IR: Plagiarism detection

Sam Legrand
Lander De Roeck
Nils Charlet

January 2021

1 Introduction

An accurate and fast plagiarism detection system is important for educational institutes like universities or for sites hosting publications. The uses however aren't only limited to these aforementioned ones, a similar system could be used to automatically flag content that is under copyright protection, content that is graphical and much more. Detailed in this report are the fundamentals on our own implementation of a plagiarism detection system, this system utilises locality-sensitive hashing with MinHash and is implemented in Python. The source code for the project can be found on GitHub: `https://github.com/SamLegrand/lsh`. All code has been written from scratch. External libraries were only used for some hashing functions, to generate plots and to import/export csv files. To run the code, a file `usersettings.py` has to be created which contains the amount of threads the system can use. For more information, please refer to the ReadME inside the repository.

2 Jaccard similarity & shingling

To calculate the Jaccard similarity, we need to create shingles, word n-grams, for our text. Increasing the n-value will decrease the computation time, but will also decrease the sensitivity. We need to find an acceptable n-value, for which the computation time is not too long, but the sensitivity is still high enough. We will be using 3-grams because we determined that these strike a good balance between sensitivity and computation time. The following formula shows how to compute the Jaccard index of two documents A and B where $ngrams(A)$ and $ngrams(B)$ are their respective sets of n-grams.

$$JACCARD(A,B) = \frac{|ngrams(A) \cap ngrams(B)|}{|ngrams(A) \cup ngrams(B)|}$$

Since utilising integer sets is significantly more performant than string sets to compute the Jaccard similarity, we hash the word n-grams before inserting them into the shingle set. For this we simply take 64 bits of the 128 bit MD5 hash of the shingle. This also reduces the amount of memory required to store the shingles.

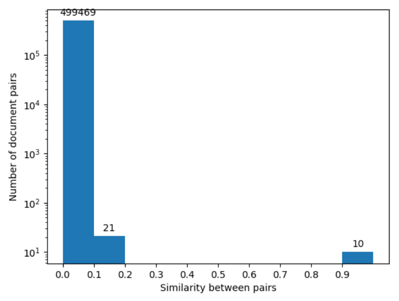
3 Pre-processing the data

Pre-processing the data could help to simplify and speed up the plagiarism problem. We need to be careful however, too much processing can be detrimental to the ability to detect plagiarised content. Some information loss always occurs when pre-processing the data. We decided to take a look at 3 different pre-processing techniques, namely removing capitalization, removing punctuation and removing stopwords. Since we wanted to make sure none of these would negatively impact our detection, we've compared the resulting Jaccard similarity distributions of the small dataset between the different techniques. The following results can be obtained by executing the `sim_analysis.py` file.

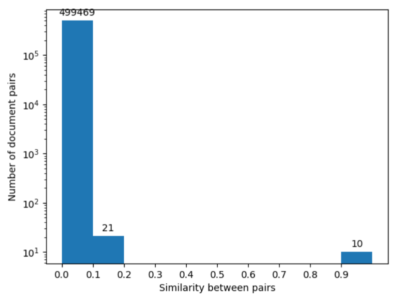
Technique	**Time for computing similarity distribution (incl. pre-processing time)**
None	8.7631 seconds
Punctuation filtering	8.3526 seconds
Stopword filtering	5.8796 seconds
Capitalization removal	8.3549 seconds
Only using 3-grams starting with a stopword	3.5001 seconds
Using 3-grams starting with a stopword and other filtering methods	4.1448 seconds

Table 1: Pre-processing benchmark

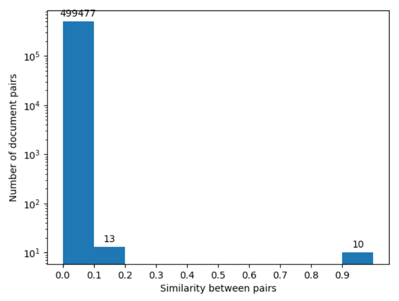
Luckily, as we can see in figure 1, it seems our ability to detect plagiarised documents doesn't degrade significantly for any technique. The time to compute the similarity has however decreased, as can be seen in table 1. This is because filtering the document results in a decreased size of the shingle sets. This will also have an effect on the generation of minhash signatures later on, and thus will also be very important for the LSH indexing performance. Another interesting approach we read about [1] was to enforce each 3-shingle to start with a stopword, which requires the removal of stopwords to not be active. This seemed to work very well for our data, and almost halved our computation time compared to having no pre-processing enabled. Eventually we settled on utilising the 3-shingles which have to start with stopwords, utilising capitalisation and punctuation removal to make shingle sets as small as possible while still capturing the document information very well. We will be using this pre-processing approach for the remainder of the project.



