UNIVERSITAT POLITÈCNICA DE CATALUNYA

ALGORTIHMS

Detection of similarity between documents with hashing

Roger Vilaseca Darné, Xavier Lacasa Curto and Xavier Martín Ballesteros

10th December 2018

Contents

1 Introduction 2

2 Jaccard Similarity 2

3 Shingling of Documents 2

3.1 k-Shingles 3

4 Compairing Similarity Using The Sets 3

4.1 Matrix Representation 3

4.1.1 Time Complexity 4

4.1.2 Space Complexity 4

4.2 Minhashing 4

4.2.1 Preserving the Jaccard Similarity 5

4.2.2 Optimizing the Time for Permutations 5

4.2.2.1 Hashing Functions 6

4.2.2.1.1 Modular Hashing 6

4.2.2.1.2 Robert Jenkins' 96 bit Mix Function 6

5 Locality Sensitive Hashing (LSH) 7

5.1 Band partition 7

5.2 Choice of b and r values 8

6 Algorithms 8

6.1 Jaccard Similarity 8

6.1.1 k-Shingles 8

6.1.1.1 Cost 10

6.1.2 Minhash Signatures 10

6.1.2.1 Cost 11

6.2 LSH 11

6.2.1 Cost 13

7 Plots 13

7.1 Jaccard Similarity 13

7.1.1 k-Shingles 13

7.1.2 Signature Matrix 16

7.1.2.1 Using Modular Hashing 16

7.1.2.2 Using Robert Jenkins' 96 bit Mix Function 18

7.1.2.3 Modular Hashing vs. Robert Jenkins' 96 bit Mix Function 20

7.2 LSH 22

1 Introduction

In this experiment we are going to create differents algorithms to compute the difference between various documents in order to be able to detect if there may have been a possibility of copy in any pair of the documents. To do it, we have started with a very basic algorithm that returns the Jaccard Similarity with two documents and we have finished obtaining a Locality-sensitive hashing algorithm that given the paths of several documents will return which documents may have been copied. (All the algorithms are explained in the document).

2 Jaccard Similarity

The Jaccard Similarity, also known as Intersection Over Union (IOU), calculates the percentage of similarity between two sets.

For any pair of sets S and T, the Jaccard Similarity is defined as:

$$J(S,T) = \frac{|S \cap T|}{|S \cup T|} \tag{1}$$

We can easily deduce that the more common words, the bigger the Jaccard Similarity, which means that it is more probable that one set is a duplicate of the other.

Example 2.1. *In Figure 1 we see two sets S and T. There are 3 elements in their intersection ("I", "love", "chocolate") and 6 in their union ("I", "love", "chocolate", "and", "pizza", "white"). Thus, J(S, T) = 3/6.*

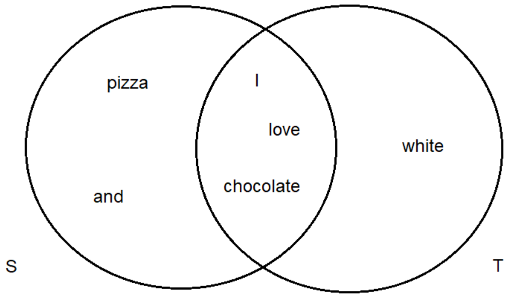


Figure 1: Two sets with Jaccard Similarity 3/6.

3 Shingling of Documents

Any pair of documents can be compared by watching the number of repeated strings they have. The more common strings, the more probable is that one is a duplicate from the other. One way to represent a document as a set is to insert in the set each string that appears in it. If we do so, then duplicated documents that have reorganized the sentences or even the entire text will have plenty of common strings, and will be detected as duplicated.

3.1 k-Shingles

The idea is not to insert in the set all the words, but a set of characters of size k. Thus, each element of the set will have the same size as the others.

The question now is how big k should be? If we take a small value of k, this will result in many shingles that are present in all documents. Suppose we choose the extreme case ($k = 1$). Then, all documents would result to be similar, as the most used characters are present in all documents. However, if we take a big value of k, then any pair of documents would not share a shingle.

The value of k depends on the size of the documents. A poem will not have the same k value than an article. Otherwise, we could have the problems mentioned before. According to Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman (2011), ”k should be picked large enough that the probability of any given shingle appearing in any given document is low.” (p. 78).

4 Compairing Similarity Using The Sets

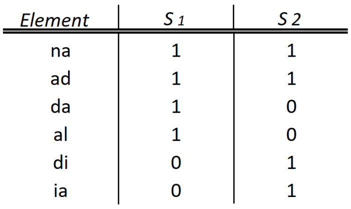
If we succeed in shingling the documents by using the k-Shingling technique, we will only have to compare all pairs of documents using the Jaccard Similarity and say if there is similarity between them or not. In order to do this, we have to store all the information in a data structure, for instance, a matrix. By doing this, we have two problems: time and space complexity.

4.1 Matrix Representation

To represent the matrix, we will put the documents' sets in the columns and the union of all the documents' sets in the rows. The values of the matrix will be the following:

$$\begin{cases} 1, & \text{if column } c \text{ contains row } r \\ 0, & \text{otherwise} \end{cases}$$

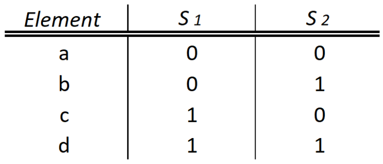
Example 4.1. *For this example we will use two sets representing the words ”Nadal” and ”Nadia”. Let $k = 2$ to form the k-Shingles.*



Element	*S_1*	*S_2*
na	1	1
ad	1	1
da	1	0
al	1	0
di	0	1
ia	0	1

Figure 2: Representation of the matrix with two sets S and T.

In this type of matrices, for any pair of columns we can have 4 types of results, which are the permutations of 0s and 1s of size 2:



Element	*S 1*	*S 2*
a	0	0
b	0	1
c	1	0
d	1	1

Figure 3: Representation of the matrix with all the possible permutations.

Note that as the matrix is sparse, most of the rows will be of type a. If we try to calculate the similarity between two sets S_1 and S_2 using the matrix and the Jaccard Similarity, we will have the following result:

$$J(S_1, S_2) = \frac{Q(d)}{Q(b) + Q(c) + Q(d)} \tag{2}$$

Where Q(x) is the number of rows of type x. Q(d) is the intersection of the sets and Q(b) + Q(c) + Q(d) is the union of the sets.

4.1.1 Time Complexity

Imagine we have n documents. Then, we have to compare each document with all the rest. Thus, the number of comparisons we have to do is $n * (n-1)/2$ which is equal to $O(n^2)$.

Example 4.2. *Suppose we have 1 million documents. The number of comparisons would be* $5 * 10^{11}$ *which is a huge number.*

$$\frac{(1 * 10^6) * 999.999}{2} = 499.999, 5 * 10^6 \approx 5 * 10^{11} \tag{3}$$

4.1.2 Space Complexity

In typical applications the matrix is sparse, which means that there are more 0s than 1s.

If we take k shingles, then the document have relatively few of the possible shingles. Another way to think about this is with the toys in Christmas Day. Kids would be the columns of the matrix and toys, the rows. Usually, kids would like to have a specific toy, which is very popular at that moment. Then, lots of toys would not be buyed for any kid.

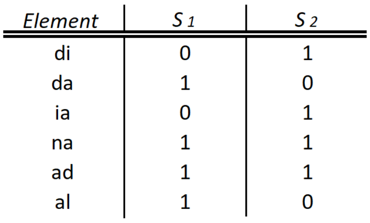
4.2 Minhashing

The main goal using minhashing is to reduce a lot the space complexity. We can achieve this by subsituting the matrix shown before by another matrix called ”signature matrix”.

Signatures are smaller representations of the sets, but they still preserve the similarity of the sets they represent. We will demonstrate this in the next section.

To minhash a set, first pick a random permutation of the rows. Then, the minhash value is the value of the first row that has a 1, preserving the permuted order.

Example 4.3. *In this example we will reuse the two sets of Example 4.1. Suppose that the random permutation has given the following order: "di", "da", "ia", "na", "ad", "al". Let h be the minhash function.*



Element	*S 1*	*S 2*
di	0	1
da	1	0
ia	0	1
na	1	1
ad	1	1
al	1	0

Figure 4: Permutation of the matrix of Example 4.1.

In the first column, we can see that $h(S_1) =$ "da" and in the second one we see that $h(S_2) =$ "di".

With this technique, we can see that every time we do a permutation, we only occupy one new row of the "signature matrix", which is reducing a lot the space.

4.2.1 Preserving the Jaccard Similarity

As we mentioned before, the Jaccard Similarity in the matrix is equal to the number of rows of type d divided by the number of rows of type $b + c + d$. And that Similarity is preserved in the "signature matrix".

Proof. Look down through the permuted columns C_1 and C_2 until we see a 1 in any of the two columns. Then, we can have a row of type b or c, where only one of the columns have a 1, or a row of type d, where both columns have a 1 in it. If we find a row of type d, then the minhash function will take the same row. Thus, $h(C_1) = h(C_2)$. Otherwise, we must have a row of type b or c and $h(C_1) \neq h(C_2)$.

