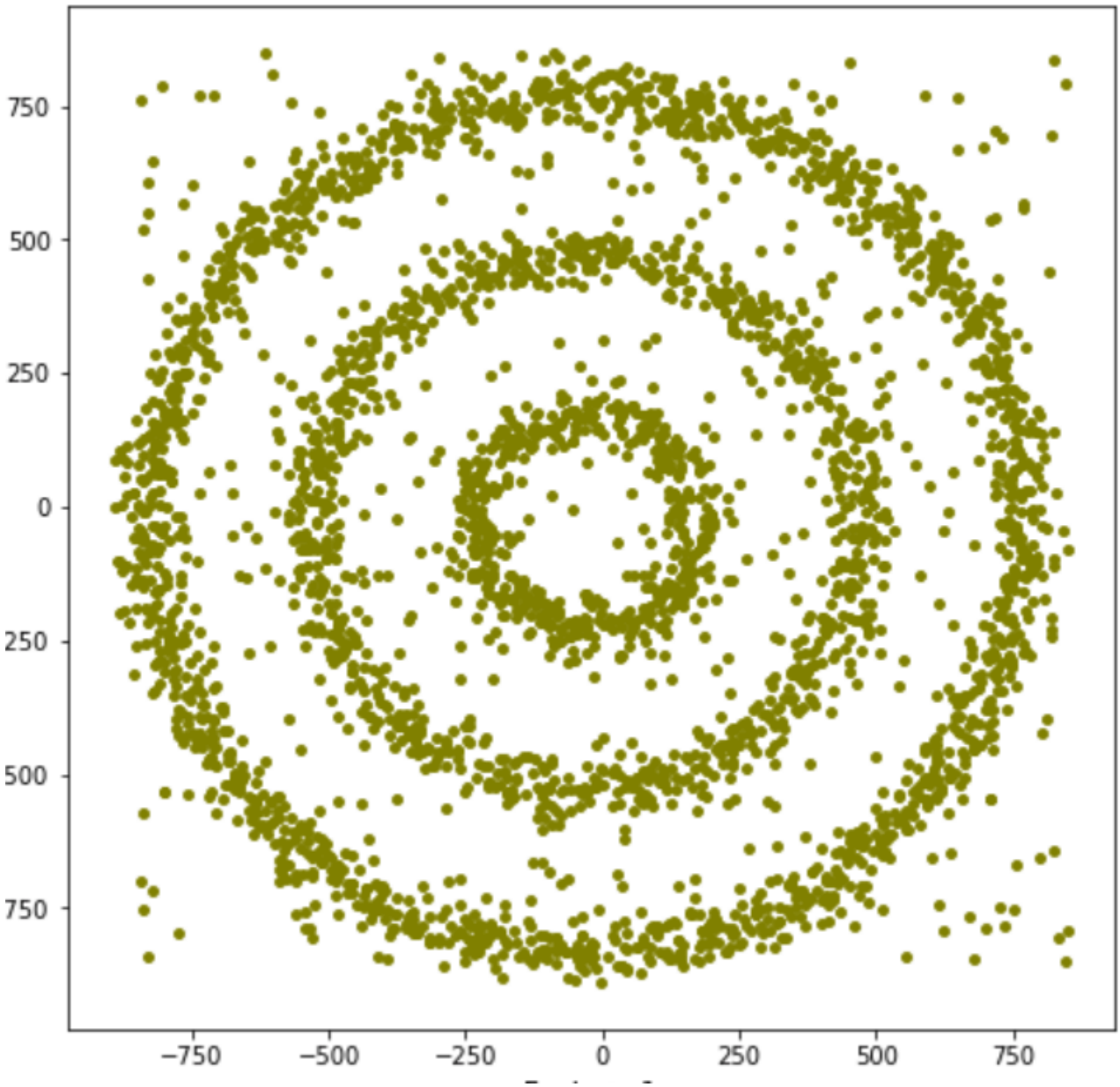
DBSCAN : Density-based spatial clustering of applications with noise ¶

Introduction

Les techniques de classfication (clustering en anglais) sont des méthodes qui permettent de réaliser le partitionnement des données en les regroupant en fonction de leurs caractéristiques. Fort de leurs succès dans des domaines tels que le marketing pour la segmentation de la clientèle, trois méthodes ont été développées pour répondre aux besoins de classification :

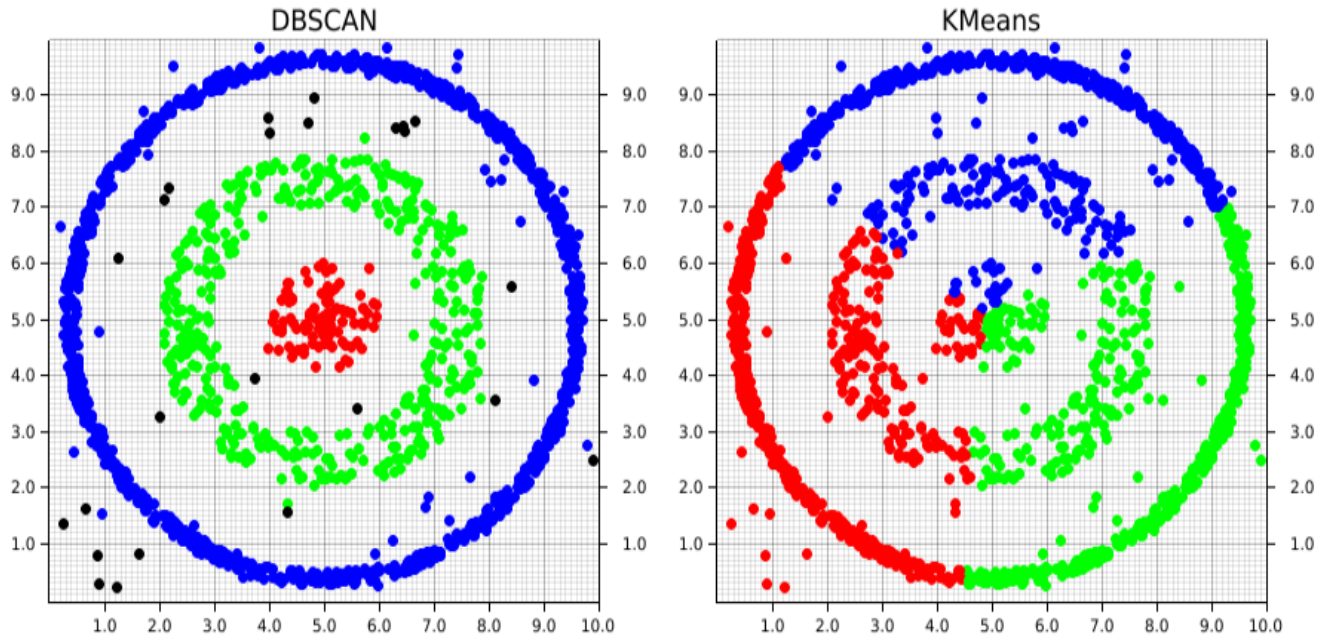
- *Les méthodes hiérarchiques* : Elles forment des connexions entre les individus et disposent d'une matrice de distance. Une méthode bien connue est la CAH.
- *Les méthodes centroïdes* : cette méthode utilise la méthode des k-moyennes (Kmeans). Grâce à cette technique, le choix de départ se fait seulement en une seule fois. On doit initialiser l'algorithme avec k points parmi les n individus. À la fin de la première étape, chaque classe se caractérise par la moyenne des sommes de chaque individu. On a donc k moyennes pour les k classes.
- *Les méthodes à densité* : il s'agit d'une méthode basée sur la densité. Les zones ayant plusieurs points sont beaucoup plus proches par rapport aux autres zones.

