Math 241: Final Project Options

For each of the following project options, you will be expected to do the following things:

1. Read some of the linked material, and do any other necessary research, to understand what this topic is about, and the algorithms that might be applied to solve the question indicated.
2. Write some code, as described in each individual project, to perform the associated algorithm. In many cases this will involve you making some choices about parameters. Your code will be read by a human, so as long as the choices you make are reasonable and mathematically justified, they should all be fine.
3. Write a short paper explaining what the algorithm is used for and how/why it works.

You can work either by yourself or with one partner. If you have an algorithm using Linear Algebra concepts that we have discussed in this class that is NOT on this list that you might like to explore, please let me know! I may approve a different project if you have a well thought-out idea.

Your project will be evaluated on the following criteria:

1. Coding portion:
   (a) Does your code compile when it is given appropriate data? Does it output something appropriately?
   (b) Does your algorithm do what it purports to do? Do the choices you have made and parameters you have set make sense in context?
   (c) Is your code readable and well commented? Can someone look at it and understand what you are doing, and how?
2. Written portion:
   (a) Is your paper clear and readable? Do you demonstrate a strong understanding of the mathematical concepts in play?
   (b) Do you provide a complete, concise explanation of what your chosen algorithm does, and the circumstances under which it might be used?
   (c) Do you provide a complete, concise explanation of why your chosen algorithm works?
   (d) Have you provided a bibliography that shows the research you have used to build your algorithm and understand it?

1 Spectral Clustering

As we have seen with k-means, the question of how to divide data into clusters is one that is still of interest to the computer science and data science community. For this project, you will look at some ideas in how to find clusters among data sets using eigenvalues and eigenvectors to help.
There are many algorithms for computing these kinds of clusters. Most of them work in a way sort of like the following:

- First, put a graph on your data. Typically, this graph is some kind of similarity graph, where two data points will be connected by an edge for which the weight of the edge is higher if the data points are more similar, and is lower if the data points are more dissimilar. There are many ways to build such a graph.

- Next, build the adjacency matrix or normalized Laplacian matrix for this graph. If you don't know what the Laplacian is, don't worry! It's in the reading. But essentially it is a way to normalize the adjacency matrix so that each row/column (corresponding to each vertex of the graph, aka data point) is the same weight.

- Now the fun begins.... project your data onto the vector space generated by the eigenvectors corresponding to the k largest eigenvalues of the matrix you have built. Why to do this: essentially by building a similarity graph on the data, you've oriented each point in space, where in each dimension it's indicating "closeness" to another data point. But that gives every data point its OWN dimension in space, which is, well, ludicrous. So we can reduce dimensionality, as with PCA, to a good approximation of that existing n-dimensional embedding.

- Once we've reduced to a k-dimensional space via projection, we can treat each row of the projection matrix as our "new" location for the corresponding data point. We can then cluster these points using a typical k-means type algorithm, or any other preferred clustering algorithm that you might uncover in your reading.

As you can see, there are a number of steps here, but the fundamental structure of this type of algorithm is as follows: data in, clusters out. In between, we use a little graph theory and a bunch of linear algebra to help us decide how to make clusters.
For this project, you will write an function called `cluster` that should take in a set of data and output a set of clusters, as follows:

- Input: A list `input_data` of lists, and a positive integer `k`. Each list here represents a data point. For example, if your data represented people, and showed three characteristics, say, male/female represented by a 0/1, height in inches, and weight in lb, then each list in `input_data` would have a form such as `[0.0, 60.0, 125.0]` or `[1.0, 73.2, 153.2]`. You shouldn't expect to know *a priori* how long the lists are. You can assume that the data on these lists will be represented as floats.

- Output: A list, of the same length as `input_data`, each of whose entries are an integer between `0` and `k-1`. This list should be full of integers that indicate the clusters, so that if the i^{th} element of `input_data` were assigned to cluster number j, then the output would have a j in the i^{th} position.

In between, you can use any kind of spectral clustering algorithm you prefer. The specifics of the algorithm you choose will be explained in your paper. Your paper should also address the mathematics underlying this idea, and why it gives a reasonable clustering of the data. Depending on your choices, you may get different answers than someone else who uses a different set of choice for a spectral clustering algorithm: that is ok!
Some recommended resources for understanding spectral clustering:

- A Tutorial on Spectral Clustering by Ulrike von Luxburg
- Spectral Clustering for Beginners by Amine Aoullay
- Spectral Clustering: A Quick Overview by Charles H Martin

2 Power Method

As we have seen in class, singular values and eigenvalues can be very useful tools to understanding a matrix and how it works. Unfortunately, the problem of precisely computing the singular values of an $n \times n$ matrix A is of order n^3, which is, well, a lot. Consider, for example, the case of performing a PCA on a large image with $n \times n$ pixels. The value of n can rise pretty quickly.
For this project, you will explore a common numerical method for approximating the singular values of a matrix called the Power Method. This method is at the core of many other, more sophisticated numerical techniques for approximating singular values. The basic structure is as follows.