We can see that this is exactly the Jaccard Similarity. The probability of two columns have the same minhash value is equal to the number of rows of type d divided by the number of rows of type $b + c + d$.

$$P[h(C_1) = h(C_2)] = J(C_1, C_2) \tag{4}$$

□

4.2.2 Optimizing the Time for Permutations

Doing a set of permutation would be very expensive, but we can use a trick to simulate the n permutations. We can use n random hash functions that maps row numbers to as many buckets as the size of the rows of the matrix. We say that this is a trick because we are not doing a literal permutation, as we can have collisions using the hash functions but the point is that the probability of a collision with a large number k for the shingling is very small. Thus, we can think of this new algorithm as a perfect permutation of the rows. The algorithm is the following:

```
for each row $r$ do
```

```
    for each hash function $h_i$ do
        compute $h_i(r)$;
    end for
end for
for each column $c$ do
    if $c$ has 1 in row $r$ then
        for each hash function $h_i$ do
            if $h_i(r)$ is smaller than $M(i,c)$ then
                $M(i,c) := h_I(r)$;
            end if
        end for
    end if
end for
```

Every time we go to the next row, we first compute all the hash values using the row value as their input. Then, for each position in that row, if there is a 1 we compare the hashed value of that position with the value in the signature matrix for all the hash functions computed. In case that this new value is smaller we change the value of the signature matrix by this new one. Doing this, in the end we have the smallest possible number in each position of the signature matrix.

4.2.2.1 Hashing Functions

We will present you now all the types of hashing functions we have used in the previous algorithm and in the algorithms in the Section 6.

4.2.2.1.1 Modular Hashing

In this case we will recieve a key k, a prime number p, a random number modulus the number of shingles m and the number of k-shingles z. The result will be a number modulus the number of k-shingles of the document.

modularHashFunction(k, p, m, z) = (k * p + m) mod z

```
Require: Four integers $k, p, m, z$
Ensure: An integer $h$
  return $(k * p + m)$ mod $z$
```

4.2.2.1.2 Robert Jenkins' 96 bit Mix Function

In this case we will recieve a prime number p, a random number modulus the number of shingles m and a key k. We will compute the returning value h, that will be found making a sequence of subtraction, exclusive-or, and bit shift.

mix(p, m, k) = h

5 Locality Sensitive Hashing (LSH)

We have talked before about how to improve the space complexity, but we still have not introduced a technique to improve the time complexity.

The general idea is that only a few pair of documents will have a high Jaccard Similarity. Thus, we need an algorithm that chooses the candidates that are more likely to be similar. The Locality Sensitive Hashing will help on this a lot.

The approach of LSH is to hash groups of rows into buckets in the sense that if two or more documents are hashed into the same bucket for at least one time, they will be very likely to be similar and will be put as a candidate pair.

5.1 Band partition

LSH will divide the signature matrix into b bands of r rows per band. Then, the algorithm will hash the set of rows that form a band for all the documents. Finally, it will create a new pair of candidates for every pair of documents hashed into the same bucket (for at least one time).

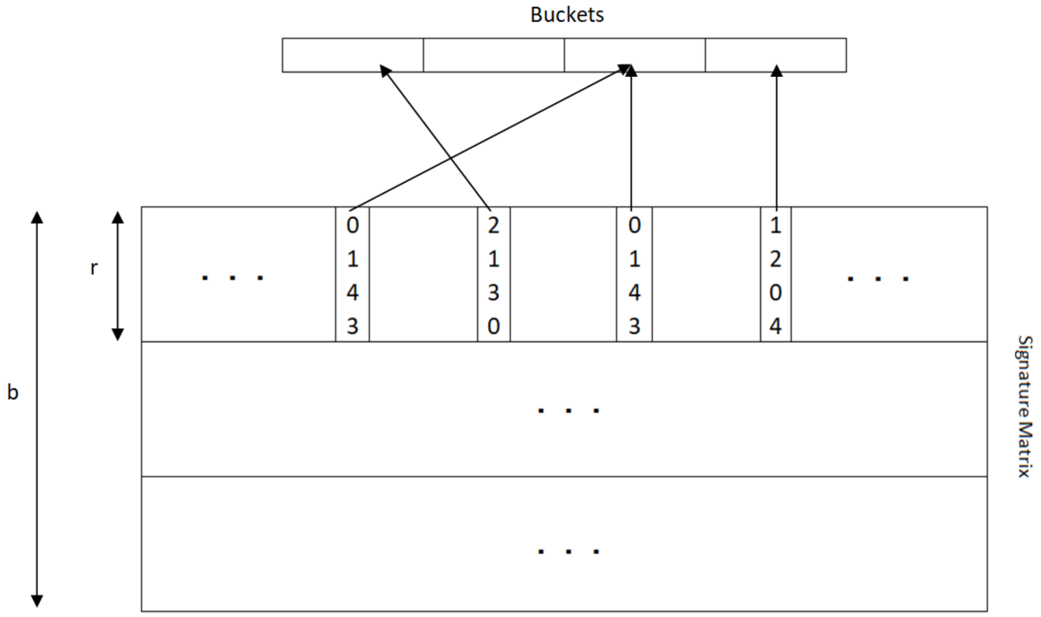


Figure 5: Band partition of a signature matrix and hashing the values of the rows of the first band int othe buckets.

We can see that the set of rows of value ”0143”, from top to bottom, are hashed into the same bucket, and will be put as a candidate pair. However, the other two set of rows (”2130” and ”1204”) will not be hashed into the same bucket, and will not be put as a candidate pair (with this band).

5.2 Choice of b and r values

The election of b and r values is a very important part of the algorithm. See that the bigger r, the less probable is to hash two sets of rows into the same bucket. This is because the probability that two sets of documents hashes into the same bucket decreases exponentialy each time we add one row into r.

We want to get the least number of dissimilar documents that are hashed into the same bucket. These are called *false positives*. On the other hand, we also want to minimize the number of similar documents that are hashed into different buckets. These other ones are called *false negatives*.

The idea is to select a threshold, which is the minimum value that the Jaccard Similarity can have to consider two documents as similar, and then compute the b and r values.

According to Anand Rajaraman, Jure Leskovec, and Jeffrey D. Ullman (2011), ”a good aproximation to the threshold is $(1/b)^{1/r}$” (p. 90). So if we take a threshold t, then we can calculate the b and r values as following:

Having that r = nhashFunctions/b,

$$t \approx (\frac{1}{b})^{\frac{1}{r}} = (\frac{1}{b})^{\frac{b}{nhashFunctions}} \tag{5}$$

Thus, using an arbitrary threshold, we can calculate the best approximate value of b and r.

6 Algorithms

The objective in this section is to show the idea behind the algorithms we have programmed and give you the time complexity for each one. If you want to see the code, you will have to go to the algorithms folder.

6.1 Jaccard Similarity

The main objective in this section is to calculate the Jaccard Similarity between two documents in different ways.

6.1.1 k-Shingles

The idea is to create two sets, one per document, and insert all substrings of size k of the documents. Afterwards, we will just have two make a division: the number of shingles in the intersection divided by the number of shingles in the union.

For each document:

```
**Require:** $k$
**Ensure:** Returns an unordered_set with all the substrings of size k of the document
  $words$ := entire document
  $pos$ := 0
  unordered_set $S$ := $\emptyset$
  **while** $pos + k <= words.size()$ **do**
```

```
        sub := substring from pos to pos + (k - 1)
        insert sub into S
    end while
    return S
```

Once we have calculated the k-Shingles, we need to compute the intersection and the union of the sets. Note that we insert the substrings in an unordered set. This will be very usefull to improve the time complexity in the next two algorithms:

```
Require: Two sets $S_1$ and $S_2$
Ensure: Returns the intersection set between $S_1$ and $S_2$
    unordered_set intersection := ∅
    for each element in $S_1$ do
        if $S_2$ contains the element in $S_1$ then
            insert the element into intersection
        end if
    end for
    return intersection
```

```
Require: Two sets $S_1$ and $S_2$
Ensure: Returns the union set between $S_1$ and $S_2$
    unordered_set union := $S_1$
    for each element in $S_2$ do
        if the element in $S_1$ is not contained in $S_1$ then
            insert the element into union
        end if
    end for
    return union
```

As you can see, we visit only one set in each algorithm. The good thing is that finding if an element belongs to an unordered set or not is $O(1)$. Thus, the time complexity for these two algorithms is $O(n)$, where n is the size of the smallest unordered set[1], and the total time complexity is $O(n)$, as inserting elements in an unordered set is $O(1)$.

On the other hand, if we would have used the predefined functions *set_intersection* and *set_union*, we would have needed two ordered set, as it is a precondition of these two functions. Thus, the total cost would have been $O(n * log(n))$.

Finally, we just have to divide the size of the intersection set by the size of the union set (and multiply by 100 if we want a percentage).

$$Jsim(D_1, D_2) = \frac{intersection.size()}{union.size()} * 100 \tag{6}$$

[1] Note that we can change the set we are visiting by the other one. In the intersection, if S_2 is smaller than S_1, we can visit S_2. In the union, if S_2 is bigger than S_1, we can match the unordered set with S_2 and visit S_1.

