Data Mining
Assignment 2 : Locality Sensitive Hashing

Sasank Amavarapu : s4047303
Ayush Sengar : s3528995
Group : 43

November 27, 2023

Introduction

This assignment focuses on applying Locality Sensitive Hashing (LSH) to identify similar Netflix users, utilizing Jaccard, Cosine, and Discrete Cosine Similarity metrics. Our study refines the original Netflix Challenge dataset, which initially contained around 500,000 users, by focusing on a subset of users who rated between 300 and 3000 movies. This narrowed dataset, renumbered sequentially for user and movie IDs, consists of 65,225,506 records, each with a user ID, movie ID, and movie rating. This approach aims to reveal insights into user preferences on Netflix, enhancing understanding in user behavior analysis and recommender systems.

1 Jaccard Similarity

Jaccard Similarity is a measure of the likeness between two sets, quantified as a percentage score from 0% to 100%. The higher the score, the greater the similarity between the sets.

Consider two sets, S and T. Their Jaccard Similarity is calculated using the formula:

$$\text{SIM}(S,T) = \frac{|S \cap T|}{|S \cup T|}$$

This formula represents the ratio of the intersection (common elements) to the union (total unique elements) of sets S and T. Essentially, it reflects the extent of overlap between sets, such as user preferences or item selections, with a higher score indicating more significant similarity.

Calculating this similarity across a large dataset with billions of user pairs is impractical due to its sheer size. To circumvent this, we use minhashing and banding techniques, enabling us to efficiently compute Jaccard Similarity for large-scale data.

1.1 Import Data

For Jaccard Similarity, we focus on whether users have rated the same movies, regardless of the rating value. We use a Sparse Matrix format, specifically the Compressed Sparse Row Matrix from SciPy, to represent this data. In this matrix, users are rows, movies are columns, and a value of 1 indicates a user has rated a movie (0 otherwise). The `csc_matrix((data, (user_ids, movie_ids)), dtype=np.int8)` function constructs this binary representation, effectively capturing user-movie interactions.

1.2 Parameters

- **Number of Rows (r):** A total of 70 rows are used, indicating the permutation of rows during the computation of the signature matrix.

- **Number of Bands (b):** The algorithm employs 30 bands in the process.
- **Max Bucket Size (max_bucket_size):** To optimise performance and prevent the creation of excessively large buckets, a maximum bucket size of 800 is chosen based on hint 4.
- **Threshold:** The threshold for a pair to be considered similar is 0.5.

1.3 MinHashing

Minhashing is a technique designed to estimate the similarity between sets through the use of hash functions. In our scenario, the application of Minhashing involves the computation of MinHash values for columns corresponding to movie IDs within a SparseMatrix. These MinHash values essentially indicate the first row index where a column contains the value 1 in a permuted sequence.

To create Minhash Signatures, we introduce an additional step. Random permutations of rows are selected. For each column, we then calculate MinHash values based on these permutations. This systematic process results in a signature matrix. In this matrix, each column serves as the Minhash Signature for a specific column in the dataset.

In essence, Minhashing helps us gauge the likeness between sets by leveraging hash functions, and the resulting signature matrix encapsulates this similarity estimation for each column in our dataset.

1.4 Buckets

The next phase involves repeatedly hashing items to ensure they're grouped together if they're similar. By doing this, we pinpoint pairs within these groups and determine their similarity. The goal is to exclude dissimilar pairs from contributing to the similarity calculation. However, this method might accidentally group dissimilar pairs together, leading to false matches. To accomplish this, we employ a technique called banding, dividing the signature matrix into several bands, each consisting of multiple rows. In each band, a hash function operates on subsets of the matrix, directing them to various buckets. With separate bucket arrays for each band, columns sharing similar subsets across bands won't hash into the same bucket.

1.5 EXPERIMENTS AND RESULTS

Note: The results populated were computed on Apple MacBook Air with 8GB ram as suggested and Apple M2 chip.*

We experimented with different r & b combinations (at a fixed seed of 82) to arrive at the ideal pairs which would compute within 30 minutes and give an output of above 50 similar pairs.