Nous étudierons un algorithme qui utilise cette dernière méthode : la DBSCAN (Density-based spatial clustering of applications with noise).
Supposons que vous devez classifiez les points sur cette image :



Combien de classes identifiez-vous ?

A vu d'oeil, nous pouvons observer trois classes en considérant chaque cercle comme étant une classe. Si nos données sont traitées avec les algorithmes de DBSAN et K-means, voici les résultats que nous avons :



Source (https://rust-ml.github.io/book/4_dbscan.html)

Quelle méthode se rapproche t-elle de notre détection naturelle ? Eh bien, l'algorithme du DBSCAN que nous étudierons en détail dans les paragraphes suivants.

Sommaire :

1. Définition

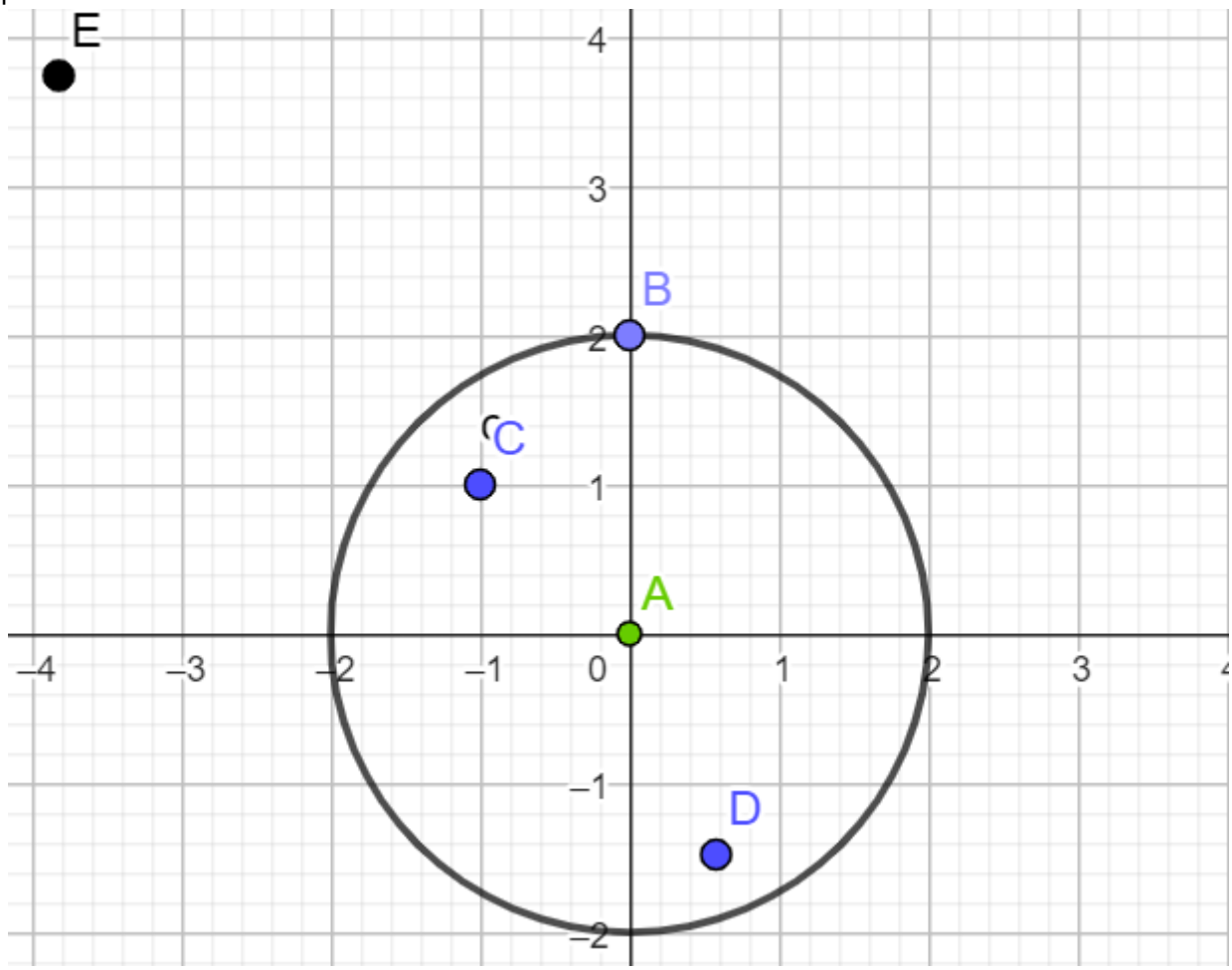
La méthode du DBSCAN forme des clusters là où la densité d'observations est importante. L'idée est de créer un chemin pour passer d'un point à un autre de proche en proche en restant à l'intérieur de la même classe. Elle dépend de deux paramètres ϵ et N_{min} qui sont fixés par l'utilisateur :

- ϵ : La distance minimale entre deux observations pour qu'elles soient voisines.
- N_{min} : Le nombre de voisins minimal qu'un point doit avoir pour être un point central. Nous définirons la notion de point central dans les lignes qui suivent.

