Université des Sciences et de la Technologie
Houari Boumediene

Rapport de TP Fouille de données:

Implémentation de l'algorithme K-means et implémentation de l'approche Self organizing map pour automatiser le nombre de clusters

Binôme

OMARI Hamza	181831030062	Groupe 02
HAMZAOUI Thameur	181831030061	Groupe 02

Mai 2023

Introduction

L'algorithme K-means est une technique d'apprentissage non supervisé largement utilisée dans l'analyse de données et la segmentation de données. L'objectif principal de l'algorithme K-means est de regrouper un ensemble de données non étiquetées en K groupes ou clusters distincts en fonction de la similarité des caractéristiques entre les données. Dans ce rapport nous allons expliquer comment on a implémenter cet Algorithm vu en cours avec le langage Python

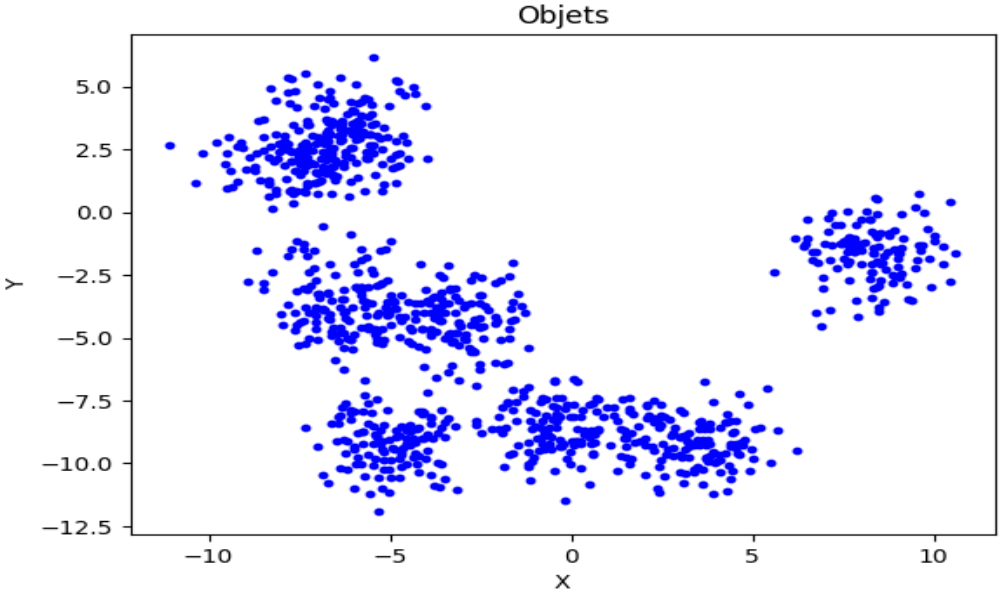
1.Implémentation de l'algorithme K-means

1.1 Les fonctions nécessaires pour l'implémentation

Tout d'abord nous allons commencer par générer des données aléatoires mais aussi qui ne sont pas dispersées n'importe comment. Pour cela nous allons utiliser la bibliothèque **sklearn** pour utiliser la fonction make_blobs. Cette fonction nous permettra de générer des données aléatoires dans un nombre de régions précisé en paramètre. Cette étape a pour but pour vérifier à la fin l'exactitude de notre implémentation

```
# Générer des données pour k centres en utilisant make_blobs
X, _ = make_blobs(n_samples=1000, centers=k, n_features=2,
                  shuffle=True, random_state=41)
```

Le résultat de cette étape nous donne l'ensemble d'objets suivant :



Par la suite nous allons initialiser les K centre de clusters arbitrairement pour cela nous utilisons une fonction qui prend les objets (data) et K pour donner des valeurs Ci aléatoire.

```
def initialiser_centroids(k, data):
    n_dims = data.shape[1]
    centroid_min, centroid_max = np.min(data, axis=0), np.max(data, axis=0)
    centroids = np.random.uniform(centroid_min, centroid_max, size=(k, n_dims))
    centroids = pd.DataFrame(centroids, columns=data.columns)
    return centroids
```

