{-1}((U^T)^{-1}X)^T (U^T)^{-1}Y$$

This is very messy but if we let $Z = (U^T)^{-1}Y$ and $W = (U^T)^{-1}X$ the previous system is just $\beta = (W^T W)^{-1}W^T Z$ which is just the standard OLS estimator for regressing Z on W.

So the steps we want to take are to first find the Cholesky decomposition of Σ and get U^T then solve for the inverse of U^T (which is relatively easy since it is a triangular matrix and only is $O(n^2)$) then perform $Z = (U^T)^{-1}Y$ which is $O(n^2)$ and $W = (U^T)^{-1}X$ which is $O(n^3)$ but this is done in a vectorized fashion so it can be much quicker then use OLS for the rest.

After this we can simply solve the system $W^T W\beta = W^T Z$. One could implement a QR decomposition for W but I found it faster to just perform the computation (though not by a large margin).

This is implemented in code below:

```
def gls(X, Sigma, Y):
  n = len(Y)
  # Perform cholesky composition and get U^T
  UT = np.linalg.cholesky(Sigma)
  # Invert U^T (we can solve triangular for this)
  UT_Inv = sp.linalg.solve_triangular(UT, np.identity(n))
  # Transform the data
  Z = UT_Inv @ Y
  W = UT_Inv @ X
  # Find the OLS on the transformed data
  B = np.linalg.solve(W.T @ W, W.T @ Z)
  return(B)
```

b.

Now we want to test our gls function to just computing the inverse directly and performing the calculations.

```
n = 8000
p = 225
# Make W to be turned into positive definite Sigma
W = np.random.uniform(low = 1, high = 2, size = [n, n])
Sigma = W.T @ W

# Generate regressed data
Y = np.random.normal(size = [n, 1])
X = np.random.normal(size = [n, p])

# Test the timing of my function
t0 = time.time()
B_gls_fun = gls(X, Sigma, Y)
t1 = time.time()
t1 - t0
```

```
2.572314739227295
```

```
# Test the timing of just performing the linear algebra
t0 = time.time()
Sigma_Inv = np.linalg.inv(Sigma)
Z = Sigma_Inv @ Y
Z = X.T @ Z
Inv = np.linalg.inv(X.T @ Sigma_Inv @ X)
B_linalg = Inv @ Z
t1 = time.time()
t1 - t0
```

```
3.834266424179077
```

We can see that my implementation is faster. This is consistent with what we saw in problem 1. Just solving for the inverse and performing the linear algebra is less efficient than solving systems of equations directly.

c.

Now we want to check the results from the different computations.

```
B_gls_fun[:10]
```

```
array([[-0.0201309 ],
       [ 0.00286906],
       [ 0.06025487],
       [-0.02340209],
       [-0.00263268],
       [ 0.02242805],
       [ 0.03435201],
       [ 0.00687819],
       [ 0.00840855],
       [-0.04374226]])
```

```
B_linalg[:10]
```

```
array([[ 0.03865918],
       [ 0.01610533],
       [ 0.01247227],
       [ 0.08899987],
       [ 0.08750445],
       [-0.00209594],
       [ 0.00271635],
       [-0.10935375],
       [ 0.00979319],
       [ 0.13634264]])
```

As we can see these are definitely not the same up to machine precision. They agree more or less in the magnitude of the values but definitely don't match on the values (even closely) relative to machine precision.

The reason for this becomes apparent when we look at Σ and some of the other variables used. The values in there are large so when we perform the computations the absolute error is increasing because we are working on a larger scale. Then when we perform a large number of computations at this scale the innacuracies add up hence the difference in the results of the computations.

```
Sigma
```

```
array([[18642.96044726, 18009.09375952, 17906.94558726, ...,
        18114.86123806, 17989.32677897, 17998.27877209],
       [18009.09375952, 18712.84068603, 17950.01630421, ...,
        18128.59325174, 18003.27747087, 18029.25954487],
       [17906.94558726, 17950.01630421, 18516.71542214, ...,
        18034.52370824, 17927.02659199, 17925.92745874],
       ...,
       [18114.86123806, 18128.59325174, 18034.52370824, ...,
        18896.447478  , 18101.95344908, 18124.79724315],
       [17989.32677897, 18003.27747087, 17927.02659199, ...,
        18101.95344908, 18663.56022544, 18015.66645919],
       [17998.27877209, 18029.25954487, 17925.92745874, ...,
        18124.79724315, 18015.66645919, 18678.44020436]])
```

```
sp.linalg.det(Sigma)
```

```
inf
```

Clearly the values in Σ are large, and in fact taking a look at the determinant we see it is too big for a computer to calculate even so this further supports the fact that there is a large amount of computational innacuracy.

In this particular case Sigma had large values but were it to have small values there would be a similar case.

```
np.linalg.cond(Sigma)
```

```
154480490446420.97
```

This is supported by looking at the condition number of Σ which is very large again illustrating that a small change in the inputs can cause a large change in the results. The condition number of the entire calculation is at least this since adding in more potential for variation through additional multiplications can only increase the condition number by the definition.

