

REPORT 2

DATA MINGING

Clustering Analysis: K-Means and Iterative Algorithms

M1 IV
GROUP 1
BY: ABOURA Yasmine
LATIF Meriem

2023/2024

Introduction:

Our analysis aims to explore clustering techniques on the Mall_Customers dataset to uncover underlying patterns in the data. To achieve this goal, we have devised a two-part plan.

In the first part, we utilize K-Means clustering, a popular algorithm for partitioning data into clusters, by systematically varying the number of clusters (k) and observing the resulting cluster formations. This approach allows us to assess the impact of different k values on the clustering outcome.

In the second part, we delve into the challenge of determining the optimal number of clusters without prior knowledge, addressing this using an iterative algorithm. This algorithm iteratively adjusts the clustering parameters to find the optimal k value. Additionally, we aim to automate the process of determining the minimum distance between clusters, enhancing the efficiency of our clustering analysis.

By combining these two approaches, we aim to gain comprehensive insights into the structure of the Mall_Customers dataset and identify meaningful clusters that can inform targeted business strategies and marketing campaigns.

1. K-Means Clustering with Different Values of k:

Kmeans Clustering:

KMeans is a popular clustering algorithm used in machine learning for partitioning a dataset into K distinct, non-overlapping clusters. It aims to group similar data points together and discover underlying patterns in the data. The algorithm works iteratively to assign each data point to one of K clusters based on the features provided. It minimizes the sum of squared distances between data points and their respective cluster centroids, making the clusters as compact and distinct as possible. KMeans requires specifying the number of clusters (K) in advance and may converge to different solutions based on the initial placement of cluster centroids.

In this initial phase, our approach involves experimenting with different values of k to observe the variations in clustering outcomes. Our objective is to identify the optimal number of centroids that best suits the characteristics of the dataset.

We used Mall_Customers csv as a dataset that comprises customer information including CustomerID, Genre (gender), Age, Annual Income (in thousands of dollars), and Spending Score (ranging from 1 to 100).

Loading data

PART 1 : K-MEANS with 'k' as a parameter.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift, estimate_bandwidth
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import pairwise_distances
```

[]

Python

```
# Load the data
data = pd.read_csv("Mall_Customers.csv")
# Encoding 'Genre' as numeric
data['Genre'] = data['Genre'].map({'Male': 0, 'Female': 1})
# Selecting all features for clustering, except 'CustomerID'
X = data.drop(['CustomerID'], axis=1)
# Reducing data to two dimensions using PCA for visualization
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

[94]

✓ 0.0s

Python

We first Encode 'Genre' as numeric and then we reduce the date to two dimensions using PCA for visualization.

K=2

We implemented the K-means algorithm using the scikit-learn library to cluster the data.

```
k= 2
```

```
# Applying KMeans clustering
kmeans = KMeans(n_clusters=2, init='k-means++', max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(X)

# Visualising the clusters
plt.scatter(X_pca[y_kmeans == 0, 0], X_pca[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')
plt.scatter(X_pca[y_kmeans == 1, 0], X_pca[y_kmeans == 1, 1], s=100, c='blue', label='Cluster 2')

# Plotting the centroids
centroids_pca = pca.transform(kmeans.cluster_centers_)
plt.scatter(centroids_pca[:, 0], centroids_pca[:, 1], s=300, c='yellow', label='Centroids')

# Adding titles and labels
plt.title('Clusters of clients')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.legend()
plt.show()
```

[106] Python

We start our iterations with k=2.

