MEGA Adrian
RABARY Anaïs

5SDBD

TP Apprentissage Non Supervisé

Introduction

Ce rapport a pour but de rendre compte de nos travaux en TP d'apprentissage non supervisé. Les deux chapitres traiteront des deux parties du TP, respectivement sur DBSCAN et SNN. Le code peut être trouvé sur le dépôt suivant

https://github.com/dx07/5SDBD-Apprentissage/tree/master/Apprentissage-Non-Supervise

Nous parlerons dans ce rapport des résultats de nos algorithmes de clustering sur les 4 jeux de données proposés. Il sera question d'appliquer les algorithmes développés et d'étudier les paramétrages, la qualité du clustering, etc…

Table des matières

I – DBSCAN de scikit-learn

I.1/ Présentation

DBSCAN est une méthode de clustering de données non labélisées. DBSCAN présente l'intérêt de ne pas obliger de fixer le nombre de clusters à l'avance (contrairement à Kmeans), tout en étant robuste au bruit et aux anomalies. De plus, il va nous permettre d'identifier des clusters concaves et d'apprécier leur qualité. En faisant varier les paramètres de la fonction DBSCAN de scikit-learn, nous allons tenter d'affiner la qualité de nos clusters.

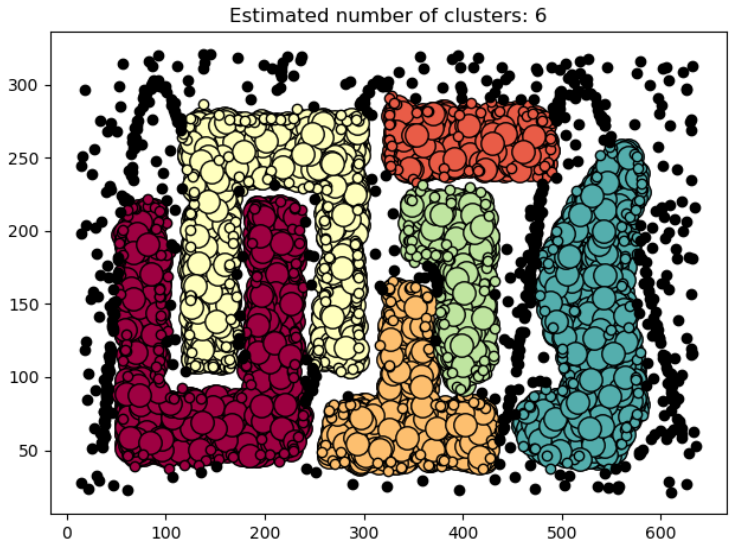
Pour faire varier ces paramètres, nous avons tout d'abord testé plusieurs valeurs à la main par petites variations (jusqu'à manipuler des valeurs d'epsilon à décimales). Nous avons ensuite testé des valeurs en boucle pour tenter d'obtenir des clustering différents et de meilleure qualité.

Lors de notre recherche en modifiant les paramètres dans des boucles, nous avons constatés que nous pouvons obtenir plusieurs solutions satisfaisables. On remarque aussi une tendance : plus epsilon est élevé, plus min_sample doit être élevé pour obtenir une solution satisfaisable (par solution satisfaisable, on entend des solutions qui nous détecte un nombre de cluster correct par rapport à ce qu'on pourrait s'attendre à l'œil nu).

