HW4

Shengyuan Wang

2023-02-16

Homework guidelines:

- Turn in this assignment as a PDF on Moodle, please! To create a PDF, I suggest knitting to HTML, then opening the HTML file in a browser and saving to PDF from there.
- You're invited to collaborate and discuss with each other, and then each person should turn in their own assignment, which should be their own work. "Discussing" is very different from copying and I trust students to stay on the right side of this line. In general, anything you say out loud to another person is fine, and looking at a screen together (in person or on Zoom) is fine. Sharing files or screenshots is a bad idea. Name the people you work with below (question 0).
- If you start early, you're giving yourself the chance to ask questions and turn in a polished product. If you start late, you won't be able to get help as easily.

Q0 Collaborators

Who did you work with on this assignment? You'll get a bonus 5 points if you name someone and they name you. *No scores above 100% are allowed, but these bonus points can repair mistakes on this assignment*.

Name: Wenxuan / Carissa

I worked with Wenxuan and Carissa on this assignment.

Q1: Sherman-Morrison

Let v and w be vectors in $\mathbb{R}^n$ and let $A = I + wv^T$. A matrix like A is called a "rank-one perturbation of the identity." Assuming $v^T w \neq -1$, let

$$B = I - \frac{1}{1 + v^T w} wv^T.$$

In this problem your goal is to prove on paper that $B = A^{-1}$. *Hint:* B is the inverse of A if $AB = I$. Alternately, you can let z be an arbitrary vector in $\mathbb{R}^n$ and show that $ABz = z$. This problem is a special case of a famous identity known as the Sherman-Morrison formula.

$$\begin{aligned}
AB &= (I + wv^T)(I - \frac{1}{1 + v^T w} wv^T) \\
&= I^2 + wv^T I - \frac{Iwv^T}{1 + v^T w} - \frac{wv^T wv^T}{1 + v^T w} \\
&= I^2 + Iwv^T - (\frac{Iwv^T}{1 + v^T w} - \frac{I(v^T w)wv^T}{1 + v^T w}) \\
&= I^2 + Iwv^T - \frac{Iwv^T}{1 + v^T w}(1 + v^T w) \\
&= I^2 + Iwv^T - Iwv^T \\
&= I
\end{aligned}$$

Thus, we can say $B = A^{-1}$.

Q2: Derivative matrix

In this problem we'll build a 7×7 matrix that can take derivatives. We'll let any vector $c \in \mathbb{R}^7$ represent a trigonometric polynomial of degree 3. Specifically, let

$$f_c(x) = c_1 + c_2 \cos x + c_3 \sin x + c_4 \cos(2x) + c_5 \sin(2x) + c_6 \cos(3x) + c_7 \sin(3x)$$

Let's suppose c and d are two vectors in $\mathbb{R}^7$, and the corresponding functions are f_c and f_d. If the derivative of f_c is f_d, that is, $f_c'(x) = f_d(x)$, then we can find the entries of d if we know the entries of c. For example, if $c^T = (0, 2, 0, 0, 0, 0, 0)$ then we would have $d^T = (0, 0, -2, 0, 0, 0, 0)$ because the derivative of $2\cos(x)$ is $-2\sin(x)$. In particular, $Ac = d$ where A is a certain 7×7 matrix with 43 zeros. Find this matrix A. Then show that it works by creating and describing a simple example where you find the derivative of some trig polynomial both by hand and also by multiplication with A.

