Information Retrieval
Assignment 3:
Plagiarism Detection using LSH

Daniëlle Jongstra
s0172260

Jens Leysen
s0201907

Johannes Voshol
s0200178

1. Introduction

For the course "Information Retrieval", the goal of the third assignment is to implement a near-duplicate detection algorithm using locality sensitive hashing (LSH). We will be implementing this system and document its capabilities, seeking to learn more about LSH.

The goal of section 2 is to give an overview of LSH. Here, we'll explain the Jaccard similarity and talk about concepts such as shingling, minhashing and signatures.

In section 3, we write about the implementation of our near-duplicate detection system. We discuss the pre-processing that has been applied to the dataset and what frameworks and libraries have been used.

In section 4, we discuss the influence of different parameters, such as the amount of bands and rows per band.

Finally, we discuss things we wanted to improve or revisit in section 5.

2. Background

In this section different processes and techniques are described that are needed for Locality-Sensitive Hashing, which can be used to detect near-duplicate documents in an efficient way.

2.1. Jaccard similarity

The Jaccard similarity can be calculated using the formula

$$Jaccard(doc1, doc2) = \frac{doc1 \cap doc2}{doc1 \cup doc2} \quad (1)$$

In this formula, doc1 and doc2 are different documents that are compared for the amount of similarity. The resulting Jaccard similarity will be a value between 0 and 1. A similarity of 1 indicates that the compared documents are identical. The similarity value becomes smaller if the documents are less alike. If all documents in a certain data set with a total of N documents need to be compared, an amount of

$$\frac{N * (N-1)}{2} \quad (2)$$

comparisons are needed, as stated in [1].

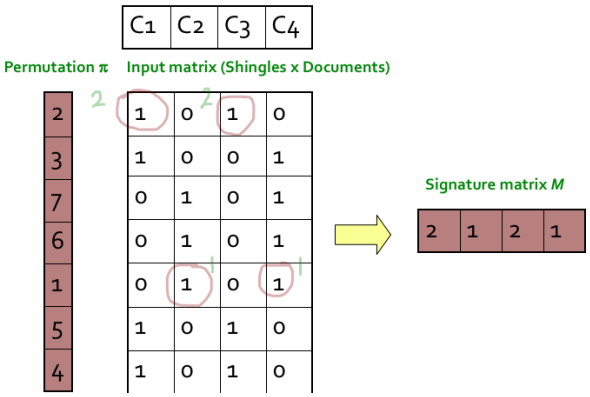

Figure 1. Creation of a minhash signature [2]

2.2. Shingling

To retain some of the structure of a document, it is possible to use shingles as explained in [1]. A shingle is a sub-string that appears in the document of k words, with k an integer greater than zero. It is also possible to take k as the amount of characters in a shingle instead of the amount of words. To generate all the shingles in a document with words "A B C D E", we take all possible sub-strings: "A B C", "B C D" and "C D E".

2.3. Minhashing

The shingles and documents form a matrix, as described in [2]. This matrix has a one entry if a shingle appears in a certain document and a zero entry if it does not. The result is a sparse matrix with very few one values. Minhashing is used to eliminate this sparseness, while still trying to maintain the similarity of the documents. For this, there is first a random permutation taken of the row indices in the shingle-document matrix. The random permutation is used to create a signature of a column. This is done by repeatedly hashing the rows using different permutations. A hash function can be seen in figure 1. It uses the one entries of the row that is the lowest value of the permutation. In figure 2 is the generation of a signature matrix using multiple

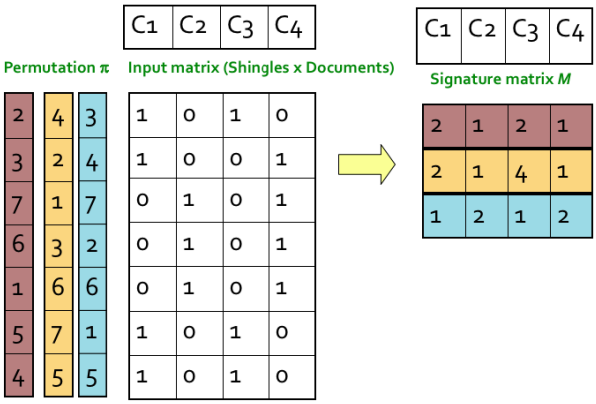

Figure 2. Creation of a minhash signature matrix [2]

