Algorithms for Massive Datasets Project

Implementation from Scratch of a Detector of Similar Items

Giovanni Avati

November 2022

Contents

1 Plagiarism 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 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.

2 Dataset

The dataset took into account for the project is a collection of tweets about the Ukraine War. The dataset is organized in csv file, each of them having tweets related to one day. The dataset is available at this *link*.

The last version of the dataset used for the project is the number 280, but the code should be worked fine also with future versions. I mainly worked on tweets from the months of September and October, but it was not mandatory. I only do not recommend using tweets from August 8 or earlier, because as stated by the dataset author: "*I've noticed that I had a spike of Tweets extracted, but they are literally thousands of retweets of a single original tweet*". Therefore, having thousands of copies of the same tweets it is not the best for the purpose of the project.

3 Pre-processing

As previously mentioned, tweets are organized in csv files. There is a lot of information in the csv files for each tweet, such as user id, username, location of the tweet, the number of follower and following, tweet id, text of the tweet, number of retweet, etc.. However, for the purpose of the project the only information needed from each tweet are tweet id, the text of the tweet and the language.
Based on this, I applied three simple steps in order to get the the RDD ready

3.1 Filtering

First of all I filter the tweets by language. I choose to work with English tweets because they were in greater numbers than those of other languages.

3.2 Cleaning

The second step was the one of cleaning the text of each tweets from undesired substring. First, I obviously decided to cut from the tweets all the punctuation and special characters such as line breaks. Other things that I decided to remove

from the tweets are email, emoji and urls (every tweet has the link to itself at the end of the tweet). I only maintained hashtag because they could be meaningful during the process of finding similar tweets.
Moreover, I didn't remove stop words because having k-shingles defined by a stop word followed by other two words could be really useful in finding similarities in texts, especially for news article. So, I decided to use this approach because project case is not so far from finding similarities in news articles.

3.3 Tokenization

After having cleaned up tweets from all undesired objects I tokenized the clean text to get the list of the words that make it up, a more convenient representation.

4 Algorithm Implementation

The algorithm that I decided to implement is a combination of techniques, for instance k-shingles, minhash signatures, Locality-Sensitive Hashing and Jaccard Similarity for sets.

4.1 *K*-Shingles

First step is to get form the list of words of each tweets a set of shingles. I decided to work with shingles build by words, in particular 3 word shingles.

After having collected for each tweet the list of its shingles, I built a collection of all shingles in the corpus of documents, that from now on I will call for simplicity *Bag of Shingles*. The next step was to sort the bag of shingles and to encode it as a list of progressive integers from 0 to n, where n is the length of the bag of shingles. In this way, each shingle is represented as an integer and not as a string and it made easier subsequent steps.

Then, for each tweet instead of going for one-hot encoding its list of shingles I decided to work with an other type of encoding. In particular, each shingle is encoded as the integer representing itself in the *Bag of Shingles*. This way, the representation of the tweet is far smaller, for instance with a tweet with a list of 10 shingles and a bag of Shingles of 5000 shingles, I can store for the tweet a list of 10 integers instead of one of 5000. This is also useful because the integers are not only a mere encoding, but they also represent the position of the shingle in the *Bag of Shingles* array and this speed up operations, like comparison, in the following steps.

A brief schema of how this encoding work is shown in Fig. 1.

4.2 Minhashing and Permutations

After preprocessing and k-shingles operations, the goal is to calculate the *minhash signature* for each tweet, and do this for a number n of permutations of the Bag of Shingles rows.

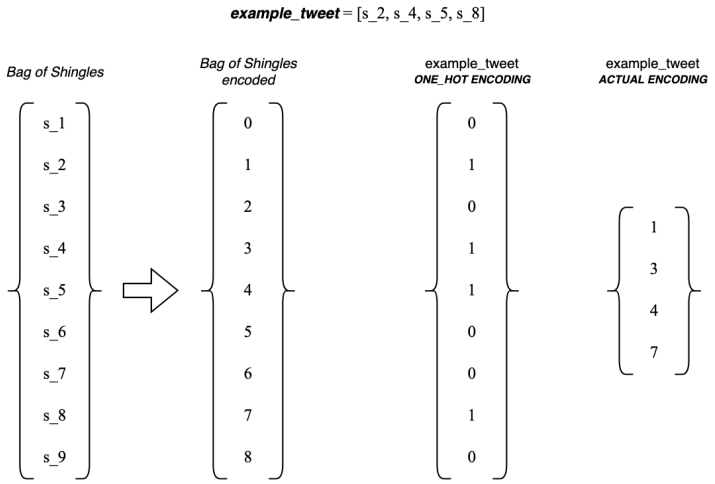

Figure 1: Schematic representation of how tweets have been encoded

To generate a permutation, because shuffling an array is not efficient, I defined a h function to be performed on the Bag of Shingle encoded as progressive integers in order to get a permutation of its rows. The h function is applied to every element of the shingle array of each tweet and it is defined as follow:

$$h(x) = (a \cdot x + b) \mod D$$

where x is the element to be hashed, a and b are two random generated positive integers and D is the dimension of the window within which the numbers must be generated. In this case, D will be the size of the Bag of Shingles array, because hashed numbers have to be in the interval $[0, len(Bag\ of\ Shingles))$. Applying the h function, with the same value of a and b, to each element of the Bag of Shingles guarantee to obtain a permutation of the value in the original array. So, by simply changing n times the values of a and b I can generate n different permutations.

