Homework 1 - Finding Similar Items: Textually Similar Documents

By Group 16: Antonios Mantzaris & Ya Ting Hu

Description

The goal of this project is to find textually similar documents. The implementation is done in Python with the use of Jupyter Notebook on the following dataset https://storage.googleapis.com/dataset-uploader/bbc/bbc-text.csv regarding BBC news articles which was extracted from Kaggle[1]. Initially, five sample classes were implemented for different stages of finding textually similar documents, namely:

1. ***Shingling*** that constructs k–shingles of a given length k from a given document, computes a hash value for each unique shingle, and represents the document in the form of an ordered set of its hashed k-shingles;
2. ***CompareSets*** that computes the Jaccard similarity of two sets of integers (in this context: two sets of hashed shingles);
3. ***MinHashing*** that builds a minHash signature in the form of a vector of a given length n from a given set of integers (in this context: a set of hashed shingles);
4. ***CompareSignatures*** that estimates similarity of two integer vectors (in this context: minhash signatures) as a fraction of components, in which they agree;
5. and ***LSH*** that implements the Locality Sensitive Hashing (LSH) technique, that is, given a collection of minhash signatures (integer vectors) and a similarity threshold t, the LSH class (using banding and hashing) finds candidate pairs of signatures agreeing on at least fraction t of their components.

The final implementation is a Python script which is run via terminal (see instructions below). The libraries used are ***numpy***, ***pandas***, ***random***, ***math***, ***io***, and ***itertools***.

Instructions

In this section, the functions in each class and the commands to run the final script are briefly discussed.

[1] https://www.kaggle.com/chiranjibpatra/bbc-text-categorization/data

How to Build

For each of the classes mentioned previously functions are created. For the class ***Shingling,*** firstly there is the function ***shingles(****input_text,* k***)***, that given an input text and an integer k, transforms the input text into a set of k-shingles. Then ***ordered_hash(****S****)***, given S the set of k-shingles, hashes each element of S (which is a string) to an integer value and returns this ordered list (with the properties of a set) after ordering.

For the class ***CompareSets***, the function ***Jaccard(****A, B****)*** given two sets of integers A and B, returns the Jaccard similarity of A and B by dividing $|A \cap B|$ over $|A \cup B|$.

For the class ***MinHashing*** the function ***hashing(****I, n****)***, given a set I of hashed shingles and an integer n, hashes each element from the set I to an integer (i.e. hashcode) and takes the minimum value of all the hashcodes. This is repeated n times so that it is left with a vector of length n consisting of all the minimum hashcode values. This vector, also called the minHash signature, is then returned.

For the class ***CompareSignatures*** the function ***estimate(****$\bar{u},\bar{v}$****)***, given vectors $\bar{u}$ and $\bar{v}$ which represents the minhash signatures, estimates the similarity of $\bar{u}$ and $\bar{v}$ as a fraction of components by calculating the occurrence of the same element in both $\bar{u}$ and $\bar{v}$ given the same index divided by the total length of $\bar{u}$ or $\bar{v}$.

Lastly, for the class ***LSH*** the function ***candidate(****M, n****)***, given a collection of signatures M and number of bands n, the number of rows is defined to be the signature length divided by n. Then from M a list of candidate pairs is generated, i.e. pairs of elements whose similarity must be evaluated, by using banding and hashing the columns of M to many buckets and making elements of the same bucket candidate pairs. Lastly, the ***check_similarity(****l, t****)*** function, given a list of final candidates l and a similarity threshold t, uses ***CompareSignatures*** to find the candidate pairs of signatures agreeing on at least fraction t of their components.

How to Run

Requirements are Python 3.8.5 and the libraries ***numpy***, ***pandas***, ***random***, ***math***, ***io***, and ***itertools***. There is a python script called ***classesFile.py*** which contains all the five classes and a python script ***Homework1.py*** including extraction of the dataset, calling the classes, and measuring the execution time based on the user input.

Open the terminal and go to the directory in which the aforementioned scripts are located. Run the python script ***Homework1.py*** by the following command line in terminal:

```
python Homework1.py
```

It prompts the following *Input number of documents to be compared (max:100)* as the user input. In Figure 1 an example is shown for the user input number of documents to be compared equal to 5.

```
Input number of documents to be compared (max:100):5
number of candidates= 1
Column 3 and 4 have Jaccard similarity: 0.3 which is lower than
0.6 so they are not similar
Column 0 and 1 have Jaccard similarity: 0.3
Column 0 and 2 have Jaccard similarity: 0.25
Column 0 and 3 have Jaccard similarity: 0.25
Column 0 and 4 have Jaccard similarity: 0.35
Column 1 and 2 have Jaccard similarity: 0.1
Column 1 and 3 have Jaccard similarity: 0.25
Column 1 and 4 have Jaccard similarity: 0.3
Column 2 and 3 have Jaccard similarity: 0.15
Column 2 and 4 have Jaccard similarity: 0.25
Column 3 and 4 have Jaccard similarity: 0.3
#similar= 0 specifically: [] from them 0 were found by candidates
(for finding errors in candidates)
--- 0.3929173946380615 seconds ---
```

Figure 1: *Results displayed in terminal after execution of the **Homework1** python script for user input number of documents equal to 5, where the number of candidates, Jaccard similarity of each pair of documents, number of similar pairs, and execution time are shown.*

If it is desired to not execute all classes simultaneously, the user can use the Jupyter Notebook by running the cells of the desired classes and/or functions instead.

