Homework assignment 1

Author: Vendramin Nicolò

In this document I am going to first introduce all the classes and then explain the main script representing the logic of the programme.

A. Shingling class:

```
class Shingling:

    def __init__(self, shingling_size=10):
        self.shingling_size = shingling_size

    def shingle(self, document, k=None, shingle_by_word=False):

        if k==None:
            k = self.shingling_size

        shingles = []
        if shingle_by_word:
            document = document.split(" ")

        for i in range(0,len(document)-k+1):
            shingles.append(document[i:i+k])

        return list(set(shingles))

    @staticmethod
    def shingling(document, k, shingle_by_word=False):

        shingles = []

        if shingle_by_word:
            document = document.split(" ")

        for i in range(0,len(document)-k+1):
            shingles.append(document[i:i+k])

        return list(set(shingles))
```

A

The class can be used both by instantiating an object and calling the shingle method on it, or using the shingling static method.

The shingle and shingling methods require as input the document to be shingled, the length k of the shingle and the boolean indicating wether we want to apply the shingling by characters or by words (suggested by character).

The method simply iterates over the document sliding it by character (or word depending on the setting) and at each iteration it records the sequence of length k characters (or words…) starting from that position. At the end the shingles of the document are converted to a set and back to a list to make them unique and returned.

B. Primes class:

```
class Primes:

    @staticmethod
    def isPrime(num):
        for i in range(2, num/2 + 1):
            if float(num)/i - int(num/i) == 0:
                return False
        return True

    @staticmethod
    def firstAfter(num):
        while(True):
            if Primes.isPrime(num):
                return num
            else:
                num = num + 1
```

B

This class contains two methods to rapidly check wether a number is prime and to find out the first prime number after a given one.

```
class CompareSets:

    @staticmethod
    def JaccardSimilarity(one, two):
        set_one = set(one)
        set_two = set(two)
        common_elements = len(set.intersection(*[set_one, set_two]))
        total_elements = len(set.union(*[set_one, set_two]))

        return float(common_elements) / total_elements
```

C

C. CompareSets class:

This class offers a static method to compute the Jaccard similarity between two sets of integers. It simply computes the size of the intersection of the two sets, and the size of the union of the two sets, to finally return the ratio between those two quantities.

D. MinHashing class:

```python
class MinHashing:

    def __init__(self, tot_shingles, n=5):
        self.n = n
        self.a = []
        self.b = []
        self.c = Primes.firstAfter(tot_shingles)
        for i in range(0, n):
            self.a.append(np.random.randint(1, tot_shingles))
            self.b.append(np.random.randint(1, tot_shingles))

    def minHash(self, document):
        signature = []
        for i in range(0, self.n):
            min_ = self.c + 1

            for shingle in document:
                val = (self.a[i] * shingle + self.b[i]) % self.c
                if(val<min_):
                    min_ = val

            signature.append(min_)

        return signature

    def compareSignatures(self, signature1, signature2):
        common_elements = len(set.intersection(*[set(signature1), set(signature2)]))

        return float(common_elements)/self.n
```

To initialise a min hashing object we have to pass the number of total shingles (tot_shingles) and the size of the output signatures (n). The initialisation function will initialise the parameter of n different hashing functions of the shape (a*x + b) % c where a and b are chosen between 1 and the total number of shingles and c is the first prime after the number of shingles.

Once the min hashing object is initialised is possible to call the minHash function on a document (set of shingles). For each document the minHash function iterates over the n different hashFunctions, computing them on all the shingles of the document and finding the minimum value for each one. The minimum values of the n function will compose the signature of the document.

This class also offers a method compareSignatures() to estimate the Jaccard similarity by comparing two signature, simply by counting the number of common elements divided by the length of the signature.

E. LocalSensitivityHashing class:

The LocalSensitivityHashing class offers methods to apply the LSH method to a set of documents in order to reduce the number of comparison to be done, only to pairs of documents that are likely to be similar basing on their signatures.

At first the builder function (__init__) receiving the number of bands and buckets as input initialises the hash function to be applied, of the shape (a*x + b) % c where a and b are chosen between 1 and the number of buckets and c is the first prime after the given number of buckets (we want the number of buckets to be prime). The initialisation also instantiates the empty list of buckets for each band.