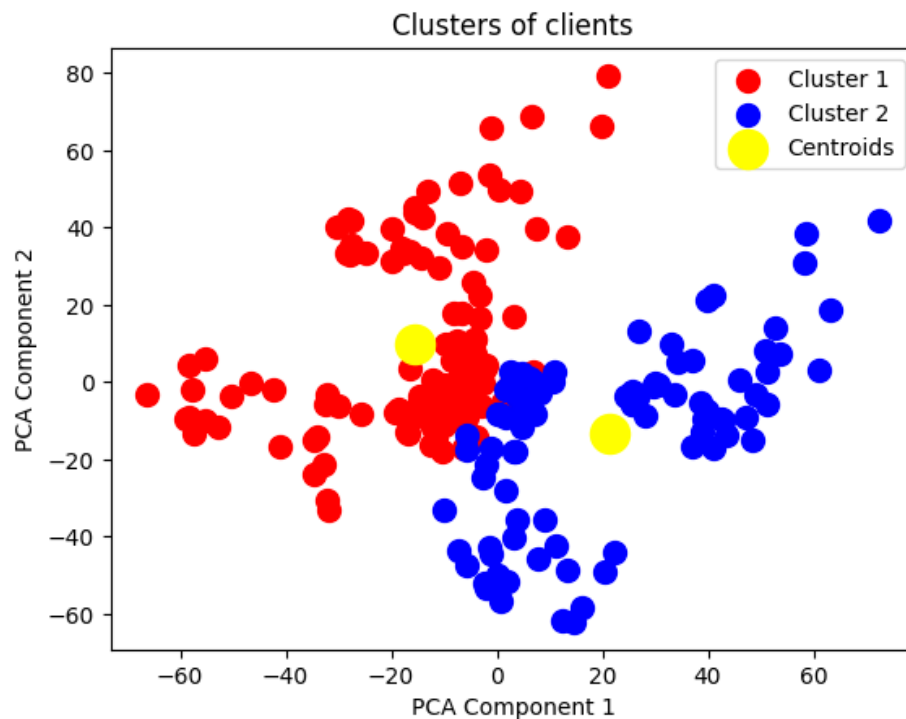


Figure 1 kmeans k=2

K=3

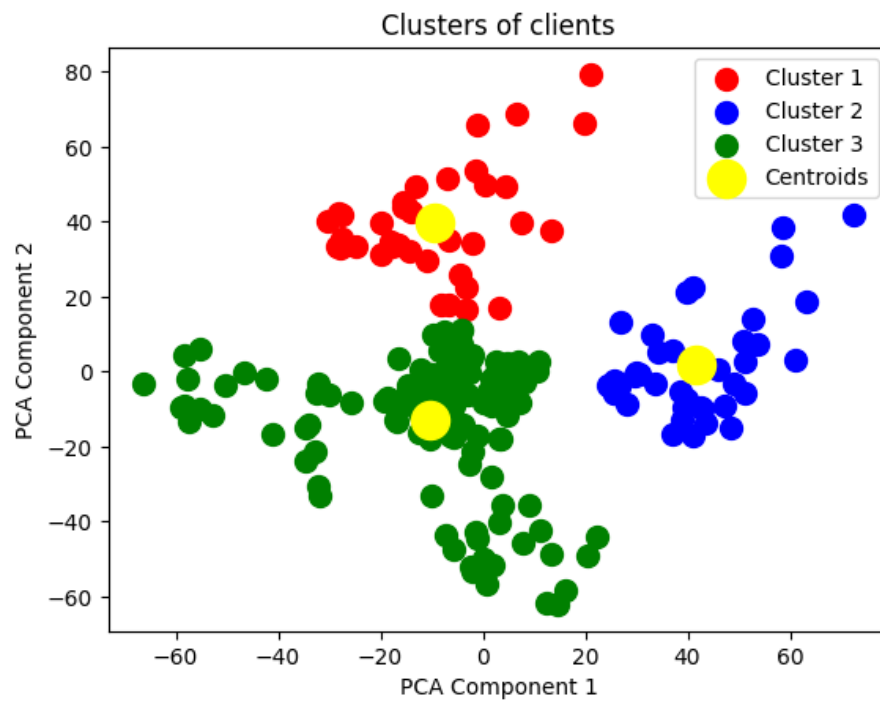


Figure 2 kmeans k=3

K=4

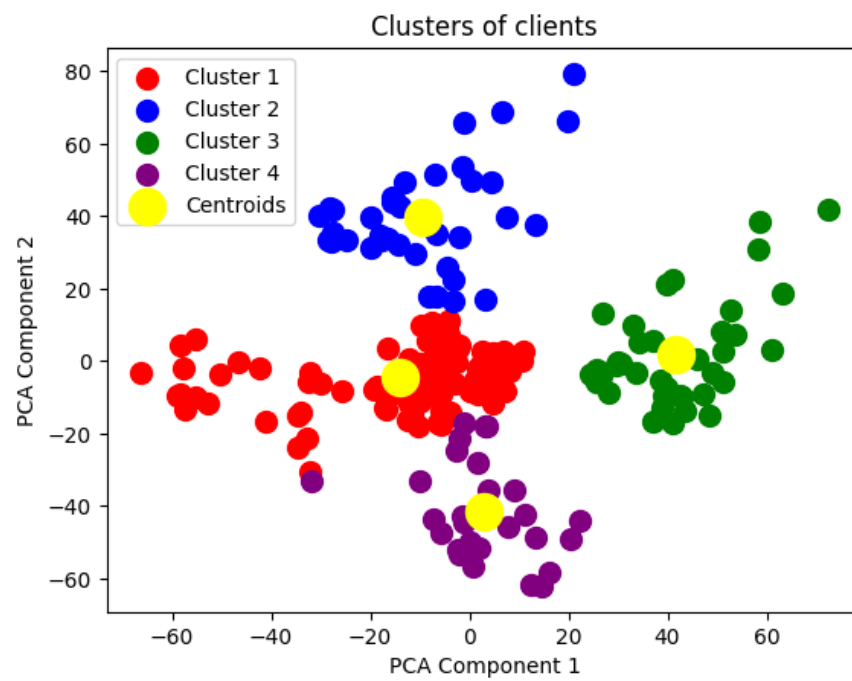


Figure 3 kmeans k=4

K=5

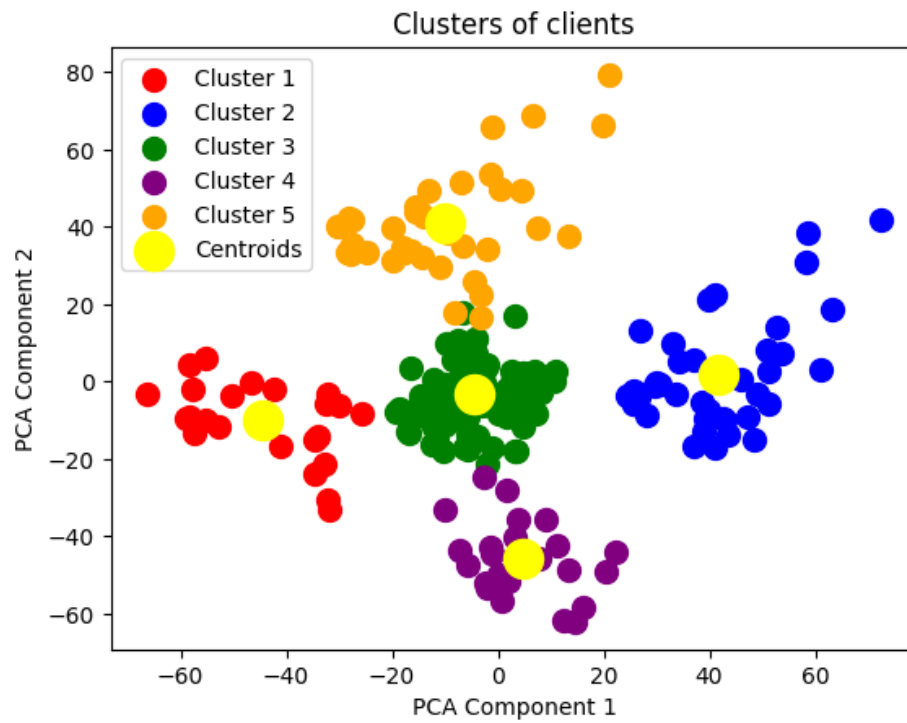


Figure 4 kmean k=5

K=6

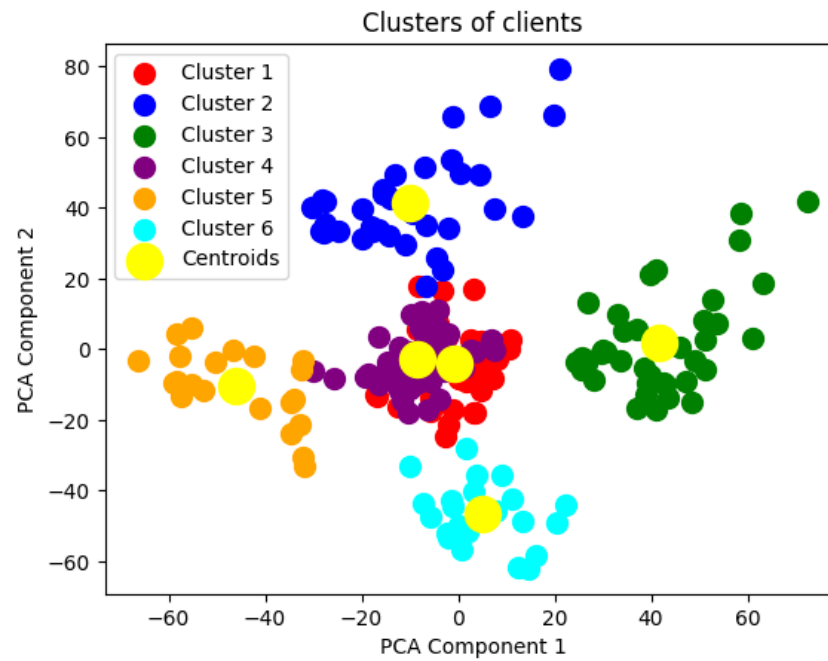


Figure 5 kemans k= 6

In our initial experiment with K-means clustering, we began with a relatively low value of $k=2$. This approach yielded a simple partitioning of the dataset into two distinct clusters. While this method provided a clear and straightforward division of the data, it also revealed that some data points were significantly distant from their assigned cluster centroids. This observation suggested that the data might contain more nuanced patterns or variations that were not adequately captured by a single cluster. To explore this further, we incrementally increased the number of clusters to $k=3$ and $k=4$. These iterations resulted in more clusters that