STAT 243 Problem Set 7

Treves Li

2024-11-13

Collaboration Statement

I did not collaborate with anyone.

Question 1

We've seen how to use Gaussian elimination (i.e., the LU decomposition) to solve $Ax = b$ and that we can do the solution in $\frac{n^3}{3}$ operations (plus lower-order terms). Suppose I want to know how inefficient it is to explicitly invert the matrix A and then multiply, thereby finding $x = A^{-1}b$ via matrix-vector multiplication. If we look into the C++ code behind `numpy.linalg` at the `inv` function, we see it solves the system $AZ = I$ to find $Z = A^{-1}$, using a Lapack routine DGESV, which uses the LU decomposition. Count the number of computations for

a. transforming $AZ = I$ to $UZ = I^*$ (where I^* is no longer a diagonal matrix),
b. for solving for Z given $UZ = I^*$, and
c. for calculating $x = Zb$.

Then compare the total cost to the $\frac{n^3}{3}$ cost of what we saw in class.

For this whole question, I've assumed that A is a $n \times n$ square matrix.

Part a

This part involves decomposing A using the LU decomposition such that $A = LU$, and we know from class that this is $\frac{n^3}{3}$.

We now have $(LU)Z = I$. To find UZ is a forward-substitution problem, given that L is lower triangular. We worked out in class that a forward solve is $\frac{n^2}{2}$ (similar cost to a back solve). We need to repeat this for the n columns in I, so that the cost in this step is $n \times \frac{n^2}{2}$.

So the total cost for these two steps is:

$$\frac{n^3}{3} + \left(n \times \frac{n^2}{2}\right) = \frac{n^3}{3} + \frac{n^3}{2} = \frac{5n^3}{6}$$

Part b

Since U is also (upper) triangular, the computational cost in this part is similar to the second step of Part a, except that we are now backsolving instead of forward-solving. I.e., the cost is dominated by $\frac{n^3}{2}$.

Part c

At this point we have Z which is a $n \times n$ matrix, and b which is a $n \times 1$ vector. We know intuitively that this has computational cost of n^2.

Total cost

The total cost considering all three parts is thus:

$$\begin{aligned} &\frac{5n^3}{6} + \frac{n^3}{2} + n^2 \\ =&\frac{5n^3}{6} + \frac{3n^3}{6} + n^2 \\ \approx&\frac{4n^3}{3} \end{aligned}$$

We see then that in theory, the computational cost of using the inverse is 4 times that of using the LU decomposition (when disregarding lower-order terms).

Question 2

Suppose I need to compute the generalized least squares estimator, $\hat{\beta} = (X^\top \Sigma^{-1} X)^{-1} X^\top \Sigma^{-1} Y$, where X is $n \times p$ and Σ is a positive definite $n \times n$ matrix. Assume that $n > p$ and n could be of order several thousand and p of order in the hundreds.

Question 2a

First write out in pseudo-code how you would do this in an efficient way - i.e., the particular linear algebra steps and the order of operations. Then write efficient Python code in the form of a function, `gls()`, to do this - you can rely on the various high-level functions for matrix decompositions and solving systems of equations, but you should not use any code that already exists for doing generalized least squares. Warning: the Cholesky functions in numpy and scipy don't return the result in the same form.

Since Σ is positive-definite, we can use the Cholesky decomposition to express Σ as:

$$\Sigma = LL^\top \tag{1}$$

Additionally, we make use of the following relations, leveraging the triangular nature of L:

$$LW = X \quad \Rightarrow \quad W = L^{-1}X \tag{2}$$

$$Lz = Y \quad \Rightarrow \quad z = L^{-1}Y \tag{3}$$

We can then rewrite the GLS estimator such that we don't need to compute the inverse Σ^{-1}:

$$\begin{aligned}
\hat{\beta} &= (X^\top \Sigma^{-1} X)^{-1} && X^\top \Sigma^{-1} Y && \\
&= (X^\top (LL^\top)^{-1} X)^{-1} && X^\top (LL^\top)^{-1} Y && \text{substitute in Eq. 1} \\
&= \left[X^\top (L^\top)^{-1} L^{-1} X\right]^{-1} && X^\top (L^\top)^{-1} L^{-1} Y && \\
&= \left[(L^{-1}X)^\top (L^{-1}X)\right]^{-1} && (L^{-1}X)^\top (L^{-1}Y) && \\
&= (W^\top W)^{-1} && W^\top z && \text{substitute in Eq. 2 and 3}
\end{aligned}$$

