Advances in Data Mining
Assignment 2
Implementing Locality Sensitive Hashing

Xintong Jiang
2898144

Wei Chen
3156486

Abstract

In this assignment we applied 3 methods : Jaccard Similarity (JS), Cosine Similarity (CS), Discrete Cosine Similarity (DCS) to find pairs of most similar users of Netflix.

1 Data

The data comes from the original Netflix Challenge (www. netflixprize. com). It contains 65.225.506 ratings from 103703 users to 17770 movies.

2 Standard format of command line argument

Note that the format of the data file path (relative path) should be
very/long/path/to/data.npy
or
./very/long/path/to/data.npy
rather than
/very/long/path/to/data.npy (mentioned in the assignment requirement)
A correct command line that run our script could be:
python main.py -d
very/long/path/to/data.npy -s 2021 -m dcs
or in different order:
python main.py -m dcs -s 2021 -d
very/long/path/to/data.npy
In addition, we only receive js, cs or dcs after command **-m**.

3 Task 1 Jaccard Similarity

3.1 Method

Jaccard Similarity is defined as the size of the intersection of two sets divided by the size of their union. Since the data set is too large, we can not compare each user directly. We need to first "hash" each column C to a smaller signature and maintain the same similarity between groups. And then use the LSH to hash similar pairs in the same buckets to avoid comparing users to each other.
In this task, the ratings of every users were represented in columns and the ratings of every movies were represented in rows.
It is like table 1 and stored as a csc matrix.

	User ID 1	User ID2	...
Movie ID 1	1		...
Movie ID 2	2	2	...
Movie ID 3		2	...
Movie ID 4	4		...
Movie ID 5		5	...
...	...	...	...

Table 1: Data

For example, the user 1 is (1,2,4),the user 2 is (2,3,5),the similarity is their intersection/union is 0.2. This data set is small,instead of hash functions, random permutations of rows were applied. Then we look down permuted columns until we see a non-zero number. It is easy to do this with a csc matrix, because the row indices for column i are stored in indices[indptr[i]:indptr[i+1]]. When we want to find the first non-zero number of column, we only need to find the minimum of indices[indptr[i]:indptr[i+1]].We made 256 random permutations of rows in total.

Then columns of signature matrix M were divided into b bands of r rows. If two users have the same rows in one of the bands, they will be hashed into a same bucket. Number of bands and rows should be selected based on t $(1/b)^{1/r}, where is 0.5$ But in this task there is a time limit and we are not asked to find out similar pairs. So We first tried different number of rows (r) in each band (b) to find when there is not too many users in one bucket and then selected number band based on 30 minutes limit.
The way to hash users into baskets is setting their rows (tuple) as key and and their column numbers as values.

Buckets for a band	
Key	Value
(2,17,8,...,32)	11
(2,6,9,...,15)	12,66
...	...
(30,1,8,...,9)	13,19,45
(40,17,8,...,6)	14
(41,15,2,...,3)	15

Table 2: The storage way of the buckets

3.2 Result

We first tried different number of rows in each band and computed how many comparison (n) we need to compute in each band (average of 20 bands) as well as expected number of band we should make to find all similar pairs.

	n	expected number of band	all comparisons
r=6	$4.6 \cdot 10^6$	66	$3 \cdot 10^8$
r=7	$3.0 \cdot 10^6$	133	$4 \cdot 10^8$
r=8	$7.1 \cdot 10^5$	256	$2 \cdot 10^7$
r=9	$7.1 \cdot 10^5$	522	$4 \cdot 10^7$
...	...	...	...

Table 3: number of comparison

We found that it is reasonable to choose r=8, because it has lowest number of all comparisons we need to make. To avoid very large bucket, if a bucket that has more than 20

users in,it will be ignored. And we tested the running time on our own computer. Around 40 bands can be finished in 30 minutes. With seed=2021, we found 82 similar pairs in 30 minutes.

4 Task 2&3 (Discrete) Cosine Similarity

Due to the fact that Task 1 and Task 2 are similar in terms of implementation, these two are combined into the same section 2. The only difference is the

4.1 Data pre-processing

The original data set is converted into sparse matrix using **csr_matrix** from package scipy, where the shape of sparse matrix here is (User × Movie) → (103703, 17770).

