Big Data Computing **A.Y. 2022/2023**
Homework 2 **Deadline: November 18th**

Instructions. You should hand in your homeworks within the due date and time. Late deliveries will be penalized, please check homework delivery policies in the exam info.

Handing in. You should submit your work as *one zip file* containing: i) one folder with your code for the assignment, clearly organized; ii) a brief report (say 1-2 pages), clearly explaining 1) whatever is needed to understand your code and run it, 2) the implementation choices you made and 3) the results of the testing experiment. It is perfectly fine if your code is contained in a *clearly commented* Colab notebook addressing points 1), 2) and 3) above. In this second case, you do not need to write a separate report. However, it is important that the code and your choices are adequately commented and explained and that I can run the code myself. Part of the score will depend on this.

Please deliver your homework by attaching the above zip file to an email addressed to me as follows:
To: `becchetti@diag.uniroma1.it`
Subject: `HW2 <Your last name> <Your first name> <Your Sapienza ID number>`

Typesetting in Latex. Latex is very handy for typesetting (especially math), but you need to install it. If you do not want to install Latex, you can go for Overleaf, providing an integrated, Web interface, accessible for free in its basic version (which is enough for your needs). It allows you to both type in Latex using a Web interface, and compiling your code to produce a pdf document. Overleaf's documentation also contains a tutorial on Latex essentials.

Important note. Grading of your answers to the theoretical assignments will depend on i) correctness and solidity of the arguments, ii) clarity of the presentation and iii) mathematical rigour. E.g., ill-defined quantities, missing assumptions, undefined symbols etc. are going to penalize you. Rather than writing a lot, try to write what is needed to answer and write it well.

Assignment 1.

The goal of the first assignment is to use LSH for cosine similarity. In this case, you are requested to write a Colab notebook that applies LSH to textual data. The general goal is, given one document, to efficiently identify the documents that are closest to the document under consideration, in terms of cosine similarity computed on the tf-idf representations of the documents. As a test case, we will use the relatively large Amazon Fine Food Reviews dataset, available at `https://www.kaggle.com/datasets/snap/amazon-fine-food-reviews` and already used in a previous notebook. Please

refer to the Colab notebook on exploratory analysis of the above dataset for dataset retrieval and text feature extraction. In more detail you should:

- Implement all necessary text prepropressing required to transform each review into a tf-idf vector. To this purpose, you can reuse (and improve!) the code available in the above mentioned notebook on exploratory analysis of the Amazon Fine Food Reviews dataset.
- Implement a class that, given a collection of tf-idf vectors (each representing a review), returns the collection of signatures of the vectors, where each signature is computed by applying the hyperplane method as seen in class. The length of the signature should be $m = r \times b$, where as usual r and b are the number of rows and bands (see next point) of the signature matrix and are input parameters.
- Implement a LSH class implementing the banding technique seen in class (note that the number of dictionaries to use is equal to b).

Testing. *Remove* a small (and uniform) sample of the reviews from the dataset (say, 0.5%, which corresponds to more or less 2500 reviews). This will be your test set, while the rest of the dataset will be your training set. For each test review: i) compute the *exact* subset A of reviews in the training set, whose cosine similarity with the review under consideration is above θ; ii) compute the same subset, for the same threshold, this time time approximately using LSH. In general, this will be a subset B that is different from A (of course, we would like B to be as close to A as possible); iii) compute $Jacc(A, B)$, as a measure of how A is close to B; iv) keep track of the times needed to compute A using exact method and B using LSH. You should report averages (over the number of test reviews) of the values computed in ii) - iv).

Note that exact similarity computation is already available in python libraries.

Fixing the threshold. Fixing the threshold θ is necessary to identify near neighbours and to fix the values of r and b. To this purpose, you need to get a feeling of the average pairwise cosine similarity between documents. A rudimentary way to proceed is the following (but you can think of something more principled of course): i) sample a small fraction of the reviews (e.g., 1% of them) uniformly at random and compute the average pairwise cosine similarity between them (note that this is likely to be a relatively small value); ii) fix the threshold θ to a value that is a few times larger than average pairwise cosine similarity, e.g., 5 times or more. Note that if the threshold is too high you might have very few (or no) near neighbours for most documents, while you might have too many if it is too low.

Hints and suggestions. You will need some hash functions that maps vectors into signatures. It might be something similar to this, where `inp_vector` and `inp_dimensions` respectively denote an input vector and the number of its dimensions:

```
class HashTable:
    def __init__(self, hash_size, inp_dimensions):
        #Other initializations ...

        self.projections = np.random.randn(self.hash_size, inp_dimensions)

    def generate_hash(self, inp_vector):
        bools = (np.dot(inp_vector, self.projections.T) > 0).astype('int')
        return ''.join(bools.astype('str'))

    #Possibly other functions ....
```

Note that this way, if we use k hash functions, the signature of each vector is a string corresponding to k binary values. If each character (`1` or `0`) requires 1 byte, the signature requires k bytes. This kind of considerations can be useful to avoid your signature matrix being unnecessarily large in terms of memory requirements.

Another thing you may want to consider in a practical implementation: for each document (i.e., its tf-idf vector), you can produce b signatures of r bits each if you have $m = r \times b$. This is perfectly equivalent to producing a unique signature of m bits and then splitting it into b chunks. In this way, however, the signature of each document is now a set of b strings, each of length r and corresponding to a different band. Moreover, each of these strings can be used a key for a different dictionary (remember that, when using the banding technique, you need to maintain a separate dictionary/hash table for every band).

Important note. Please check the collaboration policy on the course web page.