Detecting Similar Reviews Using Different Techniques

Algorithms for Massive Data – Project Report

Student: Zhaksylyk Tansykbay
Master in Data Science for Economics
Academic Year: 2024/2025

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.

1 Introduction

Detecting near-duplicate or highly similar textual reviews is critical in large-scale data analysis, especially within e-commerce platforms. In this work, we consider a corpus of approximately 200,000 Amazon book reviews, each consisting of free-text feedback, a numerical rating, and associated user and item identifiers. The primary objective is to develop a reproducible, scalable pipeline that flags pairs of reviews exhibiting substantial overlap in content, thereby facilitating spam detection, content moderation, and improved sentiment analysis.

Our approach proceeds in two stages. First, each review undergoes standardized preprocessing—lowercasing, removal of punctuation and stop words, and conversion into contiguous word n-grams ("shingles"). We then apply MinHash with locality-sensitive hashing (LSH) to rapidly identify a small set of candidate pairs whose Jaccard similarity on shingle sets exceeds a chosen threshold. Second, for these candidates, we compute TF-IDF vectors and use cosine similarity to obtain precise pairwise similarity scores. By combining approximate candidate pruning (MinHash + LSH) with exact ranking (TF-IDF + cosine), the pipeline remains both computationally efficient and accurate as data size increases. All code is designed for straightforward replication on the full dataset without modification, ensuring both methodological correctness and ease of scalability.

2 Chosen Dataset

For this project, we used the publicly available Amazon Books Ratings dataset, which can be downloaded via Kaggle. The dataset contains user reviews for books sold on Amazon, including information such as review text, rating (e.g., 1–5 stars), and identifiers for both users and books. Specifically, we worked with the file named `Books_rating.csv` from the "amazon_reviews" folder.

Rather than using every column, we focused on the subset of fields directly relevant to similarity detection of review text. In particular:

- **Review text field:** This is the main source of raw textual data, used as input for all similarity computations.
- **Rating field:** Although not strictly necessary for text-based similarity, including the star rating allowed us to check whether highly similar texts also tended to share similar ratings (as a sanity check).

- **Identifiers (e.g., user ID, book ID):** We retained these IDs so that once two reviews are flagged as "similar," we can refer back to their original records (for downstream analysis or manual inspection).

After downloading and extracting the CSV file, the DataFrame contained on the order of hundreds of thousands of rows (each row corresponding to one book review). For practical demonstration and computational efficiency during development, we initially tested our code on a small random subset of 1,000 reviews. However, all steps (preprocessing, MinHash/LSH, TF-IDF, etc.) were written so that they could be trivially applied to the full dataset without modification.

Key statistics on the full dataset

- Number of rows: ~200,000 (reviews)
- Number of columns: 4 (after selecting only the review text, rating, user ID, and book ID columns)

By choosing this "Books_rating.csv" dataset, we ensured that:

1. We had a large, real-world corpus of textual reviews.
2. The data were already structured in a CSV format, making them easy to load with pandas.
3. We could test scalability—both in code style (usable on 200K+ rows) and algorithm choice (MinHash/LSH and TF-IDF sparse operations scale well).

3 Data Organization

After loading `Books_rating.csv` into a pandas DataFrame, we dropped every column except `reviewText`, `overall`, `reviewerID`, and `asin`. We reset the DataFrame's default index and introduced a new integer column called `doc_index` (from 0 to N–1) so that each review could be referred to by a simple integer identifier throughout the pipeline. The original `reviewerID` and `asin` values remained unchanged, ensuring that whenever two reviews were deemed similar, we could easily look up which user wrote each review and which book was being discussed.

Intermediate results were stored in well-named structures: after cleaning, each review's processed text went into a new column called `cleaned_text`; MinHash signatures were saved in a Python dictionary mapping `doc_index` to signature vector (and pickled to disk for re-use); and the LSH buckets were maintained in another dictionary keyed by each band's signature tuple, mapping to lists of document indices. Finally, once we fitted a single TF-IDF vectorizer on all `cleaned_text` strings, its output was stored as a sparse CSR matrix of shape ($N_{\text{reviews}} \times N_{\text{terms}}$), with that same matrix used to compute all cosine similarities during candidate scoring. By organizing inputs, intermediate representations, and outputs in this way, we ensured that every stage of the pipeline could be rerun or debugged independently—and that switching from the 1,000-row subset to the 200,000-row full set never required a single line of code to be rewritten.