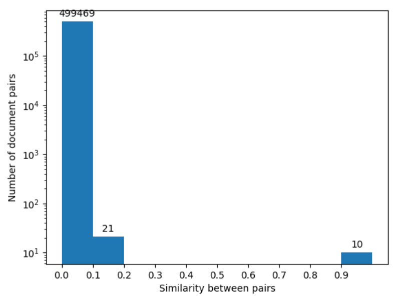
(a) No pre-processing



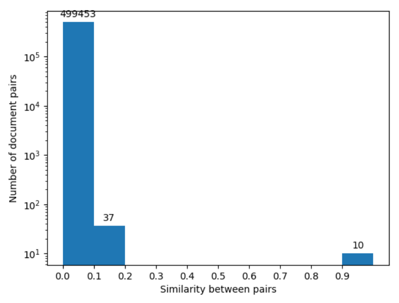
(b) Punctuation filtering



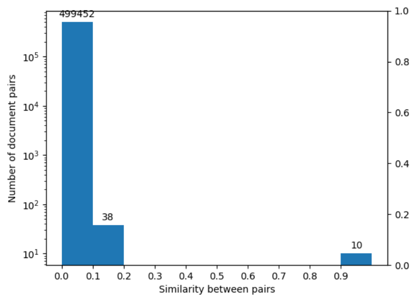
(c) Stopword filtering



(d) Capitalization removal



(e) Only using 3-grams starting with a stopword



(f) Using 3-grams starting with a stopword and other filtering methods

Figure 1: Similarity distributions for different pre-processing techniques

4 Creating the signature matrix

Since we can create $\frac{n*(n-1)}{2}$ unique document pairs from n documents, computing the Jaccard similarity for every document pair is an $\mathcal{O}(n^2)$ problem. It doesn't matter how efficient we make the computation, it will always grow to be an expensive operation. Because of this, we utilise a method to estimate the Jaccard similarity, which can be done by utilising a signature. Creating the signature for every document results in the signature matrix. Each signature is of length M, and as we are using the minhash approach to compute these, we need M different hash functions. As this implies calculating M hashes per hashed shingle per document, we decided to try some simple hash functions first for performance reasons. All code in this section can be found in `signature.py`.

The first hash function we came up with is a simple xor function `Xorhash` calculated as $h_a(x) = x \oplus a$, where $\oplus$ denotes a bitwise xor. This means we just need to choose M random values for a to generate our hash functions, and calculating such an xor is very efficient. However, after some initial experiments, we figured that this simple hash function might not be good enough. We used two sets $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$ that represent the hashed shingles of two documents. These two sets have a Jaccard index of $\frac{1}{5}$, so we expect that if we take many random hash functions, there would be a one in five chance that the $MinHash$ of sets A and B is the same. However, they were actually the same around one in eight times. It is quite easy to see why this is happening in this particular example. First of all, as all values in both sets are smaller than 8, all bits that come after the 3 least significant bits are always zero. This means that changing bit i of the xor constant a will simply add (when changing it from 0 to 1) or subtract (when changing it from 1 to 0) 2^i to/from the hashes when $i \geq 3$. This in turn adds/subtracts 2^i from the $MinHash$ value of both documents, and thus whether or not $MinHash_A$ is the same as $MinHash_B$ is not changed. So, we can simply try the 8 combinations of the lowest 3 bits of a and calculate $MinHash_A$ and $MinHash_B$. The following table shows these eight combinations.

a	$min(1 \oplus a, 2 \oplus a, 3 \oplus a)$	$min(3 \oplus a, 4 \oplus a, 5 \oplus a)$
0	1	3
1	0	2
2	0	1
3	0	0
4	5	0
5	4	0
6	4	2
7	4	2

We can see that indeed in only one of the eight rows, the MinHash value is the same, resulting in a calculated similarity of only $\frac{1}{8}$ instead of $\frac{1}{5}$. Similar examples can be found where the calculated similarity is too high instead of too low.

