Advances in Data Mining
Assignment 2 - Implementing Locality Sensitive Hashing

Dimitrios Ieronymakis
d.ieronymakis@umail.leidenuniv.nl
Maria Ieronymaki
m.ieronymaki@umail.leidenuniv.nl

Contents

Introduction **2**

Dataset **2**

Similarities **2**

Jaccard Similarity . . . 3

Cosine Similarity . . . 3

Discrete Cosine Similarity . . . 3

Implementation **3**

Signature Matrix . . . 4

Min-Hashing . . . 4

Random projections . . . 5

Locality-Sensitive Hashing . . . 5

Post processing . . . 5

Parameter tuning and results **6**

Conclusion and Future Work **7**

References **7**

Introduction

As the years go by, the volume of data produced is growing at an exponential rate, leading to deal with deluge data, which literally means “flood of data”. This refers to the so-called high-dimensional data that is a collection of an extravagant number of dimensions, variables and columns which has made the process of analysis or manipulation of data more and more difficult as far as time and space complexity are concerned. One field in which high dimensional data is used are recommendation systems that opt to predict the ”rating” or ”preference” a user would give to an item by filtering the data and looking for similarities between users. Obviously, following a Naive approach by finding the exact results for each user, would lead to large storage or time requirements, so to address this problem, a solution is to use algorithms that are used to optimize the extraction of approximate results without losing important information. This report deals with finding pairs of most similar users out of the Netflix dataset by implementing three ways of measuring similarity: the Jaccard, the Cosine and the Discrete Cosine similarities. It also addresses three very crucial techniques when dealing with high dimensional data, which are the ”minhashing” and ”random projection” techniques used to compress large sets while preserving similarity and the ”locality-sensitive hashing” technique which deals with the complication when there may be far too many pairs of items to test for their degree of similarity. The report is structured in such a way that it begins by giving a theoretical definition of the different similarity measures used to find the pairs of similar users, then it continues with the description of the steps followed for the implementation of the techniques mentioned above, in order to find a valid algorithm which is able to find a satisfactory number of pairs in the shortest time and memory possible. It is important to underline that the Implementation section concerns all three measures since most of the steps followed are common, however the differences will be mentioned.

Dataset

The data comes from the original Netflix Challenge and provides information about the movies that have been rated by users. The original number of users (around 500.000) was already reduced; users that rated a few movies were eliminated by selecting only users who rated at least 300 and at most 3000 movies. No pre-processing of the data was necessary as the original user ids and movie ids were renumbered by consecutive integers, starting with 1, in order to avoid “gaps” in the data. Overall, the dataset is a list of 65.225.506 records, each record consisting of three integers: user_id, movie_id, rating, where the first column contains ID’s of users, the second one ID’s of movies, and the third one the ratings.
In order to represent the input data and save memory, we used the sparse matrix of Scipy library (csr_matrix), that stores only non-zero elements, with rows being the Movies and columns the Users (Movie x User). For simplicity, we will refer to this matrix as csr_sparse.

Similarities

Similarity measures are used as a reference to determine how similar two users are based on the ratings they have given. A high similarity score means that the pair of users considered is similar. The following subsections provide a definition of the three similarities implemented.

Jaccard Similarity

The Jaccard similarity is the similarity of sets calculated by looking at the relative size of their intersection and it is defined as follows: given a set S_1 and a set S_2, the Jaccard similarity $SIM(S_1, S_2)$ is the ratio of the size of the intersection of S_1 and S_2 to the size of their union.

$$SIM(S_1, S_2) = \frac{S_1 \cap S_2}{S_1 \cup S_2}$$

In our case, S_1 and S_2 denotes the set of movies rated by the user U_i, for i=1, 2. It is important to underline the fact that when calculating the Jaccard similarity, the ratings are irrelevant, and we consider only if the movies have been rated. This means that if a user has given a rating to a specific movie, the latter is replaced by 1, otherwise is 0.

Cosine Similarity

Given two vectors v_1 and v_2 which in our case represent the rating columns of the two users taken into consideration, the Cosine similarity is a score calculated with the following formula:

$$CS(v_1, v_2) = 1 - \frac{\theta}{180}$$

where θ is the angle in degrees between the two vectors and it is calculated by finding the arccos(θ) of the normalized dot-product of the two vectors, $cos(\theta) = \frac{v_1 \cdot v_2}{\|v_1\| \times \|v_2\|}$.

