X2AI1100 – 算法分析与设计 / Algorithm Design and Analysis

– *K*-means clustering –

211.561.13.752 – louis tomczyk

January 9, 2022

Contents

Foreword

In this document, different environments will be displayed.

```
Here is some code...
```

> Here is some examples.

Such font is used each time a technical word is used and for function names.

The chosen programming language is : Python3.8.

The IDE is SPYDER : https://www.spyder-ide.org/.

The computer RAM size is :15.5 GiB.

The computer processor is : Intel® Core™ i7-10510U CPU @ 1.80GHz × 8.

The OS is : Ubuntu 20.04.1 LTS / 64-bit.

Introduction

I chose to study the k-means clustering algorithm I implemented in the Machine Learning course. In this work we will first remind the objective of such algorithm. Then we will study the complexity of the algorithm. Finally we will see what improvements can be made.

1 k-Means

1.1 Introduction

In unsupervised learning, data are not labelled, which means that we do not know to what they correspond to. One may want to group the data into groups with common properties. These groups are called clusters.

The steps followed by the algorithm are given below.

1/ **centroids initialisation :** choose the initial centroids. Different methods exists. Here we randomly choose centroids among the dataset points with the function init_centroids.

2/ **cluster assignation :** assign points to clusters by computing the distance between each point and each centroid, with the function assign_cluster.

3/ **centroids computation :** compute the gravity center of each cluster with the function update_centroids.

4/ **centroid movement :** move the centroids to the new gravity center with the function update_centroids.

5/ **cost evaluation :** compute the cost function with the function cost_function. If its value is lower than a threshold, we break the loop. A second condition is needed, the maximum number of iterations, to avoid infinite loop if the algorithm is stuck.

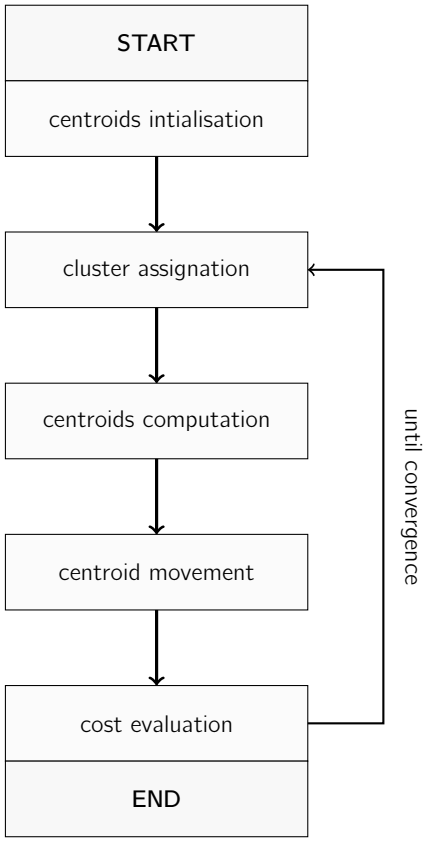


Figure 1: $\kappa-$means algorithm

The figures 2,3,4,5 (from [7]) shows how the algorithm is affecting the centroids and clusters. From figure 4 to5 we can notice that the centroids are getting closer to the barycenter of the expected clusters.

Steps $(0, 1, 3, 5)$ are the one that be improved which will be discussed in the last part. However in the following subsection we will describe more precisely the method implemented.

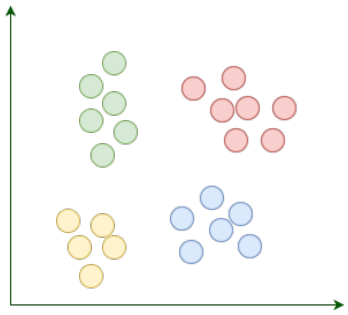


Figure 2: original data set

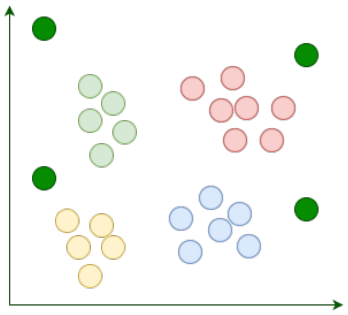


Figure 3: step 1 – initial centroids

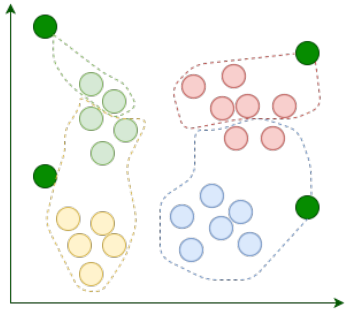


Figure 4: step 2 – 1^{st} assignation

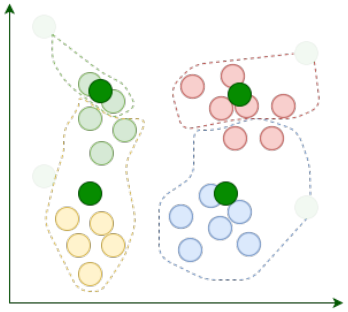


Figure 5: step 4 – 2^{nd} assignation

1.2 Step 0 - hyperparameter setting

For now we only choose a random number of clusters such as :

$$\boxed{k \in [\![1, n\_samples]\!]} \tag{1}$$

Where ($n\_samples$) is the number of points in our dataset. The two extreme cases $k \in \{1, n\_samples\}$ are useless, but possible. In the former, we consider that all the samples belong to the same cluster. Which is the orginal case as data should come from only one set.

The latter is useless as we want to group different samples. However we can mention that this case is the orignal case for the hierarchical clustering algorithm which shows a complementary approach to k-Means.

Eventually, usually $k \in \left[\!\left[2, \left\lfloor \frac{n\_samples}{2} \right\rfloor\right]\!\right]$ as $k = \left\lfloor \frac{n\_samples}{2} \right\rfloor$ will only give two (or maximum three) samples per cluster.

1.3 Step 1 - init_centroids

Here we implemeted a basic way to initialise which randomly selects (k) samples from the dataset to be the Initial Centroids (IC). We can notice, that this is not the initialisation method chosen in the figure 3.

From now we can guess that such method will shows the following complexities :

$$\text{time} \quad : \quad T_{IC} = \Theta(k) \tag{2}$$

1.4 Step 2 - assign_cluster

In order to Assign a Cluster (AC) to a point, we have to compute the distance between each centroid and each point, as shown in figure 6.

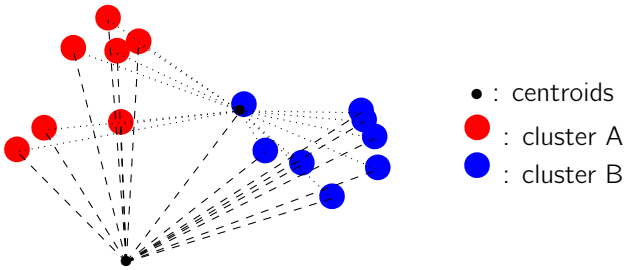


Figure 6: Distance calculations between each centroid and each data sample.

Different calculations methods exist, according to the statisitcal distributions of the coordinates or to the attribute format (ordinal or not) etc. Here we assume that the dataset contains only ordinal attributes. We implemented the function distance which enables to chose among different methods among the Minkowski distance family : the Manhattan (or block) and Euclidian distances. The latter is set by default. From now, we can guess that the distance calculation shows the following complexities :

$$\text{time} \quad : \quad T_{dist} = \Theta(n\_samples \cdot p \cdot k) \tag{3}$$

Where (p) is the number of coordinates (or the space dimension). Once distance computations are done, we have to look for the closest dataset samples from each centroid and then assign the clusters.

From now, we can guess that the closest dataset samples search shows the following complexities :

$$\text{time} \quad : \quad T_{closest} = \Theta(n\_samples \cdot k) \tag{4}$$

Finally, we can guess that the cluster assignment step shows the following complexities :

$$\text{time} \quad : \quad T_{AC} = T_{dist} + T_{closest} = \Theta(n\_samples \cdot p \cdot k) \tag{5}$$

1.5 Step 3 - cost_function

First, let define the notations :

$$\text{centroids} \quad : \quad \mathcal{C} = \{C_1, \cdots C_k\} \tag{6}$$
$$\text{clusters} \quad : \quad \mathcal{K} = \{K_1, \cdots K_k\} \tag{7}$$

where :

$$\forall \kappa \in [\![1, k]\!] \qquad C_\kappa \in \mathbb{R}^p \tag{8}$$
$$\forall \kappa \in [\![1, k]\!] \qquad K_\kappa = \{K_{\kappa,1}, \cdots, K_{\kappa,m_{K_\kappa}}\} \tag{9}$$
$$\tag{10}$$

For example, let us take an example with the figure 7. There are (4) centroids, so $\kappa \in [\![1, 4]\!]$. The (4) different clusters contain different number of elements.
(K_1) : contains (8) elements so $m_{K_1} = 8$.
(K_2) : contains (7) elements so $m_{K_2} = 7$.
(K_3) : contains (8) elements so $m_{K_3} = 8$.
(K_4) : contains (7) elements so $m_{K_4} = 7$.