The final line re-expresses the generalised least squares estimator in a simple OLS form, which we can solve with the LU decomposition (i.e., with a series of calls to `np.linalg.solve`).

The corresponding pseudo-code would be:

1. Apply Cholesky decomposition to Σ (using `sp.linalg.cholesky`)
2. Solve $LW = X$ for W (using `sp.linalg.solve_triangular`)
3. Solve $Lz = Y$ for z (using `sp.linalg.solve_triangular`)
4. Solve $(W^\top W)^{-1} W^\top z$ to get $\hat{\beta}$ (using `np.linalg.solve`)

And the code implementation would be:

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

def gls(X, Y, Sigma):
  """Performs GLS regression efficiently."""

  # Set the number of threads for parallel computing
  os.environ["OPENBLAS_NUM_THREADS"] = "8"
  os.environ["OMP_NUM_THREADS"] = "8"

  # 1. Apply Cholesky decomposition to Sigma (with timing)
  t0 = time.time()
  L = sp.linalg.cholesky(Sigma, lower=True)
  chol_time = time.time() - t0
  print(f"Cholesky implementation took {chol_time} s.")

  # 2. Solve LW = X for W
  W = sp.linalg.solve_triangular(L, X, lower=True)

  # 3. Solve Lz = Y for z
  z = sp.linalg.solve_triangular(L, Y, lower=True)

  # 4. Solve OLS form
  beta_est = sp.linalg.solve(W.T @ W, W.T @ z)

  # Reset num_threads to default
  del os.environ["OPENBLAS_NUM_THREADS"]
  del os.environ["OMP_NUM_THREADS"]

  return beta_est, chol_time
```

Question 2b

> Now compare your approach to directly using the inverse in Python (but make sure to only calculate the inverse once and to use an efficient order of operations). Is the timing consistent with the results of Question 1 and the computational efficiency results from Unit 10? For simplicity, you can construct a positive definite matrix Σ as $\Sigma = W^\top W$, with the elements of the $n \times n$ matrix W generated randomly. (In a real problem, Σ would be set up based on the context of course.) You can also generate X and Y randomly.

Let's test the speed of using our previously defined `gls()` function, versus using `np.linalg.inv`. Note that since $n > p$, it would be more efficient to do $\Sigma^{-1}X$ first (since X is "skinny") rather than $X^\top\Sigma^{-1}$. Similarly, it's more efficient to do $\Sigma^{-1}y$ first rather than $X^\top\Sigma^{-1}$.

```
# Generate random matrices
np.random.seed(42)

n = 5000
p = 300

X = np.random.randn(n, p)   # design matrix
Y = np.random.randn(n)      # response vector
A = np.random.randn(n, n)
Sigma = A.T @ A             # p.d. n x n matrix

# Sanity check that Sigma is p.d.
# assert np.all(np.linalg.eigvals(Sigma) > 0)

# Implementation using np.linalg.inv()
def inv_gls(X, Y, Sigma):
  """Performs GLS using inverse."""

  # Calculate the inverse once (with timing)
  t0 = time.time()
  Sigma_inv = np.linalg.inv(Sigma)
  inv_time = time.time() - t0
  print(f"Inverse implementation took {inv_time} s.")

