Randomized Low Rank Approximations

10 points

We have learned about the SVD as one method to obtain a low-rank approximation for a matrix. Low-rank approximation is useful in many settings, for example:

- Dimensionality reduction in Machine learning (http://scikit-learn.org/stable/index.html)
- Image processing (/course/cs450-s19/file-version/dd9a1d98076c43c7f314f5733421d849d1acad38/demos/upload/linear_least_squares/Image%20compression.html)
- Graph analytics (https://research.facebook.com/blog/fast-randomized-svd/)

Unfortunately, the SVD is a fairly expensive way to obtain a low-rank approximation, and so there is an incentive to try to obtain a less accurate answer at a smaller cost. Randomized methods, which you'll learn about in this problem, provide a way of doing this.

Given an $n \times n$ matrix A, we wish to find a $n \times k$ matrix Q, such that:

1. k is much smaller than n
2. $Q^T Q = I$ (where I is $k \times k$).
3. $A \approx QQ^T A$.

The last point means that A projected onto the column span of Q is approximately A itself. In other words, the columns of Q "almost" span the range of A.

Knowing this basis is the core to a low-rank approximation. But it can also help bootstrap to bigger and better things. For example, knowing Q we can compute the SVD of $Q^T A = \tilde{U}\Sigma V^T$ (which is much smaller and thus cheaper than A!) and $A \approx Q\tilde{U}\Sigma V^T = U\Sigma V^T$ is an approximate singular value decomposition of A.

But, how do we find Q? We use the following algorithm.

Algorithm: Randomized Adaptive Range Finding (click to view)

1. Create an $n \times \ell$ matrix Ω using normally distributed random numbers. We should have $\ell < n$. We will take $\ell = 2$.
2. The most basic step is to then let $Y = A\Omega$, which gives us ℓ columns in the range of A.
3. Compute the QR factorization $Y = \tilde{Q}R$. This gives an orthonormal sampling of the range of A. Take $Q = \tilde{Q}$.
4. Is Q good enough? Check the Frobenius norm (https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm) of $A - QQ^T A$. If it's smaller than some tolerance (which we give you), stop the algorithm. (We use the Frobenius norm because the 2-norm involves computing the SVD... which we are trying to avoid!).
5. If that Q was not good enough, we need to keep going. Repeat steps 1 and 2, (with a new Ω) to obtain a new Y. This is another sampling of the range of A.
6. Next, we want to orthogonalize the columns of Y against the previously computed columns of Q, in order to build more orthogonal basis vectors for the range of A. Theoretically, we should just be able to replace Y with $Y - QQ^T Y$ (why does this work?) However, we lose orthogonality pretty quickly. So after doing this, we compute the QR factorization $Y = \tilde{Q}R$ and overwrite Y with $\tilde{Q}$, so that we have an orthonormal basis for the range of Y. Then we subtract off $QQ^T Y$ again.
7. Compute the QR factorization $Y = \tilde{Q}R$ (yes, another one!). Replace Q with $[Q, \tilde{Q}]$. (Essentially, stack Q and $\tilde{Q}$ together horizontally). This gives you a new Q, and a larger orthonormal approximation to the range of A.
8. Repeat steps 4 through 7 until the Frobenius norm of $A - QQ^T A$ is sufficiently small.

9. In step 2, experiment with applying a few rounds of the power method by using $Y = (AA^T)^p A\Omega$ for $p \in \{0, \dots, 3\}$ instead of $Y = A\Omega$. ($(AA^T)^p A$ has the same singular values as A.) To avoid the problems of the power method, you may additionally want to apply a round of QR after each application of A or A^T, and you may want to orthognalize Y against Q before you start power iteration. Does the behavior of the algorithm change? Why? Does it improve or deteriorate? Record your observations for each value of p in a print statement in your code.

Implement the above algorithm using a matrix `A` and a tolerance `tol` that we will provide. Once your algorithm has converged, save your matrix Q as `Q`. The autograder will test whether $\|A - QQ^T A\|_F$ is within tolerance.

Note: There is one more caveat. Since this is a randomized algorithm, there is a *chance* it will fail. One option is to test that $\|Q^T Q - I\|$ is small after every iteration (under the Frobenius norm). If it gets too big, raise a runtime error. E.g: (raiseRuntimeError("Q doesn't have orthonormal columns")). If you don't do this, expect your code to time out once in a while.