The function lsHashing receives as input the signature and signatureId of a document and stores the signature id in the correct bucket for each of the bands. This function is designed to be called on all the documents (after shingling and minHashing) and after being applied to m document will have the LSH bucketing of all those m documents stored in the buckets variable. The procedure to apply the LSH bucketing is simply to iterate over the bands, to sum the signature in the range of the band, to apply the hashing function to the result and to store the signature id in the corresponding bucket for the band on which we are iterating.

```
class LocalitySensitiveHashing:

    def __init__(self, bands=5, buckets_num=29):
        self.bands = bands
        self.buckets_num = Primes.firstAfter(buckets_num)
        self.a = np.random.randint(1, buckets_num)
        self.b = np.random.randint(1, buckets_num)
        self.buckets = [[[] for i in range(0, self.buckets_num)] for i in range(0, self.bands)]

    def lsHashing(self, signature, signature_id):
        step = int(len(signature)/self.bands)
        partial_sum = 0
        band = 0

        for i in range(1, len(signature)+1):
            partial_sum += signature[i-1]
            if i%step == 0:
                self.buckets[band][(self.a*partial_sum + self.b) % self.buckets_num].append(signature_id)
                partial_sum = 0
                band += 1

        if i%step!=0:
            self.buckets[(self.a*partial_sum + self.b) % self.buckets_num].append(signature_id)

    def getPossibleMatches(self):
        possible_matches = []
        for band in self.buckets:
            for bucket in band:
                for i in range(0, len(bucket)-1):
                    for j in range(i+1, len(bucket)):
                        possible_matches.append((bucket[i], bucket[j]))

        possible_matches = list(set(possible_matches))

        return possible_matches

    @staticmethod
    def localSensitivityHashing(signatures_matrix, bands, buckets_num):
        a = np.random.randint(1, buckets_num)
        b = np.random.randint(1, buckets_num)

        similar_item_ids = []
        number_of_documents = signatures_matrix.shape[0]
        signature_length = signatures_matrix.shape[1]

        step = signature_length / bands

        for band in range(bands):
            buckets = []
            for document in range(0, number_of_documents):
                value = 0
                for signature in range(band, band + step):
                    value = value + (signatures_matrix[document][signature]*a + b)

                hashed_value = value % buckets_num
                buckets.append(hashed_value)

            for candidate_one in range(0, len(buckets)):
                for candidate_two in range(candidate_one+1, len(buckets)):
                    if(buckets[candidate_one] == buckets[candidate_two]):
                        similar_item_ids.append((candidate_one, candidate_two))

        return list(set(similar_item_ids))
```

The function getPossibleMatches just iterates over the buckets and outputs a set of unique couples (signatureIdOne, signatureIdTwo) that should be checked because were having colliding LSH in at least one band. To do so in each bucket for each band we attach to the possibleMatches variable the combination of all the elements with the following one. In this way we produce all the possible colliding pairs. We turn this list into a set to delete repeated occurrences and we return it as a list.

Finally the static method localSensitivityHashing is just a second alternative implementation of the lsHashing applied directly to the whole signature matrix containing all the different documents.

The difference between the two methods is that while the static one is more memory efficient, it requires an higher computational effort that is shown in a small difference in the execution time. The nice thing of the non static method is that it would be possible to easily adapt it in order to be able to call getPossibleMatches also in intermediate phases in case there is the need of start processing possible couples while LSH is still running for a better pipelining.

F. Imports and argument parser

```python
import numpy as np
from scipy.sparse import csc_matrix, csr_matrix
from progressbar import ProgressBar
import argparse
import math
import time
import matplotlib.pyplot as plt

parser = argparse.ArgumentParser()
parser.add_argument('--k_shingle', type=int, default=10)
parser.add_argument('--signature_length', type=int, default=10)
parser.add_argument('--word_shingling', type=bool, default=False)
parser.add_argument('--threshold', type=float, default=0.5)
parser.add_argument('--documents_directory', type=str, default="data")
args = parser.parse_args()

```

The main imports are numpy, scipy.sparse and math to perform the mathematical operations. In addition we use argparse to get input arguments, progressBar to check the advancement of the process and time to measure duration of phases. Finally matplotlib is used to show the results.

