Amazon Book Review Similarity Detection using MinHashing and Locality Sensitive Hashing

Saji Hamati
Saji.hamati@studenti.unimi.it
55364A

June 26, 2025

Abstract

The implementation of a system for identifying similar book review pairs from the Amazon Books Review dataset is described in this report. Using approximate similarity measures to process large textual datasets efficiently is the main component of the solution. I used MinHashing to produce compact signatures that probabilistically preserve Jaccard similarity after shingling review texts into sets of k-shingles. Locality Sensitive Hashing (LSH) is used to rapidly identify candidate pairs, significantly lowering the number of direct similarity comparisons required, in order to increase efficiency and scale up the detection process. The report goes into detail about the algorithms used, the data acquisition, and preprocessing. It also fully discusses how important parameters, especially the k-shingle size and similarity threshold, affect the detection performance.

1 Introduction

Because of the sheer volume of data, the growth of online reviews offers both a wealth of information and a challenge for analysis. For many applications, including detecting duplicate content, summarising shared sentiments, or even spotting fraudulent activity, the ability to detect similar reviews can be essential. Implementing a system to identify similar book review pairs from a subset of the Amazon Books Review dataset is the main goal of this project. Conventional brute-force comparison techniques are computationally prohibitive due to the potentially enormous size of review datasets. In order to overcome this, the method used makes use of advanced data stream mining techniques like shingling, MinHashing, and Locality Sensitive Hashing (LSH), which allow for effective approximate similarity detection.

2 Dataset Description and Organization

The dataset considered for this project is the Amazon Books Review dataset, publicly available on Kaggle.

- **Dataset Link:** `https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews`
- **Considered Part:** The primary focus of this project is the `Books_rating.csv` file within the dataset, specifically the `'review/text'` column, which contains the textual content of the book reviews.

2.1 Data Acquisition

The dataset is downloaded during code execution using the Kaggle API. This ensures that the most recent version of the dataset is acquired (as of the access date). Authentication for the Kaggle API is handled via environment variables (`KAGGLE_USERNAME` and `KAGGLE_KEY`), with placeholder values in the final code to prevent the sharing of sensitive information, as recommended by Kaggle. The `download_and_load_data` function make this process. If the `CSV_FILE_NAME` (`Books_rating.csv`) is not found locally, the function initiates the download and unzipping of the dataset.

2.2 Data Organization

Once downloaded, the `Books_rating.csv` file is loaded into a Pandas DataFrame. Rows with missing values in the `'review/text'` column are dropped to ensure data integrity and avoid errors in subsequent text processing steps. For testing purposes and to manage computational resources, the code includes a subsampling mechanism. If `SUBSAMPLE_DATA` is set to `True`, a random subset of `SUBSAMPLE_SIZE` (e.g., 30,000) reviews is selected from the filtered DataFrame. Otherwise, the entire filtered dataset is utilized. This subsampling is handled within the `download_and_load_data` function, ensuring that the working dataset is appropriately sized for the task.

3 Applied Pre-processing Techniques

Text preprocessing is a critical first step to prepare raw review text for similarity analysis. The `preprocess_text` function is responsible for transforming the raw review strings into a format suitable for shingling.

- **Lowercasing:** All text is converted to lowercase. This standardizes the text, treating words like "Book" and "book" as identical, which is crucial for accurate similarity comparison.

- **Non-alphanumeric Character Removal:** Regular expressions are used to remove any characters that are not lowercase letters, numbers, or spaces (`r'[^a-z0-9\s]'`). This cleans the text by eliminating punctuation, special symbols, and other irrelevant characters that do not contribute to the semantic content for similarity detection.

- **Tokenization and K-Shingling:** The cleaned text is then split into individual words. These words are subsequently combined to form "k-shingles." A k-shingle is a contiguous sequence of k words. For instance, if $k = 1$ (as set by `k_shingle_size=1` in the provided code), each shingle is simply an individual word. If $k = 2$, shingles would be pairs of adjacent words (bigrams). The function returns a set of these unique k-shingles, which serves as the "set representation" of the review. This set representation is fundamental for applying MinHashing.

4 Considered Algorithms and Their Implementations

The core of this project's similarity detection pipeline relies on three main algorithms: Shingling, MinHashing, and Locality Sensitive Hashing (LSH).

