Universiteit Leiden
The Netherlands

Advances in Data Mining

Assignment 2
Implementing Locality Sensitive Hashing

Authors:	*Student number:*
Freek van Geffen	s2633256
Justin Kraaijenbrink	s2577984

Teacher: Dr. W.J. Kowalczyk

November 15, 2020

Introduction

Since the invention of the World Wide Web by Tim Berners-Lee in 1989, the developments in the production of data have been extremely impressive. Especially the last couple of years show an explosive rise in data production. Nowadays, on a regular 24-hour interval, every person creates around 1.7MB of data *each second* (!). [1] These data are produced from several sources: social media such as Facebook, Twitter, and Instagram, regular internet usage, travel data, production of videos, online communication, transaction data and of course the smart devices at people's homes that are connected to the internet, such as televisions, fridges and thermostats.

All these data carry several advantages as well as disadvantages. For example, information about the products that people like and dislike can be highly valuable for webshops, because they are able to make personalized recommendations about products that people might be interested in. Similarly, video streaming companies such as Netflix benefit from personalized recommendations as well, because their assortment is huge and people can find it hard to choose movies and series they like. One way in which recommendations are often done is to find so-called 'similar' users. Say users A and B are in some sense similar, we can suggest to user A some of the movies that are liked by user B and not yet seen by user A. Although the availability of data makes such algorithms more accurate, it also suffers from heavy computational load because there are so many users and movies available.

This assignment will focus on implementing so-called Locality Sensitive Hashing to find pairs of most similar users of Netflix. We will investigate the performance for several ways of measuring similarity by using data from the original Netflix Challenge (`www.netflixprize.com`). We are trying to find as much as possible similar pairs without running an algorithm longer than 30 minutes.

This report is organized as follows. First we will discuss the data, the study design and the different similarity measures. Subsequently, the results will be shown. Finally, the results will be discussed and a conclusion is given.

Methodology

Netflix Challenge Dataset

Originally the dataset contained around half a million users. To make the dataset more workable, it was somewhat reduced by eliminating users that rated less than 300 movies or more than 3000 movies. This resulted in 65,225,506 records of user-movie-rating pairs. The preprocessed dataset used in this report has been retrieved from `https://drive.google.com/file/d/1Fqcyu9g6DZyYK_1qmjEgD1LlGD7Wfs5G/view?usp=sharing`. The reduced Netflix Challenge dataset contains 17,770 movies and 103,703 users. Table 1 shows that on average an user rated 630 movies, the least rated movie was rated only twice, and the most rated movie was rated 89.173 times.

Analyzing the distribution of the given ratings in Figure 1, shows that the least given rating was one. The most given rating was four, followed by three, and five.

[1] `https://techjury.net/blog/how-much-data-is-created-every-day/#gref`

Table 1: Descriptive statistics about the number of ratings given per user and the number of ratings received per movie.

	Mean	Std	min	25%	50%	75%	max
User	628.96	353.62	300	385	514	747	2997
Movie	3670.54	9682.38	2	104	345	1952	89173

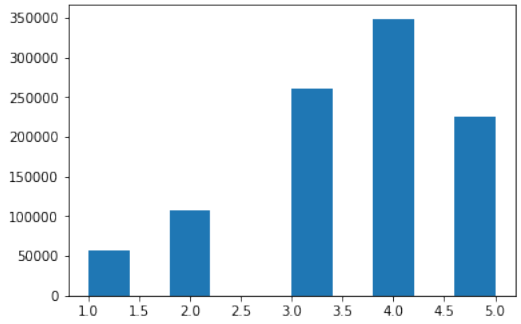

Figure 1: Histogram of the given ratings.

Study Design

In this research, the main interest is in finding a set of most similar Netflix user pairs without exceeding a computing time over 30 minutes. Such a set is found by implementing the Locality Sensitive Hashing (LSH) algorithm in combination with three different similarity measures: Jaccard similarity, cosine similarity and discrete cosine similarity. An overview of these three measures is provided in the sections below. A high-level overview of the LSH algorithm is presented in Algorithm 1, followed by more elaborate descriptions of the different steps.