Par la suite pour calculer les distances d'un objet xi avec le centre d'un cluster Ci nous devons introduire une mesure de distance, nous utilisons donc la distance euclidienne

```
def calculer_distance(x1, x2):

    distance = np.sum(np.square(x1-x2))
    return distance
```

Enfin la dernière fonction qu'on devrait construire pour que nous puissions mettre en oeuvre Kmeans, c'est de pouvoir désigner pour chaque objet quel est le cluster le plus proche, tout en retournons aussi la distance minimale, sur-ce, la fonction suivante :

```
def assigner_centroid(data, centroids):
    n_objets = data.shape[0]
    centroid_assign = []
    centroid_distances = []
    k = centroids.shape[0]
    centroids_arr = centroids.iloc[:, :2].to_numpy()

    for objet in range(n_objets):
        # Cauculer les distances avec les centroids
        distances = np.sum(
            np.square(centroids_arr - data.iloc[objet, :2].to_numpy()), axis=1)
        # Calculer le centroid le plus proche et ca distance
        centroid_plus_proche = np.argmin(distances)
        centroid_distance = np.min(distances)

        # Assigner les les meilleurs centroids aux objets
        centroid_assign.append(centroid_plus_proche)
        centroid_distances.append(centroid_distance)

    # retourner les deux listes l'assignation avec index, et les distances
    return (centroid_assign, centroid_distances)
```

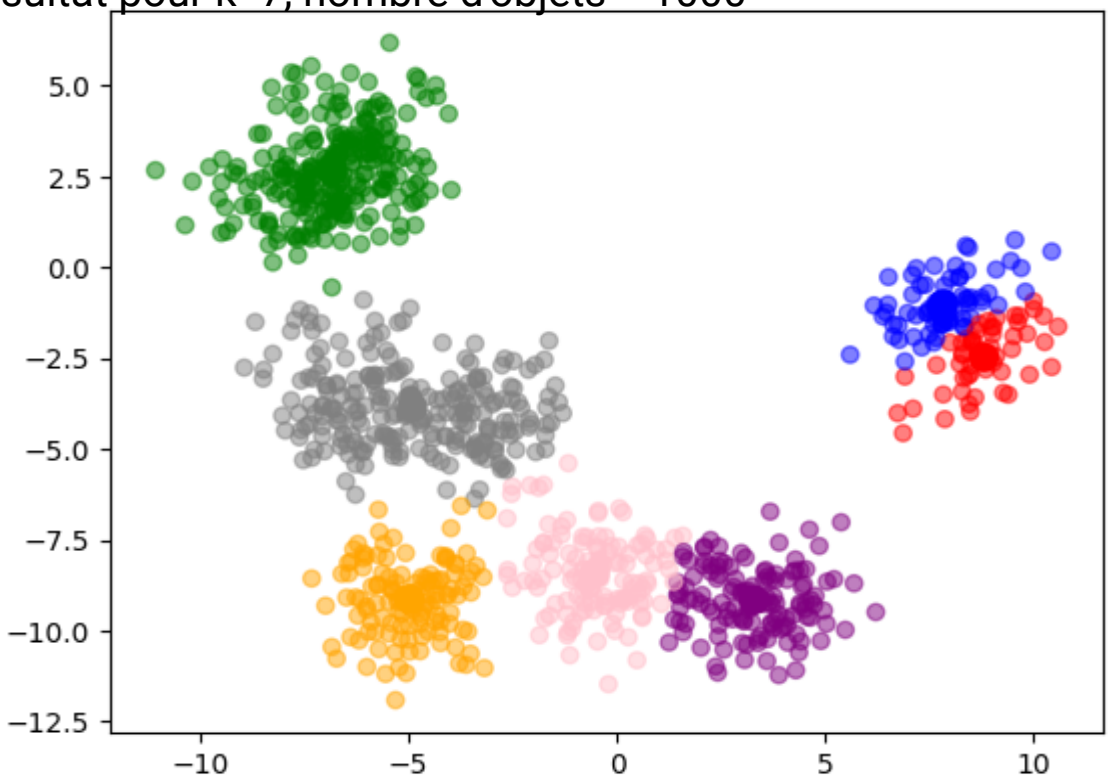
1.2 Implémentation de de la fonction principale kmeans