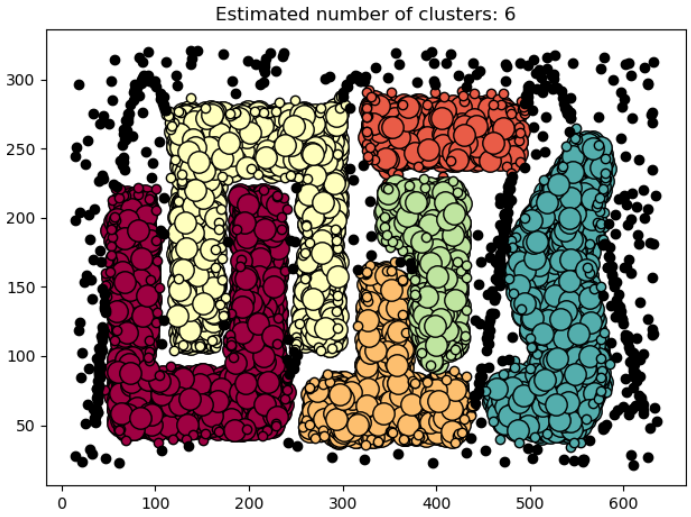
Ci-après sont exposés les meilleurs résultats que nous avons trouvés, pour chaque dataset. Ce ne sont pas forcément les résultats où il y a un nombre de bruit faible, car on observe du bruit à l'œil nu. Mais on a privilégié la valeur du coefficient de silhouette. Plus ce dernier est proche de 1, meilleure est l'assignation des points à leur clusters.

I.2/ Paramètres epsilon et min_samples

Dataset 1



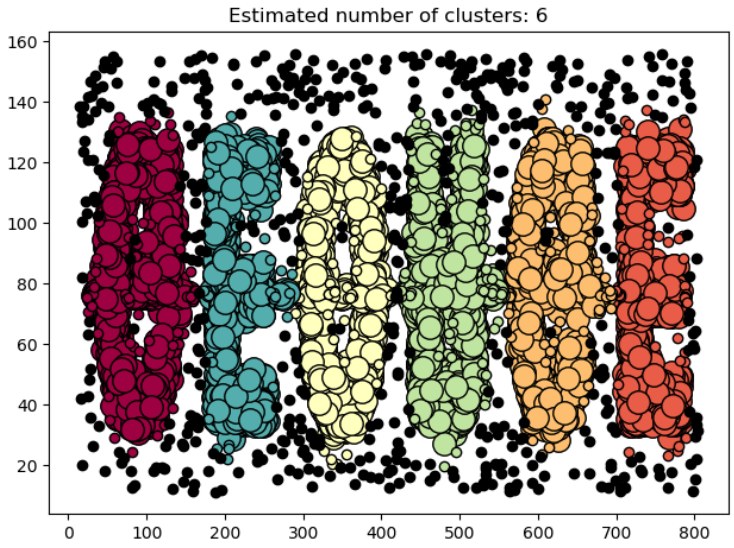
Epsilon : 10
Min Samples : 18
Estimated number of clusters : 6
Estimated number of noise points : 630
Silhouette Coefficient : 0.246



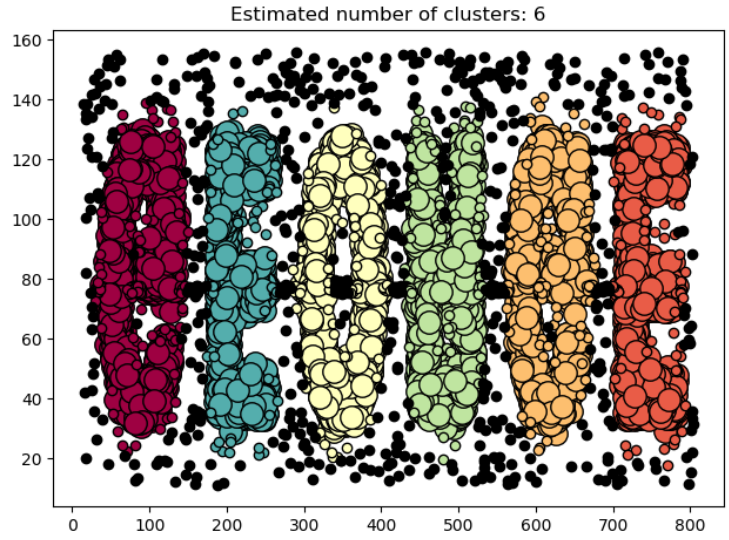
Epsilon : 12
Min Samples : 22
Estimated number of clusters : 6
Estimated number of noise points : 543
Silhouette Coefficient : 0.254