Algorithm 1 LSH algorithm

1: Create sparse movie×user matrix U
2: Create signature matrix based on U using either minhashing or random projections
3: Use the signature matrix to perform the actual LSH, where users are placed in buckets
4: Obtain the set of most similar users

Step 1. This first steps converts the user-movie-rating representation of the data to a matrix U where rows represent movies and columns represent users. Since not every user has rated each movie, this matrix is highly sparse. The different similarity measures require different representations of the ratings. For Jaccard and discrete similarity, matrix U contains two different numbers: one when a user has rated a movie, and zero otherwise. For cosine similarity, matrix U contains the actual ratings.

Step 2. Matrix U is converted to a dense matrix with signatures. These signatures are a different representations of the data, so that we can determine similarity more efficiently. For Jaccard similarity, we created the signatures using minhashing, whereas for (discrete) cosine similarity we used random projections:

- **minhashing** Figure 2 presents the general idea of minhashing. The colored columns

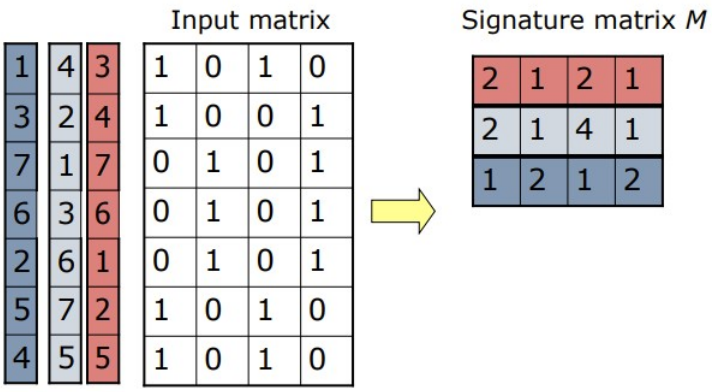

Figure 2: Schematic presentation of the minhashing procedure. *Retrieved from the lecture slides on similarity search.*

on the right are random permutations of the row numbers of input matrix U. Say we pick the red permutation. Now, for each column in U, we look at the first 'row' of the permutation for which U is nonzero, starting at 1. So in the permutation we go to the row with a 1, which is row 5 in this case. In matrix U we go to row 5, see that it is zero, so we go back to the permutation and look at the row with 2 (row 6). Row 6 in U is nonzero, so our signature matrix contains a 2 for the first column and first permutation. This process is repeated for all columns (i.e., users) and all permutations (say $n = 100$).

- **random projections** This procedure creates so-called *sketches* that are stored in the signature matrix. The procedure is as follows: we first produce n random vectors containing only -1 and $+1$ of length n_{movies}. Say we take the first random vector. We compute the dot product of this vector with the first each column in matrix U. If the outcome of the dot product is smaller than 0, the first entry (row 1, column 1) of the signature matrix gets a -1, otherwise it will get a $+1$. This process is repeated for all random vectors to obtain the first column of the signature matrix. The sketches of all other users are obtained in similar way, and by applying some clever matrix multiplications we can obtain the complete signature matrix pretty fast.

Step 3. Now that we have a more dense representation of the data, we perform the actual LSH to place users in so-called buckets. For this we use the banding technique. The signature matrix is divided in several bands of r rows. We decided that the size of the bands is equal to the largest integer not greater than the rows of the signatures matrix divided by the number of r rows. This results in equally sized bands, all having exactly r rows. All users for who the entire band is similar are placed in the same bucket. Once two or more users are in the same bucket, they are considered candidate pairs since they have at least partially the same signature. In order to avoid duplicates in the buckets, we have made use of Python's built-in function `set()`.

Step 4. The final step in obtaining the set of similar pairs is to apply the actual similarity measures - Jaccard, cosine or discrete cosine - on the set of candidate pairs. Two important

notes are in place here. First of all, for Jaccard similarity we included an additional filtering step to reduce the number of candidate pairs. This step was included to lower the computational load, which was not necessary for (discrete) cosine similarity. The filtering step used so-called sign similarity, which is simply the percentage of equal entries of two vectors. When two users have a sign similarity that exceeds a particular threshold they go for candidate pair, and subsequently the Jaccard similarity is calculated. When this Jaccard similarity exceeds the threshold of 0.5, the pair is added to the set of most similar parts.

