Homework 1: Finding Similar Items: Textually Similar Documents

Assignment group 37 Eliane Birba

1. Introduction

The purpose of this project is to implement the stages of finding textually similar documents based on Jaccard similarity using the *shingling*, *minhashing*, and *locality-sensitive hashing* (LSH) techniques and corresponding algorithms. The implementation is done using, Python with jupyter notebook installed through anaconda.

2. Dataset

In this project I investigate similarities among 5508 documents from a cleanup collection of documents were made available by Reuters and CGI for research purposes. The data set is well structured.
The collection used consists of 6 data files, an SGML DTD file describing the data file format. Each of the first 6 files (reut2-000.sgm through reut2-005.sgm) contain 1000 documents. The AIM of this Assignment is to discover relationships between these texts, using kShingles, 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.

3. Clearning

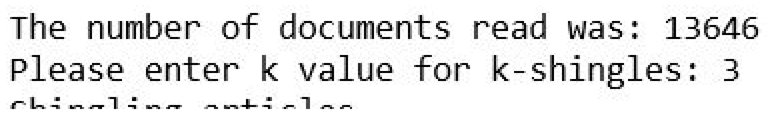
I used BeautifulSoup module for extracting the text. It enabled us to distinguish only the body tags and ignore the rest, gaining time resources than processing the whole document. I only kept the main body from each document. I then deleted the punctuation (full stops".", commas",", exclamation marks"!", 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

After data cleaning, I 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. However 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.

Then I hash each possible string of k consecutive words from the documents to integers.

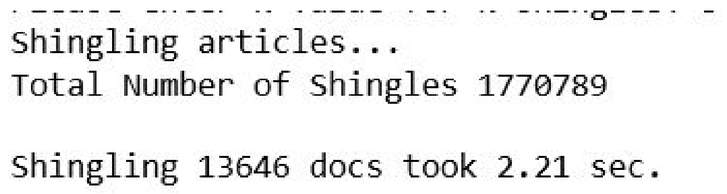
I let user interacts with the program to give the value k>0 that shingle would consist of.



```
The number of documents read was: 13646
Please enter k value for k-shingles: 3
```

class Shingling
the shingling procedure begins with class . For each document sets of k words are created, where each set can be viewed as an instance of a sliding window rolling over the 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 in python for hashing to 32 bytes.

Every shingle 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

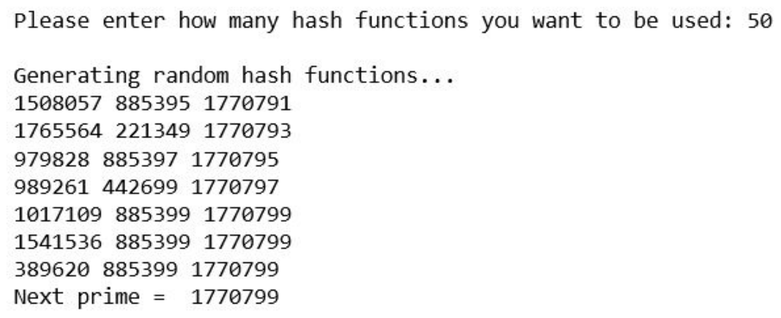


```
Shingling articles...
Total Number of Shingles 1770789

Shingling 13646 docs took 2.21 sec.
```

5. Minhashing/Jaccard similarity

 Measure the similarity of documents by comparing the documents' signatures and not the documents themselves. The length of the signature of each document depends on how many hash functions will be used on each document.
I enabled the user to decide it.
To compute jaccard smirilary the the intersection of the sets divided by their union, for compares
I used the minhashing: (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. I calculated C using probabilistic , methodMillerRabinPrimalityTest(number), 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.
I 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.



```
Please enter how many hash functions you want to be used: 50

Generating random hash functions...
1508057 885395 1770791
1765564 221349 1770793
979828 885397 1770795
989261 442699 1770797
1017109 885399 1770799
1541536 885399 1770799
389620 885399 1770799
Next prime =  1770799
```

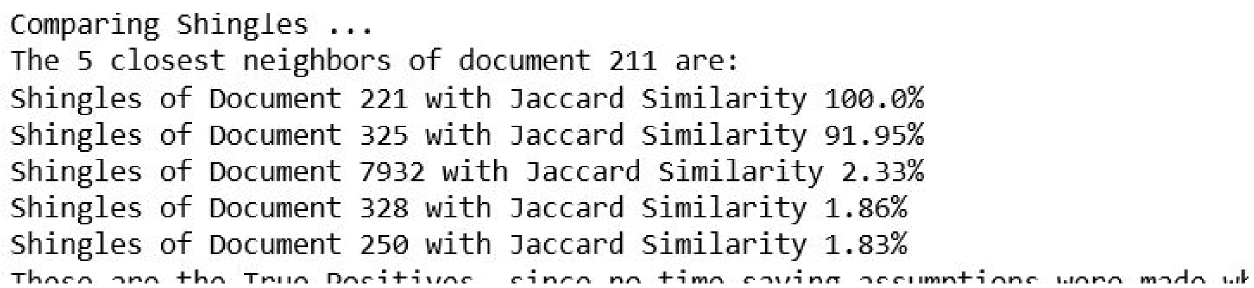
6. Locality-Sensitive Hashing (LSH)

I let user give the size of the band which is actually the number of rows per band
 Class LSH function input consists of the pre calculated signature matrix, the size of the band and the number of hashes. I 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.