Maintenant, avec les fonctions précédentes nous pouvons implémenter k-means comme suit:

- Sélectionner K clusters initiaux contenant des objets choisis arbitrairement avec leur centres
- Utiliser la fonction assigner_centroides() pour assigner pour chaque objet le cluster le plus proche.
- calculer les nouveaux centres du custer
- Répéter les étapes sauf la première jusqu'à convergence

```
# Initialiser les centroids et les distances
centroids = initialiser_centroids(k, data)
distance = []
compr = True        # pour tester la boucle
i = 0
while (compr):
    # calculer les centroids et la distance , et l'assignation
    data['centroid'], iter_distance = assigner_centroid(data, centroids)
    #Ajouter la somme des distances renvoyé par assigner_centroid pour voir si les centroids sont fix ou pas
    distance.append(sum(iter_distance))
    # Recalculer les centres des clusters et MAJ leur nouveaux coordonnes
    centroids = data.groupby('centroid').agg('mean').reset_index(drop=True)

    # Vérifier si on a changé ou pas
    if (len(distance) < 2):
        # Ici pour skiper la premiere iteratin car on ne peut pas comparer
        compr = True
    else:
        if (round(distance[i], 3) != round(distance[i-1], 3)):
            compr = True
        else:
            # Cas de convergence
            compr = False
    i = i + 1
```

Résultat pour k=7, nombre d'objets = 1000



2. Implémentation de la méthode elbow

Nous utilisons l'approche "elbow curve" ou bien "la méthode du coude en français". c'est une technique graphique utilisée pour déterminer le nombre optimal de clusters à utiliser, donc le K , suivant ces étapes:

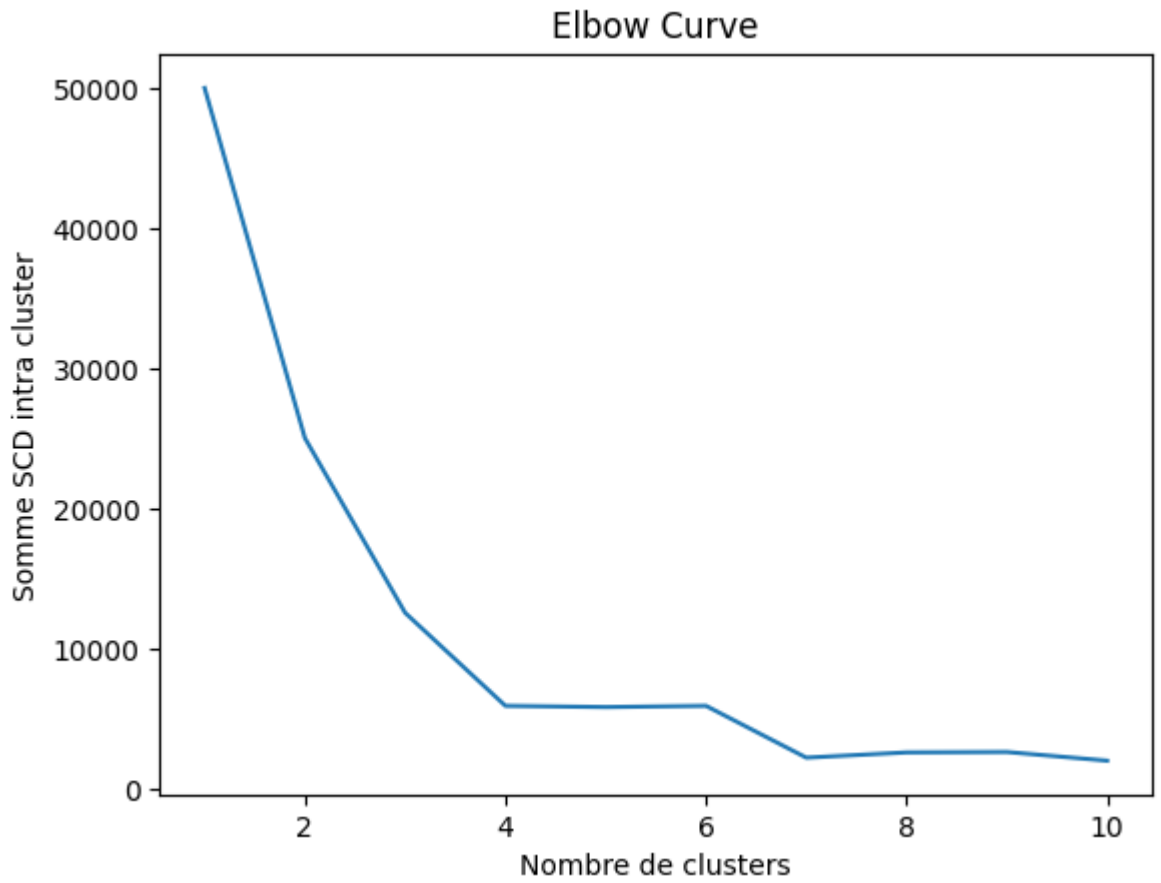
- Appliquer la méthode de clustering pour différents nombres de k (par exemple de 1 à 10)
- Calculer la somme des carrés des distances intra-cluster pour chaque solution de clustering.
- Tracer un graphique du nombre de clusters en Axe X et de la somme des carrés des distances intra-cluster en axe Y .
- voir le point sur le graphe où l'ajout d'un nouveau cluster n'améliore plus significativement la qualité du clustering. Ce point est appelé le "coude" de la courbe