We can let $c^T = (a, b, c, d, e, f, g)$

$$f_c(x) = a + b\cos(x) + c\sin(x) + d\cos(2x) + e\sin(2x) + f\cos(3x) + g\sin(3x)$$

$$f_c'(x) = f_d(x) = 0 - b\sin(x) + c\cos(x) - 2d\sin(2x) + 2e\cos(2x) - 3f\sin(3x) + 3g\cos(3x)$$

Thus, we can get $d^T = (0, c, -b, 2e, -2d, 3g, -3f)$ So, the matrix A here is to rearrange the entries of c^{T} and add scalar to change into d^T.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & -3 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \end{pmatrix} = \begin{pmatrix} 0 \\ c \\ -b \\ 2e \\ -2d \\ 3g \\ -3f \end{pmatrix}$$

For example, we have $c^T = (1, 1, 1, 1, 1, 1, 1)$. If we do that by hand, we can say

$$f_c(x) = 1 + \cos(x) + \sin(x) + \cos(2x) + \sin(2x) + \cos(3x) + \sin(3x)$$

$$f_c'(x) = f_d(x) = 0 - \sin(x) + \cos(x) - 2\sin(2x) + 2\cos(2x) - 3\sin(3x) + 3\cos(3x)$$

So, we can get $d^T = (0, 1, -1, 2, -2, 3, -3)$

When we do that by multiplication with matrix A, we will get

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & -3 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ -1 \\ 2 \\ -2 \\ 3 \\ -3 \end{pmatrix}$$

Q3: Theft

The point of this problem is twofold: (i) to illustrate what can happen if you accumulate many truncations of numbers and (ii) to give you practice writing programs.

In the 1999 movie Office Space, a character creates a program that takes fractions of cents that are truncated in a bank's transactions and deposits them into a secret illegal account controlled by the hacker. This is not a new idea, and hackers who have actually attempted it have been arrested. In this exercise you will simulate the program to determine how long it would take to become a millionaire this way.

Assume the following details:

- You have access to 10,000 bank accounts
- Initially, the bank accounts have values that are uniformly distributed between $100 and $100,000
- The nominal annual interest rate on the accounts is 5%
- The daily interest rate is thus .05/365
- Interested is compounded each day and added to the accounts, except that fractions of a cent are truncated
- The truncated fractions are deposited into an illegal account that initially has a balance of $0
- The illegal account can hold fractional values and it also accrues daily interest

Your job is to write an R script that simulates this situation and *finds how long it takes for the illegal account to reach a million dollars*.

Here is some R help:

The following code generates the initial accounts:

```
accounts=runif(10000,100,100000)
accounts = floor(accounts*100)/100
```

The first line sets up 10,000 accounts with values uniformly between 100 and 100,000. Note that *runif* stands for *random uniform*, not *run if*.
The second line removes the fractions of cents (look at the data before and after that line is applied). To calculate interest for one day:

```
interest = accounts*(.05/365)
```

Initialize the value of the illegal account at 0 and then, for each day, add the fractional-cent interest truncations to the illegal account. Depending on how you do it, you might want to use an if-then statement. For example, you might use something like

```
if (illegal > 1000000) break
```

inside of a for-loop, where *illegal* is the value of your illegal account. The `break` command breaks out of a loop. Or, perhaps more elegantly, you might use a `while` loop

```
while (illegal < 1000000) { do stuff here }
```

```
accounts <- runif(10000, 100, 100000)

illegal <- 0
step <- 0
while (illegal < 1000000) {
  step <- step + 1
  before <- sum(accounts)
  accounts <- floor(accounts*100)/100
  after <- sum(accounts)
  illegal <- illegal + before - after
  interest <- accounts*(.05/365)
  accounts <- accounts + interest
  illegal_interest <- illegal*(.05/365)
  illegal <- illegal + illegal_interest
}
step
```

```
## [1] 9627
```

Q4 Timing of A = QR

Let's see how large a matrix A you could factor into QR if you had an hour of computing time on your machine. One way to do this is to check the time before and after you run the code. For example, here is how long it takes to add up the first 10^6 integers using a for-loop:

```
tstart = Sys.time()
t = 0.0
for (j in 1:1000000){
  t <- t + j
}
tend <- Sys.time()
print(tend - tstart)
```

```
## Time difference of 0.01859403 secs
```