6.1.1.1 Cost

The cost of the first algorithm is $O(k*(t-k))$, where k is the k-Shingle value and t is the size of *words*. As k is always a constant, the final cost of this algorithm is $O(k*t-k^2) = O(t)$. The cost of the next two algorithms are $O(n)$, as we have said in the previous section. Finally, the cost of a division and a multiplication is $O(1)$.

Thus, the total cost of calculating the Jaccard Similarity using only k-Shingles is $O(n)+O(t)+O(1) = O(n)$, as usually the number of shingles is much larger than the size of *words*.

6.1.2 Minhash Signatures

We know that implementing just k-Shingling is very expensive in terms of size and time (if we want to compare n documents between them, the complexity is $O(n^2)$). Implementing minhash signatures will help on this a lot. However, the Jaccard Similarity will not be the exact value. To do this, we will use as our input the k-Shingle sets calculated in the previous section.

```
Require: Two sets $S_1$ and $S_2$
Ensure: Returns an approximate Jaccard Similarity value between $S_1$ and $S_2$
  unordered_set union = $S_1 U S_2$
  matrix signatures = infinity
  vector h = all the hash functions we will use
  for each row $r$ do
      for each hash function $h[i]$ do
          compute $h[i](r)$;
      end for
      for each column $c$ do
          if $c$ has 1 in row $r$ then
              for each hash function $h[i]$ do
                  if $h[i](r)$ is smaller than $signatures[i][c]$ then
                      $signatures[i][c] := h[i](r)$;
                  end if
              end for
          end if
      end for
  end for
  intersection := 0
  doc1 := 0
  doc2 := 1
  for each hash function $h[i]$ do
      if $signatures[i][doc1] == signatures[i][doc2]$ then
          if $signatures[i][doc1]! = infinity$ then
              ++intersection
          end if
      end if
  end for
```

```
return $\frac{intersection}{h.size()}$
```

We use the hash functions that we introduced in section 4.2.2.1. We also have tried to use MurmurHash and Multiplicative Hash as hashing functions but we have had some problem when using them.

6.1.2.1 Cost

We can deduce by just watching the code that the time complexity will come from the first for loop, as it is the one that has to compute most things. Inside the loop, we have two more loops:

1. This loop is responsible of calculating the hash functions taking as the input the row number. Let h be the number of hash functions we are using. Then, the cost is $O(h)$ because the calculus done inside this loop is constant.
2. This other loop is the one that updates (or not) the signature matrix. For every column c:
   - Watch if the set representing the document has the shingle of the row r. The cost of doing this is $O(1)$.
   - Compare the results of all the values computed in the previous for. If any value is smaller than the one that is in the signature matrix at position $[h][c]$, just replace it by the new value. The cost of this for is $O(c * h)$, where c is the number of documents and h is the number of hash functions.

Thus, the cost of this function is $O(r*(h+c*(h+1))) = O(r*h+r*c*h+r*c) = O(r*c*h)$, where r is the number of rows and h and c the values explained before. We should point that h is a constant value (usually 200 is correct), so the real cost is $O(r*c)$.

6.2 LSH

The first thing we need to do in this algorithm is finding the right values for b, number of bands, and r, number of rows per band depending on the value of a *threshold*. To do so, we need an algorithm that calculates the b value [2] that has the minimum error with the threshold. The error is calculated as the function (5) minus the *threshold*.

```
Require: nhashFunctions = number of rows of the signature matrix, and a threshold
Ensure: Returns the $b$ value
  b := 1
  minError := 1.0
```

[2] It is also possible to calculate the r value and later divide *nhashFunctions* by r to get the b value, but it is more difficult to calculate it.

```
for each row $h$ of SM do
    if nhashFunctions mod h == 0 then
        aux := $(1/h)^{(h/nhashFunctions)}$
        absolute := $|aux - threshold|$
        if absolute < minError then
            minError := absolute
            b := h
        end if
    end if
end for
return $b$
```

Once we have calculated the *b* value, we just need to divide *nhashFunctions* by *b* to get the number of rows per band.

Now that we have the best possible values for *b* and *r*, it is time to begin with the LSH algorithm. This algorithm will hash each set of rows from each band and if two set of rows from the same band hashes to the same bucket, both documents will be inserted in a data structure of possible candidates.

```
Require: The signature matrix of the $c$ documents we have SM, $b$ and $r$
Ensure: Returns the condidate pairs
  unordered_map buckets := Ø
  unordered_map candidates := Ø
  for each band $b$ do
      for each document $c$ do
          hv := hash value of SM[b][c]
          buckets[hv].insert(c)
      end for
      for each element $val$ of buckets do
          for each element $d$ of buckets[val] do
              j := 1
              while position of $d$ + j < buckets[val].size() do
                  elem := element of buckets[val] in position ($d$ + j)
                  if absolute < minError then
                      insert elem into candidates[d]
                  else
                      insert $d$ into candidates[elem]
                  end if
              end while
          end for
      end for
      buckets.clear()
  end for
  return $candidates$
```

At this moment, we have the list of pairs of candidates that we want with the threshold *t*.

The last thing we have to do is calculate the Jaccard Similarity between all the pairs of candidates of the list. As we have done an aproximation when calculating the values b and r, it is very probable that some pair of candidates have less Jaccard Similarity than t.

6.2.1 Cost

The LSH algorithm has two parts. The first one calculates the best possible values of b and r. The second one chooses the candidate pairs of documents.

1. The cost of this part is $O(h)$, where h is the total number of hash functions used in the signature matrix (h is the number of rows of the signature matrix).

2. Let c be the number of documents and b the number of bands. The cost of this part of the algorithm is $O(c * b^2)$. For each band b:

   - For each document compute the hash value of each set of rows of band b. The cost of doing this is $O(c)$, as the computational cost of calculating the hash values is $O(1)$.

   - For each non-empty position of the buckets, create pairs of candidates between all the documents of the bucket without repetitions. This means that documents 3 and 4 will only have one candidate pair (3, 4), and not two (3, 4) and (4, 3). The cost of this for loop is $O(c^2)$, as in the worst case all the documents will be in the same bucket and we will have to make $(c * (c - 1))/2$ candidate pairs.

7 Plots

All plots have been done using Python. For the experiments, we have choose random pairs of documents of 50 words permuted randomly. Each plot has been done using the same 5 pairs of documents (chosen randomly) as samples.

7.1 Jaccard Similarity

7.1.1 k-Shingles

The following plot shows the evolution of the Jaccard Similarity when we vary the value of k.

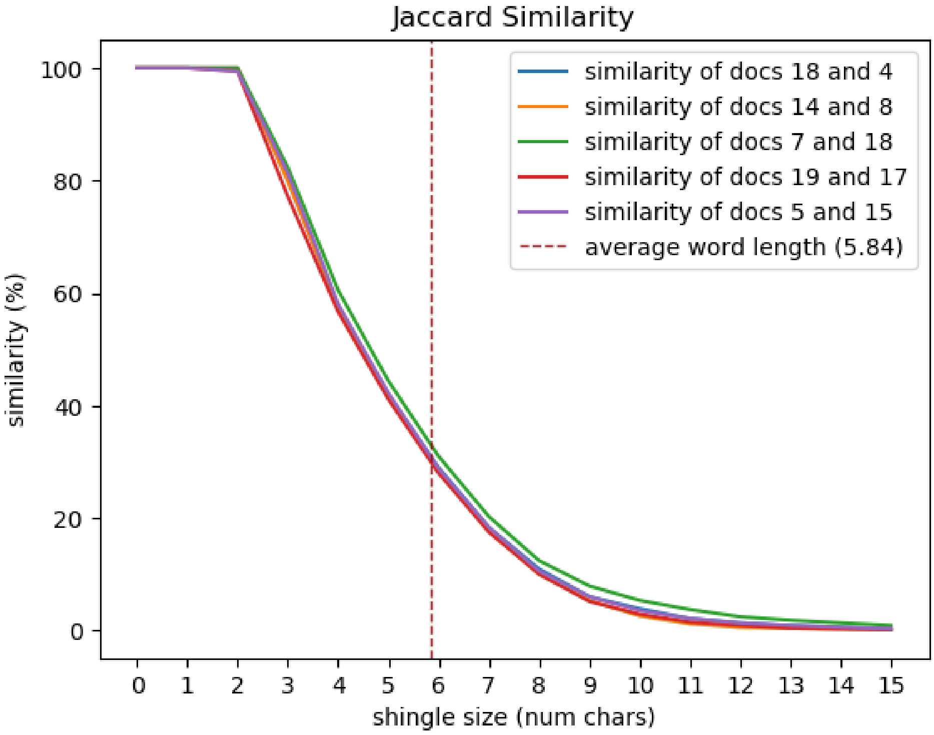


Figure 6: Jaccard Similarity varying the k value.

As you can see, the Jaccard Similarity of the different pairs is very similar because all the documents are a random permutation of the same 50 words. You can see that because when the shingle size is equal to 1 (we only take the characters), the Jaccard Similarity is 100%[3].

Besides, it is important to remark the direction of the graphic. The bigger the shingle size, the lower Jaccard Similarity we get. This is because every time we increase the k value, it is more likely that the substrings start to take some characters of the next word. Thus, the probability that two different documents have the same shingles decreases a lot. You can see that when the shingle size is bigger or equal than 6, we begin to have a very low Jaccard Similarity (less than 20 from $k = 7$) because most of the substrings will have characters of the next words.