Pour ces deux tests, nous obtenons bien le nombre de 6 clusters, identifiables à vue d'œil. Le coefficient de silhouette reste proche de 0.25, cela étant les formes de nos clusters sont tout à fait satisfaisantes. Le bruit de sinusoïde est (légèrement) davantage ignoré sur le premier schéma (e = 10, ms = 18). On considèrera le bruit plus faible (543), cela étant il n'est pas nécessaire de chercher à le diminuer. Nos clusters doivent rester cohérents.

Dataset 2



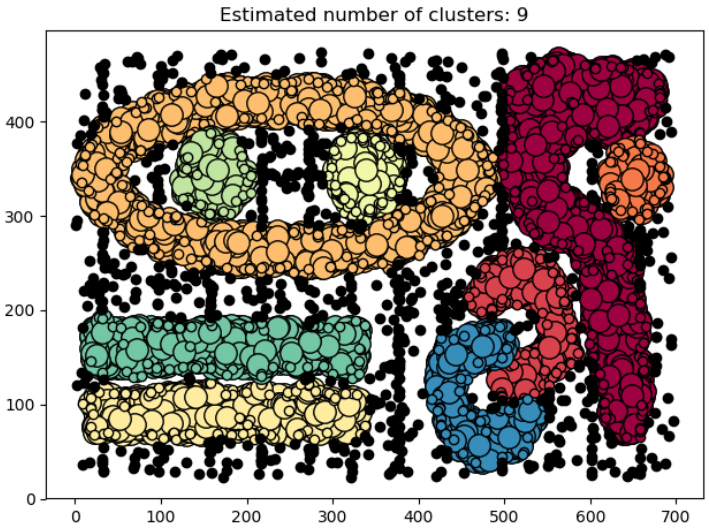
Epsilon : 9.8
Min Samples : 18
Estimated number of clusters : 6
Estimated number of noise points : 566
Silhouette Coefficient : 0.490



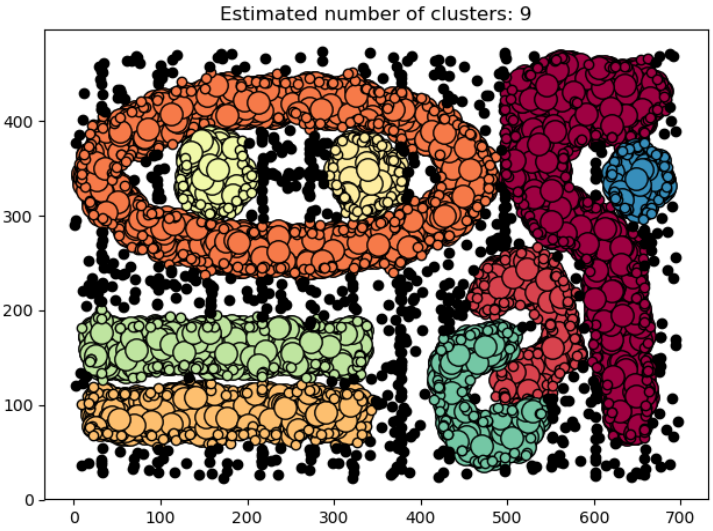
Epsilon : 12
Min Samples : 38
Estimated number of clusters : 6
Estimated number of noise points : 630
Silhouette Coefficient : 0.486

Sur ce second dataset, nous avons un meilleur coefficient silhouette avec des paramètres plus faibles. Sur le deuxième test, on peut constater le bruit au milieu du graphe (barre horizontale), qui est pourtant assimilé aux clusters sur le premier test. Il semble plus logique d'ignorer ce bruit donc nous considérons les seconds paramètres plus pertinents.