Table 1: Parameter tuning JS

Experiment No	Random Seed	Rows (r)	Bands (b)	Number of pairs	Execution time (mins)
1	82	**60**	**30**	**365**	**1.2**
2		25	25	33	0.47
3		30	40	0	0.13
4		60	35	63	0.62
5		**70**	**30**	**352**	**1.31**

We started our experimentation with $(r, b) : (25, 25)$. We tried increasing the number of rows & bands to (60, 30) and were able to compute more than 50 pairs in the expected time. We tried decreasing the rows

to 30 and bands to 40 which resulted in 0 pairs. Then we decided to increase (r, b) from (60, 30) to (60, 35) & (70,30).

Out of our experiments (r, b) values being (60, 30) & (70,30) gave better results.

The ideal set with more than 50 pairs and execution time is less than 30 minutes for

$(r, b) : \{(60, 30); (70, 30)\}$. For each of these sets we have tested on 4 different seed values to check for consistency of producing more than 50 similar pairs of users:

Table 2: Experiment Results JS

Experiment No	Random Seed	Rows (r)	Bands (b)	Number of pairs	Execution time (mins)	Mean Execution time (mins)
1	50	**60**	**30**	438	1.42	1.35
2	82			365	1.2	
3	100			439	1.53	
4	182			297	1.25	
5	50	**70**	**30**	423	1.5	3.03
6	82			352	1.31	
7	100			452	8	
8	182			310	1.32	

We found that for the values of $(r, b) : \{(60, 30); (70, 30)\}$ we find the ideal results as per the standards mentioned in the assignment across different seed values. We also find the average in the number of pairs computed for both the set of values is very similar. We have the best (or maximum as per the assignment guidelines) in the number of pairs computed for **(r, b) : (70,30)** for seed value 100, hence we have used these values in our final code.

2 Task 2: Cosine Similarity

Cosine similarity is a crucial metric for assessing the similarity between two vectors, often used in analyzing movie ratings. In Python, it's computed by determining the cosine of the angle α between two vectors, as per the formula:

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

where $\mathbf{a} \cdot \mathbf{b}$ is the dot product of the vectors and $\|\mathbf{a}\|$, $\|\mathbf{b}\|$ are the magnitudes of the vectors. The angle α is calculated using:

$$\alpha = \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|}\right)$$

and converted to degrees. Cosine similarity, ranging between 0 and 1, is then obtained by:

$$\text{CosineSimilarity} = 1 - \frac{\text{angle\_degrees}}{180}$$

Higher values indicate stronger similarity.

2.1 Import Data

The dataset comprises individual user ratings. We process this data by separating user IDs, movie IDs, and ratings into lists named `rowU`, `columnM`, and `valueR`. These are then used to create a CSR matrix (`matCosine`) via SciPy's `csr_matrix`, where movies form columns and users form rows. The CSR format is ideal for handling large datasets, allowing efficient computation and neatly organizing user ratings.

2.2 Random Projection and Banding

- **Initialization for Reproducibility:** We begin by setting a seed for the random number generator. This ensures that our results are reproducible, a critical aspect in scientific computations.
- **Dimensionality Extraction and Projection Vector Creation:** After determining the dimensions of the dataset (N data points and D features) from the input data X, we generate a matrix of projection vectors (`projection_vectors`). These vectors, drawn from a normal distribution, are key in reducing the dimensionality of the dataset.
- **Data Projection and Transformation:** The dataset is projected onto these vectors, resulting in `random_projections`. This step effectively transforms the data into a new, lower-dimensional space, specifically $N \times (b \times r)$, where b represents the number of bands and r is the number of rows in each band.
- **Binary Nature Assignment:** In the projection process, we assign binary values to the vectors based on their position relative to the hyperplane - '1' for vectors above and '0' for those below.
- **Matrix Reshaping and Banding:** The projection matrix is reshaped into a 3D format of shape N, b, r. This structure is essential for the banding process, which groups binary values into bands. Banding is a pivotal technique in efficiently identifying candidate pairs that exhibit similar characteristics within the same band.
- **Application in LSH:** This approach to banding significantly enhances the efficiency of the Locality-Sensitive Hashing (LSH) algorithm. It streamlines the process of identifying potential matches, thereby improving the efficacy of similarity assessments in large datasets.