7. Jaccard similarities for Shingles/Signatures/LSH

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.

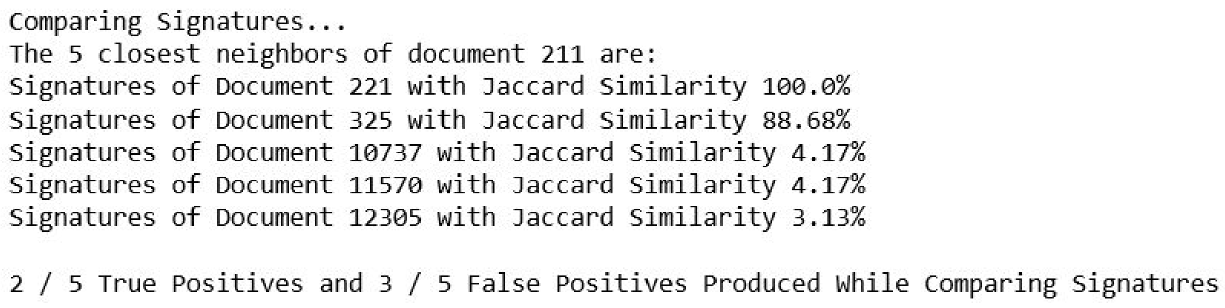
I chose a random document, the document 211, to present the results using each comparison method alone. Starting from shingles comparison alone our program gave the results:



```
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 7932 with Jaccard Similarity 2.33%
Shingles of Document 328 with Jaccard Similarity 1.86%
Shingles of Document 250 with Jaccard Similarity 1.83%
These are the True Positives, since no time saving assumptions were made wh
```

I will consider the results docs 221, 325, 7932,328, 250 as True Positives.
My parameters are : 3-words shingles, 50 hash functions, document ID 211 and 5 nearest neighbors.I used for min hashing and LSH as well.

I created a signatures matrix with all the signatures of all documents.
Jaccard similarity with minhashing:



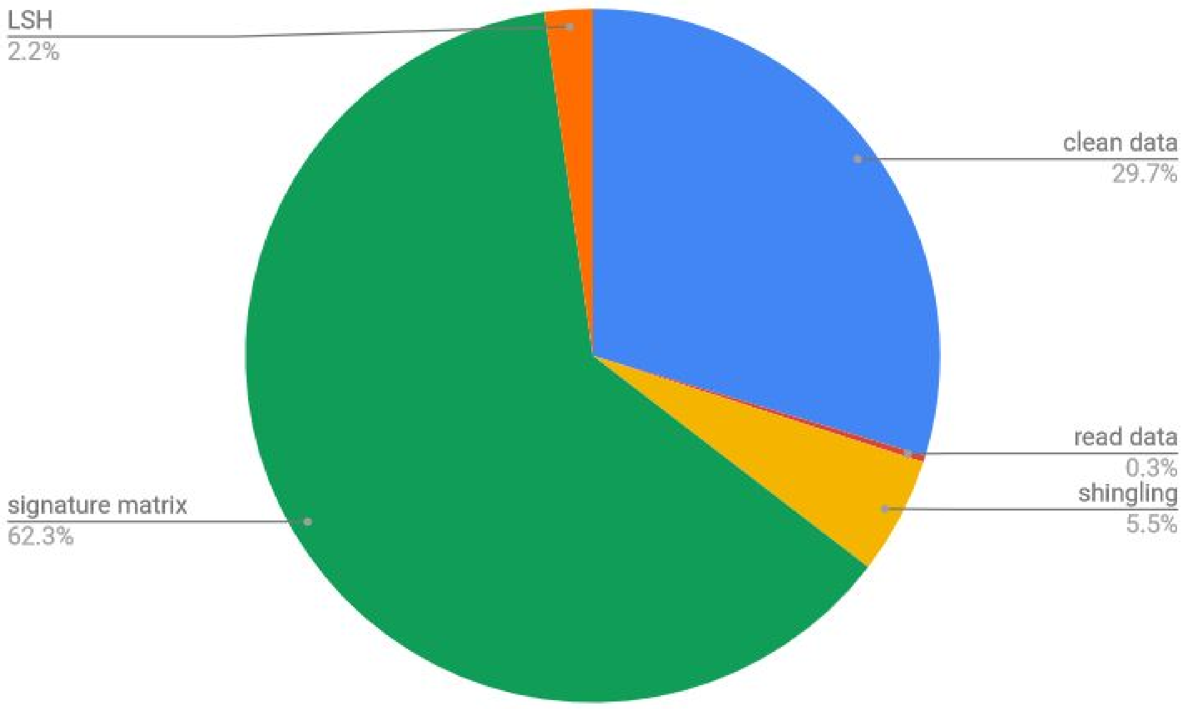
```
Comparing Signatures...
The 5 closest neighbors of document 211 are:
Signatures of Document 221 with Jaccard Similarity 100.0%
Signatures of Document 325 with Jaccard Similarity 88.68%
Signatures of Document 10737 with Jaccard Similarity 4.17%
Signatures of Document 11570 with Jaccard Similarity 4.17%
Signatures of Document 12305 with Jaccard Similarity 3.13%

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

Comparing the Jaccard similarity of shingles with the Jaccard similarity of signatures.
Jaccard similarity of shingles as True Positives. Having that in mind, we can identify 2 out of five pairs of figure 5 as True Positives and 3 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 7932,328, 250 and others which are less than 10% similar to 211. 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

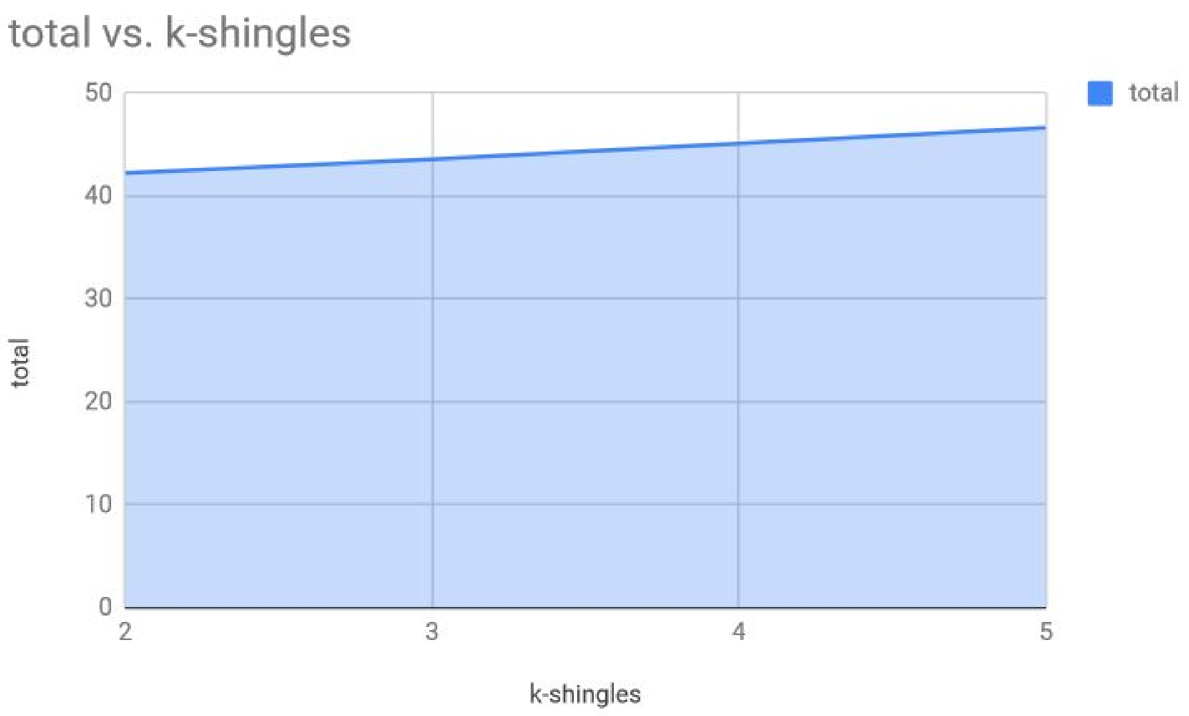
The LSH approach I used the same parameters for the same document, adding 5 rows per band as an extra parameter.
I have a list containing the documents which fell at least one time in the same bucket with document 211 as well as their Jaccard similarity of them. This list will then be sorted by descending similarity, and the k most similar pairs will be printed on the user's screen.

```
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 92.31%

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

Evaluating the 2 pairs which fell in the same bucket...
2 out of 2 documents which fell in the same bucket are TP 100.0 %
0 out of 2 documents which fell in the same bucket are FP 0.0 %
```

8. Time Consumption for program procedure



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



10. How to run

For the whole analysis we used jupyter notebook Python 2.7.15
For graphs we used google sheet.
Install anaconda
Create environment with python 2.7
Install jupyter to this environment.
Open the terminal of this environment and:
Install bs4
Install numpy
Pip install bs4 or numpy
Open jupyter notebook to run the code

You can also run using python lab1.py
But make sure that you have enough memory or remove some files from data folder

The default values for our parameters to be used are:

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

11. References

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-testThefa
stest-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-fromala
rge-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-nearestneighborshyper
parameters-py
http:// scikit-learn.org/dev/auto_examples/
neighbors/plot_approximate_nearest_neighbors_scalability.html#sphx-glr-autoexamplesn
eighbors-plot-approximate-nearest-neighbors-scalability-py