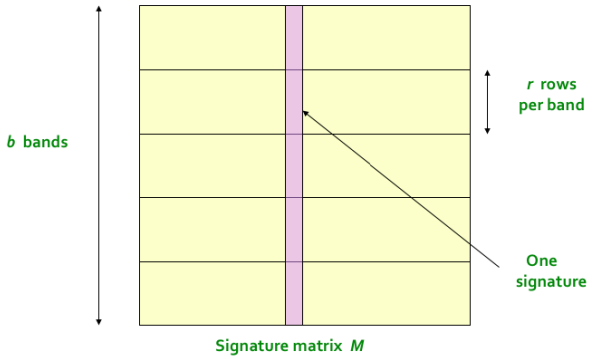

Figure 3. LSH bands and rows [2]

random permutations. The similarity of two documents is now the amount of rows of the signature matrix in which the documents have the same value, divided by the total amount rows.

2.4. Locality-Sensitive Hashing

The idea behind Local-Sensitive Hashing (LSH) is to place documents that are similar to one another in the same hash bucket, while placing dissimilar documents in another bucket. The signatures of the documents are used to determine whether it is likely that the documents have a high similarity, making them a candidate pair. The minhash signature matrix is used to decide which pairs of documents are candidate pairs. This by hashing the columns of the signature with several different hash functions. The documents are a candidate pair if the documents hash to the same bucket for at least one of these hash functions. The LSH algorithm as stated in [2] creates the different hash functions by dividing the signature matrix into b bands that each contain r rows. This is visualized in figure 3. For each band and column, the part of the column that belongs to the band is hashed into a hash table with k different buckets. The candidate pairs are now those that hash to the same bucket for at least one of the bands. For these candidate pairs,

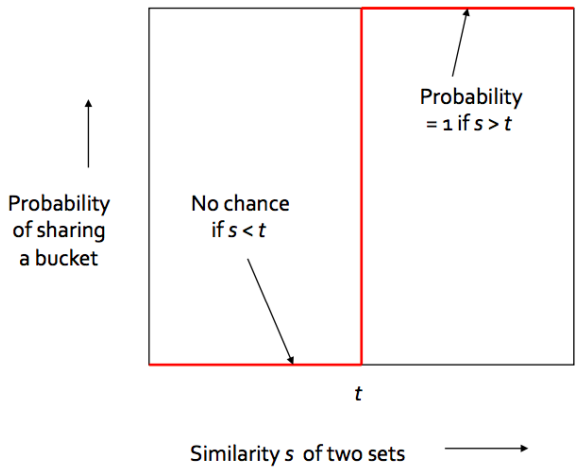

Figure 4. The ideal LSH probability function of documents with a certain similarity being candidate pairs if the similarity threshold is t [2]

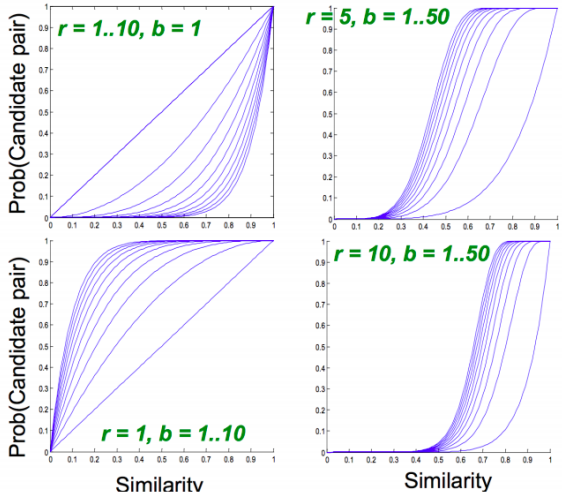

Figure 5. Probabilities of documents with a certain similarity being candidate pairs for different values of b and r [2]

the Jaccard similarity can be calculated to confirm that the documents are indeed above a set similarity threshold. Good results from LSH means we have found a lot of the similar documents, while not getting many dissimilar documents. For this it is important to pick good values for the amount of bands b and the amount of rows r. We will try to find the best values for b and r specific to our system in section 4. The ideal situation is visible in figure 4. Different attempts of approaching this situation with certain values for b and r can be seen in figure 5 for a general case.