Thus to run the program is just necessary to activate an environment with Python3.5 and numpy, scipy, progressBar and matplotib installed, since all the others are standard python libraries.

G. Main function - data import phase:

```python
def main(file_path, num):
    threshold = args.threshold

    # Importing the dataset

    documents = []

    f = open(file_path, "rU")
    for i in range(0, num):
      words = f.readline()
      documents.append(words[words.find(" ")+1:])
    f.close()
```

In this part of the program we just navigate the file provided as input to fetch the documents present inside the file. Documents are one per line and we removed the first part until the first space to cut the article id that is at the head of each line. Thus, documents are stored in a list as strings.

H. Main function - shingling and preprocessing phase:

```python
14.     # SHINGLING PHASE
15.     print("Shingling Phase:")
16.     # I backup the textual version of the dataset before starting to transform it
17.     docs = list(documents)
18.     shingling_time = time.time()
19.
20.     # For each document i map it into the set of its shingles
21.     documents = [Shingling.shingling(document, args.k_shingle, args.word_shingling) for document in documents]
22.
23.     # I unify all the shingles to have the set of all the shingles present in all the analysed documents
24.     shingles = list(set([shingle for document in documents for shingle in document]))
25.
26.     # I build a dictionary mapping each shingle to an integer value
27.     shingle_dic = {shingles[i]: i for i in range(0, len(shingles))}
28.
29.     size = len(documents)
30.     shingle_doc_pairs = []
31.     rows = []
32.     cols = []
33.     data = []
34.
35.     # I generate the auxiliary data structures to build the sparse representation of the characteristic matrix
36.     for document in range(0, len(documents)):
37.         for shingle in documents[document]:
38.             data.append(1)
39.             # each row represents a shingleId
40.             rows.append(shingle_dic[shingle])
41.             # each columns correspond to a document
42.             cols.append(document)
43.
44.     # I transform each document from list of shingles to list of shingleIds
45.     documents = [[shingle_dic[shingle] for shingle in document] for document in documents]

        # Initialization of the characteristic document_matrix
        document_matrix = csc_matrix((data, (rows, cols)))

        shingling_time = time.time() - shingling_time
        print("Shingling completed.")
```

This part of code uses the Shingling class to turn documents into numerical sets. After backing up the textual documents we map each document into the set of its composing shingles. We also create a set containing all the unique shingles encountered in the parsing of the document and we map them to an Id using a dictionary.

Between line 16 and 29 the characteristic matrix is built as a scipy sparse matrix stored by column. After this each documents is mapped from list of shingles to list of the corresponding shingle ids.

I. Main function - MinHashing phase:

```
    # I instatntiate a minHasher and an empty list to contain the signatures
minHasher = MinHashing(len(shingles), args.signature_length)
signatures = []

# I start the minHashing procedure
print("MinHashing Phase:")
min_hashing_time = time.time()
pbar = ProgressBar()

# Iterate on the columns of the characteristic matrix (each column corresponds to a document)
for i in pbar(range(0, len(document_matrix.indptr)-1)):
    # Defining the limits of the iteration on the indices
    min_ = document_matrix.indptr[i]
    max_ = document_matrix.indptr[i+1]
    # I append to the signatures the signature of the document as a minHash of the shinglesId it contains
    signatures.append(minHasher.minHash(document_matrix.indices[min_:max_]))

I build the signature matrix as an numpy matrix
natures = np.asarray([np.asarray(signature) for signature in signatures])

_hashing_time = time.time() - min_hashing_time
```

After initialising a MinHashing object we iterate on the columns of the characteristic matrix to generate the signature starting from the shingles of the document corresponding to the column.

