Université de Bourgogne

TP Big Data
Implémentation de K-Means

Auteurs :
Clément GHYS
Benjamin MILHET

Professeur :
M. KIRGIZOV

clement_ghys@etu.u-bourgogne.fr
benjamin_milhet@etu.u-bourgogne.fr

2023 - 2024

Table des matières

Introduction

Nous avons décider de réaliser le premier projet de codage qui consiste à implémenter l'algorithme de clustering K-means. Le clustering est un algorithme de d'apprentissage non-supervisée, cela signifie que les données qu'il reçoit ne sont pas étiquetée, il ne connaît pas l'origine des données, mais lors des phases d'apprentissage, il va découvrir des patternes et des caractéristiques communes entre les données et se basé sur es informations pour réaliser ses prédictions. L'algorithme K-means est l'algorithme de clustering le plus populaire et permet de divise les données en K groupes en minimisant la distance entre les points de données et le centre de leur cluster. L'objectif de ce projet est d'implémenter l'algorithme de clustering k-means et certaines de ses variations comme k-means++ et la version mini-batch, le tout en faisant varier la dimensionnalité et la distribution des points.

Le code de notre projet est disponible sur notre dépôt : Github

Création et visualisation d'un ensemble de données artificielles

La première étape du projet est de générer 5 clusters gaussiens qui contiennent 500 points bidimensionnels au total en utilisant la fonction sklearn.dataset.make_blobs.

Listing 2.1 – Script pour générer les 5 clusters

```
nb_of_clusters = 5 # Nombre de clusters
nb_of_samples = 500 # Nombre de points
dataset, _ = datasets.make_blobs(n_samples=nb_of_samples, centers=nb_of_clusters, n_features=2) #
    Generation de donnees aleatoires en 2D
```

Nous avons ensuite créer une fonction appeler visualize_clusters qui nous permet d'afficher dans un graphique nos différents clusters. Pour cela, nous avons utiliser la bibliothèque Python Matplotlib permettant de réaliser des graphiques. Cette bibliothèque nous permet aussi de coloriser chacun de nos clusters avec une couleur différentes afin de bien les différenciés.

Listing 2.2 – Fonction pour afficher les différents clusters

```
def visualize_clustersV1(dataset, centers=None, title=None):

    colors = list(mcolors.CSS4_COLORS.keys()) # Liste des couleurs CSS4_COLORS
    plt.scatter(dataset[:, 0], dataset[:, 1], c=_, cmap=mcolors.ListedColormap(colors)) #
        Affichage des donnees avec une couleur par cluster

    if centers is not None: # Si les centres des clusters sont fournis, on les affiche
       plt.scatter(centers[:, 0], centers[:, 1], c='red') # Affichage des centres des clusters
    plt.show() # Affichage du graphique


visualize_clustersV1(dataset) # Affichage des donnees sans les clusters
```

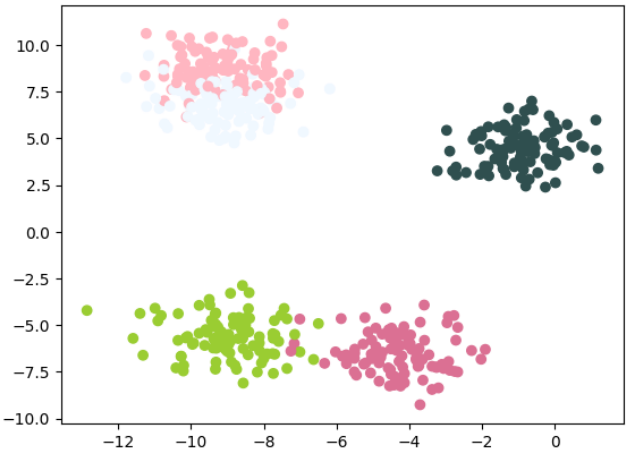


FIGURE 2.1 – Visualisation des 5 clusters pour un total de 500 points dans une dimension de 2

Le code ci-dessous est une légère modification du code précèdent permettant d'afficher les clusters dans une dimension de 3.

Listing 2.3 – Fonction pour afficher les différents clusters

```
def visualize_clustersV1_3D(dataset, centers=None, title=None):
    fig = plt.figure() # Creation d'une figure
    ax = fig.add_subplot(111, projection='3d') # Ajout d'un subplot 3D

    colors = list(mcolors.CSS4_COLORS.keys()) # Liste des couleurs CSS4_COLORS

    ax.scatter(dataset[:, 0], dataset[:, 1], dataset[:, 2], c=_,
        cmap=mcolors.ListedColormap(colors)) # Affichage des donnees avec une couleur par cluster

    plt.show() # Affichage du graphique
visualize_clustersV1_3D(dataset)
```

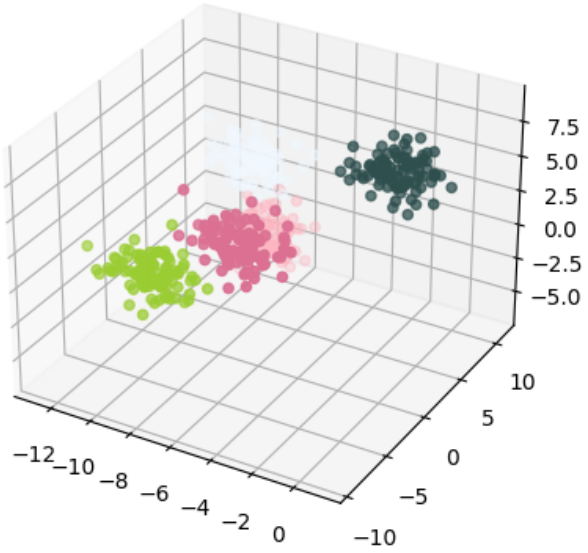


FIGURE 2.2 – Visualisation des 5 clusters pour un total de 500 points dans une dimension de 3

Le code ci-dessous est un regroupement des deux premières fonctions visualize et permet de s'adapter automatiquement pour afficher en 2 ou 3 dimensions.