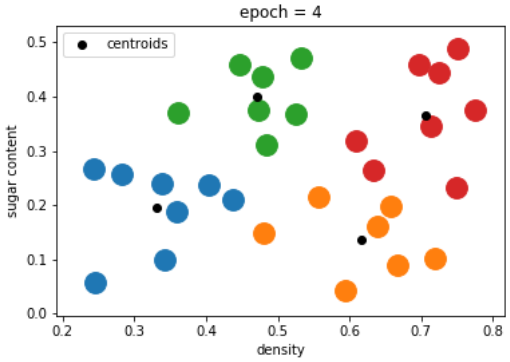


Figure 7: Example of k-Means iteration

This Cost Function (J) quantifies the convergence of the algorithm and is a given by 11[1] [2].

$$\forall (\mathcal{C}, \mathcal{K}) \in M_{k,p}(\mathbb{R}) \times M_{m_{K_\kappa},p}(\mathbb{R}) \qquad J(\mathcal{C}, \mathcal{K}) = \frac{1}{k} \cdot \sum_{\kappa=1}^{k} \left(\frac{1}{m_{K_\kappa}} \cdot \sum_{i=1}^{m_{K_\kappa}} d(C_\kappa, K_{\kappa,i}) \right) \in \mathbb{R}^+ \tag{11}$$

The Cost Function (J) must be decreasing as the distance should decrease. In order to guess, from the 11 formula, the complexities one may first notice that we have to compute the distance between each cluster's centroid and each cluster's element related to this centroid, for each cluster. Thus for each cluster we have (m_{K_κ}) operations. This number of operations is changing for each cluster, but we know that in total, there will be ($n\_samples \cdot k$) computations.

Thus we get :

$$\text{time} \quad : \quad T_{CF} = n\_samples \cdot k \cdot T_{dist}(1,1) = \Theta\left(n\_samples \cdot p \cdot k\right) \tag{12}$$
$$\tag{13}$$

The space complexity is constant as we store only one value : the cost.

1.6 Step 4 - update_centroids

Updating the Centroids (UC) boils down to change its (p) coordinates by the barycenter of the updated cluster, for the (k) clusters. Thus this step shows the following complexities :

$$\text{time} \quad : \quad T_{UC} = \Theta(n\_samples \cdot p \cdot k) \tag{14}$$

1.7 Step 5 - looping

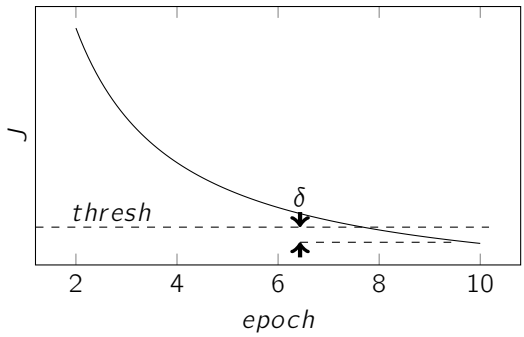
Usually we name ($epoch$) the variable used to count the number of iterations. The cost function should set the loop interruption condition. If, for a given ($epoch$), its value gets below a threshold ($thresh$), it breaks the loop.

[1] $d(\bullet_1, \bullet_2)$ stands for the distance between two elements $(\bullet_1, \bullet_2)$
[2] Taking only the sum instead of the mean is enough. It does not change the complexity anyway.

$$\text{loop break} \iff \exists\ epoch \in \mathbb{N}^* \ \setminus \ J(epoch) \leq thresh \tag{15}$$

However it may not reach it. Thus a second condition can be set. If the variations of the function gets below a threshold (δ), then the loop should also :

$$\nexists\ epoch \in \mathbb{N}^* \ \setminus \ J(epoch) \leq thresh \implies \text{loop break} \iff \exists\ epoch \in \mathbb{N}^* \ \setminus \ J(epoch+1) - J(epoch) \leq \delta \tag{16}$$



Eventually, if this is not enough, we can set an upper bound $epoch_{max} \in \mathbb{N}^*$ for the epochs such as :

$$\text{loop break} \iff epoch \geq epoch_{max} \tag{17}$$

With the study of each function, we saw that the complexities are function of $(n\_samples, p, k)$. letassume the complexity of the algorithm, for each loop, to be given by :

$$T_{kmeans,epoch} = \Theta\left[f(n\_samples, p, k)\right] \tag{18}$$

The impact of the two first cases will will be such as if the number of needed epoch is $epoch_{needed}$ then the complexity is:

$$T_{kmeans} = \Theta\left[f(n\_samples, p, k) \cdot epoch_{needed}\right] \tag{19}$$

However if the last condition is set, it will limit the complexity in any cases such as :

$$T_{kmeans} = \Theta\left[f(n\_samples, p, k) \cdot epoch_{max}\right] \tag{20}$$

1.8 Example

Here we give an example application. Let us take the watermelon dataset 4.0 made of 30 samples with two features : the density and the sugar content, units were not given. The dataset is shown in the figure 8.

The figure 9 was obtained with the following parameters.

```
k               = 2
init_method     = "random"
dist_method     = "euclidian"
max_epoch       = 10
thresh          = 1e-3
```

We can see with this example that the maximum number of iterations was useless as the 8 epochs were enough for reaching one of the two loop breaking conditions.

The figures 10,11,12,13,14, 15,16,17,18, show the different epochs.