Quelques mots de vocabulaires en plus :

- V_{ϵ}^{x} : ***Voisinage espilon de x***, l'ensemble des points dont la distance entre x et eux est inférieure ou égale à ϵ
- x est appelé ***point central*** si son voisinage espsilon V_{ϵ}^{x} contient au moins N_{min} points ce qui équivaut à dire que $|V_{\epsilon}^{x}| \geq N_{min}$
- x est appelé ***point non central*** s'il a au moins un point central dans son voisinage epsilon et au maximum $N_{min} - 1$ voisins c'est à dire : $|V_{\epsilon}^{x}| < N_{min}$ et $\exists\, x^{'}$ un point central tel que $distance(x^{'}, x) < \epsilon$
- ***Bruit*** : Toute observation qui n'est ni un point central ni un un point non central.

Exemple :



Si $\epsilon = 2$ et $N_{min} = 3$ alors :

- le point A est un point central
- le point E est un bruit
- les points B,C,D sont des points non centraux

Comment fonctionne cet algorithme de proche en proche ? Tout simplement en regroupant tous les points centraux qui sont voisins et leurs points non-centraux pour former un cluster. Pour y arriver, il faut procéder de la manière suivante :

- Etape 1 : Classer les différents points selon leur type (bruit, central, non-central)
- Etape 2 : Choisir un point central aléatoire
- Etape 3 : Identifier tous les k points centraux du voisinage du point choisi précédemment. Il faut identifier également tous les q points centraux voisins des k points centraux. Les points k, q et le point choisi aléatoirement feront partie du premier cluster.
- Etape 4 : Idenfier tous les z points non centraux voisins des q points trouvés précédemment (y compris le point choisi aléatoirement). Ces points z feront également parti du premier cluster.
- Etape 5 : Il faut choisir un autre point aléatoire parmi les points centraux et répeter les étapes 3 et 4 avec les points qui ne font pas partie des clusters déjà formés.
- Etape 6 : Lorsqu'il n'y aura plus de point central à choisir, tous les points qui resteront sont assimilés à des bruits.

2. Algorithme pas à pas

Pour mieux comprendre l'algorithme du DBSCAN, nous allons le dérouler avec un exemple fictif. Supposons que nous avons le jeu de données suivants :

	x	y
0	-0.473247	-0.891544
1	0.882596	2.235497
2	-1.209943	0.614270
3	-0.873258	-0.594967
4	0.866963	0.064704
5	0.292188	0.574638
6	1.367613	-0.684759
7	-1.046335	-0.402593
8	1.311132	0.565163
9	-1.117709	-1.480408

Essayons de classifier les 10 points que nous avons avec la méthode du DBSAN.

2.1. Choix des paramètres ϵ et N_{min}

Ces deux paramètres doivent être choisis par l'utilisateur.

- ϵ = 1
- N_{min} = 2

2.2. Calcul des distances

Nous calculons les distances euclidiennes entre les différents points. Nous choisissons naturellement cette distance pour utiliser les inertes intra-classe et inter-classe afin de mesurer la qualité de notre répartition. Toutefois, il est possible de choisir le type de distance que l'on souhaite. Voici ce que nous obtenons :

	1	2	3	4	5	6	7	8	9
2	3.408328								
3	1.676364	2.647093							
4	0.497962	3.330848	1.255233						
5	1.646382	2.170850	2.148385	1.861057					
6	1.653959	1.762679	1.502653	1.651132	0.768374				
7	1.852438	2.960260	2.886394	2.242669	0.901302	1.656086			
8	0.753328	3.268072	1.029940	0.258773	1.969537	1.657294	2.430384		
9	2.303476	1.724430	2.521553	2.473349	0.669138	1.018988	1.251198	2.548372	
10	0.872978	4.220091	2.096707	0.918565	2.515212	2.492192	2.609575	1.080176	3.175472

2.3. Identification des points centraux

Les points centraux sont tous les points qui ont au moins 2 points dans leur voisinage espilon. Les distances en orange sont toutes les distances inférieures ou égales à epsilon. Ainsi, nous avons :

- 1 avec V_1^1 = {4,8,10}
- 4 avec V_1^4 = {1,8,10}
- 5 avec V_1^5 = {7,9,6}
- 8 avec V_1^8 = {1,4}
- 10 avec avec V_1^{10} = {1,4}

Nous avons donc :

- Points centraux {1,4,5,8,10}
- Points non-centraux {6,7,9} car ils ont seulement l'observation 5 dans leurs voisinages
- Points bruits : {2,3}

2.4. Formation des clusters

- Formation du premier cluster :

- Choisissons un point central de manière aléatoire : 4. Le premier point du cluster sera donc 4
- Ajoutons tous les points centraux qui appartiennent au voisinage espilon de 4 : Nous ajoutons les points 8 et 10 . Le premier cluster contient maintenant {4,8,10}
- Ajoutons tous les points centraux qui appartiennent au voisinage epsilon des points centraux contenus dans le voisinage epsilon de 4 :
  - Les points centraux contenus dans V_1^8 sont 1 et 4. On ajoute 1 car 4 a déjà été ajouté
  - Les points centraux contenus dans V_1^{10} sont 1 et 4. Ils ont déjà tous été ajoutés
  - Si l'un des points ajoutés est un point central, on rajoute les points centraux de son voisinage. Ici 1 est un point central, mais on a déjà ajouté les points centraux de son voisinage {4,8,10}.
  - A chaque fois que nous croiserons un point central, nous ajouterons les autres points centraux de son voisinage jusqu'à ce qu'il n'y ait plus de points centraux à ajouter. On s'arrête donc ici avec {1,4,8,10} dans notre premier cluster...
- Ajoutons tous les points non centraux appartenant au voisinage espilon des points que nous avons ajouté : Ici, il y'en a pas. La formation du premier cluster est donc terminé, il contient les points {1,4,8,10}. Il reste

les points {2,3,5,6,7,9} à classifier.

- Pouvons nous former un deuxième cluster ? Nous le pouvons seulement s'il y'a encore des points centraux qui n'appartiennent pas au premier cluster. C'est le cas du point 5. Oui, nous pouvons donc continuer notre algorithme en reprenant les étapes précédentes

- Formation du deuxième cluster :

Nous choisissons parmi les points centraux restants un point aléatoire. Il ne reste plus que le point 5.