4.2 Random projection

For generating signature matrix, we use minHash in Jaccard task, but in cosine similarity task, we use the ideas of random projection. Furthermore, random seed is used here to be able to repeatedly experiment with some hyperparameters.

However, instead of randomly generating the hyperplanes, picking random vectors that are normal to the hyperplanes would suffice and still has the same function.

For this reason, we generate **100** normal vectors instead and concatenate them into a 'normal vectors matrix' (here use only one line i.e., numpy.random.randn) with the shape (Movie, NormalVector) → (17770, 100). Next, we compute the dot product between the original data matrix and 'normal vectors matrix':

$$Signature\ Matrix = O \cdot N$$

where O and N denote the original data matrix and normal vectors matrix respectively. The shape of the resulting signature matrix is:

$$(103703, 100) = (103703, 17770) \cdot (17770, 100)$$

In addition, according to the definition of normal vector, a hyperplane is a set of points whose dot product with a normal vector to this hyperplane is 0. We can then yield the following inferences that indicate which side of a hyperplane the given points are in:

$$v \cdot A > 0 \rightarrow (< 90^0)$$

$$v \cdot B < 0 \rightarrow (> 90^0)$$

where v, A, and B are an normal vector and two different points (could be also seen as vectors), respectively, in a n-dims space. The inferences above means that points A ('above' a hyperplane) and B ('below' a hyperplane) are on different side of a hyperplane that has a normal vector v. Therefore, a user could be represented as a point (or vector) in a n-dims space where n is the number of movies this user rated. Here in our implementation for random projection, if the user A (point) is above the side of a hyperplane (i.e. $v \cdot A \geq 0$), we change the result of the dot product between v and A to 1, and if the user B (point) is below the side of a hyperplane (i.e. $v \cdot B < 0$), we change the result of the dot product between v and B to 0. For instance,

$$v \cdot A = 2.6 \rightarrow 1$$

$$v \cdot B = -0.1 \rightarrow 0$$

It can be found that a user (say user 1) might be represented as the following binary vector with the shape (1, 100) after random projection:

$$[0, 1, 1, 0, 1, 1, 0..., 1, 1, 1, 0]$$

where the 100 for column denotes the number of normal vectors (or hyperplanes) we selected. Accordingly, the signature matrix might be as follow (Table 4):

	User 1	...	User 103703
Normal Vectors 1	0	...	1
Normal Vectors 2	1	...	0
Normal Vectors 3	1	...	1
Normal Vectors 4	0	...	0
...	...	...	...
Normal Vectors 99	1	...	1
Normal Vectors 100	0	...	1

Table 4: Signature matrix

Now, we achieve the algorithm **random projection** that converts large sets to short signatures while preserving similarity.

4.3 LSH

In our implementation for LSH for cosine task (as well as discrete cosine), we set hyperparameters **h** (number of normal vectors), **b**, and **r** to **100**, **6** and **15** as default setting. Also, we have experimented other combinations of hyperparameters (see later subsection).

Firstly, banding strategy is applied to split the resulting signature matrix into $b = 6$ bands (or parts):

$$(103703, 100) \rightarrow 6 \times (103703, 15) + 1 \times (103703, 10)$$

Once $h = 100$ cannot be divisible by $b = 6$, the last part (103703, 10) will be ignored in our implementation.

Next, we hash the similar pairs whose representations (or signature) within a bind are the same, to the same buckets using dictionary. The keys are the signature vectors in string format, while their corresponding values are the UserIDs' of candidate pairs (could be more than 2 users) in string format. For example, consider the v, b, and r are 12, 3, 4 respectively for ease of demonstrating our idea. A bucket has 3 user X, Y and Z which means 3 candidate pairs (XY, YZ, XZ), and the userIds' for these 3 users are, say, 55, 66, 77. On top of that, the key for this bucket is '1101' because these three users share the same signature vector 1101. Hence, the possible form of a bucket in our implementation might be:

$$\{'1101' : [55, 66, 77]\}$$

The possible buckets dictionary responsible for a band (or part), where $r = 4$ might be (a bucket might have no candidate pair, e.g. Key 0111):

Buckets for a band	
Key	Value
1101	55, 66, 77
0001	1, 99321, 42, 3, 8671
...	...
0011	4, 15,101
1111	2, 88
0111	103703

Table 5: The storage way of the buckets for a band (r=4), where the Key stand for the unique identity of bucket and Value denotes the similar users (stored in ID format) with the same signature vector. Each row represents a bucket that contains at least one user.