We then tried a second hash function, inspired by linear congruential generators: `Linconhash`, calculated as $h_{a,b}(x) = (ax+b) \bmod p$ where p denotes a large prime of around 64 bits. We randomly chose a value of 533603009383305529 for p. We tried using this hash function on the example sets A and B given above and got a incorrect result of around 17% matching MinHashes. After this, we decided to try a well known cryptographic hash function: we added `MD5hash` which takes 64 bits from the MD5 hash generated from the input and a random parameter a. While `MD5hash` is slower than `Linconhash`, which was already slower than `Xorhash`, it does not give biased results on specific input sets of small numbers such as the sets A and B described above. However, later on we discovered that using `Xorhash` or `Linconhash` on real hashed shingles does in fact yield correct results, while being more efficient to compute than `MD5hash`. No hash function is perfect, and one weakness of our `Xorhash` function is indeed when its inputs are very small. As it is more intuitive to use small values for initial tests, we falsely concluded that the hash function was not sufficiently random. Similarly, for small input values, the outputs generated by `Linconhash` are correlated and not sufficiently random. In conclusion, we recommend using `Xorhash` as it gives results that are just as good as the other two hash functions on real collections while being significantly more efficient to compute. The function `example_simple` in `signature.py` can be changed to try different hash functions on different hashed shingle sets. Changing it does not affect how the index in the next chapter is built.

5 Locality-sensitive hashing

We've now generated a signature matrix: this is a matrix with one column per document containing the M signatures. The Jaccard index of two documents A and B with signatures S_A and S_B respectively is approximated by $\frac{j}{M}$ where j is the number of indices $1 \leq i \leq M$ for which $S_{A,i} = S_{B,i}$. However, with n documents, calculating the approximated Jaccard index of all pairs using the signature matrix instead of directly calculating the Jaccard index is not much of an improvement, as it still requires $\mathcal{O}(n^2)$ calculations.

This is where locality-sensitive hashing comes into play. Our implementation can be found in `lsh.py`. We'll partition our matrix into b bands of r rows per band, with $b * r = M$. Then, we choose a hash function $h(x)$ which transforms any band of one document (so r values of the signature) into a natural number in the range $[0, k-1]$, where k depends on the chosen hash function. For each band, we initialize k empty buckets, each containing an empty set. Per band, we loop over each document and use the previously chosen hash function to hash this band of this document. The output c of this hash function determines to which of the k buckets we will add this document's id. Then, when we want to find near-duplicates of a new document Q, we first calculate the signature S_Q of Q. Then, we use this hash function $h(x)$ on each band of S. This will yield us one bucket per band to which Q would be added if we were to index Q. Then, instead of comparing Q directly to each document, we can look at the union of documents put in these b buckets. Ignoring the chance that two different band entries hash to the same bucket, which is very small if k is large, the union of these documents is the set of all documents whose signature has at least one band that is identical to the corresponding band in S_Q. These documents are our candidates, and we will only calculate the Jaccard index of Q and each candidate.

The idea behind this is as follows: suppose that we have two documents A and B with a similarity of s. We've created two signatures S_A and S_B of M values, and are dividing them into b bands of r rows each. We would like to know the chance that A and B will be a candidate pair. This is the chance that S_A and S_B have at least one matching band. With a similarity of s, the chance that one particular band is identical is s^r, as there are r independent values that are the same with a chance s. Then, the chance that S_A and S_B have at least one matching band is equal to $1-(1-s^r)^b$. For instance, suppose we are using 10 bands of 10 rows each. Now if A and B have a Jaccard index of 0.2 and are thus not too similar, they will be candidate pairs with a chance $1-(1-0.2^{10})^{10}$ which is around a one in one million chance. Instead, if they have a Jaccard index of 0.8 and are thus quite similar, we'd probably want them to be a candidate pair. The chance $1-(1-0.8^{10})^{10}$ works out to be around 68%. So, they are somewhat likely to become candidates. The number of bands and rows can be tuned to make the system more or less sensitive.

