Project Report: Finding Similar Book Reviews

Algorithms for massive data (DSE)

Author: Karan Kooshavar (28904A)

Abstract

This report presents a project at finding similar book reviews in a dataset using a scalable approach. The main challenge is to compare a larg scale of text documents to find pairs that are meaningfully similar, not just literally identical. To achieve this, we use the Apache Spark for data processing and the Locality Sensitive Hashing (LSH) with the MinHash method to find the Jaccard similarity. The methodology has data pre-processing, including the adding of a unique ID for each review, text cleaning, tokenizing, and stemming, followed by vectorization. The LSH is then used to find candidate pairs, which are subsequently refined through a series of filters to remove unnecessary duplicates and get a better quality in the final results. The report discusses the chosen dataset, the implementation of the pipeline, the scalability, and provides a compelete analysis of the results, showing the usefulness and correctness of the method.

Table of Contents

1. **Introduction**
    - 1.1. Problem Statement
    - 1.2. Methodological Approach
    - 1.3. Report Structure
2. **The Dataset**
    - 2.1. Source and Description
    - 2.2. Data Organization and Considered Fields
3. **Data Pre-processing Techniques**
    - 3.1. Initial Cleaning and Sampling
    - 3.2. Text Normalization
    - 3.3. Tokenization, Stopword Removal, and Stemming
    - 3.4. Feature Vectorization
4. **Algorithms and Implementation**
    - 4.1. Core Algorithm: Locality Sensitive Hashing (LSH)
    - 4.2. Jaccard Similarity and MinHash
    - 4.3. Implementation in Spark ML
    - 4.4. Post-processing and Filtering Logic
5. **Scalability of the Solution**
    - 5.1. The Role of Apache Spark
    - 5.2. Caching Strategy for Performance
    - 5.3. Tuning LSH for Scalability
6. **Description of Experiments**
    - 6.1. Environment Setup
    - 6.2. Experimental Pipeline
    - 6.3. Replicability
7. **Comments and Discussion on Experimental Results**
    - 7.1. Analysis of Candidate Pairs
    - 7.2. Distribution of Jaccard Distances
    - 7.3. Effectiveness of Filtering
8. **Conclusion**
    - 8.1. Summary of Findings

1. Introduction

1.1. Problem Statement

Modern data-mining procedures, often called "big-data" analysis, require us to manage large amounts of data fast. In many of these cases, the data is regular, and there is big opportunity to use parallelism. Actually a million items ends up ina half a trillion pairs to process, which is very heavy even with an large scale of hardware and super-computers. This project presents the case of finding similar book reviews in a big dataset in a scalable and optimized manner.

1.2. Methodological Approach

To overcome the challenge of scalability, this project employs a robust, multi-stage pipeline built on Apache Spark. This pipeline is designed to get its parallelism not from a "super-computer," but from "computing clusters"—big collections of hardware connected. The core of the workflow is the **Locality Sensitive Hashing (LSH)** method. LSH is a set of techniques that will focus on pairs that are more likely to be similar, without looking into all pairs. LSH is a technique that hashes likely input items into the same "buckets" with high probability, extremely reducing the amount of comparisons needed.

The overall methodological approach is as follows:

1. **Data Ingestion and Sampling:** Load the dataset and take a representative sample for experimentation.
2. **Text Pre-processing:** Clean and normalize the review text to prepare it for analysis.
3. **Feature Engineering:** Convert the cleaned text into numerical feature vectors using the Bag-of-Words model (CountVectorizer).
4. **LSH Modeling:** Apply the MinHash for Jaccard Distance LSH model to find candidate pairs of similar reviews.
5. **Result Refinement:** Apply a series of intelligent filters to remove trivial duplicates.
6. **Analysis:** Analyze and visualize the final results to understand the distribution of similarity.

1.3. Report Structure

This report is structured to present a clear and understandable overview of the project. Section 2 presents the dataset used. Section 3 shows the data pre-processing methods. Section 4 explains the LSH algorithm and its use. Section 5 explains the scalability of the chosen methods and techniques. Sections 6 and 7 show the experiments and analyze the results. Finally, in section 8 we conclude the whole report with a short summary.

2. The Dataset

2.1. Source and Description

The project uses the **Amazon Books Reviews** dataset, published on Kaggle. This dataset is a big amount of text data, with some metadata for each review. For the aim of replicability, the dataset is downloaded with code execution by the use of with the Kaggle API.

2.2. Data Organization and Considered Fields

The dataset has a structured CSV file named Books_rating.csv. Following columns are considered useful for the project:

- **Title:** The title of the book being reviewed.
- **User_id:** A unique identifier for the user who wrote the review.
- **review/text:** The full text content of the review.

