Advances in Data Mining Final Assignment: Implementing Locality-Sensitive Hashing for Finding Similar Netflix Users

Monish Shah (4401379)   Manuel Perez Belizon (4416880)

Uzer Ahmed

December 11, 2024

1 Introduction

This project aims to identify pairs of Netflix users with similar movie preferences based on their rated movies. The similarity metric used is the **Jaccard similarity**:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where A and B are the sets of movies rated by User 1 and User 2, respectively. Users with $J(A, B) > 0.5$ are considered similar. Identifying such similar user pairs is instrumental for enhancing recommendation systems, improving user engagement, and personalizing content delivery on platforms like Netflix.

With a dataset comprising approximately **100,000 users**, **17,770 movies**, and **65 million rating records**, a brute-force similarity computation (5 billion pairs) is computationally infeasible. To address this challenge, **Locality-Sensitive Hashing (LSH)** combined with **MinHash signatures** is utilized [1]. This approach approximates similarities and efficiently narrows down the search to likely similar pairs, significantly reducing computational overhead while maintaining high accuracy in similarity detection.

2 Data Representation

Efficient data representation is crucial for handling large-scale datasets. A **Compressed Sparse Row (CSR) matrix** from the `scipy.sparse` library stores user-movie interactions, offering memory savings and fast row slicing to retrieve movies rated by each user. Rows represent users, columns represent movies, and non-zero entries indicate ratings (converted to 1 for binary representation). This facilitates efficient extraction of each user's rated movies.

Additionally, a **NumPy 2D array** stores, for each user, the number of movies rated and their corresponding movie indices. This structure enables quick access during signature computation and similarity verification.

To manage memory during LSH banding, candidate pairs are first collected in a **Python list** and later converted to a NumPy array with duplicates removed using

NumPy's `unique` function, significantly reducing memory consumption compared to using a Python set.

Data Structure	**Library**	**Purpose**
CSR Matrix	`scipy.sparse`	Stores user-movie interactions efficiently, allowing fast row-wise access to retrieve movies rated by each user.
NumPy 2D Array	`numpy`	Stores, for each user, the number of movies rated and the corresponding movie indices, facilitating quick access during signature computation and similarity verification.
Python List / NumPy Array	`python/numpy`	Collects candidate pairs during LSH banding. Initially stored in a list for memory efficiency, then converted to a NumPy array for deduplication using `unique`.

Table 1: Data Structures Used in the Implementation

3 Methodology

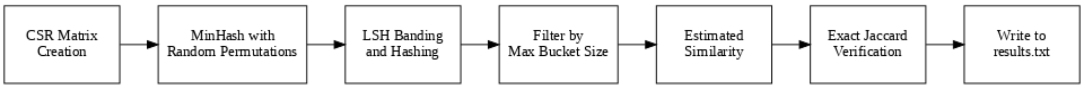
CSR Matrix Creation
MinHash with Random Permutations
LSH Banding and Hashing
Filter by Max Bucket Size
Estimated Similarity
Exact Jaccard Verification
Write to results.txt

Figure 1: Flowchart of the Methodology for Identifying Similar Netflix Users

3.1 Data Loading and Preprocessing

The user-movie rating data is loaded using NumPy's memory-mapped file (`mmap_mode='r'`) to handle large files without full memory loading. User and movie IDs are adjusted to 0-based indexing. A CSR matrix is created to efficiently represent user-movie interactions. Subsequently, a 2D NumPy array is initialized to store the count and indices of movies rated by each user.

3.2 MinHash Signature Computation

MinHash signatures approximate Jaccard similarity without explicit intersection and union computations. This involves generating n random permutations of movie indices, where n is the signature length (100). For each permutation, the rank of each movie is computed. For each user and hash function, the minimum rank among their rated movies is identified and stored as the user's MinHash signature.

The expectation and variance of the MinHash estimator are given by:

$$\mathbb{E}[\hat{J}(A, B)] = J(A, B)$$

$$\mathrm{Var}[\hat{J}(A, B)] = \frac{J(A, B)(1 - J(A, B))}{n}$$

where $\hat{J}(A, B)$ is the estimated Jaccard similarity using MinHash signatures, and n is the number of hash functions (signature length).

3.3 Locality-Sensitive Hashing (LSH) Banding

LSH reduces the number of candidate pairs by grouping similar signatures into the same buckets. For each band, the sub-signature is hashed into a bucket using a hash function (we used MD5 hashlib to keep the similar pairs found more consistent across different seeds). Users with identical sub-signatures in a band are placed in the same bucket. Within each bucket, all possible user pairs are generated. To optimize memory usage and prevent combinatorial explosion from large buckets, a maximum bucket size threshold (we used 2000 as we were able to get a low time and thus only removed buckets if they were extremely big) is set. Candidate pairs are collected in a list and later deduplicated using NumPy's `unique` function.

