Information Retrieval:
Locality Sensitive Hashing

Stijn Rosaer, Quentin De Haes, Jolan Depreter

January 30, 2021

1 Introduction

In this report, our task was to build a plagiarism detector based on a collection of news articles. The goal was to apply the theoretical concepts of Local Sensitive Hashing (LSH). We decided to implement every step of the algorithm ourselves because this made everything more clear and increased our understanding about the algorithm of LSH.

We divided the project into several steps, starting with the preprocessing of the dataset, computing the Jaccard index between every pair of documents and finishing up with the LSH algorithm using a plethora of different configurations. In order to make it easier to make these varied configurations, we decided to group all necessary variables together in a config file (config.py).

2 Preprocessing

We use a number of different preprocessing techniques. To begin, we ensure all terms are lowercase. Next we remove all punctuation and other special characters along with standalone letters and all numbers, to then remove all excess white spaces. Afterwards we stem all verbs and remove the stopwords defined by the nltk library. We write the resulting dataframe back to a new csv so the preprocessing doesn't need to be redone each time we rerun the project. We used the nltk library for Python to achieve this.

3 Jaccard Index

We calculated the Jaccard Index ourselves. To compute it we used shingles of the documents. We used shingles of size three, but this can be changed in the config file. Our final formula is as follows:

$$J(A,B) = \frac{|shingles(A) \cap shingles(B)|}{|shingles(A) \cup shingles(B)|}$$

We compute this for every document pair to get our 'ground truth'. In figure 1, we plot the results of the Jaccard Index. The amount of docs are plotted on a logarithmic scale for a similarity range. Note that the total amount does not equal the total amount of possible document pairs. This is because document pairs with a similarity of zero are excluded from the graph. Observe that there are no document pairs with a similarity between 0.2 and 0.9. This will be important later on when choosing what kind of sensitive hash function we require to solve our near-duplicate detection problem.

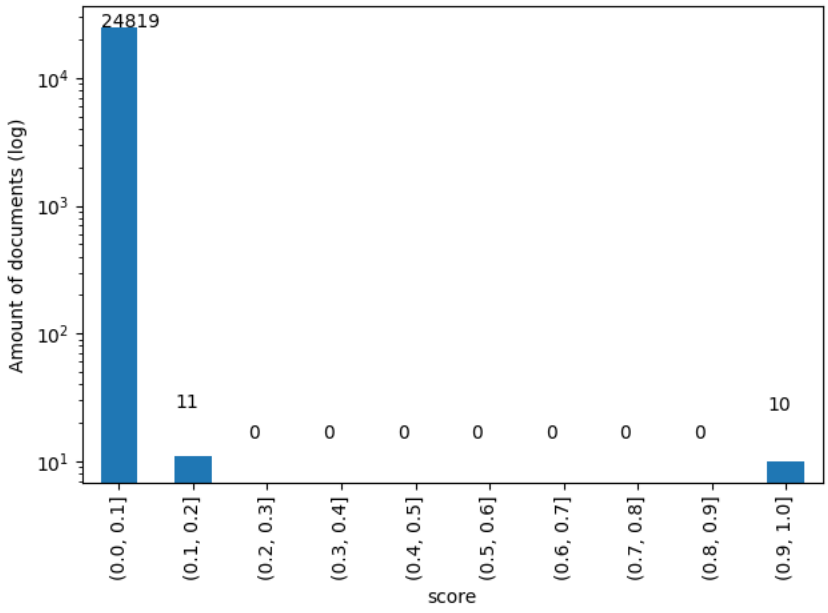


Figure 1: Plot of Jaccard distribution for the small dataset

4 Local Sensitive Hashing

In order to find near duplicates without introducing the computational complexity of calculating the Jaccard index, it is advised to use a local sensitive hashing algorithm. The general idea of LSH is the use of a family of hash function that hash data points into buckets so that data points that are near each other are located in the same bucket with a high probability.

4.1 Shingles and Hashing

Shingling is the conversion of a document into a set of consecutive words of a specific length. This gives us groups of words that occur together in the document. When looking for a specific sequence of words, we then only need to find a shingle containing these words and not looking for every occurrence of that sequence.

There are several properties that can be related to shingling which are useful for local sensitive hashing.

- Documents that are more similar share a larger amount of shingles
- Shingles preserve the order of words, but are not affected by the order of paragraphs or sentences

The creation of shingles is done with the nltk library. We use the ngrams function to create shingles with a set size. This size can be changed in the config file ("shingle size"). This is also how we created them when we computed the Jaccard Index.

