Algorithms for Massive Data

Final Project

"Finding Similar Items"

Using the MeDAL Dataset

Ashkan Samavatian

Matriculation number: 965235

(Student in Data Science and Economics)

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. I understand that plagiarism, collusion, and copying are grave and serious offenses 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. "

1) Abstract

The current project is focused on analyzing the "MeDAL" dataset, which is available on Kaggle and released under the Apache 2.0 license, to find textual similarities. Considering the "TEXT" column in a random subset of the "full-data.csv" file and implementing the codes in "PySpark" on the "Google Colab" service, the methodology contains text preprocessing, tokenization, and the exclusion of the stop words.

The main process of similarity detection was performed by using the "Locality Sensitive Hashing (LSH)" technique to estimate "Jaccard similarities". With a defined similarity threshold, several text pairs demonstrating significant similarity were identified as the results.

2) Introduction

Medical **D**ataset for **A**bbreviation Disambiguation for Natural **L**anguage Understanding (MeDAL) is a large medical text dataset curated for abbreviation disambiguation, designed for natural language understanding pre-training in the medical domain. It was published at the "ClinicalNLP" workshop at "EMNLP".

The aim of this project is to analyze the "MeDAL" dataset, which is available on Kaggle and released under the Apache 2.0 license, to extract those pairs of texts from the dataset that are similar to each other based on "Jaccard similarity" using the "Locality Sensitive Hashing (LSH)" technique.

3) Dataset Overview

The "MeDAL" dataset is a large medical text dataset curated for abbreviation disambiguation. It consists of three different columns: "TEXT", "LOCATION", and "LABEL" with 14,393,619 entries. For this project, I considered the "TEXT" column in the "full_data.csv" file (The CSV file of the MeDAL), which contains the related articles for abbreviation disambiguation for natural language understanding to find similar pairs.

4) Methodology

4-1) Data Acquisition

I have used my "Kaggle API" ("Kaggle username" and "Kaggle key") to download the dataset directly from Kaggle to the Google Colab service. The dataset's identifier is "xhlulu/medal-emnlp", and it was downloaded in a compressed format. Then I unzipped only the "full_data.csv" file for the upcoming process.

4-2) Data Handling

After unzipping the "full_data.csv" file and storing it in the "MeDAL_df" variable, I used a code to select a random subset with a desired size for the purpose of my analysis. I decided to work with a representative sample for analysis without compromising computational efficiency. Thus, different subsets with different sizes (e.g.: 100,000, 200,000, 500,000, and 700,000 records) were applied for the project.

After extracting the random subset, I tried to reset all the indexes in order not to have any computational issues for further processes. Then I monitored the subset and saved it in CSV format.

4-3) Computational Environment and Tools Setup

The analysis was performed in Google Colab, using PySpark due to its scalability and suitability for processing massive data. To work with PySpark, it was necessary to establish a Spark environment, including the installation of Java, downloading Spark, extracting its files, and setting up findspark to help locate and initialize Spark on Colab.

After Setting the environment paths and initializing findspark, the vital step was importing the necessary libraries to work with PySpark. To finalize the PySpark setup, I started a PySpark session with a reasonable capacity for my project.

4-4) Data Pre-Processing Techniques

After starting the PySpark Session, the subset data was loaded into PySpark and saved in a variable named "df" to start the project. In the first step, I decided to apply some preprocessing on the "TEXT" column to have normalized data. Hence, I performed the preprocessing techniques below:

4-4-1) Converting the "TEXT" column to lowercase: To have more uniformity in the data.

4-4-2) Removing the punctuations: Ensuring not to have any characters except the alphabetic ones, to increase the main process efficiency.

4-4-3) Word Tokenization of the "TEXT" column: To make the data ready for removing the stop words.

4-4-4) Removing the stop words: To make lighter text documents for the LSH technique.

4-4-5) Concatenating words: To prepare the filtered text data as arrays for the shingling process.

After these steps, I monitored and checked the preprocessed subset ("df") for the last time before the main process of the LSH technique.

4-5) Implementing the Jaccard Similarity using LSH

As we have learned to use the LSH technique for implementing the Jaccard Similarity on massive data, I used this technique for my project since the "MeDAL" dataset (and even its subsets) is considered massive data.

Locality-Sensitive Hashing (LSH) is a method for reducing the dimensionality of high-dimensional data. The idea behind LSH is to hash input items in a way that similar items map to the same buckets with high probability. By doing this, it is possible to reduce the number of comparisons required to find similar items.