The Cosine similarity is a score between [0,1] and the smaller the angle the more similar the vectors are.

Discrete Cosine Similarity

The Discrete Cosine similarity is defined as the Cosine similarity applied to “truncated vectors of ratings”. Specifically, ratings themselves are irrelevant and every non-zero rating is replaced by 1, similar to Jaccard similarity. The cosine similarity is then calculated on columns of zeros and ones.

Implementation

Since the naive approach of checking every pair of users may be prohibitive when working with large datasets, we based our algorithm on the *locality−sensitive hashing* (LSH) technique. The general idea behind the LSH mechanism is that we hash items of the *csr_sparse* using many different hash functions. These hash functions are designed so that pairs of similar users will end up in the same bucket. We can then examine the users of each bucket to calculate our candidate pairs by using an approximate similarity measure. By using this method we can avoid the quadratic growth in computation time that is required for checking every pair of users.

Our approach can be summarized with the following steps:

1. Create a *signature matrix* from the sparse matrix csr_sparse, where each user is assigned a signature of length p, by using Min-Hashing or Random Projection methods.
2. Divide the signature matrix signatures in bands of length b with r rows each.
3. For each *band* we hash the sub-signature of users and add the user to the corresponding bucket.
4. Calculate an $approximate\ similarity$ between all users in the same bucket and keep the ones with a similarity above our $threshold$ to add in our candidate pairs.
5. Calculate the true similarity, either with Jaccard similarity, Cosine or Discrete Cosine similarity on the csr_matrix.

Below we describe the implementation for the most important concepts and steps needed to implement our algorithm.

Signature Matrix

The signature matrix is used to reduce the size of large datasets with smaller representations called signatures, but at the same time preserving the similarity. The main characteristic that our signatures will have is that they will be comparable between each other to estimate their similarity. In our case, we will create a signature for each user, based on their movie ratings, with a length of 80-150 elements, in order to compare their similarity and create candidate pairs.
The techniques we will use to create our signature matrices are called $Min-Hashing$ and $Random\ Projections$ and are explained below.

Min-Hashing

The signatures for each user are composed from the results of a large number of calculations, each of which is a "minhash" of the characteristic matrix. The minhash functions $h_1, h_2, \ldots, h_p$ are used to map data to fixed-size values. Since our dataset is not very large instead of using hash functions, to represent sets, we pick at random a number p of permutations of the rows of our csr_sparse matrix. In this way, we avoid using loops which take a lot of computational time. The *signature matrix* has the same number of rows r as the columns of the csr_sparse matrix but only c = p columns and its values are initialized to ∞.
The creation of the signature matrix can be summarized as follows: let SIG(i, c) be the element of the signature matrix for the ith random permutation and column c. We handle row r by doing nothing if c has 0 in row r, otherwise for each i = 1, 2, . . . , p set SIG(i, c) to the smaller of the current value of SIG(i, c) and $h_i(r)$. As we already mentioned, we will use the rows of this matrix to calculate an approximate similarity for our user pairs.

Random projections

In order to build the signature matrix for the cosine and discrete cosine similarity, it is necessary to pick a hyperplane that intersects the plane of the two datapoint vectors (columns of the sparse matrix) taken into consideration. Hyperplanes in this method are used to split our data and assign a value of -1 for those data points that appear on the negative side of our hyperplane — and a value of 1 for those that appear on the positive side. To do so, we actually pick the normal vector to the hyperplane and compute the dot product with the datapoint vector. In our case, instead of choosing a random vector, we create a collection of p vectors with normally distributed values and calculate the dot product between each vector and each column of the csr_spare (each user). If the result of the dot product is positive, the result is replaced by 1, otherwise with -1. The result obtained is a *sketch*, which is considered to be the signature matrix used afterwords for LSH. Similar to the signature matrix built with hashing, this one, will have the same number of rows as the columns of the csr_sparse matrix but only p columns.