  # Calculate feature matrix
```

```
  feature_matrix = X.T @ (Sigma_inv @ X)

  # Calculate cross-product vector
  cross_prod_vec = X.T @ (Sigma_inv @ Y)

  # Solve for beta
  beta_est = np.linalg.solve(feature_matrix, cross_prod_vec)

  return beta_est, inv_time

beta_inv, inv_time = inv_gls(X, Y, Sigma)
beta_cholesky, chol_time = gls(X, Y, Sigma)

# Compare times
speed_up = inv_time / chol_time
print(f"Cholesky speed-up: {speed_up:.3f}\n")

# Sanity check that results are the same
print(f"Inverse betas:  {beta_inv[:5]}...")
print(f"Cholesky betas: {beta_cholesky[:5]}...")
```

```
Inverse implementation took 2.413102865219116 s.
Cholesky implementation took 0.6421122550964355 s.
Cholesky speed-up: 3.758

Inverse betas:  [-0.00194635 -0.03503132  0.02302941  0.04753565 -0.04956376]...
Cholesky betas: [-0.00194635 -0.03503132  0.02302941  0.04753565 -0.04956376]...
```

We know from Question 1 that the computational complexity of the inverse is $\mathcal{O}\left(\frac{4n^3}{3}\right)$. We also know from Unit 10 that the Cholesky is $\mathcal{O}\left(\frac{n^3}{6}\right)$. So we expect the Cholesky to be 8 times as fast as doing the inverse.

However, the results above show that I was only able to get a 4x speedup at most. This is despite constraining the timing to only the factorisation steps, and also optimising the number of available threads for parallel computing (based on CPU usage from `top`, and manually adjusting the `OMP_NUM_THREADS` and `OPENBLAS_NUM_THREADS` variables accordingly).

Reasons for the reduced speedup could be that running the code in Python (as opposed to C++ or Fortran) introduces significant overhead, especially for large matrix operations. This overhead could diminish the efficiency of the underlying BLAS/LAPACK routines. Additionally, `sp.linalg.cholesky` may be performing extra checks that aren't factored into the speedup calculations, such as verifying positive definiteness (although the documentation isn't clear if this is done or not).

Question 2c

> Are the results for the solution, $\hat{\beta}$ the same numerically for the two approaches (up to machine precision)? Comment on how many digits in the elements of $\hat{\beta}$ agree, and relate this to the condition number of the calculation.

To see if the results are numerically the same, we can compute the absolute differences across the two solutions, and see if these values are greater than or less than machine precision (which is called using `np.finfo(np.float64).eps`):

```
eps = np.finfo(np.float64).eps
diff = np.abs(beta_inv - beta_cholesky)

print(np.all(diff < eps))
```

```
False
```

This shows that while the answers are close, they are not close enough numerically, since the difference between the values are greater than machine precision of 10^{-16}.

We can see how many digits match by finding the average of the absolute differences:

```
print(f"{np.mean(diff):.1e}")
```

```
1.6e-12
```

We see that the elements of $\hat{\beta}$ agree to around 12 digits on average. We can verify this by checking arbitrary elements from the two sets of $\hat{\beta}$'s, and comparing their matching digits:

```
print(beta_inv[24])
print(beta_cholesky[24])
```

```
-0.08676768601854372
-0.08676768601810798
```

We learnt in class that the condition number is a measure of how sensitive the solution is to small changes or perturbations in the input. Larger conditions represent ill-conditioned problems, and from the course notes: "a condition number of 10^8 means we lose 8 digits of accuracy relative to our original 16 on standard systems". We can compute the condition number using `np.linalg.cond` (with some modification to the number of available threads to speed up the calculation).

