6/1/2017

# Assignment 1 "Reuters-21578"

## Finding Similarities Between Documents

BOUROS NIKOS
GIANNATOU EVA
VALLILA MARIA

## Contents

## 1. Introduction

Many big-data problems can be expressed as finding "similar" items. In this project we will investigate similarities among 21578 documents from a cleanup collection of documents were made available by Reuters and CGI for research purposes. The collection appeared in 1987 and after processing in 1996 the data set had the form we know today with 21578 text categorization collection. As the name indicates, this collection contains 21578 text documents from Reuters Ltd. To be more precise, the collection consists of 22 data files, an SGML DTD file describing the data file format, and six files describing the categories used to index the data. Each of the first 21 files (reut2-000.sgm through reut2-020.sgm) contain 1000 documents, while the last (reut2-021.sgm) contains 578 documents.
The **AIM** of this Assignment is to discover relationships between these texts, using k-Shingles, Jaccard similarities through Minhashing and Locality Sensitive Hashing. We are interested to investigate how similar the texts are. For this purpose we think data as "Sets" of "Strings" and convert shingles into minhash signatures.

For the whole analysis we used Python 2.7. For graphs we used Microsoft excel.
In more detail

## 2. Dataset

The documents used for this project appeared on the Reuters newswire in 1987. In 1990, the documents were made available by Reuters and CGI for research purposes at the University of Massachusetts at Amherst. Formatting of the documents and production of associated data files was done in 1990. Further formatting and data file production was done in 1991 and 1992 at the Center for Information and Language Studies, University of Chicago. This version of the data was made available for anonymous FTP as "Reuters22173, Distribution 1.0" in January 1993. From 1993 through 1996, Distribution 1.0 was hosted at a succession of FTP sites maintained by the Center for Intelligent Information Retrieval of the Computer Science Department at the University of Massachusetts at Amherst. At the ACM SIGIR '96 conference in August, 1996 a group of text categorization researchers discussed how published results on Reuters-22173 could be made more comparable across studies. It was decided that a new version of collection should be produced with less ambiguous formatting, and including documentation carefully spelling out standard methods of using the collection. The opportunity would also be used to correct a variety of typographical and other errors in the categorization and formatting of the collection. One result of the re-examination of the collection was the removal of 595 documents which were exact duplicates (based on identity of timestamps down to the second) of other documents in the collection. The new collection (which one we used in analysis) therefore has only 21578 documents.

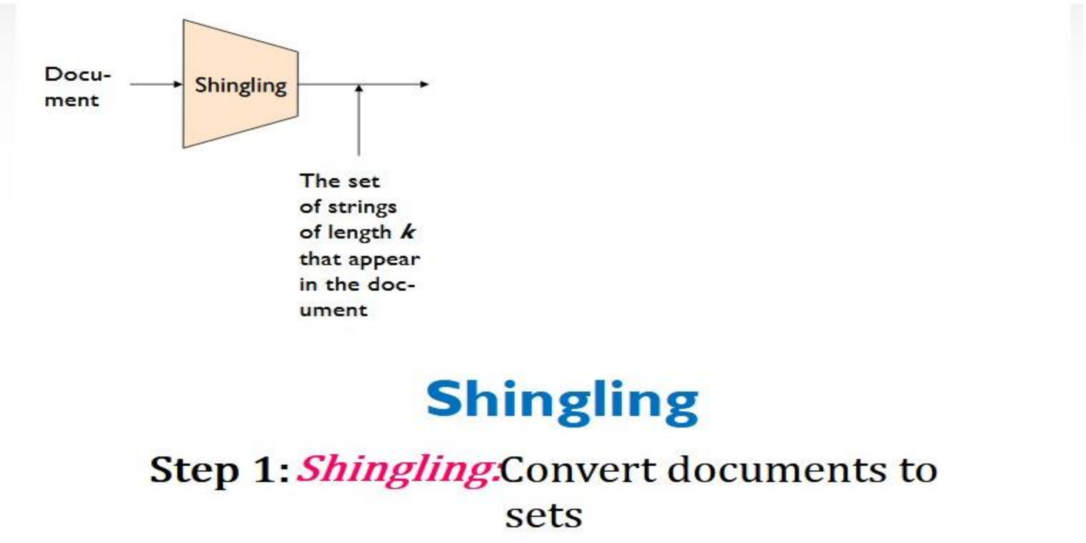
## 3. Cleansing

Python was used for data cleansing and transformation**.** We only kept the main body from each document. This was done by using the Beautiful Soup module (see code), which enabled us to distinguish only the body tags and ignore the rest, gaining time resources than processing the whole document. We then deleted the punctuation (full stops".", commas",", exclamation marks"!", semicolons ";”, apostrophe "'", question marks "?" etc) and the symbol "/n" which is used to declare the changing of the row. Finally, all text was converted to lowercase.

## 4. K-shingles creation and comparison

### Shingles

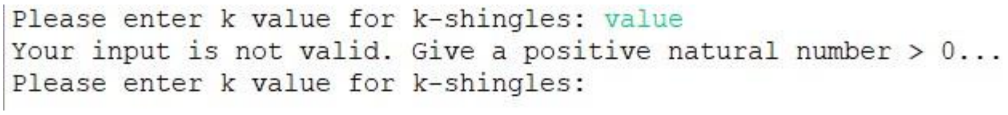
Having cleared our data previously, our next step was to split each document into words. In this way, we could select different number of words each time to create our shingles depending on the user’s input. It has to be clarified that the order of the words remained unchanged, as the sequence of the words is the one that gives meaning to the specific word shingling we applied. A k-shingle for a document is a sequence of k-tokens that appears in the doc. Tokens (or strings) can be words in our example. Converting (hashing) each possible string of k consecutive words from the documents to integers. This technique of hashing substrings is referred to as “shingling”, and each unique string is called a “shingle”. To be more precise our example shingles are "bags of words".



Shingling

Step 1: *Shingling:*Convert documents to sets

The user interacts with the program for the first time by giving the number of words that a shingle would consist of. The user is free to choose any natural positive number greater than 0. In case the input is out of this range the program informs the user that the value was given is not valid and ask him to re-enter one until it's appropriate. The following figure shows such an occasion.



```
Please enter k value for k-shingles: value
Your input is not valid. Give a positive natural number > 0...
Please enter k value for k-shingles:
```

*Figure 1 Not valid input by user for k value*

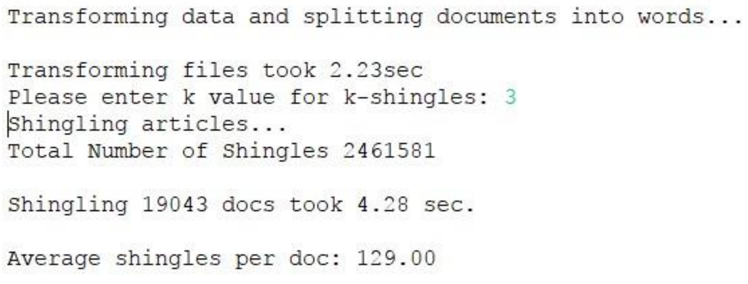
After the user gives a valid value of how many words will make a shingle, the shingling procedure begins. For each document sets of k words (where k is the value given before) are created, where each set can be viewed as an instance of a sliding window rolling over the document. We give an example with one of the smallest documents for 3-words shingles:

```
['sri lankas central', 'lankas central bank', 'central bank offered',
'bank offered 250', 'offered 250 mln', '250 mln rupees', 'mln rupees
worth', 'rupees worth of', 'worth of threemonth', 'of threemonth
treasury', 'threemonth treasury bills', 'treasury bills at', 'bills at
its', 'at its weekly', 'its weekly tender', 'weekly tender closing',
'tender closing on', 'closing on march', 'on march 6',
'march 6 a', '6 a bank', 'a bank spokesman', 'bank spokesman said',
'spokesman said reuter']
```

As the information we need in this stage is which shingles exist in the document and not how many times a specific shingle appears in the document, we uniquified the shingles in each document. The next step was to hash all these shingles to integers using a hash function. Integers are much faster to be checked for equality than a set of strings when the cost of storing them in memory is quite smaller. The hash function we chose is binascii.crc32() of binascii module.

