Finding Similar Items

Sara Manfredi

July 2022

1 Introduction

The project consists in finding similar documents in a collection using an implementation of the algorithm Locality Sensitive Hashing (LSH). It will be used the Map-Reduce framework in order to deal with the big amount of data to be processed.

1.1 Concept of similarity

In this project the concept of similarity between two documents will be equal to the Jaccard Similarity (JS) between them.

The JS is defined on two sets of items as follow: $|S \cap T|/|S \cup T|$ where S and T represent the sets.

1.2 Big Data

The project has to deal with big data since the problem of collecting the pairs of similar documents need to create and test all the pairs. If we have m documents, and m is already a big number, the number of pairs to be checked for similarity is $\binom{m}{2}$ that is well approximated by $\frac{1}{2}m^2$.

1.3 Locality Sensitive Hashing

The LSH algorithm permits to obtain a set of candidate similar pairs without have to compare all the possible pairs of documents. It's an approximate algorithm based on the intense use of hashing functions. In the output it will produce some truly similar pairs of documents, (True Positive) and some pairs that are not similar (False Positive).

2 Algorithms used

The algorithm used is the Locality Sensitive Hashing (LSH). It's an approximate algorithm that is able to find candidate pairs of similar elements, that is a list of pairs that contains some elements that are truly similar and some that are

not, without looking to every possible pair of elements in the dataset. This algorithm is composed of several steps:

- Represent each element as a set, so that we can exploit the Jaccard Similarity to calculate the similarity between elements.
- Convert each item in the set to an integer.
- Create a smaller representation of the set using a family of hashing function.
- Applying the banding technique to the smaller representation of the set in order to get the candidate pairs.
- Check the true similarity of the candidate pairs.

2.1 Represent each element as a set

In this project an element is a document, and so to represent it as a set a technique called k-Shingle is used.

This technique takes a text and divide it in all the string of k-length contained in it.

2.2 Hashing the shingles

Each shingle will be hashed using the polynomial rolling hash function to an integer value. In this way, the shingle will need less space to be represented in the data.

Moreover, it will be useful to have an integer and not a string to represent the elements of the set in the subsequent analysis.

2.2.1 Polynomial Rolling Hash Function

Polynomial rolling hash function is a hash function used for strings that uses only multiplications and additions. The following is the function:

$hash(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + \cdots + s[n-1] \times p^{n-1} \quad mod \quad m$

or simply,

$hash(s) = \sum_{i=0}^{n-1} s[i] \cdot p^i 1 \quad mod \quad m$

Where

- The input to the function is a string s of length n.
- p and m are some positive integers.
- The choice of p and m affects the performance and the security of the hash function.

- If the string s consists of only lower-case letters, then $p = 31$ is a good choice.
- m shall necessarily be a large prime since the probability of two keys colliding (producing the same hash) is nearly $\frac{1}{m}$
- The output of the function is the hash value of the string s which ranges between 0 and (m - 1) inclusive.
- Common values for m are $10^9 + 7$ and $10^9 + 9$.

2.3 The Characteristic Matrix

It's a way to represent a collection of set. Each row represents an element of the vocabulary of the sets, and each column a set. A column will have a 1 in correspondence of a row if and only if that document/set contains that item/shingle.

Since the number of rows, the vocabulary, in this setting will be very high, and the documents will have only a small fraction of the vocabulary in their set representation, is it usually not used this representation. Instead, a document/set is represented only by the rows/shingles that it contains.

2.4 Signatures

It is necessary to create a smaller representation of the sets since the set representation of a document require more space than the document itself and so it will create more problem to be stored in main memory. To create a compressed representation of the set, called "signatures", it will be necessary to permute randomly the rows if the characteristic matrix and the signature of a column is the number of the first row, in the permuted order, in which the column as a 1.

The important property that we want to ensure is that similar documents will also have similar signatures. This statement is more true with as more signatures are calculated for every column/document.

2.4.1 Actual Implementation of The Signatures

Since it's very time consuming create a permutation of a big number of rows, what is done is to pick a hash function that has as many buckets as number of rows and pass through this hash function all the element of a column that have a value equal to 1 as argument of the hash function and the signature will be equal to the smaller value found. These hash functions will be Min-Hash functions.