4.1 Shingling

As introduced in the preprocessing section, shingling is the process of converting a document (in our case, a review text) into a set of fixed-size contiguous sequences of tokens (words). The size of these sequences is denoted by k, and the resulting sequences are called k-shingles. The `preprocess_text` function is used for this purpose. Each review is represented as a set of its unique shingles. This transformation is crucial because it allows us to approximate the similarity between documents using set-theoretic operations, specifically the Jaccard similarity. The Jaccard similarity between two sets A and B is defined as $|A \cap B|/|A \cup B|$.

4.2 MinHashing

MinHashing is a technique used to estimate the Jaccard similarity between two sets efficiently. Instead of computing the Jaccard similarity directly on potentially very large sets of shingles, MinHashing generates a small, fixed-size "signature" for each set. The Jaccard similarity

between two sets can then be approximated by the fraction of components that are identical in their respective MinHash signatures. The `MinHasher` class implements this algorithm.

4.2.1 Class: `MinHasher`

- `__init__(self, num_permutations, prime_modulo, seed=42)`:
  - Initializes the MinHasher with `num_permutations` (number of hash functions) and a large `prime_modulo`.
  - It generates `num_permutations` pairs of random coefficients (`hash_coeffs_a`, `hash_coeffs_b`) for universal hashing. These coefficients are used to simulate random permutations of the shingles.
- `_hash_shingle(self, shingle_str)`:
  - A private helper method that takes a shingle string and returns its hash value, ensuring it fits within the positive integer range modulo the `prime_modulo`.
- `get_signature(self, shingle_set)`:
  - Takes a set of shingles as input.
  - For each of the `num_permutations` hash functions, it computes the hash value for every shingle in the input set.
  - The signature for a document is a vector of `num_permutations` elements, where each element is the minimum hash value observed for that particular permutation across all shingles in the set.
  - If the `shingle_set` is empty, it returns a signature of infinity for all permutations.
- `calculate_minhash_jaccard_similarity(sig1, sig2)` (Static Method):
  - This method approximates the Jaccard similarity between two documents by comparing their MinHash signatures.
  - It counts the number of positions where the two signatures have identical values.
  - The similarity is calculated as the ratio of matching hash values to the total number of permutations (`len(sig1_np)`).

4.3 Locality Sensitive Hashing (LSH)

MinHashing significantly reduces the size of data needed for similarity comparisons, but comparing every pair of MinHash signatures ($O(N^2)$ for N documents) is still too slow for very large datasets. LSH addresses this by providing a mechanism to quickly identify "candidate pairs" that are highly likely to be similar. It works by partitioning each MinHash signature into "bands" and hashing these bands into buckets. If two documents are similar, they are likely to have at least one band that hashes to the same bucket, thus becoming candidate pairs.

4.3.1 Class: `LSHIndex`

- `__init__(self, num_bands, rows_per_band)`:
  - Initializes the LSH index with the number of bands (`num_bands`) and the number of rows (permutations) per band (`rows_per_band`).
  - It creates a list of hash tables, one for each band.
- `index(self, signatures_with_ids)`:
  - This method builds the LSH index. It iterates through each document's MinHash signature along with its ID.
  - For each signature, it divides it into `num_bands` non-overlapping segments (bands).
  - Each band is then hashed to produce a "bucket key." The document ID is stored in the corresponding hash table bucket for that band.
  - Documents that fall into the same bucket in at least one band are considered candidate pairs.
- `query_candidates(self, minhash_signatures_dict)`:
  - This method retrieves candidate pairs based on the built LSH index.
  - It iterates through the MinHash signatures. For each signature, it re-calculates the band hashes.
  - If a band's hash matches a key in any of the hash tables, all other document IDs stored in that same bucket for that band are retrieved as candidate similar reviews.
  - To avoid redundant comparisons and ensure each pair is unique, pairs are stored as sorted tuples (e.g., `(doc_id1, doc_id2)` where `doc_id1 < doc_id2`).

5 System Architecture and Workflow

The overall workflow of the similar review detection system is orchestrated within the `if __name__ == "__main__":` block. The steps are sequential, building upon the output of the previous stage.

1. **Download and Load Data:**

   - The process begins by calling `download_and_load_data` to acquire and load the specified subset of Amazon reviews into a Pandas DataFrame. This step handles Kaggle authentication and optional subsampling.

2. **Preprocessing and MinHash Signature Generation:**

   - For each review in the loaded DataFrame, the text is first processed using the `preprocess_text` function to generate its set of k-shingles.
   - These shingle sets are then passed to the `MinHasher.get_signature` method, which computes and stores the MinHash signature for each review in the `minhash_signatures` dictionary, mapping document IDs to their signatures.

