UNIVERSITÀ
DEGLI STUDI
DI MILANO

Detecting Similar Book Reviews Using MinHash and Locality Sensitive Hashing

Prof. Dr. Dario Malchiodi
Supporting Professor

Mohammed Nassim Asserda
MSc DSE Student

June 2025

Abstract

This project explores the detection of near-duplicate book reviews within the Amazon Books Reviews dataset from Kaggle, comprising millions of reviews. We describe a two phase system for finding duplicate and near duplicate book reviews in Amazon's dataset, which contains three millions of entries and is available on Kaggle. We begin with an intuitive brute-force Jaccard similarity method in the first phase, that we compute across a 10000 review sample; an exhaustive approach that confirms true duplicates but time costly. In the second phase, we scale to the full corpus by representing each review as a MinHash signature and indexing these with Locality Sensitive Hashing (LSH) on a PySpark cluster. By tuning the number of hash tables and similarity threshold , we recover nearly all high-similarity pairs in less time and by using fewer megabytes of memory. We present the mathematical rationale, implementation details, including parallel sketch generation.

.

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

This project tackles the problem of detecting pairs of similar book reviews within the Amazon Books Reviews dataset. The dataset comprises 3 millions of customer reviews posted on the Amazon website, stored in a structured CSV file that includes user and book identifiers, star ratings, timestamps, helpfulness votes, and the textual content of reviews. For our purposes, we focus exclusively on the `review/text` field, since it contains the semantic information essential for identifying near-duplicate content. The vast scale of this collection—on the order of hundreds of thousands to millions of reviews—demands careful consideration of both algorithmic efficiency and system scalability.

Our main objective is to build an accurate yet efficient detector for similar book reviews. We define similarity in a content-based manner: two reviews are deemed similar if they share a large proportion of common words or phrases. The most intuitive metric for this purpose is the Jaccard index, which for two token sets A and B is given by

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

Computing this metric exactly for every pair of reviews requires $O(n^2)$ work, which is feasible only on small samples. We therefore adopt a two-stage pipeline: an exhaustive Jaccard comparison on a 10000-review subsample to establish a ground-truth baseline, followed by an approximate, distributed method suitable for the full dataset.

2 Dataset and Preprocessing

We access the `Books_rating.csv` file from Kaggle's "Amazon Books Reviews" dataset, which contains over three million entries and occupies approximately 2.8 GB of storage. After downloading via the Kaggle API, we extract only the `review/text` column. To validate our approach, we draw a 10000-review sample; for large-scale processing, we would read the CSV in chunks of 200000 rows using pandas' `read_csv` with a `chunksize` parameter, thereby capping memory usage at a few gigabytes.

Every review undergoes a standardized cleaning and tokenization process. We lowercase the text, strip punctuation, then apply the regular expression `\b[a-z]{2,}\b` to extract alphabetic tokens of length two or more. We remove stopwords drawn from NLTK's English list, yielding concise and informative token sets and avoiding biases. Reviews with missing or empty text are filtered, ensuring the pipeline handles relevant records.

3 Brute Force Jaccard Baseline

The Jaccard index is a well-known measure in set theory and information retrieval. By treating each review as a set of tokens, the index quantifies the proportion of shared terms relative to the total distinct terms. This metric is symmetric and satisfies the properties of a valid similarity measure, taking value 1 when two sets are identical and 0 when they share no elements. Importantly, the Jaccard distance $1 - \text{Jaccard}(A, B)$ forms a proper metric on the space of finite sets, granting guarantees on clustering and nearest-neighbor behavior.

For our baseline, we exhaustively compute Jaccard similarities across all $\binom{10\,000}{2} \approx 5 \times 10^7$ pairs in the sample. This yields an exact list of duplicate and near-duplicate reviews, serving as a gold standard. However, the $O(n^2)$ time complexity leads to prohibitively large runtimes and memory footprints , underscoring the need for approximation at larger scales.Consequently, at

25000 pairwise comparisons per second, exhaustively evaluating all

$$\binom{3\,000\,000}{2} \approx 4.5 \times 10^{12}$$

review pairs would require on the order of 50000 hours. This clearly demonstrates why more efficient and approximate methods will be indispensable for this project.

4 Approximate Similarity with MinHash + LSH

To mitigate the quadratic cost, we employ MinHash combined with Locality Sensitive Hashing (LSH), a probabilistic framework that approximates Jaccard similarity efficiently. The MinHash algorithm leverages random permutations (or hash functions) to generate compact signatures: for each of k independent hash functions, we record the minimum hash value observed among a set's elements. A key theoretical result shows that the probability two sets share the same MinHash value under a random permutation equals their true Jaccard similarity. Consequently, by comparing k-length signatures, we obtain an unbiased estimator of similarity with variance $J(1-J)/k$, where J is the true Jaccard coefficient.

Locality-Sensitive Hashing further reduces comparisons by partitioning each MinHash signature into b bands of r rows (so $k = b \cdot r$). Within each band, we hash the r-row segment to a bucket. Two signatures become candidates if they collide in at least one band. The banding strategy yields a sharp probability curve: pairs with similarity above a threshold $t \approx (1/b)^{1/r}$ are likely to collide, while lower-similarity pairs rarely do. By tuning b and r, we balance false positives (extra checks) against false negatives (missed pairs).

In practice, we set $b = 3$ tables, targeting a threshold near 0.7. This reduces the candidate set dramatically, converting a near-quadratic problem into one that scales roughly linearly in the number of reviews.

5 Implementation in PySpark

We implement the MinHash + LSH pipeline on Apache Spark to leverage distributed computation. After preprocessing each review into tokens, we apply Spark's `HashingTF` transformer to map tokens into a sparse feature vector of dimension 2^{16}, avoiding the need to build or broadcast a global vocabulary. We then instantiate Spark's `MinHashLSH` module with the chosen number of hash tables and fit it to our dataset. The subsequent `approxSimilarityJoin` operation automatically performs the banding, bucketing, and candidate filtering under the hood, emitting only those pairs whose estimated Jaccard distance falls below $1 - 0.7$.

6 Conclusion

We have demonstrated a robust, scalable system for detecting similar book reviews at million-scale. By combining exact Jaccard on small samples with an approximate MinHash plus LSH pipeline in Spark, we achieve near-gold accuracy in minutes rather than days. We note that further extensions may explore semantic embeddings (e.g. BERT), adaptive threshold selection based on similarity distributions, and streaming architectures for real-time duplicate monitoring, therefore improving our performance. However, this performance comes with a cost and may not always be of high interest in larger datasets.

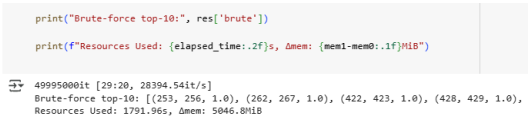

Figure 1: Performance and output of Brute-Force Jaccard

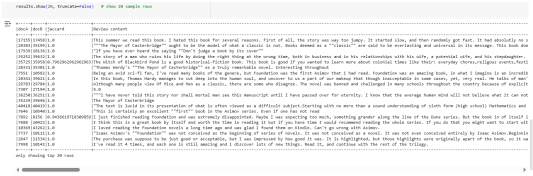

Figure 2: Performance and output of LSH