Information retrieval - Assignment 3

Plagiarism Detection

Group members:

- Laura Gotra
- Mirko Gaslini
- Martina Ceccon

Contents

1 Introduction

The topic of this third assignment is Plagiarism Detection. The goal of this report is to show how to implement a Plagiarism Detection algorithm to detect similarity and, in the worst case, plagiarism between all the documents inside a given corpus(in this case our dataset). Moreover it will be shown in details our implementation, in particular our algorithm, the hyper-parameters choice and optimization, and the analysis and the evaluation of the work. All the code and results can be view on this **link.**

2 Implementation's detail

2.1 Environment, Framework & other software

The software that was used to write, compile, run and debug codes is **Google Colab**. Google Colab is a Python development environment that runs in the browser using Google Cloud. In order to implement the Plagiarism Detection algorithm have been used a set of classic libraries used for the management of dataframes, lists, series, token and plots of graphs, is now shown the list of major libraries used like:

- pandas;
- numpy;
- nltk;
- matplotlib.

The dataset for this project is a corpus of news articles documents. There are two main sets of data, a large version and a small version, for both version the first column is the unique ID. For each news article int the small dataset IDs start from **0 to 999**, and for the large dataset start from **0 to 9999**. For both datasets the second column represents the content of the individual documents.

- The small dataset, **"news_articles_small.csv"** consists of 1000 news articles.

- The large dataset, **"news_articles_large.csv"** containing 10000 news articles.

The small dataset is used to test the tuning of parameters; b, r, m, and perform a useful analysis that will serve to determine the best setting of parameters that will be used on the large dataset useful to generate the final result.

2.2 Algorithm

The Plagiarism Detection algorithm is based on the theory and the structure of a plagiarism detection system that have been explained during the lectures. The functions that will be seductively illustrated were inspired by the following code [1]. The first part of the algorithm called **Jaccard similarity - Ground thruth** is relative to implement the **Jaccard similarity** scores between each pair of documents in the dataset, in particular each document is divided in set of n-grams with length equal to 3, to be used as ground-truth data to evaluate the **LSH** algorithm that will be further explained in the paper. In particular this first part is divided in the following functions:

- **pre_processing(word)** this function takes in input one word and perform the following operations; remove punctuation from the string, in the case where the punctuation is at the end of the string, display the unpunctuated string and transform the word into lower case word;

- **get_tuples_nosentences(txt)**: this function takes in input a text of one document in the collection and transform all the document text into **lowerCase** text then split the text in n-grams where n=3 (length of the shingle) and return the list of all n-grams of one document;

- **jaccard_similarity(document1, document2)**: this function takes in input a pair of documents and calculate the Jaccard similarity between the shingle sets;

- **createDataset(list_doc1,list_doc2,weights)**: this function takes in input 2 lists of documents and the relative jaccard similarity weight and create a dataframe with all the pairs between the documents with a column which is insert the value of Jaccard similarity between them;

- **doc_jaccard(preprocessed_documents)**: this function takes in input one list of preprocessed document, that have been processed under previous functions already mentioned, and call the Jaccard_similarity function per each pair of documents in the collection, the process is created in the way that given 2 documents it will be calculated the jaccard similarity between them only one time because sim(D1, D2) = sim(D2, D1), and will never calculated the similarity on the diagonal because sim(D1, D1) = 1.

The next part is called **Shingles and MinHash implementation**, is used to create all the shingles and implement the MinHash. This components are implemented using 3 custom functions:

- **docsShingles(documents)**: this function takes in input a set of documents and apply the following operations; per each document divide the corpus in tokens and then create the shingles set, each shingle is composed by 3 words, call the pre-processing function **pre_processing(word)**, and in the end the function return all the docsID (documents ID), docsShingleSets (sets of shingles), totalShingles (total number of shingles);

- **pickRandomCoeffs(k, maxShingleID)**: this function takes in input a value k which is representing the number of hashes that we want to generate and maxShingleID that is an integer number with value $2^\wedge 32 - 1$, this function append into the list called **randList** some randIndex (random index) randomly generated and in the end return this list;

- **minHash(numHashes)**: this function takes in input the desired number of hash functions in order to generate the coefficient that will be used later to create the hashMin matrix (matrix formed by all the hashed shingles of each document in the way that each column represent one document and the rows are the hashed values of the shingles). We used this approach to find find all the hash codes for the shingles based on the [1], [2].