2.3 Hashing

Following the matrix reshaping and banding step, we proceed with the hashing process, which involves the creation of hashed values for efficient similarity assessments. The operation starts with the formulation of a set of values using the power of three. This set is constructed by repeating an array of row indices (np.arange(num_rows)) for each band across all users. The resulting array is then replicated for each user in the dataset. This array, denoted as values, serves as the basis for subsequent operations.

Next, we perform an element-wise multiplication between the binary projection matrix (random_projections) and the values array. This multiplication aligns the binary values with the corresponding power of three, forming a set of scaled values that retains the binary essence introduced during projection.

Subsequently, we sum along the third axis (axis=2), collapsing the matrix into a two-dimensional array (hashed_values). This array represents the hashed values derived from the binary projections. Each element in hashed_values encapsulates the cumulative effect of the binary values scaled by powers of three, providing a concise representation of the original data in a hashed form.

In essence, this hashing process efficiently condenses the binary projections into a set of hashed values, laying the groundwork for the application of Locality-Sensitive Hashing (LSH) in identifying candidate pairs with similar characteristics

2.4 Parameters

- **Number of Rows (r):** A total of X rows are used, indicating the permutation of rows used during the computation of the signature matrix.
- **Number of Bands (b):** The algorithm employs X bands in the process.
- **Threshold:** The threshold for a pair to be considered similar is 0.73.

2.5 EXPERIMENTS AND RESULTS

Note: The results populated were computed on Apple MacBook Air with 8GB ram as suggested and Apple M2 chip.*

We experimented with different b & r combinations (at a fixed seed of 82) to arrive at the ideal pairs which would compute within 30 minutes and give an output of above 50 similar pairs.

Table 3: Parameter tuning CS

Experiment No	Random Seed	Bands (b)	Rows (r)	Signature Length (b*r)	Number of pairs	Execution time (mins)
1	82	20	25	500	50	5.81
2		25	20	500	224	33.38
3		**25**	**25**	**625**	**82**	**7.92**
4		**30**	**25**	**750**	**66**	**8.05**
5		**35**	**25**	**875**	**81**	**5.7**
6		25	30	750	14	4.49

We started our experimentation with $(b, r) : (25, 25)$. We tried reducing the rows while keeping the bands constant, though the resulting pairs increased still the execution time was higher than 30 minutes. We tried reducing the bands keeping the rows at constant and found the pairs to be dropping. Also, we have tried increasing rows while keeping the bands at constant and found pairs to be dropping. Hence we finally fixed the rows at our sweet point of 25 and then started to increase the bins to 30 and 35, and hence also increasing the signature length from 625 to 750 and 875 respectively.

The ideal set with more than 50 pairs and execution time is less than 30 minutes for

$(b, r) : \{(25, 25); (30, 25); (35, 25)\}$. For each of these sets we have tested on 4 different seed values to check for consistency of producing more than 50 similar pairs of users:

Table 4: Experiment Results CS

Experiment No	Random Seed	Bands (b)	Rows (r)	Signature Length (b*r)	Number of pairs	Execution time (mins)	Mean Execution time (mins)
1	50	**30**	**25**	**750**	122	10.38	9.21
2	82				66	8.05	
3	100				88	8.76	
4	182				63	9.66	
5	50	**35**	**25**	**875**	95	5.57	6.18
6	82				81	5.7	
7	100				106	6.36	
8	182				86	7.08	
9	50	25	25	625	41	15.09	9.07
10	82				82	7.85	
11	100				41	6.48	
12	182				39	6.84	

We found that for the values of $(b, r) : \{(30, 25); (35, 25)\}$ we find the ideal results as per the standards mentioned in the assignment across different seed values. Though $(25, 25)$ also gave an ideal value for one

value of random seed, for the other values we get similar pairs as less than 50. We have the best average in the number of pairs computed for **(b, r) : (30,25)** across different seed values, hence we have used these values in our final code.