```
os.environ["OPENBLAS_NUM_THREADS"] = "12"
print(f"Condition number: {np.linalg.cond(Sigma):.1e}")
del os.environ["OPENBLAS_NUM_THREADS"]
```

```
Condition number: 1.7e+08
```

The result suggests that we might achieve around 8 digits of precision ("at worst", since the condition number is an upper bound). However, this does not mean we are guaranteed to lose exactly 8 digits—rather, the condition number represents the maximum possible loss in precision due to error sensitivity. The fact that we previously calculated around 12 digits of precision means that the numerical methods we are using (either Cholesky or inverse) are robust and stable enough to provide relatively precise solutions for this particular calculation, despite what the condition number might suggest. This higher-than-expected precision could be due to specific characteristics of Σ, such as a structure less sensitive to perturbations, or beneficial error cancellations during computations.

Question 3

Two-stage least squares (2SLS) is a way of implementing a causal inference method called instrumental variables that is commonly used in economics. Consider the following set of regression equations:

$$\hat{X} = Z(Z^\top Z)^{-1} Z^\top X$$

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

which can be interpreted as regressing Y on X after filtering such that we only retain variation in X that is correlated with the instrumental variable Z. An economics graduate student asked me how he could compute $\hat{\beta}$ if Z is 60 million by 630, X is 60 million by 600, and Y is 60 million by 1, but both Z and X are sparse matrices.

Question 3a

Describe briefly why I can't do this calculation in two steps as given in the equations, even if I use the techniques for OLS discussed in class for each stage.

As mentioned in the notes, the product of two sparse matrices would not be sparse in this case. So when calculating $\hat{X}$, the resulting matrix size will be large. We can estimate the memory required as follows:

$$\begin{aligned}
\hat{X} &= Z && (Z^\top Z)^{-1} && Z^\top X \\
&= (60M \times 630) && (630 \times 630) && (630 \times 600) \\
&= (60M \times 630) && (630 \times 600) \\
&= (60M \times 600)
\end{aligned}$$

This results in a dense matrix $\hat{X}$ with dimensions 60M x 600, which requires storing approximately 36 billion elements. Assuming that each element is of `float64` type (8 bytes), the memory required for $\hat{X}$ alone would be:

$$60M \times 600 \times 8 \text{ bytes} = 288 \text{ GB}$$

Given that $\hat{X}$ is dense, many systems will struggle to handle this size. If we were brave and wanted to continue and compute $\hat{\beta}$, we would also need to calculate $X^\top X$. While $X^\top X$ is a 600×600 matrix and smaller than $\hat{X}$, we would still need additional memory usage (potentially

another 288 GB) to store $X^\top$ and other intermediate dense matrices. This makes the two-step calculation impractical for typical systems.

This reasoning also explains why techniques like triangular matrix decompositions, Cholesky decomposition, or QR decomposition are not suitable for this calculation. These methods generally assume the matrices fit into memory, but the sizes involved here would result in prohibitively high memory consumption.

Question 3b

> Figure out how to rewrite the equations such that you can actually calculate $\hat{\beta}$ on a computer without using a huge amount of memory. You can assume that any matrix multiplications involving sparse matrices can be done on the computer (e.g., using `scipy.sparse`). Describe the specific steps of how you would do this and/or write out in pseudo-code.

We want to compute the matrices without making them dense during the intermediate calculations, by using `scipy.sparse`. Specifically, `lsmr` lets us solve linear systems iteratively, without needing to explicitly compute intermediate matrices like $Z^\top Z$ or $X^\top X$.