Now, we are going to represent the time cost of the same pair of documents when we vary the shingle size.

If we focus on the time cost we can see that the time does not vary at all when we use different shingle sizes. This may be because the documents have the same number of words. Thus, the number of shingles for each document must be very similar to the others.

We should also point that the time to compute the Jaccard Similarity of these pairs of documents is very small because the number of words is very small (50 words are very few compared to the number of words of an article).

[3] Note that in a document with few words, it is very probable that not all the characters are included in the document.

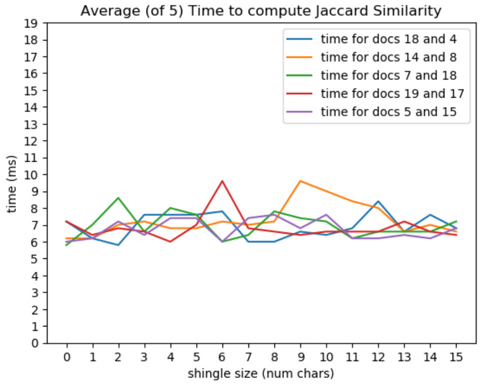


Figure 7: Time to compute the Jaccard Similarity when varying the k value.

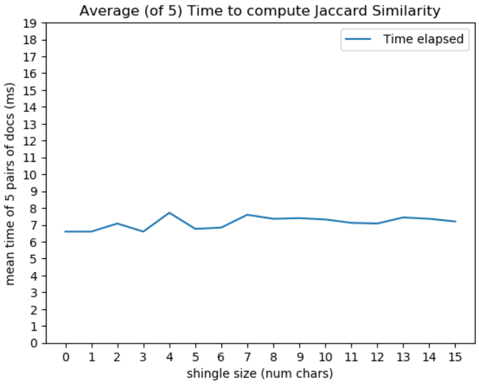


Figure 8: Average time to compute the Jaccard Similarity of the Figure 7.

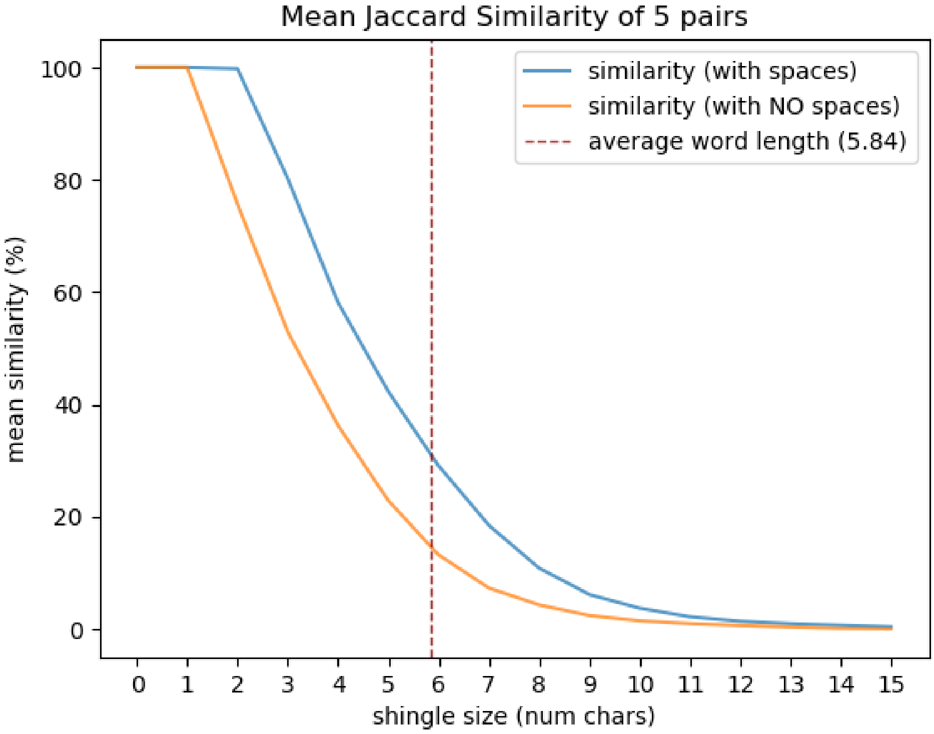


Figure 9: Time to compute the Jaccard Similarity when varying the k value.

In the following plot we can see the differences between the behaviour of the algorithm taking into account the spaces or not. The blue line (coutning spaces) shows that the Jaccard Similarity calculated counting the spaces in the documents is higher than not counting them (orange line). This is due to the difference between the k-shingles created in both algorithms, because if we consider the spaces then we will need a higher k for the substring to start taking characters from the next word.

From now on, we will use spaces to experiment with the documents in the following plots.

We also have put the option to make all characters to lowercase or not. This does not affect on these documents, as all the characters are already in lowercase.

7.1.2 Signature Matrix

7.1.2.1 Using Modular Hashing

In the following plot, it is represented an approximation of the result of calculating the Jaccard Similarity using the signature matrix.

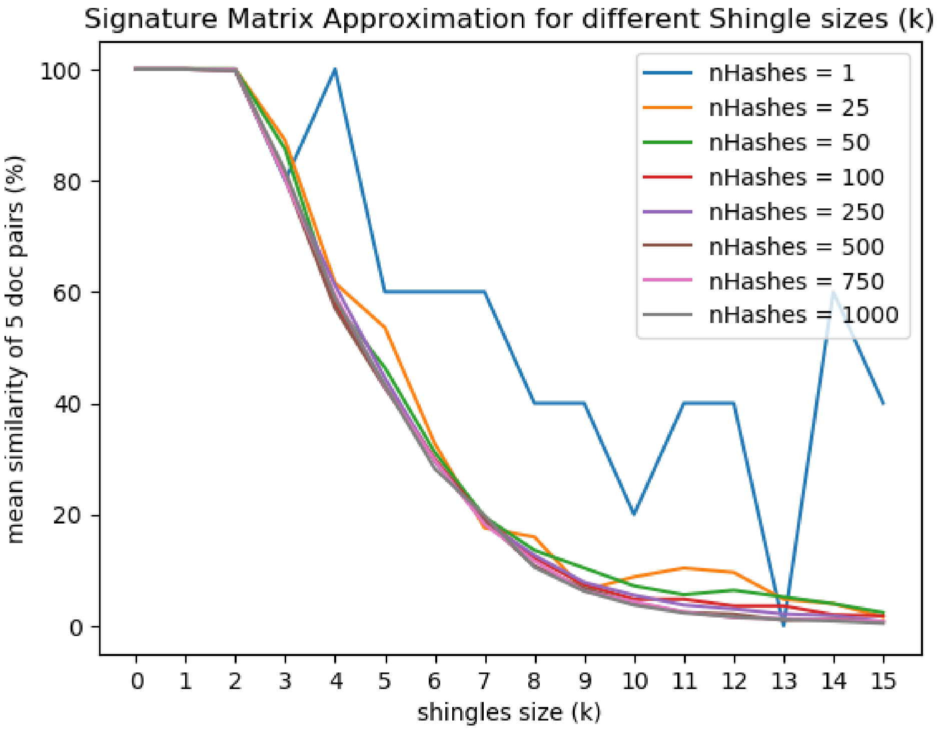


Figure 10: Jaccard Similarity aproximation varying the shingle sizes.

The similarity between the documents is strictly related with the number of hashing functions that we use. When the number of hashing functions is a small value it is very different than the Jaccard Similarity, because the signature matrix has few rows and then, there are few chances that one row has the same hash values on two different columns. But when the number of hash functions increases, the functions are more alike to the Jaccard Similarity (Figure 6).

See that when the nHashes is 1, the Jaccard Similarity should only be 100 or 0 per cent, as there is only one row in the signature matrix, but remember that this plot is using the mean value of the result of computing the Jaccard Similarity of 5 pairs using the signature matrix.

Now, we are going to focus on the number of hash functions.

We can see in Figure 11 that the plot form begins to stay stable from number of hash functions equal to 200 more or less. Furthermore, in Figure 12 the time cost starts to increase a lot from number of hash function equal to 250.

Thus, even though using more number of hash function would be better to calculate an approximation of the Jaccard Similarity, it is not usefull to do so because with 200 hash functions we compute a very similar value in less time.

It is important to remark that in Figure 12 the time stabilize near the k equal to the average word length.

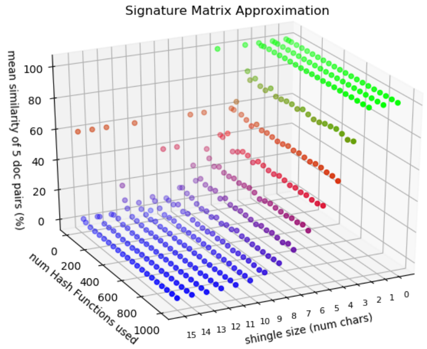


Figure 11: Jaccard Similarity approximation varying the number of hash function and the size of the shingles.

