Final Assignment Advances in Data Mining

Implementing Locality Sensitive Hashing

Introduction

The purpose of this assignment is to find pairs of most similar users on Netflix with help of the LSH technique. We will be basing the similarity on Jaccard Similarity: 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.

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:

$user_1 = (0, 5, 4, 0, 3, 0)$
$user_2 = (5, 4, 3, 0, 0, 0)$

and

$$JS(user_1, user_2) = JS(\{2,3,5\}, \{1,2,3\}),$$

since we only care about movies that were rated but not the actual ratings.

Your challenge is to find, given a set of users and movies they rated, pairs of users with high similarity: Jaccard Similarity should be bigger than 0.5. 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 and the banding technique, as described in the lecture of Week 4.

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 corresponding .npy file can be downloaded from
https://drive.google.com/file/d/1Fqcyu9g6DZyYK_1qmjEgD1LlGD7Wfs5G/view?usp=sharing%20%20

Your Task

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 the threshold listed above. The output of your algorithm should be written to a text file as a list of records in the form u_1,u_2 (two integers separated by a comma), where $u_1<u_2$ 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 30 minutes, on a computer with 8GB RAM.

Along with the specified computer program, you should submit a report of around 5 pages, written in Latex or Word. The quality of your report is the primary component of your grade, followed by the correctness, tidiness, and efficiency of your code. 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?)
- The number of similar pairs ($u_1<u_2$) you found, running the submitted program on your hardware. Note that the number should be reasonable given the specified constraints, but otherwise will not play a role in the assessment, so doesn't need to be optimized extensively.

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 the 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 small enough to 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 $k(k\text{-}1)/2$ pairs, when the bucket has k elements. Postpone evaluation of such a bucket until 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.

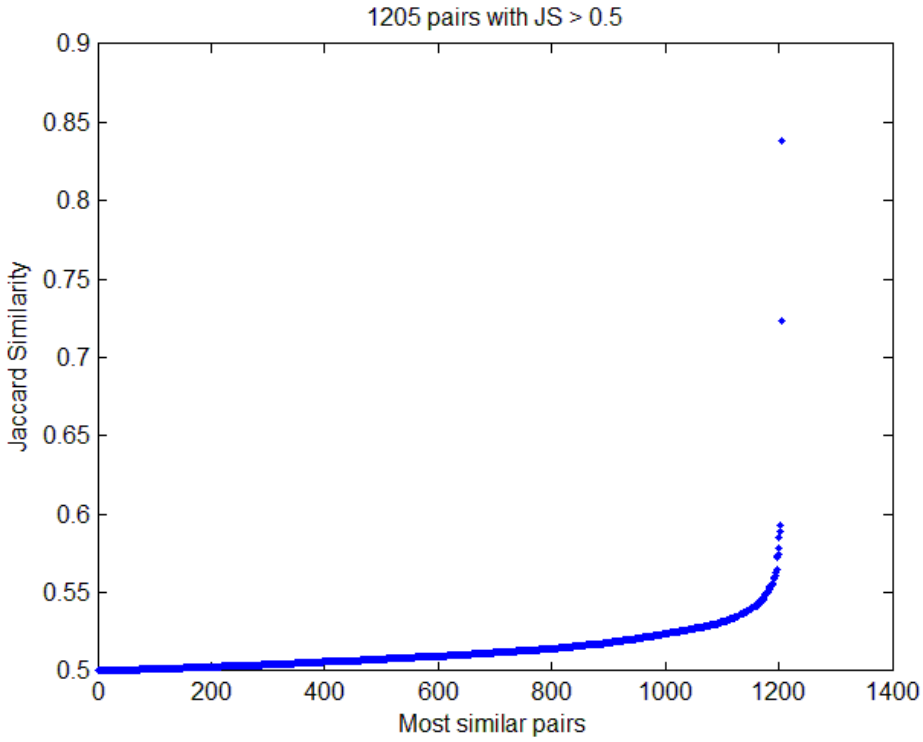
5) Note that $b*r$ doesn't have to be exactly the length of the signature. For example, when you work with signatures of length $n = 100$, you may consider e.g., $b = 3$, $r = 33$; $b = 6$, $r = 15$, etc.

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**.

8) The random seed may have a substantial influence on the number of pairs found. It is not recommended to cherry-pick a good seed, but rather optimize you program to work well across different random seeds.

Finally, for your convenience, we illustrate the distribution of similarities of ALL similar pairs that satisfy the conditions of this assignment – see the figure below – computed in a brute force manner (which you should not do).


1205 pairs with JS > 0.5
Jaccard Similarity
Most similar pairs
0.9
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
0
200
400
600
800
1000
1200
1400