LSH One : Locality Sensitive Hashing

Yassine Abou Hadid, Clément Préaut, Benjamin Sykes

Tuesday 11th October, 2022

1 Introduction to LSH

The key idea for LSH in recommendation is to find the relevent pairs of users on which we compute the similarity to infer a rating using following formula 1 (where R is a rating and N is the number of users we have in the dataset)

$$R_{user=i,movie=j} = \frac{\sum_{k=1}^{N} \text{UserSim}(i,k).R_{user=k,movie=j}}{\sum_{k=1}^{N} \text{UserSim}(i,k)} \tag{1}$$

The issue with equation 1 is the fact that in order to predict a rating for a single movie, we need to have had **computed the similar for the specific user to all the other users, which is very costly**.
By methods we will present in the following parts, **LSH allowes to determine which users are similar and computes the rating only using those users similarity**.

A hash function h allowes to map a user vector to a much smaller dimension. We use several hash functions h_i for $i \in [1, S]$ where S is the "signature size" to compute a user's full signature. If for one of those hash functions, two users output the same value, we then compute the similarity. Equation 1 then becomes the same sum on a restricted set :

$$R_{user=i,movie=j} = \frac{\sum_{k/\exists h_\alpha, h_\alpha(i)=h_\alpha(k)} \text{UserSim}(i,k).R_{user=k,movie=j}}{\sum_{k=1}^{N} \text{UserSim}(i,k)} \tag{2}$$

The UserSim function can be stored as a symmetric matrix S, where $\boldsymbol{S}_{i,j} = \text{UserSim}(i,j)$ thus computing a prediction for a movie indexed m for a user i is equivalent to computing

$$R_{i,m} = \boldsymbol{S}_{i,.}\text{R}_{,m} \tag{3}$$

The whole LSH algorithm works because of the fact that similarities are preserved by the hashing :

$$\mathbb{P}(h(i) = h(j)) = \text{UserSim}(i,j) \tag{4}$$

The complete LSH for recommendation pipeline is represented in figure 1. Items on the top of the diagram represent the parameters on which we could play to improve the recommendation algorithm.

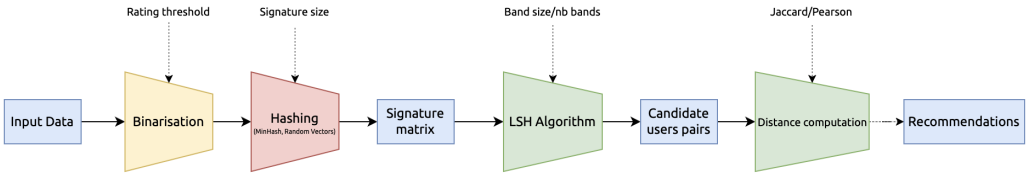

Figure 1: LSH workflow for item recommendation

1.1 Similarities

The following two similarities are the ones that we use in our models. Jaccard is the most simple one, Pearson takes into account the level to which a person rated a movie.

Pearson Similarity : the correlation of the two users ratings where $M_{i,j}$ is the intersection set of movies rated by users i and j (Where μ_k if the average rating of movie k) :

$$\text{Pearson}(i,j) = \frac{\sum\limits_{M_{i,j}} (r_k - \mu_k)(r_l - \mu_l)}{\sqrt{\sum\limits_{M_{i,j}} (r_k - \mu_k)^2 \sum\limits_{M_{i,j}} (r_l - \mu_l)^2}} \tag{5}$$

Jaccard Similarity : the *intersection over union similarity* :

$$\text{Jaccard}(i,j) = \frac{|R_i \cap R_j|}{|R_i \cup R_j|} \tag{6}$$

1.2 S-curve

The property described in equation 4 shows that for a given band, the probability that two items i and j with similarity s fall in the same bucket is (with r the number of rows in the band) :

$$\mathbb{P}(i,j \text{ in same bucket}) = s^r \tag{7}$$

Thus, the probability for two users to fall in at least one bucket is the following, where b is the number of bands ($r.b$ is the signature size)

$$\mathbb{P}(i,j \text{ in at least one bucket}) = 1 - (1 - s^r)^b \tag{8}$$

The target s-curve would look like a step function, changing where the threshold is considered as best. In order to find this optimal threshold, we will try our algorithms for different sizes of r and b. Different s-curves corresponding to varying values of r and b are displayed in 2. The idea is to reduce the false positive and false negatives rates that are entailed by the fact that with the curves on figure 2, some points have a similarity below threshold t and have a non null probability of being in a same buckets (**false positives**), and some have a similarity $s > t$ but have a probability $\mathbb{P} \neq 1$ of being in a same bucket (**false negatives**).