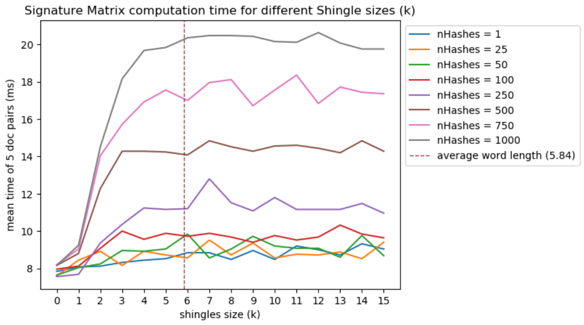


Figure 12: Average time to compute the Jaccard Similarity approximation varying the shingle sizes.

In Figures 13, 14 and 15 there is represented how much is the error value obtained using the Jaccard Similarity and the Signature Matrix Approximation. Approximately we can arrive to the same conclusions as the ones that are before. The more number of hash functions, the less the error is.

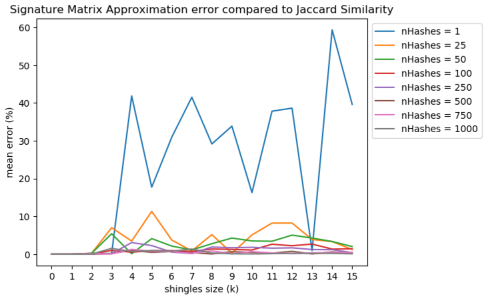


Figure 1: Figure 13: Error between the Jaccard Similarity and the Jaccard Similarity approximated in the signature matrix.

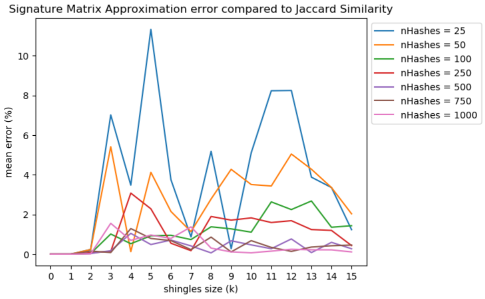


Figure 14: The same than Figure 13 but starting from nHashes = 25.

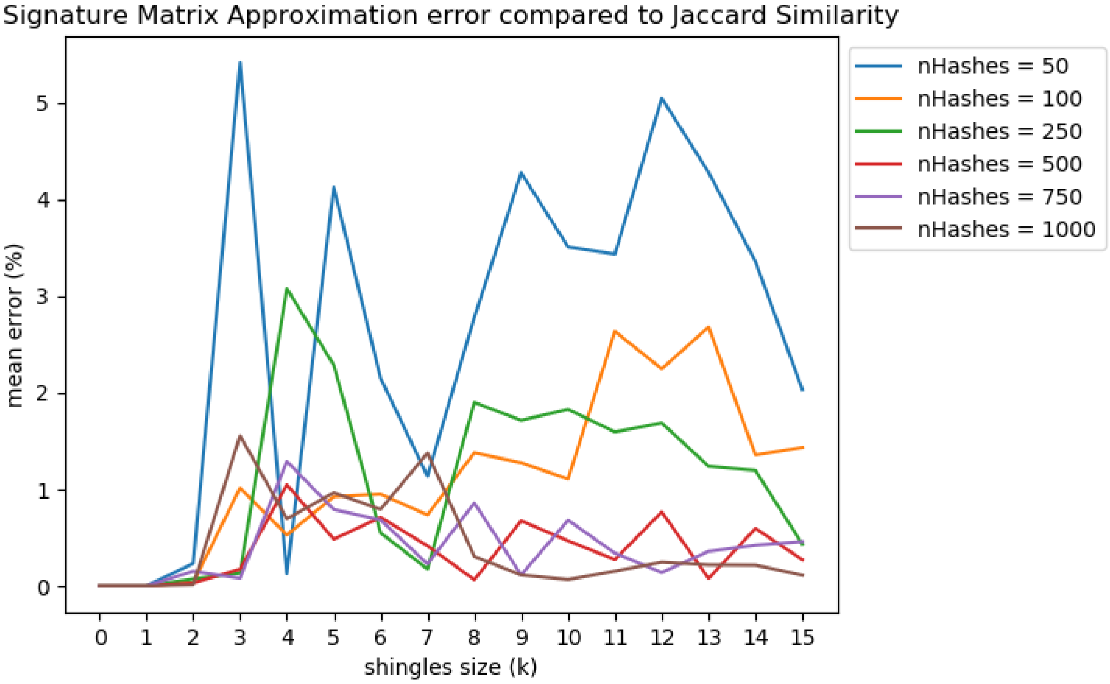


Figure 15: The same than Figure 13 but starting from nHashes = 50.

We have put the same plot three times so that you can see more closely the values of the mean error.

7.1.2.2 Using Robert Jenkins' 96 bit Mix Function

We will show the same plots shown in the previous section. If we do not say anything about any plot, it means that it has the same description as the previous section.

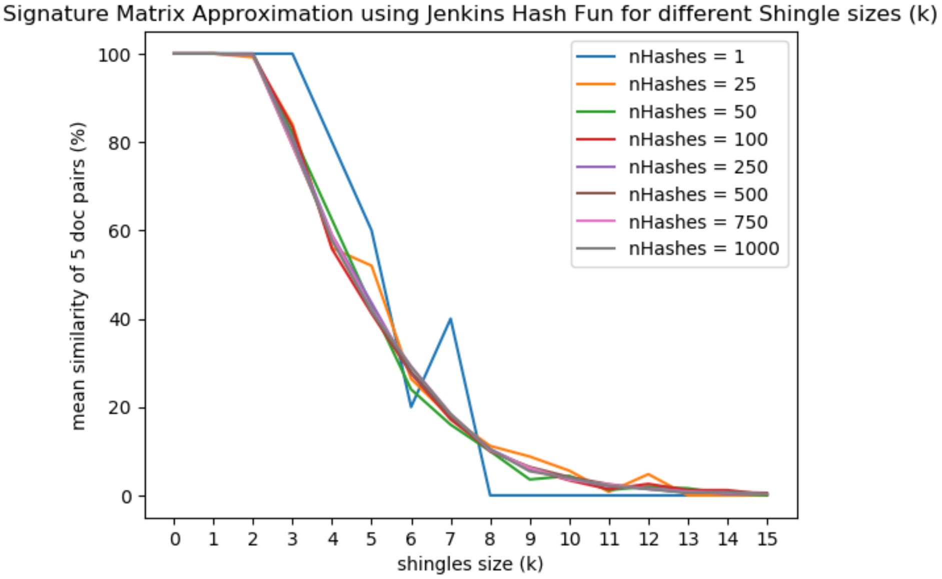


Figure 16: Jaccard Similarity aproximation varying the k value.

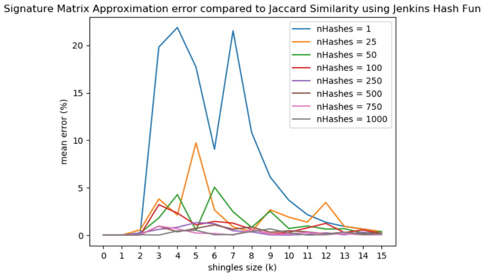


Figure 17: Time to compute the Jaccard Similarity aproximation when varying the k value.

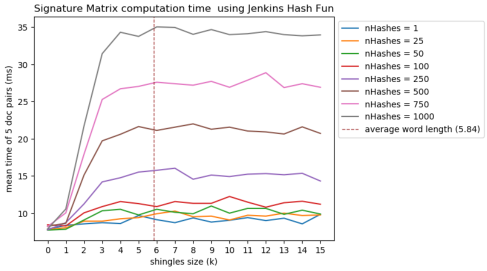


Figure 18: Average time to compute the Jaccard Similarity aproximation varying the shingles sizes.

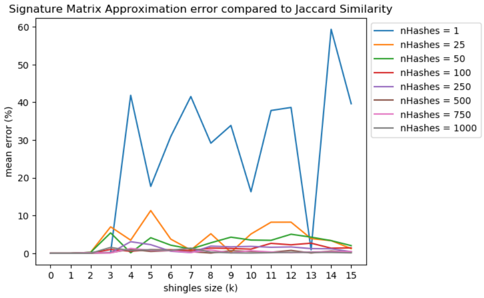


Figure 19: Error between the Jaccard Similarity and the Jaccard Similarity approximated in the signature matrix.

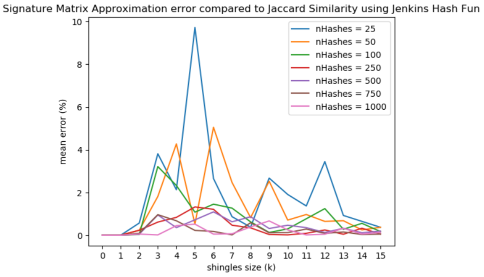


Figure 20: The same than Figure 19 but starting from nHashes = 25.

The only difference we should say is that in the first plot, whe the number of hash functions is 1, the function is closer to the rest of number of hash functions, which is what we want.