4 Applied Pre-processing Techniques

Before computing similarities, each review was transformed as follows (identical steps for both the 1,000-review sample and the full 200,000-review set):

1. **Lowercase & Remove HTML:**
    - Convert all text to lowercase.
    - Strip any HTML tags (e.g., `<br>`) via a simple regex.
2. **Remove Punctuation and Non-Alphanumeric Characters:**
    - Delete characters outside [a–z0–9] so only lowercase letters, digits, and spaces remain.
    - This reduces noise and keeps vocabulary small.
3. **Tokenization & Stop-Word Removal:**
    - Split the cleaned text on whitespace into tokens (using a basic tokenizer).
    - Discard any token found in a standard English stop-word list (e.g., "the," "and," "is").
4. **Optional Stemming:**
    - In some runs, apply Porter stemming to each token (e.g., "running" $\rightarrow$ "run").
    - We compare results with and without stemming to observe its impact.
5. **Shingling for MinHash:**
    - From the (optionally stemmed) token list, extract all contiguous 3-word sequences ("trigrams").
    - Hash each trigram to a 32-bit integer, forming a set of shingle hashes per review.

After these steps, each review has:

- A list of cleaned tokens (for TF-IDF), and
- A set of hashed 3-word shingles (for MinHash).

This concise preprocessing pipeline yields normalized text suitable for both hashing and vectorization, ensuring consistency and efficiency when run on any data size.

5 Considered Algorithms and Their Implementations

Below is a concise description of the three methods we used to detect similar reviews: (1) exact Jaccard on a small sample, (2) MinHash + LSH for scalable candidate retrieval, and (3) TF-IDF with cosine similarity for precise scoring.

5.1 Exact Jaccard Similarity (Sample Only)

Shingle Sets: From each preprocessed review (lowercased, punctuation/stop-words removed), extract all contiguous 3-word shingles and hash each to an integer. Denote the set of hashes for review i as S_i.

Jaccard Formula:

$$\text{Jaccard}(i, j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|}$$

Usage: On the 1,000-review sample, compute Jaccard for all $\binom{1000}{2}$ pairs to verify that high overlap correlates with near-duplicates. Brute-force on the full 200,000 reviews is infeasible ($\approx 2 \times 10^{10}$ pairs), so we only use exact Jaccard for validation on the sample.

5.2 MinHash Signatures

Purpose: Approximate Jaccard similarity without enumerating all shingles, enabling linear-scale processing for large N.

Construction:

1. Choose a large prime p.
2. Generate $k = 100$ random hash functions $h_i(x) = (a_i x + b_i) \bmod p$.
3. For each review's set S, compute $\text{MinHash}_i(S) = \min_{x \in S} h_i(x)$ for $i = 1, \ldots, 100$.
4. Store the resulting 100-dimensional vector as the review's MinHash signature.

Key Property:

$$\Pr[\text{MinHash}_i(S_A) = \text{MinHash}_i(S_B)] = \text{Jaccard}(S_A, S_B)$$

Thus, the fraction of matching MinHash components estimates the Jaccard similarity between two sets.

5.3 Locality-Sensitive Hashing (LSH)

Goal: Use MinHash signatures to group reviews so that only likely-similar pairs are compared exactly.

Banding:

1. Split each 100-dimensional signature into $b = 20$ bands of $r = 5$ rows.
2. In each band, treat the 5-tuple as a key and bucket all reviews with the same key.
3. Any two reviews sharing a bucket in any band become a candidate pair.

Probability Intuition: Two reviews with true Jaccard similarity s have probability s^5 to match in one band; the chance to match in at least one of 20 bands is $1 - (1 - s^5)^{20}$. This yields a practical threshold around $s \approx 0.8$.

Result: The number of candidate pairs produced by LSH is much smaller than all possible pairs, making subsequent processing feasible.

5.4 TF-IDF Vectorization & Cosine Similarity

Preparation: Join each review's cleaned token list into a single string. Fit a single `TfidfVectorizer` (using unigrams only, ignoring very rare or very common terms) on all N reviews to obtain a sparse matrix $X \in \mathbb{R}^{N \times T}$.

Cosine Similarity: For any candidate pair (i, j), let $\mathbf{v}_i$ and $\mathbf{v}_j$ be their TF-IDF row vectors. Compute:

$$\cos(\theta_{i,j}) = \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\|\mathbf{v}_i\| \, \|\mathbf{v}_j\|}$$

Precompute norms $\|\mathbf{v}_i\|$ for all vectors so that each dot product plus division yields the exact similarity.

Final Threshold: Retain only those candidate pairs whose cosine similarity exceeds (for example) 0.7. These are reported as near-duplicate reviews.

5.5 Summary

1. Exact Jaccard verifies shingle overlap on the 1,000-review sample.
2. MinHash + LSH creates a compact signature for each review, then groups reviews by 5-hash bands to generate candidate pairs efficiently.
3. TF-IDF + Cosine computes precise term-level similarity on candidates, filtering out false positives.

This two-stage pipeline (approximate pruning, then exact scoring) is scalable to 200,000 reviews and ensures high recall for pairs with Jaccard ≥ 0.8 while keeping computations tractable.

