# Plagiarism Detection

Jenna Ingels Robbe Lauwers

January 5, 2022

**Abstract**

Plagiarism is a big problem in many industries. It has become unfeasible for companies to hire a person to look for offenders. Therefore we attempted to write a plagiarism detector. We will be describing our approach using shingles, Locality Sensitive Hashing and the minhash algorithm. We fine-tuned the number of shingles, the amount of bands and the band size for minhash in order to get an optimal result.

## 1 Introduction

You have no doubt been told many times not to plagiarise. Nevertheless, for many media houses it is still a big problem. Because they get paid for every time someone views an advertisement on the page, some of shadier publishers resort to plagiarism. After all, while unethical, it increases the amount of articles users can read on their website, and therefore the amount of time spent on there. It turns out that other media houses don't like copyright infringement, resulting in a sort of tug of war between them. For this paper, we set out to create an automatic plagiarism detector. We will be using various techniques sourced from information retrieval research. It is our hope that our technology can help publishers more readily detect and defeat plagiarism.

## 2 Implementation

Our Github repository is here: https://github.com/RobbeLauwers/Information_retrieval_3. The readme explains which files were used to generate which output.

We began by implementing and fine-tuning our program using a smaller corpus of around a thousand articles. By using a smaller set of articles we aimed to have a shorter iteration time between different experiments. Additionally, a smaller data set also enabled us to use more expensive techniques that would otherwise be unavailable to us in order to verify our results more easily.

### 2.1 Preprocessing

While we received a set of articles to test our program with, it became clear that it would not be sufficient. Therefore we did some preprocessing on the data in order to make it easier to work with. In order to make the data more usable, we removed all

non alphanumeric characters except spaces: a lot of the articles contained punctuation marks in wrong places that might trip up our program. Additionally we would have replaced all text with lowercase letters, but that proved unnecessary as this was already done for us.

### 2.1.1 Shingling

In order to use most of the techniques we used in our development, we needed to view every article as a set. We could have chosen to use the set of words used in the article, but the problem with that approach is that it removes the context from all words. This is why we instead used shingles. Converting a text into n-shingles involves taking every combination of n subsequent words in the text and grouping them.

As an example, take the sentence "The effect is the cause of the effect". If we 3-shingle this sentence, the resulting set of shingles is {"the effect is", "effect is the", "the cause of", "cause of the", "of the effect"}

## 2.2 Jaccard Index

In order to compare articles, we need a metric. We chose to use the Jaccard index. [To21] The Jaccard index is a statistical tool used to compare the similarity of sets. The index is calculated by dividing the intersection of two sets by the union.

$$J(A, B) = A \cap B / A \cup B \tag{1}$$

The higher the resulting Jaccard index between two articles, the more similar they proved to be. In order to detect plagiarism, we have to choose a cut-off point for similarity. For simplicity's sake we ended up choosing to use 80% similarity to mean there was plagiarism.

## 2.3 Locality Sensitive Hashing

While the Jaccard index gives good results, it becomes infeasible to calculate the Jaccard index for every article pair once the set grows[Bry17]. We needed to find a way to approximate the same results, while using less computing time. We decided to use Locality Sensitive Hashing in combination with the Minhash algorithm.

Locality Sensitive Hashing is a hashing algorithm that tries to place similar input items into the same buckets. We can then check if any buckets contain more than one input item, in which case we deem it a candidate for plagiarism. The algorithm has several input parameters which we will describe later in the section.

### 2.3.1 Minhash

In order to reduce the amount of information we need to store per article, we use the minhash algorithm. The basic idea behind the algorithm is that, instead of storing a matrix consisting of all shingles that appear in all articles and where, we instead create a so-called signature matrix[Ker21]. The signature matrix is a much smaller $K \times N$ matrix created by shuffling all shingles in all N articles in K different ways and for each article and permutation we store the position of the first word that the article contains.

Instead of actually making all these permutations, we instead use hash functions because it is faster. After hashing each shingle, we keep the lowest hash value. This simulates making a permutation and keeping the lowest index.

In order to create our minhash table, we used the Python mmh library. This library is a Python wrapper of the MurmurHash algorithm, created by Austin Appleby[Sen13]. In addition to taking the article as input, MurmurHash also uses a number as seed to determine the result of the hash. Because of this, if we know we need $x$ hashes, we simply run MurmurHash with each number from one up to $x$.

### 2.3.2 LSH

After generating the signature matrix, we could actually use the actual LSH algorithm. The algorithm has two input parameters: the band-width R and the amount of bands B. The product of R and B is supposed to be the size K of the signature matrix. As can be inferred from the input parameters, we have divided the signature matrix into several bands of width R. The goal of the algorithm is to put every article into B buckets. When two articles are placed within the same bucket, they should be sufficiently similar: at least one band of width R must have the exact same values.