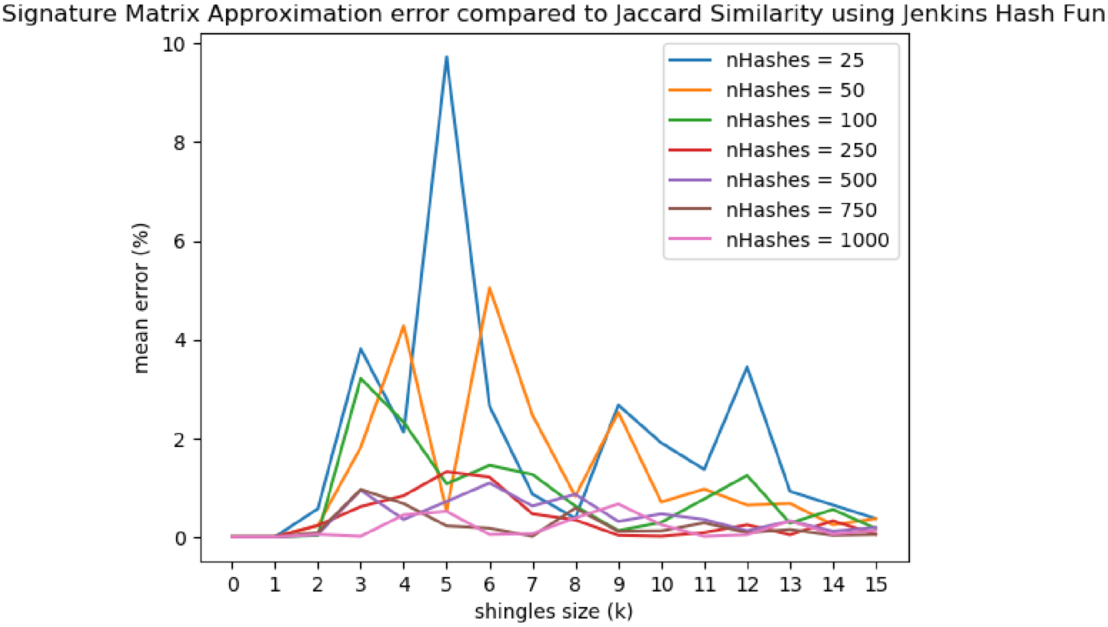


Figure 21: The same than Figure 19 but starting from nHashes = 50.

7.1.2.3 Modular Hashing vs. Robert Jenkins' 96 bit Mix Function

The following plots represent the error using the modular hashing minus the error using the jenkins hashing. If the line is in the positives Y, it means that the error using the modular hashing is greater than the error using the jenkins hashing, and viceversa.

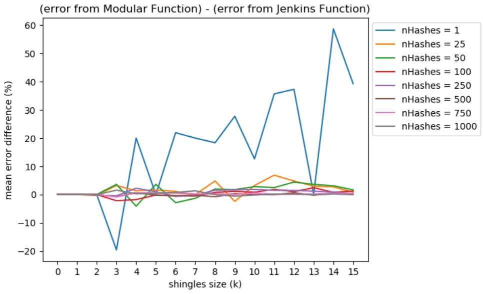


Figure 22: Diference between the error using the modular hashing and the one using the jenkins hashing.

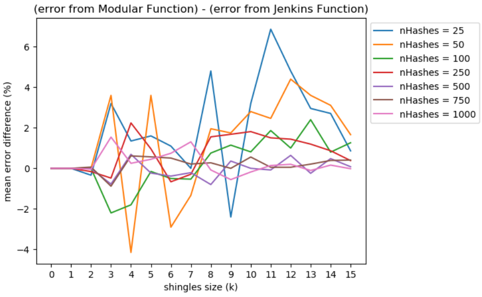


Figure 23: The same than Figure 22 but starting from nHashes = 25.

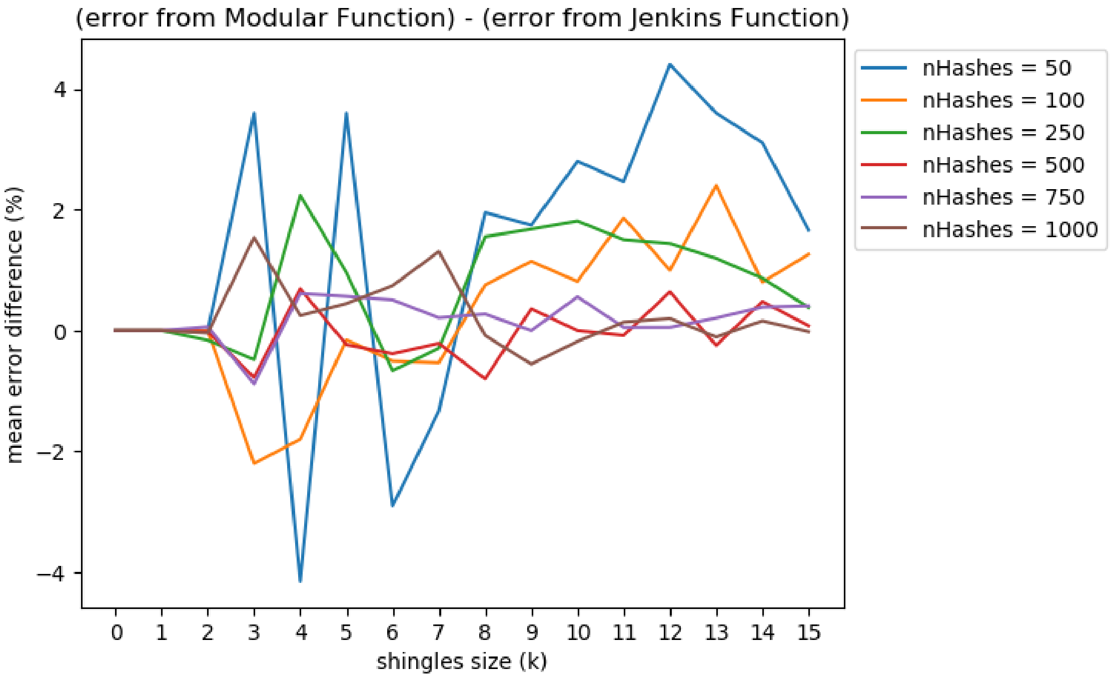


Figure 24: The same than Figure 22 but starting from nHashes = 50.

We can see that the error using the modular hashing in Figures 22, 23 and 24 is greater most of the time, specially at the begining. Thus, the Jaccard Similarity approximation using the jenkins hashing is better than the one using the modular hashing.

In the Figure 25 we have represented the time cost using modular and jenkins hashing. If we are in the positive Y, the modular functions spends more time on the algorithm. But we see that it is the contrary: the jenkins hashing is the one that spends more time even though it has less error than the modular hashing.

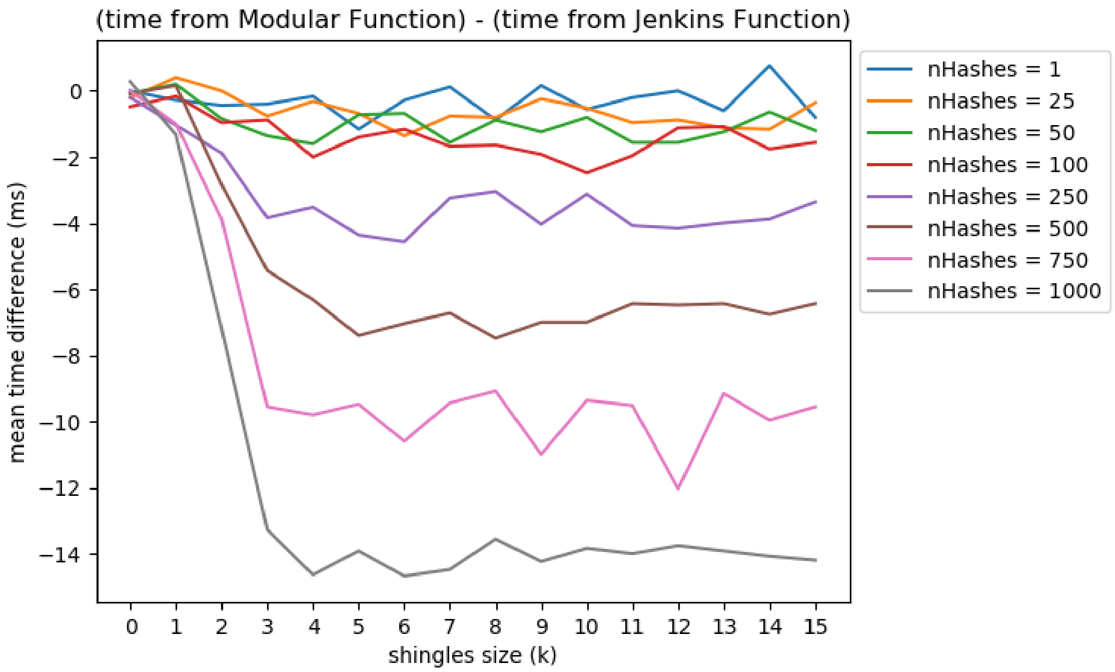


Figure 25: Difference of the time between the modular function and the jenkins function.

7.2 LSH

For these set of plots, we are going to use 500 hash function, even tough we know that 200 would be enough.

The following plot shows the mean time spent running the LSH algorithm and also the time that we need to compute all the candidate pairs without using LSH.