Dataset 3



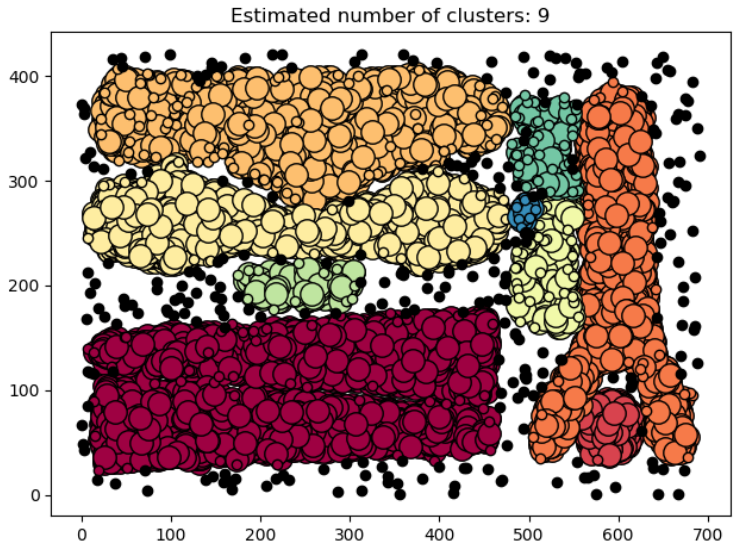
Epsilon : 10
Min Samples : 13
Estimated number of clusters : 9
Estimated number of noise points : 774
Silhouette Coefficient : -0.068



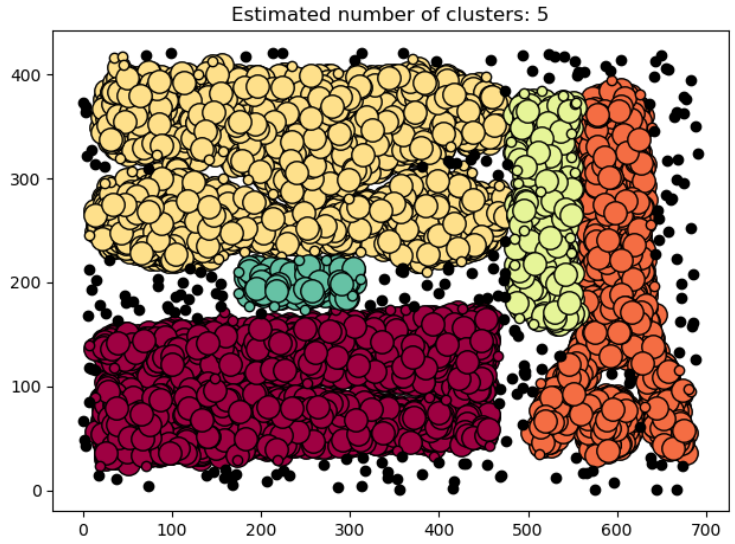
Epsilon : 14
Min Samples : 26
Estimated number of clusters : 9
Estimated number of noise points : 660
Silhouette Coefficient : -0.062

Très peu de différences sont visibles sur ces deux tests du troisième dataset (hormis la couleur des clusters …). Le bruit symbolisé par les lignes verticales parallèles est peu pris en compte (davantage sur le test 2). Notre clustering reste cohérent malgré une différence d'une centaine de points de bruits, qui n'ont pas l'air d'affecter grandement la qualité des clusters.

Dataset 4



Epsilon : 10.1
Min Samples : 7
Estimated number of clusters : 9
Estimated number of noise points : 277
Silhouette Coefficient : -0.078



Epsilon : 12
Min Samples : 6
Estimated number of clusters : 5
Estimated number of noise points : 215
Silhouette Coefficient : -0.007

Il nous a été difficile d'identifier les bons paramètres pour ce dernier clustering sur le quatrième dataset. Certains clusters sont tellement proches qu'ils peuvent se retrouver groupés pour un epsilon trop grand. Nous pensons tout de même que notre premier clustering est plus pertinent car il distingue davantage de clusters que le second (plus cohérents à vue d'œil). Toutefois nous aurions aimé trouver un paramétrage pour distinguer les deux pavés mauves en bas à gauche.