The second important note concerns (discrete) cosine similarity, since we make use of an important theorem to speed up the algorithm considerably:

Theorem 1 *$P[h(u_1) = h(u_2)] = CS(u_1, u_2)$, where $h(u_1), h(u_2)$ are the hashed user vectors and CS is the cosine similarity. In other words, the probability that two hashed user vectors are exactly the same converges to the true cosine similarity of the user vectors.*

Note that this probability is exactly what the sign similarity represents! For the (discrete) cosine similarity, we therefore use the sign similarity of the candidate pairs as measure for the cosine similarity. When a candidate pair exceeds the threshold of 0.73 for sign similarity, it is considered a similar pair and thereby added to the set of most similar pairs. As a final step, to validate the obtained results, we computed for all most similar pairs the true (discrete) cosine similarity on the sparse matrix to see whether Theorem 1 holds.

Similarity measures

In this section we provide an overview of the different similarity measures that have been investigated for finding similar Netflix user pairs.

Jaccard similarity

The Jaccard similarity is often used as similarity measure to measure the similarity between two objects:

$$JS(u_1, u_2) = \frac{S_i \cap S_j}{S_i \cup S_j} \tag{1}$$

where u_i and u_j represent user i and j and S_i and S_j denote sets of movies that were rated by user i and user j.

Cosine similarity

This technique can be used to measure the similarity between two vectors u_1 and u_2:

$$\begin{aligned} CS(u_1, u_2) &= 1 - \theta/\pi, \text{ with} \\ \theta &= \arccos \frac{u_1 \cdot u_2}{\|u_1\|\|u_2\|} \end{aligned} \tag{2}$$

Note that u_1 and u_2 represent user vectors of ratings for all movies, with zeros for movies that have not been rated by a user.

Discrete cosine similarity

The formula to obtain discrete cosine similarity is actually exactly the same as Equation 2. The difference is in the vectors u_1 and u_2. Where for regular cosine similarity these vectors contain the actual ratings, the vectors for discrete cosine similarity contain only zeros (not rated) and ones (rated). An example: imagine we have vectors $u_1 = (3, 0, 2, 4, 5)$ and $u_2 = (0, 0, 3, 2, 4)$ for cosine similarity. For discrete cosine similarity these vectors will be represented as: $u_1 = (1, 0, 1, 1, 1)$ and $u_2 = (0, 0, 1, 1, 1)$.

Now that we have defined all the concepts necessary to understand our quest for most similar pairs, we can move on to the actual results.

Results

The Locality Sensitive Hashing algorithms were implemented to assess the performance of the different similarities measures. The number of similar pairs and computing time were analyzed as measurement values. The conducted experiments and final results are discussed per similarity measure.

Jaccard Similarity

To the performance of the Jaccard Similarity algorithm, some initial experiments were conducted with a subset of the dataset. During these first experiments, the signature matrix consisted out of 100 rows. Figure 3 shows the results from analyzing the first 30 million observations of the dataset. Unfortunately, running the algorithm with band with rows r equal to four was not executed in less than 30 minutes, thus $r \leq 4$ were left out as possible optimal values. Figure 3 shows that decreasing the row size of the bands results in an higher amount of similar pairs. Therefore, we can conclude that we should find an amount of rows as small a possible, but still running the algorithm in less than 30 minutes. Moreover, we noticed that decreasing the sign similarity threshold increases the amount of similar pairs, however the increase of similar pairs is diminishing, and the increase in computation time is not. Therefore, we should look for an optimal sign similarity threshold between 0.4 and 0.5, which gives a lot of similar pairs and has a relative small effect on the computing time.

Based on our findings from initial experiments, we started experimenting with the complete dataset. Unfortunately, the algorithm was not executed in less than 30 minutes using bands with five rows. Figure 4 shows that having bands with rows larger than six strongly decreases the amount of similar pairs. Therefore, we decided that the optimal number of rows should be six. Experimenting with the sign similarity thresholds resulted in the finding that at a threshold of 0.42, there is a favorable trade off between the amount of similar pairs and computing time.