- First, recall that the singular values of a matrix A can be computed as the square roots of the eigenvalues of A^TA. Hence, we could calculate the first singular value by taking a limit of $(A^TA)^k$, and noting that the first singular value/vector dominates that limit. Unfortunately, taking a limit is a difficult thing for a computer to do. Hence, we can approximate that limit by simply taking A^TA to a reasonably high power, and using that to estimate the limit.

- Starting with a random vector $\mathbf{x}$, we can consider $(A^TA)^k\mathbf{x}$. Since the limit converges to the largest singular value/vector pair, we should have that $(A^TA)^k\mathbf{x} \approx \sigma_1^{2k}c_1\mathbf{v}_1$, where σ_1 is the largest singular value, and $\mathbf{v}_1$ is its corresponding singular vector. We can then normalize this result to give us an approximate value for $\mathbf{v}_1$.

- Once you have the first singular value/vector, you can subtract this part off from the matrix and proceed to find the next. That is, repeat the above procedure for the matrix $A^TA - \sigma_1\mathbf{u}_1\mathbf{v}_1^T$.

Obviously this kind of method works best when the matrix is reasonably sparse, so that matrix multiplication can be simplified to a lower order algorithm. If you would like to implement a sparse matrix product for this project, that is your prerogative.
For this project, you will write a function called `power_method` that takes a matrix and approximates the first k singular values using the power method.

- Input: A matrix `input_matrix`, as a numpy array, and a positive integer `k`.

- Output: A list of lists `sing_decomp`, where the i^{th} element of `sing_decomp` takes the form `[sigma_i, v_i, u_i]`, where `sigma_i` is the approximate i^{th} singular value of A, and `v_i, u_i` are the approximate i^{th} left and right singular values corresponding to `sigma_i`.

Some recommended resources for understanding power method:

- Foundations of Data Science: Section 3.7 by Avrim Blum, John Hopcroft, and Ravindran Kannan

- Singular Value Decomposition by Edo Liberty (the bit about power method appears at the end of this document)

3 PageRank, Markov Chains

For this project, you will be building an algorithm to rank states in a Markov chain based on importance. This is similar to a Google PageRank algorithm, using random walks to estimate the importance of a website.
The structure of this kind of method looks something as follows:

- We begin with a Markov chain on a finite state space. You can think of this is a list of websites, with the links that go between them begin indicated.
- From here, we generate a matrix P that indicates the probabilities of moving from one site to another. That is, the ij entry of the matrix is $\mathbb{P}(X_n = i \mid X_{n-1} = j)$, the chance of moving from the j^{th} element to the i^{th}. You should be able to verify that if you multiply P by a vector indicating your current position (represented as the probability you are in a given state) you should get a vector representing the probabilities of landing in each of the states after one random step.
- Using the Perron-Frobenius Theorem, we can observe that the largest eigenvalue of this matrix, even up to absolute value, is 1, and it has a positive eigenvector. All the other eigenvalues will have absolute value less than 1.
- Now, suppose we begin our random walk at some vector $\mathbf{x}$. If we write $\mathbf{x}$ as a linear combination of eigenvectors and consider the limit $\lim_{n\to\infty} P^n\mathbf{x}$, notice what happens to the eigenvalues... only 1 does not go to 0. (Specifics, of course, can be found in the reading!)
- Hence, the limit of the random walk is always the same. We use this as our ranking vector, and rank nodes higher if we are more likely to visit them on our walk, in the limit.

For this project, you will write a function called `rank` that will take in a Markov chain and output a set of rankings, as follows:

- Input: A Markov chain represented in the following way. You will be provided with a list of lists `links`, whose elements are integers, representing links between elements. The list `links[i]` represents all links emanating from node `i`. That is to say, if `links[2]=[0,1,4]`, it indicates that node 2 is linked to nodes 0, 1, and 4, and hence we could travel from 2 to each of these in our random walk.
- Output: A list `ranks` that gives the ranking of each element in the original list. Of course, in a traditional pagerank algorithm, you would only return some of these, namely those corresponding to nodes that matched your keyword search, but for our purposes ranking all websites is fine. For example, if there are 4 nodes in the original list, your output might take the form `ranks=[2, 1, 3, 0]`, indicating that the most important node is 2, and the least important is 0.

Some recommended resources for understanding PageRank and Markov chains:

- How Google works: Markov chains and eigenvalues by Christiane Rousseau.
- Notes on PageRank Algorithm by Kenneth Shum
- PageRank Algorithm: The Mathematics of Google Search by Raluca Tanase and Remus Radu