In addition, we set up a list to store dictionaries responsible for distinct bands in case of collision. Then, in this way, we keep unique.

Finally, we iterate over the all buckets (rows) from all bands to compute the cosine similarity or discrete similarity based won the vectors in the original data matrix, to write the 'real similar pairs' to local .txt files. The similarity computation for both the cosine and discrete cosine are the same:

$$cos(\theta) = \frac{a \cdot b}{|a| \cdot |b|}$$

$$\theta = arccos(cos(\theta))$$

$$cosinesimilarity = 1 - \frac{\theta}{180}$$

where a and b are the user vectors with the shape (1, 17770), which define the angle θ.

The process of selecting buckets to be tested is roughly described above. We thentake a example below to make it more clear: Suppose we are now visiting a bucket at the last 3rd line in Table 5 (i.e., 0011: [4,15,101]), which means there are 3 candidate pairs(4&15,15&101,4&101) waiting to be tested. Next, we take user vectors that correspond to these 3 candidate pairs' UserID for similarity computation respectively. If a result is greater than the threshold (0.73), we write the corresponding UserId of the candidate pair to the local file.

4.4 Difference between cosine similarity and discrete cosine similarity

. The only difference between cosine and discrete cosine for similarity calculation is the representation of user vectors (or points). In other words, they differ in the representation of the original data matrix when doing similarity computation. For example, suppose we have two user vectors with the shape (1,17770) shown below, say usr1 and usr2, from the original data matrix. To do the similarity calculation for cosine, the form of the vector should be:

$$usr1 \rightarrow [0, 0, 5, 0, 0, ..., 0, 0, 3]$$

$$usr2 \rightarrow [1, 0, 0, 0, 0, ..., 4, 0, 0]$$

However, to compute the discrete similarity, we need to make a slight change to the form of user vector:

$$usr1 \rightarrow [0, 0, 1, 0, 0, ..., 0, 0, 1]$$

$$usr2 \rightarrow [1, 0, 0, 0, 0, ..., 1, 0, 0]$$

where every non-zero ratings (or elements) is replaced by 1.

4.5 Performance of different selection of hyperparameters

Even though for our implementation, the default hyperparameters setting is $b = 6, r = 15, h = 100$ mentioned above, we also experiment with other combinations of hyperparameters.

In this section, we demonstrate the performance of different selection of hyperparameters with tables.

Hyperparameters and results with seed 2021			
h	b	r	Result
100	6	15	A single run takes less than 30 minutes basically and the number of output (real similar pairs) are around 150 for both 2 task on average
100	2	50	Fast running time but almost no real similar user pairs for both 2 task
80	4	20	Running time is at least 4 times faster than the 1st try but there are no too many real similar user pairs output (around 40 for cosine and 30 for discrete cosine)
80	8	10	Relatively slow running time compared to the 1st and 3rd try but the number of output is greater than the 3rd try and intuitively this number is more likely to be greater than the 1st try in a single run time
150	3	50	Fast running time but no real similar user pairs for both 2 task
150	4	30	Fast running time but almost no real similar user pairs for both 2 task (only one pair for discrete cosine)

Table 6: Performance (or results) of different selection of hyperparameters

4.6 Conclusion

From the table 6, it can be seen that the running time is proportional to the hyperparameters **b**, and the smaller b and the larger r lead to the smaller number of user vectors being hashed to the same buckets which means the majority of buckets containing not too many candidate pairs being picked as the real similar user pairs. Furthermore, it can be found that once the value $\frac{1}{b}^{\frac{1}{r}}$ are adjusted to a small value, say 0.5, it results in buckets containing considerable number of candidate pairs which need more comparisons which means it takes more time to do the similarity computation. We also found that the number of the real similar user pairs for the discrete cosine task tends to be greatly less than the number of those for the cosine task and the discrete cosine task tends to take more time than the cosine task.

Instead of generating hyperplanes, we generate random normal vectors to achieve the random projection algorithm. The difference between cosine and discrete cosine is the representation of user vectors when calculate the similarity of two user vectors. Finally, different random seed result in different number of real similar pairs.