In our implementation, we decided on using MD5 as our hash function $h(x)$. With n documents, only $n*b$ of these hashes have to be calculated. When we were choosing a hash function for the minhashes, this number was significantly larger: M hashes per hashed shingle per document. We argue that now, we should choose a hash function that is of high quality and reduces the number of collisions to a minimum, even if it comes at the cost of a relatively expensive calculation. From our tests, it appears that even when using MD5, calculating the minhashes is still taking significantly more time than hashing the bands with MD5. As MD5 has a 128-bit output, $k=2^{128}$. In our implementation, we do not initialize 2^{128} buckets, but rather we create the bucket as soon as a document must be inserted into it. The maximum number of buckets is so large that the chance of two non-identical bands hashing to the same bucket is $\frac{1}{2^{128}}$, which is extremely unlikely to happen, even in the case of millions of documents. All implemented features of the LSH index can be found inside the ReadME.

As described above, it is quite simple given a document Q and its signature S_Q to find the set of candidates: hash each band and take the union of the b selected buckets. The buckets also provide a very convenient way to find all near-duplicate candidate pairs given an indexed collection of documents. For each bucket, we can generate a set of candidate pairs. For each bucket containing n entries with $n>1$, generate the $\frac{n*(n-1)}{2}$ pairs (i,j) where $i<j$ as candidate pairs. The actual candidate pairs are simply the union of the candidate pairs generated by all buckets. For these candidate pairs, we calculate the actual Jaccard index to filter out any false positives.

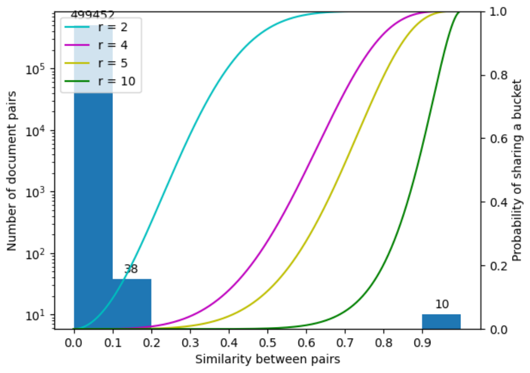
6 Analysis

6.1 S-curves

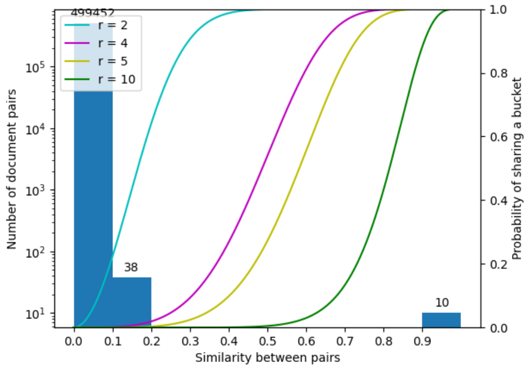
In order to analyse the performance of the LSH index for different signature lengths M and number of rows per band r, we first take a look at the S-curves of a set of combinations of those parameters. These are generated by the same `sim_analysis.py` file we've used for comparing the different pre-processing techniques. The S-curve denotes the probability of a pair mapping to the same bucket in function of the similarity between the pair. Note that the selection of M and r immediately results in a number of bands $b = \lfloor \frac{M}{r} \rfloor$, thus if $M \bmod r \neq 0$, only $b \cdot r$ rows of the signature are used for building the index. In figure 2, the S-curves for the different parameter combinations are plotted on top of the Jaccard similarity distribution. This way, it becomes immediately clear what the probability of false positives and negatives will be for particular thresholds. We can derive some interesting information from these plots. First of all, the S-curve seems to become 'narrower' for increasing signature size. In general, this reduces the number of false positives and negatives, which in turn could allow to select a lower similarity threshold if necessary. For $M = 20$, it seems quite challenging to have an optimal solution unless we set the similarity threshold very close to the common near-duplicate similarity range of [0.9, 1.0]. Thus, we will most likely opt for an M value of 50 or 100. We determine the optimal parameters based on some additional metrics in the next section.

