Gil Landau
10/1/18
ESE 545

Project 1 Report

Part 1:

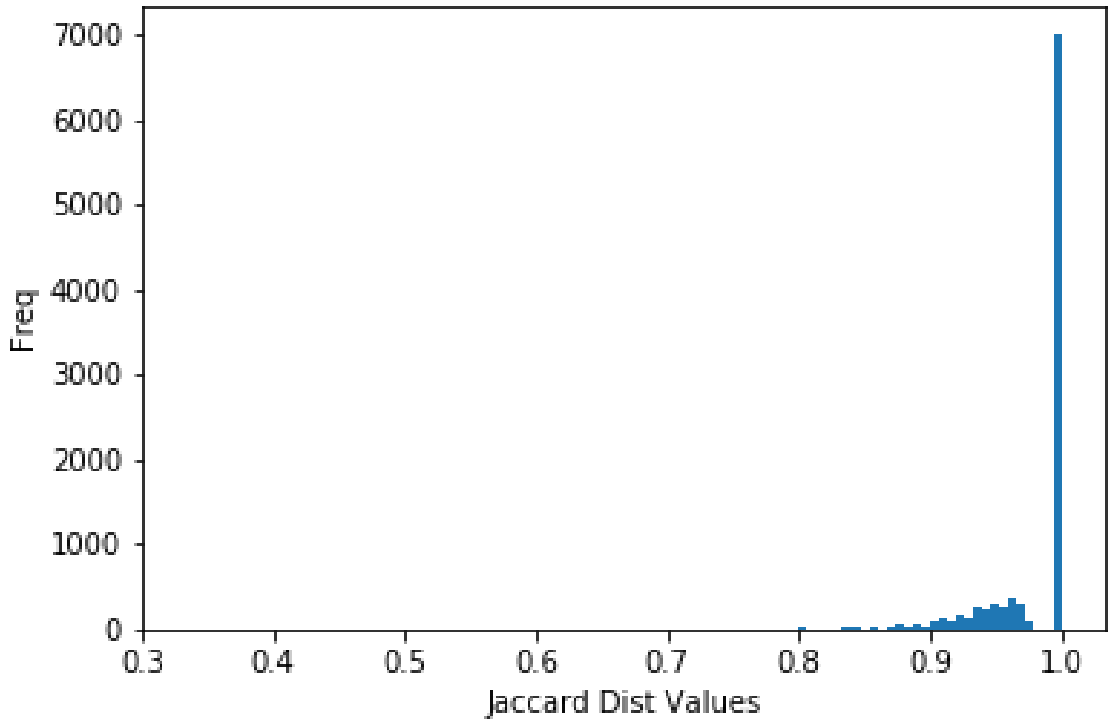
This part was relatively simple. I parse the given dataset to find all users that gave between 1 and 20 movies a rating of 3 or higher. I then do a second pass of the dataset, only adding users to the Numpy matrix that meet the correct criteria. The result is a matrix of size 4499x231424. 4499 movies and 231,424 users.

The relevant code is labelled as "Part 1" in the python file.

Part 2:

For this part, I implemented the Jaccard distance calculation. I get the size of the intersection by doing a *bitwise-and* operation on the two vectors and summing the result. I get the size of the union by summing each vector (getting the "size" of each vector's set) and then subtracting the si ze of the intersection. The Jaccard distance is simply $1 - \frac{|A \cap B|}{|A \cup B|}$. Applying this on a randomly sele cted sample of 10,000 pairs, I find that the **minimum Jaccard distance is 0.333333** and that the **average Jaccard distance is .9788542.**

The relevant code is labelled as "Part 2" in the python file. You can find the histogram of pairwise Jaccard distances below. As you can see, the large majority of distances are **.9 or above**, meaning that any given pair of users have vastly different movie tastes.



Part 3:

The solution is to do Locality-Sensitive Hashing. I do this, first, by converting the large matrix into a much more compact signature matrix. I generate 700 unique hash functions of the form **ax+b mod p**, where p is 8017 (a very large prime number, which minimizes the number of collisions) and (a,b) are a pair of integers between 0 and p-1. The variable x is simply the row number where a given column in the matrix has a 1.

