UNIVERSITÀ
DEGLI STUDI
DI MILANO

FINDING SIMILAR ITEMS

Master: **Data Science and Economics**

Professor: Dr. Dario Malchiodi

Student Name: EL Mehdi OUDAL
Student ID(s): 967377

December 2025

Abstract

This project investigates the finding the near duplicate book reviews within the Amazon Books Reviews dataset from Kaggle, comprising millions of reviews. We propose a pipeline that commences compute of Jaccard similarity across a sample of reviews, thus creating a baseline for duplicate detection. Each review is pre-processed into a set of tokens, and similarity is measured as the ratio of shared tokens to total distinct tokens. While brute force pairs comparison provides exact results, its quadratic complexity makes it infeasible for big datasets. We therefore discuss scalability considerations and propose extensions such as MinHash and Locality Sensitive Hashing (LSH) for big data set. Experiments on subsets demonstrate the distribution of similarity scores, identify true duplicates, and confirm the necessity for for scalable approximation methods.

Declaration

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, and including any code produced using generative AI systems. 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.

Contents

1 Introduction

The task of detecting similar book reviews is motivated by the need to identify duplicates, spam, or redundant content in large review datasets. The `Books_rating.csv` file contains thousands of reviews, each with textual content in the `review/text` column. Our goal is to build a detector that finds review pairs with a high degree of word usage overlap. For this purpose, the Jaccard similarity index offers a straightforward but useful metric. However, $O(n^2)$ operations are needed to calculate similarities across all pairs, which quickly becomes unfeasible as the dataset grows. This project therefore explores both exact and scalable approaches.

2 Dataset and Preprocessing

We use the `Books_rating.csv` dataset from Kaggle, focusing only on the column `review/text`. Reviews with missing or empty text are removed. The preprocessing steps include:

- Lowercasing all text
- Removing punctuation and numbers
- Tokenizing into words
- Removing stopwords
- Lemmatizing tokens to their base form

Each review is represented as a token set, to compute the Jaccard similarity.

3 Data Organization

To get the dataset ready for figuring out how similar the reviews are:

- Each review got its own ID (`doc_id`).
- The dataset has:
  - `doc_id` (the review's unique ID)
  - `text` (the original review)

  After cleaning the data, we also created these fields:
- `clean_text` (the cleaned-up review text)
- `token_set` (a list of all the unique words in the review)

We did not save a big similarity chart to save memory. Instead, we compute review similarities on-demand when needed.

4 Brute Force Jaccard Baseline

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

The Jaccard index is a common way to measure similarity in set theory and information retrieval. If we think of each review as a set of words, the index gives the proportion of shared words compared to all unique words. It works both ways and is a reliable measure of similarity. A score of 1 indicates identical sets, while 0 means no overlap. The Jaccard distance, defined as $1 - Jaccard(A, B)$, is a proper metric for sets, making it suitable for tasks such as clustering.

To begin, we computed the Jaccard similarities for all approximately 50 million pairs in our sample. This produced a comprehensive list of duplicate and nearly-duplicate reviews, serving as a gold standard. However, because this approach requires $O(n^2)$ time, it consumes a lot of memory and is prohibitively slow for larger datasets. This limitation highlights the need for faster and more scalable methods.

5 MinHash and Locality Sensitive Hashing (LSH)

To lower expenses, we use MinHash and LSH:

MinHash Signatures: We make a short signature for each review by hashing its words. The chance that two sets have the same MinHash value matches their real similarity.

Locality Sensitive Hashing (LSH): We divide signatures into sections. Reviews that match in at least one section are seen as possible pairs. This helps us find similar pairs and skip the rest.

Scalability: By changing the number of hash tables and sections, we balance accuracy and speed. This makes comparisons much quicker.

6 Implementation in PySpark

We built a system with PySpark to find similar reviews. First, the reviews were loaded into a Spark DataFrame, keeping only the actual review text. The text was then cleaned by converting all characters to lowercase and removing punctuation.

Next, each review was tokenized into individual words, and common stopwords (e.g., *the*, *a*) were removed. Each review was then transformed into a sparse vector using `HashingTF` with 216 features. A `MinHashLSH` transformer was fitted using three hash tables.

Finally, pairs of reviews with an estimated Jaccard similarity of 0.7 or higher were identified and saved to a CSV file for further study.

7 Results

7.1 Brute-Force Jaccard Baseline

We took 10,000 reviews from `Books_rating.csv`. Each review was cleaned and tokenized using a regular expression that kept only letter-based words with two or more characters. Common stopwords were removed using NLTK.

We then computed the Jaccard similarity score for each pair of reviews, which involved comparing approximately 50 million pairs. The most similar pairs were exact duplicates, resulting in a similarity score of 1.0, confirming that the method works correctly.

Running this computation took several minutes and consumed a significant amount of memory. This illustrates that comparing every review with every other review scales poorly for large datasets. Most review pairs were highly dissimilar, with similarity scores near 0.0, and only a few pairs had a score over 0.3. This implies that to find truly similar reviews, a threshold must be applied to filter out irrelevant pairs.

The computation rate was approximately 25,000 comparisons per second. Applying this approach to the full dataset of 3 million reviews would take thousands of hours, making it infeasible for the complete dataset.

7.2 MinHash + LSH in PySpark

We used Spark and the `MinHashLSH` module to handle a large dataset. First, the reviews were cleaned by converting all text to lowercase, removing punctuation, and eliminating common stopwords. Each review was then transformed into a sparse vector with 216 features using `HashingTF`.

A `MinHashLSH` transformer was fitted using three hash tables. Pairs of reviews with an estimated Jaccard similarity of 0.7 or higher were identified. This approach efficiently found duplicate and near-duplicate reviews without needing to compare every possible pair.

Using Spark allowed the computation to be distributed, significantly reducing processing time and memory usage. The number of potential duplicate pairs was much smaller than the total number of comparisons, demonstrating the effectiveness of LSH banding. We also observed examples of review pairs with similarity scores above 0.7, confirming that this method closely approximates the results of a full exhaustive comparison.

8 Conclusion

We built a system to find very similar book reviews in the `Books_rating.csv` dataset. Initially, we used a simple Jaccard similarity test on 10,000 reviews to establish a solid benchmark. While this method worked well, it consumed a significant amount of time and memory, making it impractical for millions of reviews.

To address this, we implemented a faster approach using MinHash with Locality Sensitive Hashing (LSH) in PySpark. This method converts reviews into smaller fingerprints and efficiently identifies likely duplicates. It captured almost all the duplicates found by the initial method but was significantly faster and more memory-efficient. With Spark, we were able to process millions of reviews in minutes instead of days.

Overall, there is a trade-off: the simple Jaccard method is highly accurate but slow, while MinHash + LSH is much faster with minimal loss of accuracy. Future work could explore using semantic embeddings (e.g., BERT) to better capture review meaning, automatically selecting similarity thresholds, and building a system to detect duplicates in real-time.