The following part called **LSH implementation** is treating the LSH algorithm implementation, in particular is based on 2 functions:

- **similarDocs(signatures,r,b)**: this function takes in input the signatures (column's matrix), the value r (number of rows per each band) and the value b (number of bands). Per each signature (column of the matrix) are taken the correct values of a particular band, after this it will be created a dictionary where the key is the minhash value and the field is/are all the documents that have as a minHash value the key value so, if 2 or more documents share the same minHash values they will be put together;
- **createListCandidates(hashmap)**: this function takes in input the dictionary having as a key the minHash value and as a field all the documents having a particular minHash value, and find which documents are in the same bucket becoming then candidates to be similar documents.

The last part of the algorithm is called **Main**, in this section of the algorithm all the functions previously presented are called to; execute the code, perform the operations, retrieve the performance measures and obtain a result of candidate pairs for both, small and large dataset.

3 Analysis and evaluation

Before proceeding with the estimation of the parameters M, b and r, the Jaccard Similarity was calculated between each pair of documents in the dataset **"news_articles_small.csv"**. For the comparison of documents it has been chosen to use 3 words tokens. As the bar plot shows 1 the small dataset is very unbalanced: most documents have similarities between **0.0 - 0.2**, values too low to consider similar documents, while the remaining part (10 pairs of documents) is between **0.9 - 1** and are considered near-duplicate/duplicate.

The following part of the analysis is focused on the parameters' tuning: **b** (number of bands), **r** (number of rows per each band), and **M** (number of hash functions). To achieve this values various attempts were performed to find out the best results. At first, several number of hash function values have been selected: **M = 20, M = 24, M = 30, M = 36, M = 40**. Furthermore, per each hash functions, **6** combinations of the parameters are considered including also the extreme cases where the value of **b** or **r** is equal to 1.

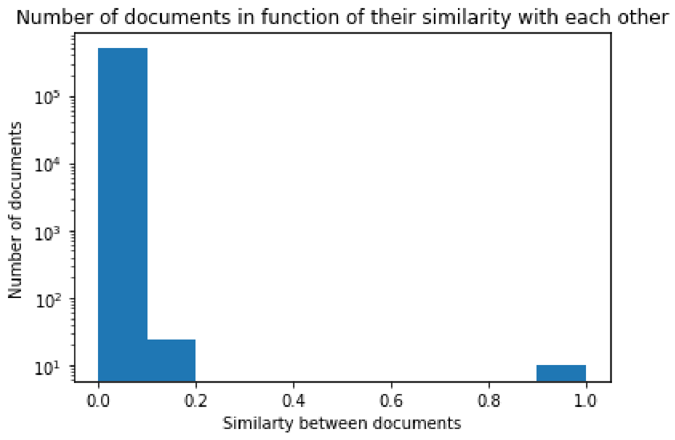

Figure 1: Jaccard similarity between each pair of the documents inside the "news_articles_small.csv"

As shown in the tables 1,2,3,4, the following values are calculated: threshold $\frac{1}{b}^{\frac{1}{r}}$, number of True Positive (TP), number of False Positive (FP), number of False Negative (FN) and the value p2 that is calculated as $1-(1-t^r)^b$. This last value represents the probability that two documents are considered candidate pairs when they have similarity over or equal to the threshold value (**0.8**).
In order to obtain the numbers of TP, FP, FN, the ground truth of the small dataset is used.

3.1 Results with 20 hash functions

In the first case with number of hash functions equal to **20** the best combination is the case with $\mathbf{b = 5}$, $\mathbf{r = 4}$ that produces a threshold value $\mathbf{t = 0.668}$ and **p2 = 0.928**. Of course, the extreme cases are excluded because of their extremely high/low value of the threshold that leads to false positive and false negative pairs.