To create a family of independent hash functions we used the default python hash function. This will return a different value as long as the objects themselves are different. In our case different objects with the same values will still result in the same hash. For example, two different instances of 'a-rose-is' will results in the same hash value. To create k different hash functions we start by taking the default python hash of the shingle and then XOR it with a random 64-bit mask. This allows us to create as many hash functions as we want (as many as there are 64-bit integers). We use these hash values to compute our min-hash value and store it with the document id. After generating i min-hash values we get our signature of the document. We do this for every document (using the same hash functions). These signatures are used to find the similarity between two documents. The chance that two documents x and y are mapped to the same value is then defined as:

$$P[h_\pi(x) = h_\pi(y)] = sim(x, y)$$

where $h_\pi(x)$ denotes the min-hash value of the shingles of x.

This results in a signature matrix as defined in figure 2. Each column in this matrix corresponds to the signature of a document and each row corresponds to the min-hash value of h_{row}.

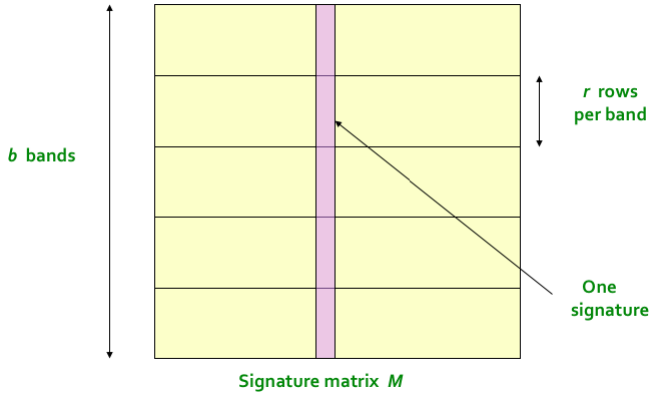


Figure 2: Signature matrix

We will experiment with the length of our signature and look at the results. In other words what value for i gives us the best results?

4.2 Bands, Rows and Iterations

We decided to take an alternative approach to achieve the optimal combination between the amount of rows per band in the signature matrix. The config file contains all the necessary parameters to derive this. We have d1 and d2 to indicate the minimum and desired similarity between two documents. They define the tolerance zone. p1 and p2 give the probability and also the minimum and maximum amount of iterations that bound the range of allowed iterations. These iterations directly correspond to the number of permutations that are done on the signatures of the documents. Thus d1, d2, p1 and p2 correspond to a (d1, d2, p1, p2)-sensitive hash function:

$$P[h(x) = h(y)|sim(x, y) < d1] < p1$$
$$P[h(x) = h(y)|sim(x, y) \geq d2] \geq p2$$

Our implementation then computes the amount of iterations, bands and rows that result in the closest (d1, d2, p1', p2')-sensitive hash function when compared with the original (d1, d2, p1, p2)-sensitive hash function.

The reason a range of permitted iterations is utilised rather than a single amount of permutations is because we have deduced that a bigger signature does not always end up with a superior result. For example when using 7 iterations, only 7 buckets of each 1 row or 1 bucket of 7 rows can be made, which have either a very high number of false positives or a very high number of false negatives. Whereas 6 iterations has 4 differing combinations of buckets and rows that can be made, allowing for better results regardless of the fact that less permutations are used.

For each number of iterations within our range, we attempt every possible combination of rows and buckets and calculate the probability of two d1 similar documents being found by the LSH algorithm and the probability of two d2 similar documents being found by the algorithm, we return the combination of rows and buckets allowed by our range of iterations that has probabilities p1' and p2' closest to p1 and p2 respectively.

5 Results

We want to find the optimal value for the signature length and the similarity threshold (d2). The signature length is related to the amount of permutations which we chose arbitrarily as the values $4, 7, 8, 9, 12, 15, 18, 24$ and 200. For the similarity threshold we took the values in the range $[0.7, 0.9]$ with steps of 0.05, where the lower bound for the tolerance zone (d1) is always chosen as 0.05 less than the similarity threshold.

Since we know that the result is dependent on both the signature length and the similarity threshold, we took every possible combination between those two parameters.

We will only discuss and present the meaningful results, but the complete output of our program can be found in the Github repository.

We chose to only take 0.7 as minimum because as can be seen in figure 1 there are no documents that have a similarity between 20% and 90%. We can use the 0.7 threshold to see if the similarity sore, given by the LSH algorithm would lower the similarity compared to the Jaccard index.

Bands	2
Rows	2
False positive chance	0.6664937500000001
False negative chance	0.2601
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 5844), (198, 373), (219, 221), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774), (499, 587), (624, 627), (631, 635)

Table 1: LSH Results for Similarity threshold: 0.7 and Signature length: 4

5.1 Influence of Iteration Amount / Signature length

The amount of iterations (and permutations) directly influence what values for the amount of bands and rows can be used. As described in section 4.2 a larger amount of iterations does not necessarily increase the precision of LSH. Prime numbers are especially bad because they are only divisible by themselves and thus only generate two possibilities for rows and bands. The same can be argued for numbers with few common dividers.

