Project 1: Finding Similar Book Reviews

Algorithms for Massive Data

Gianluca Morena (42508A)

Master in Data Science for Economics

Academic Year 2024/25

Università degli Studi di Milano

"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/We 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."

1 Introduction

This report shows the implementation and results of an algorithm for finding similar pairs of book reviews in a scalable way, according to the requirements defined in Project 1 of the *Massive Data Algorithms* course. The methodology used involves first transforming the Amazon Books Reviews dataset, importing it via API from the respective KaggleI page, creating a 1% sample of the data, performing data cleaning procedures, and finally calculating the jaccard similarity. The code is implemented using Python version 3.12.12 and, for working with large datasets, PySpark version 4.0.1, implemented using a Jupyter Notebook executable available on Google Colab. The code can be downloaded from the following link on GitHub

2 Dataset

The dataset from which the data is imported is Amazon Books Review, which can be downloaded from this link. The dataset contains two .csv files, `books_data.csv` and `Books_rating.csv`, representing the dimensions table and the facts table, respectively. For the purpose of this project, only Books_rating.csv is imported, containing 3M book reviews for 212,404 unique books and the users who gave these reviews for each book.

3 Data Selection and Data Organization

The data is imported via the Kaggle API, specifically for some subdatasets as reported in Section 2. The Books_rating.csv dataset was organized into a Spark Dataframe, with each row representing a book review, identified by the reviewed book's ID (`Id`), the reviewer's ID (`User_id`), and the review

text (`review/text`).

Subsequently, given the large size of the dataset (1.06G), to make the code implementable and scalable, I opted to create a sample of the original file using a Python user-defined function. The algorithm of the Python user-defined function is the following:

Algorithm 1 Sample Size

Require: Dataset df, sampling fraction $percent$, dataset name $output\_name$ (default: "dataset"), random seed $seed$

Ensure: Sampled dataset df_{sample}

if $percent$ is not numeric **then**
  raise error: "The percent value must be numeric"
end if
if $percent \leq 0$ **or** $percent > 1$ **then**
  raise error: "The percent value must be in the interval $(0, 1]$"
end if
Print $output\_name$ "–> sampling" $percent \times 100$ "% of the dataset"
$df_{sample} \leftarrow$ RandomSample(df, without replacement, $percent$, $seed$)
return df_{sample}

4 Preprocessing Techniques

The preprocessing phase primarily focuses on removing and transforming string values that are not useful for the implementation of the algorithm, such as stop words, whitespace, and repeated values, as they can significantly increase execution time. In addition, a tokenization step is applied in order to transform textual sequences into arrays of words (tokens) that are relevant for similarity search.

The first step consists in removing all rows from the DataFrame containing null values in the `user_id` and `text` columns, as these entries represent noise for the algorithm. Subsequently, the DataFrame is converted into a *Resilient Distributed Dataset* (RDD) using the `.rdd` attribute of Spark's `DataFrame` class, which simplifies the implementation of the tokenization procedure by allowing custom transformations to be applied element by element.

The tokenization procedure is implemented as a userdefined Python function and is applied to the textual field of each record (the `review_text` attribute). In the first transformation, a new pair is constructed for each element of the RDD, where the original fields of the record (identifier, rating, and review text) are preserved, while the tokenization function is applied to the text, returning an array of tokens. Subsequently, the resulting nested structure is flattened through a second `map` transformation, producing an RDD in which each element is represented by a tuple containing the identifier, rating, original text, and the corresponding set of tokens. This step enriches each review with its tokenized representation while preserving the original information required for the subsequent stages of the analysis. The tokenization algorithm processes each review text by applying regular-expression-based splitting implemented through the Python `re` library. The resulting tokens are then filtered to remove stopwords, numeric tokens, and empty strings. Next, they are converted from RDD to a Spark DataFrame. A filter operation is then applied to the resulting DataFrame to remove all records with empty token arrays. As a result, only revisions containing at least one valid token are retained, ensuring that the input to subsequent analysis steps is informative and meaningful.