Listing 2.4 – Fonction pour afficher les différents clusters en 2D ou en 3D

```
def visualize_clusters(dataset, centers=None, title=None):
    """
    Cette fonction permet de visualiser les clusters et les centres des clusters en 2D ou 3D.

    Keyword arguments:
    dataset -- ensemble de donnees de points
    centers -- centres des clusters
    title -- titre du graphique

    Return: None
    """

    colors = list(mcolors.CSS4_COLORS.keys()) # Liste des couleurs CSS4_COLORS
    if dataset.shape[1] == 2: # Si les donnees sont en 2D
        plt.scatter(dataset[:, 0], dataset[:, 1], c=_, cmap=mcolors.ListedColormap(colors)) #
            Affichage des donnees avec une couleur par cluster
    elif dataset.shape[1] == 3: # Si les donnees sont en 3D
        fig = plt.figure() # Creation d'une figure
        ax = fig.add_subplot(111, projection='3d') # Ajout d'un subplot 3D
        ax.scatter(dataset[:, 0], dataset[:, 1], dataset[:, 2], c=_,
            cmap=mcolors.ListedColormap(colors)) # Affichage des donnees avec une couleur par
            cluster
    else: # Si les donnees sont en plus de 3D
        raise ValueError("Visualization not implemented for more than 3 dimensions") # Erreur si
            les donnees sont en plus de 3D

    if centers is not None: # Si les centres des clusters sont fournis, on les affiche
        if centers.shape[1] == 2: # Si les centres des clusters sont en 2D
            plt.scatter(centers[:, 0], centers[:, 1], c='red', marker='x', s=100) # Affichage des
                centres des clusters
        elif centers.shape[1] == 3: # Si les centres des clusters sont en 3D
            ax.scatter(centers[:, 0], centers[:, 1], centers[:, 2], c='red', marker='x', s=100) #
                Affichage des centres des clusters
        else: # Si les centres des clusters sont en plus de 3D
            raise ValueError("Visualization not implemented for more than 3 dimensions") # Erreur
                si les centres des clusters sont en plus de 3D

    plt.show() # Affichage du graphique
```

Implémentation de l'algorithme de base de Lloyd

Il nous faut maintenant implémenter l'algorithme de base de Lloyd, étant repris pour l'algorithme de clustering K-means et adapter pour du clustering de données multidimensionnelles.

3.1 Initialisation des centres

La première étape de l'implémentation de l'algorithme est de créer une fonction qui initialise aléatoirement les centres de chacun de nos clusters. Pour cela, on réalise une boucle du nombre de nos cluster et on prend un point aléatoire sur l'ensemble de nos cluster. Le principale problème de cette fonction est qu'a l'initialisation certain cluster n'auront pas de centre.

Listing 3.1 – Fonction pour initialiser le centre des différents clusters

```
def initialize_centers(data, k):
    """
    Cette fonction permet d'initialiser les centres des clusters

    Keyword arguments:
    data -- Dataset a utiliser pour initialiser les centres
    k -- Nombre de clusters
    Return: liste des centres des clusters
    """

    indices = np.random.choice(len(data), k, replace=False) # Selectionner k indices aleatoires
    centers = data[indices] # Recuperer les points correspondants aux indices
    return centers # Retourner la liste des centres des clusters
```

