INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport – Big Data

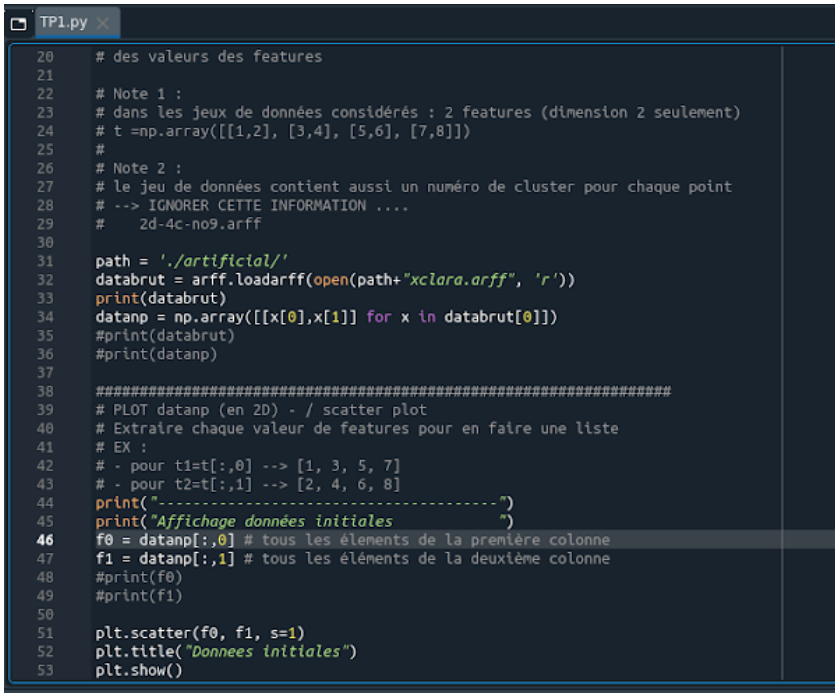
Chadi Amour, Redouane Amour, Arnaud Bouisson

5ISS, 2021

Github :

I – Points forts et faibles identifiés pour les méthodes de clustering étudiées

Le premier travail consiste à comprendre l'utilisation des libraires numpy, scipy, et matplotlib afin d'afficher graphiquement un jeu de données disponible au format arff.



Librairie scipy.io : on télécharge le jeu de données (dataset) initial avec la fonction « arff.loadarff » en indiquant en argument le chemin du fichier arff. Cette fonction retourne un type de structure, qui est stockée dans « databrut ». On repère les coordonnées (x,y) et un attribut correspondant à la classe du cluster selon une méthode utilisée par défaut.

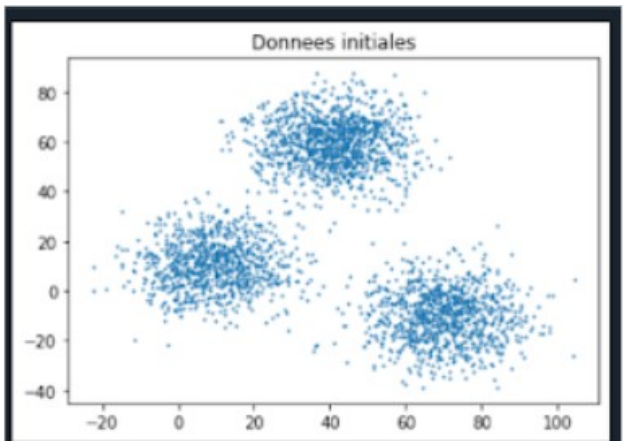
Librairie numpy : on extrait ensuite les coordonnées (x,y) en créant un tableau « datanp » avec la fonction « np.array » permettant de parcourir le tableau avec l'instruction « for x in databrut[0] ».

Librairie matplotlib : création de deux vecteurs f0 et f1 correspondant respectivement à la colonne X et à la colonne Y de « datanp ». Ces deux vecteurs sont ensuite passés en paramètre à la fonction « plt.scatter » afin d'afficher un nuage de points, fonction qui contient un troisième argument « s » permettant de dimensionner la taille des points.

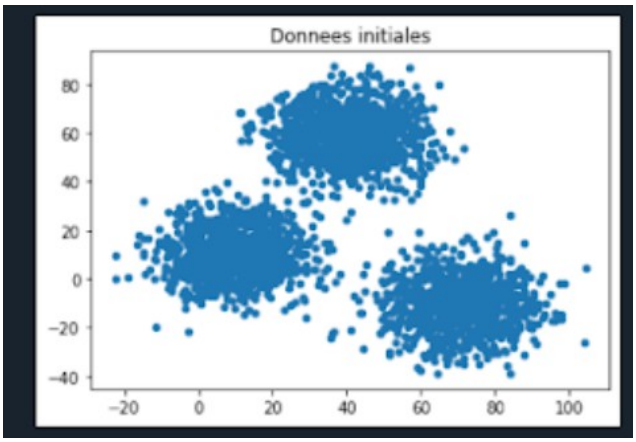
On étudie un jeu de données à 3 dimensions, donc on modifie notre code en conséquence :

```
path = './artificial/'
databrut = arff.loadarff(open(path+"xclara.arff", 'r'))
datanp = np.array([[x[0],x[1],x[2]] for x in databrut[0]])
#print(databrut)
#print(datanp)

#########################################################################
# PLOT datanp (en 2D) - / scatter plot
# Extraire chaque valeur de features pour en faire une liste
# EX :
# - pour t1=t[:,0] --> [1, 3, 5, 7]
# - pour t2=t[:,1] --> [2, 4, 6, 8]
print("----------------------------------------")
print("Affichage données initiales             ")
f0 = datanp[:,0] # tous les élements de la première colonne
f1 = datanp[:,1] # tous les éléments de la deuxième colonne
f2 = datanp[:,2] # tous les éléments de la troisième colonne
#print(f0)
#print(f1)
#print(f2)

plt.scatter(f0, f1, f2, s=1)
plt.title("Donnees initiales")
plt.show()
```

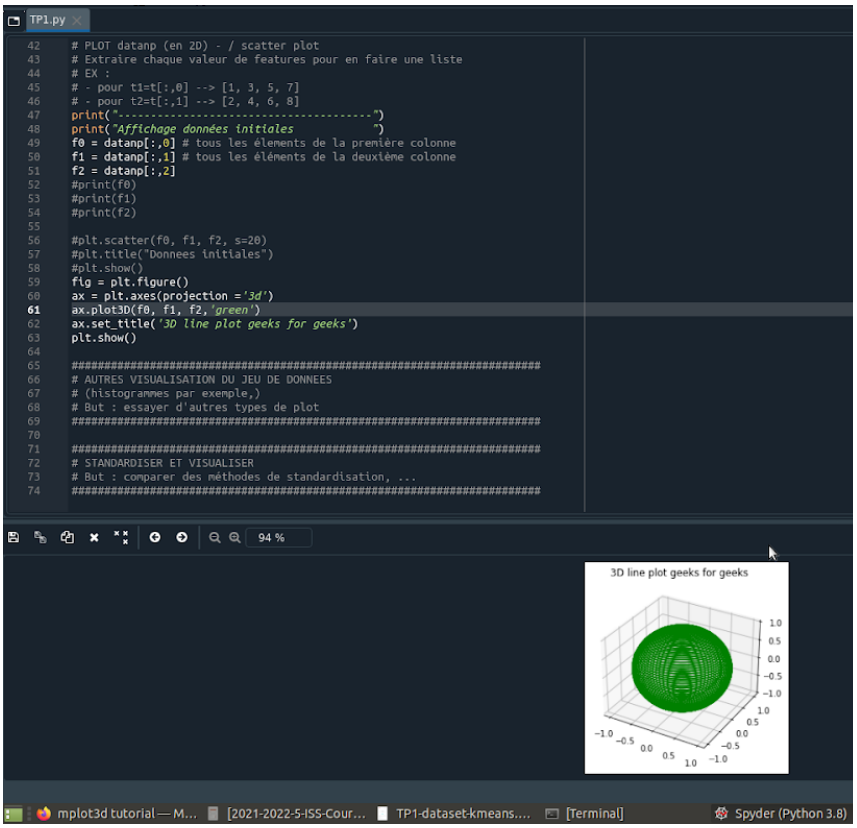
Cas s = 1 :



