Finding Similar Book Reviews

Algorithms for Massive Data
Master in Data Science for Economics

Academic Year 2024/25

Stefano Chiesa
Università degli Studi di Milano

May 21, 2025

1 Introduction

This report presents the work for Project 1 of the course *Algorithms for Massive Data*. The goal is to develop a system to identify similar pairs of book reviews in a scalable manner, using the Amazon Books Review dataset. The idea is to compute textual similarity between user reviews in a scalable way. The code of the project can be found at the following GitHub repository.

2 Dataset and Data Selection

The dataset used is the *Amazon Books Review* dataset, available on Kaggle. Only the `Books_rating.csv` file has been considered, focusing on the `review/text` column, which contains users' reviews. To ensure reproducibility and control over data volume, a global variable allows switching between different sampling percentages. A 1% sample (around 30k entries) was primarily used during the test phase. After that, different sampling percentages has been used to test the scalability of the solution.

3 Data Organization

The data has been loaded using the Kaggle API during code execution. Reviews were organised into a Spark Dataframe structure, with each row corresponding to a single review. Pre-processing steps were applied to prepare the text for similarity computation.

4 Pre-processing Techniques

The following pre-processing steps were applied to the text: lowercasing of all characters, since capital letters are not relevant for our analysis; then, every null or empty value was removed from the dataset; after that, tokenisation was done using Spark's RegexTokenizer, which returns as an output every word. The tokenisation still needed some

processing, since not all tokens are relevant if the goal is to find meaningful similarities: stopwords such as "the", "a" and "thus" appear in several documents, but are not meaningful alone. Hence, they were removed using an in-built Spark function. The same logic applies to numbers, which were removed as well. Since the in-built function has a dictionary of stopwords made from several texts of different topics, we needed a personalised list. Words as "book" are not considered stopwords in general, but in a dataset containing book reviews, we can consider it not meaningful in understanding the similarity between different records. Hence, tokens appearing more than a certain number of times were removed. The formula for the threshold is the following:

$$(20\%) * S \tag{1}$$

Where S is the sample size or the number of records. The percentage was chosen with a qualitative analysis of the top frequent words, taking into consideration a 1% sample, but it could be fine-tuned. The list of high-frequency tokens was then analysed: the words *"Good"* and *"Great"* were considered high frequency, but I considered them meaningful in a sentiment analysis perspective. Hence, I decided not to discard these tokens. Discarding them, I was obtaining that the top similar pair of reviews were *"Good for everyone* and *"Not for everyone"*, which have an opposite meaning. After preprocessing, reviews were represented as sets of tokens, suitable for computing Jaccard similarity. Empty token sets were discarded, since Spark does not work very well with empty tokens. In the end, duplicates in the column `review/text` were discarded, since the reviews that are identical are not very meaningful in my opinion. It might be useful to find duplicates, maybe to identify bots or bought reviews, but it is not what I want to do here.

5 Algorithms and Implementations

5.1 Local Sensitive Hashing

I implemented a similarity detection pipeline using Locality Sensitive Hashing with MinHashing. The algorithm works like this: each set is converted into a compact signature that captures its content. These signatures are obtained with h hash functions, in our case $h = 8192$. The number is not arbitrary, but correspond to 2^n, where n is the number of bits. I used 8192, since it is the first power of 2 which is greater than 5000, the size of the vocabulary. This is important, since we want to avoid collisions in the hashing process. See HashTF documentation for more information. The signature is then divided into bands, and each band is hashed into a bucket. In my case, the number of bands is equal to $b = 10$, an arbitrary number that could be fine-tuned. If two reviews land in the same bucket in at least one band, they are considered candidate pairs. Only these candidate pairs are then compared in detail to check for similarity. This approach allows for saving computational resources and time, but could ignore reviews that are similar. It is a trade-off. More information can be found by reading (Rajaraman and Ullman 2014) in Chapters 3.3.4 and 3.4.1. The algorithms were implemented using the `HashingTF` and `MinHashLSH` functions from `Spark`.

5.2 Jaccard Similarity

The Jaccard Similarity is a measure of similarity between two sets, defined as the size of their intersection divided by the size of their union. Formally, for two sets A and B, the

Jaccard Similarity $J(A, B)$ is given by:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

It ranges from 0 (no overlap) to 1 (identical sets). To be precise, in my code, I used the Jaccard Distance, which is:

$$D(A, B) = 1 - J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|} \tag{3}$$

It ranges from 0 (identical sets) to 1 (completely disjoint sets). The reason is that the Spark library does not allow for computing the Jaccard Similarity directly. In practice, it is the same, once you know how to interpret the result.

6 Experimental Setup

Experiments were conducted by varying the dataset size, respectively 1%, 2% and 4%. These percentages help estimate the scalability of the solution, checking for the pattern of change in the execution time. All experiments were run on a local machine. The 1% solution was also tested on Google Colab, but the execution time depends on the available resources from Google. The MinHash LSH model was trained using the `MinHashLSH` class from PySpark's MLlib.

7 Results and Discussion

Info	**1% sample**	**2% sample**	**4% sample**
Time(s)	212	451	2947
Computed?	Yes	Yes	Yes

Table 1: Computation of Jaccard Similarity across different sample sizes

In all the examples took in consideration, the local machine was able to execute the whole code. This means the result could fit the 30GB memory that was available on it. On the other hand, the time taken for the first test compared to the second is approximately doubled, but the duration for the third test explodes (+553%). This indicates that the computation time does not follow a linear trend. In terms of the quality of the results, the code can effectively find similar reviews. Without any other data pre-processing, the top pair for Jaccard Similarity is almost a couple of duplicates, with minor differences. This means that the code could be improved if the goal is also to filter out potentially bot-published reviews. To be fair, I think that nowadays bots do not publish identical reviews anymore, since with generative AI, it has become very easy to write thousands of unique lines of text. The computational time difference in data pre-processing was negligible.

8 Conclusion

The approach successfully found similar pairs of reviews while significantly reducing computational cost through Local Sensitive Hashing. The experiments showed that while the

algorithm performs efficiently on small samples (1% and 2%), the computation time increases more than linearly as the dataset size grows. This suggests that, although the method is scalable, to apply this method on the whole dataset, multiple machines in parallel or a more powerful machine might be needed. Further improvements could involve better parameter tuning and better tokenisation, for example, using SpaCy library that has several in-built dictionaries and also allows for lemmatisation of the words.

9 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.

References

Rajaraman, Anand and Jeffrey Ullman (2014). *Mining of Massive Datasets*. Free to download at `http://infolab.stanford.edu/~ullman/mmds/book.pdf`.