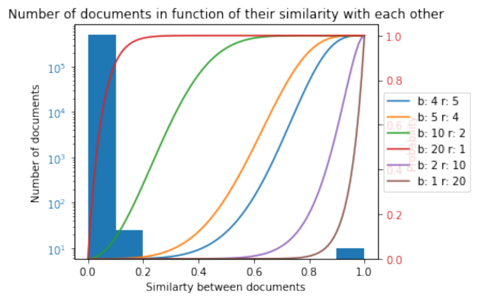

Figure:3.1. Curves with numHash equal to 20

b	r	Threshold	TP	FP	FN	p2
1	20	1.0	5	0	5	0.011
20	1	0.05	10	279	0	0.999
10	2	0.316	10	2	0	0.999
2	10	0.933	7	0	3	0.203
5	4	0.668	10	0	0	0.928
4	5	0.757	10	0	0	0.795

Table 1: Values obtained with numHash equal to 20

In the case of **b = 10** and **r = 2** the value of p2 is equal to 0.999, but the threshold value is too low and this combination will lead to false positive pairs.
In the case of **b = 2** and **r = 10** the value of the threshold is very high (0.933) but the value of p2 is equal to 0.203. This is because the threshold value in the calculation of p2 is 0.8 and therefore highlights the possibility of not considering pairs of documents with similarity between 0.8 and 0.933 (false negative pairs).
The third case, **b = 4** and **r = 5**, is similar to the optimal case (b = 5, r = 4) but the value of p2 is lower.
The graph 3.1 visually shows the results obtained. In particular it is clear that in the extreme case with r = 1 false positives are many.

3.2 Results with 24 hash functions

With regard to the next analysis, the assumptions made previously remain valid. Also to avoid repetitions, only the best results will be reported for each value of numHash with an explanatory comment on why this particular result is considered as better than all the others.

In the second case with number of hash functions equal to **24** it is considered as best result the pair **b = 4**, **r = 6**, having a threshold value very near to **0.8**, a quite high value of p2, and because comparing to the other combinations, here, the number of FP and FN values are 0.

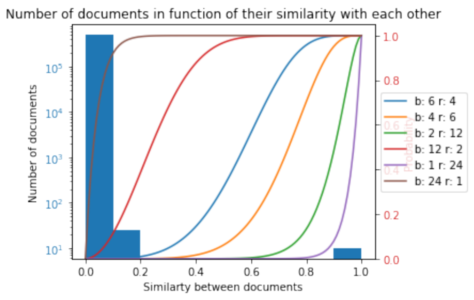

Fig 3.2: Curves obtained with numHash equal to 24

b	r	Threshold	TP	FP	FN	p2
6	4	0.63	10	1	0	0.957
4	6	0.793	10	0	0	0.703
2	12	0.943	7	0	3	0.132
12	2	0.288	10	2	0	0.999
1	24	1.0	4	0	6	0.004
24	1	0.041	10	395	0	1.0

Table 2: Values obtained with numHash equal to 24

As the graph 3.2 shows the curve trend that reduces as much as possible the obtaining of false positives and false negatives is the orange line. This curve represents the best combination of b and r just discussed.

3.3 Results with 30 hash functions

In the third case the number of hash functions is equal to **30** and the pair **b = 6**, **r = 5** is considered the best result.
With these values the propability p2 is high and the threshold differs slightly from 0.80 (almost 0.70) not producing any case of FN or FP.

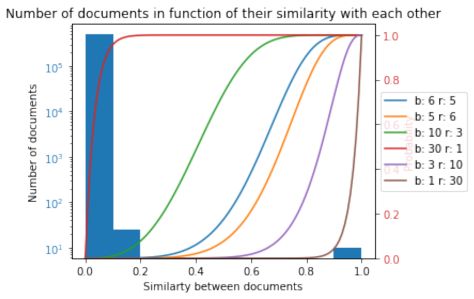

Figure 3.4: Curves obtained with numHash equal to 30

b	r	Threshold	TP	FP	FN	p2
6	5	0.698	10	0	0	0.907
5	6	0.764	10	0	0	0.781
10	3	0.464	10	0	0	0.999
30	1	0.033	10	200	0	1.0
3	10	0.895	9	0	1	0.288
1	30	1.0	5	0	5	0.001

Table 3. Values obtained with numHash equal to 36

As the graph 3.3 shows the curve trend that reduces as much as possible the obtaining of false positives and false negatives is the blue line. This curve represents the best combination of b and r just discussed.