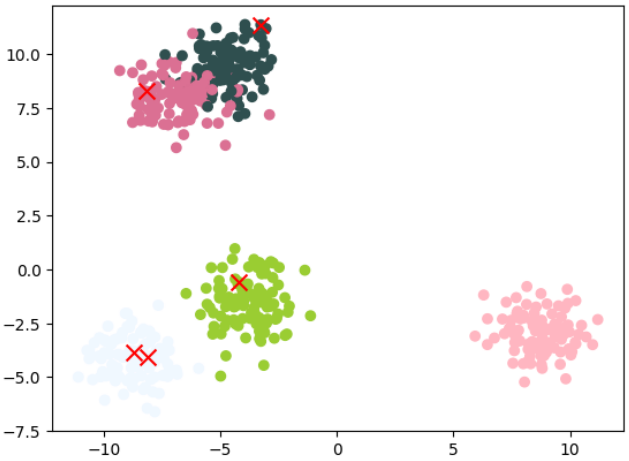


FIGURE 3.1 – Visualisation des centres pour 5 clusters en 2 dimensions

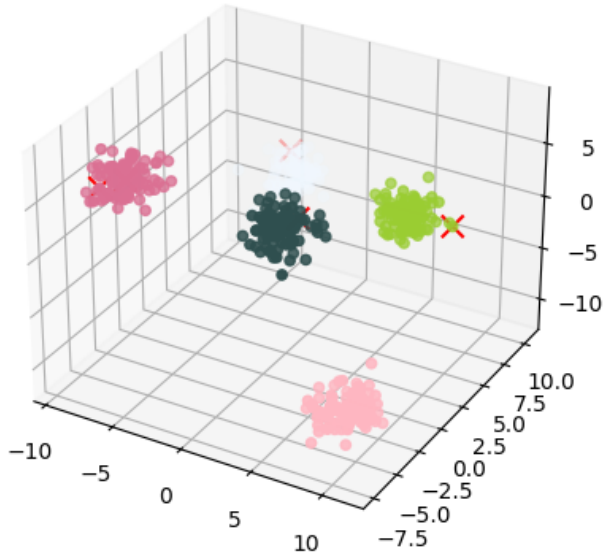


FIGURE 3.2 – Visualisation des centres pour 5 clusters en 3 dimensions

3.2 Calcul de la distance euclidienne

La seconde fonction nécessaire pour notre algorithme est la fonction sse_distance qui permet de calculer la distance euclidienne carrée entre deux points. Cette fonction nous sera utile pour calculer la distance entre les points d'un cluster et les différents centres.

Listing 3.2 – Fonction pour calculer la distance euclidienne carrée entre deux points

```
def sse_distance(a, b):
    """
    Cette fonction calcule la distance entre deux points a et b
    en utilisant la somme des erreurs au carre (SSE)
    """

    return np.sum((np.array(a) - np.array(b))**2) # On retourne la somme des erreurs au carre
```

3.3 Recherche du centre le plus proche

L'objectif de cette fonction est de déterminé le meilleur centre par rapport à une liste de centre pour un point donné. Cette fonction reprend la fonction précédemment créer permettant de calculer la distance euclidienne entre 2 points et parcours l'ensemble d'une liste de centre pour trouver un centre optimal.

Listing 3.3 – Fonction pour rechercher le centre le plus proche d'un point

```
def find_closest_center(centers, data):
    """
    Cette fonction trouve le centre optimal dans une liste de centres pour un point donne

    Keyword arguments:
    centers -- liste des centres
    data -- point dont on cherche le centre le plus proche
    Return: l'index du centre le plus proche
    """

    distances = [] # Liste des distances entre le point et les centres
    for center in centers: # Pour chaque centre
        distances.append(sse_distance(center, data)) # On calcule la distance entre le point et le
            centre
    return np.argmin(distances) # On retourne l'index du centre le plus proche
```

3.4 Création des clusters

La prochaine étape est l'implémentation de la fonction compute_clusters qui permet d'attribuer chaque point de données à un cluster en fonction de la proximité avec les centres de ces clusters. Cette fonction reprend les deux fonctions précédemment créer nos différents clusters par rapport à nos différents centres.

Listing 3.4 – Fonction pour créer nos différents clusters

```
def compute_clusters(centers, data):
    """
    Cette fonction calcule les clusters pour chaque point de la liste de nos donnees

    Keyword arguments:
    centers -- liste des centres
    data -- liste des points
    Return: liste des clusters
    """
    clusters = [] # Liste des clusters
    for point in data: # Pour chaque point
        clusters.append(find_closest_center(centers, point)) # On trouve le centre le plus proche
            et on l'ajoute a la liste des clusters
    return clusters # On retourne la liste des clusters
```

3.5 Calcul de la SSE

La dernière fonction nécessaire avant l'implémentation de notre fonction K-means. Cette fonction calcule la somme des carrés des distances euclidiennes entre chaque point de données et son centre de cluster. De plus, elle réutilise la fonction compute_clusters que nous avons créer précédemment pour attribuer chaque point à son cluster le plus proche. Ensuite, nous calculons la distance euclidienne de chaque point à son centre de cluster correspondant et sommez les carrés de ces distances. Cette fonction nous permet de calculer la SSE, une mesure courante de la performance d'un modèle de clustering qui vise à minimiser l'erreur de distance intra-cluster.

Listing 3.5 – Fonction pour calculer la SSE

```
def sse_error(centers, data):
    """
    Cette fonction calcule l'erreur SSE

    Keyword arguments:
    centers -- liste des centres
    data -- liste des points
    Return: l'erreur SSE
    """

    clusters = compute_clusters(centers, data) # On calcule les clusters
    error = 0 # Erreur SSE initiale a 0
    for i in range(len(clusters)): # Pour chaque cluster
        error += sse_distance(data[i], centers[clusters[i]]) # On ajoute la distance entre le
            point et le centre du cluster
    return error # On retourne l'erreur SSE
```

3.6 La fonction K-means

Grâce aux 4 fonctions que nous avons crées, nous pouvons maintenant créer la fonction principale de notre algorithme, la fonction K-means. L'algorithme K-means commence par initialiser les clusters et leurs centres. Ensuite, nous améliorons la positions des centres de nos clusters et nous vérifions à l'aide du calcul de notre SSE si le changement de centre est une amélioration ou non. Une fois une stagnation de la position de nos centres, on peut en déduir eune convergence vers la solution optimale pour cette version de notre algorithme.