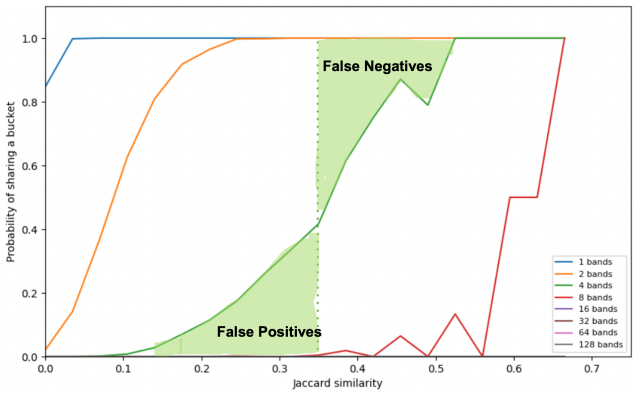

Figure 2: Different s-curves for different values of r and b

2 Work stream

2.1 MinHash : permutations on rows of the data matrix

The idea of this approach is to randomly permute the indexes of columns in the binary matrix. The hash function value of the ith user is then the index of the first permuted column in which the row R_i

has a non nan value. Repeating this process several times will create the row signature of this user.

2.2 Random vectors : hashing by dot product with random vectors

The random hyperplanes (also called random projection) is a hashing method based on the idea of generating a random vector and computing the dot product with each of the user matrix rows. The hash function value of the ith user is equal to 1 if the dot product is positive or 0 otherwise.
We repeated the random vector generation process to create the signature of each user.

3 Results

3.1 Method for testing the average error

We implemented a function which makes many tests by changing parameters, in order to compute an average error of prediction according to the band size. For each user, we choose a rated movie, then we remove the rate from the rating matrix. For each user, we generate several signatures. A single signature can be divided into several bands with variable band size, which is equivalent to different series of hash. Figure 3 show how many different similarities are computed as a function of the number of bands b.

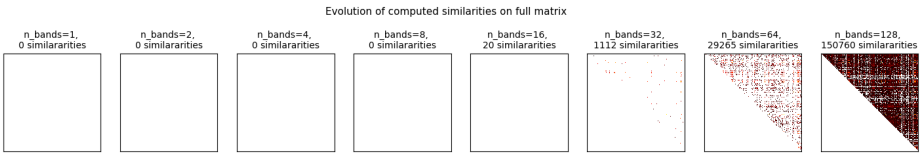

Figure 3: Evolution of the computed similarity with minhash algorithm

In our test for example, a signature length is a power of 2. Then we make the band size vary from 1 (2^0) to the signature length, by multiplying the band size by two at each iteration. The band size enables to compute the number of bands by a division of the signature length.

Besides, as we have several signatures set for users, we can compute several predictions.

So, for a fixed signature length, band size and rating threshold, we compute the mean squared error for the prediction and then we compute the average error for all users. Finally, we vary the band size and we can display a graph of the MSE approximations depending on the band size, as we can see in figure 4.

In figure 4, we can first observe the error slightly decreases before increasing after. Indeed, a small band size implies a small hash so a user has many neighbors in this case. Therefore, we will take into account many ratings from neighbors who are not compulsory very similar to the user. So in the second iteration, we could purify a bit the set of neighbors.

But globally, we can see that prediction is the most precise for small band size. In this case, many neighbors will influence the rate of the movie, which globally improves the accuracy of the prediction.

We also see that the error skyrockets rapidly to high values. Indeed, the ratings matrix is sparse. So in numerous cases, for a film rating we want to predict, no neighbor of the user has seen the movie. In our algorithm, by default, we return a 0 as prediction. Therefore, for many cases, the error is huge which parasitizes the error measure.

3.2 Comparaison between two models

In figure 4, we display the error according to the band size for a signature of 64 digits. We display the error generated by the combination of random hyperplanes and Pearson similarity computation and the combination of minhash and Jaccard similarity. We see that the first combination of methods (Random Vectors + Pearson) performs better than the second (minhash + Jaccard). In consequence, we will use the combination of random hyperplanes and Pearson similarity for the rest of the results.

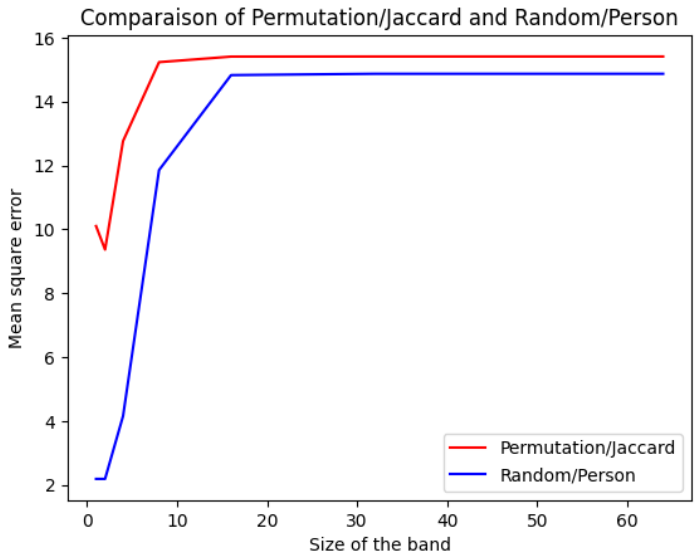

Figure 4: Comparison of the two implementations of LSH with a signature size of 64

3.3 The impact of the execution time

Although the small band sizes offer better results, the time complexity grows with the high number of neighbors selected.