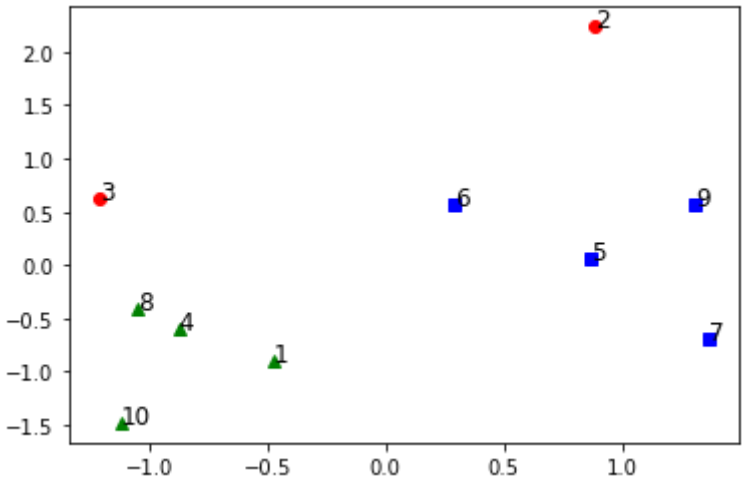
- Ajout des points centraux appartenant à V_1^5 : Nous n'en avons aucun car les points {7,9,6} qui font partie du voisinage de espilon de 5 ne sont pas des points centraux. Il n'y aura qu'un seul point central dans ce cluster.
- Rajoutons tous les points non centraux appartenant au voisinage espilon des points centraux que nous avons identifié précédemment. 5 étant le seul point central de ce cluster, nous ajouterons les points 6,7 et 9 de son voisinage.
- La formation du deuxième cluster est terminé. Il est composé des points {5,6,7,9}
- La formation d'un troisième cluster est elle possible ? Non, car tous les points centraux sont répartis dans les deux clusters.

2.5. Recapitulatif :

Nous avons formé deux classes :

- Classe 1 : {1,4,8,10}
- Classe 2 : {5,6,7,9}

Quid des points 2 et 3 ? L'algorithme des points DBSCAN ne les classe pas. Ces points sont appélés des "bruits". R renvoie 0 comme numéro de classe pour les bruits et Python -1.



Ci-dessus les résultats de la classification pour notre exemple. Les points de la classe 1 sont en vert, ceux de la classe 2 en bleu et les "bruits" en rouge

3. Implémentation avec Python

3.1. Chargement des librairies

Entrée [1]:

```
import numpy as np
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.cluster import KMeans
import timeit
from IPython.display import display_html
```

3.2. Présentation des données

Nous allons générer des données aléatoires grâce à la fonction make_moons du package sklearn. Elle va créer deux groupes de points qui forment chacun un demi-cercle. Cette fonction prend en paramètres d'entrée :

- n_samples : le nombre de points qu'on souhaite générer
- noise : l'écart type du bruit gaussien ajouté aux données
- random_state : pour fixer les points générés (un peu comme set.seed dans R)

Nous aurons en sortie un tableau data dans lequel nous stockerons les coordonnées(x,y) de nos points et un vecteur classe qui contient le numéro de classe de chaque point.

Un tableau nommé résult montrera les coordonnées de chaque point et sa classe.

Entrée [2]:

```
data,classe=make_moons(n_samples=600,random_state=170,noise=0.03)
data = pd.DataFrame(data, columns = ['x','y'])
result = data.copy()
result= result.assign(Classe=classe)
display('Coordonnées des points et leurs classes pour les 10 premiers points',result.head(1
```

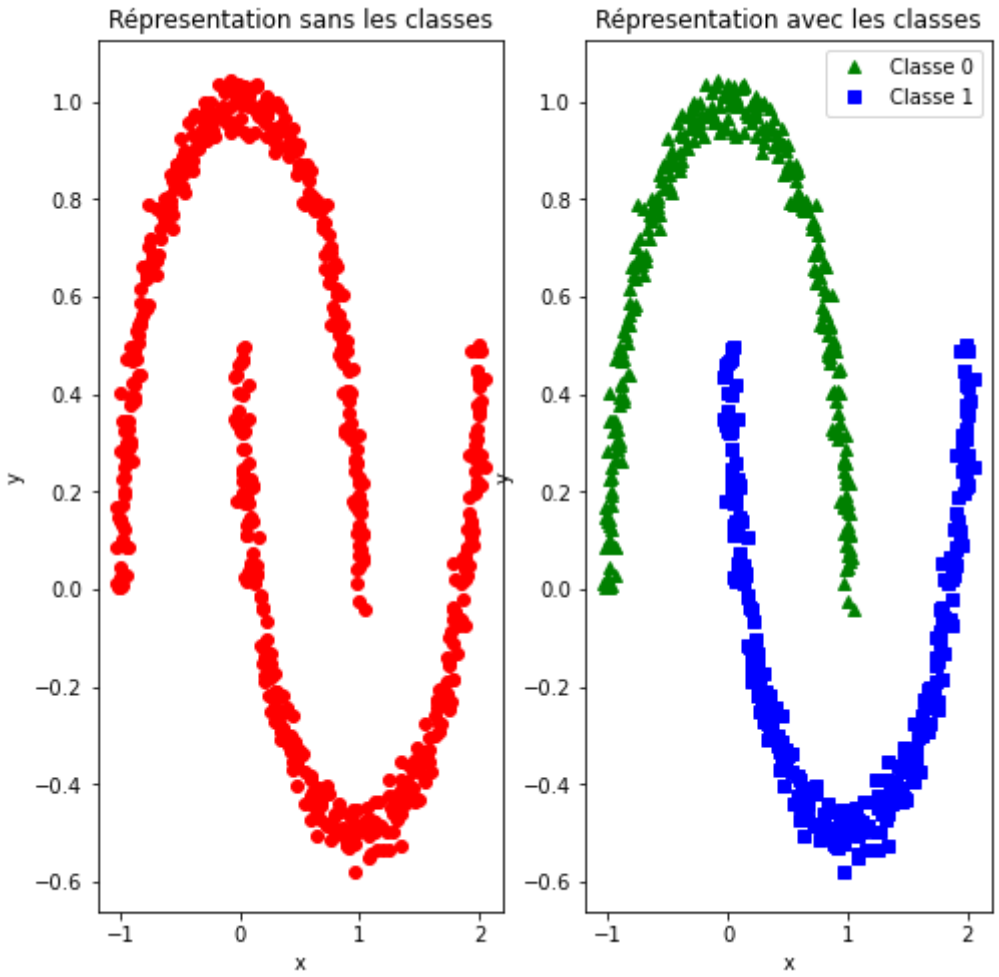
```
'Coordonnées des points et leurs classes pour les 10 premiers points'
```

	x	y	Classe
0	-0.879828	0.386436	0
1	1.842640	0.013196	1
2	1.057935	-0.471450	1
3	1.767575	-0.084397	1
4	-0.581656	0.795273	0
5	-0.896129	0.423548	0
6	0.727861	0.790193	0
7	0.224720	0.968445	0
8	0.491920	0.858591	0
9	-0.792281	0.662499	0