encompassed a broader portion of the data. However, the improvement was not as pronounced as expected, indicating that while additional clusters helped to better represent the data, there was still room for improvement. Upon increasing the number of clusters to $k=5$, we observed a significant improvement in the clustering results. The data was almost perfectly partitioned into five distinct clusters, which appeared to capture the underlying structure of the dataset more accurately. When we increased k to 6, the results indicated no significant improvement. The addition of a new cluster did not provide any meaningful segmentation and appeared redundant, as it overlapped with existing clusters.

This outcome suggests that $k=5$ is a more appropriate number of clusters for this dataset, as it allows for a more detailed and nuanced representation of the data.

Silhouette analysis, a method used to find the optimal number of clusters (k) in a dataset for clustering algorithms like KMeans. It measures how similar an object is to its own cluster compared to other clusters. The silhouette score ranges from -1 to 1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters.


```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt

silhouette_scores = []
for k in range(2, 11): # Trying different values of k from 2 to 10
    kmeans = KMeans(n_clusters=k, random_state=0)
    cluster_labels = kmeans.fit_predict(X)
    silhouette_avg = silhouette_score(X, cluster_labels)
    silhouette_scores.append(silhouette_avg)

# Plotting the silhouette scores
plt.plot(range(2, 11), silhouette_scores, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette score')
plt.title('Silhouette Analysis')
plt.show()
```

[111] Python

To perform silhouette analysis, we calculated the silhouette score for different values of k (number of clusters) ranging from 2 to 10. For each value of k, we applied the KMeans algorithm to the data and calculated the average silhouette score across all samples.