Locality-Sensitive Hashing

As already mentioned, the main purpose of LSH is to hash items multiple times in order to create pairs of similar users called *candidate pairs*.
Since we have already created our *signature matrix* with the methods described above, an effective way to choose hashings is to use *banding* to divide the signature matrix in b bands and r rows of signature splits. For each band we use a hash function on the vectors representing the signature splits in order to hash each user in a large number of *buckets*. In our implementation, the buckets will be represented with a dictionary. Keys of our dictionary will be the hash of the vectors and the values of the dictionary will be lists of users that hashed to the same bucket (and therefore have the same hash output). To hash the vectors we will simply use python's *hash*() function.
In order to choose our *bands* and *rows* we need to consider another variable called *threshold* (t). The threshold is defined as such:

$$t = (\frac{1}{b})^{\frac{1}{r}}$$

The best *threshold* variables are well-known values and to perform well they depend on the similarity measure considered. For Jaccard similarity we will consider $t = 0.5$ and for Cosine similarity and Discrete Cosine similarity we will consider a $t = 0.73$. These values will be used to find the best configuration possible of bands and rows, by using the formula above, and to calculate the similarities of our candidate pairs in the post processing section below.

Post processing

Before adding all possible combinations of users from our buckets to our list of candidate pairs, we need to restrict our results or the comparisons we will need to make will be too many. We will use two techniques in order to restrict our candidate pairs and lower the number of false negatives. Firstly we will only consider buckets with size greater than 1 and smaller than 100.
Secondly, for each pair in our dictionary we will calculate an *apprximate similarity*. This approximate similarity is calculated by taking the whole signatures of the users and calculating the percentage of elements they have in common, in the same position. When this similarity is greater than the *threshold*, we will add

the pair of users to our candidate pairs.
After having found all the candidate pairs from each band, we implement a further evaluation step to find which of the candidate pairs are really similar. In this last step, for each candidate pair, we calculate the similarity (js, cs or dcs) on the sparse matrix's columns corresponding to the candidate pair in order to find which of the candidate pairs are truly similar.

Parameter tuning and results

The parameter tuning to find the parameters that are critical for the "success rate" of the algorithm was conducted manually and experiments were run on an Intel Core i7-10700K clocked at 4.7 Ghz, which makes computations really fast, as we can see from the Tables below.

For the Jaccard similarity the optimum values of the parameters found were: signature length $h = 120$, number of rows $r = 6$, number of bands $b = 20$ and threshold $t = 0.5$. The number of pairs found by our algorithm with three different seeds (s= 0,42,100) are reported in the table below:

Seed	Candidate pairs	True pairs	Time (min)
0	16387	310	2
42	22667	317	2
100	34542	424	2.2

Table 1: Results obtained for the Jaccard Similarity using the best parameters and three different seeds (s= 0,42,100). The Time column indicates the total run time of the algorithm.

For both the Cosine and the Discrete Cosine similarity the optimum values chosen were: signature length $h = 150$, number of rows $r = 15$, number of bands $b = 10$ and threshold $t = 0.73$. The number of pairs found by the two algorithms with three different seeds (s= 0,42,100) are reported in Table 2 and 3:

Seed	Candidate pairs	True pairs	Time (min)
0	263790	75	2.13
42	359613	149	2.36
100	354609	154	2.32

Table 2: Results obtained for the Cosine Similarity using the best parameters and three different seeds (s= 0,42,100). The Time column indicates the total run time of the algorithm.

Seed	Candidate pairs	True pairs	Time (min)
0	201767	49	1.54
42	342185	87	1.28
100	355290	88	1.29

Table 3: Results obtained for the Discrete Cosine Similarity using the best parameters and three different seeds (s= 0,42,100). The Time column indicates the total run time of the algorithm.

Conclusion and Future Work

In conclusion, we successfully implemented the LSH algorithm for finding similar users with each similarity measure. It might appear that the Jaccard similarity performs better than the cosine similarities however, the reality is that the cosine similarity and the discrete cosine similarity provide a more precise measure of similarity, which makes them more accurate in their predictions. Also, an interesting observation can be made for the computational power required to calculate the actual similarities. Although all three methods have very similar running times, we can see that the number of candidate pairs for the cosine similarities is approximately 10 times bigger than the candidate pairs found with the Jaccard similarity. This means that the cosine similarity implementation is much faster in calculating actual similarities of long vectors of values.
Finally, it would be interesting to continue with the parameter tuning of the cosine similarities in order to see if further improvement is possible, or even better, find a way to restrict the candidate pairs and spend more time looking for them rather than evaluating if they are valid or not.

References

[1] Leskovec, J., Rajaraman, A., Ullman, J. D. (2020). Mining of Massive Data Sets. Cambridge University Press.

[2] Slides from: Leskovec, Rajaraman, Ullman, J. A. J. (n.d.). Mining of Massive Datasets. http://www.mmds.org/