J. Main function - Local Similarity Hashing phase:

```
    similar_docs = []


    lshasher = LocalitySensitiveHashing(buckets_num=Primes.firstAfter(len(shingles)))

    print("LSHashing Phase:")

    lsh_comparison_time = time.time()

    sig_size = len(signatures)
    for i in range(0, sig_size):
        lshasher.lsHashing(signatures[i], i)

    to_test = lshasher.getPossibleMatches()

    for candidate_one, candidate_two in to_test:
        minHashSimilarity = minHasher.compareSignatures(signatures[candidate_one], signatures[candidate_two])
        if minHashSimilarity >= threshold:
            jack = CompareSets.JaccardSimilarity(documents[candidate_one], documents[candidate_two])
            similar_docs.append((candidate_one, candidate_two, minHashSimilarity, jack))

    print("LSHashing completed.")
    lsh_comparison_time = time.time() - lsh_comparison_time
```

After initialising an object of the LocalitySensitiveHashing class we fill the buckets by calling the lsHashing function on all the signatures and we get the colliding hash as candidate similar documents to compare using the getPossibleMatches function.

Iterating on all the candidate pairs we compute the minHashSimilarity of each pair and if it overcomes the threshold given as input we compute the Jaccard similarity of the two documents and append the pair to the list of similar documents.

K. Main function - Local Similarity Hashing phase:

```
    print("Evaluation phase:")
    avg_diff = 0
    std_deviation = 0

    if len(similar_docs) > 0:

        for similar_doc in similar_docs:
            avg_diff += math.fabs(similar_doc[3] - similar_doc[2])

        avg_diff = avg_diff / len(similar_docs)

        for similar_doc in similar_docs:
            std_deviation += math.pow(math.fabs(similar_doc[3] - similar_doc[2]) - avg_diff, 2)

        std_deviation = std_deviation / len(similar_docs)

    print("Evaluation completed.")

    print("RESULTS:{} similarity pairs,with {} average difference ({} std. deviation) between estimated and jaccard similarities"
        ).format(len(similar_docs), avg_diff, std_deviation)

    return  num, len(similar_docs), avg_diff, std_deviation, shingling_time, min_hashing_time, lsh_comparison_time
```

If at least one couple of similar documents has been found we compute the average difference between the minHashing estimated similarity on signatures and the Jaccard similarity computed on the whole document. We print the results and we return the number of analysed documents, the average difference, the standard deviation and the time spent in each phase.

The data resulting from the run of the main function on a dataset of news articles is going to be shown in the section Analysis of the results.

L. Installation, usage and arguments:

To use the program is necessary to create an environment with Python3.5, argparse, numpy, scipy, progressBar and matplotib installed. To run the code is sufficient to run the command

"python completePipeline.py"

In the folder of the project. A folder *data* should be present in the same folder of the project, containing files with the following naming convention "*articles_NUMOFARTICLES.train*" containing articles stored one per line. The dataset is provided with the source code.

It's possible to launch the program with some input parameters:

1. —k_shingle: the length of each shingle. Default value is 10.
2. —signature_length: is the size of the signature that is generated for each document. Default is 10.
3. —threshold: is the minimum required similarity to consider two documents as similar. Default is 0.5.
4. —word_shingling: is a boolean parameter deciding wether the shingling is made by word or by characters. Default is false, corresponding to character shingling.
5. —documents_directory: is the directory in which the .train files are stored. The default value is "data".