The probability that two users with Jaccard similarity s will be placed in the same bucket in at least one band is:

$$P_{\text{collision}} = 1 - (1 - s^r)^b$$

This equation indicates that as the number of bands b increases or the number of rows per band r increases, the probability of similar pairs colliding increases.

3.4 Exact Jaccard Similarity Verification

To ensure accuracy, Jaccard similarity is computed exactly for each candidate pair identified by LSH. An estimated similarity check precedes the exact computation to optimize performance. First, the fraction of matching MinHash signature elements is calculated for each candidate pair:

$$\hat{J}(A, B) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\text{signature}_i(A) = \text{signature}_i(B))$$

where $\mathbb{I}$ is the indicator function. Pairs with $\hat{J}(A, B) \geq 0.4$ are considered for exact computation.

Exact Jaccard similarity is then computed as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Pairs exceeding the final threshold ($J(A, B) > 0.5$) are retained. Using 0.4 instead of 0.5 for the estimated check ensures that no potentially similar pairs are prematurely discarded, while the final threshold guarantees the accuracy of the identified similar pairs. This two-step verification method significantly reduces computation time by eliminating unlikely dissimilar pairs before performing the more expensive exact Jaccard similarity calculations.

4 Results

Experiments were conducted with various parameter configurations to evaluate the effectiveness and efficiency of the LSH implementation. The selected parameters for the final implementation are $b = 20$, $r = 5$, $n = 100$, and $max\_bucket\_size = 2000$, offering the best balance between the number of candidate pairs, similar pairs identified, and computation time.

4.1 Experimental Results

Table 1 presents results from different parameter configurations, while Table 2 shows results across five different random seeds using the optimal configuration. Figure 2 visualizes the distribution of found pairs across different seeds.

Seed	**Signature**	**Bands**	**Rows**	**Bucket**	**Threshold**	**Candidate**	**Similar**	**Time (s)**
359	80	5	20	300	0.4	0	0	120
359	80	6	15	300	0.4	558,788	58	123
359	80	15	6	300	0.4	3,373,921	240	161
359	80	20	5	300	0.4	9,586,005	413	227
359	100	15	6	300	0.4	2,695,196	314	194
359	100	20	5	300	0.4	12,887,723	456	286.98
359	120	20	5	300	0.4	12,887,723	457	328
359	150	20	5	300	0.4	12,887,723	457	340
359	100	20	5	1000	0.4	19,023,358	652	367.21
359	100	20	5	100	0.4	6,581,117	279	120
359	150	30	5	300	0.4	21,013,732	618	430
359	**100**	**20**	**5**	**2000**	**0.4**	**21,792,361**	**756**	**395.46**

Table 2: Experimental Results Across Different Parameter Configurations

Observation: Table 1 illustrates that increasing the number of bands (b) and the signature length (n) leads to a higher number of candidate and similar pairs, accompanied by longer computation times. For example, with a signature length of 80, raising the number of bands from 5 to 20 results in an increase of similar pairs from 0 to 413 and candidate pairs from 0 to approximately 9.6 million, with computation time rising from 120 to 227 seconds. Similarly, extending the signature length to 100 while maintaining 20 bands increases similar pairs to 456 and candidate pairs to nearly 12.9 million, and computation time to 287 seconds. Additionally, allowing larger bucket sizes significantly boosts both candidate and similar pairs, as seen with 21.8 million candidate pairs and 756 similar pairs, though computation time extends to 395 seconds. These results highlight the trade-off between detection accuracy and computational resources: higher values of b and n enhance the probability of identifying true similar pairs but also generate more candidate pairs that require verification, thereby increasing processing time. Optimal parameter selection balances the number of similar pairs identified with manageable computation times, ensuring both efficiency and effectiveness in the LSH process. Keeping the Signature as 100 gives a good balance of candidate pairs generated and similar pairs found, meaning it gives us buckets with more similar pairs together while keeping the computation time manageable which is why $h = 100$, $b = 20$, and $r = 5$ was found to be the best for us.