3.4 Results with 36 hash functions

In the fourth case the number of hash functions is equal to **36** and the pair **b = 9**, **r = 4** is considered the best result. The p2 value is very high and does not produce FP and FN cases.

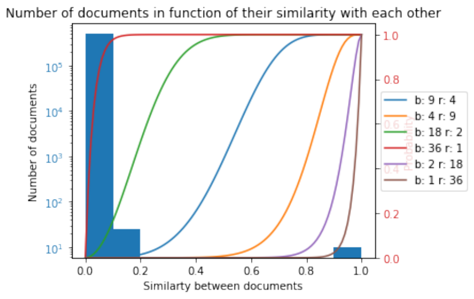

Figure 3.6: Curves obtained with different b and r for numHash equal to 36

b	r	Threshold	TP	FP	FN	p2
4	9	0.857	10	0	0	0.438
9	4	0.577	10	0	0	0.991
18	2	0.235	10	1	0	0.999
36	1	0.027	10	138	0	1.0
2	18	0.962	9	0	1	0.035
1	36	1.0	4	0	6	0.0003

Table 4: Values obtained with numHash equal to 36

As the graph 3.2 shows the curve trend that reduces as much as possible the obtaining of false positives and false negatives is the blue line. This curve represents the best combination of b and r just discussed.

3.5 Results with 40 hash functions

In the last case the number of hash functions is equal to **40** and the pair **b = 8**, **r = 5** is considered the best result. With these values a threshold value very near to **0.8**, a quite high value of p2 and the number of FP and FN values equal to 0 are obtained.

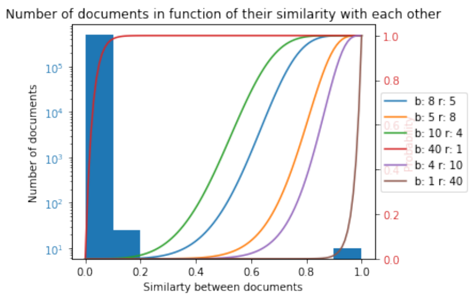

Figure 3.5: Curves obtained with numHash equal to 40

b	r	Threshold	TP	FP	FN	p2
8	5	0.741	10	0	0	0.958
5	8	0.794	10	0	0	0.600
10	4	0.235	10	0	0	0.994
40	1	0.027	10	394	0	1.0
4	10	0.962	9	0	1	0.0365
1	40	1.0	4	0	6	0.0001

Table 5: Values obtained with different b and r for numHash equal to 40

As the graph 3.2 shows the curve trend that reduces as much as possible the obtaining of false positives and false negatives is the blue line. This curve represents the best combination of b and r just discussed.

3.6 Other analysis

This analysis has shown that the values of precision and recall are both always high excepting for the extreme cases where b or r is equal to 1, this is because the documents that are part of the ground truth have very high similarities in case they are similar (all with a Jaccard similarity greater than 0.90), and very low in the opposite case (less than 0.2). This distribution allows the system to always locate the candidate pairs in the correct way without almost never obtaining false positive or false negative documents. For this reason, the choice of b and r values was made by first evaluating the other parameters.

After choosing the optimal values of b and r for each number of hash function, we want to define the lower bound of the tolerance zone and identify the final optimal values for M, b and r. This choice was made to minimize the probability that they would be rated as candidate pairs (p1) for the lower bound, maximize the probability for the upper bound and have a threshold close to 0.8. Knowing that the upper bound is equal to 0.8 we tested different values of lower bound (0.5, 0.6, 0.7).
The following table shows the results.

