Assignment 2 - Implementing Locality Sensitive Hashing - AIDM

SHUBHAM BHATT AND JASMIN KAREEM
s3287467 AND s3357937

November 2021

1 Introduction

In this assignment we worked with a cleaned version of the original Netflix Challenge dataset. In the cleaned dataset, the number of users was reduced, from around 500.000, in order to remove users that only rated a few movies and hence wouldn't help when calculating similarities between users. What was left was users who rated at least 300 and at most 3000 movies. In addition, the user and movie ids were renumbered such that there are no gaps. The resulting dataset is a list of 65.225.506 records with user-ids, movie-ids and ratings.

The goal of this assignment is to test out 3 different similarity measures, jaccard similarity, cosine similarity and discrete-cosine similarity, in order to obtain a list of pairs of users that are similar to each other in the way that they rate or don't rate movies. We achieve this by creating a signature matrix using either random permutations or min-hashing and applying this to the locality sensitive hashing algorithm to obtain candidate user pairs to input into the chosen similarity measure.

2 Methods

In this section we will describe the various methods we used to achieve similar user pairs using jaccard, cosine and discrete-cosine similarity.

2.1 Creating the signature matrix

To create the signature matrix we first needed to convert the dataset into a sparse matrix. In our case, we decided to make our sparse matrix using the scipy package for creating a Compressed Sparse Column or CSC matrix. In the case of using jaccard as a similarity measure, we also needed to convert all the ratings in the CSC to binary values. This is because for jaccard, what matters is if the movie was rated, rather than what the rating was. Hence, movies that were rated by a user get a value of 1 and movies that did not get rated by the user get a value of 0. By contrast, for cosine and discrete-cosine similarity measures, we will just be using the CSC without any other changes.

After this step, we created a *make_signature* function, which takes the sparse matrix, a random seed, the similarity method and a chosen number of hash functions n as input. The output of this function is then the corresponding signature matrix.

In order to create the signature matrix when using jaccard similarity, we must use min-hashing. This process starts by taking n random user-ids that are distinct and using these random lists to generate n hashed values using the hash function: $h(x) = ax + b \ mod \ \hat{n}_{users}$, where $\hat{n}_{users}$ is the nearest prime number rounded upwards to the number of users in the dataset. Following this, we collect these hashed values and reshape it into a hashed matrix where its' shape is given as (n, n_{users}). We then initialize an empty signature matrix of the same size and use the obtained hashed matrix to fill in the values in the signature matrix.

The process is slightly different when using the cosine and discrete-cosine similarity methods. Instead of min-hashing to create the signature matrix, we use random permutations. We begin this process, by randomly permuting the movie-ids n times, where n is a chosen number of permutations. Once this step is done, all that is left to do is initialize an empty signature matrix and use the permutations to fill it in.

2.2 Locality-sensitive Hashing (LSH)

Once we have a signature matrix ready, we need to find a list of candidate user pairs that we can use to calculate the similarity measure on. We created an LSH algorithm that takes the signature matrix and b bands as input, with the output being a list of potentially similar pairs of users. To elaborate further, we split the signature matrix into b bands with r rows where $b * r \approx n$. Next we create empty buckets and fill these buckets using entries based on which band number the element is in. Hashing users into buckets in this way has the beneficial property that if two objects are similar, they are far more likely to end up in the same bucket of a hash function than if they are not.

Once we have all the users placed in buckets, we can start to create pairs. If a bucket only has one user in it, then we do not add it to a pair. In cases where the size of the bucket is larger than or equal to 2, we make pairs based on all possible combinations of the users in the bucket. What we are left with is a list of candidate pairs of users.

2.3 Similarity measures

For jaccard similarity, the formula is quite straight forward. Where U_1 and U_2 are ratings lists for users 1 and 2 respectively in the format (m_1, m_4, m_2) (where m_i is the movie-id), the jaccard similarity between these two users simply is:

$$J(U_1, U_2) = \frac{|U_1 \cap U_2|}{|U_1 \cup U_2|}$$

For cosine similarity, the inputs U_1 and U_2 vary slightly, with the format being $(r_1, r_2, r_3, ..., r_i)$ where r_i is the rating for movie i. Then the cosine similarity is given by:

$$cos(U_1, U_2) = 1 - \frac{arc(\theta)}{\pi}$$

where $\theta = \frac{U_1 \cdot U_2}{||U_1|| * ||U_2||}$

Discrete-cosine similarity is essentially the same as cosine similarity, however, input user-rating lists U_1 and U_2 are converted into binary values.

In our case, when the number of candidate pairs is high, it may take a really long time to compute the similarity between two users. Therefore, we added a pre-selection of candidate users before applying the chosen similarity measure. For jaccard, we added an if-statement that only selects the candidate pairs if at least 50 percent of the corresponding sigantures of those two candidate users are the same. In this way, we do not need to calculate similarity for every candidate pair, only pairs that meet this threshold. We did the same for cosine and discrete-cosine, but changed the threshold to 70 percent, instead of 50 percent. We did this because cosine and discrete cosine give many more candidate pairs for some of the same values of b, when compared to jaccard.

3 Results

In this section we will discuss the results that we found from the three different similarity measures. In all tests we did to find similar pairs, we only used the value $n = 100$, since we were primarily interested in the bands b and due to the time limit, we did not get the chance to run everything again on different values of n hash functions or permutations. We were also not interested in testing different seeds so we kept the random seed constant throughout our experiments and set it to 123.

