# Assignment 2: Implementing Locality Sensitive Hashing

*W. Kowalczyk*

*wojtek@liacs.nl*

*15.10.2020*

## Introduction

The purpose of this assignment is to find pairs of most similar users of Netflix with help the LSH technique. There are 3 ways of measuring similarity of two users:

1. Jaccard Similarity (JS). Similarity between two users $u_1$ and $u_2$ is measured by the Jaccard similarity of the sets of movies that they rated, while ratings themselves are irrelevant. Thus, if $S_i$ denotes the set of movies rated by the user $u_i$, for $i=1, 2$, then the similarity between $u_1$ and $u_2$ is $\#intersect(S_1, S_2)/\#union(S_1, S_2)$, where $\#S$ denotes the cardinality (the number of elements) of $S$.

2. Cosine Similarity (CS). Similarity between two users $u_1$ and $u_2$ is measured by the cosine similarity of vectors of ratings given by these two users to movies they rated. Unrated movies are supposed to be rated as 0.

3. Discrete Cosine Similarity (DCS). It is defined as the Cosine Similarity applied to "truncated vectors of ratings", where every non-zero rating is replaced by 1.

   For example, suppose that we have 6 movies and $user_1$ rated movies 2, 3 and 5 with 5, 4, 3, respectively, while $user_2$ rated movies 1, 2, 3 with 5, 4, 3, respectively.
   Then vectors of ratings are:

   $u_1=(0,5, 4, 0, 3,0)$
   $u_2=(5, 4, 3, 0, 0,0)$

   and

   $JS(u_1, u_2)=JS(\{2,3,5\}, \{1,2,3\})$,
   (we only care about movies that were rated but not in actual ratings)

   $CS(u_1,u_2)=cossim((0,5, 4, 0, 3,0), (5, 4, 3, 0, 0,0))$,

   $DCS= cossim((0,1, 1, 0, 1,0), (1, 1, 1, 0, 0,0))$,

   where *JS* denotes the standard "Jaccard Similarity" of two sets and *cossim* is standard cosine similarity of two vectors.

Your challenge is to find, given a set of users and movies they rated, pairs of users with high similarity:

**Task 1**: Jaccard similarity should be bigger than 0.5.

**Task 2**: Cosine Similarity should be bigger than 0.67.

**Task 3**: Discrete Cosine Similarity should be bigger than 0.66.

Of course, these tasks are disjoint, i.e., you have to create 3 lists of pairs of users; one list per task. The more pairs your program finds the better. However, due to the size of the original data (more than 100.000 users who rated in total 17.770 movies, each user rated at least 300 movies), instead of using the brute force approach (i.e., calculating the Jaccard similarity of about 100.000*100.000/2=5.000.000.000 pairs) you should use minhashing (in case of Jaccard similarity), random projections (in the remaining two cases) and the banding technique in all 3 cases.

### Data

The data comes from the original Netflix Challenge (www.netflixprize.com). To reduce the number of users (originally: around 500.000) and to eliminate users that rated only a few movies, we selected only users who rated at least 300 and at most 3000 movies. Additionally, we renumbered the original user ids and movie ids by consecutive integers, starting with 1, so there are no "gaps" in the data. The result, a list of 65.225.506 records, each record consisting of three integers: *user_id*, *movie_id*, *rating*, has been saved in a file *user_movie_rating.csv* as a big array of integers (65.225.506 , 3), where the first column contains ID's of users, the second one ID's of movies, and the third one ratings. The (zipped) file can be downloaded from
https://drive.google.com/file/d/1Fqcyu9g6DZyYK_1qmjEgD1LlGD7Wfs5G/view?usp=sharing

### Your Tasks

Implement (in plain Python, possibly with the numpy/scipy libraries) the LSH algorithm and apply it to the provided data to find pairs of users whose similarity is bigger than thresholds listed above. The output of your algorithm should be written to **three text files**, as a list of records in the form ***u1, u2*** (two integers separated by a comma), where ***u1<u2*** and the similarity exceeds the provided threshold. Additionally, as your program will involve a random number generator and we will test your program by running it several times with various values of the random seed, your program should **read the value of the random seed from the command line**. The complete runtime of your algorithm should be at most 1 hour and 30 minutes (30 minutes for each similarity measure), on a computer with 8GB RAM. More details on the submission format are on page 4.

Describe in your report the choices you've made when implementing and tuning your algorithm:

- The data structures you've used for implementing minhashing (e.g., how do you represent the input data), minhash table, buckets, etc.
- The procedure for calculating the minhash table.
- The choice of the parameters that are critical for the "success rate" of your algorithm: the signature length (h) the number of rows (r), the number of bands (b).

- The “post-processing” – how do you select buckets to be tested? How do you test candidate pairs of users (i.e., are they really “similar” with the similarity bigger than the threshold?)