Subsequently, some additional experiments were executed while adjusting the number of rows of the signature matrix. Figure 5 shows that increasing the rows in the signature matrix increases the amount of similar pair and causes a relative small increase in computing time. After running the algorithm with different signatures matrices, we selected the number of rows

Figure 3: Amount of similar pairs and computing time using 30 million observations from the dataset, using different amounts of rows, and different sign similarity thresholds.

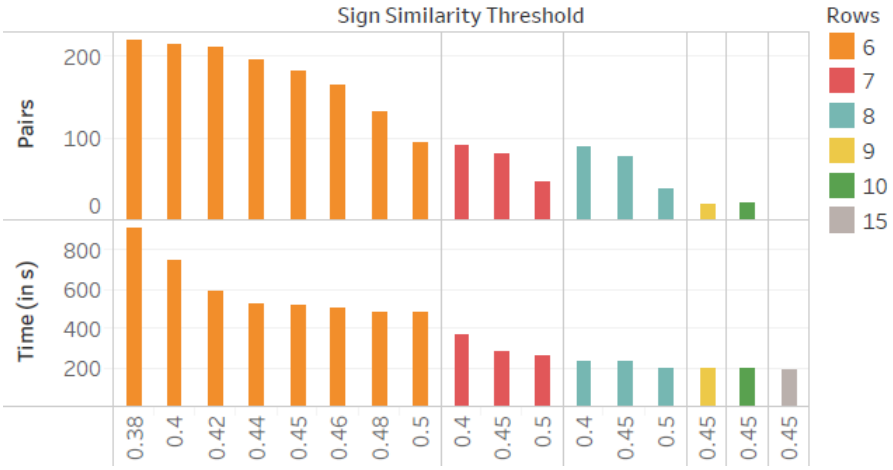

Figure 4: Amount of similar pairs and computing time using all observations from the dataset, using different amounts of rows, and different sign similarity thresholds.

in the signature matrix based on the highest amount of similar pairs. Therefore the optimal amount of rows in the significance matrix was 144 and this results in 24 bands.

Our final Jaccard similarity algorithm has a signature matrix with 144 rows, it uses a signature similarity threshold of 0.42 and has 24 bands with 6 rows. Executing the algorithm, resulted in finding 330 pairs with Jaccard similarity above 0.5. The computation time was 14 minutes and 43.1 seconds, based on a Lenovo G50-70, with 8GB RAM and a Intel Core i7-4500 CPU processor

(Discrete) cosine similarity

To find a set of most similar user pairs, we first performed a grid-search-like procedure to find proper parameters for the LSH algorithm. As discussed previously, for (discrete) cosine similarity we can adjust the number of random vectors and the number of rows (or, equivalently,

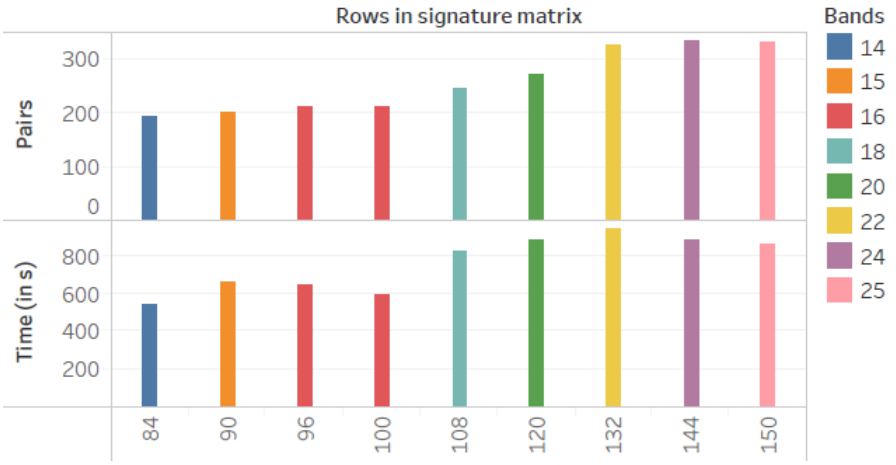

Figure 5: Amount of similar pairs and computing time using all observations from the dataset, using 6 rows per band, a sign similarity threshold of 0.42 and differing amount of rows in the signature matrix and amount of bands.