1. Turn Z and X into Compressed Sparse Row (CSR) matrix format, using `sp.csr_matrix`. This ensures that subsequent matrix operations are memory-efficient.
2. Instead of explicitly computing $Z^\top Z$, we can use `scipy.sparse.linalg.lsmr` to solve the linear system $Z^\top Z\hat{X} = Z^\top X$ iteratively. This way, we don't have to explicitly create dense matrices like $Z^\top Z$.
3. Compute the remaining part of $\hat{X}$ using the `@` operator, which is optimized for sparse matrices as stipulated in the `scipy.sparse` documentation):
4. Repeat the process from steps 2 and 3 to calculate $\hat{\beta}$, again using `scipy.sparse.linalg.lsmr` to solve $X^\top X\hat{\beta} = X^\top Y$ iteratively without creating $X^\top X$.

Using this method, we avoid creating large dense matrices, and instead make use of optimised methods in `scipy.sparse`.

Question 4

(Extra credit) In class we saw that the condition number when solving a system of equations, $Ax = b$, is the ratio of the absolute values of the largest and smallest magnitude eigenvalues of A. Show that $\|A\|_2$ (i.e., the matrix norm induced by the usual L2 vector norm; see Section 1 of Unit 10) is the largest of the absolute values of the eigenvalues of A for symmetric A. To do so, find the following quantity,

$$\|A\|_2 = \sup_{z:\|z\|_2=1} \sqrt{(Az)^\top Az}$$

If you're not familiar with the notion of the supremum (the *sup* here), just think of it as the maximum. It accounts for situations such as trying to find the maximum of the numbers in the open interval (0,1). The max is undefined in this case since there is always a number closer to 1 than any number you choose, but the *sup* in this case is 1.

Hints: when you get to having the quantity $\Gamma^\top z$ for orthogonal Γ, set $y = \Gamma^\top z$ and show that if $\|z\|_2 = 1$ then $\|y\|_2 = 1$. Finally, if you have the quantity $y^\top Dy$, think about how this can be rewritten given the form of D and think intuitively about how to maximize it if $\|y\|_2 = 1$.

First, since A is symmetric, we can factorise it using the eigenvalue decomposition:

$$A = \Gamma D \Gamma^\top$$

where

- Γ is an orthogonal matrix whose columns are the eigenvectors of A, and
- D is a diagonal matrix with the eigenvalues of A on the diagonal

We can multiply both sides by some vector z:

$$Az = \Gamma D \Gamma^\top z$$

We can also multiply both sides by $(Az)^\top$:

$$
\begin{aligned}
(Az)^\top Az &= (\Gamma D \Gamma^\top z)^\top \Gamma D \Gamma^\top z \\
&= z^\top \Gamma D^\top \Gamma^\top \Gamma D \Gamma^\top z \\
&= z^\top \Gamma D^\top D \Gamma^\top z && (\Gamma^\top \Gamma = I \text{ since } \Gamma \text{ is orthogonal}) \\
&= z^\top \Gamma D^2 \Gamma^\top z && (D^\top = D \text{ since } D \text{ is diagonal}) \\
&= (\Gamma^\top z)^\top D^2 \Gamma^\top z \\
&= y^\top D^2 y && (y = \Gamma^\top z \text{ based on the hint})
\end{aligned}
$$

We note at this point that since $\Gamma^\top$ is orthogonal, then applying $\Gamma^\top$ to z shouldn't change the length/norm of z. So if we did the substitution of $y = \Gamma^\top z$ from the hint, then the norm of y is the same as the norm of z, i.e.:

$$
\|y\|_2 = \|\Gamma^\top z\|_2 = \sqrt{(\Gamma^\top z)^\top (\Gamma^\top z)} = \sqrt{z^\top \Gamma \Gamma^\top z} = \sqrt{z^\top z} = \|z\|_2
$$

So now we have:

$$
(Az)^\top Az = y^\top D^2 y = \sum_{i=1}^{n} \lambda_i^2 y_i^2
$$

This last expression for the sum of $(Az)^\top Az$ is maximized when y aligns with the eigenvector corresponding to the largest λ_i^2 of A. Since $\|y\|_2 = 1$, we effectively want

$$
y_i^2 = \begin{cases} 1, & \text{if } \lambda_i = \lambda_{\max} \\ 0, & \text{otherwise} \end{cases}
$$

So we get

$$
\|A\|_2 = \sup_{z : \|z\|_2 = 1} \sqrt{(Az)^\top Az} = \sup_{z : \|z\|_2 = 1} \sqrt{\sum_{i=1}^{n} \lambda_i^2 y_i^2} = \sqrt{\lambda_{\max}^2} = |\lambda_{\max}| \quad \blacksquare
$$

We show then that $\|A\|_2$ is the largest of the absolute values of the eigenvalues of A.