Finding Near-Duplicate Reviews in Amazon Book Reviews using Jaccard and MinHash (LSH)

Luigi Gallo 41628A

September 14, 2025

Abstract

This project aims to implement a scalable pipeline to find near-duplicate and highly similar reviews in the *Amazon Book Reviews Dataset*[1] [1] from Kaggle. The pipeline ingests and parses the CSV file, performs text pre-processing (tokenization and stop-words removal) turning the reviews into sets of unique words, builds binary bag-of-words features, and computes the Jaccard similarity relying on MinHash signatures and Locality Sensitive Hashing (LSH) intended to approximate set similarity in large datasets. This approach focuses on scalability and replicability.

Contents

[1] https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews

1 Introduction

Modern review platforms accumulate vast amounts of user-generated text. Reposted content and near-duplicates skew statistics and waste storage.

In this project, we address near-duplicate detection with a focus on three different principles: efficiency, scalability, and reproducibility. The standard approach could be to check every pair of reviews and evaluate the similarity, but the complexity would grow quadratically ($O(n^2)$), making it infeasible very quickly. Instead, we treat each review as a set of unique tokens and use Jaccard similarity to measure the overlap. To avoid quadratic comparisons, we use MinHash with Locality Sensitive Hashing (LSH), which creates compact signatures whose collision probability follows Jaccard, plus multi-table hashing to retrieve only likely near-neighbors [2]. The pipeline is implemented in Apache Spark for scalability and reproducibility.

2 Data

Data are obtained from the *Amazon Book Reviews* Dataset. The original dataset provides two CSV files:

- **Books_data.csv**: details for 212 404 books, such as genres, authors, cover and description.
- **Books_rating.csv**: information about 3 000 000 book reviews for 212 404 books.

For the purpose of this project we will only focus on `Books_rating.csv`, more specifically the `review/text` field, any other columns are not required. We also assume that each row represents a single review.

The dataset has an identifier that refers to the book, and not the individual review, that's why we create a synthetic ID using Spark's `monotonically_increasing_id` function.

2.1 Descriptive analysis

After a process of robust CSV parsing (dealing with commas, newlines and quotes using Spark) we create a random sub-sample keeping 20% of the total reviews and then apply a 20 000 non-empty rows hard cap. We do this to limit the runtime on Colab due to the low amount of processing power of the free tier.

Looking at the data before sampling, we can see that the data quality is really good, with just 8 empty rows. In the hard-capped 20 000 review slice the length distribution is right-skewed:

- mean ≈ 842 characters
- median ≈ 531 characters
- $p_{90} \approx 1879$ characters

- $p_{99} \approx 4689$ characters
- min $= 1$ characters
- max $= 29\,336$ characters