3.1 Jaccard Similarity (JS)

We considered a pair similar if their jaccard similarity was larger than 0.5. In our first test, which can be seen in figure 1, we tried out various values of bands b but saw that a value of $b = 15$ was optimal. Using this, we found about 2 million candidate pairs. Of those candidates we found 282 similar pairs and this took about 45 minutes. We also tried other values of b. For instance, when we tried larger values of b like 20 or 33, we found that the number of candidates grew, and even-though there were more candidates, due to the fact that we would have to iterate over more candidates, it would take a lot of time to complete the entire run (see figure 1 (Left)). Therefore, at first, we tried out 5 different values of b to see what the best number of bands is and how long that might take. We tried $b = [1, 3, 5, 10, 15]$. For the first 3 values, we found no similar pairs since there were no candidate pairs. For $b = 10$ we found 39 similar pairs in about 6 minutes.

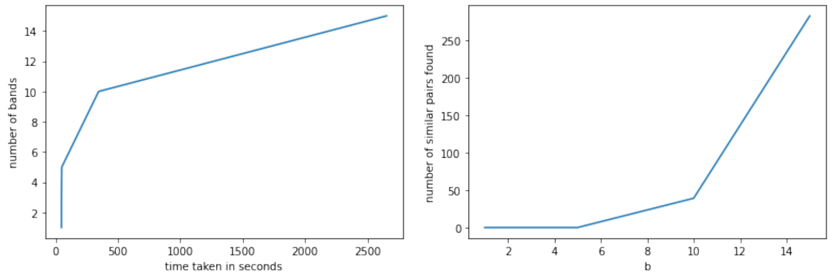

Figure 1: (Left) Number of bands versus total time taken using using the jaccard similarity measure. (Right) Number of similar pairs found for different band numbers using the jaccard similarity measure (total runtime).

We ran our code again on each value of b including $b = 20$, however, this time we capped the running time to 20 minutes because we wanted to test as many values of b for all the similarity measures and all these runs would take time. Running the experiment again on $b = [1, 3, 10, 15, 20]$, we found that the number of candidates were 0, 0, 11054, 4873380, and 20085898, respectively.

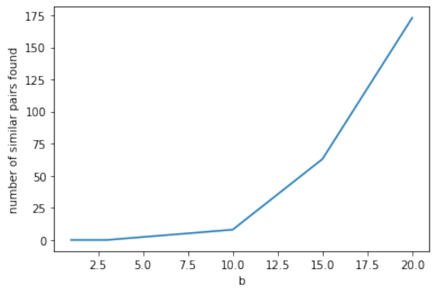

Figure 2: Number of similar pairs found for different band numbers using the jaccard similarity measure (capped at 20 minutes)

In figure 2, we see that even though we capped the running time to 20 minutes, as b increases so does the number of similar pairs found. However, compared to the unlimited running time in the right image of figure 1, we see that at $b = 15$, the unlimited version finds over 250 pairs, whilst the time-limited version only finds 63.

3.2 Cosine Similarity

Due to the number of candidates being so high, running the cosine similarity on all candidates would take hours. Therefore, we added a time limit of 20 minutes for running the similarity measure. Again, we tested out different values of $b = [1, 3, 10, 15, 20]$. Similarly to jaccard, we made a threshold to check if a pair is similar. In this case, a pair needed to have at least 0.67 as a similarity score to be considered similar.

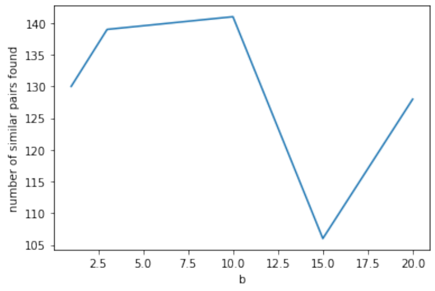

Figure 3: Number of similar pairs found for different band numbers using the cosine similarity measure (capped at 20 minutes)

The number of candidates found for each of these values of b were 203203, 203203, 211702, 2030781, and 19139694 respectively. Looking at figure 3, we see that the test with $b = 10$ yields the highest number of similar pairs. This is interesting because it shows that a higher number of bands does not necessarily result in a higher number of similar pairs found. For a 20 minute run-time, it seems that 10 bands is optimal. However, we did not check values of b higher than 20 yet, and this trend may vary depending on the chosen random seed, number of permutations and how long you let the similarity measures run. In figure 3 there seems to be a bounce back at $b = 20$. This might suggest that if we increased the number of bands, there may be higher counts of similar pairs.

3.3 Discrete Cosine Similarity

We used exactly the same set-up for discrete-cosine as we did in the cosine similarity, so we capped the time to run at 20 minutes and used the same values of b and used the same threshold. The number of candidate pair users was also the same as cosine similarity, which makes sense since they use the same methods to find the signature matrix and follow the same procedure to find candidate pairs.

In figure 4, surprisingly we find that the number of bands that gives the highest number of candidates is $b = 1$. There is even a downwards trend as b

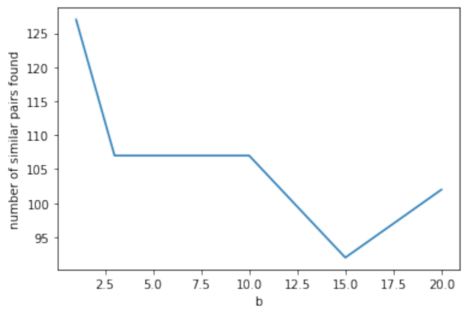

Figure 4: Number of similar pairs found for different band numbers using the discrete-cosine similarity measure (capped at 20 minutes)

increases. However, once again there is a slight uptick at $b = 20$, which could suggest that if we were to increase b to let's say 25 or 33, we could get higher numbers of similar pairs. But that's not certain since more pairs may lead to longer running time and if we stop the run after a certain amount of time, this may actually lead to less pairs.