Thanks to the permutation matrix generated as above, it is possible to calculate the minhash signature for each of the permutation. After this process, I have associated to each tweet its own list of n minhash signatures.

4.3 Locality-Sensitive Hashing (LSH)

At this point the RDD will be a tuple composed by the *tweet_id* and the list of *minhash signatures* associated to the tweet. Now, to calculate the *Locality-Sensitive Hashing* we can choose the number b of bands in which we want to divide the n signatures. The number b must be chosen to have $n = b \cdot r$, where n is the number of minhash signatures (thus, of permutation) and r is the number of rows for each bands.

By choosing the number of bands and thus the number of rows for each bands, it is possible to calculate an approximate value for the *threshold* as follow:

$$t = (1/b)^{1/r}$$

This value of *threshold* defines how similar documents have to be in order for them to be regarded as a desired "similar pair".

After having set b, it is possible to divide the list of minhash signatures into bands. To do this, I used a *flatMap* on my RDD in order to obtain rows of the following shape: *((tweet_information, band_information))*, where *tweet_information* is a tuple containing *tweet_id* and the list of all signatures of that tweet (all signatures are maintained because it will be useful in a later step), and *band_information* is also a tuple that contain the *band_id*, which is the number of the band (from 1 to b), and the list of signatures that belong to that band.

Now, the goal is to group the rows, therefore the tweets, that has the same signatures in the same band. Since the fact that list are mutable object, it is not possible to directly perform a *groupByKey* with the list of signatures as key, but there is the need of hashing the list of signatures of the band to transform it into an integer value suitable for being a key. To hash the signatures first I convert the list into a string by simply appending signature values with a blank space in between (from [12, 34, 56] I will have "12 34 56").From the list encoded as string it is now possible to use an hash function to obtain a single integer value.

This value, together with the band id, will be the key on which the *groupByKey* will be performed. So, the RDD for this step is mapped to have as key a tuple composed by the hash and by the band id, and as value the *tweet_information* like above. Now it is possible to perform the *groupByKey* and gather tweets that have the same signatures in the same bands.

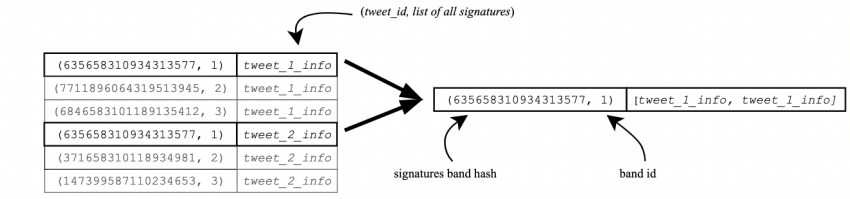

Figure 2: Schematic representation of groupByKey performed over the rdd with the signatures bands hashed

Tweets group together are considered *candidate pair* and in the next steps they will be check to get only real similar pairs.

4.4 Getting similar pairs

From this point, the goal is to get all the candidate pairs obtained by the LSH procedure and retrieve from that only similar tweets. To do that the first step is to apply 3 functions in order to get the rdd ready to compare *candidate pairs* between each others. These 3 functions are:

1. **filter** - To filter out all the bands for which there are no tweets that are similar
2. **flatMap** - To get from the previous representation of rows to a new one in which every tweet has the list of its similar tweets associated
3. **reduceByKey** - To delete all the duplicates that the previous step could have generated

An example of how these 3 steps work is shown by the figure below.