By choosing our R and B carefully we are able to fine-tune how similar articles must be in order to be placed within the same bucket. As an extreme example, consider B equal to one, then two articles will only be placed within the same bucket if they are exactly the same. On the other hand, consider B sufficiently large, and R sufficiently small, two articles might end up in the same bucket even if they just contain one similar word.

We can approximate how similar two articles in the same bucket will be using the following formula[Ker21].

$$t = (1/B)^{(1/R)} \tag{2}$$

Once we have generated all buckets, we just needed to check which buckets had more than one member. These members are our candidates for plagiarism. It would be possible to further refine the set of candidates by calculating their Jaccard indexes, which would be feasible since it is a much smaller set, but we found this did not improve the results by a lot.

# 3 Analysis

In order to improve the efficiency of our system, we did various tests with different parameters. The important factors we have to balance are run-time, precision and recall. In order to test precision and recall we first calculated the Jaccard index of every article pair in the small data set. We didn't use shingles for this particular calculation, which might affect the accuracy when comparing the results. We used external tools to plot the resulting Jaccard index values, see Figure 1. The plot omits index values of less than 0.3 since these are not relevant for plagiarism checking and they mess up the y-axis because there are so many.

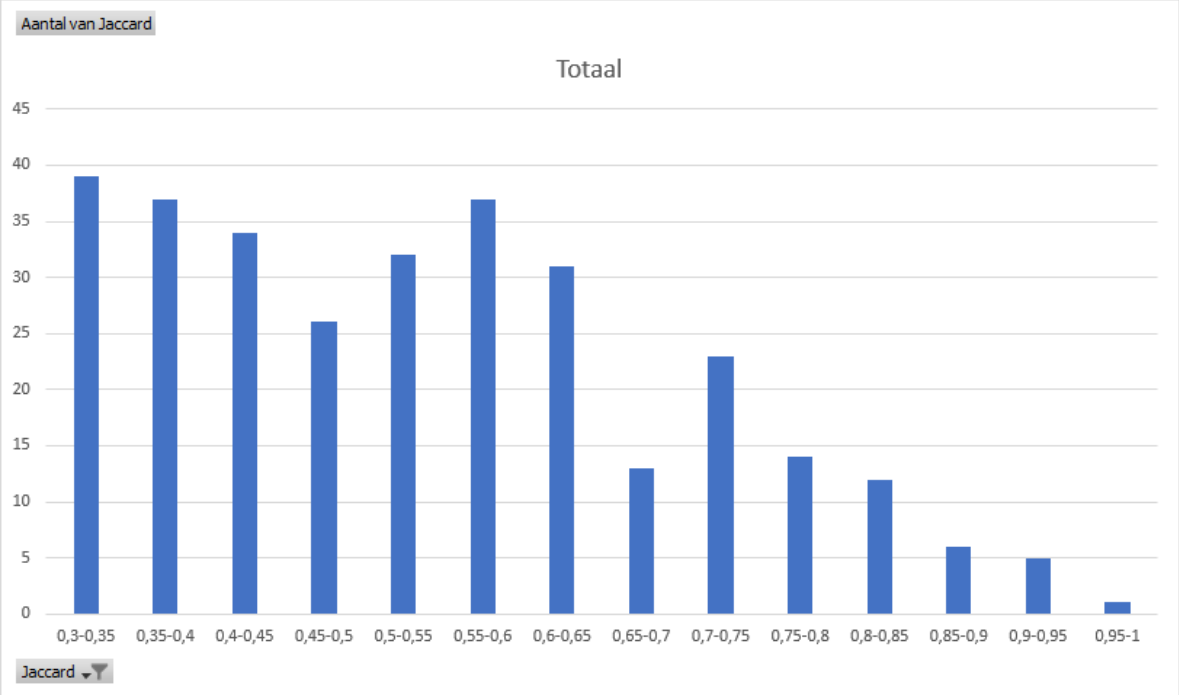


Figure 1: Jaccard Index values

## 3.1 Run-time

First we will test the run-time since it is relatively easy to test. We want the final run-time on the small data set to be reasonably quick, in order to allow us to iterate the other parameters more quickly. The values that will influence the final run-time are the data size (which is fixed) and the amount of permutations used in the LSH algorithm. The amount of permutations is a calculated factor based on the amount of bands B and the amount of permutations per band R. While testing these variations, we will attempt to keep the significance factor as close to 0.8 as possible, using Formula 2 from the previous section. Additionally, the amount of words per shingle has a certain amount of influence on the results, but we chose not to measure it for the purpose of this test.

| K | 10 | 50 | 100 | 200 | 512 | 1000 | 2000 | 5000 |
|---|---|---|---|---|---|---|---|---|
| B | 2 | 6 | 10 | 18 | 32 | 50 | 100 | 200 |
| R | 5 | 8 | 10 | 12 | 16 | 20 | 20 | 25 |
| Time | 1.17s | 4.70s | 9.52s | 20.5s | 45.9s | 91.3s | 183s | 486s |

Table 1: Run-time of different input parameters

As can be seen in Table 1, the runtime goes up as the amount of permutations increases. We would prefer not to take too much time per iteration, so we'll be using a K value somewhere between 200 and 512 for our future tests.