I then take the minimum of $h_i(r)$ and the current value at *Sig(i,c)*, where *i* is the current row of the signature matrix and *Sig(i,c)* is the value in the signature matrix at row *i* and column *c* (initialized to infinity). I set that as the new *Sig(i,c)*. The result is a matrix of size 700x231424. I chose 700 hash functions because I knew that for the banding phase, I would want $(\frac{1}{b})^{\frac{1}{r}}$ = .65, where .65 is the similarity threshold (equivalent to .35 maximum similarity distance) and

$b \times r = n$, where n is the number of hash functions. $(\frac{1}{70})^{\frac{1}{10}}$ = .6539, which is very close to threshold.

The relevant code is labelled as "Part 3" in the python file.

Part 4:

This is the real meat of the assignment. The goal here is to use the signature matrix to generate potential candidates for further analysis, balancing both space and compute time. Unfortunately, my compute time is relatively high (about 45 minutes to run), which impacted my decision to use 70 bands of 10 rows, rather than 100 bands of 10 rows. In addition, I realized that each additional band gave me diminishing returns and increased computation time.

The crux of this generation process is to hash each row in this matrix with $h_r(SIG[r,:])$, where r is the number of rows in the band, and we then sum up each hashed value in the band for each user, in order to minimize collisions. This generates a list of potential candidates, where for each band, if any two users have the same sum for that band, then they are a potential candidate pair. We then explicitly calculate their Jaccard Distance (# of difference / # of hash functions) in the signature matrix, and if it is less than or equal to .35, we add it to our final list.

Below you can find the data I used to make my decisions, such as the number of hash functions to use, the number of bandings to use, and the number of rows in each band to use. All in all, I found roughly 1.05 million pairs and my approach was definitely more conservative.