Listing 3.6 – Fonction K-means

```
def kmeans(data, k, num_it, centers=centers):
    """
    Cette fonction permet d'effectuer l'algorithme K-means

    Keyword arguments:
    data -- Dataset a utiliser pour initialiser les centres
    k -- Nombre de clusters
    num_it -- Nombre d'iterations
    Return: historique des centres et historique de SSE
    """
    centers_history = [] # Historique des centres des clusters
    sse_history = [] # Historique de l'erreur SSE
    tolerance=1 # Tolerance pour la convergence
    centers = initialize_centers(data, k) # Initialisation des centres
    #centers = initialize_centers2(data, k) # pour kmeans ++

    for i in range(num_it): # Pour chaque iteration
        centers_history.append(np.array(centers)) # Ajout des centres a l'historique
        clusters = compute_clusters(centers, data) # Attribution des clusters
        sse = sse_error(centers, data) # Calcul du SSE
        sse_history.append(sse) # Ajout du SSE a l'historique

        # Mise a jour des centres
        new_centers = [] # Nouveaux centres
        for cluster_id in range(k): # Pour chaque cluster
            cluster_points = np.array([data[j] for j in range(len(data)) if clusters[j] ==
                cluster_id]) # Recuperation des points du cluster
            new_centers.append(np.mean(cluster_points, axis=0)) # Calcul du nouveau centre

        if i > 0 and abs(sse_history[i-1] - sse) < tolerance: # Si les centres n'ont pas change,
             on a atteint la convergence
            print(f"Convergence atteinte apres {i} iterations.") # Affichage du nombre d'iterations
            break

        centers = new_centers # Mise a jour des centres
        print(f"Iteration {i+1}, SSE = {sse}") # Affichage de l'iteration et du SSE
        visualize_clustersDim(data, np.array(centers)) # Affichage des clusters

    # Derniere mise a jour des centres et SSE
    clusters = compute_clusters(centers, data)
    sse = sse_error(centers, data)
    centers_history.append(np.array(centers))
    sse_history.append(sse)

    # Affichage de l'evolution de l'erreur en fonction du nombre d'iterations
    plt.plot(range(i+1), sse_history[:-1], marker='o')
    plt.xlabel('Iterations')
    plt.ylabel('SSE')
    plt.title('Evolution de l\'erreur en fonction du nombre d\'iterations')
    plt.show()

    # Retourne l'historique des centres et l'historique de SSE
    history = {'centers_history': centers_history, 'sse_history': sse_history}
    return history

h = kmeans(dataset,nb_of_clusters, 100) # Appel de la fonction kmeans
```

3.7 Évolution de la SSE

On remarque une diminution rapide de la SSE, puis une stagnation vers une solution optimale pour une première version de notre algorithme K-means.

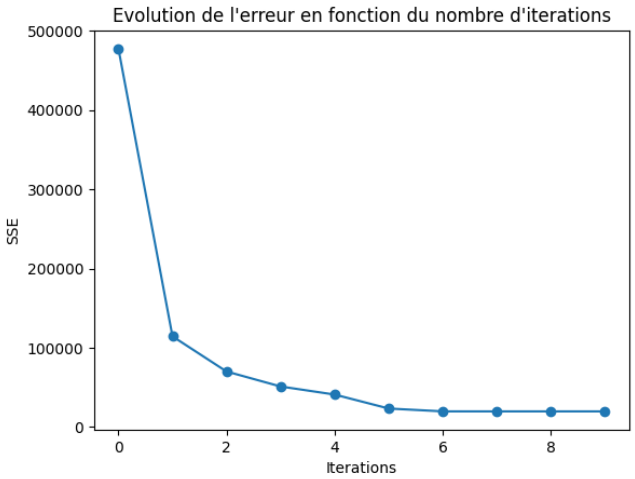


FIGURE 3.3 – Évolution de la SSE sur 10 itérations

On remarque très clairement que notre algorithme converge vers une solution avant la fin des 10 itérations. Pour optimiser notre fonction K-means, on a modifié notre fonction pour qu'il s'arrête automatiquement dès que la solution converge. Dans l'exemple ci-dessous, la fonction K-means s'est automatiquement arrêté au bout de 7 itérations :

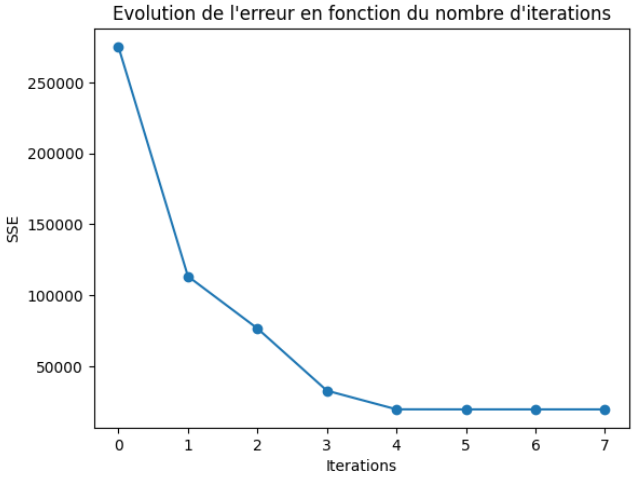


FIGURE 3.4 – Évolution de la SSE jusqu'à convergence

3.8 Visualisation des clusters en fonction des itérations

Lors du premier essai de notre algorithme K-means, nous observons après une convergence au bout de 8 itérations que nos centres sont correctement placées dans nos clusters.