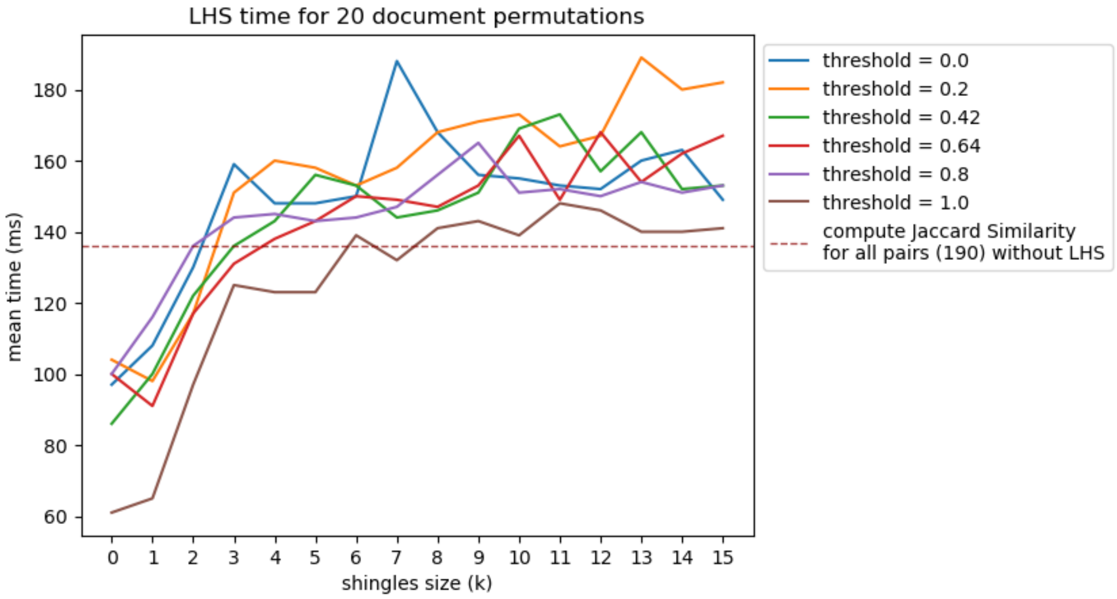


Figure 26: Difference of the time between the modular function and the jenkins function.

The ideal plot would be the one where all the lines are below the discontinuous one. Then, using any value for the threshold, we could compute the copied pairs using LSH faster than without using it.

However, we see that this is not happening in the plot. This is because 20 documents are not big enough to be faster using LSH.

Following this, there are plots shown to ilustrate the LSH algorithm performance according to various thresholds.

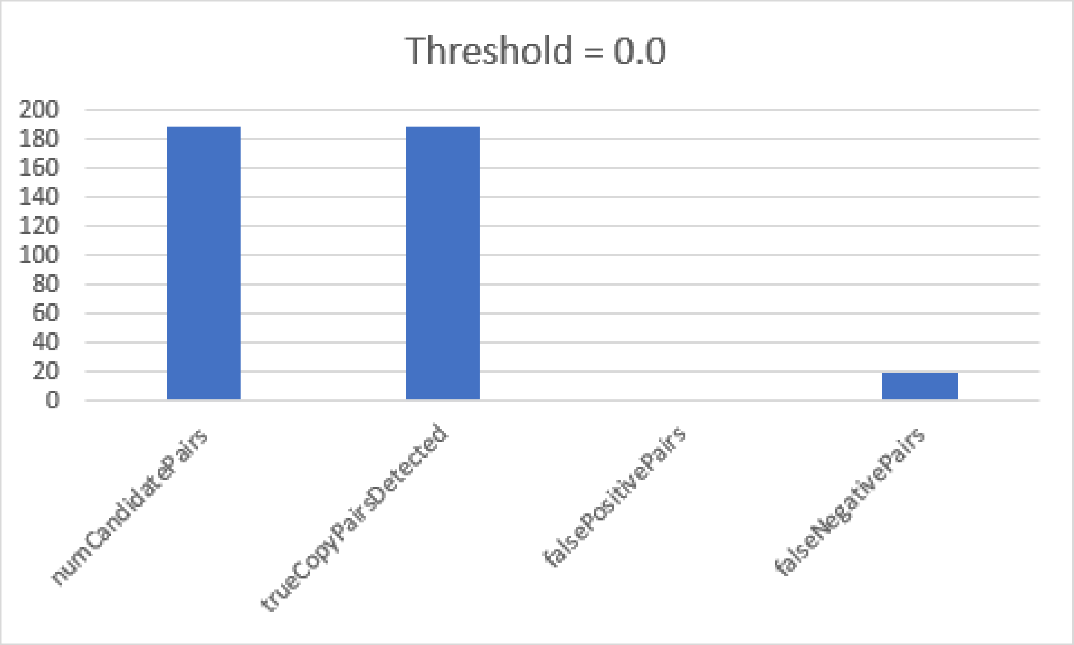


Figure 27: Average number of candidate pairs, number of copy pairs detected, false positives and false negatives with threshold = 0.0.

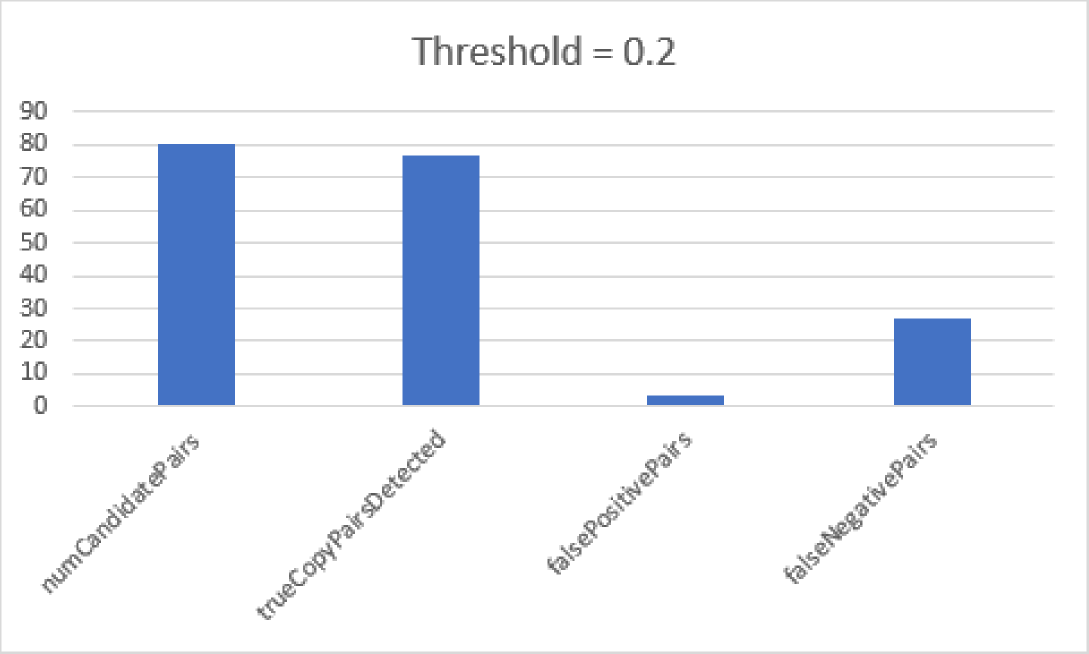


Figure 28: Average number of candidate pairs, number of copy pairs detected, false positives and false negatives with threshold = 0.2.

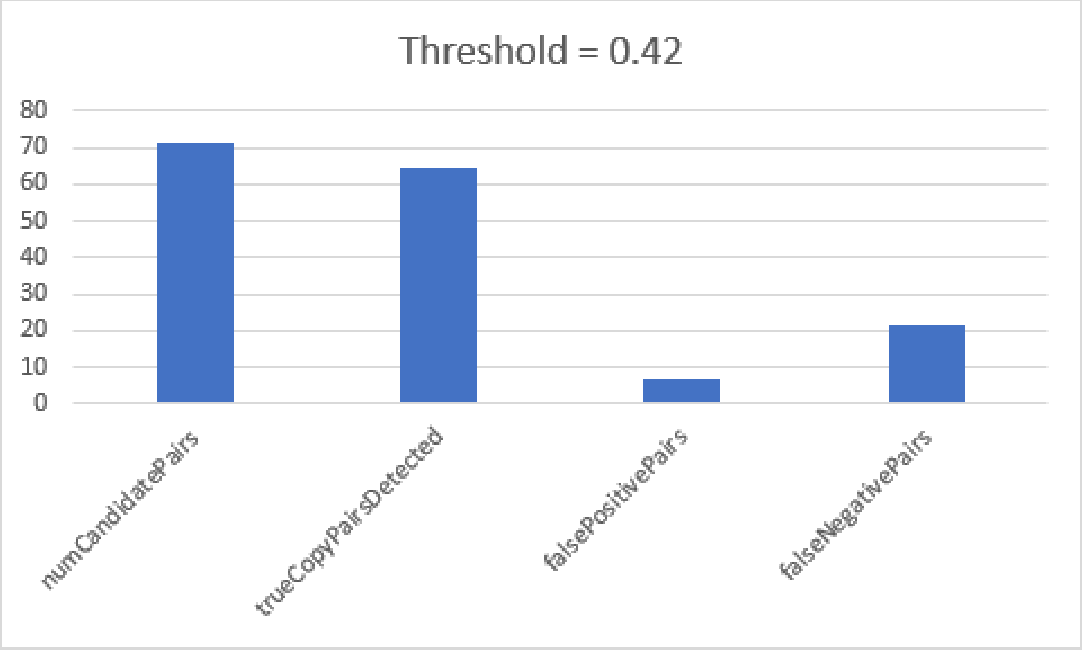


Figure 29: Average number of candidate pairs, number of copy pairs detected, false positives and false negatives with threshold = 0.42.