NumHash	b	r	p2(s2=0.8)	p1(s1=0.50)	p1(s1=0.60)	p1(s1=0.70)
20	5	4	0.92	0.28	0.50	0.75
24	4	6	0.70	0.06	0.17	0.39
30	6	5	0.90	0.17	0.38	0.67
36	9	4	0.99	0.44	0.71	0.92
40	8	5	0.95	0.22	0.48	0.77

Table 3.6: Table that shows different value of p1 varying the value of s1

As the table 3.6 shown the best cases are 3 (number of hash; 24, 30, 40) all with a lower bound equal to 0.5. In particular in the case of number of hash functions equal to 24 the value p1 is very low (0.06) therefore minimized but p2 does not turn out to be maximized (0.70). The second lower value of p1(0.17) is associated with numhash equal to 30 it also has a high value of p2 (0.90). The third lower value of p1 (0.22) is associated with hash number equal to 40 and it also has a high value of p2 (0.95). Summarizing the best two cases are with Numhash equal to 30 and 40 which with the respective values of b and r selected allow to minimize p1 and maximize p2. The final choice falls on numhash equal to 40, b equal to 8 and r equal to 5 as the threshold is closer to 0.80 as you see in the table 3 and 5. In Figure 2 the curve displays the trend of probability that two documents are considered similar to the variation of similarity between the latter. It is also indicated the tolerance zone (0.5-0.8), below the lower bound the probability that the system considers two similar documents, when in reality are not, equal to 0.22 (p1). Documents that have are near duplicates and therefore have similarities greater than 0.8 are likely to be defined such even by our system is equal to 0.95

4 Large Dataset

After the above analysis, the best parameter values obtained are used to find all pairs of documents that are duplicates or near-duplicates in the news_articles_large.CSV. Has been used the function explained in the section above to; perform the preprocessing of data, shingling, signature matrix generation, applied the LSH algorithm, and with this implementation, find candidate documents (documents in the same bucket). From the candidate documents, it has been used the Jaccard similarity

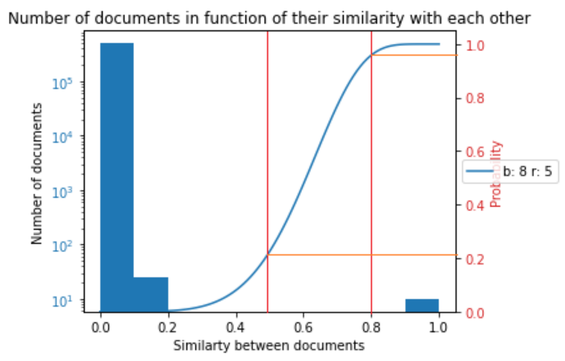

Figure 2: Graph of final choices of M = 40 b = 8 and r= 5 with tolerance zone.

measure the similarity of the candidate pairs and thus returns the documents that have a similarity of at least 0.8. In particular, 80 near-duplicate candidates have been identified, calculating the Jaccard similarity on these candidates we have noticed that all the similarities are greater than 0.8, so our system here does not produce any false positives. The results have been saved in a CSV file called **result.csv** such that each row represents a plagiarised pair, and each column represents a document in the pair (Doc id1, Doc id2).

5 Conclusion

The small dataset has been used to analyze the performance of the constrained algorithm as a detector to detect plagiarised news articles through the LSH technique. Through the construction of ground truth it has been seen how the number of pairs of documents that are near-duplicates has a high similarity above 0.90 are very few, while the majority of documents have very low similarity below 0.20. This distribution allows for easier detecting of near-duplicates for the built algorithm. The choice of the values of M, b and r are respectively 40, 8, 5 that allows a minimization of false negative and maximization of true positive in order to obtain the best trade off precision and recall.

These parameters have been used with the large dataset to generate the final result which identified 80 near-duplicate.

References

[1] Chris McCormick. “MinHash”. In: *https://github.com/chrisjmccormick/MinHash/blob /master/runMinHashExample.py* (2015).

[2] Chris McCormick. “MinHash Tutorial with Python Code”. In: *https://mccormickml.com/ 2015/06/12/minhash-tutorial-with-python-code/k* (2015).