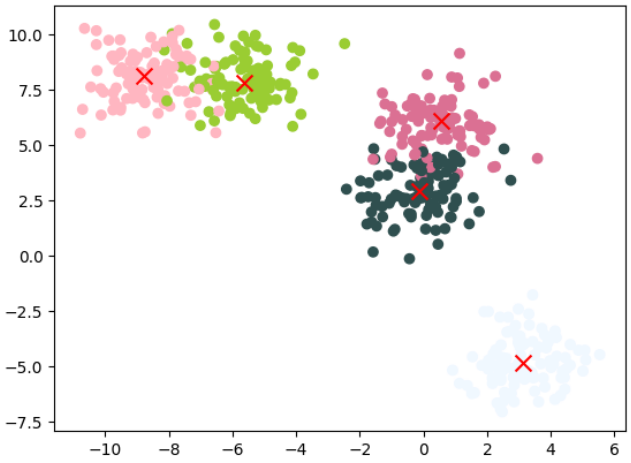


FIGURE 3.5 – Représentation des clusters et de leurs centres en 2 dimensions

Cependant, dans certain cas, nos centres ne sont pas correctement placés dans nos clusters. Ce problème est dû au placement initial de nos centre qui se fait de manière aléatoire parmi tous les points confondus. Pour résoudre ce problème, nous devons utiliser une version améliorée de l'algorithme K-means, l'algorithme K-means++.

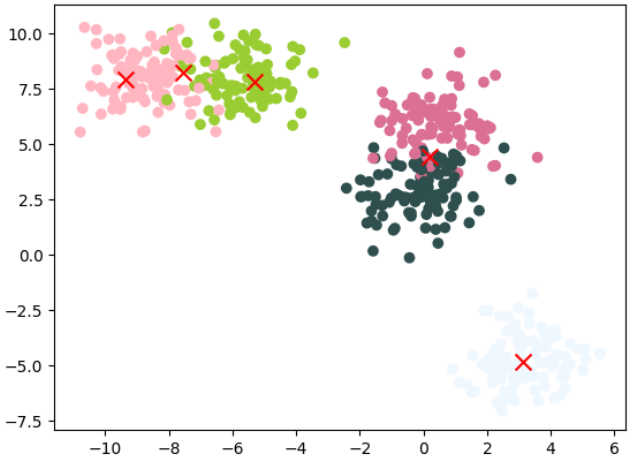


FIGURE 3.6 – Représentation des clusters et de leurs centres en 2 dimensions avec erreurs

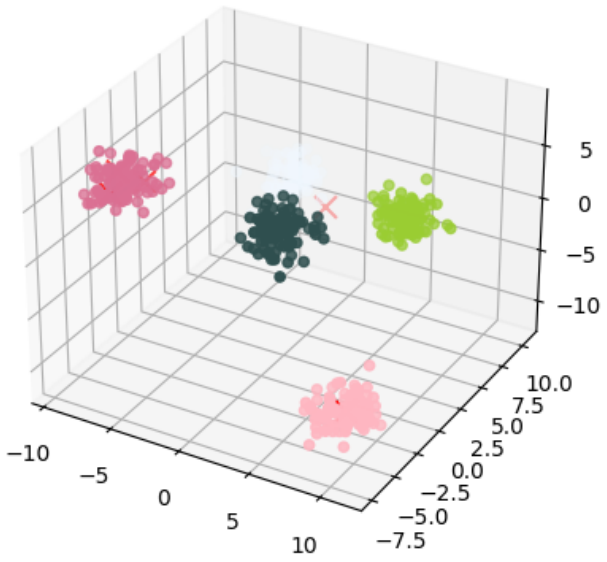


FIGURE 3.7 – Représentation des clusters et de leurs centres en 3 dimensions

Implémentation de k-means++

La version standard de l'algorithme k-means ne garantit pas la qualité de la solution obtenue. Néanmoins, une variante de la méthode d'échantillonnage initial existe, capable d'atteindre une approximation de la solution optimale avec un facteur de O(log k). Cette variante est nommée k-means++. Elle offre non seulement des garanties théoriques de qualité, mais semble également produire des résultats plus constants dans les applications pratiques.