Cas s = 20 :



Nous avons ensuite essayé de modifier les paramètres initiaux en affichant cette fois-ci des points à trois dimensions en utilisant la fonction « plot3D » dans « matplotlib » :

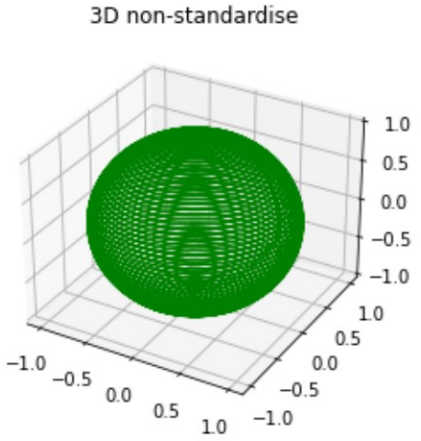


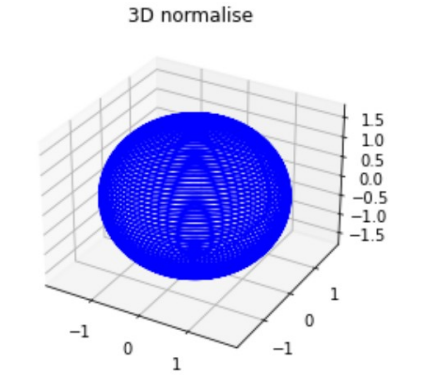
L'objectif était ensuite d'appliquer une méthode de prétraitement sur le jeu de données afin de standardiser les données. Nous avons notamment comparé le jeu de données « golf ball » non standardisé et standardisé en appliquant plusieurs méthodes de pré-traitement, à savoir la normalisation, le minimum et maximum, et la valeur absolue max.

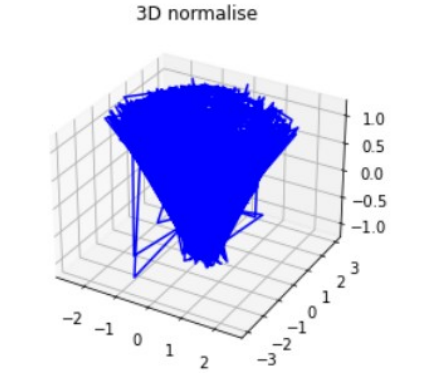
```
datanp = np.array([[x[0],x[1],x[2]] for x in databrut[0]])
#print(databrut)
#print(datanp)

scaler = preprocessing.StandardScaler().fit(datanp) # NORMALISATION PRE-PROC.
Data_set_normalise = scaler.transform(datanp)

min_max_scaler = preprocessing.MinMaxScaler() #MIN-MAX PRE-PROC.
Data_set_min_max= min_max_scaler.fit_transform(datanp)

max_abs_scaler = preprocessing.MaxAbsScaler() #MAX ABS PRE-PROC.
Data_set_max_abs = max_abs_scaler.fit_transform(datanp)
```

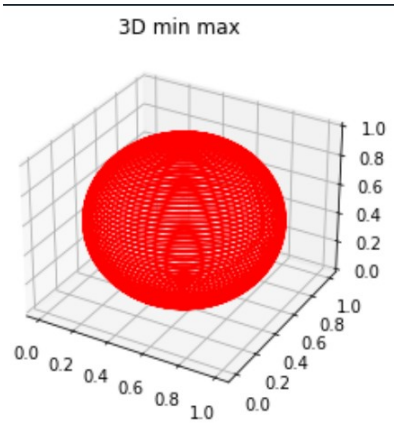
```
print("------------------------------------")
print("Affichage données initiales          ")
f0 = datanp[:,0] # tous les élements de la première colonne
f1 = datanp[:,1] # tous les éléments de la deuxième colonne
f2 = datanp[:,2] # tous les éléments de la troisième colonne

g0 = Data_set_normalise[:,0] # tous les élements de la première colonne
g1 = Data_set_normalise[:,1] # tous les éléments de la deuxième colonne
g2 = Data_set_normalise[:,2] # tous les éléments de la troisième colonne

k0 = Data_set_min_max[:,0] # tous les élements de la première colonne
k1 = Data_set_min_max[:,1] # tous les éléments de la deuxième colonne
k2 = Data_set_min_max[:,2] # tous les éléments de ma troisième colonne

I0 = Data_set_max_abs[:,0] # tous les élements de la première colonne
I1 = Data_set_max_abs[:,1] # tous les éléments de la deuxième colonne
I2 = Data_set_max_abs[:,2] # tous les éléments de ma troisième colonne
```

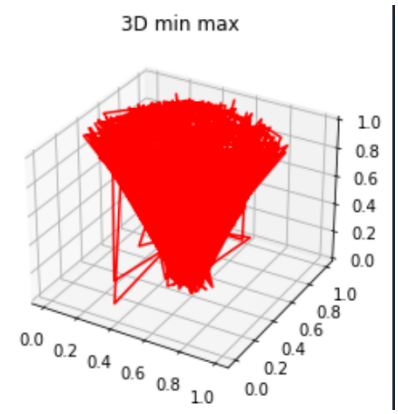
On affiche la comparaison graphique des jeux de données standardisés avec le jeu de données non standardisé pour « golf ball », en utilisant différents méthodes de pré-traitement comme indiqué avant.

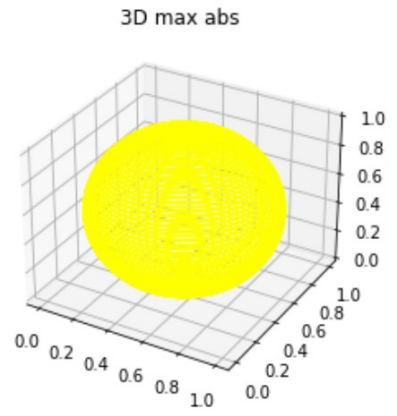


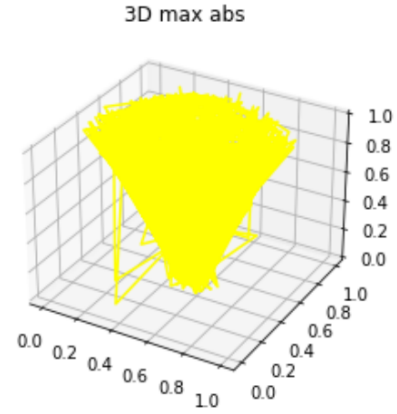




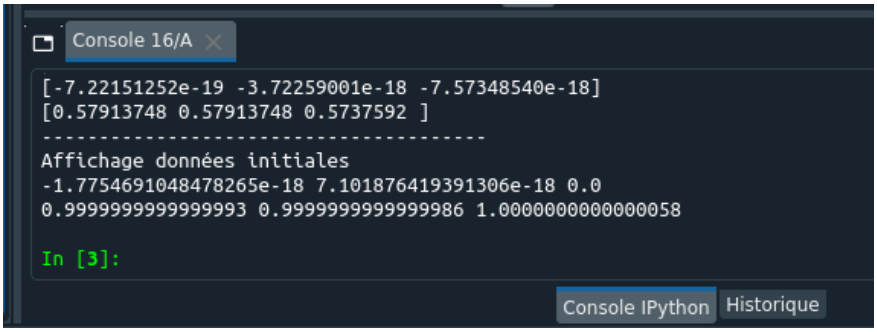

3D min max
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0
0.0 0.2 0.4 0.6 0.8 1.0


3D min max
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0
0.0 0.2 0.4 0.6 0.8 1.0


3D max abs
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0
0.0 0.2 0.4 0.6 0.8 1.0


3D max abs
1.0
0.8
0.6
0.4
0.2
0.0
0.0 0.2 0.4 0.6 0.8 1.0
0.0 0.2 0.4 0.6 0.8 1.0

Nous observons une moindre dispersion et des points centrés en 0 pour la méthode normalisée. Nous avons aussi comparé numériquement la moyenne et la variance des jeux de données pour les trois axes x, y, et z, après normalisation.