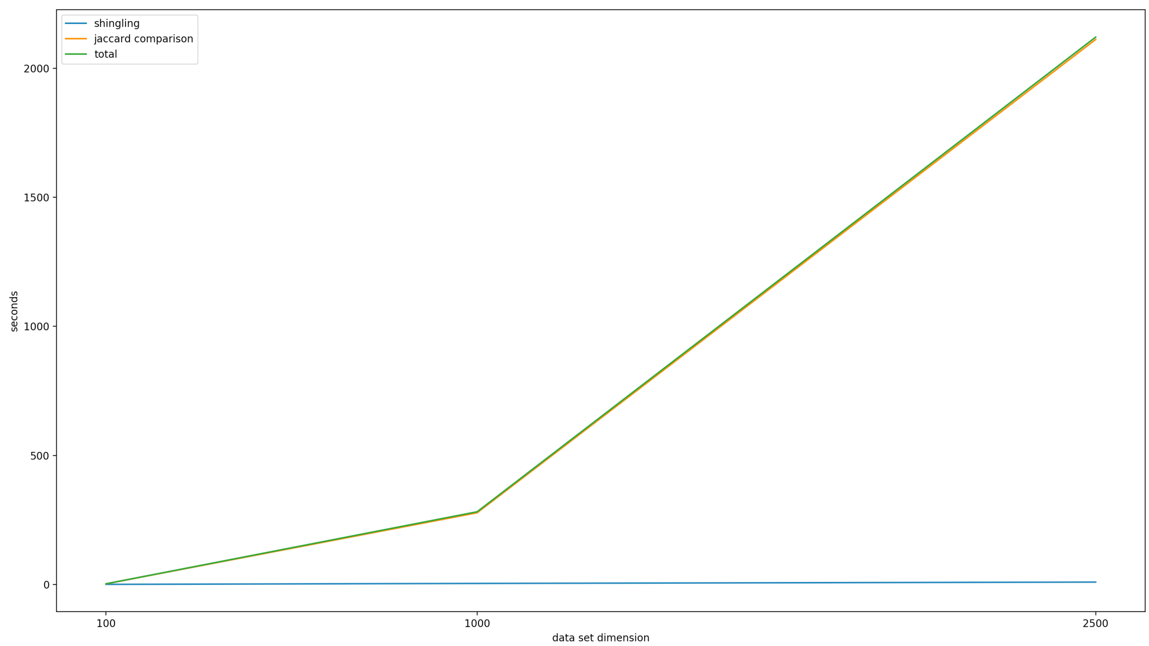
It's also possible to run the command:

"python completePipeline_faster.py"

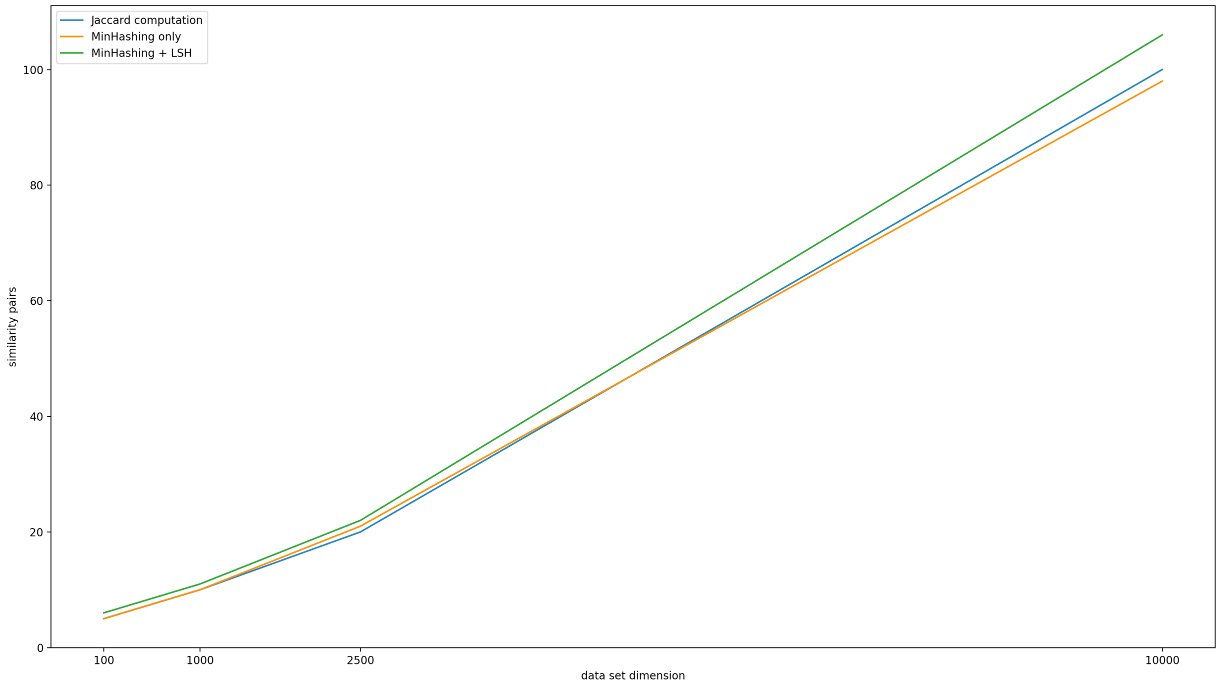
To run a second version of the code in which the characteristic matrix is not built and the min hashing is performed directly iterating on the list of documents. This second version is faster than the normal one as it's shown in the section Analysis of the results.