As shown in Figure 1 and Figure 2

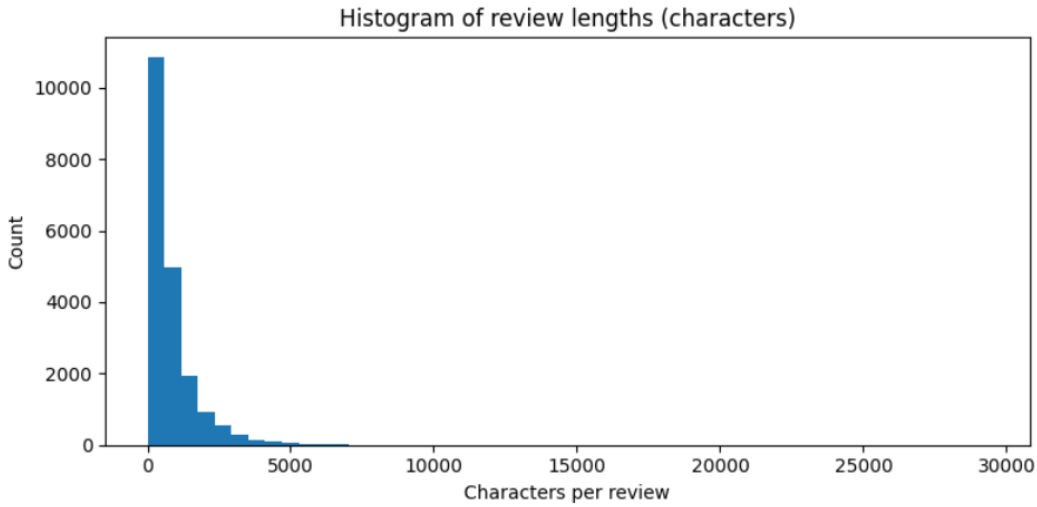

Figure 1: Histogram of reviews length distribution

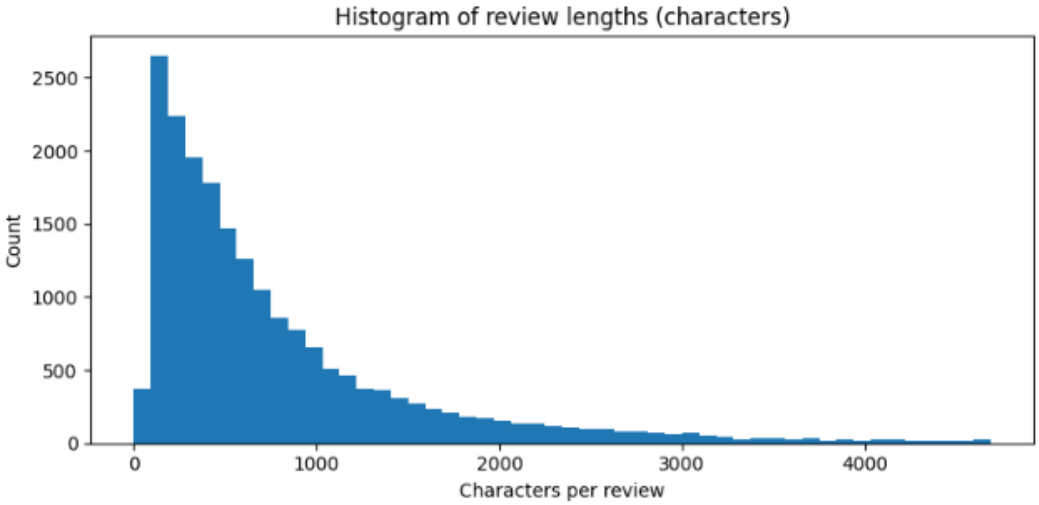

Figure 2: Histogram of reviews length distribution cut at 99 percentile

the histograms matches the results, a big mass of short-to-medium reviews and then a long, thin tail toward higher character counts that pulls the average up (heavy-tailed, right-skewed). It would be interesting to do a deeper analysis on these reviews, because a high number of really short ones (1-4 characters) would suggest the presence of random posts or mistakes.

3 Methodology

The goal is to detect near-duplicate reviews efficiently. As already discussed, instead of computing exact similarity for all pairs ($O(n^2)$), we treat each review as a set of tokens and measure overlap using Jaccard similarity, then we use MinHash combined with Locality Sensitive Hashing (LSH) in Spark to retrieve only likely neighbors.

3.1 Text representation

Each review is transformed into a single set of unique tokens. We split on non-alphanumeric characters, lowercase, and remove stop-words. A binary Bag-of-Words is used to check for presence/absence of a token, then rare tokens and empty token lists are removed.

1. **Tokenization**: split on non-alphanumeric characters and lowercase.
2. **Stop-words removal**: drop common words to reduce noise and overlap.
3. **Binary bag-of-words**: `CountVectorizer(binary=True)` encodes presence/absence (0/1) rather than counts, matching Jaccard similarity and reducing length bias.
4. **Vocabulary filtering**: `minDF` removes really rare tokens (ex. typos), improving speed and stability.
5. **Final check**: drop rows that end up with empty token lists or zero-feature vectors before LSH

3.2 Similarity and candidate generation with MinHash LSH

Let each review d map to a set of tokens A_d (presence/absence). For two reviews i, j:

$$J(A_i, A_j) \;=\; \frac{|A_i \cap A_j|}{|A_i \cup A_j|} \;\in\; [0, 1],$$

we want all pairs (i,j) with $J(A_i, A_j) \geq \tau$. Computing J for all $\binom{n}{2}$ pairs would be infeasible with a large n, that's why we first generate candidates likely to be similar and only then evaluate the similarity score.

Instead of Jaccard, Spark's LSH expects a distance $D(A, B) = 1 - Jaccard(A, B)$, where a value of 0 means identical sets, while a value of 1 means absence of overlapping.

A MinHash signature is a compact sketch of a set with the property that the probability two sets have the same hash value equals their Jaccard similarity ($\Pr[h(A_i) = h(A_j)] = J(A_i, A_j)$). Locality-Sensitive Hashing (LSH) then groups signatures into buckets across several hash tables, this technique exploits the fact that highly similar documents are more likely to collide in one or more hash tables and will end in the same bucket, making them candidates. We then check only those candidates, to see if their Jaccard similarity is above the set threshold, making the process a lot faster.