3.3. Visualisation des données

Entrée [3]:

```
fig,(ax1, ax2) = plt.subplots(1, 2,figsize=(8,8))
ax1.plot(data.x, data.y,'o',color='red')
ax2.plot(data.x[classe==0], data.y[classe==0], 'g^',data.x[classe==1] , data.y[classe==1],
ax1.set(xlabel='x',ylabel='y')
ax2.set(xlabel='x',ylabel='y')
ax2.legend(['Classe 0','Classe 1'])
ax1.title.set_text('Répresentation sans les classes')
ax2.title.set_text('Répresentation avec les classes')
```



3.4. Application

3.4.1. Centrage et réduction des données

Avant de soumettre nos données à notre algortihme, nous allons les centrer et réduire. Avec Python, la fonction qui nous permet de faire un centrage réduction est StandardScaler. Elle provient de la librairie sklearn qui fournit également l'algorithme pour appliquer la méthode DBSCAN.
Passons donc à la transformation de nos données :

Entrée [4]:

```
scaler = StandardScaler()
scaler.fit(data)
data = pd.DataFrame(scaler.transform(data),index=data.index,columns=data.columns)
print("Les cinq premières lignes de notre tableau après centrage réduction :")
data.head(5)
```

```
Les cinq premières lignes de notre tableau après centrage réduction :
```

Out[4]:

	x	y
0	-1.593198	0.277808
1	1.550354	-0.476625
2	0.644278	-1.456245
3	1.463678	-0.673890
4	-1.248908	1.104192

3.4.2 Mise en pratique

Comme je vous l'avait dit précedemment le package sklearn nous permettra d'appliquer la méthode DBSCAN. Elle prend en entrée les paramètres suivants :

- *eps* : la valeur de ϵ qui nous permettra de calculer le V_{ϵ}^{x}
- *min_samples* : $N_{min} + 1$ car il comprend le point lui même
- *metric* = le type de distance utilsé.

Nous travaillerons avec les valeurs par défaut qui sont **eps= 0.5** et **min_samples=5**. Toutefois, nous pouvons choisir les valeurs que nous voulons. Plus tard, nous verrons comment optimiser ces paramètres. Pour la distance, nous utiliserons la distance **euclidienne**.

Entrée [5]:

```
start_dbscan = timeit.default_timer()
dbscan = DBSCAN(metric="euclidean")
dbscan.fit(data)
stop_dbscan= timeit.default_timer()
time_dbscan = stop_dbscan - start_dbscan
```

3.4.3 Analyse des résultats

Les sorties de la fonction DBSCAN sont :

- *core_sample_indices*: l'indice des poins centraux que nous stockerons dans ***points_centraux***
- *labels_* : Renvoie un vecteur qui comprend le numéro de classe prédit pour chaque point. Le vecteur ***classe_predict*** contiendra le résultat de notre classification

Entrée [6]:

```
points_centraux = dbscan.core_sample_indices_
classe_predict = dbscan.labels_
print("Nombre de points centraux :",len(points_centraux))
print("Le numéro de classes des 10 premiers points:",classe_predict[:11])
print("Nombre de points classés comme bruit :",len(classe_predict[classe_predict==-1]))
```

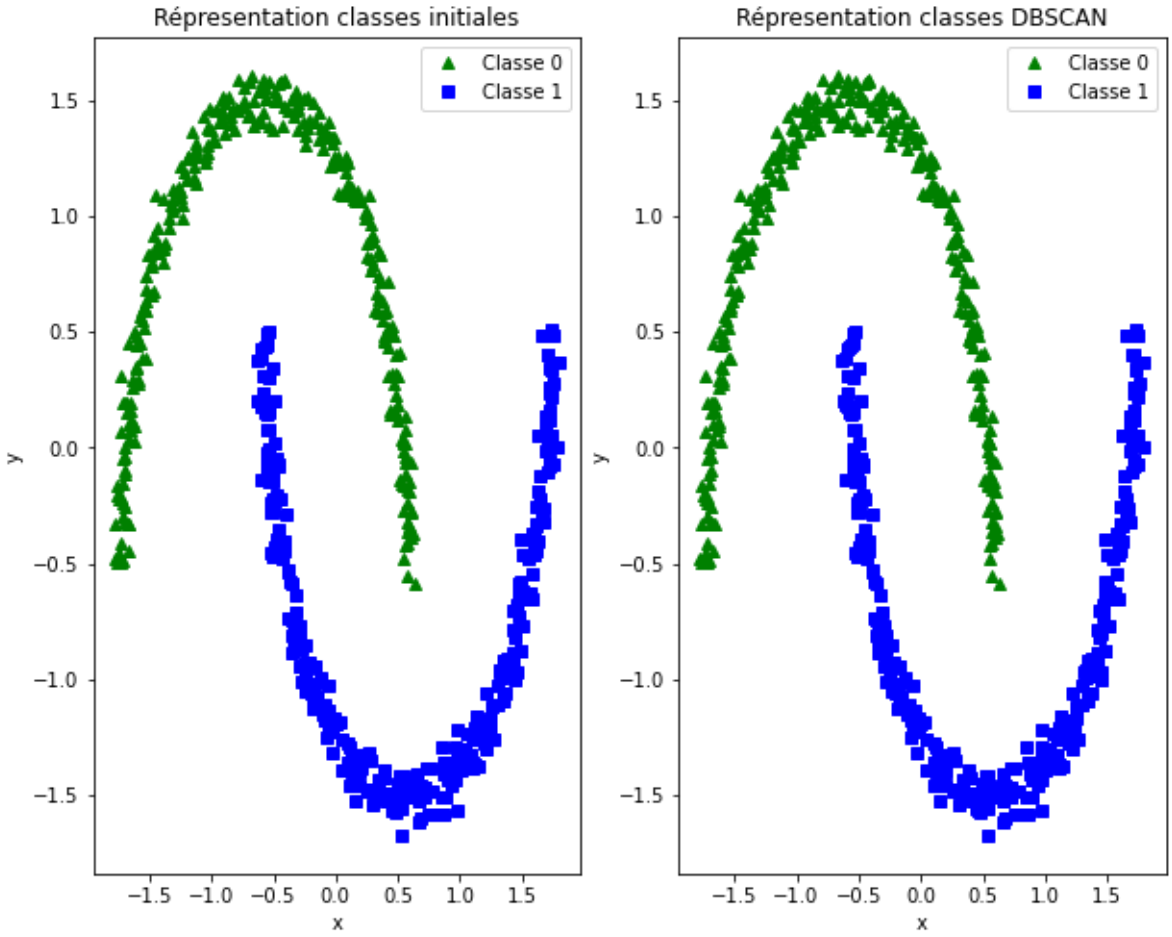
```
Nombre de points centraux : 600
Le numéro de classes des 10 premiers points: [0 1 1 1 0 0 0 0 0 0 1]
Nombre de points classés comme bruit : 0
```

Dans cet exemple, tous les points A (x,y) de nos données ont le cardinal de leur voisinage qui est supérieur à N_{min} = min_samples -1 $\Leftrightarrow$ $|V_{0.5}^{A}| \geq 4$. En effet, le nombre de points centraux est égal au nombre de points que nous avons dans notre tableau de données (data), cela explique l'absence de bruit.