M. Analysis of the results:

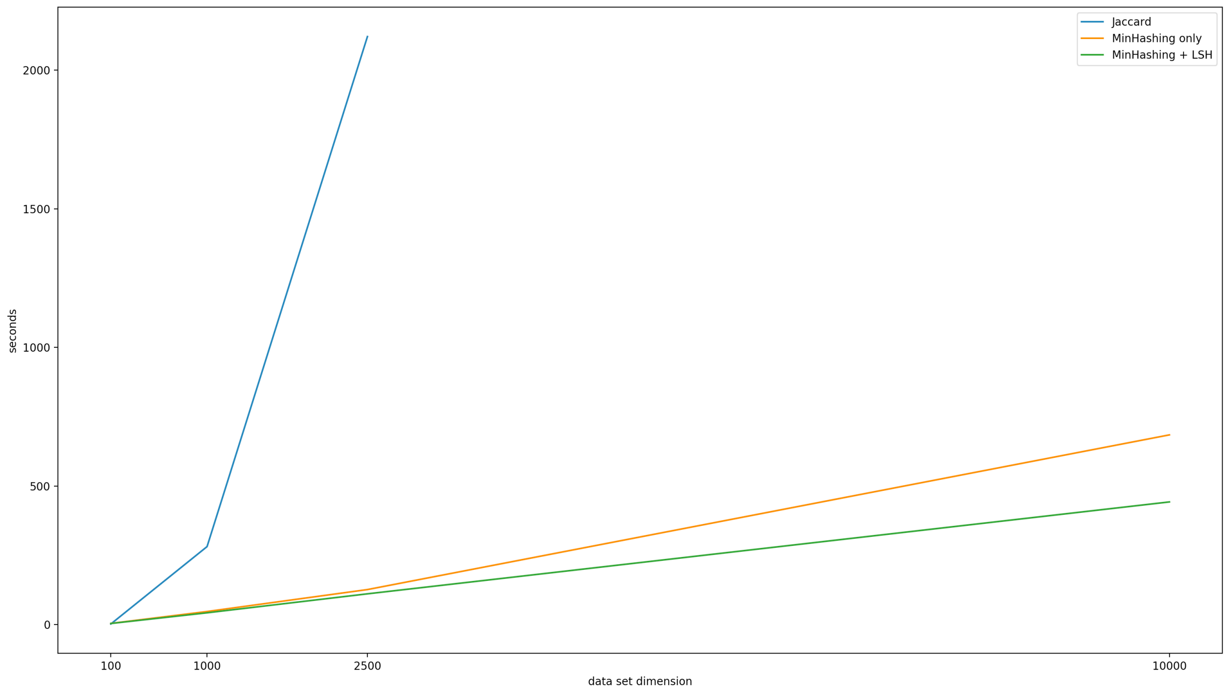
The pipeline of Shingling -> MinHashing -> LocalitySensitiveHashing has been applied to set of documents of increasing size (100, 1000, 2500, 10000) to check the scalability of the algorithm.