## 3.2 Accuracy

In the previous section we decided to use values of B and R that approximate a Jaccard index of 0.8. It begs the question: is this value optimal. After all, one might argue that it is better to retrieve too many candidates, but still not so many that a manual review becomes impossible, than it is to miss candidates. Additionally, the minhash algorithm contains an amount of randomness, which means that while the average estimate would be 0.8, it might actually miss out on some candidates if we set our boundary too high.

For this section we will be comparing the output of our program to the Jaccard indexes calculated in section 3. We will be comparing which articles have a Jaccard index of 0.75 or higher as calculated and the output of our program for various combinations of B and R. We will also be varying the shingle size between 1, 3 and 5.

We will be using Precision and Recall to measure the accuracy of our results. The definitions are as follows: Precision is the amount of articles of which we know are plagiarised which are retrieved, divided by the total amount of articles which are retrieved. Recall is the amount of articles of which we know are plagiarised which are retrieved, divided by the total amount of articles of which we know are plagiarised.

As mentioned in the first paragraph of this subsection, we will prefer obtaining too many results over too few. This correlates to a high recall. It would obviously be ideal to get both a high recall and a high precision, but in case that proves impossible, we will be prioritising recall.

$$\begin{aligned} Precision &= |Plagiarised \cap Retrieved|/|Retrieved| \\ Recall &= |Plagiarised \cap Retrieved|/|Plagiarised| \end{aligned} \tag{3}$$

| Shingles | 1 | 3 | 5 | 1 | 3 | 5 | 1 | 3 | 5 |
|---|---|---|---|---|---|---|---|---|---|
| K | 250 | 250 | 250 | 200 | 200 | 200 | 320 | 320 | 350 |
| B | 25 | 25 | 25 | 25 | 25 | 25 | 40 | 40 | 50 |
| R | 10 | 10 | 10 | 8 | 8 | 8 | 8 | 8 | 6 |
| Estimated Jaccard Index | 0.72 | 0.72 | 0.72 | 0.67 | 0.67 | 0.67 | 0.63 | 0.63 | 0.52 |
| Precision | 0.50 | 0.87 | 1.0 | 0.39 | 0.68 | 1.0 | 0.32 | 0.70 | 0.52 |
| Recall | 0.94 | 0.36 | 0.11 | 1.0 | 0.53 | 0.13 | 1.0 | 0.64 | 0.81 |

Table 2: Precision and Recall of different parameters

From the results in table 2 it would appear that using one word shingles is optimal. We believe, however, that this might be misleading. It turns out that we used one word shingles in calculating our reference Jaccard indexes. This might influence the calculations we are doing to get our precision and recall, which is not ideal.

It appears that in general, precision goes down while recall goes up as the estimated Jaccard index goes down. For one word shingles, there's not much value in going below an estimated index of 0.72 as the precision keeps going down. On the other hand, five word shingles don't seem to have the recall we want. Three word shingles appear to be the middle ground between recall and precision.

### 3.2.1 Recalculating Jaccard Index

One possibility to increase the precision would be to recalculate the Jaccard index for all pairs that are returned, and to use this index as a second reference when selecting likely candidates for plagiarism.

We are, however, uncertain about this approach. It might potentially increase the run-time when a lot of plagiarism is found, returning us to step one.

# 4 Conclusion

In creating our plagiarism checker, we had to test several approaches. We settled on using Locality Sensitive Hashing along with the minhash algorithm. We did several experiments to find a configuration that suited our needs the best. We think our final result is worth being used.

# References


[Bry17] Hubert Bryłkowski. *Locality sensitive hashing — LSH explained*. 2017. URL: https://medium.com/@hubert_46043/locality-sensitive-hashing-explained-304eb39291e4 (visited on 01/04/2022).

[Ker21] Jonathan Kermes. *Locality Sensitive Hashing: How to Find Similar Items in a Large Set, with Precision*. 2021. URL: https://towardsdatascience.com/locality-sensitive-hashing-how-to-find-similar-items-in-a-large-set-with-precision-d907c52b05fc (visited on 01/03/2022).

[Sen13] Hajime Senuma. *mmh3*. 2013. URL: https://pypi.org/project/mmh3/ (visited on 01/04/2022).

[To21] Statistics How To. *Jaccard Index*. 2021. URL: https://www.statisticshowto.com/jaccard-index/ (visited on 01/03/2022).