Other metadata from the dataset, such as the book Id, price, and review scores, were not used in the similarity comparison.

3. Data Pre-processing Techniques

A pre-processing was needed to ensure the quality of the text features and the accuratly detecting the similarity. The steps were coded in PySpark for the reason of scalability.

3.1. Initial Cleaning and Sampling

1. **Loading Data:** The Books_rating.csv was introduced into a Spark DataFrame.
2. **Sampling:** A random sample of **50,000 rows** was chosen from the 3M rows of the main dataset.
3. **Handling Nulls:** Rows with null values in the essential Title or review/text columns were dropped.
4. **Filtering Number-Only Reviews:** An early filtering step was introduced to remove reviews that consisted only of numbers and spaces, because they had no meaningful data for comparing.

3.2. Text Normalization

Title and review/text columns were normalized to create clean_title and clean_text. This involved:

1. **Lowercasing:** Converting all text to lowercase to have case-insensitivity.
2. **Removing Punctuation:** Removing all characters that are not alphanumeric or whitespace.
3. **Normalizing Whitespace:** Replacement of multiple whitespaces with a single whitespace.
4. **Trimming:** Removing extra whitespace at the beginning and the end of each string.

3.3. Tokenization, Stopword Removal, and Stemming

Cleaned review text (clean_text) was also processed to get a list of approprait tokens:

1. **Tokenization:** The text was split into individual words using Spark's Tokenizer.
2. **Stopword Removal:** Common words like (e.g., "the", "a", "is") were deleted using Spark's StopWordsRemover using the standard English stopword list from the NLTK.
3. **Stemming:** The rest of the words were turned into their root words using the **Porter Stemmer** from NLTK. For example, "reading", "reads", and "read" all become "read".
4. **Generating a Unique Review ID:** After text processing, a column, review_id, was added using monotonically_increasing_id() function. This step assigns a unique id to each review row, which will be necessary for correctly finding the duplicate reviews.

3.4. Feature Vectorization

To use machine learning algorithms, the list of tokens created must be converted into numerical vectors.

1. **CountVectorizer:** This method was used to convert the tokens into a sparse vector. Each dimension of the vector is assigned to a word, and the value is the times that word is repeated in that specific text.
2. **Configuration:** CountVectorizer was set with minDF=2 (a word must be in at least two reviews to be chosen to be in the vocabulary) and vocabSize=10000.
3. **Filtering Zero-Vectors:** After all the steps taken before, some vectors might result in an all-zero vector. And the MinHash LSH algorithm cannot use them, they were deleted.

4. Algorithms and Implementation

4.1. Core Algorithm: Locality Sensitive Hashing (LSH)

LSH is the main part of the pipeline of this project to be scalable. It is a set of techniques that hashes items using many different hashing methods. The hash functions have the property that pairs are much more likely to get into the same bucket if the items are more similar. Then we will be able to compare only the candidate pairs, which are pairs of vectors that got in the same bucket for at least one of the hash functions used by the LSH. Instead of comparing all the pairs, we only need to compare pairs that were found similar in one or more functions.

4.2. Jaccard Similarity and MinHash

For the text similarity comparison, a common method is the **Jaccard Similarity**, which computes the similarity of two sets. It is formulated as the fraction of the size of the intersection of the sets to the size of their union.

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

MinHash algorithm is a LSH made to approximate the Jaccard Similarity. It works by picking a random permutation of the rows of the characteristic matrix of the sets. The minhash value for any set is the number of the first row, in the permuted order, in which the column for that set has a 1. The probability that the minhash function for a random permutation produces the same value for two sets equals the Jaccard similarity of those sets. The set of these indices forms the "signature" for the review. The comparison of two signatures is a good approximation of the Jaccard Similarity.

4.3. Implementation in Spark ML

Spark's Machine Learning library (spark.ml) has an integrated MinHash for LSH.

1. **MinHashLSH Model:** A MinHashLSH system was created, taking the features column from CountVectorizer as input.
2. **numHashTables:** The parameter was set to 20. The trade-off between accuracy and speed. The higher the numHashTables the more the probability of finding all true positive pairs at the cost of larger amounts of processing.
3. **approxSimilarityJoin:** The approxSimilarityJoin technique was used to perform a self-join on the dataset. This method returns pairs of data which their Jaccard Distance is under a specified threshold. In this project, we used a threshold of 0.6, which corresponds to a Jaccard Similarity of 0.4 or higher.

4.4. Post-processing and Filtering Logic

The first candidate pairs from the LSH join require further filtering to be more trustable.