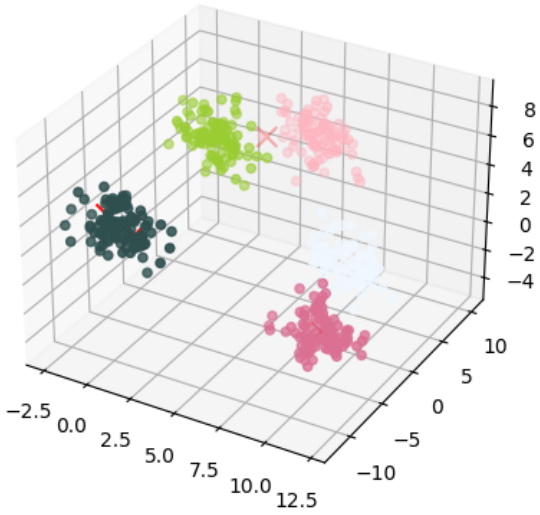
4.1 Initialisation des centres

La principale différence entre l'algorithme K-means et K-means++ est la fonction qui permet d'initialiser les centres des clusters. Dans l'algorithme K-means classique, les K centres initiaux sont choisit de manière aléatoire parmi l'ensemble des points. Cela pose des problèmes et peut enfermer notre algorithme dans une solution non optimale. L'algorithme K-means++ positionne les centres d'une manière différente. Les centres de cluster sont choisis en utilisant une méthode qui tend à répartir les centres plus uniformément sur l'espace de données. Le premier centre est choisi au hasard. Pour chaque centre suivant, la probabilité de choisir un point comme centre est proportionnelle à sa distance au carré par rapport au centre le plus proche déjà choisi. Cette approche réduit la probabilité de mauvaise convergence

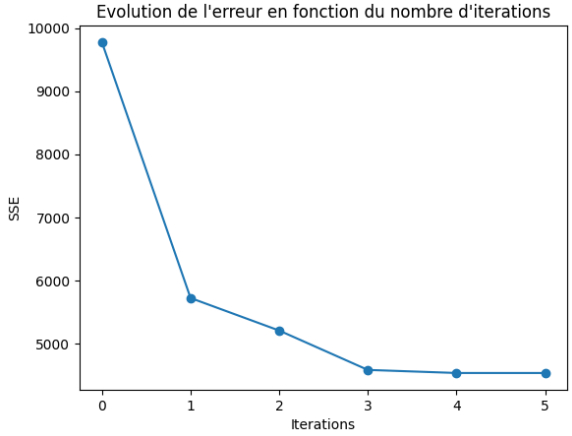
Listing 4.1 – Fonction pour initialiser le centre des différents clusters pour K-means++

```
def initialize_centers_plus(data, k):
    """
    Cette fonction initialise les centres des clusters en utilisant l'algorithme K-means ++

    Keyword arguments:
    data -- liste des points
    k -- nombre de clusters
    Return: liste des centres
    """

    centers = [] # Liste des centres
    centers.append(data[np.random.randint(0, len(data))]) # Selectionner aleatoirement le premier
        centre parmi les points de donnees

    for _ in range(1, k): # Pour chaque centre restant
        distances = [] # Liste des distances entre le point et les centres
        for point in data: # Pour chaque point
            min_distance = min(sse_distance(point, c) for c in centers) # Pour chaque point,
                calculer la distance au centre le plus proche deja choisi
            distances.append(min_distance) # Ajouter la distance a la liste des distances

        probabilities = np.array(distances) / sum(distances) # Probabilite proportionnelle a la
            distance au centre le plus proche deja choisi
        new_center_index = np.random.choice(len(data), p=probabilities) # Choisir le nouveau
            centre avec une probabilite proportionnelle a la distance au centre deja choisi
        centers.append(data[new_center_index]) # Ajouter le nouveau centre a la liste des centres

    return np.array(centers) # On retourne la liste des centres
```

4.2 Analyse des résultats

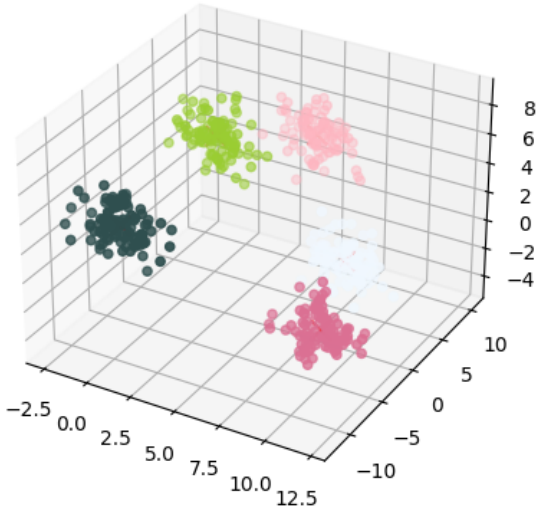


(a) Visualisation des 5 clusters pour un total de 500 points dans une dimension de 3

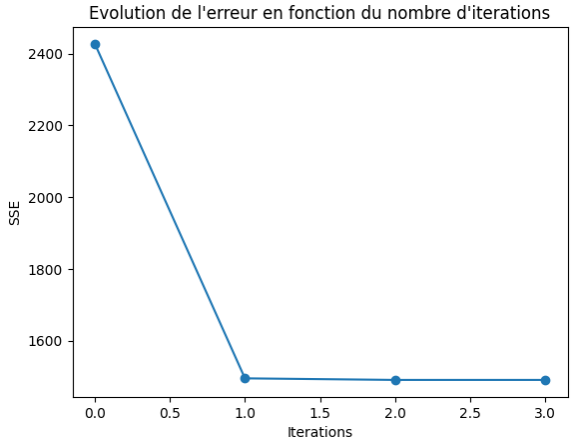


(b) Évolution de la SSE jusqu'à convergence

FIGURE 4.1 – Résultats de l'algorithme K-means



(a) Visualisation des 5 clusters pour un total de 500 points dans une dimension de 3



(b) Évolution de la SSE jusqu'à convergence

FIGURE 4.2 – Résultats de l'algorithme K-means++

Premièrement, sur plusieurs exemples, on remarque en moyenne un meilleur placement des centres des clusters grâce à l'algorithme K-means++. En comparant ensuite la SSE de l'algorithme K-means avec celui du K-means++, on remarque en moyenne une convergence nettement plus rapide pour l'algorithme K-means++, mais surtout, on remarque une SSE beaucoup plus faible autour des 1000 pour K-means++ contre 4000 à 5000 pour K-means. Plus la valeur de la SSE est faible, plus la solution obtenue par notre algorithme est pertinente.

