Assignment 3: Locality Sensitive Hashing

W. Kowalczyk

wojtek@liacs.nl

9.10.2018

Introduction

The purpose of this assignment is find similar users of Netflix with help of minhashing and LSH. This time, similarity between two users *u1* and *u2* should be 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 u_i, for *i=1, 2*, then the similarity between u_1 and u_2 is $\#intersect(S_1, S_2)/\#union(S_1, S_2)$. Our objective is to find, given a set of users and movies they rated, pairs of users with Jaccard similarity bigger than 0.5 - the more pairs the better. However, due to the size of the 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) we will use minhashing and the LSH techniques.

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 recoded the original user ids and movie ids by consecutive integers, starting with 0, so there are no gaps in the data. The result, a list of 65.225.506 records, each record consisting of two integers: *user_id* and *movie_id*, has been saved in the numpy's binary file *user_movie.npy* as an array of integers of size (65.225.506 , 2), where the first column contains ids of users and the second one ids of movies. The (zipped) file can be downloaded from: https://surfdrive.surf.nl/files/index.php/s/WwZqzkkHxg6KLlL

Your Task

Implement (in Python) the LSH algorithm with minhashing and apply it to the *user_movie.npy* data to find pairs of users whose similarity is bigger than 0.5. The output of your algorithm should be written to a text file, as a list of records in the form *u1,u2* (two integers separated by a comma), where *u1<u2* and *jsim(u1, u2)>0.5*. ~~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. The details of the submission and evaluation procedure are given below.

Submission Procedure

Submit a zipped folder with your code (without the data!) as a single file *A3_your_names.zip*. The folder should contain a file called *main.py*, which will be called (from your unzipped directory) with two arguments: a random seed (an integer) and a string that specifies the location of the *user_movie.npy* file, for example:

```
>> python main.py 1234  /home/wojtek/A3/evaluation/user_movie.npy
```

~~Your program should then load the data, set the random seed to the value provided on the command line (in our example: 1234)~~, perform computations and dump the results to a file *results.txt* , as a list of ordered pairs: *u1, u2* (one pair per line), with *u1<u2*. Our evaluation script will check if your solution is valid: i.e., if the listed pairs really represent pairs of users with similarity bigger than 0.5, and if there are no repeating records. ~~Set the seed of the random number generators with help of: *np.random.seed(seed='your seed')*~~. As the results will be evaluated by a special script it is important that the results.txt file contains only pairs of integers, separated with a comma, one pair per line, and nothing else.

Evaluation Procedure

Your program is supposed to terminate in less than 30 minutes on a machine with Intel Xeon E5-2630v3 CPUs @ 2.40Ghz, using just one core of this processor. If it doesn't terminate in 30 minutes, the process will be killed and the *results.txt* file (if exists) will be treated as the output of the run. Your program will be tested 5 times with different values of the random seed and the results will be averaged.

The baseline grade**, 6.0**, will be given if your algorithm terminates in less than 30 minutes, **producing (on average) at least 10 valid pairs of similar users**. The grade for this assignment will depend on the **average number of found pairs per minute (over 5 runs).** Additionally, we will look at:

- the **total number of found pairs** (over 5 different runs),
- the **median** run time (over 5 different runs),
- **code readability, elegance**.

We will also run a plagiarism check to make sure that the submitted code is the result of your own work.

Expected results

After trying a few setups and coding tricks your program should be able to find in 30 minutes a few hundred pairs (with similarity > 0.5), depending on the value of the random seed. In total there are 1219 pairs of users with similarity > 0.5 (found with help of deterministic exhaustive search for similar pairs).

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://docs.scipy.org/doc/scipy-0.18.1/reference/sparse.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]).~~ 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 experiment with them to find out which would work best for you. There is no single recommendation I can give - in the past few years students were successfully using all these types!

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

3) Relatively short signatures (50-150) should result in 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 signature of length n=100, you may consider e.g., b=3, r=33, b=6, r=15, etc. – any combination of b

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).

Organization

The deadline for this assignment is **Tuesday, 23:59, 23.10.2018.** You should submit only your code (no report is required). Your submission -- a single zip file named **A3_Id1_Id2.zip** -- should be mailed to **aidm@liacs.leidenuniv.nl**, with the subject line: **A3_Id1_Id2.**