I.3/ Remarques

Suivant le dataset, les paramètres optimaux sont différents. On ne peut pas donc trouver un modèle systématique. C'est normal car nos datasets ont un nombre de clusters différent avec des formes différentes.

II – Algorithme SNN

II.1/ Présentation

Pour cette partie, l'objectif est d'appliquer une méthode de clusterisation utilisant le graphe SNN, Shared Neirest Neighbors. Pour cela, il y a différentes variantes :

- Algorithme SNN-DBSCAN
- Algorithme Jarvis-Patrick

Ces 2 méthodes ont les 3 premières étapes communes :

- 1/ Calculer la matrice de similarité
- 2/ Filtrer la matrice pour ne conserver que les k voisins les plus similaires
- 3/ Construire le graphe SNN

Ensuite, dans la 1ere méthode, on applique DBSCAN sur notre graphe SNN. Dans la 2eme méthode, on applique un seuil de similarité et on cherche les composantes connexes pour obtenir des clusters.

NOTE : Nous avons passé trop de temps, en dehors du TP, pour essayer de finir au moins l'une des méthodes, mais nous n'avons pas réussi. Nous ne pourrons donc pas comparer les performances de cette méthode avec DBSCAN réalisé en 1e partie. Ci-après seront donc exposé les difficultés que nous avons rencontrées, étape par étape.

II.2/ Matrice de similarité

Au début du TP de la partie 2, Nous avons travaillé chacun de notre côté. Nous avons donc chacun calculé la matrice de similarité de façon différente.

Version Anaïs :

```
def similarity_func(u, v):
    return 1/(1+euclidean(u,v))

def similarity_Matrix(data):
    numPoints = len(data)
    matrix = np.zeros(shape=(numPoints, numPoints))
    for i in range(len(matrix)):
        for j in range(len(matrix)):
            sim = similarity_func(data[i], data[j])
            matrix[i,j] = sim
    return matrix
```

Version Adrian :

```
def similarity_mat(data):
    X_dist = np.array(squareform(pdist(data, metric='euclidean')))
    X_sim = 1 / (1 + X_dist)
    return X_sim
```

Les 2 versions sont opérationnelles, mais la version d'Adrian est beaucoup plus rapide et le code est bien plus lisible. Même en ne travaillant que sur la moitié de la matrice (car elle est symétrique), la méthode d'Anaïs reste lente sur notre jeu de données. Nous retenons donc la version d'Adrian.

II.3/ Filtrage des K-voisins les plus similaires

Pour cette méthode, nous avons pris le parti d'utiliser *kneighbors_graph* dans le package Sklearn.neighbors qui nous génère la matrice de connectivité des K plus proches voisins.

Premièrement, cette méthode nous permet de récupérer soit une liste de tuples, soit une matrice. Pour cela, il suffit de rajouter *toarray()*.

Deuxièmement, il y a 2 modes possibles : connectivity et distance. Dans le 1er cas, on récupère les indices des voisins les plus proches (ou une matrice de 0 et de 1). Dans le 2ème cas, on récupère une matrice avec la distance. Par défaut, c'est la distance euclidienne qui est appliquée.

Comme données, on peut soit lui donner les données brutes (des points), soit lui donner la matrice de similarité.