For applying the Jaccard Similarity with the LSH technique, the main steps are:

- ✓ Shingling
- ✓ Minhashing
- ✓ LSH

4-5-1) Shingling: In this step, documents were converted to sets of shingles with a fixed length. (k-grams). I applied Shingling to the data with "k=10" for my project and I defined a function in Python that takes strings and returns a list of shingles with the length of "k=10". Since I decided to perform my project in PySpark, I had to define a user-defined function (UDF) to prepare my function for the upcoming step, which was returning the shingles in the form of string arrays for the "Minhashing" action.

At the end of the shingling step, a column named "shingles" containing the shingled substrings, was ready in the "df_shingled" for the Minhashing process.

4-5-2) Minhashing: For this step first, I had to generate the hash features for the shingles. I applied the "CountVectorizer" method to vectorize the shingles into a numerical format suitable for the Minhashing process.

After obtaining the vectorized representations of the shingles, I had to check two important things in the "features" column:

First, I had to be sure that I didn't have any null values in the "features" column.

Second, I had to be sure that all vectors in the "features" column had at least one non-zero value.

It was important to do this two-step filtering process to refine the subset because if I had null values in the "features" column or if I had some vectors with all zero values, I couldn't do the whole project without receiving errors. Thus, to enhance computational efficiency for massive data like this one in my project, I filtered out the null values and the vectors with all zero values from the "features" column.

After this filtering process, I defined an ordering pattern by using the "Window" module in "PySpark's SQL" library to assign a unique and increasing zero-based "id" to each row in the vectorized data frame for computational efficiency and convenience reasons.

4-5-3) LSH: After the previous step was completed, the "MinHashLSH" from "PySpark's MLlib" was applied to implement the "MinHash Locality-Sensitive Hashing" method. For this process, the "features" was

considered as the input column, and the "hashes" column was the result. I considered "numHashTables=5" for the bands in the LSH technique on this project.

In the next step, I chose the similarity "threshold=0.8". So, the rows with Jaccard Similarity above this threshold were considered "similar".

$$\textbf{Jaccard (S , T)} = |S \cap T| / |S \cup T|$$

After assigning the threshold, the "df_vectorized" data frame was compared to itself to find similar row pairs. Then I applied a filter code to prevent having results like (A , B) and (B , A) or (A , A), (B , B) to make the whole process more agile and efficient. Finally, only the "id" columns of the similar row pairs were selected and gathered as a list in the "results". This list contains the ids of similar row pairs from "df_vectorized".

4-6) Results

After applying the "MinHashLSH" technique, several pairs of texts from the dataset were identified as similar based on the established threshold, and then in the final step, these pairs were printed out in the readable format "(id1 , id2)" for review.

5) scalability of the solution

The proposed solution in this project has the capability to scale up with an expanding dataset. PySpark is specifically designed to handle big data operations using distributed computing. This means that as the dataset grows, the workload can be distributed across multiple nodes in a cluster, allowing simultaneous processing and thereby reducing the time for completion.

In the current project, different subsets with different sizes were examined by using PySpark on Google Colab, and the required times of doing the project with respect to the size of the subsets were achieved. These time durations and the corresponding size of the subsets are presented in the table below:

Size of the subset	Project time duration (approximately)
100,000 records	15 minutes
200,000 records	40 minutes
500,000 records	1 hour and 30 minutes
700,000 records	3 hours and 15 minutes

Table1: Project time duration with respect to the size of the subset

If the dataset size multiplies, we have the option of tuning or modifying one or all the parameters that are used in the LSH technique (k for k-gram shingling, number of hash tables, and the similarity threshold) to ensure computational efficiency. However, it is necessary to be cautious as these modifications might make

some changes in the recall of the similarity search. The optimal number of these parameters depends on different issues in a project and how much someone is willing to potentially sacrifice recall for speed. It is a trade-off between accuracy and speed and the matter of encountering the False-Negative and False-Positive errors should be noted.

6) References

6-1) https://www.kaggle.com/datasets/xhlulu/medal-emnlp

6-2) Zhi Wen, Xing Han Lu, Siva Reddy. MeDAL: Medical Abbreviation Disambiguation Dataset for Natural Language Understanding Pretraining

6-3) Mining of Massive Datasets by: Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman

6-4) https://spark.apache.org/docs/3.2.0/ml-features.html

6-5) https://en.wikipedia.org/wiki/MinHash

6-6) https://en.wikipedia.org/wiki/Locality-sensitive_hashing

6-7) https://www.kaggle.com/discussions/general/51898

6-8) https://sparkbyexamples.com/pyspark-tutorial/

6-9) https://www.codemotion.com/magazine/backend/fast-document-similarity-in-python-minhashlsh/