It is much faster to create a vector and sum the entries, avoiding the for-loop! Example:

```
tstart = Sys.time()
t <- sum(1:1000000)
tend <- Sys.time()
print(tend - tstart)
```

```
## Time difference of 0.002692938 secs
```

```
my.QR <- function(A){
  # This is the numerically stable (good) version
  # Also known as "modified Gram-Schmidt" or "mgs"
  # The input A must be square and invertible
  d <- dim(A)
  n <- d[1] # the matrix A is n-by-n
  Q <- 0*A  # Start with a Q full of zeros
  R <- 0*A  # Start with an R full of zeros
  my.len <- function(v){return(sqrt(sum(v^2)))}
  for (i in 1:(n-1)){ # loop over rows of R, except the last
    R[i,i] <- my.len(A[,i])  # find diagonal entry in R
    Q[,i] <- A[,i] / R[i,i]  # find i-th column of Q
    for (j in (i+1):n){
      R[i,j] <- sum(Q[,i] * A[,j]) # fill in the row in R
      A[,j] <- A[,j] - R[i,j]*Q[,i] # modify col of A
      # Previous line is the "modified" part of the algorithm
      # We overwrite cols of A as we go.
    }
  }
  R[n,n] <- my.len(A[,n]) # get the last diagonal entry of R
  Q[,n] <- A[,n] / R[n,n] # get the last column of Q
  return(list(R=R,Q=Q))
}

ourSol <- function(n) {
  m <- matrix(rpois(25, 5), nrow = n, ncol=n)
  tstart = Sys.time()
  myQRthing <- my.QR(m)
  tend <- Sys.time()
  print(tend - tstart)
}

builtInSol <- function(n) {
  m <- matrix(rpois(25, 5), nrow = n, ncol=n)
  tstart = Sys.time()
  myQRthing <- qr(m)
  tend <- Sys.time()
  print(tend - tstart)
}
```

- Make a random 100×100 matrix A and time the factorization $A = QR$ using our code and the built-in one. *Our code is in the activity from the A=QR day in the course spreadsheet.* Which is faster?

```
# The result of factorization of matrix 100*100 using our function:
ourSol(100)
```

```
## Time difference of 0.1265011 secs
```

```
# The result of factorization of matrix 100*100 using built in function:
builtInSol(100)
```

```
## Time difference of 0.0008370876 secs
```

So, the built in solution is faster.

- Now try a 200×200 matrix and a 400×400 matrix. Use the built-in code. Complete the sentence: "When I double the dimension of A, the computation time for $A = QR$ gets multiplied by approximately a factor of 8".

```
# ------------------------------------------------------------------------
# The result of factorization of matrix 200*200 using built in function:
builtInSol(200)
```

```
## Time difference of 0.00621891 secs
```

```
# ------------------------------------------------------------------------
# The result of factorization of matrix 400*400 using built in function
builtInSol(400)
```

```
## Time difference of 0.177671 secs
```

When I double the dimension of A, the computation time for $A = QR$ gets multiplied by approximately a factor of 8

- An hour is 3600 seconds. Predict how large a matrix you can factor in one hour on your machine.

$$0.01 = c(100)^3$$

$$cn^3 = 3600$$

$$n \approx 15326$$

If use build-in function, the matrix will be 15326 * 15326

Q5 A rank-one matrix is not invertible

Let $A = \begin{pmatrix} -1 & 1 & -2 \\ 2 & -2 & 4 \\ 4 & -4 & 8 \end{pmatrix}$ and let $b = (5, -10, -12)$.

- Show that A has rank one by finding $u, v \in \mathbb{R}^3$ such that $A = uv^T$.

$$\begin{pmatrix} -1 & 1 & -2 \\ 2 & -2 & 4 \\ 4 & -4 & 8 \end{pmatrix} = \begin{pmatrix} -1 \\ 2 \\ 4 \end{pmatrix} \begin{pmatrix} 1 & -1 & 2 \end{pmatrix}$$

- Because rank-one matrices are not invertible, we know that the equation $Ax = b$ has either infinitely many solutions or none. If there is no x with $Ax = b$, explain why not. If there are infinitely many x with $Ax = b$, give two examples.