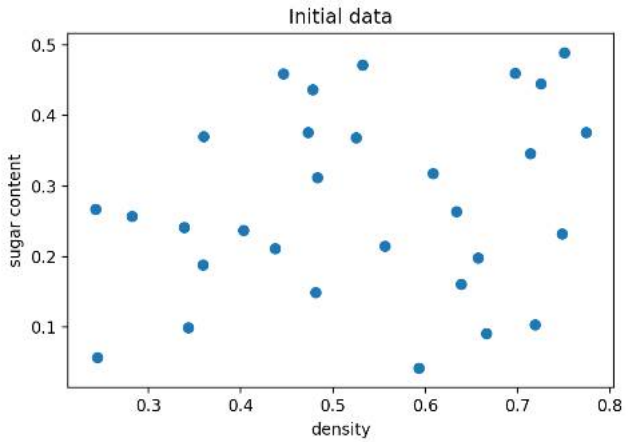


Figure 8: Watermelon dataset 4.0

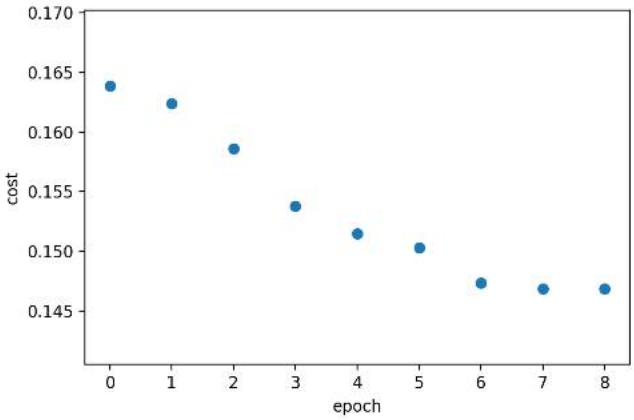


Figure 9: $epoch = 0$

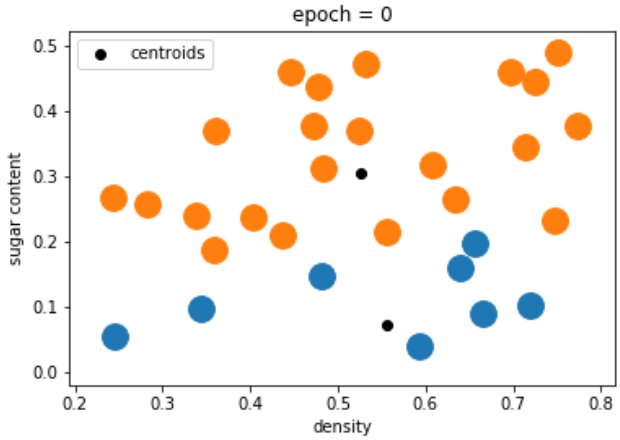


Figure 10: $epoch = 0$

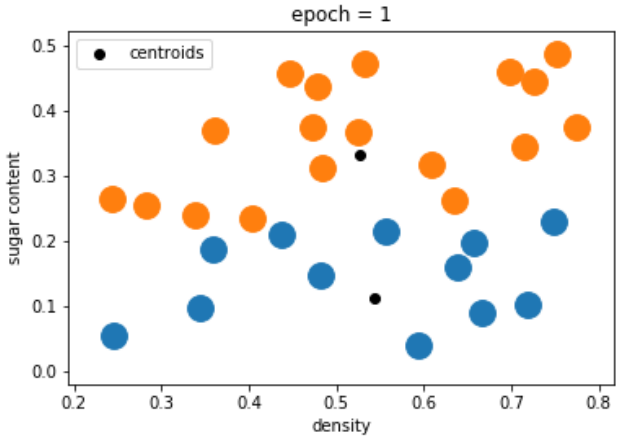


Figure 11: $epoch = 1$

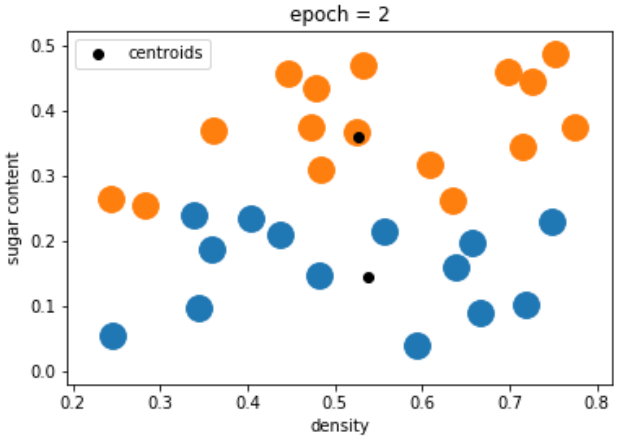


Figure 12: $epoch = 2$

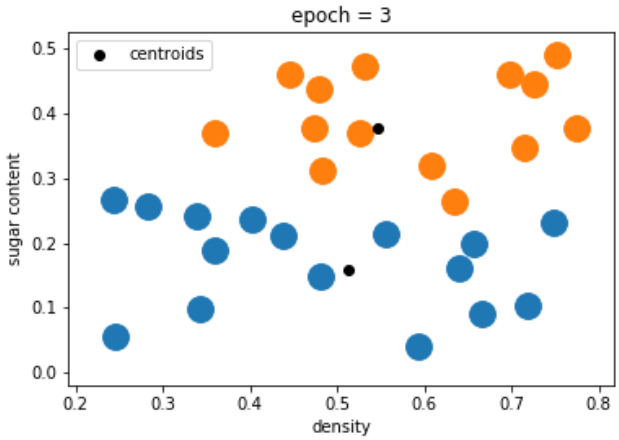


Figure 13: $epoch = 3$

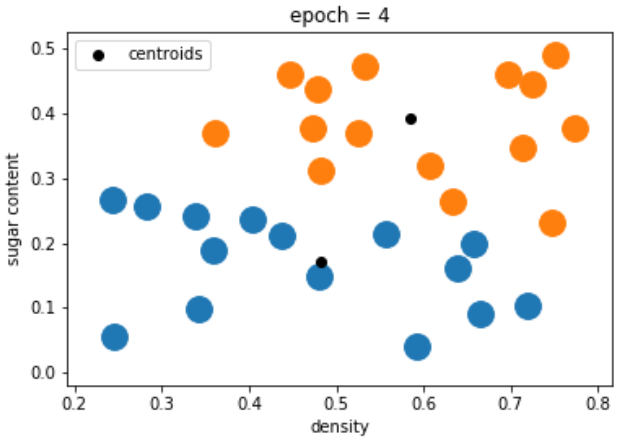


Figure 14: $epoch = 4$

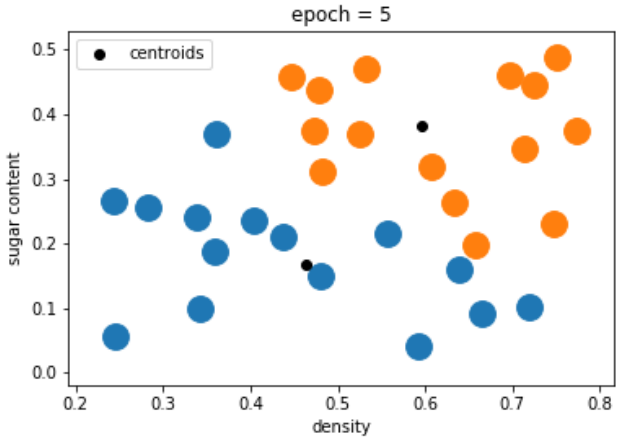


Figure 15: $epoch = 5$

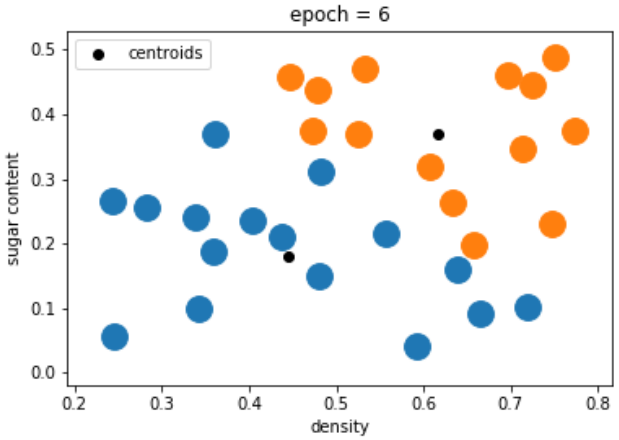


Figure 16: $epoch = 6$

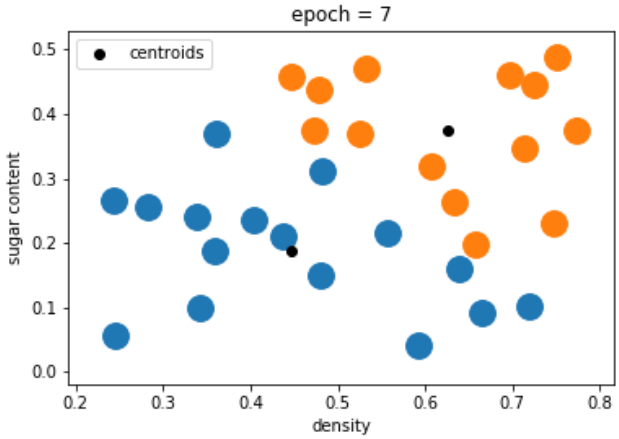


Figure 17: $epoch = 7$

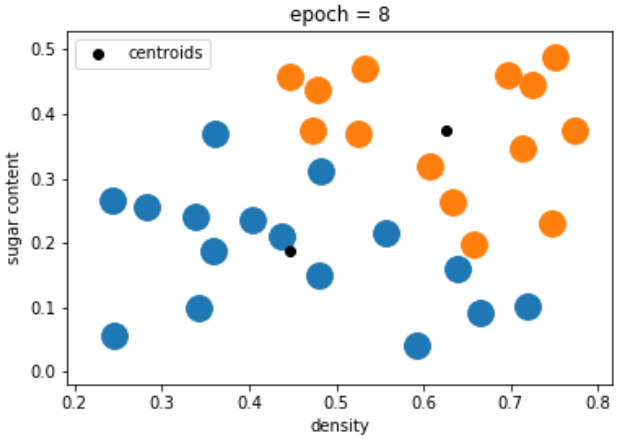


Figure 18: $epoch = 8$

2 Implementation

2.1 Introduction

In this section we show how we implemented the k-means algorithm. The whole algorithm implementation study is detailed in the end of this section. In this introduction we show the main algorithm in order to introduce the order of the study. As we discussed the objective of each functions in the first section, we will go right to the point. There is also another function plot_clusters which is used to visualise the epochs.

