Finding similar items in the Yelp dataset of reviews

Algorithms for Massive Datasets

Samuele Bompani

July 2023

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 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.

1 Introduction

In the realm of natural language processing, the task of identifying similar content holds an important role. Whether it be for recommendation systems, sentiment analysis, or fraud detection, understanding the similarity between textual data is crucial. With this in mind, my project tries to implement a custom detector for pairs of similar reviews, focusing on the text field of the dataset. The dataset used for this project is the *Yelp Dataset* available on Kaggle[1] updated on March 2022 and downloaded on the 29^{th} of June 2023. The code was executed on *Google Colab* and was developed in *Python* using *Spark*

2 Pre-processing

To efficiently identify lexically similar documents, the most effective approach is to represent them as sets by extracting the set of short strings found within each document. By doing this, documents that contain even small pieces like sentences or phrases will share numerous common elements in their sets, regardless of the order in which those pieces appear. For this project I used a technique called k-shingles, applied after the removal of the so called stop words.

[1] https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset

2.1 Stop words

Stop words are words such as "the," "and," "is," "in," etc., that appear frequently in a language but often carry little meaning or contribute to the understanding of the text. Removing them can reduce the noise in the text data and it improves the efficiency of the process. After removing punctuation and whitespaces and creating a list of words, the process filters stop words and remove them from that list.

2.2 k-shingles

The lists are then joined adding a single whitespace, creating a *new* text, which is divided in k-shingles: consecutive sequences of k terms characters extracted from the documents. By representing a document as a set of k-shingles, common sequences of k terms are captured, allowing for efficient comparison and identification of document similarity.

The choice of the value of k is crucial, because it determines the length of the shingles and affects the granularity of the representation. Since the length of the reviews is similar to the length of email, I decided to use $k = 5$, as it is suggested in the book *Mining of Massive Datasets*.[2]

3 Minhash

Minhash, short for "minimum hashing," is a technique used to estimate the similarity between sets or documents. It works by representing sets as fixed-length signatures and comparing these signatures to determine similarity. The process involves selecting a set of hash functions and applying them to the elements of the sets. The minimum hash value across all hash functions is chosen as the signature for that element.

To estimate similarity, the signatures of two sets are compared. The fraction of matching hash values indicates their similarity. Minhash is based on the idea that the probability of two sets having the same minimum hash value is assimilable to their Jaccard similarity.

Minhash provides a compact representation of sets while preserving similarity information, enabling fast similarity estimation without storing or comparing the original sets directly.

3.1 Hash functions

A crucial parameter to be decided is the number of hash function. After some experiments and according to *Mining of Massive Datasets* suggestions, the number used was 100.

Functions are created from the generic function $(ax + b)\ \%\ n$: a and b are two pseudorandom numbers, extracted with the function *randint* of the *random*

[2]Leskovec, J., Rajaraman, A., Ullman, J. D. (2020). Mining of massive data sets. Cambridge university press. section 3.2.2

library, while n is the total number of distinct tokens. Since n might not be prime, some collisions are possible, but the number of them is negligible for the purpose of this project.

3.2 Implementation

I implemented the Minhash algorithm in a trivial fashion, creating the hash functions and calling an aggregate function on the tuples ($shingle$, $list\_of\_documents$). The result is a dictionary with documents as keys and the vectors of the minimum hash value, for each hash function, as values. *Spark aggregate* function guarantees a distributed computation that can improve the performance significantly, depending on the architecture.

4 LSH

While minhashing allows us to condense extensive documents into compact signatures, maintaining their expected similarity, it can be a challenge to efficiently identifying the pairs with the highest similarity. This problem arises from the potentially vast number of document pairs, even when the total number of documents itself is not excessively high. Locality Sensitive Hashing, or LSH, is the technique used in this project, to reduce the number of pair candidates.

The idea is to hash items multiple times, favoring similar items to be hashed to the same bucket more often than dissimilar ones. Pairs that hash to the same bucket in any of the hashings are considered $andidatepairs$, and their similarity is calculated. The goal is to minimize dissimilar pairs hashing to the same bucket (false positives) while maximizing truly similar pairs hashing together (false negatives).

When employing minhash signatures, an effective method is to divide the signature matrix into b bands of r rows and use a hash function to map vectors of integers (columns within each band) to numerous buckets. Although the same hash function can be used across all bands, separate bucket arrays per band ensure that columns with the same vector in different bands won't hash to the same bucket. This enhances the overall efficiency of the LSH technique.

As in the previous algorithm, the main function is an *aggregate*, that creates the buckets as a list of b dictionaries. After that *candidate pairs* are extracted, keeping the elements of the dictionaries with more than on document. Finally the similarity is calculated and the threshold is applied, returning the pairs of similar items.

4.1 Parameters choice

The biggest challenge to implement LSH is the choice of the parameters, to balance efficiency and accuracy. Since the main goal was to keep the whole execution time around one hour, a good compromise was found dividing the

# Hash functions	10	100
# Bands	2	20
Mean error	0.0571	0.0235
Mean Jaccard sim.	0.9391	0.8869
Correctness (error < 0.01)	42.86%	46.67%
Correctness (error < 0.1)	76.19%	96.67%
Time $\sim$	7.45 m	50.06 m

Table 1: Accuracy and performace results

signature matrix in 20 bands of r rows each, with $r = \frac{\#hash}{20} = 5$. The resulting threshold is $(1/b)^{\frac{1}{r}} \approx 0.5493$.

5 Results

To evaluate the implementation of the algorithms I proposed, there are three aspects to be considered: accuracy, efficiency and scalability.

Since the algorithm is based on estimations of Jaccard similarity, I evaluated accuracy calculating the similarity for each pair of reviews returned. The error taken in consideration is the absolute difference between the Jaccard similarity of two tokenized reviews and the estimation of it. As it is shown in Table 1, increasing the number of hash functions, and consequently the number of bands and rows, produces significantly better results.

This improvement, though, is expensive in terms of efficiency. While the experiment conducted with 10 hash functions terminates in less than 10 minutes, the second one takes around an hour. This is mainly due to the high number of hash functions and could be improved with some optimizations of the modulo operation, but this would be out of the scope of this project. Given that, considered the large amount of data and the limitated power of Google Colab (only two cores are available) the results are satisfying.

Regarding scalability, the use of *Spark* for large computations, in particular to distribute the application of the hash functions, guarantees good performance with an higher number of entries, although this cannot be proven with the architecture that I used.