As every shingle of each document was read, it was immediately hashed to an integer and stored in the equivalent document's set. In the end, we had a set of integers for each document instead of a set of shingles.

Figure 2 displays some outputs of shingling procedure for 3-words shingles and all the 19.043 documents:

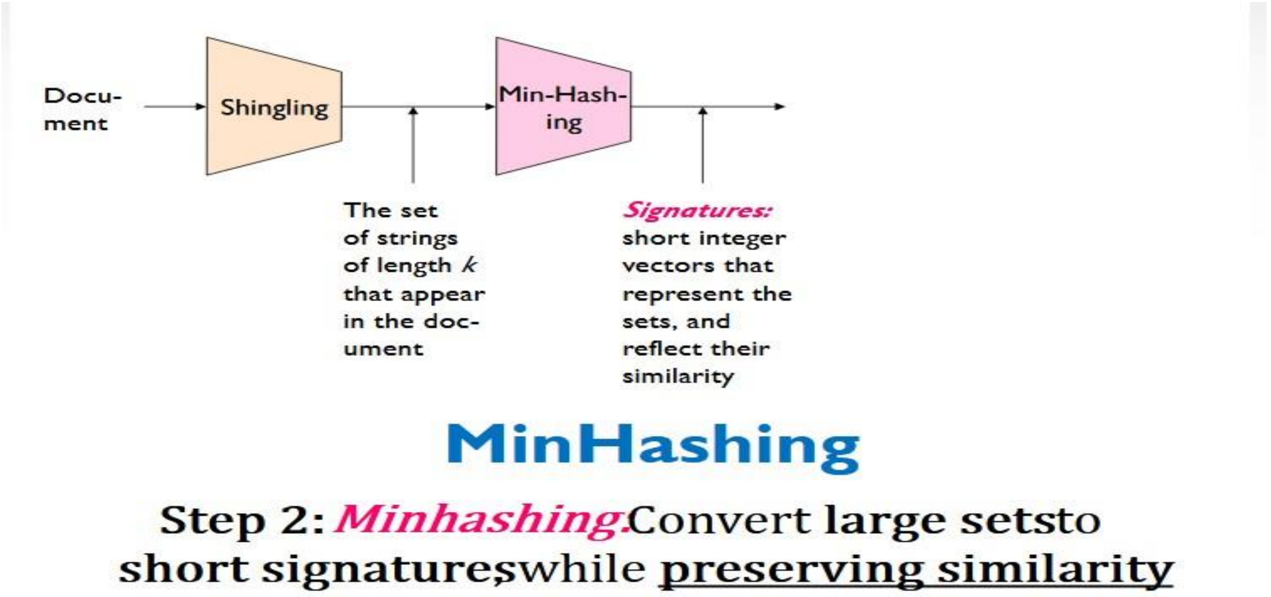


```
Transforming data and splitting documents into words...

Transforming files took 2.23sec
Please enter k value for k-shingles: 3
Shingling articles...
Total Number of Shingles 2461581

Shingling 19043 docs took 4.28 sec.

Average shingles per doc: 129.00
```

*Figure 2 Shingling procedure outputs*

## 5. Minhashing/Jaccard similarity

Originally the minhashing method aims to find ways to measure the similarity of documents by comparing the documents' signatures and not the documents themselves. In other words minhashing technique is taking large sets of items and turn them to short signatures (vector of a big number of integers) while preserving similarity. The length of the signature of each document depends on how many hash functions will be used on each document. As the assignment defines the number of hash functions as parametric for our implementation, we enabled the user to decide it.



Step 2: *Minhashing*:Convert large setsto short signatures,while preserving similarity

Minhashing uses Jaccard Similarity, which is given by the intersection of the sets divided by their union, for compares:

**Jaccard Similarity**

**Jaccard Index = (the number in both sets) / (the number in either set) * 100**

The same formula in notation is: **J(X,Y) = |X∩Y| / |XUY|**

This percentage tells you how similar the two sets are.

Two sets that share all members would be 100% similar. the closer to 100%, the more similarity (e.g. 90% is more similar than 89%).

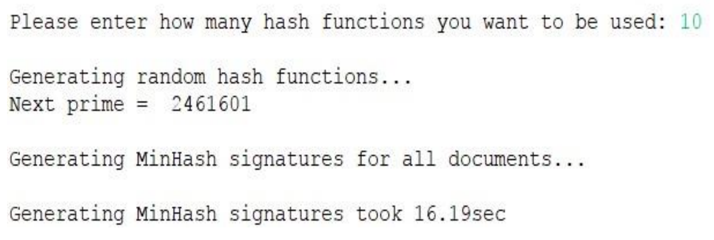
If they share no members, they are 0% similar.

The midway point — 50% — means that the two sets share half of the members.

All the hash functions to be used, whichever their number, come from the same function family, which is (Ax + B) modC, where x is the integer that came from a hashed shingle, A and B are random coefficients, always differing between different hash functions, and C is the next prime number from the total number of shingles in all documents. As x is already known each time for each document and each shingle and stored in a set, we have to give values to A, B and C. A and B took values from our method pickRandomCoeffs(k), which every time called (once to pick as many A as the hash functions and secondly for the B) returned a number between 0 and the total number of shingles, and this number could not be used again for the same coefficient. In this way, we ensured that there exist no hash functions with same A and B. As for coefficient C, we used a probabilistic method to find the next prime number after total number of shingles. This method can be found in our code as MillerRabinPrimalityTest(number) function, and takes as parameter the number of shingles (number). As this method is probabilistic, it doesn't make all divisions between numbers to find a prime one, but it gives a number slightly bigger than the parameter which is a prime number with very high probability. It is impressively fast and nearly 100% accurate in all our tests so we chose it as the most efficient method.

To finally create the signatures matrix we aimed for, we took each shingle integer from each document and use it as x in any of the hash functions asked by the user. For each hash function and for each document set, only the minimum number emerged was kept and stored as the mini-signature of a document for a specific hash function. The total of all mini signatures, whose number was obviously the same with the number of hash functions, consist the signature of a document.