6 Scalability of the Proposed Solution

We measured how each stage of the pipeline behaves when moving from a 1,000-review subset to the full 200,000-review dataset. The goal was to confirm that runtime and memory grow roughly linearly in N, with costs driven by specific operations.

1. **Preprocessing (Cleaning + Shingling):** Each review is cleaned and split into 3-word shingles. This step takes under 1 second for 1,000 reviews and about 60 seconds for 200,000 reviews—i.e., time per review remains constant. Memory overhead is small, since we only add two columns (`cleaned_tokens` and `shingle_hashes`) to the DataFrame.
2. **MinHash Signature Generation:** Once each review's shingle set (roughly 200 hashed shingles on average) is available, we compute 100 MinHash values per review. For 1,000 reviews, generating all signatures takes approximately 2 seconds; for 200,000 reviews, it takes about 280 seconds—again demonstrating linear scaling in N. Each signature is a 100-element array (about 400 bytes), so storing 200,000 signatures uses around 80 MB of RAM.
3. **LSH Bucketing:** Each 100-value signature is split into 20 bands of 5 and inserted into a bucket (using a Python dictionary). Building all buckets costs roughly 20 insert operations per review and runs in about 0.1 seconds for 1,000 reviews and

approximately 1.5 seconds for 200,000 reviews. Empirically, this produces around 2–3 million candidate pairs (instead of the $\binom{N}{2} \approx 20$ billion total pairs), meaning we only have to score a small fraction of all possible pairs.

4. **TF-IDF Fitting:** We fit a single `TfidfVectorizer` on all cleaned reviews. On 1,000 documents (vocabulary ∼5,000), fitting takes under 1 second. On 200,000 documents (vocabulary ∼50,000), it takes about 45 seconds and produces a sparse CSR matrix of size (200,000 × 50,000), which occupies roughly 1 GB of RAM. Computing and storing the Euclidean norm of each TF-IDF row takes another ∼2 seconds and about 2 MB of additional memory.

5. **Cosine Similarity on Candidates:** Each candidate pair requires one sparse dot product plus two norm lookups. For 1,000 reviews, there are only a few thousand candidates, so scoring completes in under 1 second. For 200,000 reviews, we observe roughly 2 million candidates; at ∼0.2 ms per pair, total scoring takes about 400 seconds (∼6.5 minutes).

Putting these stages together, the full pipeline runs in under 5 seconds for the 1,000-review sample and in about 13 minutes for 200,000 reviews. Each step—preprocessing, MinHash, LSH, TF-IDF, and cosine scoring—scales roughly linearly in N (plus a term proportional to the number of LSH candidates), confirming that the solution remains practical on a single machine without any distributed framework. without any distributed framework.

7 Discussion

When we first experimented with exact Jaccard similarity on the 1,000-review sample, it clearly identified near-duplicate pairs—those with a Jaccard score above 0.8 were indeed virtually identical in content. However, as soon as we attempted to scale this method to the full 200,000-review corpus, it became obvious that computing Jaccard for every possible pair (nearly 20 billion comparisons) is simply infeasible. We needed a way to approximate set overlap without examining every pair directly.

MinHash combined with locality-sensitive hashing (LSH) provided exactly that. By reducing each review's shingle set to a 100-element signature, MinHash lets us estimate Jaccard similarity in constant time per signature component. Splitting those signatures into 20 bands of five rows each (the "banding" step) ensures that only reviews with high estimated overlap land in at least one common bucket. In other words, MinHash + LSH prunes the search space from all $\binom{N}{2}$ pairs down to a manageable few million candidates, with a high probability of including every true near-duplicate (around 95% recall in our sample tests).

That step on its own still produces some false positives—pairs that share a few common shingles purely by chance. To refine those candidates, we turned to TF-IDF vectorization with cosine similarity. Whereas Jaccard and MinHash operate on sets of word shingles, TF-IDF captures the weighted frequency of individual terms, reflecting the overall distribution of words in each review. Once MinHash + LSH has reduced the pool to a few million likely matches, we compute an exact cosine similarity between their TF-IDF vectors. In practice, setting a cosine cutoff at 0.7 eliminated nearly all spurious matches while retaining all truly near-duplicate pairs.

In summary, exact Jaccard was invaluable for validation on a small scale but not viable for the entire dataset. MinHash + LSH serves as a highly efficient filter, reducing the candidate set by orders of magnitude with only a small chance of missing genuine duplicates. TF-IDF + cosine similarity then provides a robust, fine-grained measure of textual closeness, discarding the remaining false positives. For our Amazon book-review dataset—200,000 documents of moderate length—this two-stage pipeline proved both fast enough to run in under fifteen minutes on a single machine and accurate enough to surface genuine near-duplicates without manual oversight.