We estimate the average time of execution to compute the top 10 movie suggestion for a user. For a user and a band size, we compute the time execution of a rating prediction for a movie. In a top 10 prediction, we have to predict all the ratings of unseen movies in order to suggest the 10 best ones. As the proportion of movies seen is very low, we can say that it is equivalent to do as many predictions as the number of movies in the database.
Besides, for obtaining a top 10 suggestion, the algorithm can fill an array of length 10 and at each prediction, put the new rating in the array if the rating is high enough and exclude the current lowest rated movie from the ranking array. That is to say that as the order is computed following each prediction, it does not add time complexity.

In conclusion, we just have to multiply the time taken by a single prediction by the number of movies and make an average on all users with one movie by user to get an estimation of the computation time according to the band size.
In an application like Netflix, let's say that we cannot accept a computation time higher than 1 second. In figure 5, we display the computation time in seconds depending on the band size, and another curve constant to one second.
So considering the error curve, the best band size we can handle is 8 digits, which makes a mean square error of 11.85, so an average error of 3.4.

3.4 The impact of the rating threshold

We saw in our tests that the threshold taken to binarize the input dataframe has almost no impact on the accuracy of the prediction. Here again, it could be explained by the sparsity of the rating matrix, making quite low the number of movies deleted in the shingling compared to the total number of movies.

4 Scaling the solution

Most of the exploring we made was on the smaller dataset. We did this to be able to explore how parameters b and r influenced the performances of the model and to compute the s-curves for all the

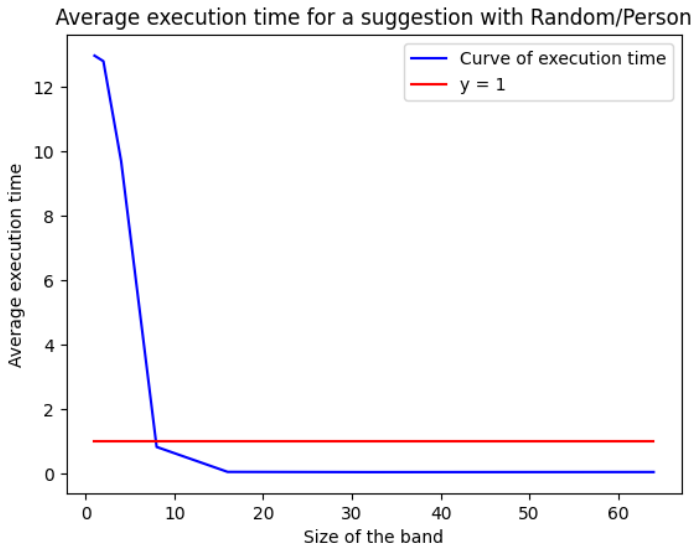

Figure 5: Time of computation for a single user

settings of parameters. But even with the small dataset we had to find strategies to optimize the computation in order to compute lots of tests.
By using LSH, the data on which the computation is made sees its size reduced from $\mathcal{O}(n_{\text{users}}.n_{\text{movies}}.8bits)$ to $\mathcal{O}(n_{\text{users}}.s_{\text{signature}}.8bits)$

4.1 Precompute the signatures

In the test described in 3.1, we use many hash tables in order to generate several signatures for each user and for which we change the number of band and the band size. But a single hash table finally returns the same global signature for a user. For a band size which will divide the signature, we want to avoid generating the hash for all users each time we have to put all users in buckets for example. Instead, for each hash table, we compute off-line all the complete signatures for all users. Then, when we want a specific band of a specific band size for a user, we just take the right portion in the signature, which is faster than computing the hash again.

4.2 Precompute the similarities

The Jaccard and Pearson similarities are very expensive in time computation. So we prefer to compute off-line all the similarities of all pairs of users and put it into the corresponding symmetric matrix. At each time we wanted to predict a rating, we just took similarities from the matrix, extracting the row of the corresponding user and multiplying it by the rating rows of the corresponding movie in the ratings matrix.

5 Conclusion

In this user-based collaborative filtering approach, we tried to predict unobserved ratings based on the similarity between users following the LSH algorithm. Implementing this model required tuning many parameters: rating threshold, signature length, band size, number of columns in a band as well as the hashing function and similarity metric (since the choice of hashing function is tightly linked to the similarity metric we're using). Besides, to attenuate the storage complexity, we computed the signatures tables (for different hashes) and the similarities tables of all pairs of users off-line. Finally, using the MSE, we noted that the combination of random hyperplanes and Pearson similarity is better than the combination of minHash and Jaccard similarity.