Computational Methods for Finding Eigenvectors and Singular Vectors

Power and QR Methods

Madeleine Hagar

21-241 Matrices and Linear Transformations

Final Project

Contents

1 Introduction

The method we learned in class to compute eigenvectors and singular vectors (by finding the roots of the matrix's characteristic polynomial), while effective, is extremely slow. Two more practical, and more importantly, faster methods to compute eigenvectors and singular vectors are the Power Method and QR Method. For my project, I chose to implement both the Power Method and QR Method to find leading eigenvalues, singular values, eigenvectors, and singular vectors of real matrices. Note that my implementations are only equipped to perform these methods on diagonalizable matrices with all real eigenvalues. To implement the Power Method, I wrote functions to compute the leading eigenvalues and leading singular values of some input matrix, as well as the leading eigenvectors and leading left and right singular vectors (for eigenvalues and eigenvectors the input matrix must be square). I then extended this implementation by using deflation to compute the "ith" eigenvalues, singular values, eigenvectors, and left and right singular vectors. To Implement the QR Method, I wrote functions to compute the leading eigenvalues and leading singular values, as well as the leading eigenvectors and leading left and right singular vectors.

2 The Power Method

The Power Method for an $n \times n$ matrix A involves picking some vector $u_0 \in \mathbb{R}^n$ and repeatedly multiplying it by A and normalizing the resultant vector until convergence. Note that we normalize the vectors to prevent the numbers from overflowing. We can denote $u_k = Au_{k-1} = A^k u_0$ as the vector obtained after the k^{th} iteration of the Power Method. As the iterations continue, the leading eigenvalue of A begins to dominate, and eventually the vector obtained by computing Au_k will converge to the leading eigenvector of A. This is because since A is diagonalizable it has n independent eigenvectors that form a basis for $\mathbb{R}^n$; thus, any vector u_0 that we pick will be some linear combination of these eigenvectors. So when we repeatedly multiply u_0 by A, we are multiplying and normalizing a linear combination of eigenvectors. Since we require that $|\lambda_1| > |\lambda_2| >= ... >= |\lambda_n|$ and normalize after each iteration, the leading eigenvector will dominate, and the components of u_0 that correspond to non-leading eigenvectors will trend towards zero, leaving just the lead eigenvector.

Pseudo-code for the Power Method is as follows:

```
for k = 1, 2, ...
   u_(k-1) = u_k
   u_k = Au_(k-1)
   u_k = u_k / ||u_k||
until convergence.
```

For any eigenvector x, $Ax = \lambda x$. Thus given A and x, we can compute the corresponding eigenvalue by computing $x^T Ax / x^T x = \lambda$ (note that here x^T denotes the transpose of vector x). This is the method I used in my function leadEigen, which returns a tuple (x, λ) where x is input matrix's lead eigenvector and λ is the corresponding lead eigenvalue. The function leadEigen takes in an $n \times n$ matrix A, computes a random vector u_0 in $\mathbb{R}^n$, and multiplies u_0 by A to obtain u_1. For the remainder of the function we use the variables u_{prev} and u_{curr} to denote u_{k-1} and u_k at the kth iteration. So initially, $u_{\text{prev}} = u_0$ and $u_{\text{curr}} = u_1/\text{norm}(u_1)$ where norm(u_1) is the magnitude of u_1 therefore normalizing u_{curr}. Then we enter the while loop in which u_{prev} is set

to equal u_{curr}, and u_{curr} is set to equal Au_{prev}. Then u_{curr} is once again normalized to prevent overflow. My implementation acknowledges the fact that the Power Method at times can only compute approximations of eigenvectors and eigenvalues, thus I decided to have an iteration variable i which is initially set to 0 and increases by 1 after each iteration and causes the while loop to halt once i $= 1000$. This is simply to prevent infinite iteration in the case where our results must be approximated. The other condition in my while loop checks for convergence, thus causing the while loop to halt once u_{curr} is equal to u_{prev}. Once we exit the while loop, we consider u_{curr} as our eigenvector and use the method described in the previously to compute the corresponding eigenvalue. I extended this method to compute all eigenvectors and eigenvalues by using deflation. Deflation involves computing a matrix A's lead eigenvector x and eigenvalue λ and computing $B = A - \lambda xx^T$, and then finding its lead eigenvector and eigenvalue; these correspond to the second eigenvector and eigenvalue of A. To compute the ith eigenvector and eigenvalue of A, one simply must repeat this process i times. This is what my function ithEigen does. The function ithEigen takes in two parameters, an $n \times n$ matrix A and an integer i. It first finds the leading eigenvector and eigenvalue using leadEigen, and then enters a while loop that iterates i times, and during each iteration, deflates A as described above and finds the resulting matrices leading eigenvector and eigenvalue. In the next iteration, it uses these values to deflate A once again. Once we exit the loop, we have computed the ith eigenvector and eigenvalue of the original A. I use these exact same processes to compute the left and right singular vectors (both leading and ith) of input matrix A, except that they are performed on A^TA to compute left singular vectors and AA^T for right singular vectors. Additionally when computing Singular vectors and values A can be $m \times n$. Singular values are computed by finding the eigenvalues (leading or ith) of A^TA or AA^T and then taking their square root, since singular values are the roots of eigenvalues, which we discussed in class. My functions leadLeftSing and leadRightSing perform similarly to leadEigen, while ithLeftSing and ithRightSing correspond closely to ithEigen.