Algorithm 2 Tokenizer

Require: Input string s, stop-word set S, splitting regular expression r

Ensure: Array of filtered tokens T

$T \leftarrow \emptyset$
$s \leftarrow \text{ToLowerCase}(s)$
$R \leftarrow \text{Split}(s, r)$
for each token t in R **do**
  if $t \neq \emptyset$ **and** $t \notin S$ **and not** $\text{IsNumeric}(t)$ **then**
    append t to T
  end if
end for
return T

5 Algorithm and Implementation

5.1 Local Sensitive Hasting

The algorithm implements a scalable pipeline for approximate similarity search over textual data by combining feature hashing and locality-sensitive hashing (LSH). Using the column of the dataframe containing the token array, the method maps text representations into feature vectors for efficient similarity computation In more specific way:

- **Feature hashing.** In the first step, the tokenized text is transformed into a fixed-dimensional vector space using the Hashing Term Frequency (HashingTF) technique. Each document is represented as a sparse numerical vector whose dimensionality is determined by the parameter `numFeatures`. The hashing function maps each token to one

of the predefined feature indices, accumulating term frequencies without explicitly constructing a global vocabulary. This approach enables memory-efficient and scalable feature extraction, as it avoids maintaining a dictionary of unique tokens and supports streaming data. In the presented configuration, the feature space dimensionality is set to `h` = 8192 to achieve a balance between representational capacity and computational efficiency. The dimensionality `h` corresponds to 2^n, where n is the number of bits, providing a sufficiently expressive hashed feature space for similarity estimation tasks.

- **MinHash LSH indexing.** In the second step, the algorithm applies the MinHash Locality-Sensitive Hashing (MinHashLSH) technique to the hashed feature vectors. MinHashLSH approximates the Jaccard similarity between documents by computing hash signatures that preserve similarity under hashing. The parameter `numHashTables` controls the number of independent hash tables used by the LSH scheme, influencing the trade-off between recall and query efficiency. In this implementation, two hash tables are employed in order to limit computational overhead while maintaining acceptable similarity detection performance.

5.2 Jacard Similarity (Distance)

The Jaccard similarity (J) is a measure of similarity between two sets (A) and (B), defined as the ratio between the cardinality of their intersection and the cardinality of their union:

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

In our setting, the Jaccard distance is employed, defined as

$$d_J(A, B) = 1 - J(A, B),$$

which takes values in the interval $[0, 1]$, where 0 denotes identical sets and 1 denotes disjoint sets.

5.3 Experimental Setup

Experiments were conducted by varying the dataset size, using sampling rates of 1%, 2%, and 10%. These configurations were adopted to evaluate the scalability of the proposed solution by analyzing how execution time changes as the dataset size increases. All experiments were executed on a local machine. The 1% configuration was also tested on Google Colab. However, execution time in this environment depends on the availability of computational resources provided by Google.

5.4 Results and Conclusion

As the sample size increases, the execution time increases. As can be seen from the table below, it increases monotonically with respect to the sample size. A linear execution time:

Sample Size	**Execution Time (min)**
1%	3
2%	8
10%	40

Table 1: Execution time for different sample dataset

The algorithm produces reliable similarity estimates for the analyzed objects. The combination of user-defined Python functions with advanced

PySpark components, such as the `MinHashLSH` class, represents an effective and flexible compromise between handling complex computational tasks and maintaining code simplicity and modularity.

However, the scalability experiments highlight several directions for improvement. In particular, additional transformations of the review texts should be considered to further reduce the computational load. Possible solutions include more text normalization, stricter filtering of low information tokens.

6 References

Rajaram, A. and Ullman, J. (2014). *Mining of Massive Datasets.* Available at `http://infolab.stanford.edu/~ullman/mmds/book.pdf`.