Similar Pairs with PySpark

Stefano Clemente

September 7, 2023

1 Introduction

Our goal was to implement from scratch a detector of pairs of similar tweets (more precisely, texual reviews).
For this purpose, we have employed a combination of two popular methods: the **MinHash procedure** coupled with a **Locality Sensitive Hashing** (LSH) technique. To ensure that the code is scalable, our detector is built on the Apache Spark engine; more precisely, we rely on its Python API (PySpark).

2 Technologies & data sources

Python 3.11, Spark 3.4.1, PySpark 3.4.1.
The dataset is updated to September 2023.

3 Theoretical approach

3.1 Why MinHash

The MinHash procedure needs no introduction. In its basic form, it's a popular method for calculating the similarity (in the sense of Jaccard) between objects.

First, assuming we are provided with both the documents and their content (in our case, each document contains a certain number of strings, which we call **items**), we start with arranging our data in a so called **characteristic matrix** M, a binary matrix where we have documents as columns and items as rows: the value of M[i][j] is 1 if the item corresponding to the i-th row belongs to the j-th documents; 0 otherwise. Then, we if consider a permutation of those rows applied on the characteristic matrix, we define the MinHash value of a document (namely, a column) as the index of the row which contains *the first non-zero element* for that document, starting from the top. Of course, the MinHash value will depend on the permutation chosen.

If we repeat this process several times, say N, we will obtain N MinHash values for each document - that is, each document will be identified by this collection of values, which it is known as **MinHash signature**.

We may represent the documents and their corresponding signature in a matrix, called **signature matrix**, which is way more concise than the characteristic matrix - yet it conveys the same amount of information: it is a well-known result that the probability for two documents to share the same MinHash value (for a given permutation) equals their Jaccard similarity; therefore, if we think of the MinHash signature as several independent ”experiments” stacked together, the number of rows of the signature matrix in which two columns agree will give us a very good approximation of their similarity (after all, its expecting value IS the Jaccard similarity!).

This approach works very well with small-sized databases, but it's not scalable with respect to the size/number of our documents. In fact, there's no guarantee that we are able to build and store the characteristic matrix if we are working with big data. Even a ”simple” permutation over a huge number of rows could pose an insurmountable problem. As always when dealing with big datasets, we need a way more efficient method to compute the MinHash signature.

A classical way (which we have indeed adopted) to address this problem is to simulate our permutation with randomly selected hash functions. Those functions must map a range of integers into the same range of buckets, thus effectively replicating a permutation. Having picked a certain number of those hash function, we calculate the signature of our documents by applying every hash function to each one of its items, and selecting the mininum value obtained (which is our MinHash!).

Finally, with our signature matrix in hand, we possess a quick way to calculate the Jaccard similarity between two pairs of document. It is quick enough?

3.2 Why LSH

Albeit fairly accurate even with a reasonably small number of hash function, the MinHash procedure alone would still require us to perform a lot of useless calculations. Having no *a priori* information on how similar our items are, we would be forced to calculate the Jaccard similarity for each pair: even if we plan to select the best results out of our computation, we would still have to take into account *a lot* of meaningless data, since most of the pairs will be discarded - most items are expected to be different, after all.

The basic idea of LSH is that we can pre-select a list of promising candidates by splitting the minhash signature into "bands" (i.e., portions of our signature) and choose the pairs whose signature completely agree within that band (imagine hosting a bunch of qualifying tournaments, the winners of which will be granted the right to compete for the special prize, which is the true similarity!). To check whether two items agree band-wise, we simply hash the corresponding slice of the signature and compare the results. In the end, we collect our candidates and explicitly compute their Jaccard similarity, discarding all those couples that doesn't surpass a chosen threshold.

One final note: the solution we propose combines the LSH step with a "prefiltering" step, in which we restrict our analysis to those pairs that share at least one *ordered*[1] minHash value.

4 Practical Approach and pre-processing techniques

We tried to keep the pre-processing part as simple as possibile.

First, an unique numerical identifier is assigned to each document (to us, documents correspond to rows of the "text" field in the original database). Note that in this way one could effectively retrieve the original review through a simple filter query on the downloaded database. Then, every document is stripped of its punctuation symbols and converted into lower case. At this point, we are able to extract the so-called **shingles**. To us, a shingle is any substring made of three consecutive words. We chose not to remove the stop words in our analysis, for we wanted to maintain generality (and avoid focusing on a single language). The shingles are not stored as strings, but hashed to 32 bit integers - that's completely harmless, and saves us a decent amount of space. As a note of colour, a variation of sha256 was selected as our hash function, but such a degree of sofistication isn't required at all.

For the MinHash calculation, we needed to randomly generate a fixed amount of hash functions. One popular way (which we found to accomodate our needs) to obtain this is by repeatedly extracting at random two values a and b, and then choose

$$h(x) = (a * x + b) \mod c$$

as the hash function, where c is the smallest prime bigger than the greatest value obtainable with a 32-bit hash (i.e. the function we have used to hash our shingles).

By calculating our minHash signature, we obtain something that closely resembles a signature matrix. In our case, the full signature is stored in a single field, each row representing a different document (with different IDs). We will be using this matrix both for the final part and for the LSH part.

For the LSH part, we "explode" our dataset (specifically, our signature, preserving its indices) into a matrix in which every row represents a single minHash for a specified document. By passing a list of indices (which are our bands), we are able to accurately split the MinHash signature: e. g., if W is the length of a single band, our first band will be composed of the first W elements of the signature. The **lsh_banding** function use the built-in **hash** method (again, it is not refined but it does the job) to hash the slice of the signatures, then, thanks to an inner join with the matrix itself, we are able to identify all the pairs of values that agree within that band. In the end, the **build_list_lsh** function stacks together all the candidates found in every band, and removes the duplicates.

