Fully dynamic k-center clustering

Project report

Adèle Mortier

February 26, 2018

Introduction

The aim of this class project was to implement an algorithm that maintains efficiently and cheaply a clustering for GPS-located tweets that had been collected on Twitter. The algorithm was based on a popular method called k-center clustering (or k-means clustering). Contrary to a standard k-means problem, the data were dynamic in the sense that at each step, the oldest tweet was removed from the dataset and the newest was added. For our implementation, we used Python and the following combinations of parameters :

Variable	Name	Value
Number of clusters	`k`	$[15, 20]$
Dataset size	`window_width`	$[10000, 15000]$
Number of updates[1]	`n_operations`	60000

1 Project description

1.1 Preprocessing

The data had been stored in a `txt` file, each line of the file representing a tweet (*i.e.* a timestamp, a latitude and a longitude). We parsed this file and put the data in a `h5` file[2] to have faster I/O for our later queries on the dataset. We also put a unique *id* (basically, an `int`) on each tweet to manipulate them more abstractly and efficiently. The data (latitude, longitude) behind each tweet *id* could be easily retrieved using a Python dictionary.

1.2 Computation of the dataset parameters

Before doing the clustering, we had to retrieve two parameters from our dataset, which were :

$$d_{min} = min_{x_1 \neq x_2 \in S}\left(dist(x_1, x_2)\right)$$
$$d_{max} = max_{x_1 \neq x_2 \in S}\left(dist(x_1, x_2)\right)$$

S being the set of tweets defined by their GPS coordinates. We wanted $dist$ to be a relevant metric w.r.t our data, so we used a custom metric called the Haversine distance that computes the distance (in kilometers) between two points defined by their GPS coordinates.
To compute d_{min} and d_{max}, we first tried a brute-force method that consisted in using the Scipy `pdist` function that computes the pairwise distance for all points i a dataset[3]. But the function ran out of memory when computing on the 1 million tweets.
We thus chose more refined methods, *i.e.* Delaunay triangulation [2] for d_{min} and Rotating calipers [3] for d_{max}. We took code snippets from here and here and adapted them such that we could deal with the Haversine distance instead of the Euclidean one. The Delaunay (D) algorithm and the Rotating calipers (RC) algorithm returned the following results :

Algorithm	Expected result	Value (km)
D	d_{min}	$8.50586308516 \times 10^{-7}$
RC	d_{max}	18680.1424983

The values compiled above look consistent with our metric and the Earth topology : we expected the upper bound of d_{max} to be about 20000 km[4], and the lower bound of d_{min} to be close to zero[5]. With these values is was then easy to compute the different values of β for each value of ϵ

[2] using the package `h5py`
[3] and then we would have taken the minimum and the maximum of these distances
[4] half the Earth's circumference
[5] tweets written by the same person...

1.3 Algorithms

We implemented the algorithms as specified in [1]. More precisely :

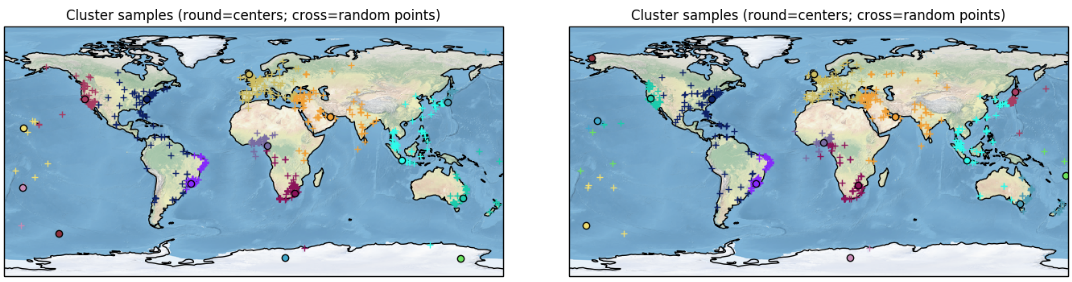
- for a fixed β, a (β-)clustering is a Python `dict`, whose keys are the tweet *ids*[6], and whose values are the centers *ids*. Each tweet *id* is then mapped to the cluster it belongs to (identified by the *id* of its center), or to -1 if it is not yet clustered.
- for a set of β_s, a β_s-clustering is a `dict` that maps each β to its corresponding clustering.
- for a set of ϵ_s, an ϵ-clustering is a `dict` that maps each value of ϵ to its corresponding β_s-clustering

We used the `dict` structure for two reasons : first the `dict` structure is more modular[7], second because of the good time complexities of this structure[8]

2 Analysis

All the plots and tables are in the Appendix. As we look at Figure 1, the algorithm seems to work well, and the Haversine metric allows to see a real geographical clustering, with areas that coincide with the different continents and the regions within them that present a huge demographic density : US East- and Westcoasts, Europe, East-Asia, Middle-East, South-Africa, South America... We can notice that the partition is quite robust regarding reclustering after center deletion. The clusters that keep the same color have not been reclustered, whereas there is a brand new cluster located in Japan, and for instance the Australian cluster has moved a bit.
As we look at the plots of Figure 2, we can observe that the execution time is very sensitive to the precision parameter ϵ. Indeed, a smaller ϵ (*i.e.* a higher precision) forces us to compute a clustering for more β_s...