Implémentation du mini-batch k-means

Dans l'algorithme K-means, le processus de mise à jour des centres de cluster est généralement réalisé sur l'ensemble complet de données à chaque itération. Cependant, lorsque le jeu de données est volumineux, cela peut entraîner des temps de calcul élevés. C'est là que le concept de mini-batch entre en jeu. Le mini-batch implique la division du jeu de données en petits lots (mini-batches) plutôt que de l'utiliser dans son intégralité. À chaque itération de l'algorithme K-means, l'un de ces mini-batches est sélectionné de manière aléatoire, et les centres de cluster sont mis à jour uniquement sur la base de ce sous-ensemble de données. L'utilisation de mini-batch présente plusieurs avantages. Tout d'abord, cela permet de réduire le temps de calcul en traitant seulement une fraction du jeu de données à la fois. De plus, cela peut également faciliter l'entraînement sur des ensembles de données trop volumineux pour tenir en mémoire.

5.1 Algorithme

1. Initialisation des centres : Les centres initiaux sont sélectionnés de manière aléatoire ou en utilisant une autre méthode.
2. Formation du mini-batch : Un sous-ensemble de données (mini-batch) est choisi à partir de l'ensemble de données complet.
3. Assignation des points au cluster le plus proche : Chaque point du mini-batch est assigné au cluster dont le centre est le plus proche.
4. Mise à jour des centres : Les centres des clusters sont recalculés en utilisant les points assignés à chaque cluster dans le mini-batch.

Répétition : Les étapes 2 à 4 sont répétées jusqu'à ce qu'un critère d'arrêt soit atteint (par exemple, convergence des centroids ou un nombre fixe d'itérations).

5.2 Implémentation

Dans notre cas, nous avons implémenté le mini-batch k-means en reprenant le code de la fonction kmean décrite plus haut.

Listing 5.1 – Implémentation du mini-batch k-means

```
def miniBatch(data, k, num_it, batch_size=32):
tolerance=1
centers_history = []
sse_history = []

# Initialisation des centres
centers = initialize_centers(data, k)

for i in range(num_it):
    centers_history.append(np.array(centers))

    # Selectionner un batch de donnees alatoirement
    batch_indices = np.random.choice(len(data), size=batch_size, replace=False)
    batch_data = data[batch_indices]

    # Attribution des clusters
    clusters = compute_clusters(centers, batch_data)
```

```
        # Calcul du SSE
        sse = sse_error(centers, batch_data)
        sse_history.append(sse)

        # Mise a jour des centres
        new_centers = []
        for cluster_id in range(k):

            # On recupere les points du batch qui appartiennent au cluster
            cluster_points = np.array([batch_data[j] for j in range(len(batch_data)) if clusters[j] ==
                cluster_id])
            if len(cluster_points) > 0:
                new_centers.append(np.mean(cluster_points, axis=0))
            else:
                new_centers.append(centers[cluster_id]) # Si un cluster est vide, on garde l'ancien
                    centre

        #  si la distance entre les nouveaux centres et les anciens centres est inferieure a la
            tolerance, on a atteint la convergence
        if np.linalg.norm(np.array(new_centers) - np.array(centers)) < tolerance:
            print(f"Convergence atteinte a l'iteration {i}")
            breaks

        ### La suite du code est identique a la fonction kmeans
        # ...
        ###

    return history
```

5.3 Hyperparamètres

5.3.1 Batch-size

Le batch size est le nombre d'échantillons qui seront utilisés pour mettre à jour les centres de clusters. Le choix du batch-size peut avoir un impact sur les aspects suivants :

- Vitesse de convergence : Un batch-size plus grand peut accélérer la convergence car il utilise un échantillon plus important pour mettre à jour les centres. Cependant, un batch-size trop grand peut également entraîner des mises à jour de centres moins fréquentes, ce qui peut ralentir la convergence ;
- Consommation de mémoire : Un batch-size plus grand nécessite plus de mémoire car il stocke un ensemble plus important de données en mémoire à chaque itération. Cela peut être une considération importante, en particulier pour de grands ensembles de données.
- Stabilité des résultats : Des mini-batchs plus petits peuvent conduire à des mises à jour de centres plus fréquentes, ce qui peut rendre l'algorithme plus stable. Cependant, cela peut également introduire une certaine variabilité dans les mises à jour, ce qui peut influencer la qualité des résultats.
- Sensibilité aux points extrêmes : Des mini-batchs plus petits peuvent être plus sensibles aux points extrêmes (outliers) présents dans le mini-batch, ce qui peut affecter la qualité des mises à jour de centres.
- Adaptabilité à la mémoire : Si la mémoire est limitée, un petit batch-size peut être nécessaire pour que l'algorithme s'exécute sur la configuration matérielle disponible.