2.5 The Min-Hash Family

This family represents all the hash functions that create a permutation of its input. In particular, these functions are expressed as: $(ax + b) \quad mod \quad p$ where

- a is an integer value between 1 and p
- b is an integer value between 0 and p
- x is the integer passed as input
- p is the maximum value contains in the input and also in the permutation

  Note: if this value is a prime number, the min-hash function will be a true permutation of its input otherwise there'll be collisions.

The compressed representation of the documents will be made up of several signatures. Each signature is the result of the application of one min-hash function. The basic way for obtaining the signature from a min-hash function is to take the document with all its hashed shingles, pass each hashed shingle through the min-hash function and store as signature the minimum value.

2.5.1 Variant of the Min-Hash procedure

In this project, we have used a slightly different version that divide the hashed shingles in d groups and apply the same min-hash functions to all the d groups creating d signatures from 1 min-hash function. This technique has the advantages of using less min-hash functions and of having information about the similarity of documents coming from different areas of the two but there is the possibility that a document doesn't have a single hashed shingles in one or more of the groups. We can deal with this disadvantage by using big groups so that the unlucky situation is rare.

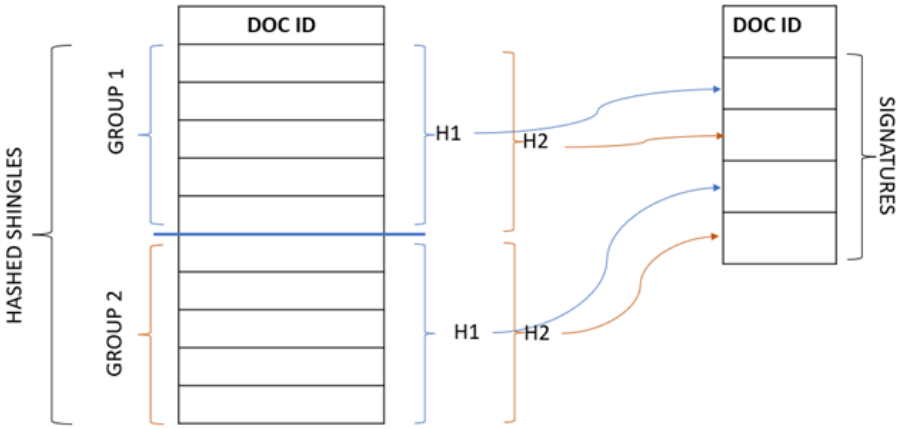

Figure 1: Variant of the min-hash procedure

It can be proved that if two documents are similar using the shingles set representations, they will also be similar using the signature representations.

Of course, the more the min-hash functions we use, the more this statement is true.

2.6 Banding technique

Once the documents are represented as a list of signatures, the space in memory to store them will be reduced but the problem of compare a big number of pairs is the same. This technique permits to find only the most similar pairs above a threshold of similarity. Once we have the documents represents as a list of signatures, we divide this list in b bands of r elements and we create a hash value from each of these b bands of r rows. If two documents will have the same hash value for at least one of the b bands, they will be considered a candidate pair.

2.7 Check the true similarity of the candidate pairs

The LSH is an approximate technique and so, the candidate pairs it will select will also contain some pairs that are not similar (False Positive). And the list of candidates won't contain some true similar pairs (False Negative). So, before presenting the results, we need to check the true similarity of each pair. For doing so, it's necessary to calculate the similarity of their sets using the Jaccard Similarity.

2.8 Parameters Setting

The parameters of the algorithm to be set are:

- k, the length of the shingles.
- $TotNumberHashFunction$, the number of signatures to be used.
- $ThresholdValue$, the threshold of similarity above which a pair is considered similar.