Seed	Signature	Bands	Rows	Bucket	Threshold	Candidate	Similar	Time (s)
3350	100	20	5	2000	0.4	28,598,508	697	295
2336	100	20	5	2000	0.4	15,723,382	367	159.28
1556	100	20	5	2000	0.4	27,027,286	552	302
2355	100	20	5	2000	0.4	36,388,140	570	436
2923	100	20	5	2000	0.4	26,173,475	538	281
Average	100	20	5	2000	0.4	26,782,158	544	294

Table 3: Experimental Results Across Different Seeds With The Best Parameter Configuration

Observation: Table 2 demonstrates the consistency of the LSH implementation across different random seeds using the optimal configuration ($n = 100$, $b = 20$, $r = 5$, $max\_bucket\_size = 2000$). Candidate pairs ranged from approximately 15.7 million to 36.4 million, while similar pairs remained stable between 367 and 697, averaging 544 pairs. Computation time averaged 294 seconds, indicating robustness in identifying similar pairs regardless of seed variations.

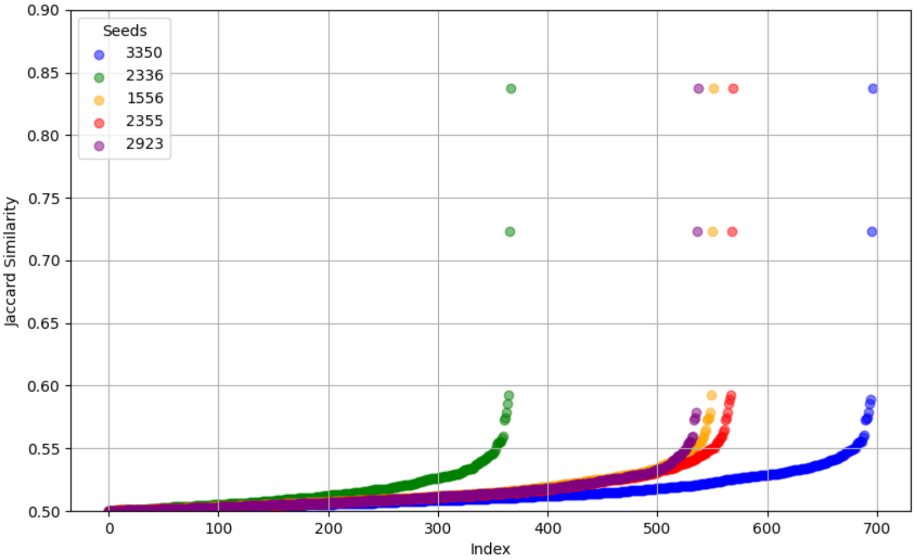

Figure 2: Distribution of found pairs with JS > 0.5 across different seeds

Observation: Figure 2 illustrates the distribution of similar pairs (Jaccard Similarity > 0.5) across different seeds. Despite variability in candidate pair counts, the number of similar pairs remains consistent, underscoring the effectiveness of the chosen LSH parameters in reliably identifying similar user pairs and showing that using hashlib.md5 was useful.

5 Conclusion

Throughout this project, we successfully implemented an optimized Locality-Sensitive Hashing (LSH) technique using MinHash signatures to identify similar Netflix users based on their movie ratings. By utilizing a sparse Compressed Sparse Row (CSR) matrix and efficiently structured NumPy arrays, we minimized memory usage and facilitated fast data access. Transitioning from a Python set to a list for candidate pair collection, followed by deduplication using NumPy's `unique` function, significantly reduced memory overhead and enhanced performance. Batch processing in the verification step ensured controlled memory usage and improved computation speed, while leveraging NumPy's vectorized operations eliminated slow Python loops, drastically reducing processing time. Effective memory management practices, such as deleting intermediate variables and performing in-place operations, maintained low memory consumption throughout the process.

A two-step verification process, introducing an estimated similarity threshold before exact Jaccard similarity computation, effectively filtered out dissimilar pairs, thereby reducing unnecessary computations. Parameter selection was crucial; choosing $b = 20$, $r = 5$, and $n = 100$ provided an optimal balance between identifying a substantial number of similar pairs (544), managing the number of candidate pairs (26,782,158 on average), and maintaining reasonable computation time (294 seconds).

The implementation was memory-efficient, with a maximum RAM usage of 6GB (Maximum), ensuring that the process remained scalable and manageable even with large datasets. This demonstrates the importance of aligning data structures with algorithmic access patterns to optimize both speed and resource utilization.

In summary, the project achieved its objective of efficiently identifying similar Netflix users and also provided us with valuable insights into the performance implications of different data structures in large-scale similarity computations.

6 References

References

[1] Jure Leskovec, Anand Rajaraman, Jeff Ullman, *Mining of Massive Datasets*, Cambridge University Press, 2014. Available at: `http://www.mmds.org`.