Directly computing the matrix inverses is likely not computationally stable so I trust the gls function I implemented more.

2.

We have two models:

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

a.

We assume that Z is a 60 million by 630 matrix, X is a 60 million by 600 matrix, and Y is a 60 million by 1 matrix. Even if we were to use techniques that could make this implementation more efficient and do this in two steps we still need to start with the matrices Z and X in memory for the first computation. Each entry is a floating point number which will take 8 bytes to store and therefore the total number of gigabytes of memory needed to store these matrices is found below.

```
B = ((6e+7)*(630) + (6e+7)*(600))*8
KB = B/1000
MB = KB/1000
GB = MB/1000
GB
```

```
590.4
```

So you would need 590 Gigabytes of memory in order to just store these matrices. Unless you have access to terrabytes of memory they will not be able to perform this calculation because they can't even store the matrices to begin with (so they have no hope of storing additional matrices needed to perform the computation).

However this is overlooking that we can save memory in storing X and Z since they are sparse. Assume that we could store them initially and that we could even do matrix multiplications with sparse matrices. We could then find $Z^T Z$ which would be a 630 by 630 matrix (so we could store that) and its inverse, then we could find $Z^T X$ which would be a 630 by 600 matrix so we could even store that. Furthermore we could even perform the computation $(Z^T Z)^{-1} Z^T X$ at this point since this is multiplying a 630 by 630 matrix with a 630 by 600 matrix. The following result would be a 630 by 600 matrix.

The problem occurs in the final s tep. Since the final product $Z(Z^T Z)^{-1} Z^T X$ (which at this point is the multiplication of a 60 million by 630 matrix and a 630 by 600 matrix) is not guaranteed to be sparse in the end even if we could perform the calculation we would need to be able to store a 60 million by 600 matrix (that is not sparse) which involves a lot of memory (calculated below).

```
B = ((6e+7)*(600))*8
KB = B/1000
MB = KB/1000
GB = MB/1000
GB
```

```
288.0
```

This would need 288 Gigabytes of memory, so again we can not perform this calculation in 2 steps.

b.

We can rewrite these two models into one equation to try and perform the calculation. First note that:

$$\hat{X}^T = \Big(Z(Z^TZ)^{-1}Z^TX\Big)^T = X^T(Z^T)^T((Z^TZ)^{-1})^TZ^T = X^TZ((Z^TZ)^T)^{-1}Z^T = X^TZ(Z^TZ)^{-1}Z^T$$

And therefore we know:

$$\hat{X}^T\hat{X} = \Big(X^TZ(Z^TZ)^{-1}Z^T\Big)\Big(Z(Z^TZ)^{-1}Z^TX\Big) = X^TZ(Z^TZ)^{-1}(Z^TZ)(Z^TZ)^{-1}Z^TX = X^TZ(Z^TZ)^{-1}Z^TX$$

Which gives us the final equation:

$$\hat{\beta} = (\hat{X}^T\hat{X})^{-1}\hat{X}^TY = (X^TZ(Z^TZ)^{-1}Z^TX)^{-1}X^TZ(Z^TZ)^{-1}Z^TY$$

$$= (X^TZ(Z^TZ)^{-1}(X^TZ)^T)^{-1}X^TZ(Z^TZ)^{-1}Z^TY$$

This looks more complicated but it is actually computationally feasible assuming we can properly store the sparse matrices X and Z and perform computations with sparse matrices.

The first step is to compute $A_1 = Z^TZ$ which again will result in a 630 by 630 matrix, and we find its inverse $A_2 = (Z^TZ)^{-1}$. Then we compute $A_3 = X^TZ$ which will result in a 600 by 630 matrix.

Now that we have each of these we perform $A_4 = X^TZ(Z^TZ)^{-1} = \Big(X^TZ\Big)\Big((Z^TZ)^{-1}\Big) = A_3A_2$ which is the multiplication of a 600 by 630 matrix with a 630 by 630 matrix and so will result in a 600 by 630 matrix.

At this point we can get rid of $A_2 = (Z^TZ)^{-1}$ in memory since we already know $A_4 = X^TZ(Z^TZ)^{-1} = A_3A_2$ and won't use A_2 anymore.

Then we perform $A_5 = Z^TY$ which will result in a 630 by 1 matrix (vector) and subsequently perform $A_6 = X^TZ(Z^TZ)^{-1}Z^TY = \Big(X^TZ(Z^TZ)^{-1}\Big)\Big(Z^TY\Big) = A_4A_5$ which will result in a 600 by 1 matrix.

Next we perform $A_6 = X^TZ(Z^TZ)^{-1}(X^TZ)^T = \Big(X^TZ(Z^TZ)^{-1}\Big)\Big((X^TZ)^T\Big) = A_4A_3$ which will be a 600 by 600 matrix.