3. Implementation

For the implementation of the plagiarism detection, the libraries pandas, numpy and seaborn have been used. The parts are described separately. The custom im-

plementation can be found under its own section in: IR_PlagiarismDetection.ipynb.

3.1. Jaccard similarity: Ground truth

The Jaccard similarity is calculated using formula 1. The code for this part is original work. To be able to use the formula, all the documents in the small news article data set are minimally pre-processed by just splitting the words to a list and converting it to a set. Every document in the data set has their own set, making it possible to differentiate between different documents. Sets are used to only have one occurrence of each word per document. With this pre-processed list of sets it is now possible to calculate the Jaccard similarity of each two documents in the data set. Every document is compared to each other document that has a higher index, which prevents two documents being compared to each other twice. The resulting Jaccard similarity value of each two documents are added to a list. This list can be used to generate a histogram to have a visual representation of in which intervals the Jaccard similarities are. The seaborn library is used for visually representing the histogram, by defining 50 bins with each a width of 0.05.

3.2. Hash functions

We need a permutation family π, such that every π_k maps values of sketches to a rank differently. Given that one hashed shingle is 32 bits, all mappings take $k \times 2^{32} \times 32$ bits of space, which corresponds to $k \times 16$ Gigabyte of space. This approach is not feasible as k grows to a large number. Using 64 bit shingles is nearly impossible with 128 Exabyte required to store a static mapping.

In LSH, hash functions are used for permutations instead of a static π. This is done by hashing a shingle and use its result as the rank of this value.

$$min_\pi(S) = min_h(S) = min_{s \in S}\{h(s)\}$$

To generate different hash functions, we use a fixed length cryptographic salt(K_r) that will be generated for all hash functions using a seed. This ensures that every h_r produces a different output for a single input value.

$$\pi_r(s) = h_r(s) = H(K_r|s)$$

We need to replace π with a hash function that maps the complete space of hashed shingles (2^{64}) to the a new space, with minimal collisions. Often a hash function maps a large space to a smaller space where collisions are guaranteed to exist. If we treat the added salt as a static input, our chosen hash function will map a 64 bit shingle to a 128 bit rank. The probability that two shingles are hashed to the same rank is astronomically small.

When shingles of strings are hashed to 64 bits to compress data, collisions are still possible, as the shingle space is larger than 64 bit.

	MinHash/ms (avg)	MSE
SHA1	50.65	16158
MD5	50.65	19820
xxhash	62.11	8286
FNV64	15.47	1208973
CRC64	11.7	771416160

Figure 6. Sketch size = 10, minHashes=200k

	MinHash/ms (avg)	MSE
SHA1	5.47	1569
MD5	5.29	1906
xxhash	7.31	2133
FNV64	1.55	16461
CRC64	1.15	1514042

Figure 7. Sketch size = 100, minHashes=200k

For the actual implementation of LSH, hash function H can be chosen from a set of existing hash functions (using existing libraries). We conducted a small experiment to choose the best hash function for our project. First of all, we need a library that is fast and performs well. This is measured by min-hashes/ms. Security of the algorithm is of no concern to us, as it does not serve to provide any security. However, we need our hash function to be random and unbiased. This means that every value in the sketch has the same probability to be chosen as the minimum value. For example, if the number of min-hashes=1000, sketch size=10, then we expect each sketch value to be chosen on average 100 times. In our experiment we took the Mean Squared Error (MSE) of minhash distrubution by comparing to these expected values. From the results in Table 6 and 7, it is clear that both hash algorithms FNV and CRC are biased towards certain sketch values, so these functions will not be used. The MSE comparison for SHA-1, MD5 and xxhash varies for different sketch sizes and amounts of min-hashes. This is why we chose for the algorithm that can perform the most min-hashes in a given time: xxhash.