3 Task 3: Discrete Cosine Similarity

Discrete Cosine similarity is a crucial metric for assessing the similarity between two discrete vectors, often used in analyzing movie ratings. In Python, it's computed by determining the cosine of the angle α between two vectors, as per the formula:

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

where $\mathbf{a} \cdot \mathbf{b}$ is the dot product of the vectors and $\|\mathbf{a}\|$, $\|\mathbf{b}\|$ are the magnitudes of the vectors. The angle α is calculated using:

$$\alpha = \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|}\right)$$

and converted to degrees. Cosine similarity, ranging between 0 and 1, is then obtained by:

$$\text{CosineSimilarity} = 1 - \frac{\text{angle\_degrees}}{180}$$

Higher values indicate stronger similarity.

3.1 Import Data

The data representation process begins by sorting all ratings based on the corresponding users, resulting in each item representing a user and their associated ratings. To handle the dataset's substantial volume more efficiently, SciPy Sparse Matrices, specifically the Compressed Sparse Row matrix (csr matrix), are employed. This matrix representation organises the data with movie IDs as column indices, user rows as row indices, and discrete ratings (0 or 1) as values. A rating of 0 indicates no rating, while a rating of 1 signifies that a rating was given. This approach optimises computational efficiency in managing large datasets.

3.2 Random Projection & Banding and Hashing

Random Projection and Banding and hashing has been implemented exactly as described in the above task for cosine similarity.

3.3 Parameters

- **Number of Rows (r):** A total of X rows are used, indicating the permutation of rows used during the computation of the signature matrix.
- **Number of Bands (b):** The algorithm employs X bands in the process.
- **Threshold:** The threshold for a pair to be considered similar is 0.73.

3.4 EXPERIMENTS AND RESULTS

Note: The results populated were computed on Apple MacBook Air with 8GB ram as suggested and Apple M2 chip.*

We experimented with different b & r combinations (at a fixed seed of 82) to arrive at the ideal pairs which would compute within 30 minutes and give an output of above 50 similar pairs.

Table 5: Parameter tuning DCS

Experiment No	Random Seed	Bands (b)	Rows (r)	Signature Length (b*r)	Number of pairs	Execution time (mins)
1	82	30	20	600	164	36.16
2		30	25	750	24	3.29
3		30	30	900	3	1.22
4		40	20	800	NA	Killed after 120
5		35	25	875	44	6.06
6		**40**	**25**	**1000**	**62**	**8.95**
7		45	25	1125	33	8.06
8		**35**	**23**	**805**	**114**	**17.6**

We started our experimentation with $(b, r) : (30, 30)$. We tried reducing the rows while keeping the bands constant, though the resulting pairs increased still the execution time was higher than 30 minutes. We found 164 pairs with $(b, r) : (30, 30)$, but time just over the limit of 30 minutes. We tried increasing the bands to 40 with the same number of rows, which took a lot of time to compute before we had to kill it. Then by keeping rows at constant of 25, we experimented with 35, 40 & 45 as values for bands. We observed an increasing trend of pairs with increase in bands till 40. Hence we fixed our first set of $(b, r) : (40, 25)$. We have also found an increase in pairs with decrease in number of rows, but also a significant increase in execution time. Hence we fixed our second set of $(b, r) : (35, 23)$.

Finally the ideal set with more than 50 pairs and execution time is less than 30 minutes for $(b, r) : \{(40, 25); (35, 23)\}$. For each of these sets we have tested on 4 different seed values to check for consistency of producing more than 50 similar pairs of users:

Table 6: Experiment Results DCS

Experiment No	Random Seed	Bands (b)	Rows (r)	Signature Length (b*r)	Number of pairs	Execution time (mins)	Mean Execution time (mins)
1	50	**40**	**25**	**1000**	61	6.82	7.51
2	82				62	8.95	
3	100				47	7.35	
4	182				62	8.92	
5	50	**35**	**23**	**805**	80	19.3	15.78
6	82				114	17.61	
7	100				73	15.71	
8	182				55	10.5	

We found that for the values of $(b, r) : \{(40, 25); (35, 23)\}$ we find the ideal results as per the standards mentioned in the assignment across different seed values, except in one case (seed=100) for the former set where the number of pairs is less than 50. We have the best average in the number of pairs computed for **(b, r) : (35,23)** across different seed values, hence we have used these values in our final code.