1. **Removing Trivial Duplicates:** The self-join will result in a situation where the pairs like (A, B) also show up as (B, A), and each review is also joined with itself. To prevent this, a filter was used by use of the unique review_id introduced in the pre-processing step. The condition datasetA.review_id < datasetB.review_id establishes a condition that only one of the two symmetric pairs is kept and that a review is not compared with itself. This also correctly allows to have for comparisons between multiple reviews for the same book.
2. **Filtering by Title Similarity:** To handle some book titles like "crime and punishment penguin classics" and "crime and punishment" to be counted as the same book if there is an identical review for them, a filter was used to delete the pairs where the reviews were identical (JaccardDistance == 0.0) and the tokenized words in one title was a subset of the other.
3. **Filtering by Same User:** Also a final filter was added to delete pairs where the reviews were identical (JaccardDistance == 0.0) and were written by the same User_id. To prevent internet trolls and spam reviews to be counted in our project.

5. Scalability of the Solution

5.1. The Role of Apache Spark

The whole pipeline was built implementing Apache Spark, a framework programmed for distributed computing. This means the solution is based on scalability. This parallel-computing architecture, sometimes also called cluster computing, is organized with compute nodes stored on racks. The nodes on a single rack are connected by a network, and racks are connected by another bigger level of network or a switch. Software stack begins with a new form of file system, called a "distributed file system," which has much larger units than conventional disk blocks and provides replication of data to protect against failures. On the other hand, programming systems such as MapReduce, and also the more advanced successor Spark, are designed to perform computations more efficiently and in a way that is hardware failures would be tolerated. As the dataset size grows from e.g. millions to billions of rows, the same exact code can be run on a bigger Spark cluster, distributing the calculation load across multiple machines without a need to make changes to the core logic of the code.

5.2. Caching Strategy for Performance

Spark's lazy evaluation makes sure that transformations are not computed until an action is called. To prevent re-computation of heavy calculation stages, a robust caching approach was used.

- df_clean.persist(StorageLevel.MEMORY_AND_DISK): The DataFrame was cached after the pre-processing and vectorization algorithms were applied. This ensures that the large text processing steps are done only once.
- candidate_pairs.cache(): The result of the approxSimilarityJoin was also cached. This is the most important caching step, as the join is the most computationally extreme part of the code. By caching this result, all the later applications such as filtering, analysis, and visualization steps become very fast.
- final_pairs.cache(): The final filtered DataFrame was also cached to speed up the multiple actions performed using it.

5.3. Tuning LSH for Scalability

The numHashTables parameter in the MinHashLSH model is a key identifier for managing scalability.

- **Low numHashTables:** Results in a faster join but with lower accuracy (it might be missing some similar pairs).
- **High numHashTables:** Results in a more accurate join but can be very slower as more pairs are generated.
  The used value of 20 gives a good trade-off for a dataset of this size, providing good

accuracy within a reasonable runtime.

6. Description of Experiments

6.1. Environment Setup

- **Framework:** Apache Spark 4.x
- **Language:** Python 3.9.6 (PySpark)
- **Libraries:**
  - pyspark
  - nltk (for Porter Stemmer and stopwords)
  - matplotlib (for visualization)
  - kaggle (for data download)
- **Execution Environment:** Jupyter Notebook running on a local machine with Spark configured to use 8g of driver memory.

6.2. Experimental Pipeline

The experiment followed the exact sequence of steps described in Sections 3 and 4. The notebook consists of logical cells, each performing a different step:

1. Library Importing and Installation.
2. Conditional Dataset Download from Kaggle API**.**
3. Spark Session Initialization.
4. Data Loading and Sampling (**50,000 rows**).
5. Text Cleaning, Null Removal, and Normalization.
6. Filtering non-text Reviews.
7. Tokenization, Stopword Removal, and Stemming.
8. Adding a unique review_id to each row**.**
9. Vectorization and Caching of the cleaned data (df_clean).
10. LSH Model Training and approxSimilarityJoin, followed by caching (candidate_pairs).
11. Application of advanced filters (title subset and same user) and caching (final_pairs).
12. Analysis and Visualization of the results.
13. Resource Cleanup (.unpersist()).

6.3. Replicability

The whole project is fully replicable. The notebook simillar_reviews_amd.ipynb contains the complete, executable code. To make sure a consistent data source, the notebook downloads the dataset from Kaggle using its API. This removes the need af manual data allocation.

To replicate the experiment, a user must:

1. **Install the required libraries:** pyspark, nltk, matplotlib, and kaggle.