3.4.4. Visualisation des résultats

Entrée [7]:

```
fig,(ax1, ax2) = plt.subplots(1, 2,figsize=(10,8))
ax1.plot(data.x[classe==0], data.y[classe==0], 'g^',data.x[classe==1] , data.y[classe==1],
ax2.plot(data.x[classe_predict==0], data.y[classe_predict==0], 'g^',data.x[classe_predict==
ax1.set(xlabel='x',ylabel='y')
ax2.set(xlabel='x',ylabel='y')
ax1.legend(['Classe 0','Classe 1'])
ax2.legend(['Classe 0','Classe 1'])
ax1.title.set_text('Répresentation classes initiales')
ax2.title.set_text('Répresentation classes DBSCAN')
```



Les graphes sont les mêmes.On peut observer que les classes sont homogènes,bien séparées et correspondent aux classes initiales.

3.5. Comment choisir les paramètres ?

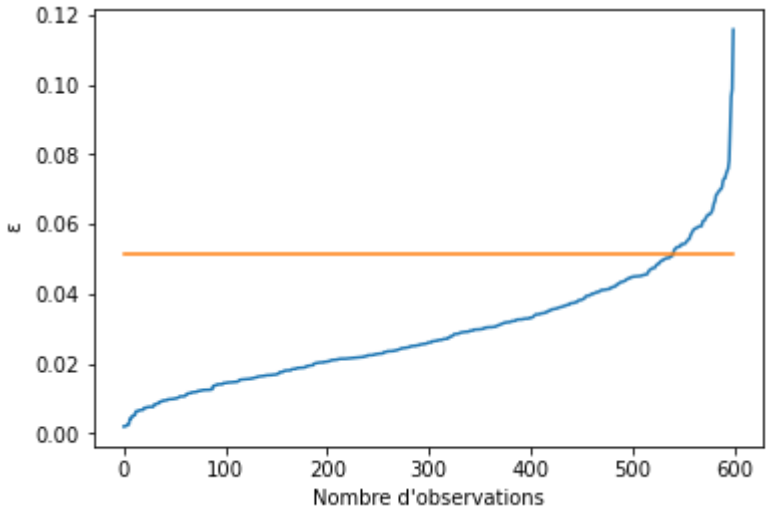
Nous allons calculer la matrice de distance (euclidienne) pour nos données. Pour chaque observation, nous allons regarder ses N_min plus proches voisins. Ensuite, nous allons représenter les distances entre les voisins proches et choisir ϵ de tel sorte que 90% des observations aient une distance au proche voisin inférieure à ϵ. Pour plus de détails, cliquez sur ce lien (https://towardsdatascience.com/machine-learning-clustering-dbscan-determine-the-optimal-value-for-epsilon-eps-python-example-3100091cfbc).

Un package qui permet de réaliser cette opération est sklearn par le biais de sa fonction NearestNeighbors.

Entrée [8]:

```
voisin = NearestNeighbors(n_neighbors=4)
nbrs = voisin.fit(data)
distances, indices = nbrs.kneighbors(data)
distances = np.sort(distances, axis=0)
distances = distances[:,1]
plt.plot(distances)
ind_eps = int(0.9 * 600)
best_eps = distances[ind_eps]
plt.plot(range(600),np.repeat(best_eps,600))
plt.xlabel("Nombre d'observations")
plt.ylabel(" ε");
print("Dans notre cas, il faut donc choisir epsilon :",np.round(best_eps,3))
```

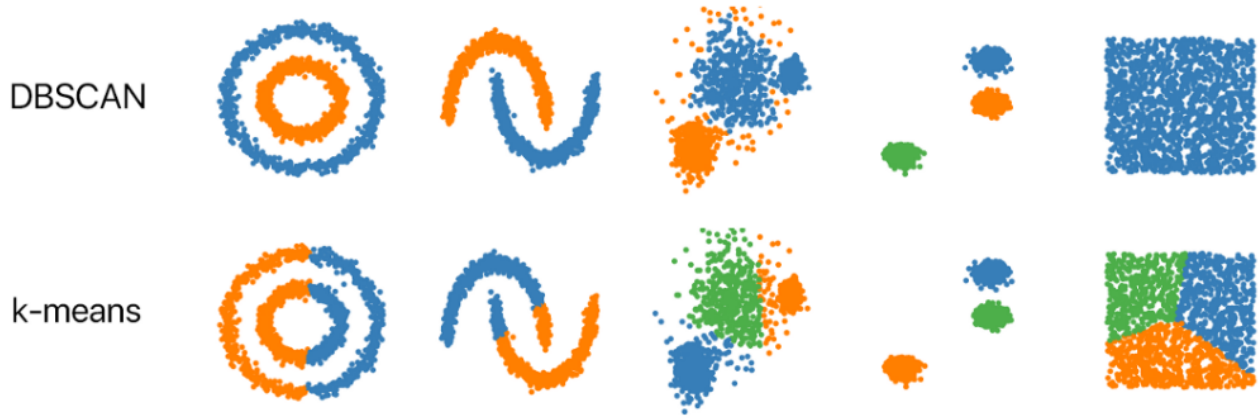
```
Dans notre cas, il faut donc choisir epsilon : 0.052
```



4. Comparaison avec Kmeans

Dans cette partie, nous allons comparer l'algorithme de Kmeans et celui du DBSCAN grâce :

- aux inerties (intra et inter) : des mesures de la qualité de la partition
- au taux de bonnes prédictions : Un indicateur qui nous dit le nombre de bonnes valeurs prédites. Même si les deux méthodes sont non-supervisées, nos données aléatoires ont été générées avec des labels grâce à la fonction make_noons. Nous pourrons donc essayer de regarder si les classes prédites par nos algorithmes sont conformes à celles initiales.
- le temps : le temps mis par chaque algorithme pour classifier. La fonction timeit.default_timer du package timeit nous permet de calculer le temps entre le début et la fin de l'application de chaque algorithme.
- aux graphiques : des graphes réprésentants les classes



