Assignment 2
Locality Sensitive Hashing
Group 5

Praneeth dathu (4174089)
Po-Chin, Chang (4272455)
Priyanjali Goel (4206559)
Luuk Deen (2984679)

April 25, 2025

Abstract

This report focuses on implementing Locality Sensitive Hashing (LSH)[3] for efficiently identifying similar user pairs in the Netflix dataset. By leveraging techniques such as Minhashing[1] and banding, we aim to optimize performance while reducing computational complexity. The report outlines the methodology, experimental setup, and future scope for improving the approach.

1 Introduction

Finding similar user pairs in large-scale datasets is a common challenge in data mining. The Netflix dataset, with over 65 million records, presents a significant computational burden for similarity detection. This assignment focuses on implementing Locality Sensitive Hashing (LSH) to efficiently identify user pairs based on Jaccard Similarity[2]. Unlike computationally expensive brute-force methods, LSH leverages hashing techniques to reduce the complexity of identifying candidate pairs.

The similarity between users is measured by the sets of movies they rated, with a threshold of Jaccard Similarity > 0.5 used to identify similar pairs. This project involves:

- Preprocessing the dataset for efficient storage and manipulation.
- Implementing Minhashing and banding techniques.
- Evaluating the performance of the LSH algorithm.

2 Dataset Description

The dataset consists of 65,225,506 records in the format `(user_id, movie_id, rating)`. Users with fewer than 300 or more than 3000 ratings were excluded to ensure data consistency. The dataset is highly sparse, making it suitable for sparse matrix representation.

Key properties:

- Total users: Approximately 100,000.
- Total movies: 17,770.
- Average ratings per user: Between 300 and 3000.

3 Methodology

3.1 Compressed Sparse Row

Compressed sparse row (CSR) [4] is a data structure for efficiently storing sparse matrix. It compresses the matrix space by representing non-zero elements and their position with data value, indices for the column, and row pointers for indicating the start index of each row. This method suggests to compress the row indices to save memory usage. In this assignment, we first use CSR to store data.

3.2 Minhasing

Minhashing is an efficient technique for further compressing sparse data. By using a predetermined number of hash maps, Minhashing can reduce the rows in a matrix to a fixed size. The result of Minhashing, known as the signature matrix, represents the features for each user ID. Additionally, the signature matrix can be used to approximate the similarity between user IDs.

In our implementation, we created 100 hashmaps, with each having random permutations of the movie IDs since the dataset is not that enormous. For each iteration of the hashmap, we find the minimum hash value with non-zero entries across the users. Also, creating hashmaps for all 100 of them before applying the minhashing algorithm reduced our running time from 10 mins to 2 mins.

3.3 Locality Sensitive Hashing

While minhashing compacts the sparse information for movie IDs, the dataset still contains a large number of user IDs. Locality Sensitive Hashing (LSH) addresses this by dividing the signature matrix into several bands. Instead of directly comparing bands, each band is passed through hash functions to generate hash values, which represent buckets.

To finalize a hash function, we tried a couple of methods like applying using a mod with hashlib, but that increased the computation time vastly and could be biased because of mod. We also considered, adding the values and considering them as a hash but that led to a lot more false positives.Finally, to avoid any bias and good computation time, we applied a built-in hash function from Python that will be applied on each band as a tuple.

Consequently, users with the same hash values are stored in the same bucket, efficiently identifying candidates with high similarity.

3.3.1 Adding pairs to buckets

Initially, we began applying the hash function directly on the tuple of the band. But that decreased the number of collisions as there are multiple values that replicate differently.

To increase collision, we found that applying the hash function over the tuple of each band with a sorted and unique set of these signature values. This led to more collisions of similar pairs in the bucket and optimized the runtime and memory. Although this operation may introduce more false positives, the hash function effectively separates similar users into different buckets. To remove the false positives, we use Jaccard's similar measure in a later stage.

3.4 Post-processing

While adding the candidate pairs to a list from the buckets, we take unique pairs of sorted user IDs avoiding the duplicate pairs identified among buckets, eventually optimizing the similarity calculation on unique pairs. Additionally, the buckets with only one user will not be considered.

After obtaining candidate pairs from LSH, we calculate their similarity using Jaccard Similarity as Eq.1, where S_1 and S_2 represent the sets of movies rated by two users. Following the given instructions, we filter out pairs with a similarity score above 0.5 to produce the final result. Once all the pairs are captured, we sort them based on the similarity score obtained in ascending order. As the similarity is calculated, the pairs are added to a text file.

$$JS(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{1}$$

3.5 Parameter Selection

Best Key parameters used in the implementation:

- Signature length (h): 100
- Rows per band (r): 8
- Number of bands (b): 12

Theoretically, fewer rows per band increase the likelihood of hash collisions, as the elements within each band are fewer. On the other hand, increasing the signature length with a fixed number of rows per band results in more bands. This creates more opportunities for users to be placed into the same bucket. However, this also requires more computational resources.

4 Experimental Results

Results of the LSH implementation will include:

- Total similar pairs identified.
- Average Jaccard Similarity of identified pairs.
- Execution time on provided hardware.

4.1 Results

Using the above methodology, we obtained 409006 candidate pairs. After computing the actual similarity using the Jaccard similarity measure with a threshold above 0.5, 125 user pairs were identified, as shown in Fig. 1. The average similarity among these pairs was 0.52. Furthermore, our results are generated by a random seed of 5 to ensure the results can be reproduced.