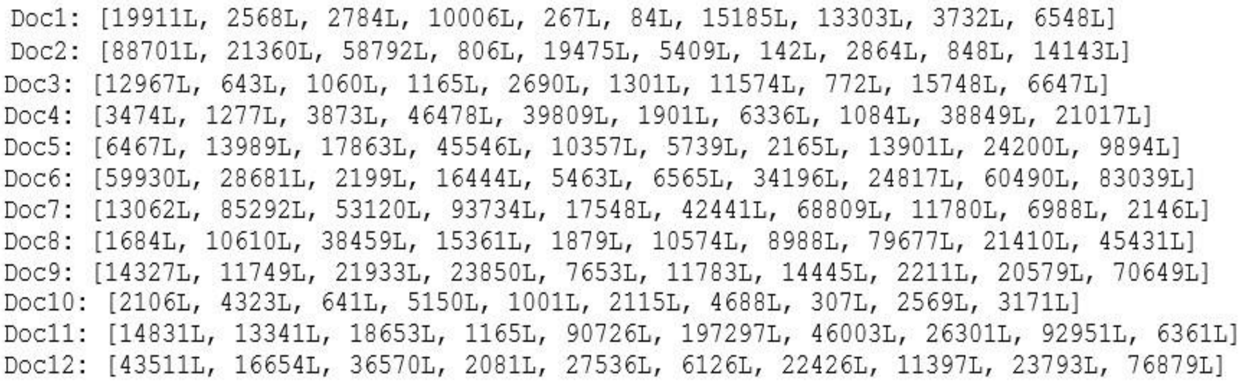
In Figure 3 the interaction with the user as far as the number of hash functions is concerned is displayed:



```
Please enter how many hash functions you want to be used: 10

Generating random hash functions...
Next prime =  2461601

Generating MinHash signatures for all documents...

Generating MinHash signatures took 16.19sec
```

*Figure 3 Generating signatures user input and output*

To make it clear how the signature matrix of the documents looks like, we show in Figure 4 a subset of it for 12 random documents. The hash functions used for this example were 10 and thus 10 numbers represent each document is consisting its signature.



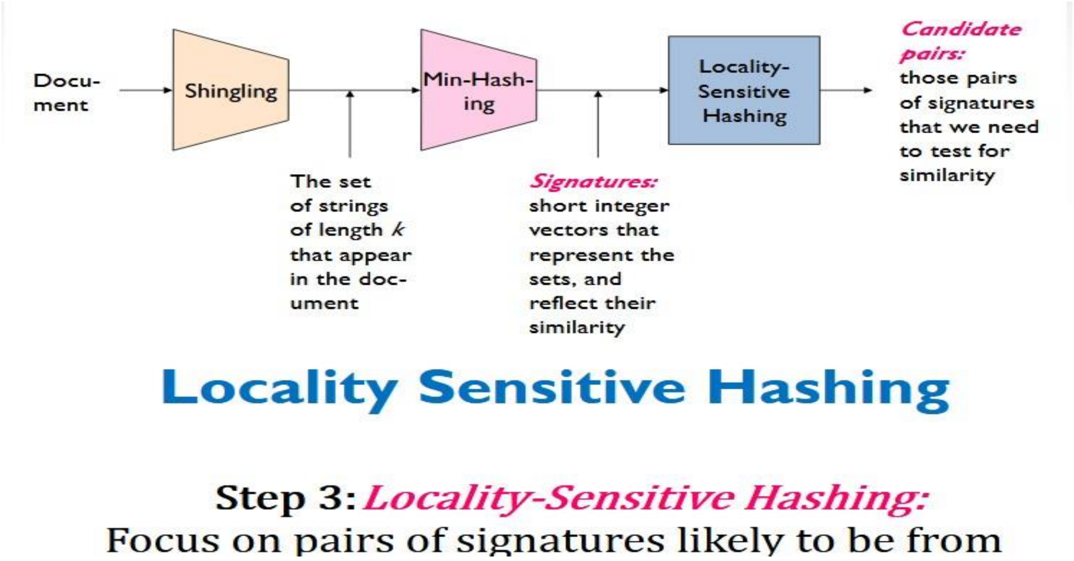
```
Doc1: [19911L, 2568L, 2784L, 10006L, 267L, 84L, 15185L, 13303L, 3732L, 6548L]
Doc2: [88701L, 21360L, 58792L, 806L, 19475L, 5409L, 142L, 2864L, 848L, 14143L]
Doc3: [12967L, 643L, 1060L, 1165L, 2690L, 1301L, 11574L, 772L, 15748L, 6647L]
Doc4: [3474L, 1277L, 3873L, 46478L, 39809L, 1901L, 6336L, 1084L, 38849L, 21017L]
Doc5: [6467L, 13989L, 17863L, 45546L, 10357L, 5739L, 2165L, 13901L, 24200L, 9894L]
Doc6: [59930L, 28681L, 2199L, 16444L, 5463L, 6565L, 34196L, 24817L, 60490L, 83039L]
Doc7: [13062L, 85292L, 53120L, 93734L, 17548L, 42441L, 68809L, 11780L, 6988L, 2146L]
Doc8: [1684L, 10610L, 38459L, 15361L, 1879L, 10574L, 8988L, 79677L, 21410L, 45431L]
Doc9: [14327L, 11749L, 21933L, 23850L, 7653L, 11783L, 14445L, 2211L, 20579L, 70649L]
Doc10: [2106L, 4323L, 641L, 5150L, 1001L, 2115L, 4688L, 307L, 2569L, 3171L]
Doc11: [14831L, 13341L, 18653L, 1165L, 90726L, 197297L, 46003L, 26301L, 92951L, 6361L]
Doc12: [43511L, 16654L, 36570L, 2081L, 27536L, 6126L, 22426L, 11397L, 23793L, 76879L]
```

*Figure 4 Signature matrix*

## 6. Locality-Sensitive Hashing (LSH)

LSH reduces the dimensionality of high-dimensional data. Focus on pairs of signatures likely to be similar LSH hashes input items so that similar items map to the same “buckets” with high probability (the number of buckets being much smaller than the universe of possible input items). LSH breaks MinHashes into a series of bands comprised of rows. For example 20 MinHashes might be broken into 5 band and 4 rows each. Each band is hashed to a bucket. If two documents have the exact same MinHashes in a band, they will be hashed into the same bucket, and so they will be considered candidate pairs. Each pair of documents has as many chances to be considered a candidate as the number of bands. The fewer the rows are in each band, the more likely it is that each document will match another. The probability of a match depends on the Jaccard similarity of a pair of documents. The more similar the documents are, the more likely they are to be considered candidates.

The advantage of LSH is that it will only calculate the similarities between the pairs which fell at least one time in the same bucket, pairs also known as candidate. Concluding, LSH only calculates Jaccard similarity for pair of documents which have a high probability to be similar while it eliminates the rest.


Docu-
ment
Shingling
Min-Hash-
ing
Locality-
Sensitive
Hashing
Candidate pairs: those pairs of signatures that we need to test for similarity
The set of strings of length k that appear in the doc-ument
Signatures: short integer vectors that represent the sets, and reflect their similarity

Locality Sensitive Hashing

Step 3: *Locality-Sensitive Hashing:*
Focus on pairs of signatures likely to be from

**LSH computational cost**

If there are N documents with an average length of D and K hash functions, we can calculate the number of operations in order to find the right bucket for document A as D*K. In order to figure out in which bucket A will land we need to take A and do a dot product between A to each one of the normal vectors for the K hyperplanes. Once A lands in a bucket, we will compare A with every other document in that bucket. If we have K hyperplanes (or bits) then we have 2K possible hash codes or regions in space (number of possible intersections of hash spaces we can possibly have). The average points in a bucket are N/(2k). The cost of comparisons equals to DN/(2k). LSH divides space into an exponential number of buckets and this is repeated L times.