2. **Provide Kaggle API credentials:** The user must replace the “xxxxxx” placeholder credentials in the data download cell with their own Kaggle username and API key.
3. **Run the notebook:** Executing the notebook from top to bottom will reproduce the exact same results, thanks to the use of a fixed seed (seed=42) in the random sampling step.

7. Comments and Discussion on Experimental Results

7.1. Analysis of Candidate Pairs

The initial approxSimilarityJoin with a Jaccard Distance threshold of 0.6 on the **50,000-row** sample resulted in a set of **2,078** candidate pairs. After applying the advanced filtering rules (for subset titles and same-user reviews), this number was reduced to **1,170** final pairs. This shows that nearly **44%** of the first candidates were not wanted duplicates, putting importance on the post-processing steps.

Final output table, which shows the original review text, titles, and user IDs, allows for presenting an overview of the results. The pairs identified present a set of reviews using very similar vocabulary and reviews talking about similar themes.

7.2. Distribution of Jaccard Distances

To visualize the distribution of similarity, histograms of the Jaccard Distances were drawn for both the candidate pairs and the final filtered pairs.

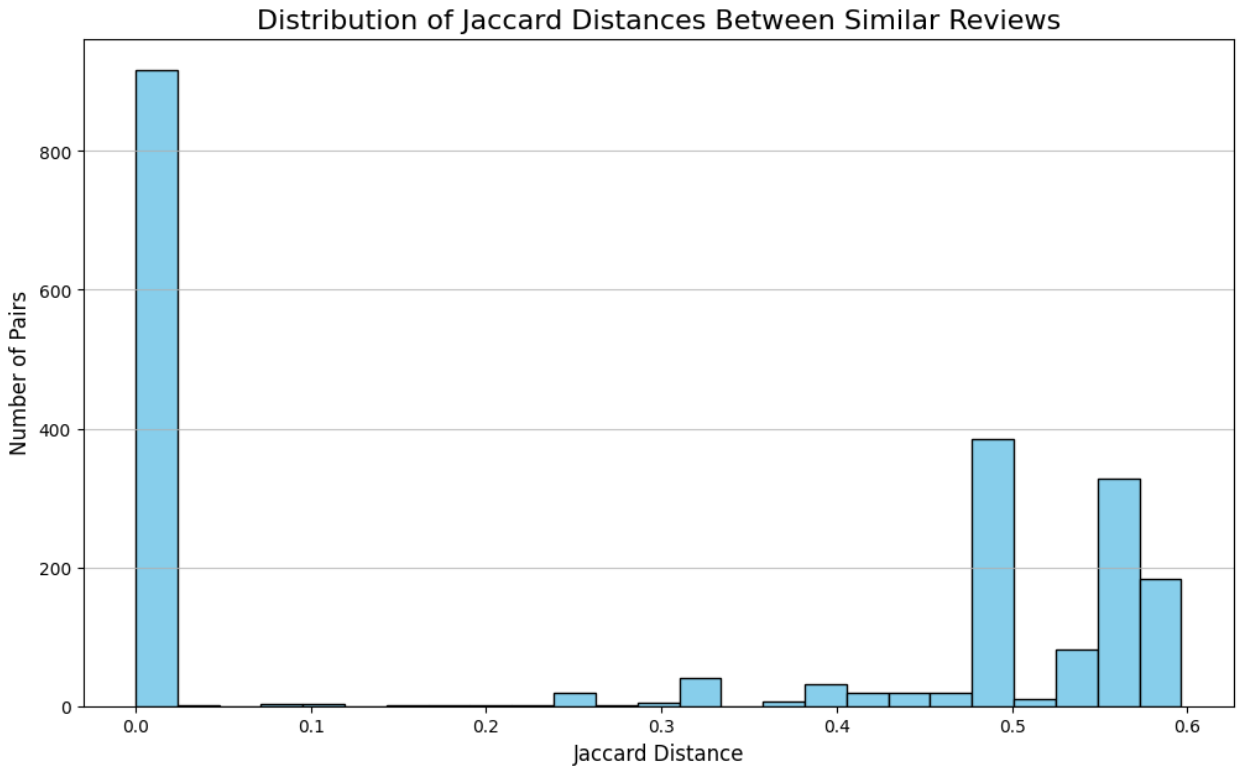

Figure 1: Distribution of Initial Candidate Pairs

Figure 1 histogram shows the distribution for all **2,078** candidate pairs found by the LSH join. A very notable specification seen is the very large spike at JaccardDistance = 0.0. This shows that pairs of reviews that were literally identical after pre-processing. While some of these are useful for our purposes, many are unnecessary duplicates, such as the same review being posted for different prints and editions of a book.