3 The QR Method

The QR method involves computing the QR factorization of some input matrix A_0, where Q is an orthogonal matrix whose columns form an orthonormal basis for the columns of A and R is an upper triangular matrix whose entries are determined by $R_{ij} = q_i^T a_j$, and then computing the matrix A_1 by reversing Q and R to get $A_1 = RQ$ and repeating the process until the resultant matrix A_k has converged to be upper triangular. This upper triangular matrix has its eigenvalues along the diagonal, and these eigenvalues are the same as those of the original input matrix A.

We can observe that the eigenvalues remain unchanged since:

$$
\begin{aligned}
&A_k = R_k Q_k \\
&\implies A_k = Q_k^T Q_k R_k Q_k \text{...since } Q_k \text{ orthogonal} \\
&\implies A_k = Q_k^T A_{k-1} Q_k \text{...by def} A_{k-1} \\
&\implies A_k \text{similar to} A_{k-1} \text{...by def similarity} \\
&\implies A_k \text{has same eigenvalues as} A_{k-1} \text{...as we proved in class}
\end{aligned}
$$

Pseudo-code for the QR method is as follows:

```
while(A not upper triangular)
  factor A into Q and R
  A= RQ
```

This gives us all the eigenvalues of A. We can compute singular values by taking the roots of the eigenvalues of A^TAor AA^T. The function QREVal takes in an $n \times n$ matrix A and returns its leading eigenvalue. It first checks whether A is upper triangular, using the helper function isUpperTri. This function takes in an $n \times n$ matrix A and returns true if all the values below the diagonal are approximately zero (it treats extremely small decimals as zeroes to account for approximation and to cut down on runtime). If A is upper triangular, then its eigenvalues appear on its diagonal so QREVal returns the top left entry of A. If A is not uppertriangular, then we use the built in qr function to compute the QR decomposition of A. Then we compute matrix B by multiplying RQ, and pass this matrix back into QREVal. This process continues until the matrix we pass in is upper triangular and we return. The function QRithEVal takes in an $n \times n$ matrix$A$and an integer i and returns the ith eigenvalue of A. It works exactly the same as QREVal except instead of returning $A_{(1,1)}$ it returns $A_{(i,i)}$. The function QREValMat takes in an $n \times n$ matrix A and returns the upper triangular matrix that A converges to, thus allowing one to read all the eigenvalues of A directly off the returned matrix's diagonal. I added a function QRDet that takes in a matrix A, computes QREValMat(A) and then takes the product of all the diagonal entries of the returned matrix in order to compute the determinant of A(since the determinant of a matrix is the product of its eigenvalues). Once again the singular values are computed almost indentically except we instead find the eigenvalues of A^TA or AA^T and take their roots.

To compute eigenvectors using the QR method we can keep track of each Q_i throughout our iterations until A converges to be uppertriangular, and compute their product to achieve the Q_k in this decomposition $A_k = Q_k^T A_{k-1} Q_k$. The matrix Q_k is an orthogonal matrix, and the first column converges to become the lead eigenvector of A. Unless A is symmetric, the rest of the columns of Q will not be eigenvectors. Once again, this process is the same for computing singular vectors but rather performed on A^TA for left singular vectors and AA^T for right singular vectors.

The function QAcc takes in an $n \times n$ matrix A and orthogonal matrix Q and performs the same computations as QREVal, except the function tracks and accumulates the product of the Q's (denoted Q_{curr}) at each step, and returns the resulting matrix Q_k once the matrix A converges to be upper triangular. This function is used as a helper in the function QRLeadEvec, which takes in an $n \times n$ matrix A, computes its QR decomposition, and the returns the first column (normalized) of the matrix returned by QAcc(RQ, Q).

The functions QRLSingVec and QRRSingVec follow almost the exact same process but instead compute the left and right singular vectors, respectively.

Lastly, I extended the QRMethod for producing eigenvalues by implementing shifting using the Raleigh Quotient shift. This involves computing the QR factorization of A, taking the last column of Q (denote it by vector q) and computing shift $= q^TAq/q^Tq$. Then we "shift" A by computing $A-$ shift(I) (where I is the identity matrix) and then finding its QR factorization. Then we produce matrix $B = RQ$ and add the shift back on to get $B = RQ+$ shift(I), and repeat the process until our matrix converges to be upper triangular. The shifting process serves the purpose of speeding up evaluation as it causes the values below our diagonal to trend towards zero more quickly, thus causing our matrix to reach upper triangular form faster.

My functions all worked as expected, although many of them did have to approximate results. You can find test cases below most of my function definitions. I found studying these methods for computing eigenvectors and singular vectors and their corresponding values to be super interesting. I think its fascinating how much

work and research has been done on these topics and how to optimize our ability to compute them. The Power Method is amazing in its simplicity and effectiveness, especially since performing deflation on this method is super simple and fairly fast. The QR method is extremely impressive and caused me to really appreciate the properties of triangular and symmetric matrices.

4 Sources

- Strang Chapter 11.3 (p. 528-529)
- Foundations of Data Science by Blum, Hopcroft, and Kannan, Chapter 3.7
- http://pi.math.cornell.edu/~web6140/TopTenAlgorithms/QRalgorithm.html.
- urlhttps://services.math.duke.edu/ jtwong/math361-2019/lectures/Lec10eigenvalues.pdf
- Matrix Computations by Golub and Van Loan, e.g. Chapter 7
- https://dspace.mit.edu/bitstream/handle/1721.1/75282/18-335j-fall-2006/contents/lecture-notes/lec16.pdf