3. **LSH Indexing:**

   - An instance of the `LSHIndex` class is created with the specified `NUM_BANDS` and `ROWS_PER_BAND`.
   - The `LSHIndex.index` method is then called, taking all generated MinHash signatures (along with their document IDs) to populate the LSH hash tables. This step organizes the signatures into buckets based on their band hashes.

4. **Candidate Pair Generation (LSH Query):**

   - The `LSHIndex.query_candidates` method is invoked. This function traverses the indexed MinHash signatures and identifies pairs of documents that have at least one common bucket across any of the bands. These pairs are collected into the `candidate_pairs` set. This significantly reduces the number of pairs that need to be fully compared.

5. **Verification with MinHash Jaccard Similarity:**

   - For each pair in the `candidate_pairs` set, their actual MinHash Jaccard similarity is computed using `MinHasher.calculate_minhash_jaccard_similarity`.

- Only pairs whose similarity score meets or exceeds the predefined `SIMILARITY_THRESHOLD` are added to the `final_similar_pairs` list.

6. **Display Results:**
   - The `final_similar_pairs` are sorted in descending order of their similarity score.
   - The top `TOP_N_SIMILAR_PAIRS` are then displayed, including their similarity score and snippets of the original review texts for context.

6 Scalability Analysis

The combination of MinHashing and LSH is designed to address the scalability challenges inherent in similarity detection for large datasets. Without these techniques, finding similar pairs would require computing the exact Jaccard similarity for every possible pair of documents, leading to a computational complexity of $O(N^2)$, where N is the number of reviews. For a dataset of millions of reviews, this is infeasible.

- **MinHashing's Role:** MinHashing converts high-dimensional shingle sets into fixed-size, compact signatures. Comparing two MinHash signatures is significantly faster than comparing two large shingle sets directly. This reduces the constant factor in similarity calculations.
- **LSH's Role in Pruning:** LSH is the primary mechanism for scaling up the process. Instead of N^2 comparisons, LSH reduces the number of required exact (or approximate, in our case) similarity computations to a much smaller set of candidate pairs. The LSH algorithm effectively prunes the vast majority of dissimilar pairs without explicitly comparing them. The percentage reduction in comparisons reported by the code quantifies this efficiency gain. This reduction is substantial, especially for sparse similarity graphs where most pairs are dissimilar.
- **Impact of Parameters:**
  - `SUBSAMPLE_SIZE`: Directly controls the size of the dataset processed. Increasing this size will naturally increase computation time, but the LSH framework ensures it scales sub-quadratically rather than quadratically.
  - `NUM_PERMUTATIONS`: A higher number of permutations leads to more accurate MinHash similarity estimates but increases the size of signatures and the computational cost of MinHashing.

– `NUM_BANDS` and `ROWS_PER_BAND`: These parameters dictate the "sensitivity" of LSH. A larger `NUM_BANDS` (and thus fewer `ROWS_PER_BAND` for a fixed `NUM_PERMUTATIONS`) makes LSH more likely to find similar pairs (higher recall) but may also increase the number of false positives (more candidate pairs). Conversely, fewer `NUM_BANDS` and more `ROWS_PER_BAND` reduce false positives (higher precision) but might miss some truly similar pairs (lower recall). The chosen values define the similarity range that LSH is most effective at detecting.

7 Experiments, Results, and Discussion

The experiments were conducted using the provided Python script, executed in a Google Colab environment. The parameters used for this specific run are derived from the constants defined in the script:

- `SUBSAMPLE_SIZE`: 30,000 reviews
- `NUM_PERMUTATIONS`: 128
- `MINHASH_PRIME_MODULO`: $2^{31} - 1$
- `NUM_BANDS`: 32
- `ROWS_PER_BAND`: 4 (calculated as 128/32)
- `SIMILARITY_THRESHOLD`: 0.6
- `TOP_N_SIMILAR_PAIRS`: 10
- `k_shingle_size`: 1

7.1 Experimental Results

1. **Data Loading:** The system successfully downloaded and loaded 3 million reviews 30K was used With the Subsample.
2. **MinHash Signature Generation Time:** Shingling and MinHashing took approximately 52.65 seconds.
3. **LSH Indexing Time:** Building the LSH index took approximately 3.71 seconds.
4. **LSH Candidate Query Time:** Finding candidate pairs using LSH took approximately 1.38 seconds.

5. **Candidate Pairs Found:** 497 candidate pairs were identified by LSH.

6. **Verification Time:** Verifying candidate pairs with MinHash Jaccard similarity took approximately 2 seconds, involving 497 MinHash comparisons.