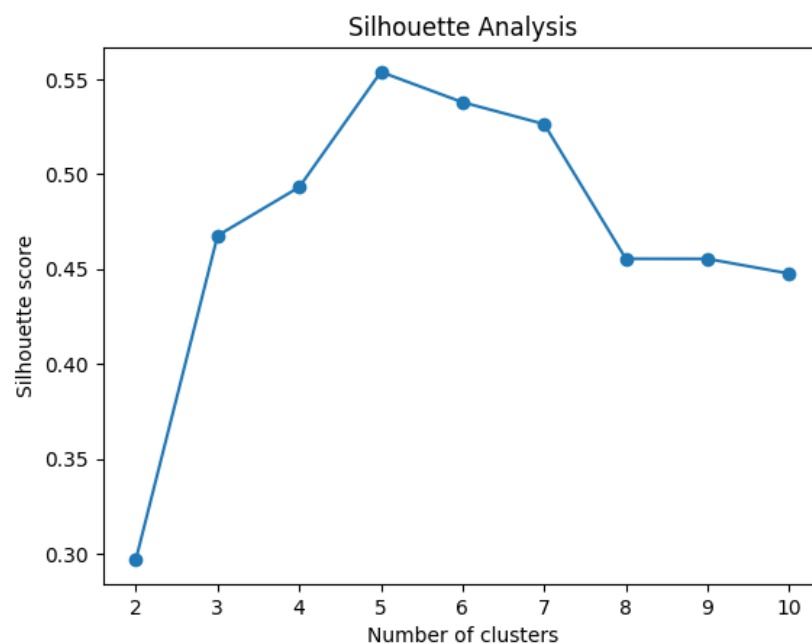


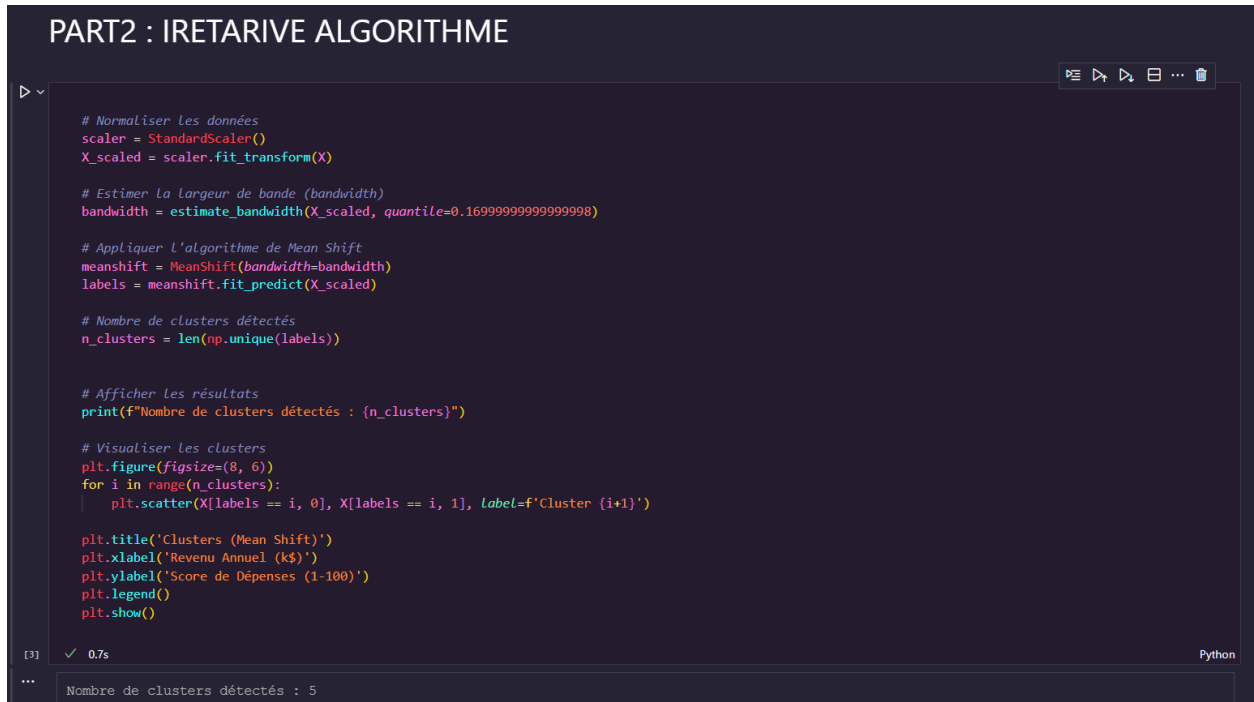
Figure 6 silhouette analysis

As we can see here the result shows that the graph is at peak at the number 5 which we chose to be the good match for our data.

2. Iterative algorithm

The goal of this project is to cluster customer data effectively. Initially, we used the K-means algorithm, manually testing different values of k to find the best number of clusters. However, this method is not optimal. Therefore, in the this second part of our analysis, we will employ an iterative algorithm to automatically determine the optimal number of clusters. This approach aims to enhance efficiency and accuracy by identifying the best k without manual trial and error

For this, we used the Mean Shift algorithm, which an iterative algorithm that automatically determines the optimal number of clusters by iteratively shifting data points towards the densest areas (modes).



```
PART2 : IRETARIVE ALGORITHMME

# Normaliser Les données
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Estimer la largeur de bande (bandwidth)
bandwidth = estimate_bandwidth(X_scaled, quantile=0.16999999999999998)

# Appliquer l'algorithme de Mean Shift
meanshift = MeanShift(bandwidth=bandwidth)
labels = meanshift.fit_predict(X_scaled)

# Nombre de clusters détectés
n_clusters = len(np.unique(labels))