Note: the number of bands and rows to be used in the algorithm, are dependant on the number of signatures and the value of the threshold.

$$s^* = \left(\frac{1}{b}\right)^{\left(\frac{1}{r}\right)}$$

where:

- s^* is the value of the threshold that has been selected, $s^* = ThresholdValue$
- b is the number of the bands, must be an integer
- r is the number of rows for each band, must be an integer

- $b * r$ must be equal to $TotNumberHashFunction$, the number of signatures to be used

Note: The choice of the k parameter is application dependant, and since in this project the text of a tweet is at maximum 280-character long, the choice of k has been 5. In this way each shingle is enough rare to be informative for the specific documents it belongs to.

3 The dataset

The dataset used in the project is the "Ukraine Conflict Twitter Dataset" dataset published on Kaggle. The dataset can be found here.

It is composed of several CSV files. Each of them contains a fraction of the tweets about the ongoing Ukraine-Russia conflict occurred in that day in different languages.

3.1 Pre-Processing

3.1.1 Selection of the data of interest

This phase will select the tweets in english and it will maintain only two columns:

- *tweetid*: is a number 19 characters long unique for each tweet.
- *text*: is the text of the tweet.

3.1.2 Text Cleaning

In this phase, only the informative part of the text will be maintained. That is, the text will be all converted in lower case, and it will lose all the non alphabetical characters. In particular it won't have anymore the punctuation and the emoticons.

On the other hand, it will maintained the mentions(@) and the hashtags(#) since, in this particular setting, they have a specific meaning in the text.

3.2 Presence of Duplicates

In the dataset there are duplicates, that is tweets that share the exact same *text*. Since the purpose of the project is to find similar documents and not identical ones, it has been decided to remove the duplicates.

3.2.1 Remove The Duplicates

In order to remove the duplicates it has been chosen to use the polynomial rolling hash function (See Section 3.2.1) to create an hash value of the entire text. Documents that share the same value are considerate as duplicates and are removed.

3.2.2 Maintain One Representative

One document between all the duplicates is maintained inside the dataset in order to use it for finding similar documents.

4 Map-Reduce Implementation

The code has been implemented using the Map-Reduce framework and so a distributed computation.

4.1 Connect with kaggle and Dowload the Dataset

To download the dataset is necessary to have an account on Kaggle, more information can be found here.

4.2 Starting Point

The procedure starts from a RDD composed of (key, value) pairs where:

- key: is the *tweetid*
- value: is the *text*

Note: in the code is possible to specify how many csv files use to create the RDD.

4.3 Setting the Parameters

This experiment has been done using the following parameters:

- $file$, the file chosen is the "$0401_UkraineCombinedTweetsDeduped.csv$"
- k is set to 5
- $TotNumberHashFunction = 100$
- $ThresholdValue = 0.7$

Imposting those parameters, the best choice for b and r found is:

- $b = 10$
- $r = 10$

4.4 Text Cleaning

In this phase each *text* will be cleaned (See Section 3.1.2) in parallel using a Map function.

4.5 k-Shingle Hashed

The output of the preceding phase will be the input of this one. The cleaned text will be divided in k-shingle and each k-shingle will be hashed using a Map Function.

Afterward, using a flatMap function, every k-shingle of one document will be part of a (key, value) pair where:

- key is the *tweetid*
- value is the k-shingle hashed

4.6 Number of Distinct k-Shingles

It will be calculated the number of distinct k-shingles since it will be needed later.

4.7 Creation of the Min-Hash Functions

The number of Min-Hash function created is based on:

- the number of distinct k-shingles
- the number of bands
- the total number of signatures

See section 2.4.1

Note: in the implementation the number of bands is equal to the number of groups in which the characteristic matrix is divided in order to create more signatures using the same min-hash functions. In this way, it's possible to reuse the groups already formed to perform the hashing of one band.

4.8 Creation of the Min-Hash Values

Every $(tweetID, hashedShingle)$ is transformed in $((tweetID, groupID), listMinHashValues)$ where:

- $groupID$ contains the group of the hashedShingle.
- $listMinHashValues$ contains the value of all the Min-Hash functions when hashedShingle is passed as value.

Note: the $groupID$ value for every hashedShingle is calculated using the module function. Then, using a ReduceByKey function, every tuple that share the same key, that is the same $tweetID$ and $groupID$, is send to the same working node that calculates the minumum value for each of the Min-Hash functions. The result will be composed of b tuples for each $tweetID$ every one composed of r signatures.