At this point we can get rid of A_3 and A_4 in memory since we won't use them anymore. Then subsequently find the inverse $A_7 = (X^TZ(Z^TZ)^{-1}(X^TZ)^T)^{-1} = A_6^{-1}$.

Finally we can perform

$$A_8 = (X^TZ(Z^TZ)^{-1}(X^TZ)^T)^{-1}X^TZ(Z^TZ)^{-1}Z^TY$$

$$= \Big((X^TZ(Z^TZ)^{-1}(X^TZ)^T)^{-1}\Big)\Big(X^TZ(Z^TZ)^{-1}Z^TY\Big) = A_7A_6$$

Assuming memory was properly cleared (i.e. getting rid of matrices as I mentioned and not keeping matrices we only needed for their inverse after computing the inverse) we see that the highest number of (full, not including sparse) matrices in memory is when we have 4 matrices in memory (at the step where we have A_3, A_4, A_5, A_6 all in memory before we get rid of A_3 and A_4 as well as when we have A_5, A_6, A_7, A_8 all in memory before we only keep A_8).

I won't go into the exact dimensions of these even though it would be quite simple because the following works. We know that the most number of elements in any of these matrices is 630^2 since all of these matrices are at most 630 by 630.

Therefore the most memory from storing these full matrices in memory at a given time is less than

```
B = (630**2)*4*8
KB = B/1000
MB = KB/1000
MB
```

```
12.7008
```

So we won't ever use more than 12 Megabytes of memory for storing these matrices. This is based just on storing the data and doesn't take into account the overhead for having the object in python or momentary memory used for computation but clearly there is plenty of room in memory for anything we might need.

3.

First note that since A is symmetric (n by n) we can write it as $A = \Gamma\Lambda\Gamma^T$ where Γ is an n by n orthogonal matrix and Λ is an n by n diagonal matrix (with non-negative diagonal entries that are the eigenvalues of A).

First note since Λ is diagonal $\Lambda^T = \Lambda$ and $\Lambda^T\Lambda = \Lambda^2$ is still diagonal. Also since Γ is orthogonal we know $\Gamma^T = \Gamma^{-1}$.

Then we can write:

$$(Az)^TAz = z^TA^TAz = z^T(\Gamma\Lambda\Gamma^T)^T\Gamma\Lambda\Gamma^Tz = z^T\Gamma\Lambda^T\Gamma^T\Gamma\Lambda\Gamma^Tz$$

$$= z^T\Gamma\Lambda^T\Lambda\Gamma^Tz = z^T\Gamma\Lambda^2\Gamma^Tz$$

Now quickly note that if $||z||_2 = 1$ and Γ is orthogonal then if $y = \Gamma^Tz$ we know $||y||_2 = \sqrt{y^Ty} = \sqrt{(\Gamma^Tz)^T\Gamma^Tz} = \sqrt{z^T\Gamma\Gamma^Tz} = \sqrt{z^Tz} = ||z||_2 = 1$.

Therefore we can say $\sup_{z:||z||_2=1}(Az)^TAz = \sup_{z:||z||_2=1} z^T\Gamma\Lambda^2\Gamma^Tz = \sup_{y:||y||_2=1} y^T\Lambda^2y$. We can rewrite this as:

$$y^T\Lambda^2y = \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix} \begin{bmatrix} \lambda_1^2 & 0 & \dots & 0 \\ 0 & \lambda_2^2 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & \lambda_n^2 \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix} \begin{bmatrix} \lambda_1^2y_1 \\ \vdots \\ \lambda_n^2y_n \end{bmatrix} = \lambda_1^2y_1^2 + \dots + \lambda_n^2y_n^2$$

More specifically if λ_{max} is the largest of the eigenvalues in absolute value then we can write $y^T\Lambda^2y = \lambda_1^2y_1^2 + \dots + \lambda_n^2y_n^2 \leq \lambda_{max}^2y_1^2 + \dots + \lambda_{max}^2y_n^2 = \lambda_{max}^2(y_1^2 + \dots + y_n^2) = \lambda_{max}^2$. We can actually achieve this upper bound by taking all of the $y_i = 0$ except for $y_{max} = 1$ where the max subscript indicates it is the term in front of λ_{max}^2.

Then if we do that $y^T\Lambda^2y = \lambda_1^2y_1^2 + \dots + \lambda_n^2y_n^2 = \lambda_{max}^2 1^2 = \lambda_{max}^2$. Since this is an achieveable upper bound it is therefore the supremum, namely $\sup_{z:||z||_2=1}(Az)^TAz = \sup_{y:||y||_2=1} y^T\Lambda^2y = \lambda_{max}^2$. From which we directly know (since $\sqrt{x}$ is monotonically increasing) $||A||_2 = \sup_{z:||z||_2=1}\sqrt{(Az)^TAz} = \sqrt{\sup_{z:||z||_2=1}(Az)^TAz} = \sqrt{\lambda_{max}^2} = |\lambda_{max}|$ which is by construction the largest of the eigenvalues in absolute value. Hence proved.