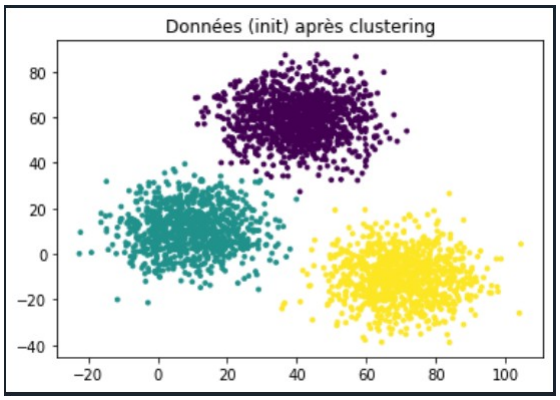


Nous allons maintenant utiliser la méthode de clustering des k-means avec cet algorithme :

```
# Run clustering method for a given number of clusters
print("-----------------------------------------------------")
print("Appel KMeans pour une valeur de k fixée (données init)")
tps1 = time.time()
k=3
model_km = cluster.KMeans(n_clusters=k, init='k-means++')
model_km.fit(datanp)
tps2 = time.time()
labels_km = model_km.labels_
# Nb iteration of this method
iteration = model_km.n_iter_

# Résultat du clustering
plt.scatter(f0, f1, c=labels_km, s=8)
plt.title("Données (init) après clustering")
plt.show()
print("nb clusters =",k,", nb iter =",iteration, ", runtime = ", round((tps2 - tps1)*1000,2),"ms")
#print("labels", labels_km)
# Some evaluation metrics
# inertie = wcss : within cluster sum of squares
inert = model_km.inertia_
silh = metrics.silhouette_score(datanp, model_km.labels_, metric='euclidean')
print("Inertie : ", inert)
print("Coefficient de silhouette : ", silh)
```

Plusieurs paramètres sont variables dans cet algorithme, à savoir le nombre de clusters noté k, les itérations, et les métriques. Pour évaluer la pertinence du nombre de clusters choisis, on utilise le coefficient de silhouette compris entre 0 et 1, en fixant la métrique. Dans le jeu de données initial « xclara.arff », il y a trois clusters. On a donc évalué ce coefficient pour plusieurs valeurs de k en utilisant la métrique euclidienne.

Cas pour k = 3 et k = 10, métrique = euclidean :




```
Affichage données initiales
-----------------------------------------------------
Appel KMeans pour une valeur de k fixée (données init)
nb clusters = 3 , nb iter = 2 , runtime =  45.89 ms
Inertie :  611605.8806933893
Coefficient de silhouette :  0.6945587736089913

In [5]: runfile('C:/Users/arnau/Desktop/BigData/tp1-read
BigData')
----------------------------------------
Affichage données initiales
-----------------------------------------------------
Appel KMeans pour une valeur de k fixée (données init)
nb clusters = 10 , nb iter = 31 , runtime =  221.95 ms
Inertie :  251493.44755557907
Coefficient de silhouette :  0.33031124747965684
```

Comme attendu, le coefficient de silhouette est plus élevé pour k = 3, donc k = 3 semble adéquat.

Cas k = 3 et k = 10, métrique = cosine :

```
Affichage données initiales
-----------------------------------------------------
Appel KMeans pour une valeur de k fixée (données init)
nb clusters = 3 , nb iter = 3 , runtime =  43.82 ms
Inertie :  611605.8806933893
Coefficient de silhouette :  0.4406851258954439

In [7]: runfile('C:/Users/arnau/Desktop/BigData/tp1-rea
BigData')
-------------------------------------
Affichage données initiales
-----------------------------------------------------
Appel KMeans pour une valeur de k fixée (données init)
nb clusters = 10 , nb iter = 16 , runtime =  223.4 ms
Inertie :  250905.1683220233
Coefficient de silhouette :  -0.05886939732280242
```

Les valeurs du coefficient de silhouette sont différentes mais k = 3 reste toujours plus pertinent. On constate même une valeur pour ce cas k = 10.

La méthode des k-means a l'inconvénient de reposer sur un algorithme assez long puisqu'il repose dans un premier temps sur un nombre d'itérations parfois grand et dans un second temps parce qu'il faut trouver le bon nombre de clusters initial k tel que le coefficient de silhouette soit le plus optimal possible. La métrique choisie influe également sur les résultats obtenus.

Nous allons maintenant chercher à déterminer automatiquement le bon nombre de clusters dans lesquels les données peuvent être regroupées. La méthode du coude est l'une des méthodes les plus populaires pour déterminer cette valeur optimale de k. Cette méthode utilise les valeurs de distorsion et d'inertie pour comparer différentes valeurs de k.

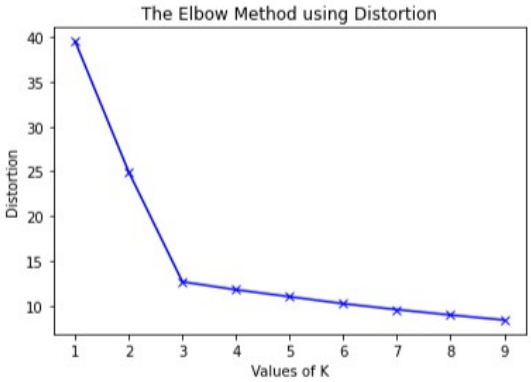
Distorsion : elle est calculée comme la moyenne des distances au carré des centres de cluster des clusters respectifs. En règle générale, la métrique de distance euclidienne est utilisée.

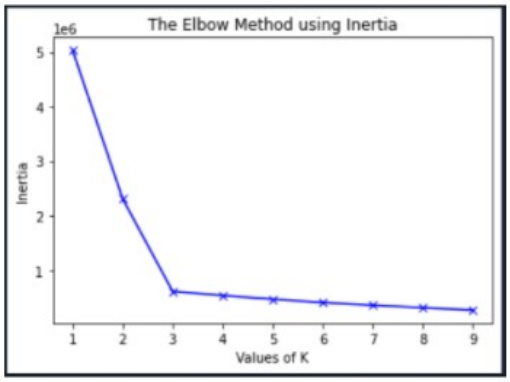
Inertie : somme des distances au carré des échantillons par rapport à leur centre de cluster le plus proche.