6.2 Performance, specificity, sensitivity and precision

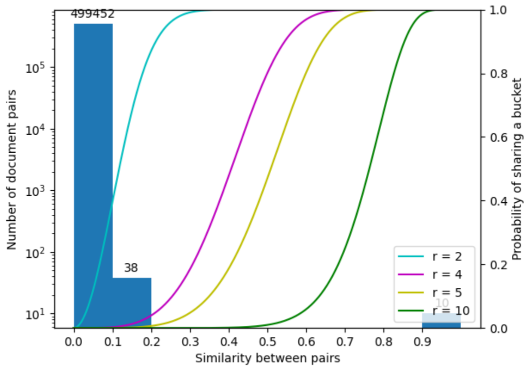
In order to determine optimal parameters for our LSH index, we also take a look at the time needed for creating the index, specificity, sensitivity and precision. Specificity is computed for each combination of M and r, for a similarity of 0.3. This similarity has been chosen as there are no non-duplicate pairs with a similarity above 0.2, while it is slightly higher in order to make it more generalizable towards the full dataset or other types of datasets. We define specificity as the probability that two documents do not share any bucket $(1 - s^r)^b$. Sensitivity is computed for similarity threshold (s) values of 0.7, 0.8 and 0.9. Clearly, lower values should not be considered as there are only near-duplicates pairs with similarity values in [0.9, 1.0]. We define sensitivity as the probability of two documents sharing a bucket $1 - (1 - s^r)^b$. Finally, precision has been determined by averaging the precision of 10 iterations, with a different LSH index in each iteration in order to reduce the influence of the randomness involved in the index generation. The precision is computed in each iteration as $\frac{|near-duplicates|}{|candidates|}(= \frac{true\ positives}{true\ positives + false\ positives})$, where the candidates and near-duplicates are determined by executing 100 different queries on the index. The queries are the first 100 documents of the given dataset with 5 mutations in random locations, in order to generate near-duplicates instead of exact duplicates. The implementation of the analysis can be found in `lsh_analysis.py`.



(a) M=20



(b) M=50



(c) M=100

Figure 2: S-curves plotted on top of similarity distribution for different M/r values

In table 2, an overview of the results for the different metrics is provided. It becomes immediately apparent that increasing the signature size increases the index creation time, which is logical as this means that more minhash values have to be computed. However, this extra creation time (and memory usage) pays off, as the specificity and sensitivity values improve when comparing with the same values for the r and s parameters. There are diminishing returns though, for example in the case of $M = 100, r = 4, s = 0.8$, sensitivity is already at 99.99% and thus cannot increase much further. Therefore, we find that an M value of 100 is optimal. It is also quite notable that a similarity threshold of 0.9 results in worse precision, indicating that some queries that are used for computing precision have a similarity of less than 0.9. As we don't want the system to be too selective (as in the larger dataset or in comparable datasets there might be near-duplicates with a similarity in the range [0.8, 0.9)), we opt for a similarity threshold of 0.8 instead. For r, it seems that setting the value too low results in a bad specificity, as can be seen for $M = 100, r = 2$. Setting the value too high results in a bad sensitivity, e.g. for $M = 100, r = 10$. From the table, our best option is to use $r = 5$ (given that $M = 100$ and $s = 0.8$), but we found that using $r = 6$ results in better specificity while only compromising on sensitivity slightly. Thus, in the optimal case we use $M = 96, r = 6, s = 0.8$ to get a (0.3, 0.9884, 0.8, 0.9923)-sensitive LSH index which can be created in 1.3523 seconds and results in a precision of 1.0 for the mutated queries.

7 Conclusion

Many real world scenarios require similarity searches in high-dimensional spaces. When dealing with collections of thousands or possibly millions of documents, it is simply infeasible to do a pairwise comparison to find pairs of documents that are similar. Locality-sensitive hashing provides an efficient solution to this problem, by efficiently generating a set of candidate pairs which are then checked to filter out false positives. With the right parameters, this system is very robust and generates very few false positives and negatives. Our implementation can efficiently index 10000 documents and find all near-duplicate pairs in less than 10 seconds on an AMD Ryzen 3700x CPU. Furthermore, due to the distribution of similarities as shown in figure 1, parameters can be chosen in such a way that there are very few false positives, and with very high probability zero false negatives. The repository root contains the file `result.csv` containing all 82 near-duplicate pairs with a similarity of over 80% that are present in the large dataset `news_articles_large.csv`. It can also be generated by running `python3 src/lsh.py` from the root.