Results

In order to test and evaluate scalability i.e. the execution time versus the size of the input dataset, the number of documents is increased and the execution time is measured, both with and without the LHS class. The results are shown in Table 1 and Figure 2.

#documents	#candidates	#similar pairs	time: no LHS	time: with LHS
2	0	0	0.107s	0.090s
5	1	0	0.219s	0.127s
10	3	0	0.518s	0.178s
20	31	0	1.676s	0.413s
50	662	7 (6 discovered from LHS candidates)	11.940s	4.436s
75	1358	12 (10 discovered from LHS candidates)	26.030s	8.895s
100	2415	34 (19 discovered from LHS candidates)	44.707s	15.221s

Table 1: *Scalability given by comparing the execution time with the size of input dataset i.e. number of documents, both with and without the Locality Sensitive Hashing (LHS) class, and the number of candidates and similar pairs found*.

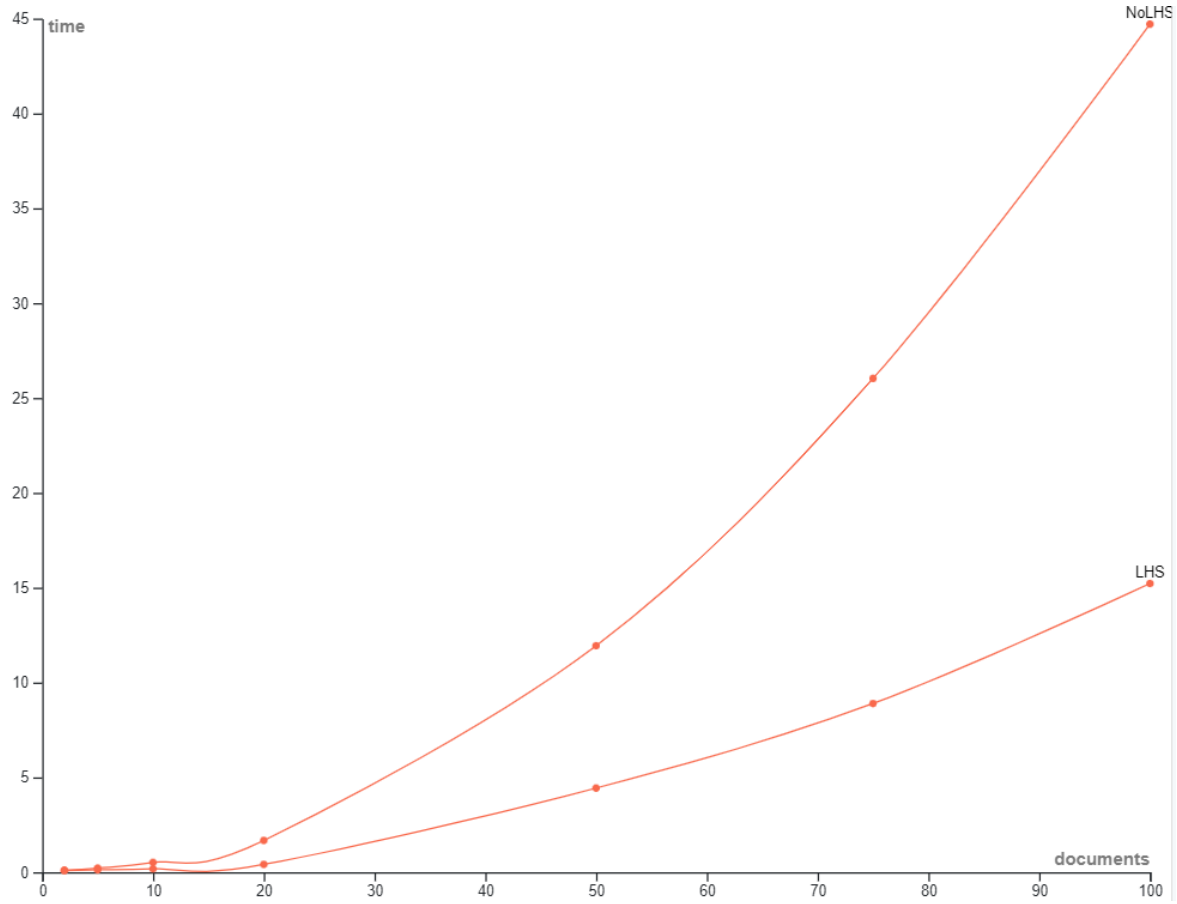

Figure 2: *Scalability given by comparing the execution time (y-axis) with the size of input dataset i.e. number of documents (x-axis) of the implementation, both with and without the Locality Sensitive Hashing (LHS) class.*