1. init_centroids
2. assign_cluster
3. update_centroids
4. cost_function

```
def kMeans(data,k,init_method,dist_method,max_epoch=10,thresh=1e-3,save=False,plot=False):

epoch       = 0
Centroids   = init_centroids(Data,k)
Clusters    = assign_cluster(Centroids,Data)
if plot == True:
    plot_clusters(Clusters,Centroids,epoch,save)


Cost = [ 10*thresh for j in range(max_epoch)]
while epoch<max_epoch:
    Centroids   = update_centroids(Centroids,Clusters)
    Clusters    = assign_cluster(Centroids[1],data)
    Cost[epoch] = cost_function(Centroids[1],Clusters)
    if plot == True:
        plot_clusters(Clusters,Centroids[1],epoch,save)
    epoch = epoch +1

    if epoch >=1:
        diff = Cost[epoch-1]-Cost[epoch-2]
        if diff == 0:
            break

return Cost,Centroids
```

2.2 Built-in functions

In order to study each function, we will have to use some built-in functions of python and the numpy library. Below we give the time and space complexity of each built-in functions. For most of them, the demonstration of the results are given in the appendices.

python - APPEND : to append an element at the end of a list

- **argument :** $1D$-list of $n \in \mathbb{N}$ elements
- **time complexity :** $T_{append} = \Theta(1)$

python - LEN : to get the number of elements of a list

- **argument :** $1D$-list of $n \in \mathbb{N}$ elements
- **time complexity :** $T_{len} = \Theta(1)$

numpy - ARGMIN : to get the minimum values along a given axis

- **argument :** $2D$-array of $(n, m) \in \mathbb{N}^2$ elements
- **time complexity :** $T_{argmin} = \Theta(n + m)$

numpy - ARRAY : to convert a list to an array

- **argument :** $2D$-list of $n \cdot m \in \mathbb{N}$ elements
- **time complexity :** $T_{array} = \Theta(log(n \cdot m))$

numpy - MEAN : to get the mean value of an array

- **argument :** $1D$-array of $n \in \mathbb{N}$ elements
- **time complexity :** $T_{mean} = \Theta(n)$

numpy - RANDOM.CHOICE : to draw m random integers in $[\![1, n]\!]$

- **argument :** integers $(n, m) \in \mathbb{N}^2$
- **time complexity :** $T_{choice} = \Theta(m)$

numpy - RESHAPE : to reshape a matrix with a number of columns[3]

- **argument :** array of $(n, m) \in \mathbb{N}^2$ elements. We fix the number of columns $n \in \mathbb{N}$
- **time complexity :** $T_{reshape} = \Theta(n)$

numpy - SHAPE : to get the shape of an array

- **argument :** array of $(n, m) \in \mathbb{N}^2$ elements
- **time complexity :** $T_{shape} = \Theta(1)$

numpy - TRANSPOSE : to transpose a matrix

- **argument :** array of $(n, m) \in \mathbb{N}^2$ elements.
- **time complexity :** $T_{transpose} = \Theta(n + m)$

numpy - ZEROS : to generate a matrix containing only zeros

- **argument :** $(n, m) \in \mathbb{N}^2$
- **time complexity :** $T_{zeros} = \Theta(n \cdot m)$

2.3 init_centroids

inputs :

- $data \in \mathbb{R}^{n \times p}$ the dataset samples
- k the number of clusters

outputs :

- $centroids$ the centroids coordinates

```
1  def init_centroids(data,k,init_method="random"):
2
3  n_samples = data.shape[0]
4
5  # other methods could be implemented, like the k-means++
6  if init_method == "random":
7
8      # we randomly choose indices
9      indices     = np.random.choice(n_samples,k)
10     # and we pick the initial centroids from the dataset
11     centroids   = data[indices,:]
12     return centroids
13
```

lign 3 : $T = \Theta(1)$
lign 6 : $T = \Theta(1)$
lign 9 : $T = \Theta(k)$
lign 11 : $T = \Theta(1)$
lign 12 : $T = \Theta(1)$

[3] We do not care, on purpose, about the number of rows

Thus :

$$T_{IC} = \Theta(k) \tag{21}$$

Which is what we have guessed in the previous section.

2.4 distance

inputs :

- $A \in \mathbb{R}^{p+1}$: the first point with p coordinates
- $B \in \mathbb{R}^{p+1}$: the second point with p coordinates
- $dist\_method$: the method to compute the distance

outputs :

- d : the distance between A and B

```
1  def distance(A,B,dist_method="euclidian"):
2
3      # formating
4      A   = np.array(A).reshape(-1,2)
5      B   = np.array(B).reshape(-1,2)
6
7      # to make formula more easy to read
8      xA = A[:,0]
9      yA = A[:,1]
10     xB = B[:,0]
11     yB = B[:,1]
12
13     # to store all the distances
14     d = np.zeros((1,A.shape[0]))
15
16     # as a reminder : d_manhattan = |xA-xB| + |yA-yB|
17     if dist_method == "manhattan":
18         d = np.abs(xA-xB)+np.abs(yA-yB)
19         return d
20
21     # as a reminder : d_euclidian = v[(xA-xB)² + (yA-yB)²]
22     elif dist_method == "euclidian":
23         d = np.sqrt((xA-xB)**2+(yA-yB)**2)
24         return d
25
```

lign 4,5 : $T = \Theta(1)$
lign 8,9,10,11 : $T = \Theta(1)$
lign 17 : $T = \Theta(1)$
lign 18 : $T = \Theta(p)$
lign 19 : $T = \Theta(1)$
lign 22 : $T = \Theta(1)$
lign 23 : $T = \Theta(p)$
lign 24 : $T = \Theta(1)$

Thus :

$$T_{dist} = \Theta(p) \tag{22}$$

However in the first section we announced $\Theta[n\_samples \cdot p \cdot k]$ in the first section. This is explained by the fact that here we only compute the distance between two points, whereas in the first section, we considered the computation of distance between each point ($n\_samples$) to each centroids (k). Eventually the complixities are matching.

2.5 assign_cluster

inputs :

- $centroids \in \mathbb{R}^{k \times p}$: the coordinates of the (k) centroids having (p) coordinates.
- $data \in \mathbb{R}^{n\_samples \times p}$: the dataset
- "dist_method"

outputs :

- $clusters \in \mathbb{R}^{n\_samples \times p}$

```
1  def assign_cluster(centroids,data,dist_method="euclidian"):
2
3      n_samples   = data.shape[0]
4      k           = centroids.shape[0]
5      clusters    = [[] for k in range(k)]
6      dist        = np.zeros((k,n_samples))
7
8      # for each centroid (l)
9      # we compute the distance of each dataset sample (j)
10     for l in range(k):
11         for j in range(n_samples):
12             dist[l,j]   = distance(centroids[l],data[j])
13
14     # we look for the closest dataset samples from each centroid
15     indices_of_minima   = np.argmin(dist,axis=0)
16
17     # then we assign the closest dataset samples to its cluster
18     for j in range(n_samples):
19         clusters[indices_of_minima[j]].append(data[j])
20
21     return clusters
```

lign 3,4 : $T = \Theta(1)$
lign 5 : $T = \Theta(k)$
lign 6 : $T = \Theta(k \cdot n\_samples)$
lign 10 : $T = \Theta(k)$
lign 11 : $T = \Theta(n\_samples))$
lign 12 : $T = \Theta(p)$
lign 15 : $T = \Theta(n\_samples + k)$
lign 18 : $T = \Theta(1)$
lign 19 : $T = \Theta(1)$
lign 21 : $T = \Theta(1)$

Thus :

$$T_{AC} = 3 \cdot \Theta(1) + \Theta(k) + \Theta(n\_samples + k) + \Theta(n\_samples \cdot k) + \Theta(n\_samples \cdot k \cdot p) \quad (23)$$
$$= \mathcal{O}(n\_samples \cdot k \cdot p) \quad (24)$$

Which is exactly what was announced in the first section.

2.6 cost_function

inputs :

- $centroids \in \mathbb{R}^{k \times p}$: the coordinates of the (k) centroids having (p) coordinates.
- $clusters \in \mathbb{R}^{n\_samples \times k \times p}$ the (k) clusters containing the $(n\_samples)$ having (p) coordinates

outputs :

- $cost$ the value of the cost function

```
1  def cost_function(centroids,clusters):
2
3      k       = len(clusters)
4      means   = [0 for l in range(k)]
5
6      for l in range(k):
7          means[l]=np.mean(distance(centroids[l],clusters[l]))
8      cost = np.mean(means)
9
10     return cost
```

lign 3 : $T = \Theta(1)$
lign 4 : $T = \Theta(k)$
lign 6 : $T = \Theta(1)$
lign 7 : $T = \Theta(m_{K_l}) \quad \forall l \in [\![1, k]\!]$
lign 8 : $T = \Theta(k)$
lign 10: $T = \Theta(1)$

Thus :

$$\begin{aligned} T_{CF} &= 2 \cdot \Theta(1) + 2 \cdot \Theta(k) + k \cdot \Theta(m_{K_l} \cdot p) && (25) \\ &= \mathcal{O}(n\_samples \cdot k \cdot p) && (26) \end{aligned}$$

Which is what we announced.