There is no x with $Ax = b$. Since A is a rank-one matrix, in order to have some solution for the equation, b have to be in the form of $c\begin{pmatrix} -1 \\ 2 \\ 4 \end{pmatrix}$. Clearly, in this question, b is not in that form, so we have no x with $Ax = b$.

Q6 : how close to “tiger”?

A “Word2vec” algorithm can produce vectors in $\mathbb{R}^{300}$ that correspond in some way to each English word. To do this, the algorithm trains on a massive data set containing many English-language text files, for example, news articles. These algorithms are notorious for carefully learning the prejudices of the authors of the training texts. Here is a recent and nuanced discussion of the issue, which you can explore if you like: link (https://arxiv.org/abs/1905.09866).

Next comes a long code listing that doesn’t actually appear in the knitted file, because of the handy “echo=FALSE” option. It defines eight vectors whose names are: *octopus, snail, lion, tiger, cat, horse, beetle, pencil.* I got these vectors by playing with the *gensim* package in Python.

Concretely, the 19th of the 300 entries in the **tiger** vector is:

```
tiger[19]
```

```
## [1] -0.2451172
```

Consider the word *tiger*. Among the other seven, I think *lion* is semantically closest, followed by *cat*. Then the next few are harder for me to rank. I might say that *horse* comes next because it’s large and lives on land, then *octopus* which is large but aquatic. *Snail* and *beetle* are living things but otherwise quite far from being tigers. Finally, I think *pencil* is completely unrelated to *tiger*. The R Markdown formatting lets me rank them in a table like this:

Word	Similarity to “tiger,” in my opinion
lion	very close
cat	close
horse	kind of close
octopus	kind of far
snail	far
beetle	far
pencil	very far

But that’s subjective; it’s informed by my own life experiences and priorities! For example, someone else might argue that horses eat plants, so they should be lower than octopuses in the table.

The word2vec embedding allows us to explore the distances between these words in a quantitative way, *within the flawed word2vec language model*. A typical way to think about the distance between two words is from the dot product. The Cauchy-Schwartz inequality says that $|v^T w| < \|v\|_2 \|w\|_2$, or for nonzero vectors, $\left|\frac{v^T w}{\|v\|\|w\|}\right| \leq 1$. The $\leq$ becomes $=$ only when v is a multiple of w, that is, when they point in the same direction.

Thus, if we compute $\frac{v^T w}{\|v\|\|w\|}$ and get 1, then v and w are very similar; if we get -1 then they are opposites; and if we get 0 they are totally unrelated. Use this idea to rank the other seven words by their similarity to **tiger**. List the words and the similarity scores in a table.

Just to ponder, not part of the problem: An alternative to this dot-product method would to say that v and w are close if $\frac{|v-w|_\infty}{|v|_\infty}$ is close to zero, or maybe if $\frac{|v-w|_1}{|v|_1}$ is close to zero, or maybe if $\frac{|v-w|_2}{|v|_2}$ is close to zero.

```
library(pracma)
check <- function(word1, word2) {
  vTw = t(word1)%*%word2
  v_value = Norm(t(word1), 2)
  w_value = Norm(t(word2), 2)
  print(vTw/(v_value*w_value))
}
```

```
# lion
check(tiger, lion)
```

```
##           [,1]
## [1,] 0.5121041
```

```
check(tiger, cat)
```

```
##           [,1]
## [1,] 0.5172962
```

```
check(tiger, horse)
```

```
##           [,1]
## [1,] 0.2876559
```

```
check(tiger, octopus)
```

```
##           [,1]
## [1,] 0.2439948
```

```
check(tiger, snail)
```

```
##           [,1]
## [1,] 0.2244429
```

```
check(tiger, beetle)
```

```
##           [,1]
## [1,] 0.3287175
```

```
check(tiger, pencil)
```

```
##           [,1]
## [1,] 0.1348347
```

—————|—————
cat | 0.5172962
lion | 0.5121041
horse | 0.2876559
octopus | 0.2439948
snail | 0.2244429
beetle | 0.3287175
pencil | 0.1348347