Detecting Similar Amazon Book Reviews

Malika Makhmudova / 34003A

June 8, 2025

1 Introduction

This report details the implementation of a system to detect pairs of similar book reviews from the Amazon Books Review dataset. The core challenge lies in efficiently comparing a potentially vast number of reviews for similarity, which demands scalable techniques. Project address this by employing **MinHash** for approximate Jaccard similarity estimation and **Locality Sensitive Hashing (LSH)** for efficient similarity search.

2 Dataset Overview and Data Organization

The dataset used for this project is the **Amazon Books Review dataset**, publicly available on Kaggle (DATASET_LINK). It contains various fields related to book ratings and reviews. For this project, focus was on the `'review/text'` column, as goal is to identify similarity between the textual content of reviews.

Due to the potentially large size of the full dataset, a **subsampling strategy** was used during development and testing to ensure code functionality and to analyze scalability. A global variable, `SUBSAMPLE_SIZE`, controls the number of reviews processed. For the experiments presented in this report, subsets of the data up to `SUBSAMPLE_SIZE = 50,000` were considered.

The dataset is loaded into a Pandas DataFrame. Reviews with null values in the `'review/text'` column are removed, and the DataFrame is re-indexed for consistent access.

The dataset can be downloaded programmatically using the Kaggle API.

3 Applied Pre-processing Techniques

Effective text preprocessing is crucial for meaningful similarity detection. Pipeline for processing each review text involves several steps:

1. **Lowercasing:** All text is converted to lowercase to ensure that words like "Book" and "book" are treated as identical.
2. **Tokenization:** Regular expressions (`\b\w+\b`) are used to extract all word tokens from the text. This captures sequences of alphanumeric characters as individual words.
3. **Stopword Removal:** Common English **stopwords** (e.g., "the", "a", "is", "and"), which often carry little semantic meaning and can inflate similarity scores between dissimilar documents, are removed using NLTK's list of stopwords. This helps to focus the similarity measure on the more informative content words.
4. **Stemming:** The **Porter Stemmer** from NLTK is applied to reduce words to their root form (e.g., "reading", "reads", "reader" all become "read"). This is vital for Jaccard similarity, as it allows to identify conceptual similarity even if the exact word forms differ, thereby capturing more relevant "similar" review pairs.

Each processed review is then represented as a **set of unique stemmed, non-stopword tokens**. This set representation is ideal for calculating Jaccard similarity. The choice to include stopword removal and stemming, while deviating slightly from pure literal word overlap, was made to better capture the semantic content of reviews, leading to more meaningful similarity results by filtering noise and normalizing word forms.

4 Algorithms and Implementation

The core of similarity detection system relies on **MinHash** for approximate Jaccard similarity and **Locality Sensitive Hashing (LSH)** for efficient search.

4.1 Jaccard Similarity

The **Jaccard similarity** between two sets A and B is defined as:

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

This metric ranges from 0 (no common elements) to 1 (identical sets). For textual content, it measures the overlap of unique words between two documents.

4.2 MinHash for Approximate Jaccard Similarity

Directly calculating Jaccard similarity for all pairs of documents in a large dataset is computationally prohibitive, scaling quadratically with the number of documents ($O(N^2)$). To overcome this, **MinHash** is used to create compact "signatures" for each review's token set. A MinHash signature is a fixed-size vector of hash values. The Jaccard similarity of two sets can be approximated by the fraction of components in their MinHash signatures that are identical. The `datasketch` library's `MinHash` implementation is used, with `num_perm=128`. This parameter controls the number of permutations used in the MinHash algorithm, which directly impacts the accuracy of the Jaccard similarity estimation. A higher `num_perm` leads to a more accurate approximation but increases computation time and memory usage. 128 is a common and reasonable choice for balancing accuracy and efficiency.

4.3 Locality Sensitive Hashing (LSH)

Even with MinHash, comparing every signature pair is still $O(N^2)$. **Locality Sensitive Hashing (LSH)** provides a highly scalable solution for finding approximate nearest neighbors. LSH hashes similar items into the same buckets with high probability. This allows us to query for similar items by looking only within a few buckets, drastically reducing the number of candidate pairs that need to be compared. Implementation uses `MinHashLSH` from `datasketch`.

- `MinHashLSH` is initialized with a `threshold` of `0.5`, meaning that only pairs with an estimated Jaccard similarity above this value will be considered as candidates by the LSH structure.
- The `num_perm` parameter is set to `128`, consistent with the MinHash signature generation.