2.7 update_centroids

inputs :

- $old\_centroids \in \mathbb{R}^{k \times p}$
- $new\_centroids \in \mathbb{R}^{k \times p}$

outputs :

- $Centroids \in (\mathbb{R}^{k \times p})^{epoch}$: the concatenation of all the centroids positions during all the excecution

```
1      def update_centroids(old_centroids,clusters):
2
3          k               = len(clusters)
4          n_features      = len(clusters[0][0])
5          new_centroids   = np.zeros((k,n_features))
6
7          for l in range(k):
8              for j in range(n_features):
9                  new_centroids[l,j] = np.mean(np.array(clusters[l])
   .T[j])
10
11             # formating to return a matrix with all the
   centroids
12         # movements
13         Centroids       = [old_centroids,new_centroids]
14
15         return Centroids
```

lign 3 : $T = \Theta(1)$
lign 4 : $T = \Theta(1)$
lign 5 : $T = \Theta(k \cdot p)$
lign 7 : $T = \Theta(1)$
lign 8 : $T = \Theta(1)$
lign 9 : $T = 2 \cdot \Theta(m_{K_l} + p) + \Theta[log(m_{K_l} + p)]$
lign 13 : $T = \Theta(1)$
lign 15 : $T = \Theta(1)$

The lign 9 should be a bit more detailed. There are, in the same line four operations : one affectation ($\Theta[1]$), a averaging ($\Theta[m_{K_l} + p]$), a conversion into a numpy.ndarray ($\Theta[log(m_{K_l} + p)]$) and a matrix transposition ($\Theta[m_{K_l} + p]$). Thus we get :

$$\begin{aligned} T_{UC} &= 4 \cdot \Theta(1) + \Theta(k \cdot p) + k \cdot p \cdot \Big(2 \cdot \Theta[m_{K_l} + p] + \Theta[log(m_{K_l} + p)] \Big) && (27) \\ &= \mathcal{O}(k \cdot n\_samples \cdot p) && (28) \end{aligned}$$

Indeed :

$$\forall \kappa \in [\![1, k]\!] \quad n\_samples \geq m_{K_l} + p \geq log(m_{K_l} + p) \qquad (29)$$

2.8 kmeans

inputs :

- *data*
- *k* : the number of clusters
- *init_method* : centroids' initialisation method
- *dist_method* : distance computation method
- *max_epoch* : maximum number of epochs
- *thresh* : threshold to reach for the cost function.
- *save* : variable for saving or not the plots
- *plot* : variable for showing of not the plots

outputs :

- *cost* : the cost evolution in function of the epoch
- *centroids* : the centroids coordinates

```
1  def kMeans(data,k,init_method,dist_method,max_epoch=10,thresh=1e
       -3,save=False,plot=False):
2
3      epoch       = 0
4      Centroids   = init_centroids(Data,k)
5      Clusters    = assign_cluster(Centroids,Data)
6      if plot == True:
7          plot_clusters(Clusters,Centroids,epoch,save)
8
9      Cost = [ 10*thresh for j in range(max_epoch)]
10
11     while epoch<max_epoch:
12
13         Centroids   = update_centroids(Centroids,Clusters)
14         Clusters    = assign_cluster(Centroids[1],data)
15         Cost[epoch] = cost_function(Centroids[1],Clusters)
16         if plot == True:
17             plot_clusters(Clusters,Centroids[1],epoch,save)
18         epoch = epoch +1
19
20         if epoch >=1:
21             diff = Cost[epoch-1]-Cost[epoch-2]
22             if diff == 0:
23                 break
24
25     return Cost,Centroids
```

lign 3 : $T = \Theta(1)$
lign 4 : $T = \Theta(k)$
lign 5 : $T = \mathcal{O}(n\_samples \cdot k \cdot p)$
lign 6,7 : not required for the purpose
lign 9 : $T = \Theta(max\_epoch)$
lign 11 : $T = \Theta(1)$
lign 13 : $T = \mathcal{O}(n\_samples \cdot k \cdot p)$
lign 14: $T = \mathcal{O}(n\_samples \cdot k \cdot p)$
lign 15 : $T = \mathcal{O}(n\_samples \cdot k \cdot p)$
lign 16,17 : not required for the purpose
lign 18 : $T = \Theta(1)$
lign 20: $T = \Theta(1)$
lign 21 : $T = \Theta(1)$
lign 22 : $T = \Theta(1)$
lign 23 : $T = \Theta(1)$
lign 25 : $T = \Theta(1)$

Thus we get :

$$\begin{aligned} T_{kmeans} &= 8 \cdot \Theta(1) + \Theta(k) + \Theta(max\_epoch) + (max\_epoch + 1) \cdot \mathcal{O}(n\_samples \cdot k \cdot p) && (30) \\ &= \mathcal{O}(n\_samples \cdot k \cdot p \cdot max\_epoch) && (31) \end{aligned}$$

3 Improvements

3.1 Introduction

In this section we are going to give some ways to improve the algorithm. We will first discuss the choice of the metric, or in other words, the method to compute de distance. Then we will shed light on the importance of the initiilisation step.

3.2 Metric choice

We implemented an algorithm which gives the choice to the user about the method to compute the distance. We only considered adistance family metric called the Minkowski distance. Given two points (x, y) in an Hilbertian space[4] (E^2) with a scalar product[5] $\left(\langle x, y\rangle\right)$.

$$x = (x_1, \cdots, x_n) \tag{32}$$
$$y = (y_1, \cdots, y_n) \tag{33}$$

The Minkowski distance $(d_{M,p})$ of order $p \in \mathbb{N}$ is defined as :

$$d_{M,p} : \begin{pmatrix} E^2 & \to & \mathbb{R}^+ \\ (x,y) & \mapsto & \left(\sum_{k=1}^{n\_samples} |x_k - y_k|^p\right)^{1/p} \end{pmatrix}. \tag{34}$$

The cases $p > 1$ show higher time complexities as the multiplication of two n-digits number takes in the best case $\mathcal{O}(n \cdot log(n))$ [8]. Same for the square root :

$$T = \left[\underbrace{\Theta(n)}_{substraction} + \underbrace{\Theta(1)}_{absolute\ value} + \underbrace{\mathcal{O}(n \cdot log(n))}_{power}\right] \cdot \underbrace{n\_samples}_{sum} + \underbrace{\mathcal{O}(n \cdot log(n))}_{p-root} \tag{35}$$
$$= \mathcal{O}(n\_samples \cdot n \cdot log(n)) \tag{36}$$

Thus we see that for $p = 1$ the complexity is only $\mathcal{O}(n\_samples \cdot n)$ So this does not make an important difference. However, according to the statistical distribution of the coordinates, chosing one distance or another can add bias.

3.3 Initilisation step

In order to answer properly to this question, we might first introduce the cost function (J) which has to be minimize at each epoch (or more explicitly, each iteration).

We have (n) data samples we would like to group into (K) clusters. Let $\mathcal{C} = \{c_1, \cdots, c_K\}$ the partition of the (n) samples into the (K) clusters. The common cost function used for the K-means algorithm is Mean Square Error (MSE)[5] :

$$J = \sum_{\kappa=1}^{K} \left(\sum_{x_i \in c_j} \|x_i - \mu_\kappa\|^2 \right) \tag{37}$$

This new initilization methods ensures a minimum distance between each centroid. As seen in the figures 20,21, 22,23 show the different steps of this new algorithm, from [8].

[4] A complete vector space with an Hermitian scalar product
[5] positively defined, symetric and bilinear linear form

where $M = \{\mu_1, \cdots, \mu_K\}$ are the (K) centers of the clusters defined as :

$$\forall \kappa \in [\![1, K]\!] \quad \mu_\kappa = \frac{1}{card(c_\kappa)} \cdot \sum_{x_i \in c_\kappa} x_i \tag{38}$$

Minimising this function leads the centroid movement to move until its displacement is null.
However such cost function can show multiple local minima, see figure 19. Thus, when the initial the centroid position is located in a local minimum, it will be stuck in it and will give the wrong results.