3.3. LSH Functionality

A super class `LSHFunctionality` is defined as an interface that can be used by the custom implementation and the library to make an effective comparison more easy. First, an LSH implementation must store the chosen configuration for n_gram, #bands and #rows. It must also provide the option to the data-set through a list of strings as well as the option to provide a path to a csv file. The actual indexing is done through the function `compute`. The user must be able to retrieve all similar documents in the data-set by using `get_all_similarities` and query a document against the indexed data-set through the function `query`. In both these retrieval functions, the user specifies a similarity threshold `s`.

3.4. Using a library

We first used the Snapy [3] library to implement the LSH functionality and is very straight forward. Looking through the source code of this library, we noticed that it returns only documents with a approximated Jaccard value that is being calculated by number of band hits divided by the total number of bands. Because of this, we could not do our analysis, as we would like all bucket hits to be returned given a document query. For this we needed our own custom implementation.

3.5. Custom implementation

The custom implementation class is built on top of the `LSHFunctionality` class and is initialized with the following variables. Matrix M, which stores the signatures for each document. A dictionary for buckets, where one band of a signature as a tuple is used as a key. We also store the similarities in a dictionary, where a document pair tuple is used as a key. Notice here that, for a configured r and b, two similar documents might not end up in any of the buckets together, and thus are not considered candidate pairs and with this do not have a stored similarity value in the dictionary.

3.5.1. Document to signature. The function `doc_to_signature` converts a document that has not been pre-processed to a signature based on the configured r and b. The following sequence of steps are taken to pre-process a document. 1) Make all the words in the news articles lowercase words. By doing this, we ensure that identical words are not treated differently if one of them are positioned at the start of a sentence. 2) After this, words with contractions are split into 2 separate words. 3) Values containing the string "&amp" are removed, as these codes are present in the csv. 4) We remove all numbers. 5) Remove double spaces 6) Remove all punctuation. 7) Split the documents in terms, which results in an array of words.

Then the list of words is used to create shingles of size n, which are then hashed to a 64 bit value. Now the document is represented as a sketch, which is smaller in size than the original document.

To create the resulting signature, the minhash is calculated for each prepared hash function and the result is put in a list.

3.5.2. Constructing the buckets. Now we would like bands of signatures to hash to the same bucket if they are equal. We achieve this by looping over each band of each signature, and use this band as a key to a dictionary. If the bucket does not exist yet, we create a new bucket. Then we store the document id in this bucket.

3.5.3. Constructing similarities. To calculate the jaccard approximations of similar documents, two documents must be candidate pairs, so they must be hashed to the same bucket at least once. All candidate pairs are found by first looping over all the buckets and check if there are at least 2 documents in this bucket. If there are more than two documents, we must create all pair combinations in this bucket ($N(N-1)/2$). We will save all pair combinations in a set, so we do not have duplicate pairs.

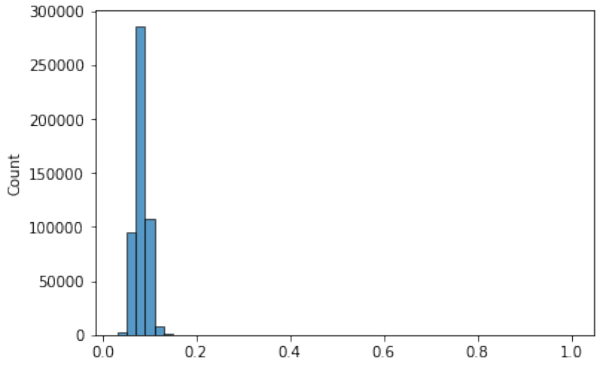

Figure 8. Distribution of the Jaccard similarties of documents in the small news article data set

For each candidate pair, we calculate a better approximated Jaccard based on the total signature. The Jaccard value will be the amount of hits divided by the signature length. In the first version of our implementation, we divided the amount of band hits by the total amount of bands (as in the library), but comparing signatures proved to approximate a better Jaccard values without adding much more cost.