Voici une procédure d'évaluation de la méthode k-means :

```
##################################################################
# Run clustering method for a given number of clusters
distortions = []
inertias = []
mapping1 = {}
mapping2 = {}
K = range(1,10)

for k in K:

    kmeanModel = KMeans(n_clusters=k).fit(datanp)
    kmeanModel.fit(datanp)

    distortions.append(sum(np.min(cdist(datanp, kmeanModel.cluster_centers_,
                      'euclidean'),axis=1)) / datanp.shape[0])
    inertias.append(kmeanModel.inertia_)

    mapping1[k] = sum(np.min(cdist(datanp, kmeanModel.cluster_centers_,
                 'euclidean'),axis=1)) / datanp.shape[0]
    mapping2[k] = kmeanModel.inertia_
```

```
for key,val in mapping1.items():
    print(str(key)+' : '+str(val))

plt.plot(K, distortions, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Distortion')
plt.title('The Elbow Method using Distortion')
plt.show()

for key,val in mapping2.items():
    print(str(key)+' : '+str(val))

plt.plot(K, inertias, 'bx-')
plt.xlabel('Values of K')
plt.ylabel('Inertia')
plt.title('The Elbow Method using Inertia')
plt.show()
```





Pour déterminer le nombre optimal de clusters, il faut sélectionner la valeur de k au « coude », c'est à dire le point après lequel après la distorsion / inertie commence à diminuer de façon linéaire. Ainsi, avec la méthode du coude, nous parvenons bien à déterminer le bon nombre de clusters avec la cassure du coude. Nous concluons que le nombre optimal de clusters pour ce jeu de données est de 3.

Nous avons aussi relevé le temps d'exécution pour cette méthode qui est de 0,16 (temps assez long pour une méthode de calcul) :

```
for k in K:
    tps1 = time.time()
    kmeanModel = KMeans(n_clusters=k).fit(datanp)
    kmeanModel.fit(datanp)
    tps2 = time.time()
    distortions.append(sum(np.min(cdist(datanp, kmeanModel.cluster_centers_,
                      'euclidean'),axis=1)) / datanp.shape[0])
    inertias.append(kmeanModel.inertia_)
    mapping1[k] = sum(np.min(cdist(datanp, kmeanModel.cluster_centers_,
                 'euclidean'),axis=1)) / datanp.shape[0]
    mapping2[k] = kmeanModel.inertia_
```

```
temps d'execution kmeans  0.1619710922241211  secondes

In [19]:
```

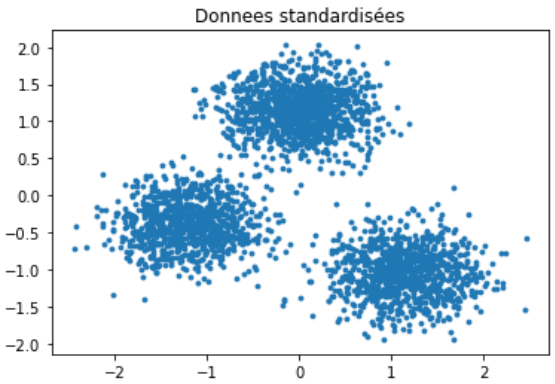
Cette technique des k-means est une méthode précise de clustering comme avantage. Le soucis est qu'elle est fastidieuse, longue, et dépendante de plusieurs paramètres initiaux (métrique, k, dataset, etc).

Cette méthode repose sur la minimisation de la somme des distances euclidiennes au carré entre chaque objet (ou sujet, ou point) et le centroïde (le point central) de son cluster. C'est une approche de clustering non hiérarchique (à l'intérieur d'un cluster, les objets ne sont pas ordonnés en fonction de leur ressemblance).

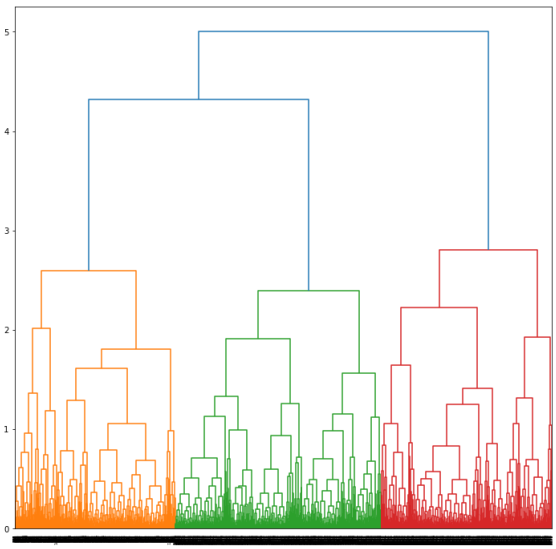
Nous allons maintenant traiter de la méthode de clustering agglomératif. Elle consiste à réunir de proche en proche les clusters les plus adjacents et de réaliser un dendrogramme permettant d'évaluer proportionnellement la distance inter-clusters. Les nœuds du dendrogramme correspondent aux agglomérations de cluster. Cette agglomération se fait selon un calcul de similarité appelé « linkage » dans les paramètres de la fonction, s'appuyant sur la distance séparant les clusters, pouvant être prise comme le minimum (single), le maximum (complete), la valeur entre centres (ward), et la moyenne (average).

```
print("Dendrogramme 'complete' données standardisées")

distance = shc.linkage(data_scaled, 'complete')


plt.figure(figsize=(12, 12))
shc.dendrogram(distance,
            orientation='top',
            distance_sort='descending',
            show_leaf_counts=False)
plt.show()

# Run clustering method for a given number of clusters
print("------------------------------------------------------------")
print("Appel Aglo Clustering 'complete' pour une valeur de k fixée")
tps3 = time.time()
k=3
model_scaled = cluster.AgglomerativeClustering(n_clusters=k, affinity='euclidean', linkage='complete')
model_scaled.fit(data_scaled)
#cluster.fit_predict(X)
```

Comme pour k-means, on évaluera la pertinence de cette méthode (en particulier le nombre de clusters choisis, les métriques, et le linkage) avec le coefficient de silhouette. Le jeu de données choisi est encore le « xclara.arff » :



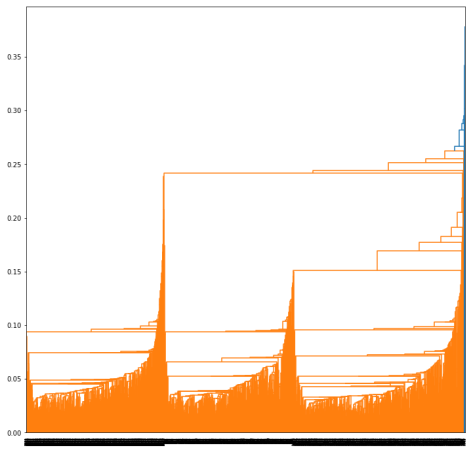
Cas k = 3, affinity = euclidean, linkage = complete :



```
Affichage données standardisées
----------------------------------------
Dendrogramme 'complete' données standardisées
------------------------------------------------------------
Appel Aglo Clustering 'complete' pour une valeur de k fixée
Coefficient de silhouette :  0.6908368385665083
nb clusters = 3 , runtime =  75955.3 ms
```

On a bien 3 clusters sur le dendrogramme avec un coefficient de silhouette situé à 0,69. On constate que le runtime est très long, ce qui paraît logique étant donné la construction du dendrogramme.