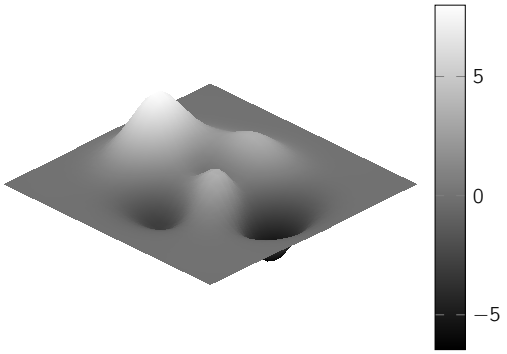


Figure 19: Example of non-convex surface with local minima

```python
# initialization algorithm
def initialize(data, k):

    # initialized the centroids for K-means++
    # inputs:
    #   data    - numpy array of data points having shape (200, 2)
    #   k       - number of clusters

    # initialize the centroids list and add a randomly selected data point to the list
    centroids = []
    centroids.append(data[np.random.randint(data.shape[0]), :])
    plot(data, np.array(centroids))

    # compute remaining k - 1 centroids
    for c_id in range(k - 1):

        # initialize a list to store distances of data points from nearest centroid
        dist = []
        for i in range(data.shape[0]):
            point   = data[i, :]
            d       = sys.maxsize

            # compute distance of 'point' from each of the previously selected centroid
            # store the minimum distance
            for j in range(len(centroids)):
                temp_dist   = distance(point, centroids[j])
                d           = min(d, temp_dist)
            dist.append(d)

        # select data point with maximum distance as our next centroid
        dist            = np.array(dist)
        next_centroid   = data[np.argmax(dist), :]
        centroids.append(next_centroid)
        dist            = []
        plot(data, np.array(centroids))
    return centroids

# call the initialize function to get the centroids
centroids = initialize(data, k = 4)
```

3.4 Conclusion

Here we mentionned two ways to improve the *k*-means. The first one relies on the method used to compute the distance. We mentionned only the Minkowski distances. But according to the clustering task (like text clustering), other methods can be used like the cosine distance or the value difference metric. The second deals with the initialisation step can avoid long convergence times or being stuck in local minima. This algorithm with such initialisation is called k-Means ++.

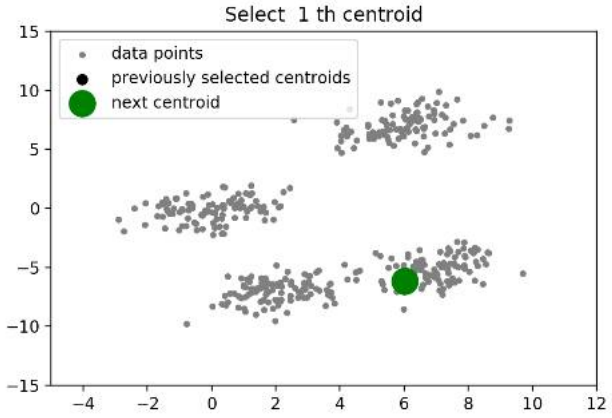


Figure 20: new initialisation method : step 1

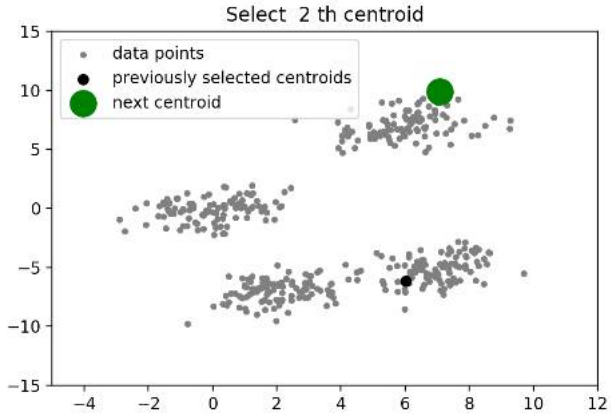


Figure 21: new initialisation method : step 2

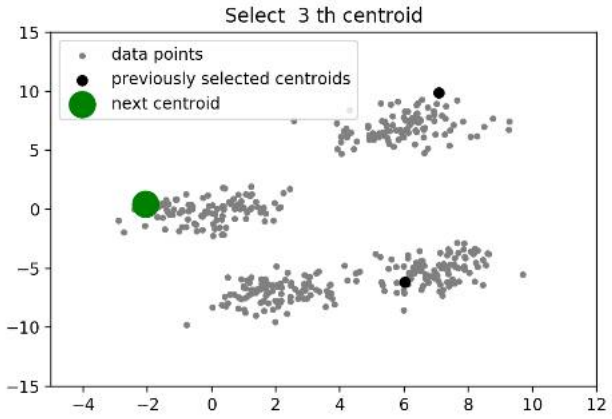


Figure 22: new initialisation method : step 3

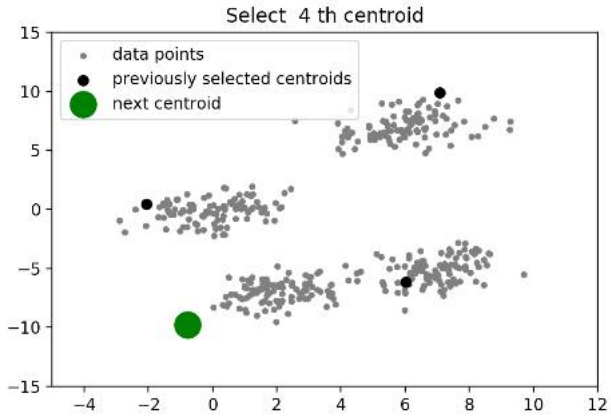


Figure 23: new initialisation method : step 4

Appendix

In this appendix we show the code produced to test the algorithm complexity is section 2.2 Built-in functions, and the plots we got.

```
# %%
import numpy as np
import time
import matplotlib.pyplot as plt
import pandas as pd

# %%

def write_in_file(file_name,data):

    data.to_csv(file_name, sep=',')

def plot(X,Y,colors,x_label,y_label,title,labels,legend_loc,line_styles,lin_or_log='lin',save_fig="no",
  resolution=100,fig_size=(13,8)):

    plt.figure(figsize=tuple(fig_size))

    n = X.shape[0]
    if lin_or_log == "lin":
        for k in range(n):
            plt.plot(X[k,:],Y[k,:],
                    color=colors[k],
                    label = labels[k],
                    linestyle = line_styles[k])

    elif lin_or_log =="semilogx":
        for k in range(n):
            plt.semilogx(X[k,:],Y[k,:],
                    color=colors[k],
                    label = labels[k],
                    linestyle = line_styles[k])

    elif lin_or_log =="semilogy":
        for k in range(n):
            plt.semilogy(X[k,:],Y[k,:],
                    color=colors[k],
                    label = labels[k],
                    linestyle = line_styles[k])
    else:
        for k in range(n):
            plt.loglog(X[k,:],Y[k,:],
                    color=colors[k],
                    label = labels[k],
                    linestyle = line_styles[k])

    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.xlabel(x_label,fontsize=20,weight = "bold")
    plt.ylabel(y_label,fontsize=20,weight = "bold")
    plt.suptitle(title[0], fontsize=30,weight = "bold")
    plt.title(title[1], fontsize=25,weight = "bold")
    plt.legend(loc=legend_loc)
    if save_fig == "yes":
        plt.savefig(title[0]+'_'+title[1]+".png",dpi=resolution)
 #       plt.savefig(title+".png",dpi=resolution)
    plt.show()


# %%
# number of samples for plotting
n           = int(1e3)

# number of realisations
n_tests     = int(1e0)

```

```
# different sizes of zeros array
n_samples   = np.round(np.logspace(0,5,n),2).astype(int)

# storing the running times
times       = np.zeros((n_tests,n))

# loop to get the running times
# for each realisations
for k in range(n_tests):

    # for each size of np.zeros array we want
    for j in range(n):
        print("j=",j)

        A = np.zeros((int(1e6),2))
        tmp = []
        # time reference
        start       = time.time()

        # creation of the zeros array
        tmp         = A[1:n_samples[j],:]

        # time after the array creation
        end         = time.time()

        # deletion of the variable
        del A

        # calculate the running time
        times[k][j] = end-start

times_copy = times.transpose()
mean_times = np.zeros((1,n))


for k in range(n):
    mean_times[0][k] = np.mean(times_copy[k])

mean_times1 = mean_times[0]*1e6

#%%
plot(
     X              = np.array([n_samples,n_samples]),
     Y              = np.array([mean_times1,mean_times2]),
      colors        = ["black",'blue'],
      x_label       = "n_samples",
      y_label       = "time [$\mu$s]",
      title         = np.array(["time complexity___py.subarray - log scale",'number of samples impact']),
      labels        = ["1 realisation","100 realisations"],
      legend_loc    = "best",
      line_styles   = ['-','-.'],
      lin_or_log    = "semilogx",
      save_fig      = "yes",
      resolution    = 500
      )


#%%
data = np.array([n_samples,mean_times1,mean_times2]).transpose()
data = pd.DataFrame(data)
columns = "n_samples,one_rea,hundred_rea".split(",")
data.columns = columns

write_in_file("complexity_python_subarray.csv",data)
```