3.6. Query new content

After indexing the data-set, new documents my be queried against the existing documents to detect plagiarism. The process to retrieve all candidate pairs is very simple. First we generate the signature of the new document (3.5.1). Then for each band in this signature we retrieve all candidates from the buckets the bands would hash to. Now the only thing to do is comparing the signatures of the candidate pairs to the signature of the queried document. This is where we would filter based on the provided similarity threshold by the user, but in our implementation we retrieve all candidate pairs with its Jaccard values so that we are able to do true positive/true negative analysis.

4. Evaluation

4.1. Theoretical evaluation

4.1.1. Ground truth from the sample set. In the histogram in figure 8 there are clearly many documents with a Jaccard similarity between roughly 0 and 0.2. Due to the amount of observations within these buckets, possible observations in the higher frequency regions are not visible. For this, a second histogram is generated for the observations with a similarity of more than 0.2 in figure 9. This makes it clear that almost all documents have a Jaccard similarity of less than 0.3. Only 10 document combinations have a high

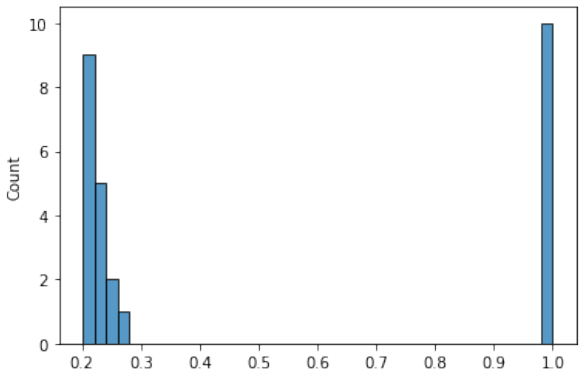

Figure 9. Distribution of the Jaccard similarties of documents in the small news article data set, focused on the similarities that are larger than 0.2

similarity. Interestingly enough, all these high similarity observations are in the highest similarity range.

4.1.2. Choosing the bands and rows. We can use a mathematical approach to find the optimal band/row combination for our data-set based on the ground truth. By putting the probability formulas in a system of equations, we can calculate r and b.

The probability that two documents A and B hash to the same bucket for any band (a candidate pair), is 1 minus the probability that the documents are never hashed to the same bucket for every band:

$$P[h(A) = h(B)|r, b, s = Sim(A, B)] = 1 - (1 - s^r)^b \quad (3)$$

We can use this formula to find r and b, such that the probability to find similar documents is high, and the probability to find documents with no similarity is low.

To find documents with a similarity of 0.8 with a probability of 99.9%, the following equation must hold:

$$1 - (1 - 0.8^r)^b > 0.999 \quad (4)$$

Now choosing r and b to satisfy this equation also affects the sensitivity of the lower similar documents. From the histograms 8 and 9, we can conclude that we do definitely not want the documents with a similarity up until 0.2 (stop wasting my time!). So we can add the following equation to our system to make sure that documents with a similarity of 0.2 do not have a higher probability than 0.1% to appear in buckets together:

$$\begin{cases} 1 - (1 - 0.8^r)^b > 0.999 \\ 1 - (1 - 0.2^r)^b < 0.001 \end{cases} \quad (5)$$

The problem here is that if we make these equations too constrained to match the ideal LSH function (Figure 4), r and b shoot up drastically, to the point where comparing each document individually is faster than constructing the signatures.

In table 11 we outputted the probabilities for different r, b and similarity values. We can see that for $r = 2$ and $b = 4$,

rows	bands	sig	s1=0.2	s1=0.3	s2=0.8	s2=0.9
2	4	8	15.1%	31.4%	98.3%	99.9%
3	8	24	6.2%	19.7%	99.7%	100.0%
4	12	48	1.9%	9.3%	99.8%	100.0%
5	18	90	0.6%	4.3%	99.9%	100.0%
6	24	144	0.2%	1.7%	99.9%	100.0%

Figure 11. Probabilities for different r and b values.

98% of documents with similarity 0.8 are found, but also 31% of documents with a similarity of 0.3 are found. With 17 documents between 0.2-0.3 similarity (see histogram 9), we would have to manually filter out approximately $17 * 0.314 = 5.3$ documents (worst case), which is not a large price to pay, given that we will have an easy time indexing with a signature length of 8. But with millions of documents, we do not want to filter thousands of documents.