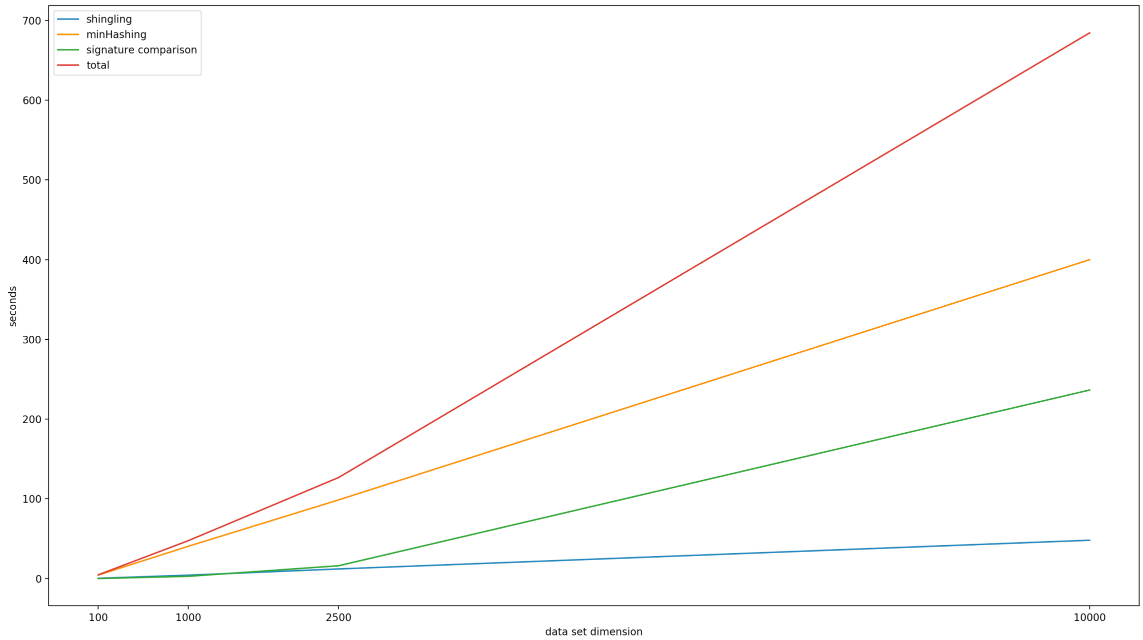
This first graph shows the time spent in the various phases while applying directly Jaccard similarities to the shingle sets of the documents. Data show that time performances degenerate according to an high order function of the input size, due to the high computational cost of the Jaccard comparison.

