Algorithms for Massive Data Project

Professor Malchiodi

Project 1: Finding similar items

Hossein Yousefian

31793A (DSE)

Introduction

The goal of this project was to develop a detector capable of identifying similar reviews within the Amazon Books review dataset. Since the reviews are primarily textual, the detector focuses on finding similar pairs based on the content of the text. To measure similarity between review pairs, I approximated **Jaccard similarity** using **MinHash**, and used **Locality-Sensitive Hashing (LSH)** to search for similar reviews. The detector is written in a way that can be used for similar datasets when the task is finding similar textual data.

To ensure scalability and efficiency, the solution was implemented using **Apache Spark**, which enables parallel and distributed processing of large datasets. Rather than coding the entire pipeline from scratch, Spark's high-level APIs and built-in tools were utilized to leverage its robust performance features.

A dedicated class, **TextSimilarityDetector**, was developed to address the problem of textual similarity detection. This class includes multiple methods that support various customization options, allowing users to customize the analysis according to their needs. While some methods—such as column renaming—were not strictly necessary, they were included to enhance flexibility. The implementation also makes use of caching to improve performance during repeated operations.

Jaccard Similarity

Jaccard similarity is a widely used metric for measuring the similarity between two sets. It is defined as the size of the intersection divided by the size of the union of the sets:

$$\text{Jaccard}(A, B) = |A \cap B| / |A \cup B|$$

This metric ranges from 0 (no overlap) to 1 (identical sets) and is particularly effective in applications involving document similarity.

Text Preprocessing and Similarity Estimation

To derive meaningful similarity measures from text data, several preprocessing and modeling steps are applied:

- **Tokenization:** The process of breaking down text into smaller units (tokens), typically words. Regular expression-based tokenizers allow customization for excluding punctuation or specific patterns.
- **Stop Word Removal:** High-frequency, low-value words (e.g., "the", "and", "is") are filtered out to reduce noise and enhance the relevance of the remaining content.
- **Feature Extraction via Hashing:** To compute similarities efficiently, text must be converted into numerical representations. One common approach is the hashing trick, which maps each token into a fixed-size vector using a hash function, resulting in a sparse representation.
- **MinHash and Locality-Sensitive Hashing (LSH):** Exact computation of Jaccard similarity can be expensive on large datasets. MinHash is a probabilistic method that approximates Jaccard similarity by using a limited number of hash functions. LSH groups similar items into the same buckets, enabling fast, approximate nearest-neighbor searches in sublinear time.
- **Parallelism and Sampling:** For large-scale processing, tasks are distributed across multiple computing units. Spark's default parallelism helps optimize this distribution. Sampling techniques, such as using 1% of the dataset, can be used during early analysis phases to reduce computational cost.

Spark-Based Optimizations and Execution Strategy

This project relied on Spark DataFrames and their underlying optimization mechanisms to efficiently process the review dataset at scale. Key features of Spark that are relevant here are:

- **Catalyst Optimizer:** Spark's query optimizer analyzes and rewrites DataFrame transformations into more efficient execution plans, reducing runtime without requiring manual tuning.
- **Lazy Evaluation:** DataFrame transformations are not executed immediately. Instead,

Spark builds a logical execution plan and optimizes it before running any action, improving overall performance.

- **Partitioning:** Spark automatically divides large datasets into smaller partitions, which are processed in parallel across the cluster. This division is very useful for scalability and effective utilization of cluster resources.
- **Caching:** Frequently reused results are cached in memory to avoid recomputation, speeding up iterative algorithms and repeated queries.

By utilizing these features, the solution scales effectively to large datasets while maintaining manageable development and execution complexity.

Implementation

This project uses a class-based design for modular and scalable similarity detection. The core component is the **TextSimilarityDetecto**r class, which encapsulates all necessary data loading, preprocessing, and similarity computation logic using Apache Spark.