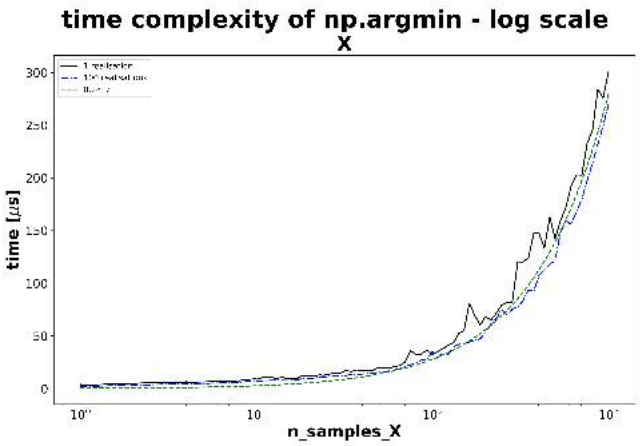


Figure 24: Time complexity of ARGMIN along X axis

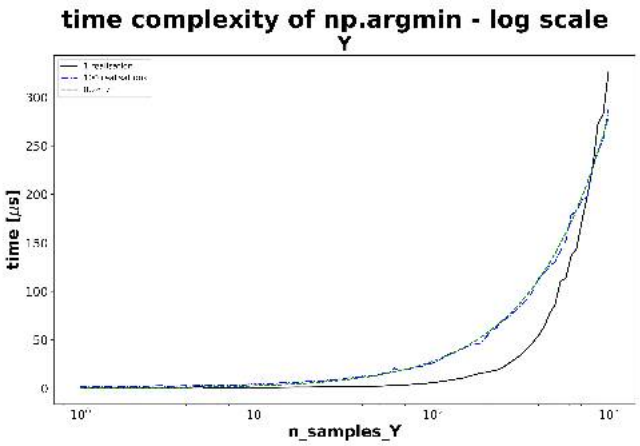


Figure 25: Time complexity of ARGMIN along Y axis

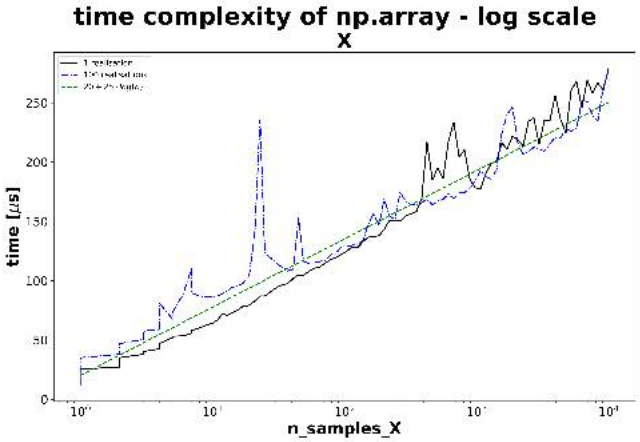


Figure 26: Time complexity of ARRAY along X axis

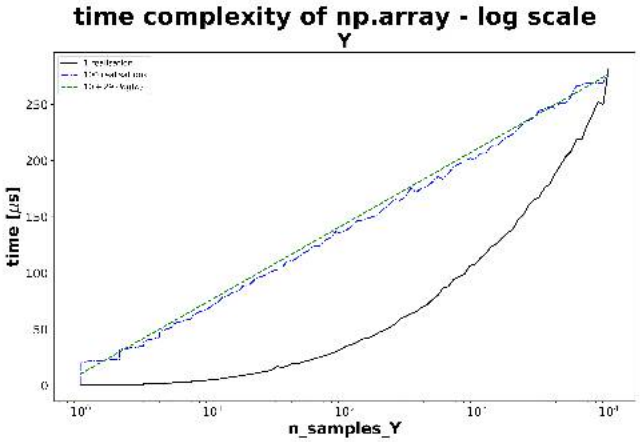


Figure 27: Time complexity of ARRAY along Y axis

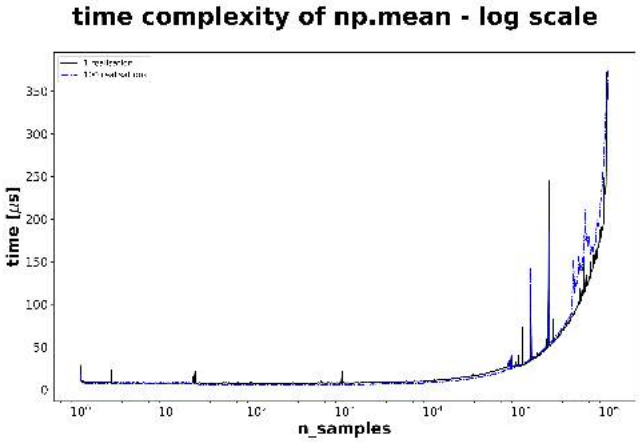


Figure 28: Time complexity of MEAN

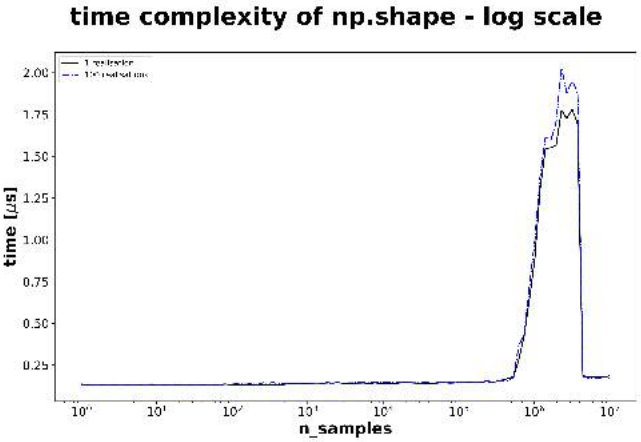


Figure 29: Time complexity of SHAPE

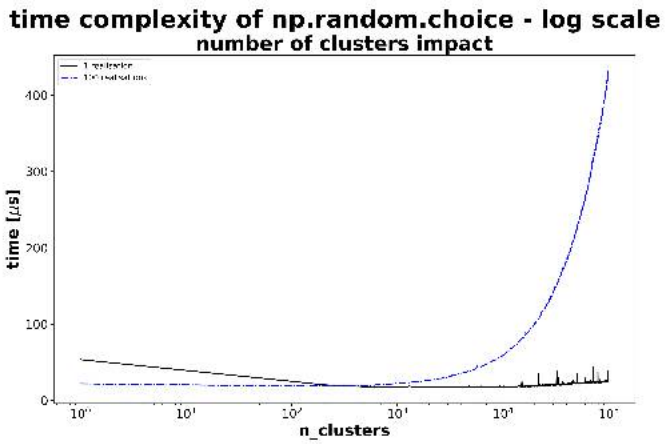


Figure 30: Time complexity of RANDOM.CHOICE according to the number of clusters we want

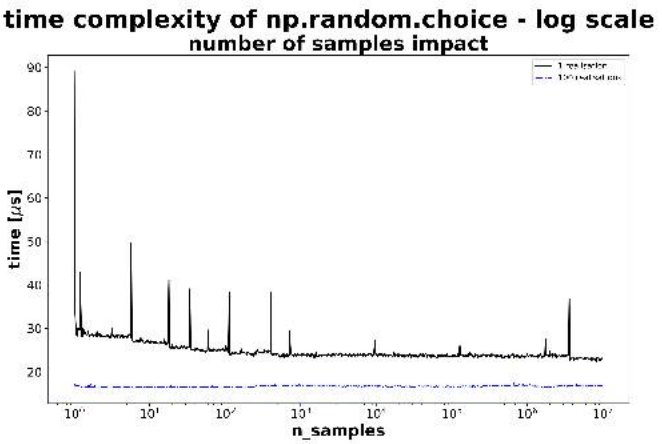


Figure 31: Time complexity of RANDOM.CHOICE according to the total number of samples