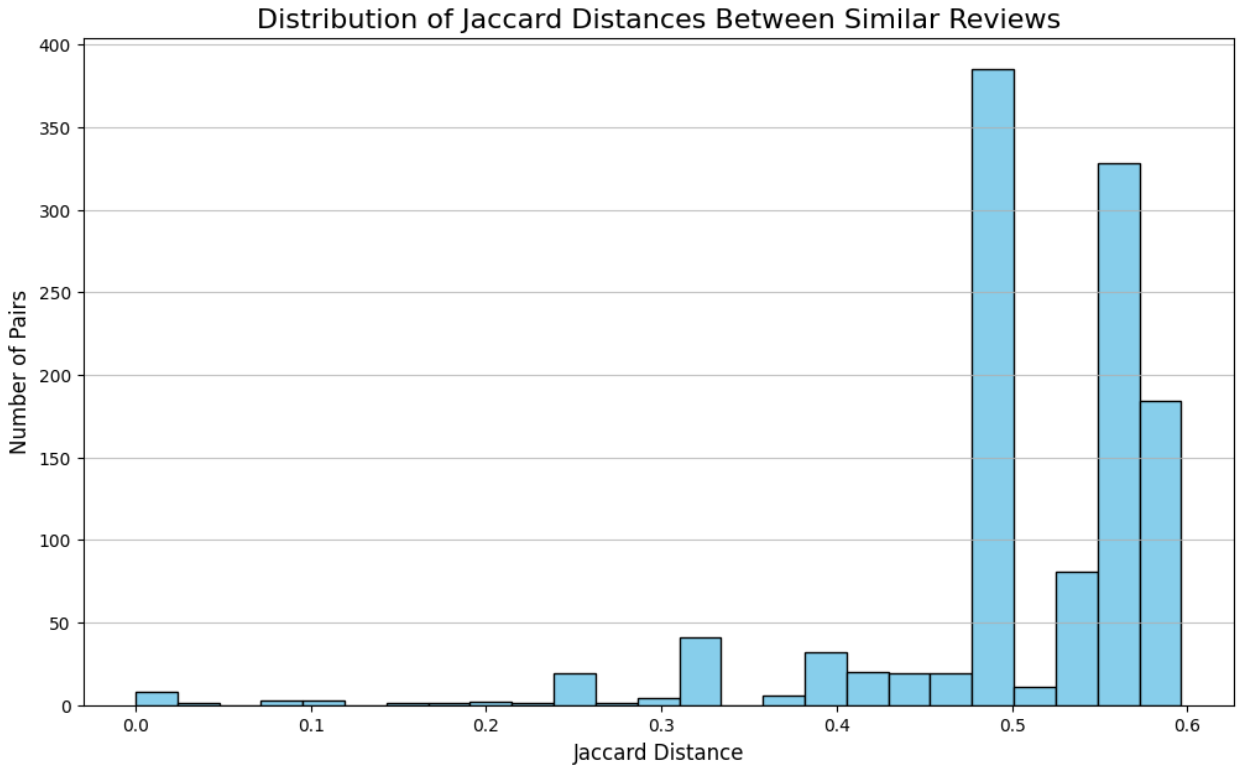

Figure 2: Distribution of Final Filtered Pairs

Figure 2 histogram shows the distribution for the **1,170** final pairs after the advanced filtering logic has been applied. The most important change is the dramatic decrease of the spike at JaccardDistance = 0.0. This confirms that our filtration method is highly effective at deleting the uninteresting, identical duplicates. The rest of the distribution is more useful, showing a logical concentration of pairs with low Jaccard Distance (high similarity) and an increase towards the 0.6 Jaccard Distance. This ensures that the final set is focusing on the pairs that are really similar but not uselessly identical.

7.3. Effectiveness of Filtering

The filtering method designed proved highly effective. The "subset title" filter successfully deleted the pairs like ("crime and punishment penguin classics", "crime and punishment") when

the reviews were identical for different versions of the same book. The "same user" filter correctly removed the results where a user had posted the same review twice for many books marking it as a spam. These steps were very important for reducing the noise ratio of the final output, as confirmed by the comparison of the two histograms.

8. Conclusion

8.1. Summary of Findings

The project successfully designed and implemented a scalable pipeline to find similar book reviews using Apache Spark and Locality Sensitive Hashing. The methodology correctly handled the computational challenges of the large-scale text similarity task. The final results represent that a mixture of text pre-processing, the MinHash LSH algorithm, and intelligent post-processing filters does effectively identifies meaningful pairs of similar reviews. The project is replicable and also scalable, giving a solid foundation for any further analysis.