Class Initialization (__init__ method)

The class TextSimilarityDetector is initialized with several parameters:

1. **csv_file_path:** Path to the input CSV file containing the text.
2. **text_input_column_name:** The name of the column containing the target textual data.
3. **columns_to_select:** A list of specific columns to load from the dataset.
4. **column_names_mapping:** A dictionary to rename columns after loading for consistency.
5. **sample_fraction:** A float value used to sample a fraction of the dataset.

These parameters allow flexible configuration of input data, selective loading, and renaming, which are useful for working with large datasets.

The class uses .cache() at multiple points (on the full dataset, sampled dataset, tokenized data, and hashed features) to store intermediate results in memory and avoid recomputation across operations. This is especially beneficial when working with iterative or expensive transformations in Spark.

Also it's worth noting that this calls throws errors early in the execution if the user input is faulty. This prevents failing in the middle of processes.

Main Method: find_similar_pairs()

This method is used to compute text similarity using MinHash and LSH (Locality-Sensitive Hashing). It takes several parameters:

1. **record_id_column_name**: Name of the column that uniquely identifies each record.
2. **pattern**: Regular expression pattern used for tokenization (default is \W for splitting on non-word characters).
3. **remove_stopwords**: Boolean indicating whether to remove common stop words.
4. **num_features**: Number of features for hashing (controls the dimensionality of feature vectors).
5. **num_hash_tables**: Number of hash tables used in MinHashLSH (higher = more accurate but slower).
6. **Jaccard_threshold**: Maximum Jaccard distance for considering two reviews as similar (smaller = more strict).

Function Calls

Both the constructor (__init__) and find_similar_pairs() call several helper methods:

1. **_load_data():** Loads the CSV file into a Spark DataFrame, optionally selects and renames columns, drops duplicates and nulls, and caches the result.
2. **_sample_data():** If sample_fraction is specified, it samples the data and caches it.
3. **_tokenize():** Applies a RegexTokenizer to split text into words and optionally removes stop words.
4. **_remove_stopwords():** Uses StopWordsRemover to filter out high-frequency common words.
5. Inside **find_similar_pairs()**, a HashingTF object transforms tokenized text into hashed feature vectors, and a MinHashLSH model is fitted on these vectors to compute pairwise similarity.

What This Enables

This implementation allows scalable, parallel processing of large-scale text data to identify similar reviews using approximate set similarity measures. By leveraging Spark's distributed architecture, it can efficiently handle datasets that are too large for traditional single-machine approaches.

Returned Object

The .find_similar_pairs() function returns a Spark DataFrame (similar_pairs) containing pairs of reviews whose Jaccard distance is below the specified threshold. This cached DataFrame includes:

1. The Jaccard distance between each pair,
2. The full record details from both matched reviews (denoted as datasetA and datasetB).

This cached returned object enables us to further investigate these pairs (with queries) without the need to fully execute code again.

Final Queries for Investigation

The last section of the code uses queries on the similar_pairs DataFrame to explore specific subsets of results:

1. Top 10 most similar pairs: Sorted by lowest Jaccard distance.
2. Side-by-side comparison: Shows metadata and full texts for both reviews in each pair.
3. Identical reviews: Filters results with Jaccard distance = 0.
4. Moderately similar reviews: Filters for Jaccard distance > 0.2.
5. Different users: Filters pairs where the users are not the same.
6. Different book titles: Filters reviews referring to different books.

Conclusion

The **TextSimilarityDetecto**r enabled us to find similar pairs of reviews and investigate why these pairs are similar. Some of the observations were:

1. Out of 24359 sampled reviews, we found 558 similar review pairs which had Jaccard distance of less than 0.6 from each other.
2. A lot of these pairs had 0 Jaccard distance from each other and most of them were identical except for review_id field.
3. Some of the other pairs that had 0 Jaccard distance were also identical but the difference between them was book_title (they were registered by the same user). An example of this is the pair with the book titles of "To kill a mockingbird" and "To Kill A Mockingbird".
4. Pairs that had 0.2 < Jaccard distance < 0.6 were identified and these were the pairs that were not identical but they had similar review text and they can be used for further tuning of the model.

Declaration of Originality

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.

Resources

Mining of Massive Datasets, written by A. Rajaraman and J. Ullman.