# Afficher Les résultats
print(f"Nombre de clusters détectés : {n_clusters}")

# Visualiser les clusters
plt.figure(figsize=(8, 6))
for i in range(n_clusters):
    plt.scatter(X[labels == i, 0], X[labels == i, 1], label=f'Cluster {i+1}')

plt.title('Clusters (Mean Shift)')
plt.xlabel('Revenu Annuel (k$)')
plt.ylabel('Score de Dépenses (1-100)')
plt.legend()
plt.show()
```

[3] ✓ 0.7s Python

... Nombre de clusters détectés : 5

However, this algorithm also has a limitation: the bandwidth parameter, representing the distance, must be manually set, which may cause errors if not chosen appropriately.

Optimizing the Iterative Algorithm: Hill Climbing Approach

To optimize the Mean Shift algorithm, we employed a method similar to hill climbing. Hill climbing is an optimization algorithm that starts with an arbitrary solution and iteratively makes small changes to improve it. In our case, we began with an initial bandwidth value 0.2 and made incremental adjustments of 0.01.

The algorithm calculates the quality of the clusters by evaluating both intra-class and inter-class distances. Intra-class distance measures the average distance within clusters, while inter-class distance measures the distance between cluster centroids. The goal is to maximize inter-class distance while minimizing intra-class distance.