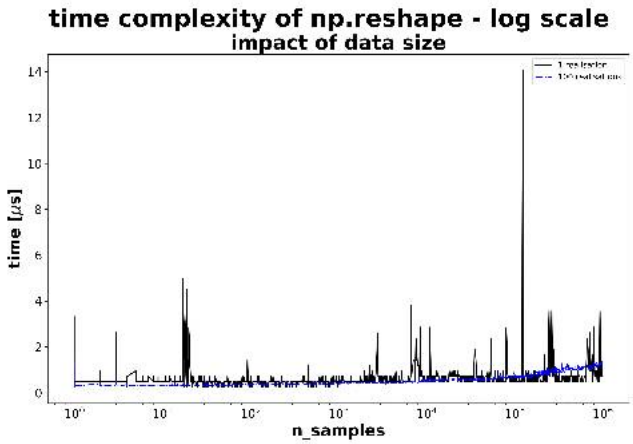


Figure 32: Time complexity of RESHAPE to the data size

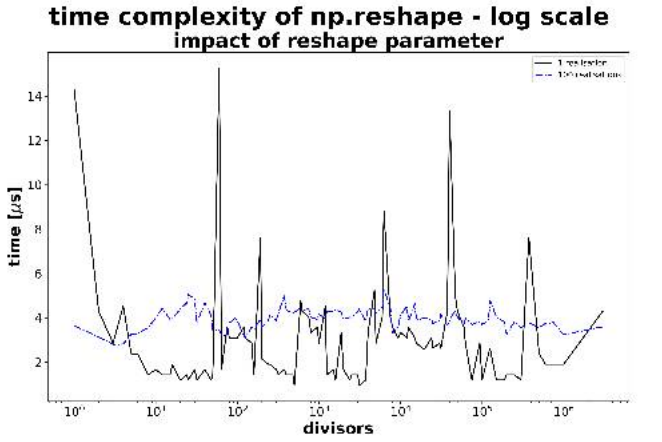


Figure 33: Time complexity of Reshape according to the number of columns we want

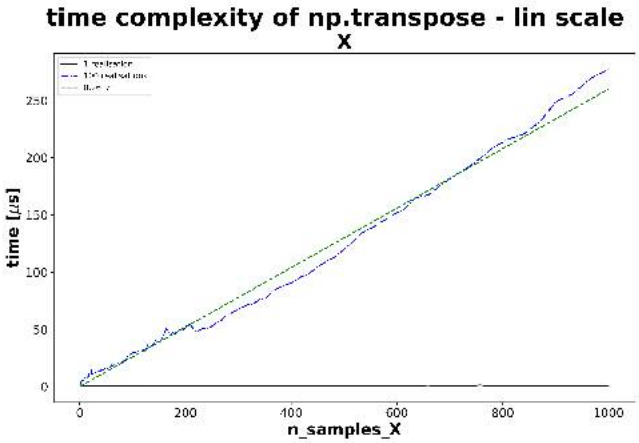


Figure 34: Time complexity of TRANSPOSE along the X axis

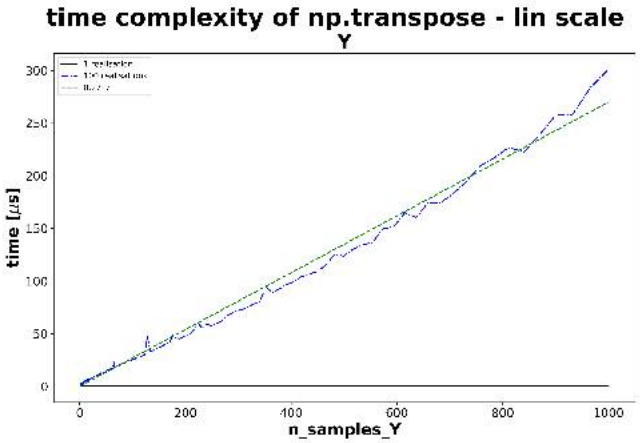


Figure 35: Time complexity of TRANSPOSE along the Y axis

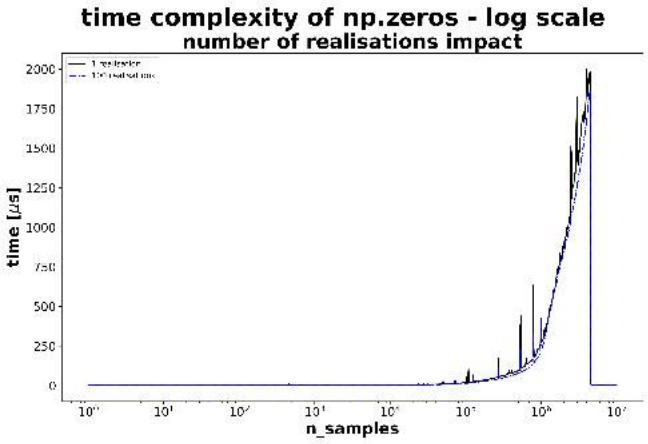


Figure 36: Time complexity of ZEROS

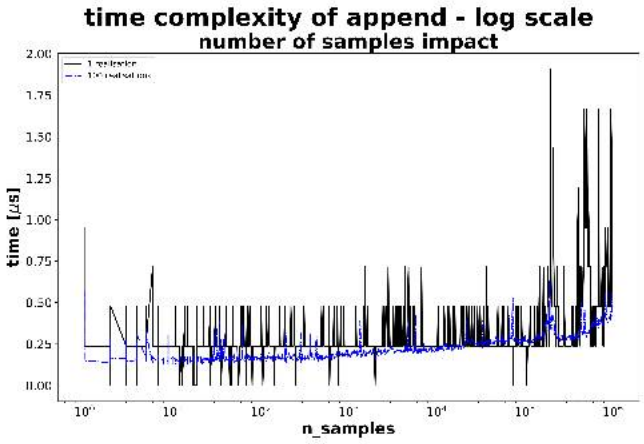


Figure 37: Time complexity of APPEND

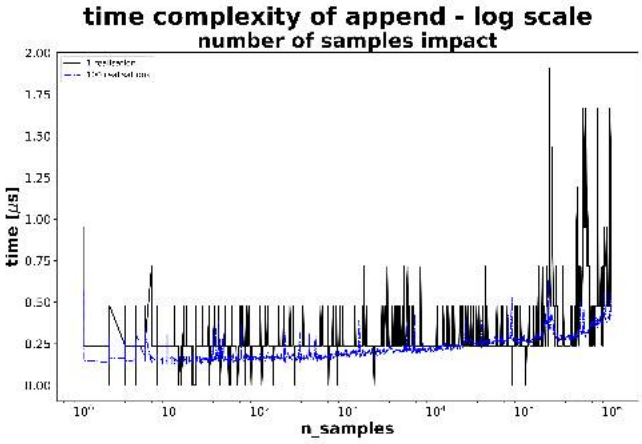


Figure 38: Time complexity of LEN

Bibliography

[1] **Jean-Luc VERLEY**, *Métriques Espaces*, (French), Encyclopædia Universalis [online], https://www.universalis.fr/encyclopedie/espaces-metriques/, [Accessed : January 9, 2022]

[2] **Archana SINGH, Avantika YADAV, Ajay RANA**, *K-means with Three different Distance Metrics*, Int. J. Comput. Appl., April 2013, Vol 67, No 10, pages 13-17

[3] **Dibya JYOTI BORA, Dr. Anil KUMAR GUPTA**, *Effect of Different Distance Measures on the Performance of K-Means Algorithm: An Experimental Study in Matlab*, Int. J. Comput. SC., 2014, Vol 5, No 2, pages 2501-2506

[4] **Zhi-Hua ZHOU**, *Machine Learning*, Springer, 2016, ISBN-13 : 978-981-15-1966-6, doi : 10.1007/978-981-15-1967-3

[5] **Thomas BONALD**, *Clustering by k-Means*, Telecom ParisTech, 2019, https://perso.telecom-paristech.fr/bonald/documents/kmeans.pdf, [Accessed : January 9, 2022]

[6] **Savya KHOSLA**, *ML | K-means++ Algorithm*, GeeksforGeeks, July 2021, https://www.geeksforgeeks.org/ml-k-means-algorithm/, [Accessed : January 9, 2022]

[7] **Ali DEHGHANI**, *The K-Means Clustering Algorithm in Java*, Baeldung, January 2021, https://www.baeldung.com/java-k-means-clustering-algorithm, [Accessed : January 9, 2022]

[8] **WIKIPEDIA**, *Computational complexity of mathematical operations*, October 2021, https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations, [Accessed : January 9, 2022]