Le nombre optimal de clusters est alors choisi comme étant le nombre de clusters correspondant au coude de la courbe.

nous implémentant cette démarche comme suit

```python
def kmeans_elbow(data):

    # Initialiser les variables
    distances = []
    max_k = min(len(data), 10)
    # processus iteratif sur les valeurs de K
    for k in range(1, max_k + 1):

        data['centroid'], iter_distance, centroids = kmeans(data, k)
        distance = sum(iter_distance)

        # Sauvegarder les distances
        distances.append(distance)

    # Plot elbow curve
    plt.plot(range(1, max_k + 1), distances)
    plt.title('Elbow Curve')
    plt.xlabel('Nombre de clusters')
    plt.ylabel('Somme SCD intra cluster')
    plt.show()
```

le résultat de cette méthode donne comme graph:



Ce qui est un bon résultat car le point elbow correspond au nombre 6 de clusters.

Méthode en utilisant la carte SOM (Self organizing map)

La carte SOM est constituée d'une grille de neurones qui sont organisés en deux dimensions (généralement en une grille rectangulaire ou hexagonale). Chaque neurone est associé à un vecteur de poids qui représente une position dans l'espace de données multidimensionnelles. Ces vecteurs de poids sont initialement choisis de manière aléatoire, puis ajustés à mesure que l'algorithme est entraîné.