For our specific (small) data-set we can still afford to index with higher signatures to maximize the retrieval of documents with similarity 0.8 and minimize the amount of documents with similarity 0.2.

As will be discussed in 4.2, we chose to use $r = 6$, $b = 24$ to index the large set. This gives us the following curve:

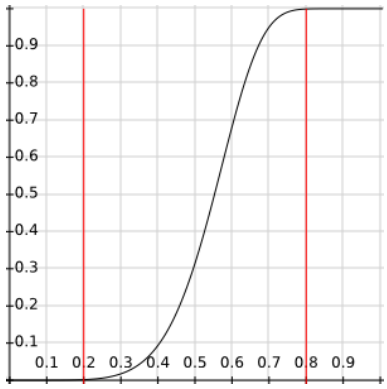

Figure 10. $p = 1 - (1 - s^6)^{24}$

We can say that the corresponding hash function is $(0.2, 0.8, 0.002, 0.999)$-sensitive.

Note that documents with a similarity of 0.5 can still be found as candidate pairs with 30% probability. However, because there are normally no documents between 0.3-0.8 similarity according to the ground truth, we can afford this.

4.2. Performance evaluation

In order to evaluate the performance of our model, we decided to create our own set of plagiarised documents and analyse the precision and recall values based on the generated candidate pairs. The evaluation has a separate section in following file: IR_PlagiarismDetection.ipynb. Please note that we stored the large and small news dataset csv files on google drive and that these should be loaded when trying to re-run the code.

4.2.1. Creating Plagiarised Documents. In order to test the performance of the system, a set of plagiarised documents was created. These plagiarised documents are created in three different ways:

- Every 'xth' word in a document is replaced by a random word.
- A word is replaced by using a uniform distribution of word positions.
- The first 'x' words in a document are replaced by random words.

The first two types of plagiarised documents represent the cases in which a person would replace some words in the document to make it look like the text wasn't plagiarised. The third way models the case in which the user copy-pastes a paragraph inside his own document. The code was written so that these plagiarised documents can automatically be generated with different values for x. The random words aren't real English words, but this won't affect our analysis. To give an example; the first document in news_articles_small.csv starts as follows:

- *"The Supreme Court in Johnnesberg on Friday postponed until March 14 a hearing on a petition..."*.

After replacing every 4th word by a random 'word' we get:

- *"sCKUG Supreme Court in krXAB on Friday postponed pteFE March 14 a YNycq on a petition..."*.

Replacing the first 10 words we just get:

- *"VfzzI rfVDS CyARw GIBRq uKqSo LNvcU rLjzk mVZAk npmZD wrJUG ZGolj rkiuI nsXCI..."*

This way, we created 60 automatically generated plagiarised documents. The names of these plagiarised documents are automatically generated and follow a fixed pattern e.g. plagiarised_doc_step_1, plagiarised_doc_range_1 or plagiarised_doc_uniform_1. Next, these plagiarised documents were filtered to approximate the distribution of the small news articles dataset i.e. only documents with a similarity larger than 0.8 and lower than 0.3 (Jaccard similarity between plagiarised document and original document) were kept. This resulted in a small test dataset of 22 documents. The similarity between these automatically generated and filtered near-duplicates and the original document can be visualised as follows:

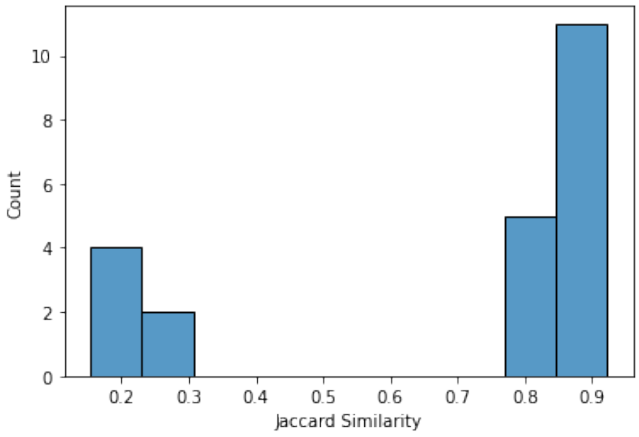

Figure 12. Jaccard Similarity between Original Document (News Article 1) and its Automatically Generated Duplicates.