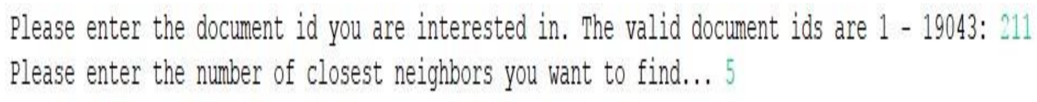
Concluding, LSH has a complexity of LDK + LDN/(2K), or O(logN). If we create an index then the complexity is O(sqrt(N)), while brute force search complexity is O(N).

## 7. Jaccard similarities for Shingles/Signatures/LSH

Having finished with the implementation of all the methods required, we headed to calculate similarities between documents. For the needs of the current assignment the user must choose a specific document ID to the program and the number of closest neighbors (documents) he wants to find and let the program find them. However, despite the fact that the end state of the project let LSH give the answer, we tried to find similarities for the documented requested each time comparing a) shingles, b) documents signatures and c) with the LSH method.

Obviously, comparing all shingles of a document with all shingles of any other document consumes an enormous amount of time and resources of the computer memory, however we had to do it in order to make a comparison between the safest and most straightforward but costly solution and the alternatives founded to keep a good result with much less cost. The reasons behind it are educational and it gave us the opportunity to measure how worse a solution like this is and moreover obtain the base to calculate the false negatives and false positives of min hashing and LSH, given that the shingles comparison can be considered the actual truth.

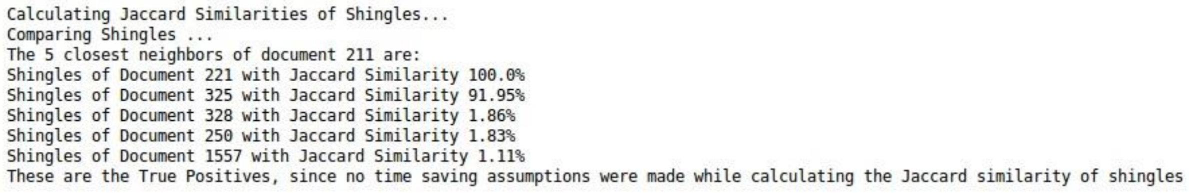
In the next Figure you can see the message the user takes when choose the document he wants to find its neighbors and how many neighbors he desires. The range available is displayed each time and for the needs of this project it is the total number of documents, which is 19043.



```
Please enter the document id you are interested in. The valid document ids are 1 - 19043: 211
Please enter the number of closest neighbors you want to find... 5
```

*Figure 5 User chooses desired document and number of neighbors*

Indicatively, we chose a random document, the document 211, to present the results using each comparison method alone. Starting from shingles comparison alone our program gave the results:



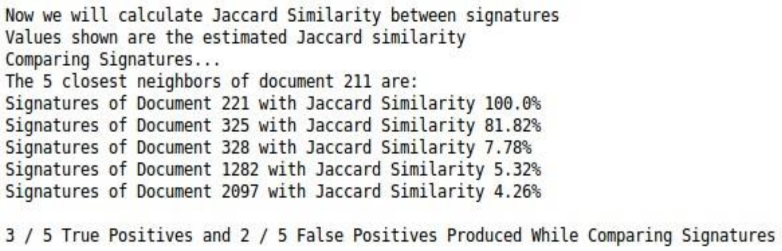
```
Calculating Jaccard Similarities of Shingles...
Comparing Shingles ...
The 5 closest neighbors of document 211 are:
Shingles of Document 221 with Jaccard Similarity 100.0%
Shingles of Document 325 with Jaccard Similarity 91.95%
Shingles of Document 328 with Jaccard Similarity 1.86%
Shingles of Document 250 with Jaccard Similarity 1.83%
Shingles of Document 1557 with Jaccard Similarity 1.11%
These are the True Positives, since no time saving assumptions were made while calculating the Jaccard similarity of shingles
```

*Figure 6 Jaccard similarity with shingles*

In this section, we directly calculated the Jaccard similarities by comparing the shingle sets. This is included here to show how much slower it is than the MinHash and LSH approach. However, the similarities calculated above are the actual similarities of the documents, since there were no assumption made. We will consider the results of figure 6 (docs 221, 325, 328, 250 and 1557) as True Prositives.

The parameters we gave for this example were: 3-words shingles, 50 hash functions, document ID 211 and 5 nearest neighbors. The same parameters will be used for min hashing and LSH as well.

As described in previous Section we created a signatures matrix with all the signatures of all documents. Despite the fact this matrix can fit the memory, making all the comparisons between all documents is again too consuming. Using again document 211 as an example we found the 5 nearest neighbors of this document based only on signatures comparison. The results are shown in Figure 7.



```
Now we will calculate Jaccard Similarity between signatures
Values shown are the estimated Jaccard similarity
Comparing Signatures...
The 5 closest neighbors of document 211 are:
Signatures of Document 221 with Jaccard Similarity 100.0%
Signatures of Document 325 with Jaccard Similarity 81.82%
Signatures of Document 328 with Jaccard Similarity 7.78%
Signatures of Document 1282 with Jaccard Similarity 5.32%
Signatures of Document 2097 with Jaccard Similarity 4.26%

3 / 5 True Positives and 2 / 5 False Positives Produced While Comparing Signatures
```

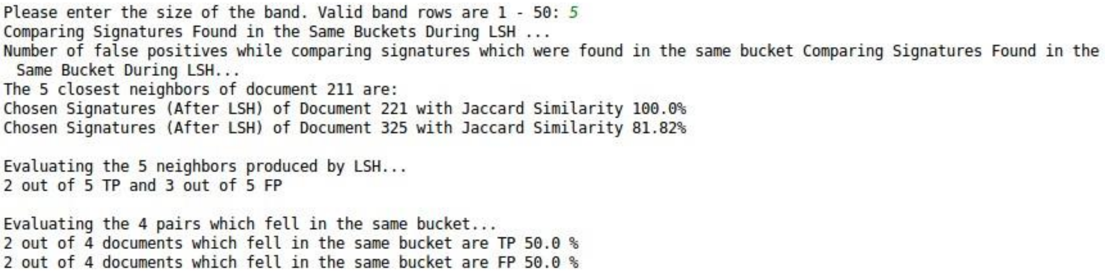
*Figure 7 Jaccard Similarity with signatures comparison*

We will now compare the Jaccard similarity of shingles with the Jaccard similarity of signatures. Remember that we considered the results of Jaccard similarity of shingles as True Positives. Having that in mind, we can identify 3 out of five pairs of figure 7 as True Positives and 2 out of five as False Positives. Here we have to notice that, document 211 is very similar with documents 221 and 325 and then there are several other documents such as 328, 1281, 2097 and others which are less than 10% similar to 221. Pairs with high similarity will always be included in our program's results, while neighbors with similarity lower than 10% will not always appear in the right order.

