Finding Similar Items

Antonio Focella

December 2025

This is my attempt at Project 1: *"Finding similar items"*. The task is to build a detector of pairs of similar Amazon book reviews. Checking every possible pair would be prohibitively slow ($O(N^2)$), so I rely on the standard techniques seen in the course: shingling, MinHashing, and locality-sensitive hashing (LSH). I implement everything in PySpark so that I can handle millions of reviews and keep the experiments replicable.

1 Goal and methodological approach

The goal is to identify pairs of reviews that are very similar (I focus on Jaccard similarity of 0.9, so near-duplicates) without doing brute-force comparisons across all pairs. To overcome the $O(N^2)$ bottleneck, I do the following:

(i) I convert each review into a set of k-shingles

(ii) I compress each set into a MinHash signature

(iii) I use LSH banding to generate a small number of candidate pairs

(iv) I verify each candidate pair by estimating Jaccard similarity from signatures and keeping only those above my threshold

As far as I know my work is not based on an existing published solution; I mainly follow what we saw in the course and focus on getting something correct, scalable, and reproducible, although probably not particularly original.

2 Dataset

2.1 Source and selected parts

I use the Kaggle dataset mohamedbakhet/amazon-books-reviews The dataset contains Amazon book reviews, so we are working with language and variable-length text.

From the unzipped archive I use Books_rating.csv. For practical reasons, I take the first 1,000,000 rows using limit(1000000).

2.2 Fields used

The only field I really need for similarity is the review text column review/text.

I also keep identifiers:

- I rename the dataset column Id to book_id (the original naming is a bit confusing/misleading).
- I create a new unique row identifier id with *monotonically_increasing_id()* so I can safely refer to reviews later.

3 Organize data

I use Spark DataFrames, starting from *reviews_df*.
Whenever I reuse intermediate results, I persist them (*MEMORY_AND_DISK*) and materialize the cache with a *count()*.

4 Pre-processing

4.1 Filtering

Before shingling, I remove:

- null review texts
- texts shorter than the shingle length k (otherwise shingling returns an empty set)

4.2 Shingling

I write a function that generates 10-shingles from each review. A “shingle” is just a substring of length k found within the document. This turns the problem into a set-based similarity problem, which can be handled more efficiently.

I use $k = 10$ character shingles: 10 should be large enough that random overlap is unlikely, but still robust to small edits. I also lowercase the text to make shingling case-insensitive. I use a set so that each review has unique shingles.

Listing 1: Shingle generator

```
shin_len = 10 # 10 should be large enough that random overlap is unlikely

def generate_shingles(text):
  if text is None:
    return []
  s = text.lower() # force lowercase
  shingles = set()
  for i in range(len(s) - shin_len + 1):
    shingles.add(s[i:i+shin_len]) # slide a window of length k across the string
  return list(shingles) # convert set to list (removes duplicates)
```

I implement shingling with a Python UDF. I also tried an alternative approach in pure Spark (without a Python UDF), but it appeared slower on my setup, so I keep it as legacy code in the notebook.

5 Algorithms and implementations

5.1 HashingTF

The set of all possible k-shingles is too big, so I reduce dimensionality using feature hashing:

- Transformer: *HashingTF*
- Dimension: 2^{20}
- Binary mode: *binary=True*

I also make sure rows with an empty shingle array are filtered to avoid exceptions in *MinHashLSH*.

5.2 MinHash signatures

I then create a MinHash signature for each set. MinHashing compresses a set into a small signature, and the key property is that the fraction of equal signature rows gives an unbiased estimate of Jaccard similarity.

Spark provides *MinHashLSH*, which I use to compute signatures efficiently.

5.3 LSH banding

Even with signatures, comparing all pairs is still too expensive, so I use LSH banding.

I split each signature into b bands of r rows:

$$b = 6 \qquad r = 16 \qquad n_{\text{sig}} = b \cdot r = 96$$

For each band I hash the band slice into a bandKey. Two reviews become a candidate pair if they share a bandKey in at least one band.

Candidate generation can still blow up if a bucket is too big (a bucket with k items produces $O(k^2)$ pairs). To keep the code from becoming too slow, I drop buckets with more than 200 elements (this can be modified by changing *MAX BUCKET* in the code). This makes the pipeline faster, but I might miss some true pairs.

5.4 MinHash and Jaccard estimate

For each candidate pair I compute the estimate of the Jaccard distance based on signatures similarity. I keep only pairs whose estimated Jaccard similarity is above a chosen threshold t (in my experiments, $t = 0.9$, although I also tried $t = 0.8 and works fine$). At that point, I consider the near-duplicate detection goal achieved.

6 Scalability

The code is designed so that most work is parallel:

- Shingling and hashing are done independently for each review
- MinHash signature computation is also for each review
- LSH reduces the pairs to likely pairs only
- Distance is estimated only on candidate pairs, not all pairs

Spark distributes the computation across available cores (though they were only 2 in my Colab...), so the approach scales to large datasets in practice, especially if more CPU cores are available.

A potential bottleneck is candidate pairs "explosion" inside large buckets. I also tried diferent (b, r) choices, but couldn't increase their values too much otherwise the code becomes simply too long.

7 Experiments

I run everything in a notebook using PySpark in local mode (`master("local[*]")`). A quick hardware check shows I have two CPU cores available. I set the Spark driver memory to `8g`.

On my setup, the code takes about an hour to execute using 2 CPU cores.

I validate the output in a simple way:

- I print a sample of returned pairs and inspect them

- I also pull a few specific `id`'s and display the corresponding review texts to confirm they are actually similar

A quick human check confirms that the pairs are indeed very similar.

8 Results

Overall, the results match what I expected: the method finds reviews that are essentially the same or extremely close (reposts, minor edits, small formatting changes, subsets of other reviews, ...).

The two main trade-offs I see are:

- **Precision VS tractability from LSH:** With $b = 6$ and $r = 16$, I bias towards efficiency of the code, higher values might be preferable
- **Speed VS recall from bucket filtering:** *MAX_BUCKET* makes the execution of the code more manageable, but it might remove some true matches inside very large buckets.

9 Replicability

To reproduce the experiments:

1. Set Kaggle credentials in the environment
2. Download and unzip the dataset
3. Run the notebook cells in order (Spark session → load Books_rating.csv → preprocessing → hashing → MinHash/LSH → filtering)
4. Optional: write the output pairs to CSV (I use coalesce(1) if I want a single file)

Key parameters

Table 1: Main hyperparameters used in my implementation.

Component	Value
Dataset file	`Books_rating.csv`
Rows used	1,000,000 (`limit`)
Shingle length k	10 (character shingles)
HashingTF dimension d	2^{20}
HashingTF binary	True
Bands b	6
Rows per band r	16
Signature length n_{sig}	96
Max bucket size	200
Similarity threshold τ	0.9

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.