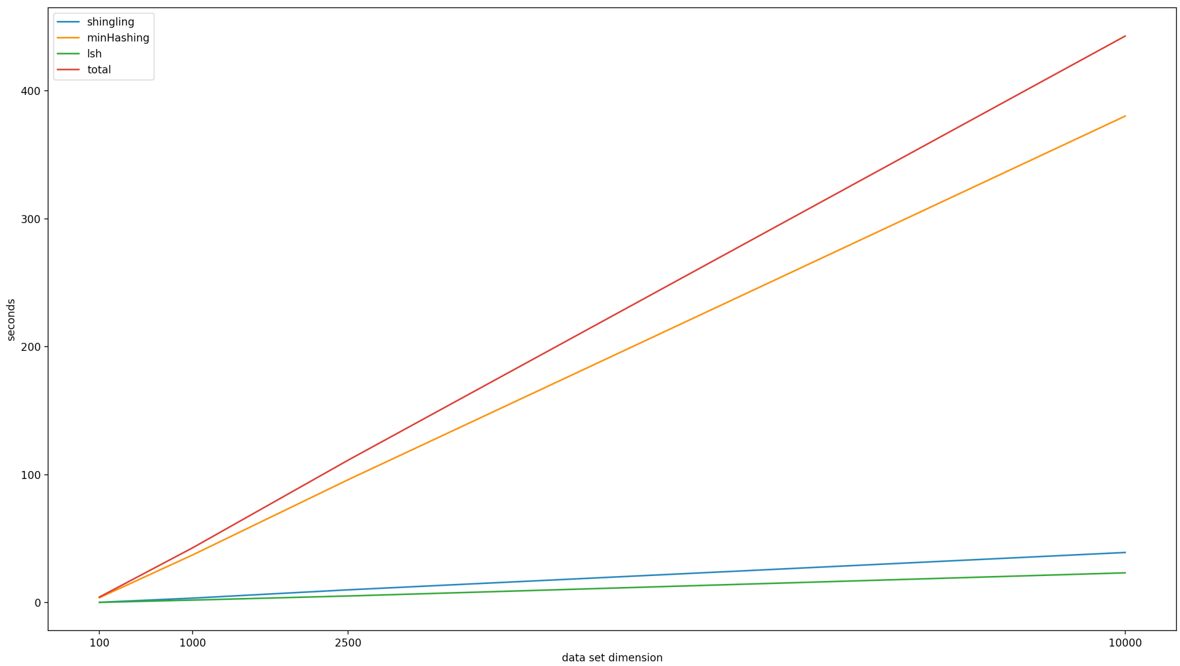


The accuracy is not penalised by the application of the method, as a matter of fact similar results are getting with or without approximating the document with its signature, and with or without using the locality sensitive hashing step.

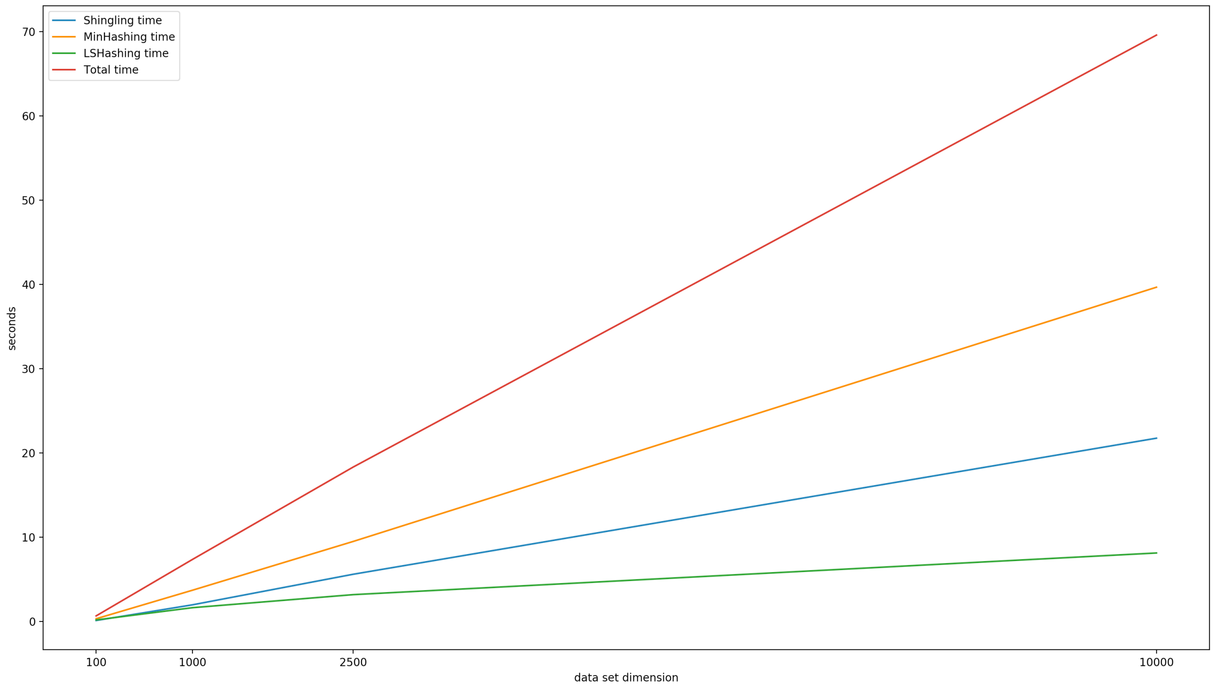


From the graph above it's clear that the scalability of the Jaccard similarity is not suitable even for relatively small documents and that the method of Shingling + MinHashing + LSH performs orders of magnitude faster even on small dimensions. Between the application of LSH or not we see that the speed gain grows with the size of the dataset. The advantage of the application of the last LSH phase becomes bigger and bigger when more documents have to be compared.





The two graphs above represent the partial run times on increasing size input, of the pipeline respectively with or without LSH phase. In case we don't apply the LSH phase we see that the comparison of all possible pairs of document introduces a super linear factor of time complexity.



Finally the last graph shows the total time and the time employed in each phase using the faster version of the algorithm (“completePipeline_faster.py”). The time performance is improved of a factor 6x.