The algorithm adjusts the bandwidth, either increasing or decreasing it, to find the optimal cluster quality. This process continues until further adjustments no longer result in improved quality, thus ensuring the best clustering solution is achieved.

```
# Valeur de quantile initiale et pas de recherche
quantile = 0.2
step_size = 0.01

# Estimer la largeur de bande initiale (bandwidth)
bandwidth = estimate_bandwidth(X_scaled, quantile=quantile)

# Appliquer l'algorithme de Mean Shift avec la valeur de quantile initiale
meanshift = MeanShift(bandwidth=bandwidth)
labels = meanshift.fit_predict(X_scaled)

# Calculer Les centroïdes de chaque cluster
centroids = []
for label in np.unique(labels):
    centroid = np.mean(X[labels == label], axis=0)
    centroids.append(centroid)

# Calculer Les distances intra-classe initiales
intra_cluster_distances = []
for label, centroid in enumerate(centroids):
    distances = pairwise_distances(X[labels == label], [centroid])
    intra_cluster_distances.append(np.mean(distances))

# Calculer Les distances inter-classe initiales
inter_cluster_distances = []
```

```

for i in range(len(centroids)):
    for j in range(i+1, len(centroids)):
        distance = np.linalg.norm(centroids[i] - centroids[j])
        inter_cluster_distances.append(distance)

# Calculer la qualité initiale des clusters (inter/intra)
initial_quality = np.mean(inter_cluster_distances) /
np.mean(intra_cluster_distances)

# Répéter jusqu'à convergence
while True:
    # Essayer une valeur de quantile plus petite
    new_quantile = quantile - step_size
    if new_quantile >= 0:
        # Estimer la largeur de bande avec la nouvelle valeur de quantile
        new_bandwidth = estimate_bandwidth(X_scaled, quantile=new_quantile)

        # Appliquer l'algorithme de Mean Shift avec la nouvelle valeur de
quantile
        meanshift = MeanShift(bandwidth=new_bandwidth)
        labels = meanshift.fit_predict(X_scaled)

        # Calculer les centroïdes de chaque cluster
        centroids = []
        for label in np.unique(labels):
            centroid = np.mean(X[labels == label], axis=0)
            centroids.append(centroid)

        # Calculer les distances intra-classe
        intra_cluster_distances = []
        for label, centroid in enumerate(centroids):
            distances = pairwise_distances(X[labels == label], [centroid])
            intra_cluster_distances.append(np.mean(distances))

        # Calculer les distances inter-classe
        inter_cluster_distances = []
        for i in range(len(centroids)):
            for j in range(i+1, len(centroids)):
                distance = np.linalg.norm(centroids[i] - centroids[j])
                inter_cluster_distances.append(distance)

        # Calculer la qualité des clusters (inter/intra)
        new_quality = np.mean(inter_cluster_distances) /
np.mean(intra_cluster_distances)

```

```

    # Si la qualité s'est améliorée, mettre à jour la valeur de quantile et
continuer
    if new_quality > initial_quality:
        quantile = new_quantile
        bandwidth = new_bandwidth
        initial_quality = new_quality
    else:
        break

# Afficher la meilleure valeur de quantile
print(f"Meilleure valeur de quantile : {quantile}")

```

We obtain the following result:

```
Meilleure valeur de quantile : 0.16999999999999998
```

Figure 7 optimal dmin

Which give us the following result:



Figure 8 optimal itrative algorithm

Conclusion

Our analysis of the Mall_Customers dataset using clustering techniques revealed valuable insights into its underlying patterns. By employing a two-part strategy, we effectively determined the optimal number of clusters.

In the initial phase, K-Means clustering identified $k=5$ as the best choice, supported by silhouette analysis. Subsequently, an iterative approach using the Mean Shift algorithm, optimized through hill climbing, automated the selection of the optimal number of clusters. This combined strategy provided robust and accurate clustering results, facilitating informed business strategies.

Overall, our comprehensive approach, combining K-Means clustering and an optimized iterative algorithm provided robust and accurate clustering results. This dual-method strategy allowed us to uncover meaningful clusters within the dataset, potentially informing more targeted business strategies and marketing campaigns.