7. **Top Similar Pairs:** The top 10 most similar pairs were displayed with their similarity scores and review snippets.

7.2 Discussion on Experimental Results

7.2.1 Impact of `SIMILARITY_THRESHOLD`

The `SIMILARITY_THRESHOLD` parameter plays a direct and critical role in determining which pairs are ultimately classified as "similar."

- **High Threshold (e.g., 0.8-0.9):** Setting a higher threshold will result in fewer but more precisely similar pairs. This leads to higher precision, meaning that a larger proportion of the reported similar pairs are indeed very close in content. However, it also increases the risk of missing genuinely similar pairs that fall just below the stringent threshold (lower recall). For instance, if reviews have minor variations or additional introductory/concluding remarks, they might fall below a very high threshold even if their core content is nearly identical.

- **Low Threshold (e.g., 0.4-0.5):** Conversely, a lower threshold will yield a larger number of similar pairs, potentially including reviews that share only a moderate amount of common content. This leads to higher recall (fewer truly similar pairs are missed) but may decrease precision, as more "false positives" (pairs that are not truly semantically identical but share some common words/phrases) might be included in the final list.

The choice of threshold depends on the specific application's requirements. For tasks requiring high confidence in similarity (e.g., deduplication), a higher threshold is preferable. For tasks aiming to discover all potential relationships (e.g., clustering related content), a lower threshold might be more appropriate, accepting some noise. The code's current threshold of 0.6 strikes a balance, aiming to capture reasonably similar reviews while still filtering out less relevant matches.

7.2.2 Impact of `k_shingle_size`

The `k_shingle_size` parameter, which was set to 1 in the provided code, defines the length of the contiguous word sequences (shingles) used to represent each document. Its impact is substantial:

- `k_shingle_size = 1` **(Unigrams):** When $k = 1$, shingles are individual words. This approach is simple and generates large sets of shingles for each document, making MinHashing effective in capturing word-level overlap. However, it can be sensitive to word order and different phrasings. Reviews might convey similar meanings using different word choices, which a unigram approach might miss. For example, "great book" and "excellent novel" might be considered dissimilar even if their intent is the same. It also means that common words (like "the", "a", "is") will contribute significantly to similarity if not removed, potentially leading to inflated similarity scores.

- **Larger `k_shingle_size` (e.g., 2 or 3 - Bigrams, Trigrams):** Using larger k values for shingles captures phrases rather than individual words.

  - **Increased Specificity:** Matching phrases (e.g., "fast-paced thriller" vs. "fast-paced mystery") provides a stronger indication of semantic similarity than matching individual words, leading to potentially more meaningful similarity results.
  - **Sparser Shingle Sets:** As k increases, the number of unique k-shingles in a document tends to decrease relative to the total possible shingles, and the overlap between documents (true Jaccard similarity) can become sparser unless reviews are almost identical. This might make it harder to find matches if reviews have minor grammatical variations or rephrasing.
  - **Computational Cost:** Generating and storing shingles for larger k can increase memory usage and processing time during the shingling phase, especially for very long documents.

The current setting of `k_shingle_size=1` suggests that the project is focusing on word-level overlap, which is a good starting point for general review similarity. For more nuanced semantic similarity, exploring higher k values or incorporating techniques like TF-IDF weighting within the shingling or hashing process could be beneficial. The choice of k directly affects the underlying Jaccard similarity between documents, which MinHash then approximates.

7.2.3 General Comments

The empirical results demonstrate the effectiveness of the MinHash and LSH pipeline in practice. The significant reduction in the number of candidate pairs ([99.66]%) compared to the total possible pairs clearly validates LSH's ability to prune the search space efficiently. This efficiency gain is crucial for handling large datasets like the Amazon reviews. The overall execution time highlights the practical feasibility of this approach for large-scale approximate similarity detection.

8 Conclusion

This project successfully implemented a robust and scalable system for detecting similar book reviews from the Amazon Books Review dataset. By leveraging a combination of shingling, MinHashing, and Locality Sensitive Hashing, the system efficiently identifies candidate pairs and verifies their similarity, significantly reducing the computational burden associated with large-scale text comparison. The modular design, with dedicated classes for MinHashing and LSH, promotes clarity and reusability. The discussion on the impact of parameters like similarity threshold and k-shingle size provides insights into tuning the system for specific performance requirements, highlighting the trade-offs between precision, recall, and computational cost.

9 Declare

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. No generative AI tool has been used to write the code or the report content.