4.2.2. Precision and Recall. Now, we can calculate the precision and recall of our implementation based on different parameters: the number of bands, the number of rows per band and this for different similarity thresholds. The precision is calculated as follows:

- Calculate the real Jaccard Similarity between a query (a near-duplicate we created) and the original doc.
- Find candidate pairs using the implementation.
- If the candidate pair has a Jaccard Similarity larger than the threshold, we increment a true positive counter.
- If the real Jaccard Similarity is lower than the similarity threshold s, increment the false positive count. They are false positives since we will have to examine them (they are candidate pairs) but their similarity is below threshold s.
- Precision is calculated as: true positive count / (true positive count + false positive count).

Recall is calculated in a similar manner as precision:

- Calculate the real Jaccard Similarity between a query (a near-duplicate we created) and the original doc.
- Find candidate pairs using the implementation.
- If the candidate pair has a Jaccard Similarity larger than the threshold, we increment a true positive counter.
- The recall is calculated as: true positive counter / (total relevant documents).

Our tests were run on the entire small articles dataset, the plagiarised documents discussed previously were used as queries. The results for a similarity threshold of 0.80 are shown in Figure 13. We've added indexing time to show that the indexing time increases the more rows per band and bands are used. Note: 2-grams were used.

Rows	Bands	Precision	Recall	Indexing time (s)
2	4	80%	100%	4.5
3	8	94%	100%	12.0
4	12	94%	100%	23.4
5	18	88%	100%	43.3
6	24	100%	100%	69.3

Figure 13. Test Results for Similarity Threshold of 0.8: Precision, Recall and Indexing Time

A precision of 100% indicates that for all queries the documents that really had a similarity of more than 80% were returned. When running the code, some traces are generated that can be used to validate these results. An example of one of these traces is shown in Figure 14.

```
### Test: b_4_r_2 ###
Document: plagiarised_doc_step_9 Jaccard Similarity: 0.8088888888888889 Result: TP
Document: plagiarised_doc_step_10 Jaccard Similarity: 0.8340807174887892 Result: TP
Document: plagiarised_doc_step_11 Jaccard Similarity: 0.8416289592760181 Result: TP
Document: plagiarised_doc_step_12 Jaccard Similarity: 0.8447488584474886 Result: TP
Document: plagiarised_doc_step_13 Jaccard Similarity: 0.8433179723502304 Result: TP
Document: plagiarised_doc_step_14 Jaccard Similarity: 0.8796296296296297 Result: TP
Document: plagiarised_doc_step_15 Jaccard Similarity: 0.8604651162790697 Result: TP
Document: plagiarised_doc_step_16 Jaccard Similarity: 0.8691588785046729 Result: TP
Document: plagiarised_doc_step_17 Jaccard Similarity: 0.8826291079812206 Result: TP
Document: plagiarised_doc_step_18 Jaccard Similarity: 0.8962264150943396 Result: TP
Document: plagiarised_doc_step_19 Jaccard Similarity: 0.8957345971563981 Result: TP
Document: plagiarised_doc_step_20 Jaccard Similarity: 0.9004739336492891 Result: TP
Document: plagiarised_doc_range_1 Jaccard Similarity: 0.9227053140096618 Result: TP
Document: plagiarised_doc_range_2 Jaccard Similarity: 0.8525345622119815 Result: TP
Document: plagiarised_doc_range_16 Jaccard Similarity: 0.25210084033613445 Result: FP
Document: plagiarised_doc_range_17 Jaccard Similarity: 0.226158038147138 97 Result: FP
Document: plagiarised_doc_range_18 Jaccard Similarity: 0.20159151193633953 Result: FP
Document: plagiarised_doc_range_20 Jaccard Similarity: 0.153652392947103 28 Result: FP
Document: plagiarised_doc_uniform_1 Jaccard Similarity: 0.9178743961352657 Result: TP
Document: plagiarised_doc_uniform_2 Jaccard Similarity: 0.8617511520737328 Result: TP
### Total TP: 16
### Total FP: 4
### Test Precision: 0.8 ###
```

