Finding similar items

Nicolas Audoux

May 2023

Algorithms for massive datasets
Prof. Dario Malchiodi
a.a. 2022 - 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.

Contents

1 Introduction

For this project, I'm going to implement an algorithm to find similar items in the *yelp dataset* published on Kaggle. My project is based on the version 4 of the dataset and I will only use the *yelp_academic_dataset_review.json* file.

2 Data Organization

In the *yelp_academic_dataset_review.json* file, we can find reviews about businesses made by different users. A review is defined by an id (review_id), a comment (text), a date (date), the business it is related to (business_id), the user who wrote it (user_id) and four grades:

- general appreciation (stars)
- usefulness of the comment (useful)
- if it's funny (funny)
- if it is seen as cool by others (cool).

Since my goal is only to find similar items based on the text of the review, I will only use the text for my algorithm.

3 Pre-processing data

In order to avoid false-negative because of the capital letters, I decided to make all texts in lowercases and I remove every punctuation leaving only white-spaces. Thanks to that I have uniform texts which make the process to find similar items a little bit easier.

4 Algorithm

In order to find similar items, I decided to use the k-shingle method. This method consist in getting all k-letters sequence (k-shingles) in each text, comparing them to a set which include all k-shingles of all texts and finally, comparing which texts contains the same k-shingles. I will describe those steps in order

4.1 K-shingle Creation

The first step of the algorithm is to take all texts already pre-processed and get a set of k-shingles for all of them. To create the k-shingles set of a text, I made a function taking the text, and k as parameters. This function run over the text and will add every k-shingle in a list. I make sure that there are no duplicates in the list. As I add those k-shingles in a list.

I also add them to another list representing my universal set. In order to save some space, I hashed all k-shingles to four-bytes values.

4.2 Signatures

To reduce the size of those lists, I will compute their signatures. To do so I start by mix all the line of my universal set using a hash function to reduce the computation time. I then separate my universal set mixed into m subsets.

For each subset, I will compute a partial signature for all my sets . This partial signature will be composed by only one number for each set.

This number represent the index of the k-shingle with the lowest index of my subset which is part of the set I'm looking at.

Once I've done that for all my sets with all my subsets, I will have a signature of size m for all my sets.

I already reduced a lot the complexity of finding similar items but I can't compute a distance between each comment I took in the first place. That's why I need to reduce the number of comparison I will do.

4.3 Finding potential candidates

In order to find potential similar items, I will divide all the signatures I have into b bands. Then, for all bands, I will perform a hash function. If two signatures are mapped to the same bucket for the same band, they become candidate for being similar.

To find the number of bands, I chose t a threshold which define the lowest similarity I expect to have for two similar items. With this threshold I can find b and r which is the number of row in each band such as

$$(1/b)^{1/r} \approx t$$

4.4 Computing the distance

Now that I have my potential candidates, I will compute their distance, and if their distance is above the minimum t I chose then I consider them being similar. if not, it means that I had a false positive. The distance I chose is the Jaccard Distance. To compute it, I take the signatures of my two candidates S1 and S2 , and I compute the distance named sim(S1,S2)

$$sim(S_1, S_2) = \frac{S_1 \cap S_2}{S_1 \cup S_2}$$

5 Scaling up

To scale up this solution, I have to adapt my code so that it can run on a distributed system. Luckily, it doesn't required to change a lot of things given the fact I have already try to reduce both time and space complexity.

Most of the map-reduced functions I have to use will be easy to implement because their either used to simply split sets or using functions I have already used in my project. Let's go again through all the algorithm considering a distributed system.

5.1 getting k-shingles

First thing I have to do is to map each comment of my dataset to a list of shingles describing it. To do so I can use an simple map-function reduce. For each comment, I am creating as many key value pairs as I have k-shingle in the comment such as the key correspond to the comment's number and the value to the hash of the shingle. In the reduce phase, I just aggregate all of those hashes based on the comment's number dropping every duplicates of a shingle. I then have for each comment, a list of shingle associated. To build my universal set, I can perform another map function on each of the key value pair I just got. For each list of shingles, I generate a list of key value pair such as the key is *UniversalSet* and the value is the hash of a shingle. I then aggregate all of this key value pair, dropping duplicate hashes and I have my universal set.

5.2 Computing the signatures

To compute signatures, I first split my universal set in m random subset. To do so, I perform another map-reduce operation over my universal set. I begin by creating a key value pair for each shingle in my set. The key will be equal to the hash modulo m, the number of subset, and the value will be the hash. I then gather all value with the same key in the reduce function. To compute the signature, I perform another hash function on each list of hash representing a comment. In this map function, the key is the index of the subset of the universal set I am using and the value is the result of the partial signature computed with this subset. I then use a reducer in this map function to build the full signature such as the key of a pair become the index of the value in the signature.

5.3 Getting similar items

Finally, as I have already done for splitting the universal set, I will splitting the signatures in bands and for each band I will split them again using the hash of the band as a key, and the comment's number as a value. I can finally have here my potential candidates gather in the same bucket. I just have to compute their distance to have my final output.

6 Experiment

For this experiment, I searched for similar comments in the first *10 000* lines of the dataset. For my variables, I decided to use *k=5*, *m=100* and a threshold *t=0.6*. After computing signatures of all my documents and searching potential candidates, I arrived to a list of more than *2 000* potential pair candidates. Which is not a lot comparing to the number of comparisons I could have done with this number of documents. Finally after filtering only pairs with a similarity above 0.6, I only found two pairs. Documents with the highest similarity are comment number 1715 and 9839 with a similarity around 0.85. When we look at the comment's we can see that those two comments have a lot in common, they both begin by the same phrase and they differ only at the end. So this result is coherent. We can observe the same pattern for the second result which is similarity around 0.7 for the comments 6523 and 7971.

7 Results

To conclude, during this experience even with techniques to reduce the number of comparisons I have to do to find similar documents, I've got a lot of false positive (more than 8000 pairs).

This can be explained by the fact that all text don't have the same length. It can be a problem when I have to choose a number of subset for my universal set. If this number is too high I will have a lot of signatures with values equals to infinity and those signatures will probably be hash to the same bucket becoming a candidate. That is why, when I'm computing the Jaccard similarity, I don't consider equals two value equals to infinity. In the other hand if I choose to divide my universal set into too few subsets, my signatures won't be useful at all and I won't get a good result.

The fact that I don't get a lot of similar documents can be explained by the fact that comments on a business are very different one from the other depending on the general appreciation, the style of the author and of course the business criticized.

But, despite all this issues, It's not impossible to find truly similar items, it only take a little bit longer to remove all false positive.