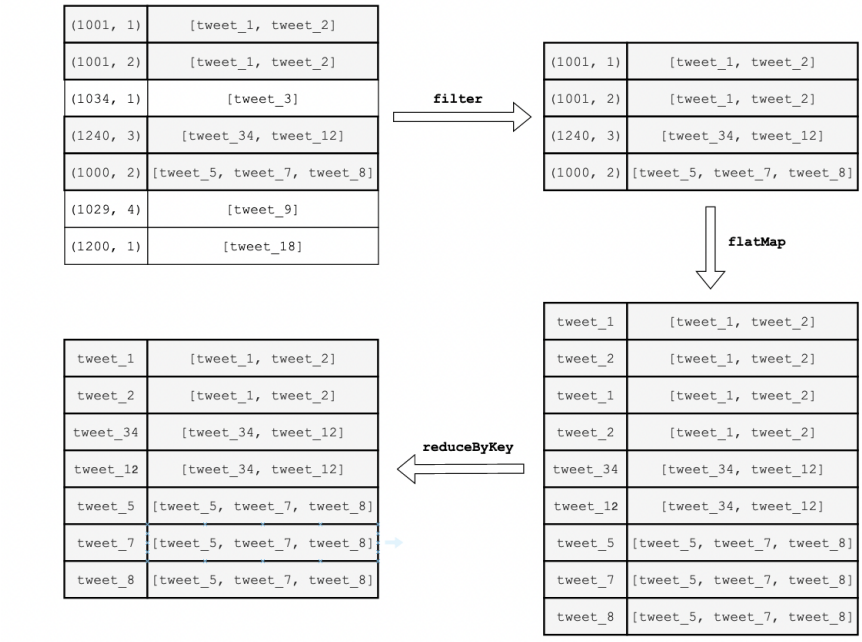

Figure 3: Representation of the 3 steps explained in the 4.4 section

It is now possible to perform an additional *flatMap* to have tweets associated in couples like `((tweet_1_id, tweet_1_signatures), (tweet_2_id, tweet_2_signatures))` where every couple is a candidate pair. From candidates pairs it is easy to calculate the Jaccard similarity of the two sets of signatures. The Jaccard similarity between the two sets of signatures will return a value between 0 and 1 that will be compared with the value t of threshold calculated

in the 4.3 section. Thanks to that comparison all the candidate pairs that have a Jaccard similarity that is less than t are deleted because they are pairs that were considered *candidate pairs* from the LSH method but which are actually not similar pairs.

Now, all the work is done and by simply grouping by key the RDD I will have for each tweet the list of truly similar tweets, so the ones for which the Jaccard similarity exceed the threshold.

5 Conclusions

At this point the algorithm has produced, from the corpus of tweets given as input, the list of those that have similar tweets associated with them. To better visualize the results, it is possible to print for each tweet in this list the text of the retrieved similar tweets.

5.1 Comparison with datasketch package

From the visualization of the results the algorithm seems to work properly because I can't find any strange or unexpected result. But, to have a more complete result I decided to compare my results with the ones coming from the datasketch package.

The datasketch package is designed and wrote to "gives you probabilistic data structures that can process and search very large amount of data super fast, with little loss of accuracy". In particular, it has a really easy way to perform minhash LSH that is the central part of my algorithm.

So, I use this package to replicate the LSH step of my algorithm and then to compare the results. I gave in input to the dataskecth LSH method the same parameters I use for my custom algorithm, such as the number of permutation, bands and rows. After having done different attempts changing parameters in order to have a more comprehensive view, I could say that there are some differences between the two algorithms, but these differences seems to be logical. More in detail, the two algorithm always retrieve approximately the same number of tweets that has similarities in the corpus, but there are differences in the list of similar tweets that the two algorithms are able to find. I summarized in the below table the numerical result from the comparison of the two algorithm done using the dataset of the 8^{th} October.

day	# tweets	n	b	r	threshold	custom	datasketch	equal rows	different rows	percentage of diff. rows
8-Oct	86471	40	4	10	0.8706	4617	4639	3526	1091	23.52
8-Oct	86471	80	8	10	0.8123	4998	4983	3701	1297	26.03
8-Oct	86471	80	10	8	0.7499	5483	5485	3940	1543	28.13
8-Oct	86471	40	8	5	0.6598	6518	6494	4045	2473	37.94
8-Oct	86471	80	16	5	0.5743	7219	7174	4179	3040	42.38
8-Oct	86471	40	10	4	0.5623	7594	7567	4243	3351	44.13
8-Oct	86471	80	20	4	0.4729	8485	8428	4509	3976	47.18

Figure 4: Table with numerical comparison between algorithms

In the table I marked as *custom* the number of tweets retrieve by my algorithm and as *datasketch* the ones retrieved form the package method. Then, for each choice of parameters I calculated how many tweets are retrieved in the same way by the two algorithms, in other words tweets that has the same list of similar tweets in both executions. From that I calculated the tweets for which the algorithms results are different and what is the percentage of error between the two.

Analyzing the table it is clear that both the algorithms work properly, because the differences in the number of tweets retrieved are always around or less than 1%. Furthermore, the number of different rows is inversely proportional to the threshold value and this make sense because higher is the threshold, the more similar as to be two tweets to be considered *candidate pair* and this also means that higher is the threshold, lower is the number of tweet retrieved. Since LSH is a probabilistic procedure, it is right that could be some different and it is also right that the percentage of these differences decreases as the threshold increases.

The percentage of different lines may nevertheless be too high, but this could be caused by the different hash functions used in the two algorithms and again by the fact that we are using probabilistic algorithms, and so having precise results is almost impossible.

5.2 Scalability

By the point of view of scalability, the implementation proposed is scalable first of all for documents that are much bigger than a simple tweet because a document of an arbitrary length is represented by a fixed length list of minhash signatures. The algrithm is also scalable in the number of document that could be processed thanks to the Locality-Sensitive Hashing that allow to compare, losing a bit of accuracy, lots of documents in a really fast way.

Moreover all the step are implemented using *pyspark* having the data collected as RDD and this allow the user to run the algorithm in parallel in a distributed environment, making the computation faster and allowing the algorithm to be used even for large datasets.