In this part of the project we will try to reduce the time needed in order to calculate the Jaccard similarity between signatures. To do so we will use a method called Locality Sensitive Hashing (LSH).

Now as far as the python implementation of LSH in concerned, we will now go on more details. The user interacts with the program for the last time by giving the size of the band which is actually the number of rows per band. LSH input consists of the pre calculated signature matrix, the size of the band and the number of hashes. For utility reasons, we kept the number of hashes constant and equal to the number of hashes which was used during the min hashing stage. LSH begins by splitting the already existing signatures into band hashes. The number of bands hashes, as it was mentioned before, depends on the given band size. More specifically, the get hash_function function loops though the MinHashes and uses modular arithmetic (MinhashRow % BandSize) in order to split the signatures into bands (NumHashes / BandSize = Bands). Next, the program creates a dictionary which includes all the pairs found in each bucket and at the same time it counts the number of times that they occurred in the same bucket. After that, it collects all pairs found in the same bucket, which contain the document id that was given as input from the user. These are the candidate pairs. For the candidate pairs only, the program calculates their Jaccard similarity. In this stage, we have a list containing the documents which fell at least one time in the same bucket with document 211 as well as their Jaccard similarity of them. This list will then be sorted by descending similarity, and the k most similar pairs will be printed on the user's screen.

Continuing with the LSH approach we used the same parameters for the same document, adding 5 rows per band as an extra parameter. Figure 8 shows the results.



```
Please enter the size of the band. Valid band rows are 1 - 50: 5
Comparing Signatures Found in the Same Buckets During LSH ...
Number of false positives while comparing signatures which were found in the same bucket Comparing Signatures Found in the
  Same Bucket During LSH...
The 5 closest neighbors of document 211 are:
Chosen Signatures (After LSH) of Document 221 with Jaccard Similarity 100.0%
Chosen Signatures (After LSH) of Document 325 with Jaccard Similarity 81.82%

Evaluating the 5 neighbors produced by LSH...
2 out of 5 TP and 3 out of 5 FP

Evaluating the 4 pairs which fell in the same bucket...
2 out of 4 documents which fell in the same bucket are TP 50.0 %
2 out of 4 documents which fell in the same bucket are FP 50.0 %
```

*Figure 8 Jaccard Similarity using LSH*

As the 3rd closest neighbor in the last example has 0% similarity with document 211 with LSH, no more than 2 documents are displayed in console.

In order to estimate the LSH's performance, we need to calculate the False Positives and False Negatives. False positives are dissimilar pairs which are hashed to the same bucket, and false negatives are similar pairs which are not dispatched to the same bucket. It means that the false positives are pairs which are mistakenly considered as a candidate pair and the false negatives are pairs which are mistakenly not considered as a candidate pair. According to figure 8 and considering the similarity of shingles as the actual similarity, two out of four documents which fell in the same bucket with 211 are True Positives and the rest are False Positives. For consistency reasons, we will also note that two out of five neighbors of figure 8 are True Positives according to the similarity of shingles.

## 8. Sensitivity Analysis - Time

In this section, we will examine how changes in parameters affect the execution time of the program. To make our conclusions clear and state, we will define some default values for all parameters used and we will change one each time.

Before continuing to our sensitivity tests, we will give a picture of which part of our program consumes most of the total time of the program to run for a single document. The following graph gives us a clear answer.

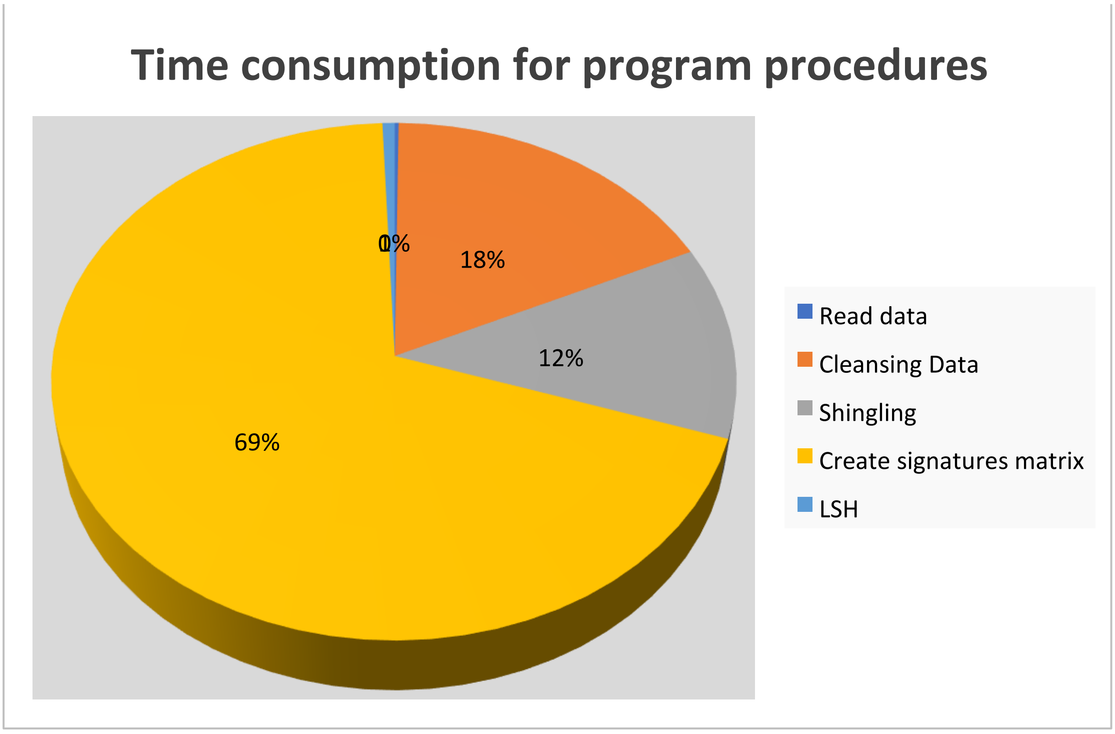


*Figure 9 Time consumption for program procedures*

Obviously, the procedure after shingles are created until the signatures matrix is ready dominates the time for a program run (69%). Reading data is negligible (lower than 1% thus only slightly displayed) as it includes only opening the files and storing their text in an object. Cleansing the data takes 18% of the time, as for our assignment we conduct a fair number of transformations to all document and shingling is the 12% of the total time. The LSH takes only 1% of the total time, indicating that it can give a fast result once the matrix is ready, and that is the reason of its existence anyway.

It has to be mentioned that the pie chart is created indicative for 50 hash functions. The more the hash functions the more the share of signatures matrix creation.

However, we chose to use a relatively small value for hash functions to stress the proportion of the rest procedures concerning time.

The default values for our parameters to be used as benchmark are:

- 3 words-shingles
- 50 hash functions
- 5 neighbors
- Band size 4

Firstly, keeping the rest stable, we experimented with k-shingles value. The number of words we put into the test were 2, 3, 5 and 8. For each value we calculated into the program the time for the 5 sections of the above pie chart. It is clear that reading and cleansing data remained the same as it doesn't depend on any of the parameters to be tested. The results of our measurements are displayed in Table 1.

| k-shingles | Read data | Cleansing | Shingling | Create signatures' matrix | LSH | Total Time |
|---|---|---|---|---|---|---|
| 2 | 0,23 | 20,31 | 4,14 | 79,25 | 0,95 | 104,88 |
| 3 | 0,23 | 20,31 | 4,43 | 82,86 | 0,98 | 108,81 |
| 5 | 0,23 | 20,31 | 4,65 | 83,24 | 0,98 | 109,41 |
| 8 | 0,23 | 20,31 | 4,66 | 80,08 | 0,98 | 106,26 |

*Table 1. Time sensitivity analysis on k-shingles values*

All time values in the table are in seconds. The total time is the sum of all other times for each k value. The total times related to the k-shingles are shown in Figure 10.