Bands	7
Rows	1
False positive chance	0.99935
False negative chance	0.00021
Candidate pairs	(0, 1), (0, 322), (0, 990), (1, 347), (1, 481), (3, 23), (4, 9), (6, 15), (8, 396), (8, 582) and 746 more

Table 2: LSH Results for Similarity threshold: 0.7 and Signature length: 7

Indeed we notice that despite the use of more permutations, we get a significantly worse result than when we take only 4 permutations (Table 1 vs Table 2).

5.1.1 Higher signature length

We'll also check whether the computationally more intense signature length of 200 has a noticeable impact in comparison to LSH with a signature length 24 . To remain consistent, we'll check both cases with the similarity threshold of 0.8 and the lower bound of 0.75 .

Bands	20
Rows	10
False positive chance	0.6862709679100705
False negative chance	0.10313091656075302
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 3: LSH Results for Similarity threshold: 0.8 and Signature length: 200

Bands	6
Rows	4
False positive chance	0.8979557588874023
False negative chance	0.04235240655244277
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 4: LSH Results for Similarity threshold: 0.8 and Signature length: 24

As the above results show, the longer signature length reduces the chances of false positives in comparison. The chance of getting a false negative is doubled however. And due to how the LSH algorithm creates candidate pairs to be checked using jaccard afterwards, it is preferable to have a low false negative chance, however having a too high false positive chance is also not preferable as this would be equal too checking everything with jaccard, which is what we're trying to avoid. Due to the skewedness of our data, this is not a problem, since no datapoints exist between 0.2 and 0.9 similarity.

5.1.2 Lower signature length

Another option would be to see if a smaller signature length still gives us a decent result. This would be beneficial since it lowers the computational cost. To do this, we compare the results that are given four 24 permutations (Table 4) versus only 4 permutations (Table 5).

Bands	2
Rows	2
False positive chance	0.80859375
False negative chance	0.12959999999999994
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (219, 221), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774), (499, 587), (624, 627), (631, 635)

Table 5: LSH Results for Similarity threshold: 0.8 and Signature length: 4

We can see that the amount of of candidate pairs increases with a lower signature length. But the chance on a false positive is lower in this example.

This is because there is a worse relation between the amount of bands and rows in Table 5 (see figure 3).

5.2 Influence of Similarity threshold

The similarity threshold is equivalent to the d2 of our (d1, d2, p1, p2)-sensitive hash function. Adjusting this will influence the values for our bands and rows. This may in turn influence the candidate pairs returned by the LSH algorithm. When we look at our 'ground truth' in figure 1, we noticed that there are no docs in the similarity range of [0.2, 0.9). Thus our tolerance zone [d1, d2] may be quite big.

To compare the amount influence of the similarity threshold on the result, we only use the best results from above, this is with 24 permutations.

Bands	6
Rows	4
False positive chance	0.6926554037119399
False negative chance	0.19254784708792538
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 6: LSH Results for Similarity threshold: 0.7 and Signature length: 24

Bands	6
Rows	4
False positive chance	0.8074521529120746
False negative chance	0.10204424111259769
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 7: LSH Results for Similarity threshold: 0.75 and Signature length: 24

Bands	6
Rows	4
False positive chance	0.8979557588874023
False negative chance	0.04235240655244277
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 8: LSH Results for Similarity threshold: 0.8 and Signature length: 24

Bands	4
Rows	6
False positive chance	0.7035943671390585
False negative chance	0.15049955856799369
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 9: LSH Results for Similarity threshold: 0.85 and Signature length: 24

Bands	3
Rows	8
False positive chance	0.6149510338821813
False negative chance	0.18473798357867965
Candidate pairs	(103, 205), (122, 523), (151, 480), (197, 544), (198, 373), (264, 880), (282, 918), (289, 746), (332, 802), (372, 774)

Table 10: LSH Results for Similarity threshold: 0.9 and Signature length: 24

If we choose a similarity threshold in the range of [0.7, 0.9] we do not see any difference in the candidate pairs for our document collection. This is highly attributed to the fact that there are no documents with similarities in the [0.3 0.9) range. We do observe that our ideal bands and rows changes for different similarity thresholds. This explains the sudden drop in false positive chance and the increase in false negatives.

5.3 Larger data file

We used the larger data with similarity threshold 0.8 and 20 permutations to get a signature length of 20. This amount was not hard-coded by us, but calculated by our program to get an optimal value.