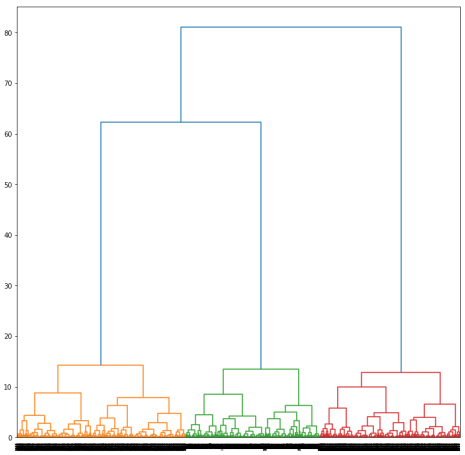
Cas k = 3, affinity = euclidean, linkage = simple :



```
Affichage données standardisées
----------------------------------------
Dendrogramme 'complete' données standardisées
------------------------------------------------------------
Appel Aglo Clustering 'complete' pour une valeur de k fixée
Coefficient de silhouette :  0.046301599230154734
nb clusters = 3 , runtime =  75384.03 ms
```

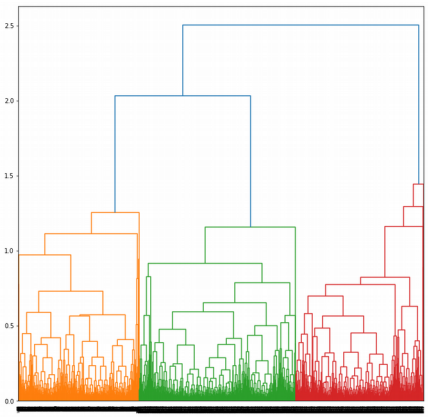
On remarque ici un problème puisque le coefficient de silhouette est proche de 0, ce qui explique la mauvaise construction du dendrogramme. Ceci peut venir du fait qu'on a utilisé la métrique euclidienne qui ne s'applique pas bien à cette méthode.

Cas k = 3, affinity = euclidean, linkage = ward :



```
----------------------------------------
Affichage données standardisées
----------------------------------------
Dendrogramme 'complete' données standardisées
------------------------------------------------------------
Appel Aglo Clustering 'complete' pour une valeur de k fixée
Coefficient de silhouette :  0.6908368385665083
nb clusters = 3 , runtime =  76854.32 ms
```

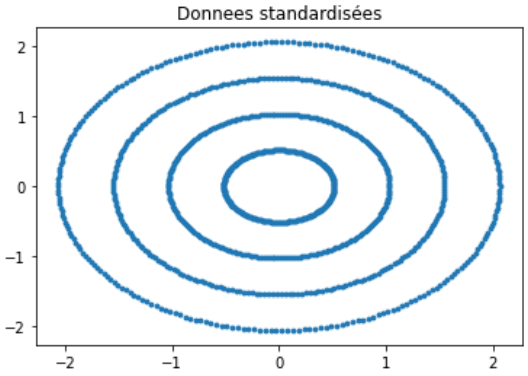
Cas k = 3, affinity = euclidean, linkage = average :



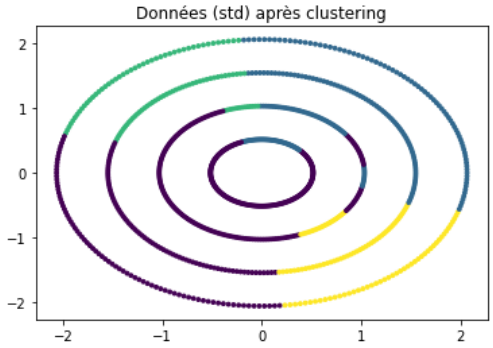
```
----------------------------------------
Affichage données standardisées
----------------------------------------
Dendrogramme 'complete' données standardisées
------------------------------------------------------------
Appel Aglo Clustering 'complete' pour une valeur de k fixée
Coefficient de silhouette :  0.6908368385665083
nb clusters = 3 , runtime =  76007.64 ms
```

Pour les 4 linkage, on constate des temps d'exécution assez importants, beaucoup plus longs que pour la méthode k-means, et des coefficients de silhouette proches de ce qu'on avait déjà observé avec k-means.

Autre test avec le jeu de données « dartboard1.arff ».

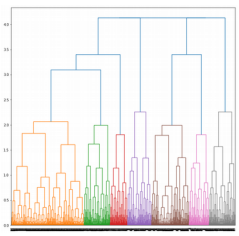


Cas k = 4, affinity = euclidean, linkage = complete :



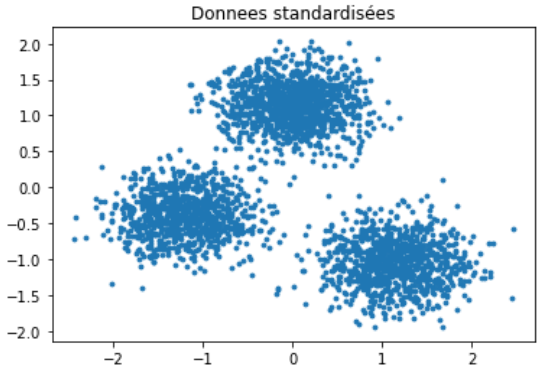

```
Affichage données standardisées
----------------------------------------
Dendrogramme 'complete' données standardisées
------------------------------------------------------------
Appel Aglo Clustering 'complete' pour une valeur de k fixée
Coefficient de silhouette :  0.21136832769058128
nb clusters = 4 , runtime =  24983.23 ms
```

On constate cette fois-ci qu'avec le linkage complete, la méthode ne fonctionne pas puisqu'il dénombre bien 4 clusters sur le dendrogramme mais la répartition spatiale est incorrecte comme le montre la figure ci-dessous. Ainsi, avec ces métriques, on ne parvient pas tout le temps à déterminer le bon nombre de clusters.

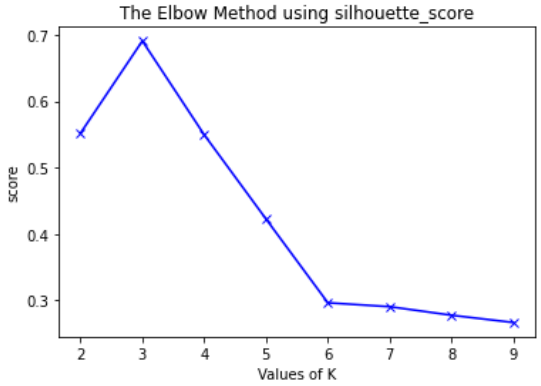


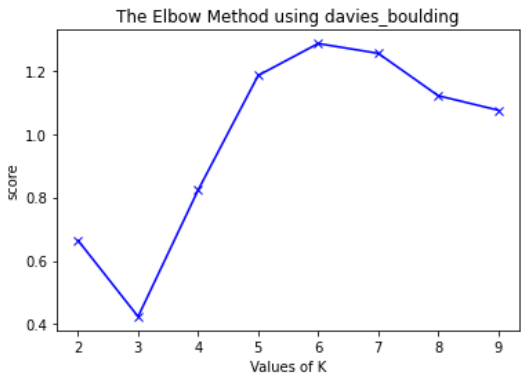
Selon le jeu de données choisi, il faudra choisir des méthodes de ressemblance différentes car toutes les métriques ne sont pas forcément adaptées pour déterminer le bon nombre de clusters. Cette méthode est beaucoup plus longue que les k-means, elle dépend aussi du nombre initial k et de la métrique. De manière générale, le choix des métriques à utiliser et la méthode, dépend de la géométrie (convexe, etc) du jeu de données, du nombre de points initiaux et de clusters.