Figure 14. Trace of a Test showing Precision Result

These traces should be interpreted as follows: for the query with the content of ”plagiarised_doc_step_9”, there was one hit. The Jaccard similarity between this hit and ”plagiarised_doc_step_9” is ”0.80888”. This means that it is a true positive since we only wanted to return docs with a similarity larger than 0.8. As can be seen, there were a total of 16 true positives, that is candidate pairs that did have a similarity of at least 80%. In total, there were 4 false positives, that is candidate pairs that had a much smaller Jaccard similarity. From the conducted tests on our small dataset, we conclude that using 6 rows per band and 24 bands returns the best result for a similarity threshold of 0.8. We repeated this test for a similarity threshold of 0.9 with exactly the same set of documents. The results are shown below in Figure 15.

Rows	Bands	Precision	Recall	Indexing time (s)
2	4	15%	100%	4.7
3	8	18%	100%	12.2
4	12	18%	100%	23.7
5	18	17%	100%	43.5
6	24	19%	100%	69.2

Figure 15. Test Results for Similarity Threshold of 0.9: Precision, Recall and Indexing Time

As can be seen, the precision is now drastically lower. The reason is of course that there are a lot of values in the 80% to 100% range. The probability that a document with a similarity of 80% is returned as a candidate is quite high. In order to properly separate values in this range, the amount of bands and rows per band would have to increase drastically. However, in most use cases a similarity between 80% to 90% wouldn't really be seen as a false positive and we would accept 80%-90% as a tolerance interval. These two examples show that the performance of LSH is directly linked to the distribution of similarities within the collection. A setup with 6 rows/band and 24 bands works great for a similarity threshold of 0.8 since there aren't any documents with similarities between 0.3 and 0.8 in our dataset. For a similarity threshold of 0.9 this setup isn't good enough because it cannot differentiate well enough between all the values in the 80%-100% similarity range. From this discussion we conclude that a setup with 6 rows/band and 24 bands works great for a similarity threshold of 0.8 i.e. it gets most pairs with similar signatures and eliminates most pairs that do not have similar signatures.

4.3. Performance on Large News Articles Dataset

The implementation was tested on the news_articles_large_dataset.csv. The goal was to return all duplicates or near-duplicates that have a similarity of at least 0.8. The setup was as follows:

- 2-grams were used
- 24 bands
- 6 rows per band

The resulting csv file can be found here: news_articles_large_RESULT.csv. As can be seen, 82 documents were returned as possible candidate pairs. This is exactly as expected, calculating the Jaccard similarity between all documents in the large dataset returns the same amount of pairs.

5. Future Work

5.1. Similarity measures

It is possible to use a different similarity measure instead of the standard Jaccard similarity. One such possibility is a Jaccard similarity that takes repeated shingles into account. It would be needed to keep a counter for each shingle, which reduces the space efficiency of the algorithm. Another option is using the cosine similarity. More research would be needed to determine whether LSH can approximate these similarity measures well. An altered version of LSH might be needed to make a good approximation of the different similarity measures.

5.2. Plagiarism detection for citing

In this assignment we saw that LSH is very efficient in finding near-duplicates for texts that differ in only a few words. However, plagiarizing a few sentences from a book will not be detected by our system. This is because the the

plagiarized text is a subset of the original text. Therefore $J(P_{plagiary}, B_{book}) = \frac{|P \bigcap B|}{|P \bigcup B|} = \frac{|P|}{|B|}, P \subset B$. If the book is very large, the Jaccard similarity is very low. To detect these types of plagiarism, we would have to index all paragraphs (like in the csv file). Still, even sentences might be a small part of a paragraph. We would have to set up our system differently to tackle these kinds op problems.

References

[1] Chris McCormick "MinHash Tutorial with Python Code" (2015) https://mccormickml.com/2015/06/12/minhash-tutorial-with-python-code/R, accessed 27-1-2021.

[2] Shikhar Gupta "Locality Sensitive Hashing" (2018) https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134, accessed 27-1-2021.

[3] Snapy LSH library, https://pypi.org/project/snapy/