Inputs: `A, tol`

Output: `Q`

Problem set-up code (click to view)

This is the code that is used to set up the problem and produce test data for your submission. It is reproduced here for your information, or if you would like to run your submission outside of . You should *not* copy/paste this code into the code box below. This code is run automatically 'behind the scenes' before your submitted code.

```
import numpy as np
import numpy.linalg as la

n = 100
#np.random.seed(15)
A0 = np.random.randn(n, n)
U0, sigma0, VT0 = la.svd(A0)

lin = np.linspace(0, 1, n)
decay = 1-(1-np.sin(np.pi/2*lin))**2
sigma = 10**(-6*decay)

A = (U0 * sigma).dot(VT0)

tol = 1e-5
```

Testing code (click to view)

This is the code that is used to generate feedback for your submission. It is reproduced here for your information, or if you would like to run your submission outside of . You should *not* copy/paste this code into the code box below. This code is run automatically 'behind the scenes' after your submitted code.

```
if not isinstance(Q, np.ndarray):
    feedback.finish(0, "Q is not a numpy array")
if Q.dtype.kind != "f":
    feedback.finish(0, "Q is not a floating point array")
if len(Q.shape) != 2:
    feedback.finish(0, "Q is not 2-dimensional")
if Q.shape[0] != n:
    feedback.finish(0, "Q's first dimension (row count) is incorrect")
if Q.shape[1] == n:
    feedback.finish(0, "Q's second dimension (column count) may not match that of the full matri

num_rank = np.sum(np.abs(sigma > tol))
print(f"target rank: {num_rank} found rank: {Q.shape[1]}")

orth_err = Q.T @ Q - np.eye(Q.shape[1])
orth_err_norm = la.norm(orth_err, "fro")
if orth_err_norm > 1e-10:
    feedback.finish(0, "Q's columns are not orthogonal to sufficient accuracy")

approx_err = A - Q @ Q.T @ A
approx_err_norm = la.norm(approx_err, ord="fro")
if approx_err_norm > 10*tol:
    feedback.finish(0, "Q's columns do not approximate A's range sufficiently well")

feedback.finish(1, "All results look good, good job!")
```

Starter code (click to view)

Answer*

```
import numpy as np
import numpy.linalg as la
# initialize Q as an array with column axis of length zero, to be filled in
Q = np.zeros((A.shape[0], 0))

#get proper Y
Fnorm = 100
Omega = np.random.rand(A.shape[0],2)
Y = A@Omega
Qtilde,R = la.qr(Y)
Q = Qtilde
Fnorm = la.norm(A-Q@Q.T@A,ord='fro')
if Fnorm < tol:
    pass
else:
    while Fnorm > tol:
        Omega2 = np.random.rand(A.shape[0],2)
        Y2 = A@Omega2
        Y2 = Y2 - Q@Q.T@Y2
        Qq,Rr = la.qr(Y2)
        Y2 = Qq.copy()
        Y2 = Y2 - Q@Q.T@Y2
```

Press F9 to toggle full-screen mode. Set editor mode in user profile (/profile/).

Overall grade

The overall grade is 50%.

Autograder feedback

The autograder assigned 5/5 points.

Your answer is correct.

Here is some feedback on your code:

- All results look good, good job!
- Execution time: 0.6 s -- Time limit: 15.0 s

Your code ran on relate-04.cs.illinois.edu.

Human feedback

The human grader assigned 0/5 points.

-5 points: did not address followup questions

Your code printed the following output:

```
target rank: 40 found rank: 76
```

The following code is a valid answer:

```
import numpy as np
import numpy.linalg as la
import matplotlib.pyplot as plt

def find_range(A, tol, power_rounds=1, chunk_size=2):
    n = A.shape[0]
    Q = np.zeros((n, 0))

    while True:
        Omega = np.random.randn(n, chunk_size)
        Y = A @ Omega

        err_est = la.norm(Q @ (Q.T @ A) - A, ord = 'fro')
        print(err_est)
        if err_est < tol:
            return Q

        for i in range(power_rounds):
            # orthogonalize against prior
            Y -= Q @ (Q.T @ Y)

            Y = A.T @ Y
            Y, _ = la.qr(Y)
            Y = A @ Y
            Y, _ = la.qr(Y)

        for i in range(2):
            # orthogonalize against prior
            Y -= Q @ Q.T @ Y

            # orthogonalize self
            Y, _ = la.qr(Y)

        Q = np.hstack([Q, Y])

        orth_err = Q.T @ Q - np.eye(Q.shape[1])
        orth_err_norm = la.norm(orth_err, "fro")
        if orth_err_norm > 1e-10:
            plt.imshow(np.log10(1e-15 + np.abs(orth_err)), interpolation="nearest")
            plt.colorbar()
            plt.show()
            raise RuntimeError("lost orthogonality: %g" % orth_err_norm)

Q = find_range(A, tol)

print("p=0:")
print("Original method, considerably overshoots rank.")
print("p=1:")
print("Improved approximation of rank. Occasional loss of orthogonality without prior orthogonal
print("p=2:")
print("Loss of orthogonality without prior orthogonalization. Slight improvement of rank approxi
print("p=3:")
print("Loss of orthogonality without prior orthogonalization. Slight improvement of rank approxi
```