the number of bands). Of course it is also possible to adjust the threshold for when two users are considered similar, but for this assignment this threshold was fixed at $CS = 0.73$. Moreover, not that we always set the number of random vectors as a multiple of the number of rows, in order to prevent from obtaining small bands. Table 2 presents an overview of our research for cosine similarity. The first three columns give the number of random vectors, the number of rows and the number of bands, respectively. In the column 'candidates' you can find the number of candidate pairs that resulted from Step 3 of Algorithm 1, and 'pairs' gives the number of most similar pairs that resulted from Step 4. Runtime in seconds is presented in the last column. The highlighted columns are the ones that we validated with true cosine similarity on the sparse matrix U, since they yield a quite sensible amount of pairs according to the guidelines for this assignment.

random vectors	**rows**	**bands**	**candidates**	**pairs**	**runtime** (s)
90	15	6	61,364	6,630	382.38
100	20	5	81,981	14,197	30.15
1000	20	50	811,678	90	275.53
100	25	4	28,086	9,366	14.00
150	25	6	40,745	5,654	19.48
1500	25	60	390,867	33	72.09
90	30	3	4,494	3,066	10.51
120	30	4	5,964	2,581	12.44
900	30	30	37,692	59	88.50
150	50	3	0	0	15.09
99	33	3	1257	899	11.18

Table 2: Overview of the number of candidates and final pairs as obtained from different (n_{vector}, n_{rows})-combinations for cosine similarity. Runtime (s) is also displayed in the last column. Highlighted are selected for further investigation and validation.

From Table 2 we observe several interesting facts. First of all, we see the number of candidate pairs linearly increase with the number of random vectors. For example, 100 random vectors give roughly 82,000 candidate pairs, but 1000 random vectors give roughtly 820,000 pairs. This makes perfect sense, since we compare ten times as much bands pairs. However, when the number of candidates increase, one might also expect that the number of actual pairs grows as well. Remarkably enough, this is not the case.

Another important observation is that the number of rows really determines the number of final candidates and similar pairs as well. A relatively small number of rows - say 15 - increases the runtime considerably. This can be explained by the fact that much more pairs will end up in the same bucket, because two vectors of length 15 are more likely to be similar than two vectors of length, say, 25 (although we cannot see this, since the final output has removed all duplicates). On the other hand, when we have relatively large bands of 50 rows, there are no vectors left that have exact equal bands in the signature matrix. The optimum seems to lie somewhere between 25 and 30 bands.

The same procedure was repeated for discrete cosine similarity, where the input matrix U had a slightly different representation with only zeros (not rated) and ones (rated). Since the results were quite comparable, we did not include a table with all the results to avoid redundancy. As final step in our research we investigated for the highlighted combinations of Table 2 whether the obtained similar pairs also have a true cosine similarity that exceeds the threshold of 0.73.

Validation Results of the validation are presented in Table 3. One can observe that the results are quite curious, since from thousands of similar pairs, there only remain 160 *in the best case scenario.*

random vectors	**rows**	**bands**	**pairs**	**cosine pairs**
150	25	6	5,654	6
90	30	3	3,066	160
120	30	4	2,581	34

Table 3: Overview of the number of similar pairs with the last column containing the number of pairs that remain by applying cosine similarity on the sparse matrix U.

Conclusion

In this report we first presented an introduction to the problem of finding similar pairs. We provided an extensive description of the LSH algorithm and the different similarity measures, as well as how we implemented LSH, minhashing and random projections. In the results section we provided an extensive overview on the influence of the parameters signature length and the number of rows and bands on the set of most similar pairs. Based on the presented results, we conclude that for Jaccard similarity we would recommend to use 144 signatures in combination with 6 rows, which yield 24 band and around 330 pairs of similar users. For (discrete) cosine similarity, we would recommend to use 90 random vectors in combination with 30 rows, which yield 30 bands and around 3,000 pairs of similar users.