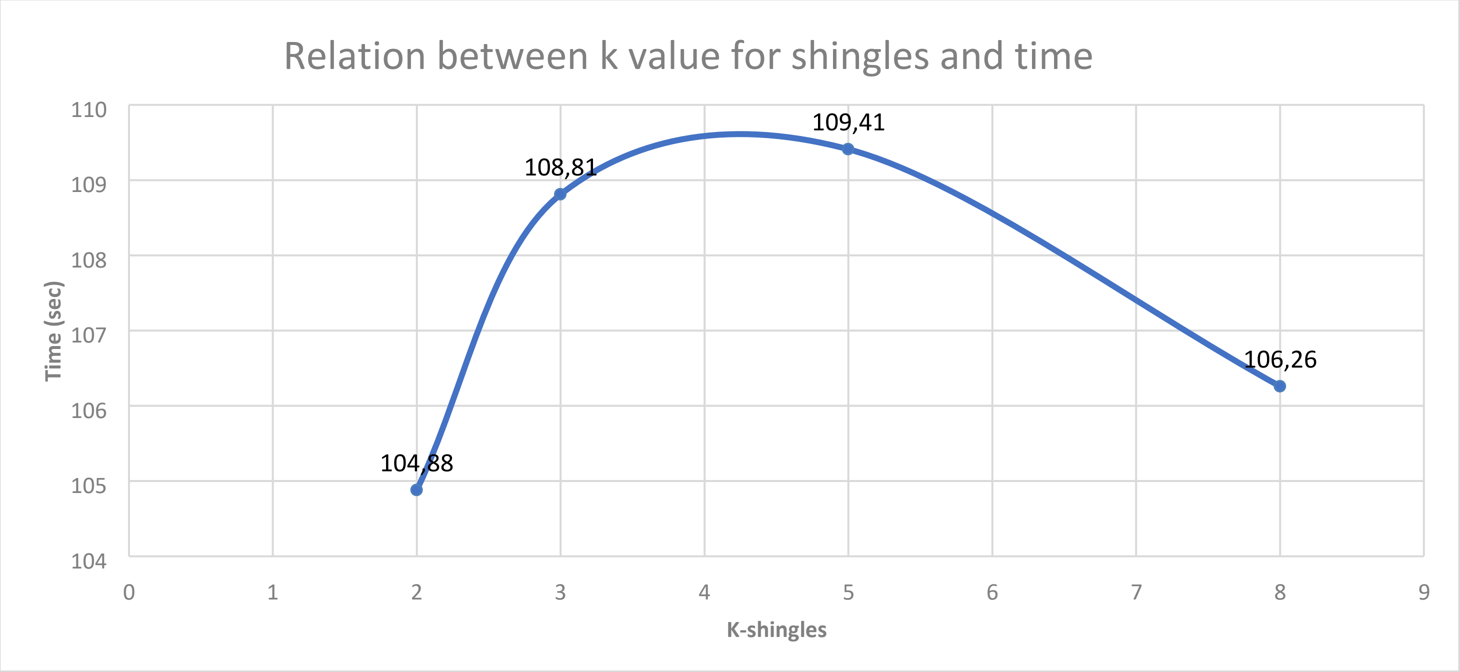


*Figure 10 Relation between k-shingles and time*

The maximum time for the program to run is for 5-words shingles and the minimum for 2 words-shingles. However, given that times are not totally constant and change due to RAM resources and computer tasks running at that time (we ran the tests one after another, however slight differences always appeared) the time differences are almost negligible. Despite the fact that we would expect execution time going lower as k-shingles rise, due to less shingles to be processed, this trend is almost displaced by shingling time, which goes up as shingles have more words, as the hashing is more consuming for bigger strings. Concluding, the number of words consist a shingle doesn't seem to affect crucially the time the program need to run.

Our next sensitivity analysis focused on the number of hash functions used to create the signatures' matrix and its relation to time. Figure 11 displays the results.

| Hash functions | Read data | Cleansing | Shingling | Create signatures' matrix | LSH | Total Time |
|---|---|---|---|---|---|---|
| 20 | 0,23 | 20,31 | 4,45 | 33,77 | 0,38 | 59,14 |
| 50 | 0,23 | 20,31 | 4,42 | 82,94 | 0,98 | 108,88 |
| 80 | 0,23 | 20,31 | 4,3 | 132,08 | 1,34 | 158,26 |
| 120 | 0,23 | 20,31 | 4,26 | 203,73 | 2,29 | 230,82 |

*Figure 11 Time sensitivity analysis on hash functions number*

Obviously, the vast change comes from the signatures' matrix where time is increasing fast as the number of hash functions increases as well. A better view of this trend can be given by Figure 12.