Nous allons maintenant tenter d'automatiser l'évaluation du bon nombre de clusters pour plusieurs valeurs de k avec le clustering agglomératif. Ayant déjà testé la méthode du coude avec l'inertie et la distorsion précédemment, nous avons décidé d'utiliser les métriques du coefficient de silhouette et du Davies_Boulding sur le jeu de données « xclara.arff ».




```
for k in K:
    tps1 = time.time()
    model_scaled = cluster.AgglomerativeClustering(n_clusters=k, affinity='euclidean', linkage='complete')
    yhat = model_scaled.fit(data_scaled)
    tps2 = time.time()
    silhouette.append(metrics.silhouette_score(data_scaled, yhat.labels_, metric ='euclidean'))
    DB.append(metrics.davies_bouldin_score(data_scaled, model_scaled.labels_))
```

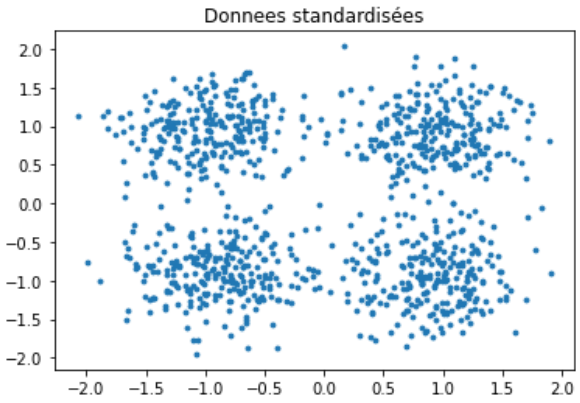


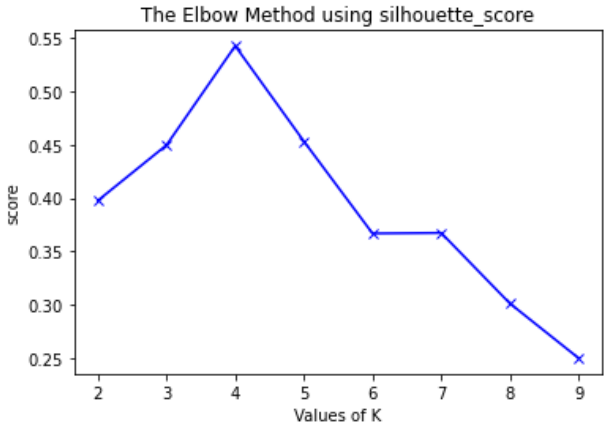


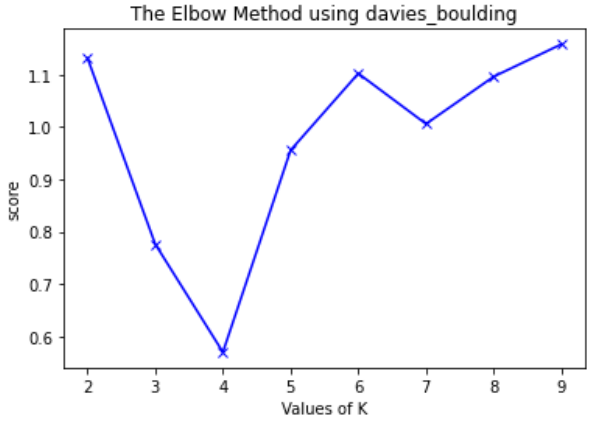
On constate pour la métrique silhouette une cassure à k = 3 avec un score de 0,7 environ et pour la métrique Davies_boulding une cassure minimum à k = 3 aussi, avec un score d'environ 0,42. Le nombre de clusters optimal est donc de 3. Nous arrivons donc pour ce jeu de données à déterminer le bon nombre de clusters pour les deux métriques.
Temps d'exécution environ de 0,15s, ce qui est ici légèrement plus rapide que k-means.

```
temps d'execution agglomeratif' :  0.14925074577331543  secondes
```

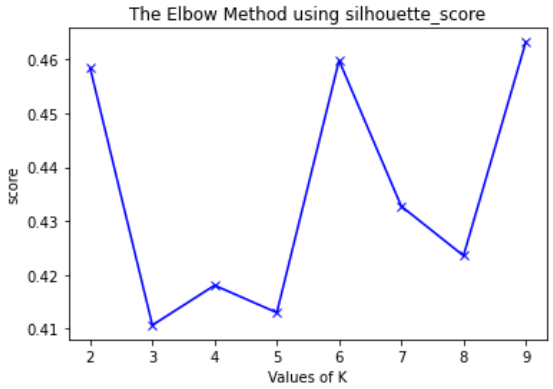
Autre test sur un jeu de données différent « square1.arff ».

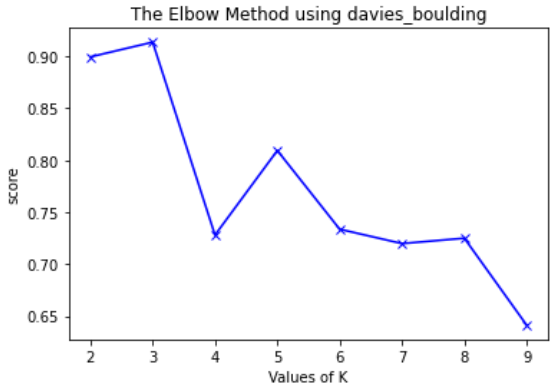






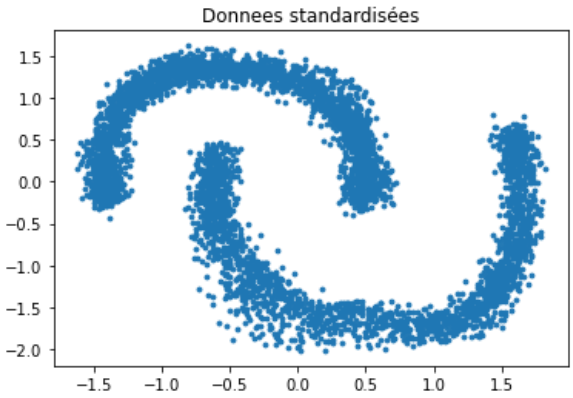
De nouveau, le nombre de clusters optimal est bien calculé avec la méthode du coude pour les deux métriques. Cependant, avec une figure un peu plus complexe, nous ne parvenons pas à déterminer le bon nombre de clusters avec précision.
Exemple :

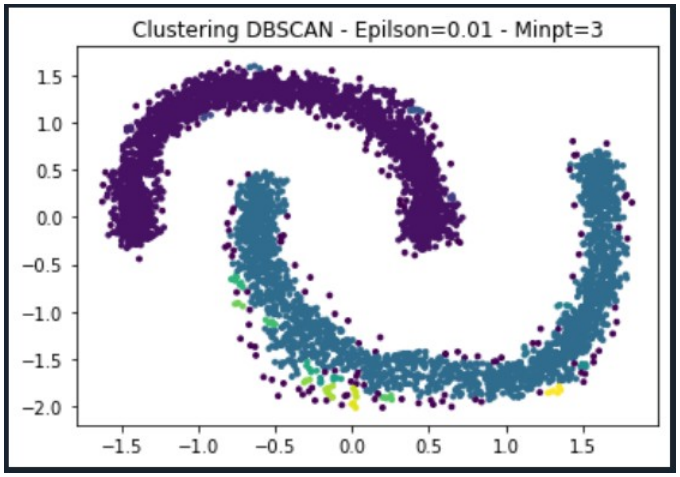




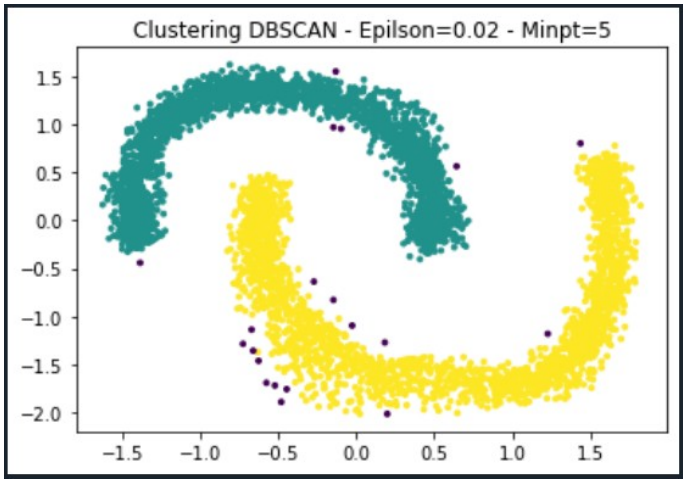
Nous passons à présent à l'étude de la méthode de clustering DBSCAN. La méthode s'appuie sur la densité de points pour un jeu de données passé en argument. Afin de déterminer le nombre de clusters k, nous devons d'abord déterminer des bons intervalles de valeur pour Epsilon (rayon du cercle) et le min-samples (nombre de points compris dans le cercle de rayon Epsilon). Pour déterminer Epsilon, nous allons utiliser la méthode des k-voisins les plus proches.