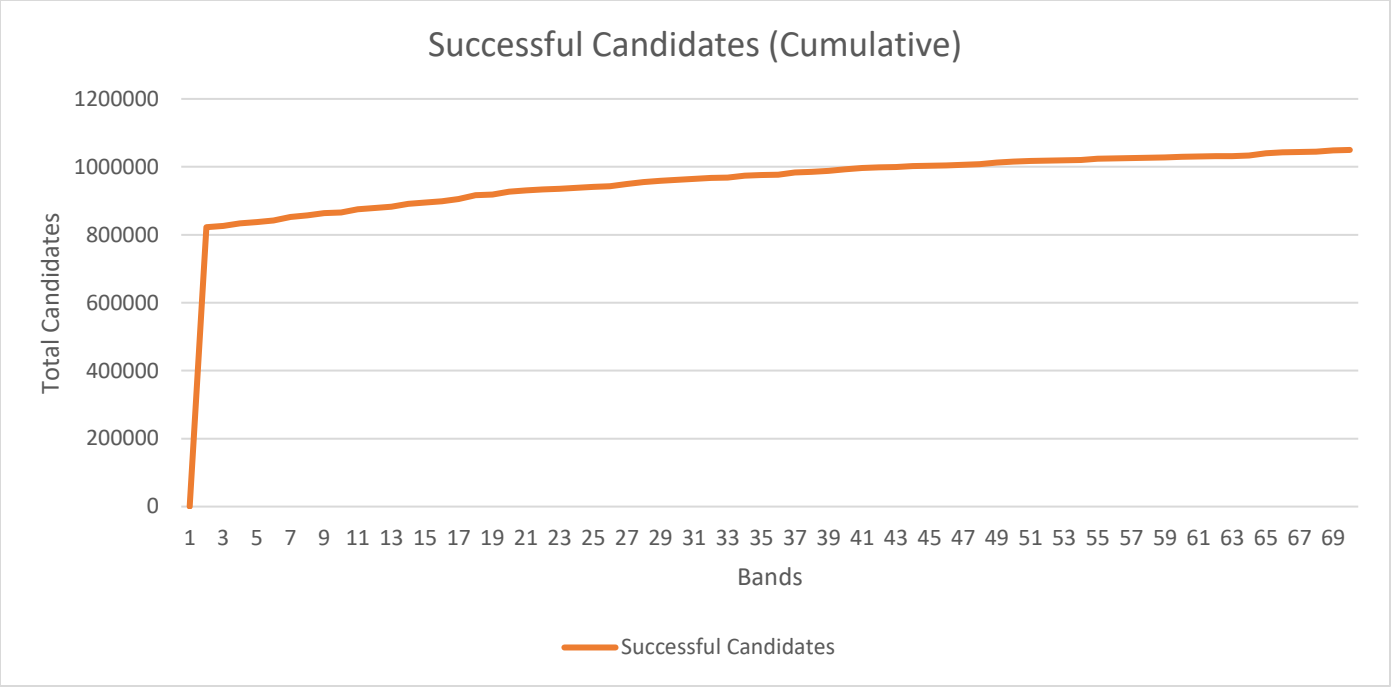


Figure 1- Total Candidates that meet similarity threshold

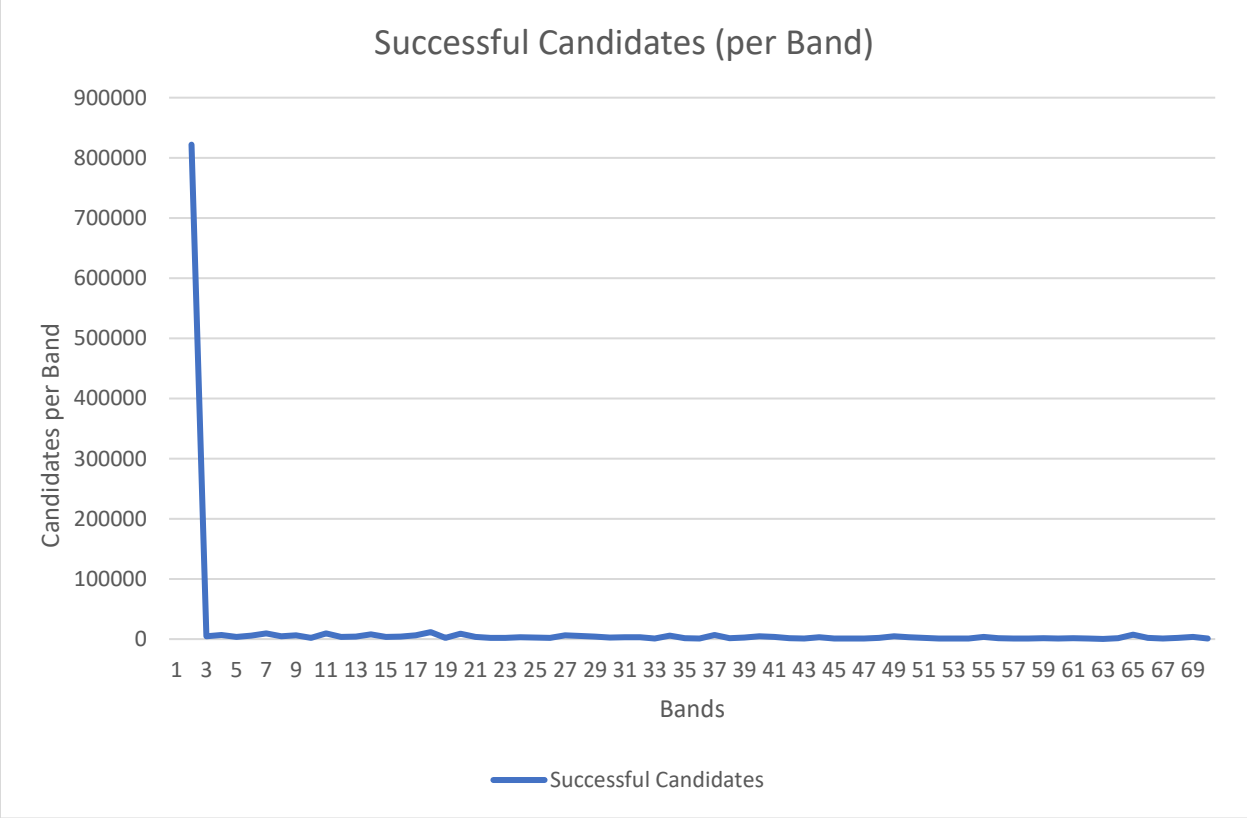


Figure 2- Total Candidates that meet similarity threshold, generated per band

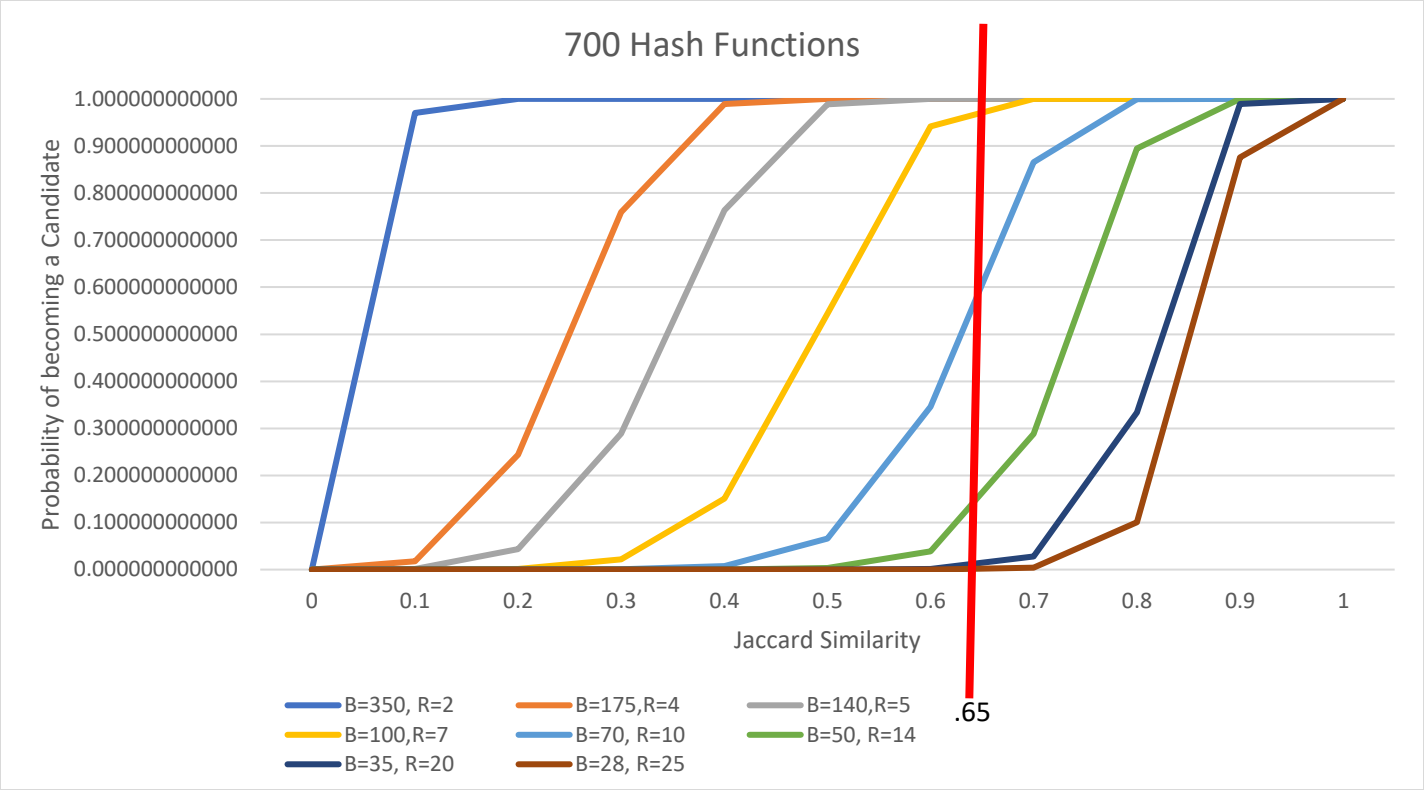


Figure 3- Choosing the optimal B and R for 700 Hashes

The relevant code is labelled as "Part 4" in the python file. You can find the pairs in the file titled similarPairs.csv

Part 5:

For this part, I created a function called user_query(), which takes in a list of indices that a user liked as an argument. It returns a set of "close" users (users with a Jaccard distance of less than .35). If no "close" users exist, the function returns a set of the closest users.

I tested three different approaches to this query: unaltered comparison, signature comparison, and banding comparison.

For the unaltered comparison, I simply calculated the Jaccard distance on the original, 4499x231000 matrix between the new user and every other user. This could take up to a minute and was very inefficient.

The signature comparison, where I calculate a new 700x231000 matrix and compare the new user with every other user, took at most a couple of seconds (if no user is found). I found that this was the most efficient approach.

The banding comparison, where I take an approach laid out in Part 4, was inefficient, mainly because setting up the banding takes longer than simply comparing the new user with the users in the signature matrix.

The relevant code is labelled as “Part 5” in the python file.