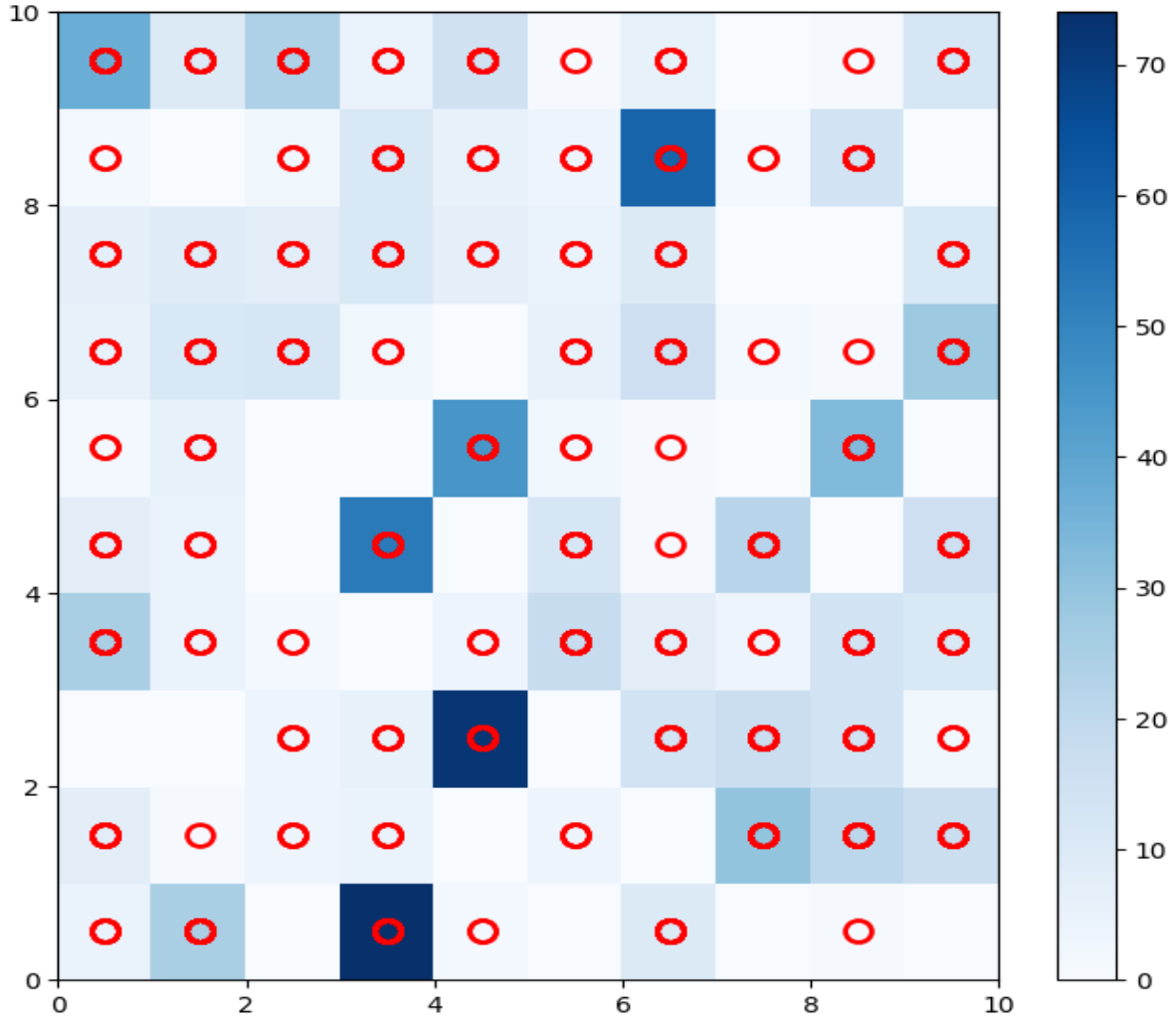
Lors de l'entraînement de la carte SOM, les données d'entrée sont présentées à l'algorithme, qui calcule la distance entre chaque vecteur

de poids et les données d'entrée. Le neurone dont le vecteur de poids est le plus proche des données d'entrée est appelé le "neurone vainqueur". Ce neurone et ses neurones voisins sur la grille sont mis à jour de manière à se rapprocher des données d'entrée. Les neurones éloignés des données d'entrée subissent une mise à jour plus faible.

Au fur et à mesure que l'algorithme est entraîné, les neurones de la carte SOM finissent par se spécialiser dans la représentation de différentes zones de l'espace de données d'origine. Les régions de la carte SOM qui sont proches les unes des autres correspondent à des zones similaires dans l'espace de données d'origine.

La carte SOM est souvent utilisée pour la visualisation et la compression de données, ainsi que pour la détection de motifs dans les données. Elle peut également être utilisée comme étape préliminaire pour d'autres algorithmes d'apprentissage automatique, tels que la classification ou la prédiction.

Après avoir implémenter cette approche nous avons obtenu le résultat suivant:



Les neurones sur la carte SOM avec les couleurs les plus foncées indiquent les régions où les données sont les plus denses, tandis que les couleurs plus claires indiquent des régions avec des densités de données plus faibles. Par conséquent, pour déterminer le nombre optimal de clusters K, vous pouvez utiliser la carte SOM pour identifier les régions où les données sont regroupées de manière significative. On peut dire que le nombre K idéal dans notre cas est égale à **6**.
C'est un résultat très très bon, parce que c'est le même nombre que l'oeil humaine vas interpréter dans notre exemple suivant