Test sur le jeu de données « banana.arff ».

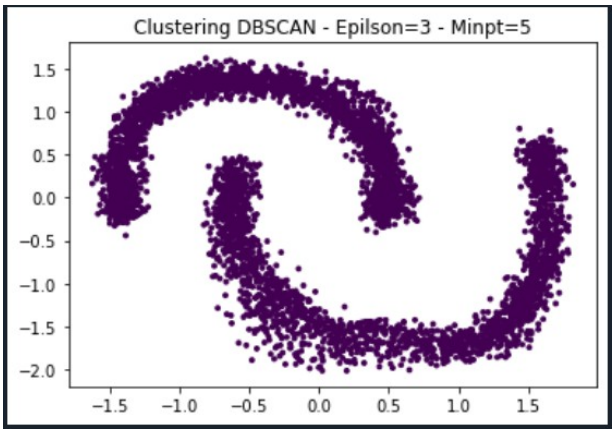




La valeur Epsilon est trop faible puisqu'on remarque trop de petits clusters voisins.

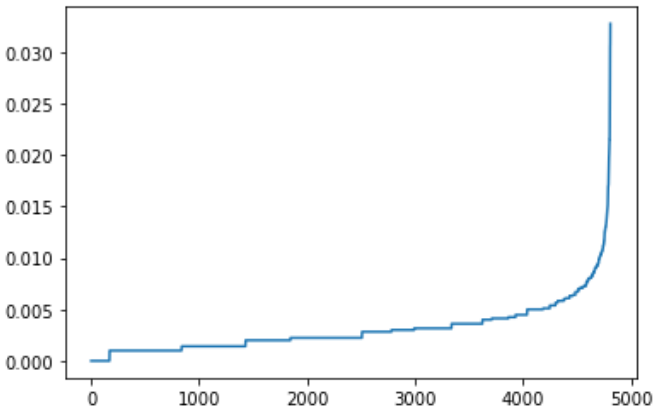


Une valeur d'Epsilon à 0,02 a l'air de bien correspondre, puisque les points voisins sont identifiés comme du bruit et non comme des clusters.



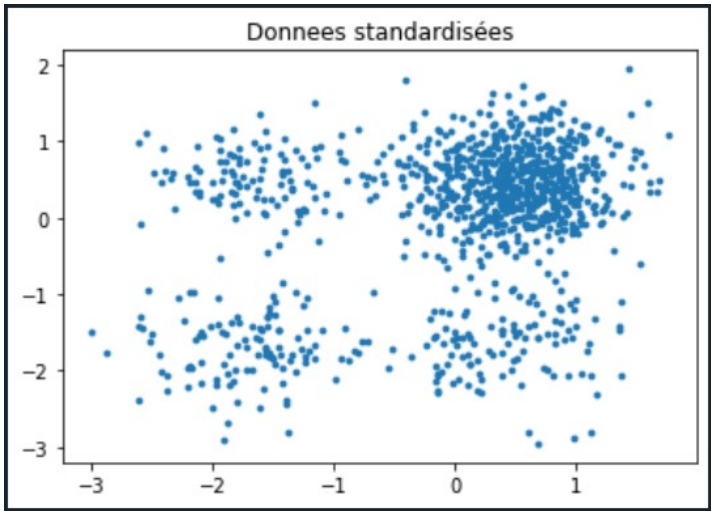
La valeur Epsilon est trop importante, le rendu ne nous montre qu'un seul et même cluster.

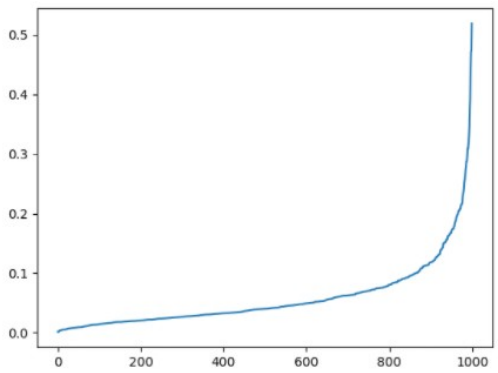
```
neigh = NearestNeighbors(n_neighbors=2)
nbrs = neigh.fit(datanp)
distances, indices = nbrs.kneighbors(datanp)

distances = np.sort(distances, axis=0)
distances = distances[:,1]
plt.plot(distances)
```



On constate que la valeur optimale pour Epsilon est lorsque la courbure est au maximum, comprise aux environs de 0,010.

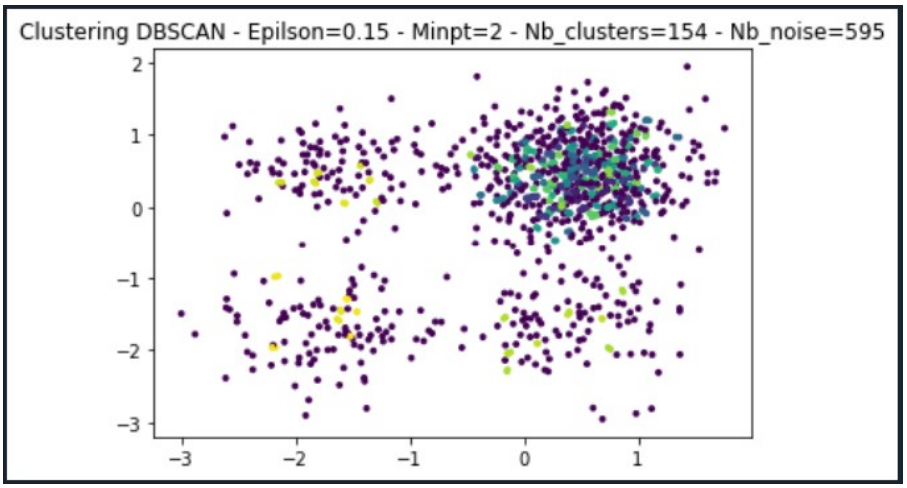
Test sur un autre jeu de données « sizes3.arff ».