We remove self-matches and symmetric duplicates by keeping $doc_i d_A < doc_i d_B$. Finally, we rank pairs by similarity in descending order and show the top-K results.

4 Results

4.1 Scalability considerations

Computing the exact Jaccard for every possible combination would be impractical even for a modest number of reviews, that's why we use MinHash LSH, at the cost of a small approximation error near the threshold. To show the selectivity/runtime trade-off, we run 2 different settings: $\tau = 0.80$ with `SUB_MAX_ROWS = 20'000`, and $\tau = 0.60$ with `SUB_MAX_ROWS = 40'000`.

Setting A: $\tau = 0.80$**,** $N = 20{,}000$**.** Returned pairs (after de-duplication $A < B$):

- count $n = 402$
- avg $J \approx 0.993$
- min $J \approx 0.825$
- max $J = 1.0$
- median $= 1.0$
- $p_{90} = 1.0$
- $p_{99} = 1.0$

More than half of the retrieved pairs are token-identical ($Jaccard = 1.0$). This could be due to duplicate templates, spam, or identical reviews across different editions of the same book. The weakest returned pairs are strongly overlapping ($J \geq 0.825$), showing that $\tau = 0.8$ yields a high precision set of near-duplicates. Given the limited set ($n = 402$), we could also manually review the results if needed.

Setting B: $\tau = 0.60$**,** $N = 40{,}000$**.** Returned pairs:

- count $n = 1019$
- avg $J \approx 0.985$
- min $J \approx 0.608$
- max $J = 1.0$
- median $= 1.0$
- $p_{90} = 1.0$
- $p_{99} = 1.0$

Again, the majority of the retrieved pairs are token-identical, but avg J went down and min J is close to the new threshold $\tau = 0.60$, as expected. This means that with at the cost of more computing power we can produce a broader and more diverse set of candidates.

4.2 Runtime behavior

On the notebook environment on Colab, $\tau = 0.60$ with a 40 000 row cap took approximately 40 minutes end-to-end, while $\tau = 0.80$ with a 20 000 row cap took roughly one fourth of the time ($\sim$ 12 minutes). The difference is consistent with more candidate collisions, due to the looser threshold and larger input.

5 Conclusions

This project results shows that near-duplicate detection on a large set of reviews can be done accurately and efficiently with this pipeline:

1. Binary bag-of-words features (set of unique tokens).
2. Jaccard similarity on token sets.
3. MinHash LSH implemented in Spark.

By looking at the overlap of token sets we can avoid length bias, and the LSH removes the $O(n^2)$ bottleneck making the approach easily scalable with enough computing power. On the seeded 20 % sample with 20 000 row cap and threshold $\tau = 0.80$, we got $n = 402$ pairs of which most are token-identical. Lowering the thresholds and increasing the cap ($\tau = 0.60$, 40 000 rows) got us $n = 1019$ and lowered the minimum to 0.608.

Overall, the results follow our project goals: efficiency (no quadratic bottleneck), scalability (Spark + LSH), and reproducibility (notebook on Colab).

5.1 Improvements

There are still some improvements that could be implemented without changing the core project:

1. **Language filtering**: We could review the languages of the reviews,excluding non-english reviews or routing them to specific stop words lists.
2. **Length filtering**: Very short reviews tend to produce high or even perfect Jaccard with other short reviews, a minimum token length (ex. 4/5 tokens) would remove trivial matches.
3. **J=1.0 check**: The results showed an extremely high number of token-identical reviews. As already hinted before, this could be due to a number of factors, probably either spam, same reviews on different publications of the same book, and very short identical reviews. Length filtering could help both ways with

this issue, identical really short reviews are trivial for our study so we can remove them even if they are genuine matches, and identical really long reviews are so statistically unlikely that we can assume they are always spam or duplicates from the same user.

4. **Tokenization**: Instead of using unigram sets (one word tokens) a more precise approach would be to switch to shingles, this helps reduce false matches driven by generic words but also increase a lot the vocabulary size.

5. **Users analysis**: It would be interesting to see who is behind these reviews, if the duplicate ones are from different users or just one, if it's promotional material, spam or bots.

References

[1] Mohamed Bekheet (2022) *Amazon Books Reviews*, URL: https://www.kaggle.com/datasets/mohamedbakhet/amazon-books-reviews.

[2] Anand Rajaraman, Jure Leskovec, Jeffrey D. Ullman (2014) *Mining of Massive Dataset*.

[3] Spark *MinHashLSH*, URL: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.MinHashLSH.html

Declaration

"I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work, and including any code produced using generative AI systems. I/We 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/us or any other person for assessment on this or any other course of study."