Dans la philosophie de ce TP, nous lui donnons notre matrice de similarité en entrée. On récupère la matrice de connectivité. Et on la multiplie avec notre matrice de similarité. On obtient donc une matrice qui représente le graphe avec les K voisins. On a les voisins « éloignés » qui sont mis à 0.

```
def filtrage_simMatrix(sim, k):
    # recuperer les k plus proches voisins ( matrice de 0 et de 1)
    A = kneighbors_graph(sim, k, mode='connectivity', include_self= False).toarray()
    # On actualise la matrice de similarité en mettant à 0 les voisins qui ne sont pas proches
    SimVoisin= sim*A
    return SimVoisin
```

II.4/ SNN et les problèmes…

C'est à partir de cette étape que nos méninges commencent à s'embrouiller. Mais avant cela, nous construisons notre matrice SNN. Pour cela, on s'est appuyé sur la définition du graphe SNN dans l'article en anglais proposé avec le sujet.

Voici l'algo de matrice SNN pour le cas simple. On considère 2 sommets, p et q, qui sont des proches voisins. On ajoute +1 à chaque fois que 2 sommets p et q ont un voisin commun.

```
def SNNmatrix(nnmatrix, k):
    snnmatrix= np.zeros((len(nnmatrix), len(nnmatrix)))
    for i in range(len(nnmatrix)):
        for j in range(len(nnmatrix)):
            # on ne considère pas le sommet lui-même
            # on considère que j est dans les plus proches voisins de i
            # on considère que i est dans les plus proches voisins de j
            if (i!=j) & (nnmatrix[i][j]!=0)& (nnmatrix[j][i]!=0):
                for l in range(len(nnmatrix)):
                    m= nnmatrix[i][l]
                    n= nnmatrix[l][j]
                    # on regarde si l est lié à i et à j et que l est différent de i et j
                    if (l!=i) & (l!=j)&(m!=0) & (n!=0):
                        snnmatrix[i][j]=snnmatrix[i][j]+1
                        # Version pondérée :
                        #snnmatrix[i][j]=snnmatrix[i][j]+(k+1-m)*(k+1-n)
    return snnmatrix
```

Notre version simple est opérationnelle.

Nous avons donc voulu mettre en place la version pondérée. Mais là un problème de compréhension subsiste avec ce que dit l'article. Au départ, on a eu l'impression que les m et n auxquels l'article fait référence concerne la distance entre les sommets et le sommet commun. Mais en fait, l'article parle de position et donc nous pensons que ce ne sont pas des distances, mais plutôt des rangs.

Par exemple, si on prend les sommets 0 et 2 qui ont le sommet 1 en commun. Si l'on considère m et n comme les positions de 1 dans la liste des K voisin, 1 pourrait être le 3ème voisins de 0 mais le 6ème voisin de 2. M serait donc égal à 3 et n à 6.

Cela prend son sens mais nous ne savons pas comment l'implémenter dans notre situation. Nous sommes donc restés sur la version simple.

De là découlent plusieurs incompréhensions :

- Comment SNN peut garder une notion de distance si l'on considère les positions dans la liste des voisins et non plus la distance ?
- Comment peut-on appliquer DBSCAN sur des données qui n'ont plus cette notion de distance ?
- Comment peut s'appliquer DBSCAN sur une matrice sparsifiée ? Et comment peut-on afficher cette solution ? (Questions techniques)

Il y a une autre partie qui reste floue dans l'implémentation de SNN : comment évaluer 2 sommets qui ont un lien direct (qui sont voisins) mais qui ne partagent pas de sommet commun. Comment aussi considérer 2 sommets qui ont un lien direct et qui partage des sommets voisins ?

Nous avons aussi discuté avec le binôme Duc Hau Nguyen – Mohamed El Filali. Ils ont pris le parti de rendre symétrique la matrice filtrée avec les K voisins et donc la matrice SNN (en imposant qu'un arc sortant soit aussi un arc entrant et qu'un arc entrant soit aussi un arc sortant). Nous ne comprenons pas ce qui motive ces choix. Dans leur application de l'algo Jarvis-Patrick, nous n'arrivons pas à savoir comment il est possible de comparer cette méthode avec DBSCAN vue dans la partie 1.