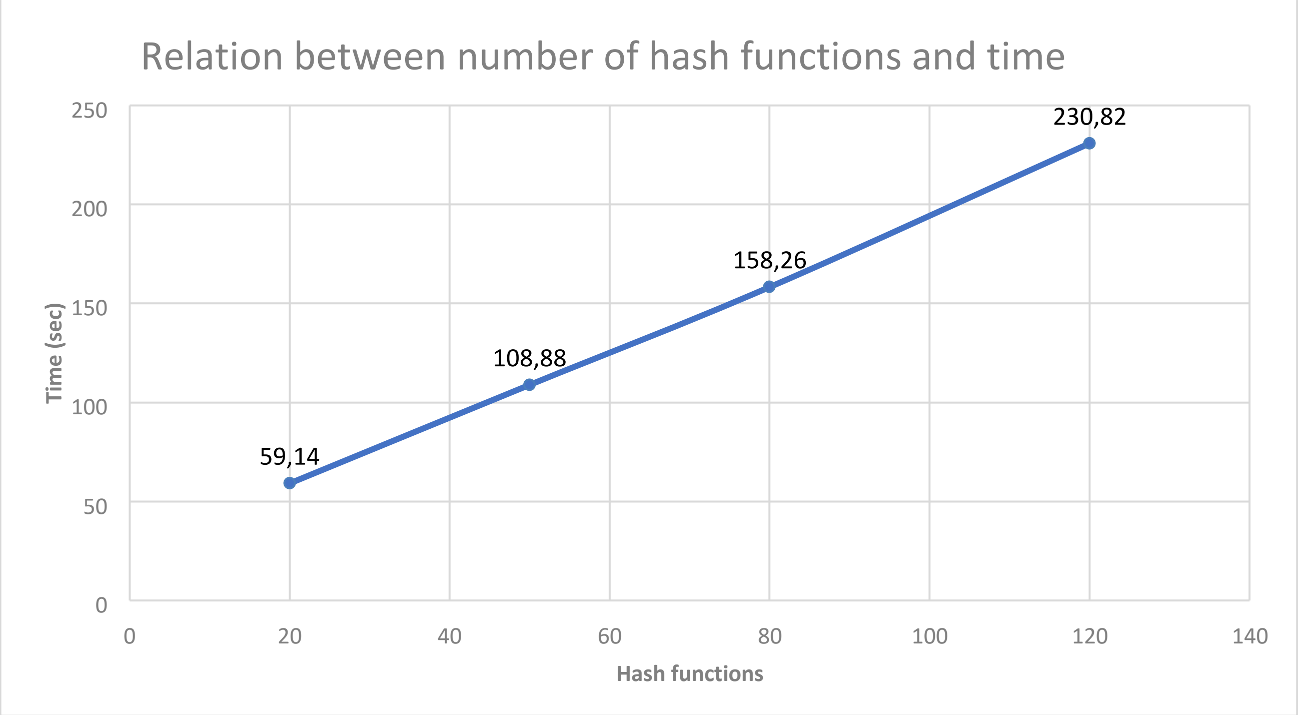


*Figure 12 Relation between number of hash functions and time*

Observing Figure 12 it is quite clear that an almost linear positive relationship between number of hash functions and time exists (in fact it is a little more increasing than linear). As hash functions become more, the computer needs to make more calculations and transformations on signatures and consequently this consumes time resources. The conclusion is that as hash functions increase the time required for the program to run and find similarities of a given document, increase as well, a bit more intensively than a straight line.

## 9. Sensitivity Analysis – Accuracy (False Positives/False Negatives)

In this section, we will examine how changes in parameters affect the accuracy of LSH. More specifically, we will examine how different shingle sizes and different number of hash functions affect the number and the accuracy of candidate pairs.

As it was mentioned before, we have considered the similarity between shingles as true positives. Moreover, we will consider false positives as dissimilar pairs which are hashed to the same bucket, and false negatives as similar pairs which are not dispatched to the same bucket. Since we have chosen to find the 5 closest neighbors of document 211, the maximum value of true positives will be 5 and the number of true positives plus the number of false negatives will equal to 5.

In figure 13 and 14, we expect to see that lower shingle sizes will result higher similarities, more true positives and less false positives and false negatives.

| | LSH | | | |
|---|---|---|---|---|
| **k-shingles** | **# Candidate Pairs** | **True Positives** | **False Positives** | **False Negatives** |
| 2 | 5 | 5 | 0 | 0 |
| 3 | 8 | 5 | 3 | 0 |
| 5 | 11 | 5 | 6 | 0 |
| 8 | 2 | 2 | 0 | 3 |

*Figure 13 Relation between size of shingles and candidate pairs/true positives/false positives/false negatives*

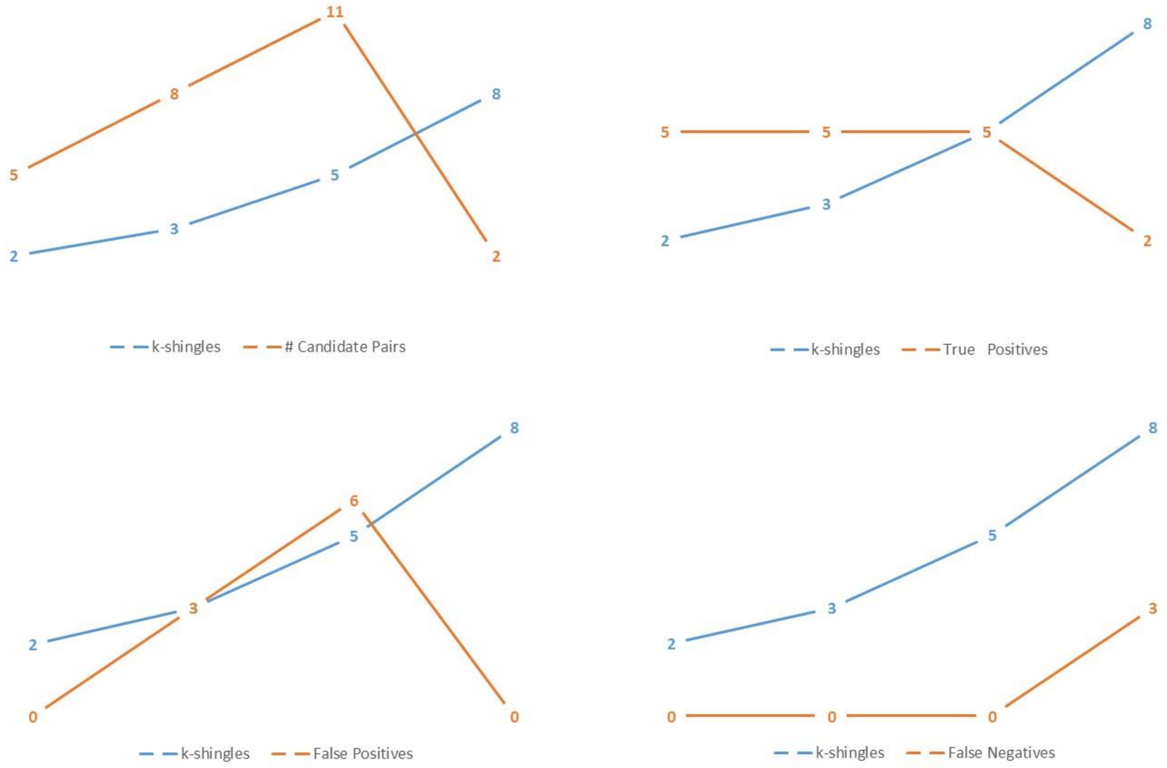


*Figure 14 Plot relation between size of shingles and candidate pairs/true positives/false positives/false negatives*

Also in figure 15 and 16, we expect to see that higher number of hash functions will have more accurate similarities, more true positives and less false positives and false negatives.

| | LSH | | | |
|---|---|---|---|---|
| **NumHashes** | **# Candidate Pairs** | **True Positives** | **False Positives** | **False Negatives** |
| 10 | 2 | 2 | 0 | 3 |
| 20 | 2 | 2 | 0 | 3 |
| 40 | 2 | 2 | 0 | 3 |
| 80 | 4 | 2 | 2 | 3 |

*Figure 15 Relation between number of hashes and candidate pairs/true positives/false positives/false negatives*