Appendices



(a) Map of the initial (static) clustering for the best ϵ and the best β

(b) Map after the first reclustering (here operation 821) for the best ϵ and the best β

Figure 1: Spatial representation of the clustering before and after a reclustering operation

| bla... |

References

[1] Hubert Chan, Arnaud Guerquin, and Mauro Sozio. Fully dynamic k-center clustering. Technical report, 2017.

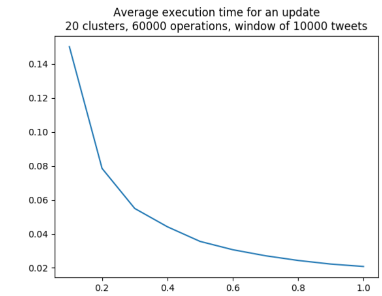
[2] Boris Delaunay. Sur la sphère vide : à la mémoire de georges voronoï. *Bulletin de l'Académie des Sciences de l'URSS, Classe des sciences mathématiques et naturelles*, 6:793–800, 1934.

[3] Michael Shamos. *Computational Geometry.* PhD thesis, Yale University, 1978. pp. 76–81.

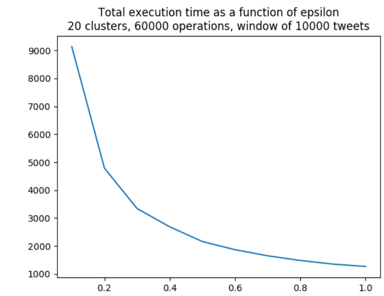
[6] for the tweets that belong to the current dataset

[7] for instance if we wanted to do something else than the sliding window model, *e.g.* add/remove the nodes in random order (and hence by random *id*), a `dict` would be much more convenient (w.r.t a `list`) to remember which point belong to which cluster...

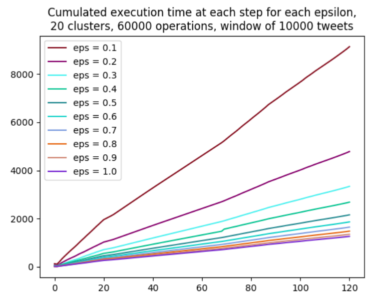
[8] for `get` and `set`, both the `dict` and the `list` are $O(1)$ on average. `dict` is better on average for `delete`, with a $O(1)$, *vs* $O(n)$ for the `list`. That is good for us because we have to do as many deletions as insertions in our framework. Conversely, we do not use so many times `get` and `set`, where the `list` is slightly better (in worst case).



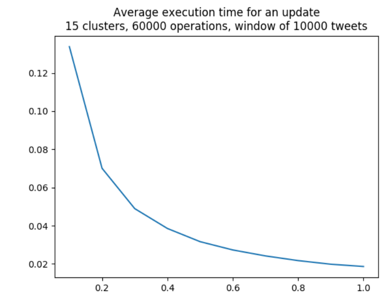
(a) Average execution time per update as a function of ϵ



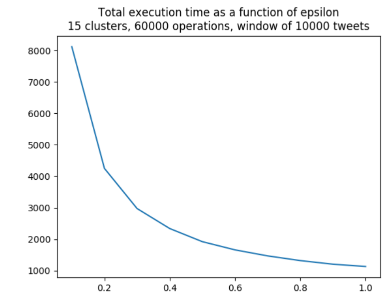
(b) Total execution time (static and dynamic clustering) as a function of ϵ



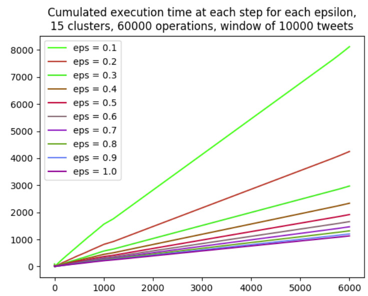
(c) Evolution of the execution time (sampled each 500 operations)



(d) Average execution time per update as a function of ϵ



(e) Total execution time (static and dynamic clustering) as a function of ϵ



(f) Evolution of the execution time (sampled each 500 operations)

Figure 2: Experimental results for several sets of parameters