The workflow for finding similar pairs is as follows:

1. **MinHash Generation:** For each preprocessed review, a MinHash signature is generated.
2. **LSH Indexing:** Each MinHash signature is inserted into the LSH structure.
3. **Similarity Query:** For each review's MinHash, we query the LSH index to retrieve a list of *candidate* documents that are likely to be similar based on the LSH bucketing.
4. **Exact Jaccard Verification:** For each candidate pair returned by LSH, we perform a precise Jaccard similarity calculation using their MinHash signatures. This step filters out false positives that LSH might return due to its probabilistic nature.
5. **Filtering and Storage:** Only pairs with a Jaccard similarity greater than the `threshold` (0.5) are stored as similar. A `visited` set ensures that each unique pair is recorded only once and avoids redundant checks (e.g., `(doc_A, doc_B)` and `(doc_B, doc_A)`).

5 Scalability of the Proposed Solution

The key advantage of the MinHash and LSH approach is its **scalability**.

- **MinHash generation:** The time complexity for generating MinHash signatures is roughly linear with the total number of tokens across all documents, $O(N \times L_{avg} \times \text{num\_perm})$, where N is the number of documents and L_{avg} is the average document length.
- **LSH indexing:** Inserting documents into the LSH index also scales efficiently, roughly linearly with the number of documents.
- **LSH querying:** The most significant gain comes here. Instead of $O(N^2)$ comparisons, LSH reduces the comparison complexity to approximately $O(N \times K)$, where K is the average number of candidate pairs returned by LSH for each query. K is significantly smaller than N, especially for sparse similarity graphs.

To demonstrate scalability, experiments were conducted by varying the `SUBSAMPLE_SIZE` parameter and measuring the total execution time for finding similar pairs.

Table 1: Experimental Results on Scalability

Subsample Size	Execution Time (seconds)	Number of Similar Pairs Found
1,000	0.02	15
5,000	0.09	85
10,000	0.18	239
50,000	1.05	3207

6 Experiments and Results

Experiments involved running the code provided on Google Colab, leveraging its computational resources. The `SUBSAMPLE_SIZE` was varied to observe the performance behavior on datasets of increasing scale. The `threshold` for Jaccard similarity was set at 0.5, meaning considered reviews were with at least 50% unique word overlap (after preprocessing) as similar.

Experiments were performed on the Jupyter Notebook and Google Colab with a high RAM runtime, running multiple times to average results.

The output from the code, such as the total number of similar pairs found and the execution time, is shown in the results section. For a `SUBSAMPLE_SIZE` of 50,000 reviews, the system efficiently identified `3207` similar pairs within `1.05` seconds. The sample output provides concrete examples of the detected similar review pairs, displaying their similarity score and truncated text, showcasing the efficacy of the approach.

7 Comments and Discussion

The implementation successfully demonstrates a scalable approach to finding similar book reviews. The choice of **MinHash and LSH** was critical, as it effectively approximates Jaccard similarity while drastically reducing the computational cost compared to brute-force $O(N^2)$ comparisons.

The inclusion of **stopword removal and stemming** in the preprocessing pipeline significantly enhances the quality of "similarity" detected. While Jaccard similarity traditionally focuses on literal word overlap, these techniques allow the system to capture **semantic similarities** that might otherwise be missed due to common words or morphological variations. For example, two reviews expressing the same sentiment but using "reading" in one and "read" in another would be correctly identified as more similar. This makes the detected similar pairs more conceptually meaningful for tasks like plagiarism detection or identifying duplicate content.

The experimental results (as presented in Section 5) clearly illustrate the near-linear scalability of the LSH-based approach. As the number of reviews increases, the processing time grows proportionally rather than exponentially, confirming the suitability of this method for larger datasets.

Potential areas for further exploration include:

- **Parameter Tuning:** Experimenting with different `num_perm` values (e.g., 64, 256) could fine-tune the accuracy-performance trade-off. Varying the LSH `threshold` could also change the strictness of similarity.
- **Alternative Embeddings:** For richer semantic understanding, one could explore word embeddings (e.g., Word2Vec, BERT) combined with similarity measures like cosine similarity, although this would significantly change the complexity and resource requirements.
- **Scalability to Petabytes:** While this solution scales to large datasets, truly "petabyte-scale" data might require distributed computing frameworks (e.g., Spark) to handle data loading and processing in parallel. However, the core algorithms (MinHash, LSH) are fundamentally suited for such distributed environments.

In conclusion, the implemented solution provides a robust, efficient, and scalable method for identifying similar textual content in large datasets, proving its efficacy for the Amazon Books Review dataset.