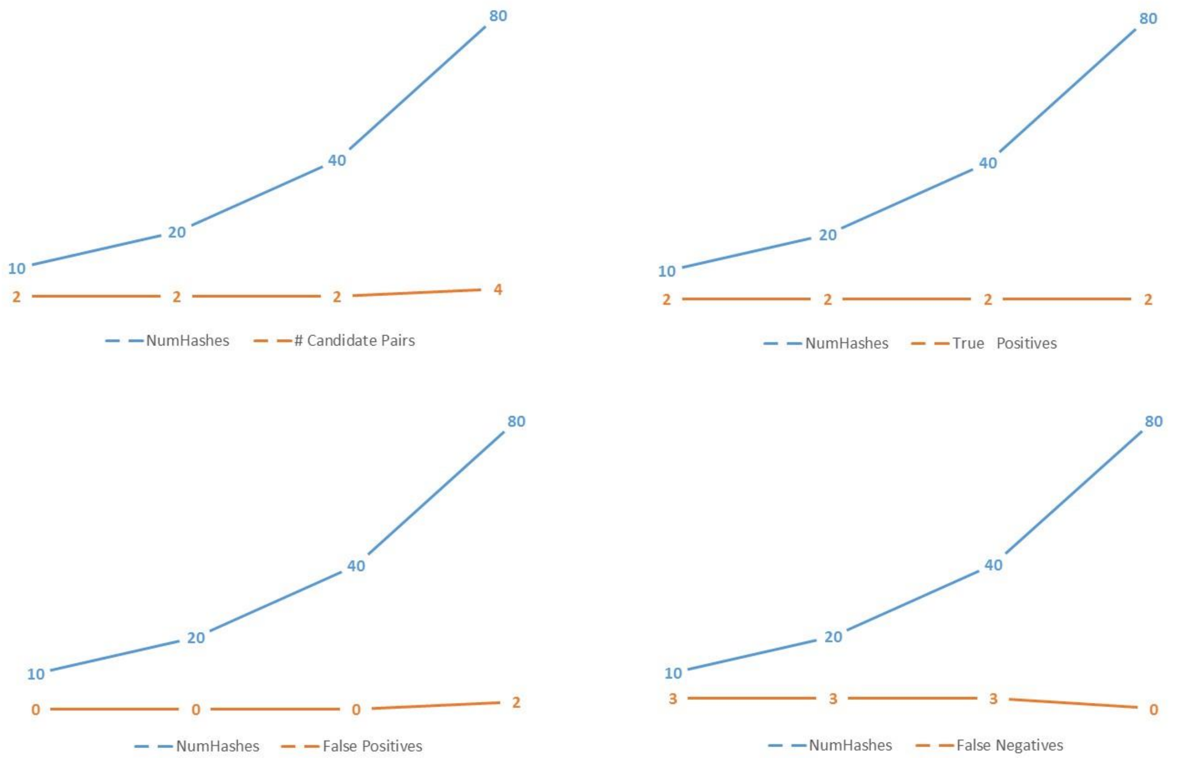
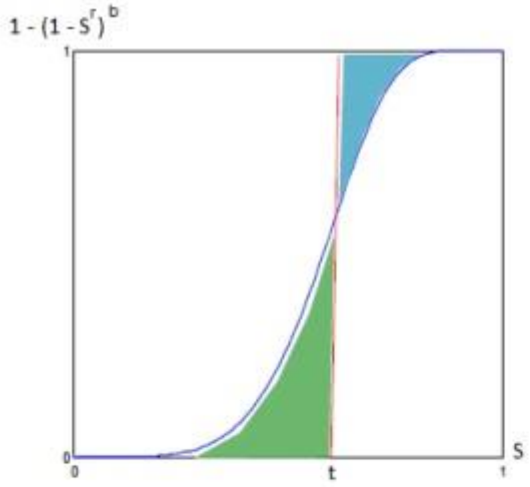


*Figure 16 Plot relation between number of hashes and candidate pairs/true positives/false positives/false negatives*

The results meet our expectations, however we cannot draw any solid conclusion from these plots and due to the fact that only one document was examined. It is also important to mention that document 211 was similar with only two other documents. The third closest neighbor of document 211 shared a similarity of 7.78% and it has a very low probability to be found in the same bucket as document 211.

The amount of time needed in order to recreate the above plots for many documents, makes it a difficult and time consuming task. However, from theory it is known that when we use b bands of r rows each, the possibility that two pairs with the Jaccard Similarity of s may become a candidate pair equals $1 - (1 - s^r)^b$.



The graph of this function along with the areas related to the quantity of false positive and false negative are shown. The area of the green region equals the quantity of false positive and the area of blue region equals the quantity of false negative. In our case, the threshold equals to the similarity between the 5th neighbor and document 211.

*Figure 17 The curve of $1 - (1 - s^r)^b$ function*

## 10. Conclusions

The AIM of this Assignment is to discover relationships among 21578 documents from a cleanup collection of documents were made available by Reuters and CGI for research purposes. We are interested to investigate how similar the texts are using k-Shingles, Jaccard similarities through Minhashing and Locality Sensitive Hashing. For this purpose we think data as "Sets" of "Strings" and convert shingles into minhash signatures. After cleaning our data as we described in section 1 we moved on in Analysis. We implemented k-shingle method, minhashing method and LSH to find similarities between documents and calculated the Jaccard similarities by comparing the shingle sets with signatures (minhashes) and signatures themselves (LSH). For sure in descending order the time of operations are: 1. Shingle sets 2. MinHash and 3. LSH approach.

Also we have shown in 7 part of our report the total time the program consumes to run for a single document. After shingles were created and until the signatures matrix is ready dominates the time for a program run is 69%. Reading data is negligible (lower than 1% thus only slightly displayed) as it includes only opening the files and storing their text in an object. Cleansing the data takes 18% of the time, as for our assignment we conduct a fair number of transformations to all document and shingling is the 12% of the total time. The LSH takes only 1% of the total time, indicating that it can give a fast result once the matrix is ready. Also we implemented time sensitivity analysis on k-shingles values (table 1) and time sensitivity analysis on hash function number (figure 11). In the first option the number k of words consist a shingle doesn’t seem to affect crucially the time the program need to run while time is increasing fast as the number of hash functions increases as well.

Concluding, in section 8 (Sensitivity Analysis - Accuracy) smaller shingles, higher number of bands and smaller band sizes lead to better accuracy (less false positives and false negatives) but also to higher computational time and resources. The optimal parameter selection depends on the nature of the problem, the number and the size of the documents and also it depends on the time and computation resources.

## 11. Bibliography

https://archive.ics.uci.edu/ml/datasets/Reuters-21578+Text+Categorization+Collection

http://slideplayer.com/slide/4552632/ https://e-

mscba.dmst.aueb.gr/mod/resource/view.php?id=1850

https://www.textroad.com/pdf/JBASR/J.%20Basic.%20Appl.%20Sci.%20Res.,%203(1s)466-472,%202013.pdf https://www.youtube.com/watch?v=Arni-zkqMBA

https://github.com/chrisjmccormick/MinHash/blob/master/runMinHashExa

mple.py https://www.youtube.com/watch?v=MaqNlNSY4gc

https://github.com/rahularora/MinHash

http://infolab.stanford.edu/~ullman/mmds/ch6.pdf

https://www.codeproject.com/Articles/691200/Primality-test-algorithms-Prime-test-Thefastest-w

https://github.com/embr/lsh/tree/master/lsh

https://github.com/go2starr/lshhdc/blob/master/lsh/lsh.py

https://github.com/rahularora/MinHash/blob/master/minhash.py

https://nickgrattan.wordpress.com/2014/03/03/lsh-for-finding-similar-documents-from-alarge-number-of-documents-in-c/

https://github.com/anthonygarvan/MinHash

http:// scikit-learn.org/dev/auto_examples/neighbors/plot_approximate_nearest_neighbors_hyperpar a meters.html#sphx-glr-auto-examples-neighbors-plot-approximate-nearest-neighborshyperparameters-py

http:// scikit-learn.org/dev/auto_examples/ neighbors/plot_approximate_nearest_neighbors_scalability.html#sphx-glr-auto-examplesneighbors-plot-approximate-nearest-neighbors-scalability-py