[1] by "ordered" we mean that we compare elements having the same index in their correspondent signature

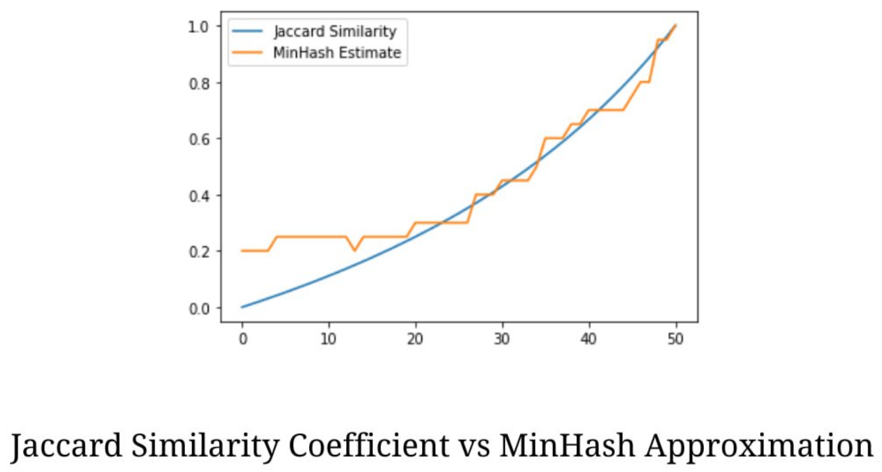


Figure 1: The error rapidly decreases around 20.

In the final part, we move back to our own signature matrix (actually, the exploded one) and perform a grouping in which we collect into a list all the elements that share at least one ordered value of the signature; then, by joining the matrix with itself, we are able to list all the pairs for which the Jaccard similarity is not zero. Note that they are not necessarily the candidates we have found in the LSH part, they just happen to agree for a single MinHash value. It is indeed a pre-filtering step that prevents us from building and storing all the possible couples.

Before computing the Jaccard similarity (which is one heavy computation), we restrict our analysis to those pairs that qualified as candidates in the LSH part.

As a result, we produce a .csv file with all the pairs listed along with their similarity.

5 Experimental results

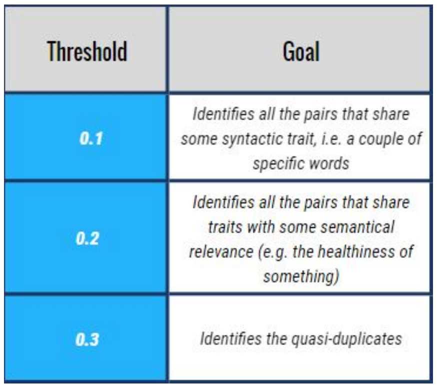
Several test regarding different aspects of our work have been carried out.

It should be pointed out that we needed to reach a compromise between computational time and meaningful results. Furthermore, we were limited to a local Spark session in which we couldn't actually modify the parameters for the memory and the off-heap size (we still have it highlighted in our code, but it produces no actual changes to the environment, since we aren't able to access the configuration files). After several tries, we figured out an upper value for the rows that our environment was able to hold in memory (in the hundreds of thousands), and we consequently selected what we feel is a reasonably-sized subset of the dataset provided (given that the functions we use tend to temporarily enlarge our dataset).

For the **MinHash signature**, we found that a size of 20 perfectly satisfies our requirements. Our results are supported by many independent tests. Figure 1 shows how the size of our signature approximates the Jaccard similarity.

For the **size of our bands**, we felt that the choice is strongly influenced by what kind of research or analysis we are conducting. If our goal is to identify quasi-duplicates, we could increase the size up to the maximum divisor of the signature size (in general, increasing the size of a single band filters out everything but documents that share entire sentences). Since the same results could be obtained by tuning the threshold, we stuck to the smallest non trivial value, i.e. 2.

For the **threshold**, a rule of thumb is to select a value around $(1/b)^{1/r}$, where b is the number of bands and r is their size. In our case, that equals to 0.3. A slightly lower threshold would include more **false positives**, which is not something we are opposed to (in principle). Nonetheless, we have decided to broaden our search and have consequently chosen to display any pairs (and their similarity) that agree in more than a single band. Our findings are summarized in Figure 2.



Threshold	Goal
0.1	*Identifies all the pairs that share some syntactic trait, i.e. a couple of specific words*
0.2	*Identifies all the pairs that share traits with some semantical relevance (e.g. the healthiness of something)*
0.3	*Identifies the quasi-duplicates*

Figure 2: Different thresholds and their reach

6 Scalability

The entire project is built using PySpark and its dataframes (chosen over RDDs), which are designed for distributed computing and ensure that our code scales well (provided that we have access to a reasonable number of parallel resources, our detector could easily handle databases even hundreds of time bigger!). We have mostly resorted to built-in optimized (performance-wise) functions, and the usage of user-designed functions (which are sometimes responsible of hampering the scalability) is limited to the pre-processing part, with little to no impact on the overall performance (there's no shuffling involved in their execution).

7 Possible improvements

7.1 Cache/persist

We have marked in the comments where caching would be appropriate. Since we are dealing with a small portion of the dataset, we have decided to avoid caching: a proper use of the cache/persist function would require a fine tuning of our partitions, which is something we have little to no control over in a local session of Google Colab.

7.2 Smaller partitions when using explode

It's worth to mention that the built-it **explode** function works at its best when we are able to contextually resize our partitions, which is something we avoided in our implementation.

8 References

Our theoretical approach is mostly based on the textbook and the course lectures. Some minor techniques/solutions (e.g. the idea behind the hash functions), even if implemented differently, are possibly found on the internet as common good practices.