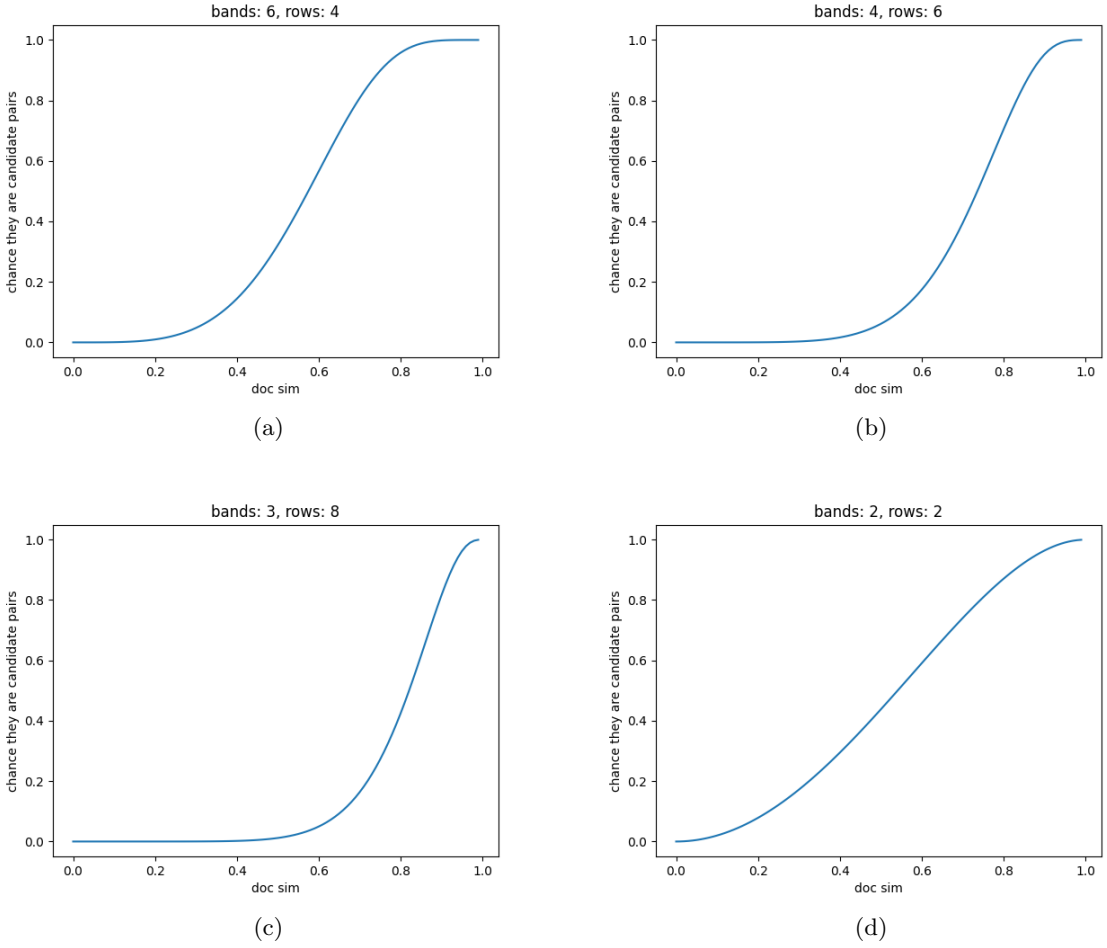
Because the dataset is not that different from the smaller one, we do not see a difference in results or accuracy. It is however very useful to see the difference in performance between the Jaccard index and LSH. Computing the Jaccard index for this larger dataset took around one hour, but the LSH algorithm that we implemented was drastically faster.

These observations are closely related to what we have seen in the theoretical lectures. The results themselves are not present in this report, but can be found in the Github repository (results.csv).

6 Conclusion

Notice that our false positive chance at 0.75 with 4 permutations (Table 5) goes down in comparison to 24 permutations (Table 4). However the precision is better with 24 permutations. This can be explained by the figure 3. The step function for 2 rows and 2 bands is not a good approximation of the step function. It is more representative of the function: $f(x) = x$.

From everything the we found, we can conclude that the relation between the amount of bands and rows is the most important to get a better result and not only the signature size. This is because the amount of bands and rows determine the chance on false positives or negatives as can be seen in figure 3 .



(a) (b) (c) (d)

Figure 3: Chance y that two docs with sim x become candidate pairs.

7 Information

The source code and output from our program can be found on Github trough this link:
`https://github.com/stijnrosaer/plagiarism-detection-ir`

We also decided to make some test cases in order to test the correctness of our functions. These can be found in test.py and can be useful to see the expected output of a function. The preprocessing of the csv file is done by running preprocessing.py and the eventual jaccard index and LSH algorithm can be executed by running main.py

References

[1] GUPTA S., *Locality Sensitive Hashing An effective way of reducing the dimensionality of your data*, source: https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134, 29-06-2018