5.3.2 Tolérance

La tolérance est la distance maximale entre les centres de clusters de deux itérations successives pour que l'algorithme s'arrête. Une tolérance plus petite peut conduire à une convergence plus précise, mais peut également augmenter le temps de calcul. Une tolérance plus grande peut conduire à une convergence plus rapide, mais peut également conduire à des résultats moins précis. Dans notre cas, nous avons choisi une tolérance de 1 étant donné les gros écarts entre les différentes valeurs de nos données.

5.4 Résultats

Nous avons testé notre implémentation du mini-batch k-means sur 10 000 points avec 5 clusters et 100 itérations maximum. Nous avons fixé le batch-size à 50 et la tolérance à 1. L'algorithme a convergé en 23 itérations et nous avons obtenu les résultats suivants :

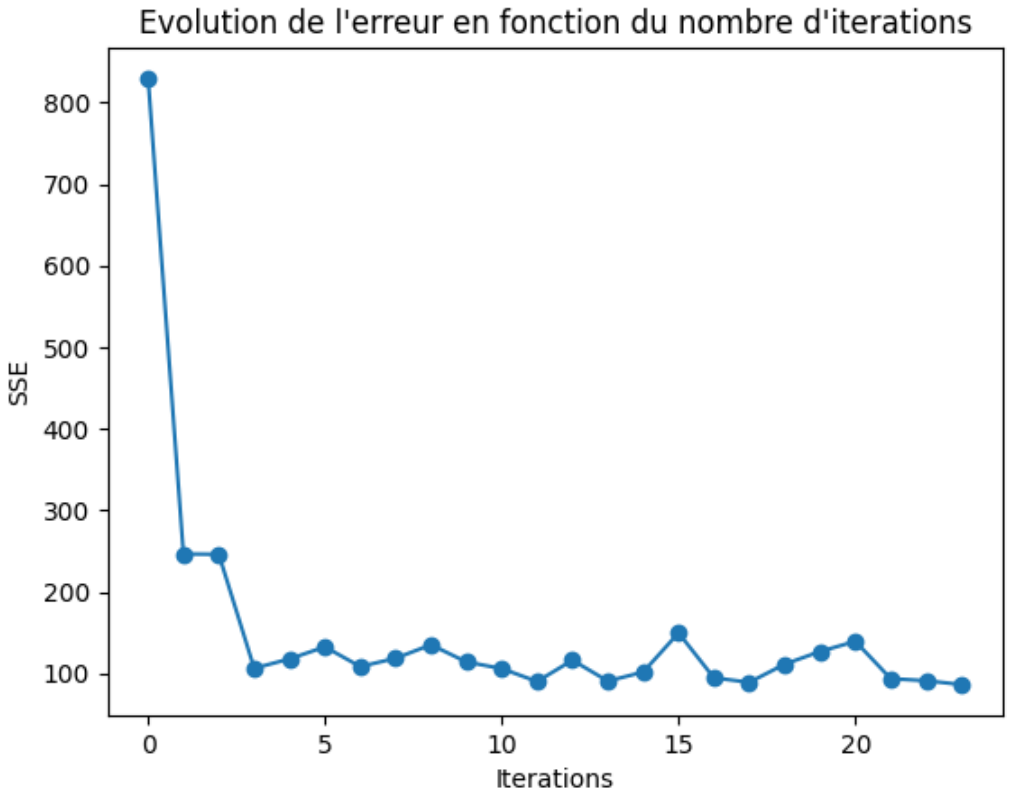


FIGURE 5.1 – Mini-batch k-means

On remarque que l'algorithme converge très rapidement malgré le grand nombre de points. On remarque également que la SSE diminue rapidement vers des valeurs plus basse que celles obtenues avec l'algorithme k-means classique (de l'ordre de 100 contre 10 000). Cela est dû au fait que l'algorithme k-means classique calcule la SSE sur l'ensemble des données à chaque itération, alors que le mini-batch k-means ne calcule la SSE que sur un sous-ensemble de données. En revanche on remarque que la SSE est plus instable que celle obtenue avec l'algorithme k-means classique et fluctue beaucoup plus avant d'atteindre la convergence.

Conclusion

Lors de ce projet, nous avons implémenté l'algorithme K-means et K-means++ en Python. Nous avons également implémenté une version améliorée de l'algorithme K-means, le mini-batch k-means. Nous avons ensuite testé nos implémentations sur des données générées aléatoirement. Nous avons pu observer que l'algorithme K-means converge rapidement vers une solution optimale, mais que cette solution peut être différente selon l'initialisation des centres. Nous avons également pu observer que l'algorithme K-means++ converge toujours vers une solution optimale. Nous avons également pu observer que le mini-batch k-means converge rapidement vers une solution optimale, mais que cette solution peut être différente selon le batch-size et l'initialisation des centres. Nous avons également pu observer que la SSE est plus instable que celle obtenue avec l'algorithme k-means classique et fluctue beaucoup plus avant d'atteindre la convergence. Ce projet nous a permis de mieux comprendre le fonctionnement de l'algorithme K-means et de ses variantes. Nous avons également pu observer les avantages et les inconvénients de ces algorithmes.