DBSCAN vs K-means, credit

Source (https://towardsdatascience.com/understanding-dbscan-algorithm-and-implementation-from-scratch-c256289479c5)

4.1. Algorithme de Kmeans

Nous allons demander à l'algortihme de kmeans de créer deux clusters avec les données que nous avons générées dans la première partie pour rester en phase avec le résultat de la classification DBSCAN.
Pour rappel, le package sklearn fournit l'algorithme KMeans. La fonction KMeans prend en entrée ***n_clusters*** le nombre de clusters et ***random_state*** le paramètre qui permet de fixer le numéro des classes. Sa sortie est appelée grâce à ***labels_*** qui permet d'afficher le numéro des classes, nous stockerons cette sortie dans le vecteur ***predict_kmeans***.

Entrée [9]:

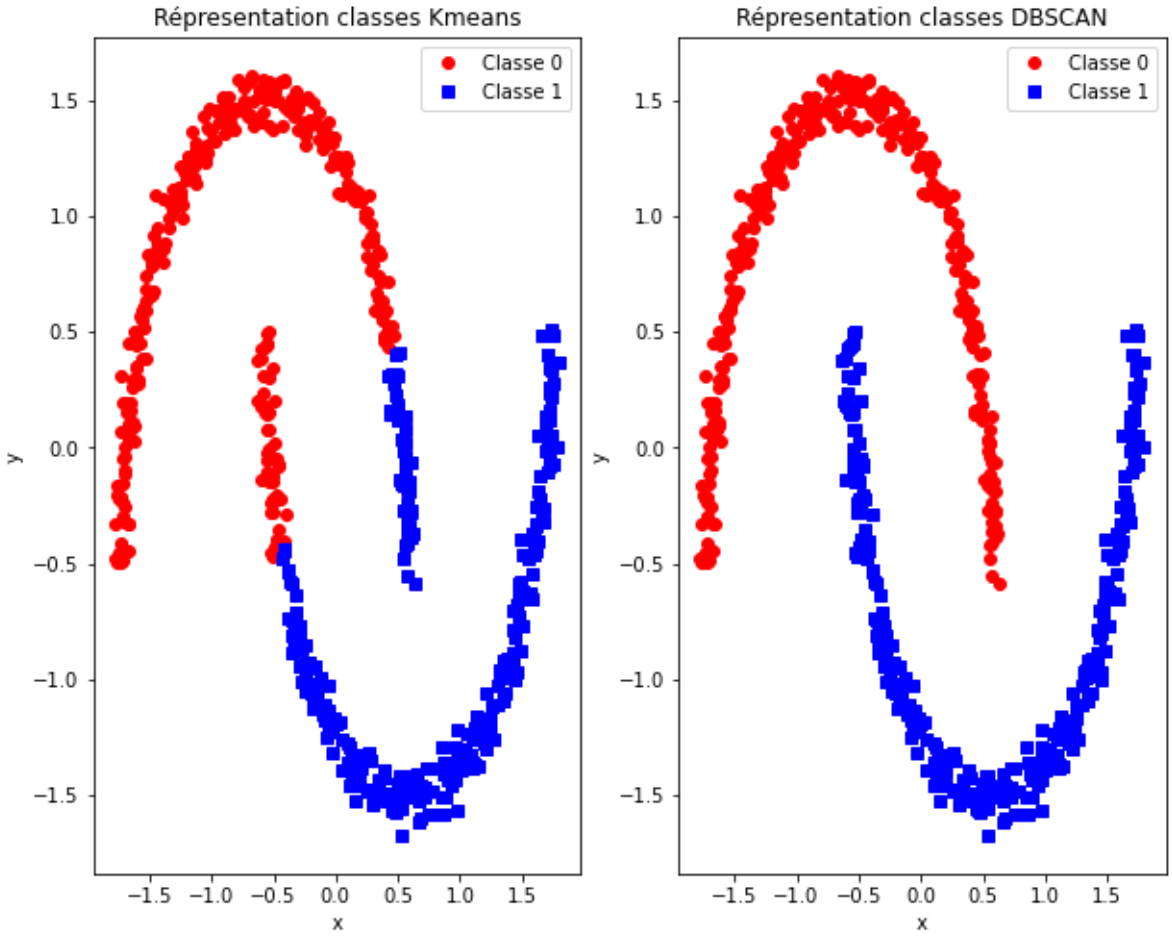
```
start_kmeans = timeit.default_timer()
km = KMeans(n_clusters=2,random_state=170).fit(data)
predict_kmeans = km.labels_
stop_kmeans= timeit.default_timer()
time_kmeans = stop_kmeans - start_kmeans
print("Le numéro de classes des 10 premiers points:",predict_kmeans[:11])
```

```
Le numéro de classes des 10 premiers points: [0 1 1 1 0 0 0 0 0 0 1]
```

4.2. Visualisation

Entrée [10]:

```
fig,(ax1, ax2) = plt.subplots(1, 2,figsize=(10,8))
ax1.plot(data.x[predict_kmeans==0], data.y[predict_kmeans==0], 'ro',data.x[predict_kmeans==
ax2.plot(data.x[classe_predict==0], data.y[classe_predict==0], 'ro',data.x[classe_predict==
ax1.set(xlabel='x',ylabel='y')
ax2.set(xlabel='x',ylabel='y')
ax1.legend(['Classe 0','Classe 1'])
ax2.legend(['Classe 0','Classe 1'])
ax1.title.set_text('Répresentation classes Kmeans')
ax2.title.set_text('Répresentation classes DBSCAN')
```



4.3. Calcul des inerties

La fonction ***inertie_clustering*** ci dessous sera utilisée afin de calculer les inerties intra-classes et inter-classes des différentes partitions obtenues pour chacune des méthodes de classification.

- Entrées :
  - D : le dataframe des points rangés en colonne x et y
  - n_clusters : le nombre de clusters qu'on souhaite obtenir
  - predict : le vecteur des classes prédites par l'algorithme de classification
- Variables calculées :
  - g : centre de gravité
  - g_groupe : centre de gravité de chaque classe (rangés en lignes)
  - n_points_cluster :nombre de points dans chaque classe
- Sortie : Nous obtenons un tableau qui renvoie l'inertie intra-classe, l'inertie inter-classe et l'inertie totale.