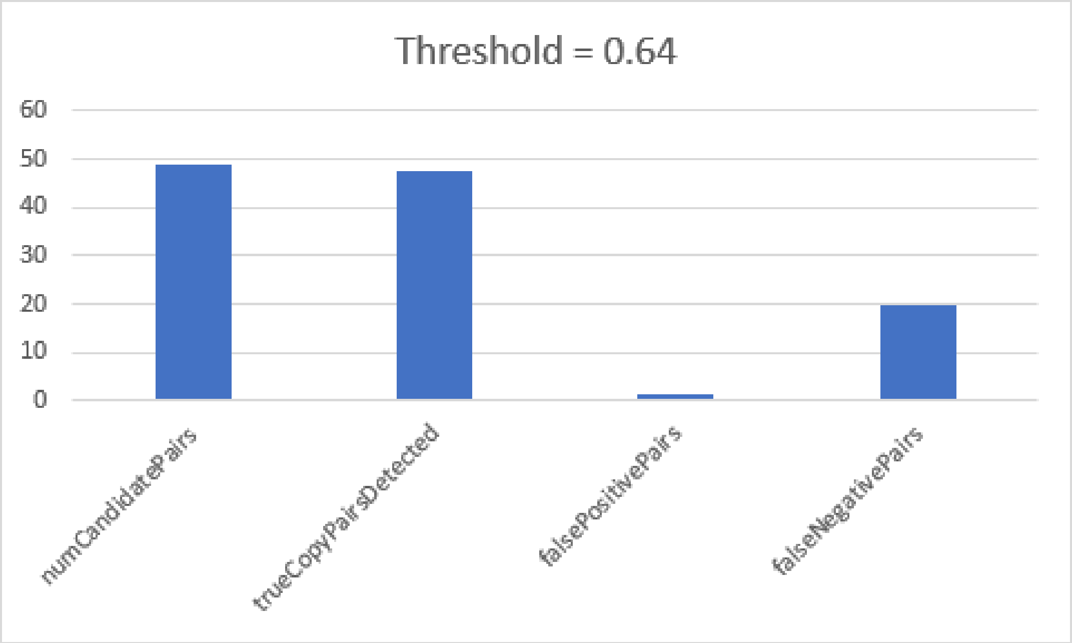


Figure 30: Average number of candidate pairs, number of copy pairs detected, false positives and false negatives with threshold = 0.64.

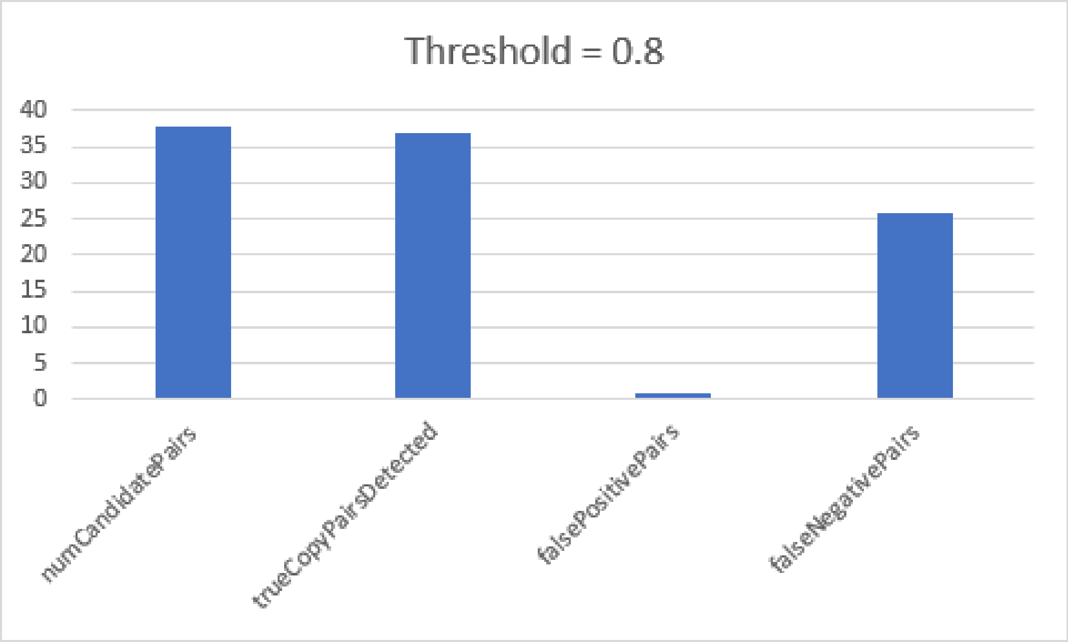


Figure 31: Average number of candidate pairs, number of copy pairs detected, false positives and false negatives with threshold = 0.80.

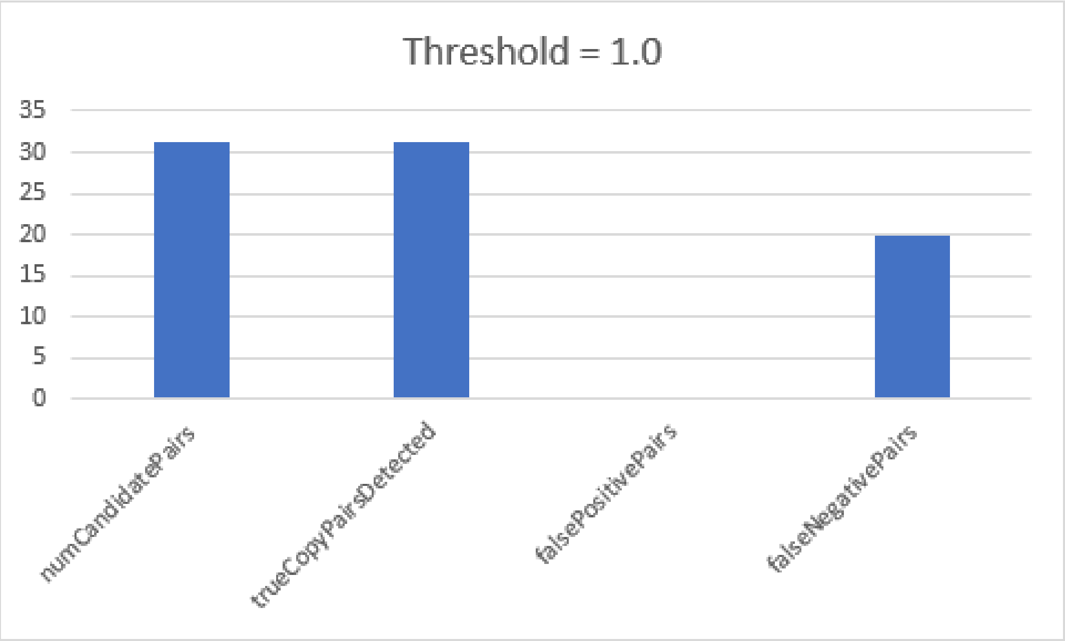


Figure 32: Average number of candidate pairs, number of copy pairs detected, false positives and false negatives with threshold = 1.0.

We can see that the higher threshold value, the lower the shingle size has to be, in order for pairs to be detected as copies. In general, we start to see a decrease in copies detected from shingle size = 5, which could be related to the fact that the average word size is 5.84.

References

[1] Anand Rajaraman, Jure Leskovec and Jeffrey D. Ullman. *Mining of Massive Datasets*. Cambridge University Press. (December 30, 2011).

[2] cmhteixeira. *Locality Sensitive Hashing (LSH)* [online]. (November 29, 2017). <https://aerodatablog.wordpress.com/2017/11/29/locality-sensitive-hashing-lsh/>[Consulted: December 12, 2018].

[3] Hubert Brylkowski. *Locality sensitive hashing −LSH explained* [online]. (October 6, 2017). <https://medium.com/engineering-brainly/locality-sensitive-hashing-explained-304eb39291e4>[Consulted: December 15, 2018].

[4] Jeffrey D. Ullman[Mining Massive Datasets]. (July 23, 2016). *3 2 Minhashing 25 18*. <https://www.youtube.com/watch?v=96WOGPUgMfw>.

[5] Jeffrey D. Ullman[Mining Massive Datasets]. (July 23, 2016). *3 3 Locality Sensitive Hashing 19 24*. <https://www.youtube.com/watch?v=_1D35bN95Go>.

[6] santhoshhari. *Locality Sensitive Hashing: Application of Locality Sensitive Hashing to Audio Fingerprinting* [online]. (n. d.). <https://santhoshhari.github.io/Locality-Sensitive-Hashing/>[Consulted: December 16, 2018].

[7] Shikhar Gupta. *Locality Sensitive Hashing: An effective way of reducing the dimensionality of your data* [online]. (June 29, 2018). <https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134>[Consulted: December 16, 2018]