On fixe Epsilon à 0,15 et min-samples à 2 pour faire le test.

```
Estimated number of clusters: 154
Estimated number of noise points: 595
```

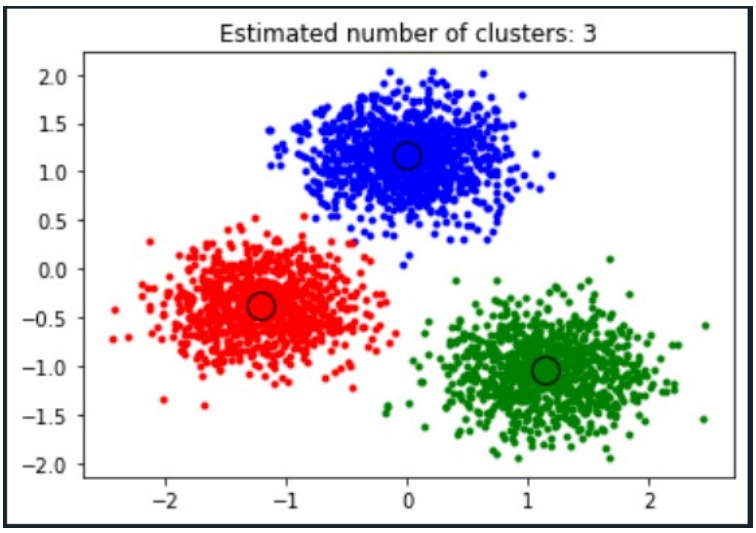


On voit ici que la méthode ne fonctionne pas très bien sur ce jeu de données, puisqu'on remarque beaucoup de bruit / clusters voisins. La densité de points est différente du nombre de clusters.

II – Analyse comparative sur les nouvelles données fournies

La méthode de clustering MeanShift vise à découvrir des « blobs » dans une densité homogène d'échantillons. Il s'agit d'un algorithme basé sur les centroïdes, qui fonctionne en mettant à jour les candidats pour que les centroïdes soient la moyenne des points dans une région donnée. Ces candidats sont ensuite filtrés dans une étape de post-traitement pour éliminer les doubles pour former l'ensemble final de centroïdes.

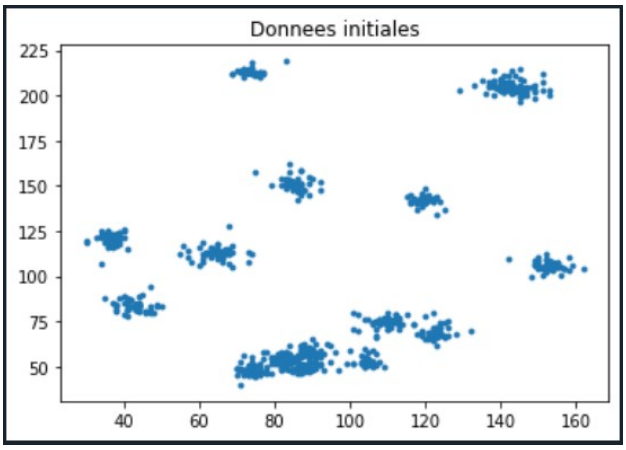
Nous allons tester notre méthode sur un jeu de données initial « xclara.arff » grâce à un algorithme MeanShift repris sur Github :

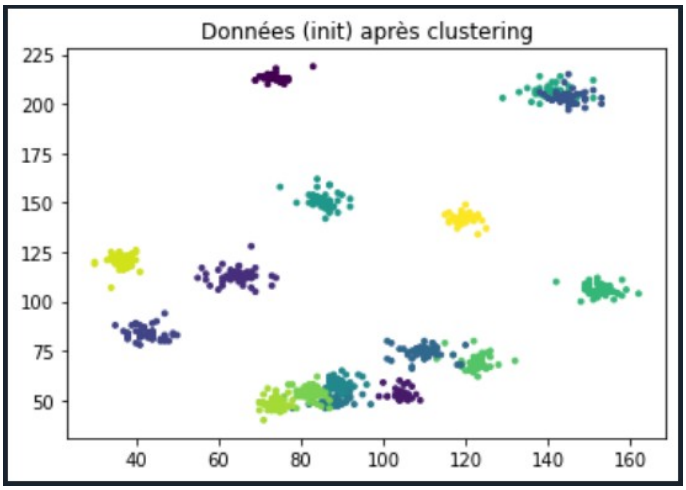


On remarque bien au centre de chacun des trois clusters les centroïdes représentant la moyenne des points dans chaque cluster.

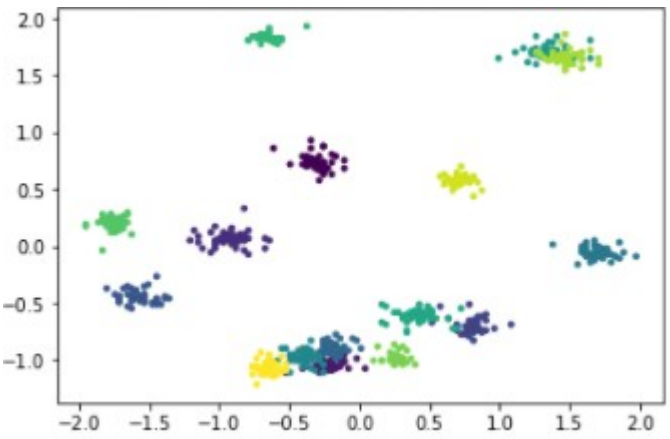
Nous allons à présent finir par de nouveaux jeux de données avec toutes les méthodes de clustering vues précédemment, dans l'ordre suivant : données standardisées, k-means, agglomératif, DBSCAN, et pour finir MeanShift.

Test « d32.txt » k = 16 / linkage = ward :

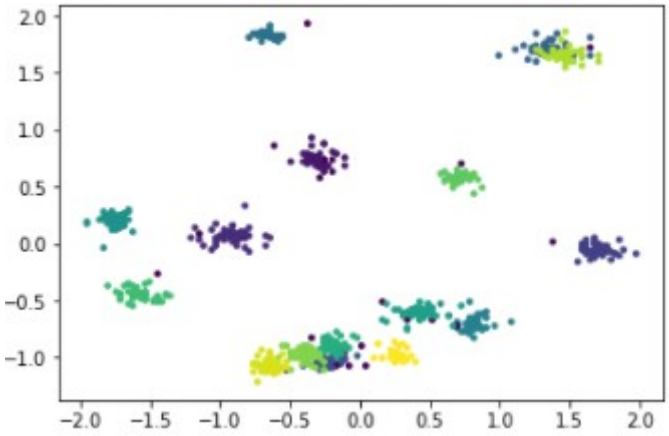




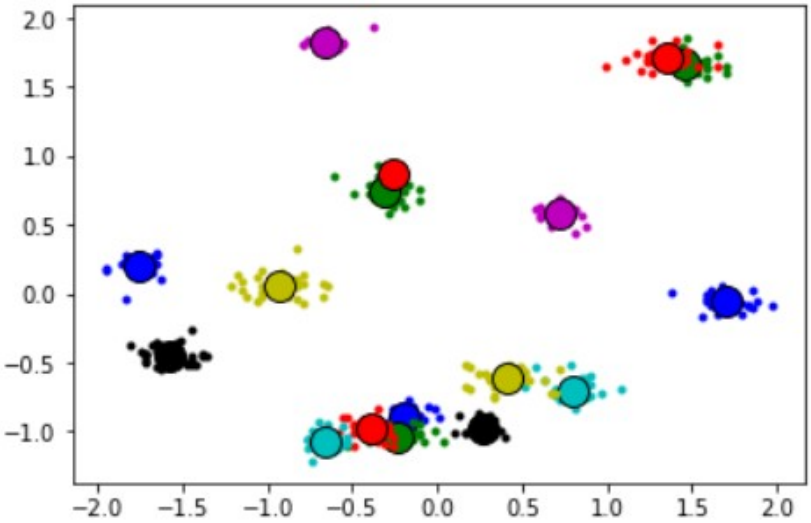
t = 0,0571s



t = 0,0149s



t = 0,069s / Epsilon = 0,6 / Min-samples = 2



t = 0,6069s

Test pour « n2.txt » :