Parameters	**Index creation time**	**(0.3, p1, s, p2)-sensitive**	**Precision**
M=20, r=2, s=0.7	1.1769s	(0.3, 0.3894, 0.7, 0.9988)	0.8777
M=20, r=2, s=0.8		(0.3, 0.3894, 0.8, 0.9999)	0.9531
M=20, r=2, s=0.9		(0.3, 0.3894, 0.9, 0.9999)	0.6401
M=20, r=4, s=0.7	1.1437s	(0.3, 0.9601, 0.7, 0.7466)	0.9990
M=20, r=4, s=0.8		(0.3, 0.9601, 0.8, 0.9282)	1.0
M=20, r=4, s=0.9		(0.3, 0.9601, 0.9, 0.9951)	0.7133
M=20, r=5, s=0.7	1.1273s	(0.3, 0.9903, 0.7, 0.5209)	1.0
M=20, r=5, s=0.8		(0.3, 0.9903, 0.8, 0.7956)	1.0
M=20, r=5, s=0.9		(0.3, 0.9903, 0.9, 0.9718)	0.7137
M=20, r=10, s=0.7	1.1314s	(0.3, 0.9999, 0.7, 0.0556)	1.0
M=20, r=10, s=0.8		(0.3, 0.9999, 0.8, 0.2032)	1.0
M=20, r=10, s=0.9		(0.3, 0.9999, 0.9, 0.5757)	0.7899
M=50, r=2, s=0.7	1.2620s	(0.3, 0.0946, 0.7, 0.9999)	0.7071
M=50, r=2, s=0.8		(0.3, 0.0946, 0.8, 0.9999)	0.8117
M=50, r=2, s=0.9		(0.3, 0.0946, 0.9, 1.0)	0.6062
M=50, r=4, s=0.7	1.2182s	(0.3, 0.9070, 0.7, 0.9629)	1.0
M=50, r=4, s=0.8		(0.3, 0.90703, 0.8, 0.9982)	1.0
M=50, r=4, s=0.9		(0.3, 0.9070, 0.9, 0.9999)	0.71
M=50, r=5, s=0.7	1.2156s	(0.3, 0.9759, 0.7, 0.8411)	0.9980
M=50, r=5, s=0.8		(0.3, 0.9759, 0.8, 0.9811)	1.0
M=50, r=5, s=0.9		(0.3, 0.9759, 0.9, 0.9998)	0.71
M=50, r=10, s=0.7	1.2515s	(0.3, 0.9999, 0.7, 0.1334)	1.0
M=50, r=10, s=0.8		(0.3, 0.9999, 0.8, 0.4333)	1.0
M=50, r=10, s=0.9		(0.3, 0.9999, 0.9, 0.8827)	0.7565
M=100, r=2, s=0.7	1.4363s	(0.3, 0.0089, 0.7, 0.9999)	0.6250
M=100, r=2, s=0.8		(0.3, 0.0089, 0.8, 1.0)	0.6572
M=100, r=2, s=0.9		(0.3, 0.0089, 0.9, 1.0)	0.4981
M=100, r=4, s=0.7	1.4012s	(0.3, 0.8160, 0.7, 0.9989)	1.0
M=100, r=4, s=0.8		(0.3, 0.8160, 0.8, 0.9999)	1.0
M=100, r=4, s=0.9		(0.3, 0.8160, 0.9, 0.9999)	0.7086
M=100, r=5, s=0.7	1.3702s	(0.3, 0.9525, 0.7, 0.9747)	1.0
M=100, r=5, s=0.8		(0.3, 0.9525, 0.8, 0.9996)	1.0
M=100, r=5, s=0.9		(0.3, 0.9525, 0.9, 0.9999)	0.71
M=100, r=10, s=0.7	1.3854s	(0.3, 0.9999, 0.7, 0.2491)	1.0
M=100, r=10, s=0.8		(0.3, 0.9999, 0.8, 0.6788)	1.0
M=100, r=10, s=0.9		(0.3, 0.9999, 0.9, 0.9862)	0.7213

Table 2: An extensive analysis of a set of metrics for different LSH parameter combinations

References

[1] Jeff M. Phillips. *Jaccard Similarity and Shingling.* URL: https://www.cs.utah.edu/~jeffp/teaching/cs5955/L4-Jaccard+Shingle.pdf. (accessed: 29.01.2021).