4.9 Hashing The Bands

Starting from $((tweetID, groupID), listMinHashValues)$ the result of this section will be $((groupID, hash(listMinHashValues), tweetID)$ where the function hash takes the list of Min-Hash values, convert each element in a string and concatenate them in a big string that is passed through the Polynomial Rolling Hashing Function. In this way, it is created a hash value for each band.

4.10 The Candidate Pairs

In order to get the candidate pairs, that is pairs of $tweetIDs$, the result of the previous phase is joined with itself, then filtered (so that only one triangular matrix is preserved) and in the end passed through a ReduceByKey function in order to maintain only one copy for each candidate pair (a pair can have in principle more hash values in common).

4.11 True Positive

As is has been already said, this procedure gives in result also some False Positives. In order to maintained only the pairs that have a JS above the threshold, their text is been retrieved and the true JS calculated.

4.12 How it scales up

This implementation is able to scale up with data because it's composed of:

- Map functions take in consideration one tweet at the time and so with the increase of number of them the time required to process them will increase only if also the number of working nodes remains the same. Moreover, since the tweets have a maximal length of 280 characters, it's sure that it won't be a problem to maintain the text in memory
- The implementation of the reduce functions are all commutative and the list that is passed as input has a limited size per definition of the implementation in most of the functions. If some of the functions will have a problem with the number of items that they receive, especially with the join function, is always possible to implement the SALT technique. SALT is a technique that adds random values to push Spark partition data evenly.

5 Experimental Results

Using the parameters as explained above, the results were:

- $TruePositive = 112.370$
- $FalsePositive = 59.891$

That is, the 65.2% of the results are pairs with a similarity equal or grether than the threshold. Analyzing a bit more the candidate pairs given in output by the algorithm, in particular by grouping them by their threshold value, this barchart has been constructed:

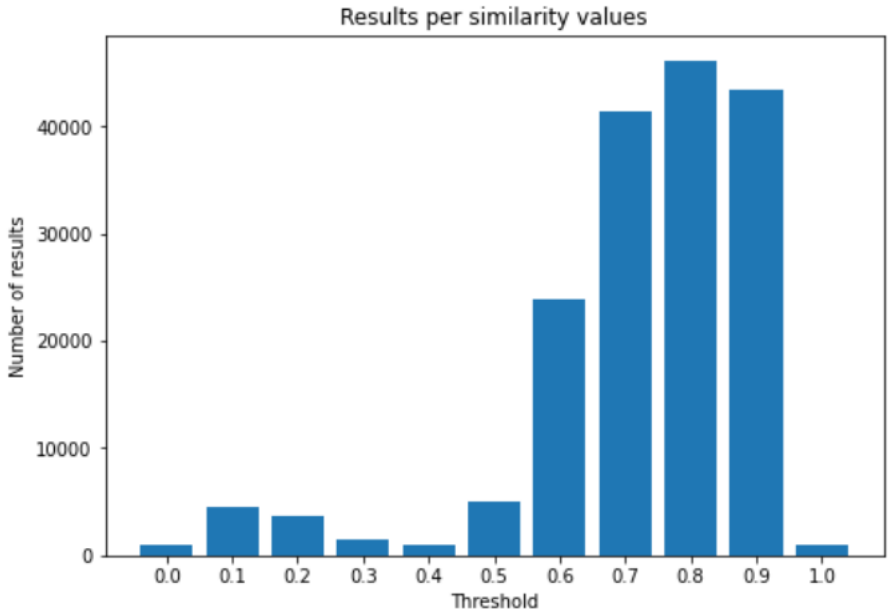

Figure 2: Barplot of the results

5.1 Creation Of The Test Set

Analyzing the dataset has been immediately clear that some tweets were almost identical except for the mentions.

From here the idea to construct a testset: that is consider all the pairs of documents that differ only in the mentions as pairs that the algorithm should find as similat and see how many of them the algorithm can find.

5.2 Results From The Test Set

In the experiments, 85.45% of the pairs in the test set has been found.