Entrée [11]:

```
def inertie_clustering (D,n_clusters,predict):
    g = D.mean(axis=0)
    g_groupe = [D[predict==i].mean(axis=0) for i in np.arange(n_clusters)]
    n_points_cluster = np.bincount(predict)
    inertie_class= [(1/n_points_cluster[i])*sum([(np.linalg.norm(g_groupe[i]-D[predict==i].
    inertie_intra = sum((n_points_cluster/ D.shape[0]) * inertie_class)
    inertie_inter = sum([(n_points_cluster[i]/ D.shape[0]) *(np.linalg.norm(g_groupe[i]-g))
    inertie_tot = (1/D.shape[0])*sum([np.linalg.norm(g-D.iloc[j])**2 for j in range(D.shape
    result = pd.DataFrame ([['Intra',inertie_intra],['Inter',inertie_inter],['Total',inerti
    return result
```

Pour appliquer cette fonction à Kmeans et DBSCAN, on aura :

- D = data
- n_clusters = 2
- predict : Ces vecteurs stockent les numéro de classe pour chaque point.
  - predict_kmeans pour Kmeans
  - predict_dbscan= classe_predict pour DBSCAN

Entrée [12]:

```
predict_dbscan = classe_predict
result_kmeans = inertie_clustering(data,2,predict_kmeans)
result_dbscan = inertie_clustering(data,2,predict_dbscan)
```

Entrée [13]:

```
df1 = result_kmeans.style.set_table_attributes("style='display:inline'").set_caption('Kmean
df2 = result_dbscan.style.set_table_attributes("style='display:inline'").set_caption('DBSCA
display_html(df1._repr_html_()+df2._repr_html_(), raw=True)
```

Kmeans

	0	1
0	Intra	0.829636
1	Inter	1.170364
2	Total	2.000000

DBSCAN

	0	1
0	Intra	1.066783
1	Inter	0.933217
2	Total	2.000000

4.4. Calcul du taux de bonnes prédiction

Calculons maintenant le taux de bonnes prédiction. Un point est bien prédit si le numéro de classe qui lui a été attribué par la méthode de classification est égal à son numéro de classe initital. Le taux de bonnes prediction est égal à :

$$\frac{Nombre\ de\ points\ bien\ prédits}{Nombre\ de\ points\ total}$$

Soient :

- N_tot : Nombre de points total
- N_kmeans : Nombre de points bien prédits avec Kmeans
- N_dbscan : Nombre de points bien prédits avec DBSCAN
- Acc_kmeans : Taux de bonnes prédictions avec KMeans
- Acc_dbscan : Taux de bonnes prédictions avec DBSCAN

Entrée [14]:

```
N_tot = len(data)
N_kmeans = len(data[classe==predict_kmeans])
N_dbscan = len(data[classe==predict_dbscan])
Acc_kmeans = N_kmeans/N_tot
Acc_dbscan = N_dbscan/N_tot
print ("Le taux de bonne prédiction avec KMeans est :",np.round(Acc_kmeans,3))
print ("Le taux de bonne prédiction avec DBSCAN est :",Acc_dbscan)
```

```
Le taux de bonne prédiction avec KMeans est : 0.842
Le taux de bonne prédiction avec DBSCAN est : 1.0
```

4.5. Analyse finale

Entrée [15]:

```
result_kmeans.loc[3] = ["Accuracy",Acc_kmeans]
result_kmeans.loc[4] = ["Temps exécution",time_kmeans]
result_dbscan.loc[3] = ["Accuracy",Acc_dbscan]
result_dbscan.loc[4] = ["Temps exécution",time_dbscan]
df1 = result_kmeans.style.set_table_attributes("style='display:inline'").set_caption('Kmean
df2 = result_dbscan.style.set_table_attributes("style='display:inline'").set_caption('DBSCA
display_html(df1._repr_html_()+df2._repr_html_(), raw=True)
```

Kmeans

	0	1
0	Intra	0.829636
1	Inter	1.170364
2	Total	2.000000
3	Accuracy	0.841667
4	Temps exécution	0.078110

DBSCAN

	0	1
0	Intra	1.066783
1	Inter	0.933217
2	Total	2.000000
3	Accuracy	1.000000
4	Temps exécution	0.020745

- L'inertie intra classe obtenue avec Kmeans est plus faible que celle obtenue avec DBSCAN, les classes sont donc plus homogènes avec Kmeans

- L'inertie inter classe obtenue avec Kmeans est plus grande que celle obtenue avec DBSCAN, les classes sont mieux séparées avec DBSCAN
- Nos données s'adaptent parfaitement à DBSCAN car l'accuracy est 1. Cependant, les partitions avec Kmeans ne sont pas médiocres puisqu'on à 0.842 de chance de bien prédire.
- Le temps d'exécution de Kmeans répresente environ 3 fois celui de DBSCAN.

5. Comment implémenter DBSCAN sur R ?

Pour utiliser DBSCAN dans R avec nos données, il faut exécuter le code suivant:

```
library(fpc) # insallation de la librairie fpc
data <- read.csv("data_appli.csv",sep=",") #chargement des données à partir d'un fichier excel ( data répresente le tableau de donné utilisé dans l'application sous format excel)
data <- data[,-1]
data
db <- fpc::dbscan(data, eps = 0.5, MinPts = 5) #algorithme dbscan
classe <- db[["cluster"]] # les différentes classes
plot(db, data, main = "DBSCAN", frame = FALSE) # representation
```

Vous pouvez télécharger le code R et le fichier à cette adresse : DBSCAN avec R (https://drive.google.com/drive/folders/1OPn4tlmXv4GEAhffVk8kpyJ01tsSjvuu?usp=sharing)

Conlusion

L'algorithme de DBSCAN est très utilisé pour classifier les données qui forment des clusters qui ne peuvent pas être séparés par une frontière linéaire (ou des clusters qui ne sont pas de forme sphérique). Un autre de ses avantages est la gestion des valeurs aberrantes(les points "bruits"). Contrairement à KMeans, le nombre de classes n'a pas besoin d'être fixé au départ mais il faut quand même choisir les paramètres ϵ et N_{min}. Cette méthode n'est pas prédictive et est difficile à appliquer sur des données de grandes dimensions.

Liens utiles

Clustering with DBSCAN, Clearly Explained!!! (https://www.youtube.com/watch?v=RDZUdRSDOok&t=418s)
DBSCAN avec sklearn.cluster (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html)
Make moons (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_moons.html)
KMeans avec sklearn (https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html)