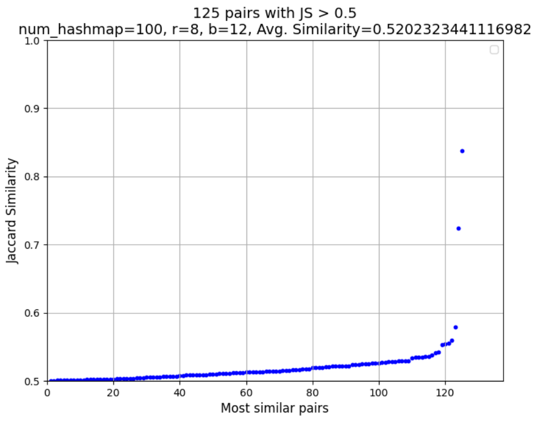

Figure 1: Results of the LSH algorithm: 125 user pairs with a Jaccard similarity greater than 0.5 were identified.

4.2 Analysis

To gain deeper insights, we analyzed the effects of varying the number of hash maps in minhashing and the number of rows per band in LSH. We also evaluated the number of final pairs and the runtime.

For the number of final pairs, increasing the number of hash maps resulted in more final pairs. This occurs because a larger number of bands increases the likelihood of collisions in the hash function in LSH, thereby generating more candidate pairs. Similarly, reducing the number of rows per band also resulted in more final pairs due to fewer elements in each band. Our result is fit with the theory. A smaller band and larger signature length will provide more final pairs.

Regarding runtime, using fewer rows per band increased computation time, as it generated more candidate pairs for subsequent similarity calculations. Additionally, increasing the number of hash maps required more processing time, as it increased both the number of bands in LSH and the computational load of minhashing.

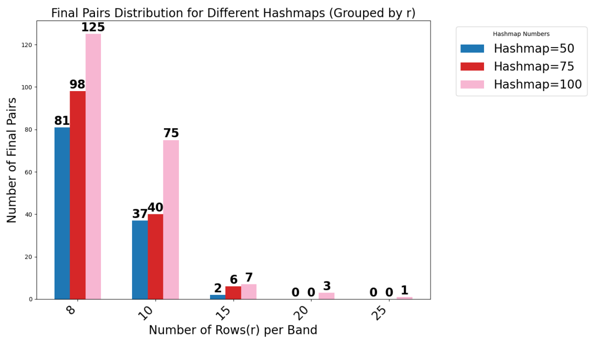

Figure 2: Comparison of the number of final pairs. The X-axis represents the number of rows per band, and the Y-axis represents the number of final pairs. Configurations with fewer rows or more hash maps generated more final pairs.

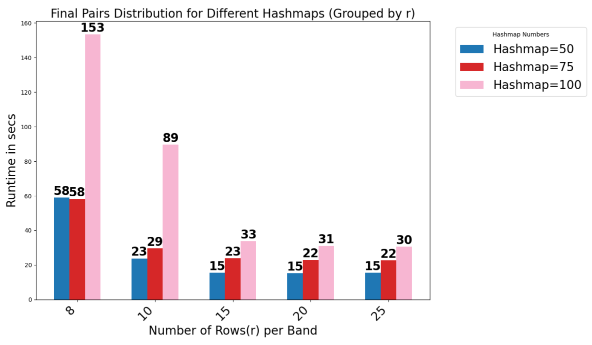

Figure 3: Runtime comparison. The X-axis represents the number of rows per band, and the Y-axis represents runtime in seconds. Configurations with fewer rows and more hash maps consumed more computational resources.

5 Conclusion

In this project, we implemented Compressed Sparse Row (CSR), MinHashing, Locality Sensitive Hashing (LSH), and Jaccard Similarity to identify similar user pairs. Using the determined signature length, row, and band numbers, our approach identified 125 user pairs with a similarity score above 0.5. While the ground truth contains 1,205 user pairs, our algorithm demonstrated significant computational efficiency compared to brute-force similarity computation.

Our analysis reveals that parameter selection plays a crucial role in determining the final results. The experimental outcomes align closely with theoretical expectations, and the observed overall trends correspond well to the predicted patterns.

Overall, this project underscores the efficiency of Locality Sensitive Hashing for large-scale similarity detection. The implementation successfully reduced computational complexity while identifying similar user pairs. Future work will focus on exploring alternative hashing techniques and optimizing parameters to further improve performance.

Acknowledgments

We thank our instructor for their guidance and our teammates for their collaboration.

References

[1] A.Z. Broder. "On the resemblance and containment of documents". In: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*. 1997, pp. 21–29. DOI: 10.1109/SEQUEN.1997.666900.

[2] Paul Jaccard. “Etude de la distribution florale dans une portion des Alpes et du Jura”. In: *Bulletin de la Societe Vaudoise des Sciences Naturelles* 37 (Jan. 1901), pp. 547–579. DOI: `10.5169/seals-266450`.

[3] Omid Jafari et al. *A Survey on Locality Sensitive Hashing Algorithms and their Applications.* 2021. arXiv: `2102.08942 [cs.DB]`. URL: `https://arxiv.org/abs/2102.08942`.

[4] *Sparse matrix - Wikipedia — en.wikipedia.org.* `https://en.wikipedia.org/wiki/Sparse_matrix`. [Accessed 09-12-2024].