**Hints**

1) It is essential to select a right method of representing the input data (the Movie x User or User x Movie data). We would strongly recommend to use the 'sparse' package from Scipy library: https://scipy-lectures.org/advanced/scipy_sparse/index.html

The advantage of sparse matrices is that they store only non-zero elements, so you save memory, and, more important, they support very efficient implementations of rearranging columns or row, for example: B=A[:, randomly_permuted_column_indices]) and other operations. There are several types of sparse matrices:

| | |
|---|---|
| *coo_matrix(arg1[, shape, dtype, copy])* | A sparse matrix in Coordinate format. |
| *csc_matrix(arg1[, shape, dtype, copy])* | Compressed Sparse Column matrix |
| *csr_matrix(arg1[, shape, dtype, copy])* | Compressed Sparse Row matrix |
| *lil_matrix(arg1[, shape, dtype, copy])* | Row-based linked list sparse matrix |

You can think and/or experiment with these alternatives to find out which would work best. There is no single recommendation we can give **-** in the past few years students were successfully using all these types (or none of them)!

2) Your dataset is so small that you can use random permutations of columns or rows (instead of hash functions). In this way you can avoid time consuming loops.

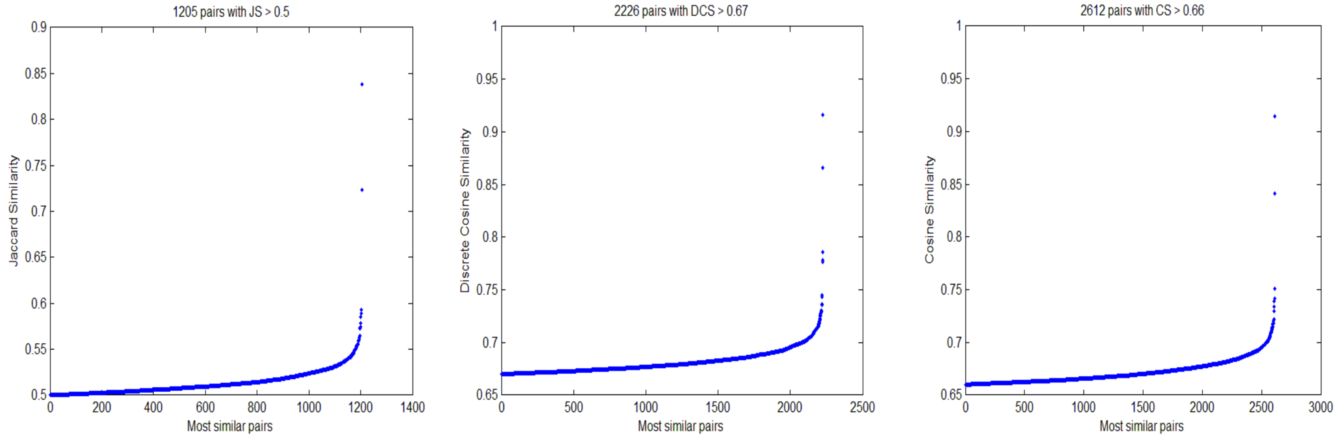
3) Relatively short signatures (80 -150) should result in sufficiently good results (and take less time to compute).

4) When there are many users that fall into the same bucket (i.e., there are many candidates for being similar to each other) then checking if all the potential pairs are really similar might be very expensive: you have to check *k(k-1)/2* pairs, when the bucket has *k* elements. Postpone evaluation of such a bucket till the very end (or just ignore it – they are really expensive). Or better: consider increasing the number of rows per band – that will reduce the chance of encountering big buckets.

5) Note that *b*r* doesn’t have to be exactly the length of the signature. For example, when you work with signatures of length *n=100*, you may consider e.g., *b=3, r=33; b=6, r=15*, etc.

6) To make sure that your program will not exceed the 30 minutes runtime you are advised to close the *result.txt* file after any new pair is appended to it (and open it again, when you want to append a new one).

7) Keep in mind that the thresholds are selected in such a way that most likely your program will not be able to find more than 50-200 similar pairs – lowering these thresholds a bit will result in a huge increase of similar pairs – tens of thousands – so please, use the threshold provided in this document.



Finally, for your convenience, we illustrate the distribution of similarities of ALL similar pairs that satisfy the conditions of this assignment – see the figures below.

**Deliverables:** your submission should consist of a ZIP file with the following contents:

1. **Report:** ***report.pdf***

2. **Code folder:** you can structure your code however you want but it must have a file '***main.py***' file that takes the following command line arguments (take a look at argparse package: https://docs.python.org/3/library/argparse.html):

   **[-d str]** → Data file path
   **[-*s int]*** → *Random seed (by using np.random.seed(int))*
   ***[-m str]*** → *Similarity measure (js / cs / dcs)*

   **Example:** python main.py **-d** */very/long/path/to/data.npy* **-s** 2020 **-m** dcs

   **Output:** your results should be dumped to *js.txt*, *cs.txt* and *dcs.txt* depending on the similarity measure (js : jaccard, cs: cosine similarity, dcs: discrete cosine similarity) , where each line in these files corresponds to a user pair in ***u1, u2*** format, while ***u1<u2****:*

```
1, 2
42, 51
121, 201
…
```

Before writing your output to a file, pleas make sure you overwrite previous files (from previous runs with different seeds). E,g., if we run your program with "-m js", make sure to overwrite a potentially existing file js.txt.

All submissions will be evaluated automatically with a script, so follow these guidelines! Your script is supposed to terminate in 30 minutes for each similarity measure and after this time the contents of your results file will be treated as the output of the run. Your grade will be influenced by the number of unique pairs that you were able to generate over 5 runs with different seeds (up to **3.0** points in total).

**The deadline of this assignment is 15th November 23:59.**

**Every day you are too late will result in -0.2 points.**